Chapter 2

Alternation as a Multiprocess System

We now examine how the alternation system can be implemented in two different ways: in Section 2.1 as a hardware protocol, and in Section 2.2 as a software system.

2.1 The Two-Phase Handshake

2.1.1 The Protocol

We now consider a system in which the call and return operations are performed by two separate hardware components, called the client and server, that communicate with one another by wires:

```
CLIENT  | arg +------>
        |        |    S
        |        |    E
        |        |    R
        |        V
        |        E
        |        R

        rtn |
```

The values of arg and rtn represent voltage levels on two sets of wires.

Let’s try to implement the alternation system in the most direct way, based on its internal specification AltSpec. Our first attempt would be to implement $x$ as the voltage level on a wire joining the components. However, that wouldn’t work because $x$ is changed by both components, and it’s hard to build a wire whose voltage can be changed by either of the two circuits it joins. Instead, we’ll
use two wires, one whose voltage is controlled by the caller and the other whose voltage is controlled by the returner. The values associated with these two wires are \( c \) and \( r \). The value of \( x \) will be the sum modulo 2 of two values:

\[
\begin{align*}
\text{Call} & \triangleq (c + r) \mod 2 = 0 \\
\text{Return} & \triangleq (c + r) \mod 2 = 1
\end{align*}
\]

Since we want \( x \) to equal 0 initially, we can let the initial condition on \( c \) and \( r \) be \( c = r = 0 \). To get the next-state relation, let’s define \( \text{AltCall} \) and \( \text{AltReturn} \) to be the actions obtained from \( \text{AltCall} \) and \( \text{AltReturn} \), respectively, by substituting \((c + r) \mod 2\) for \( x \):

\[
\begin{align*}
\text{AltCall} & \triangleq \land (c + r) \mod 2 = 0 \\
\land \text{Call} & \land ((c + r) \mod 2)' = 1 \\
\text{AltReturn} & \triangleq \land (c + r) \mod 2 = 1 \\
\land \text{Return} & \land ((c + r) \mod 2)' = 0
\end{align*}
\]

We haven’t defined what it means to prime a state function like \((c + r) \mod 2\). It means here, and it will continue to mean, the value of the state function in the new state. Thus, \(((c + r) \mod 2)'\) is the same as \((c' + r') \mod 2\), so we can rewrite this equation as:

\[(2.1) \quad \begin{align*}
\text{AltCall} & \triangleq \land (c + r) \mod 2 = 0 \\
\land \text{Call} & \land (c' + r') \mod 2 = 1
\end{align*} \quad \begin{align*}
\text{AltReturn} & \triangleq \land (c + r) \mod 2 = 1 \\
\land \text{Return} & \land (c' + r') \mod 2 = 0
\end{align*}
\]

If \( c \) and \( r \) are both 0 or 1, then the value of \((c + r) \mod 2\) is 0 when \( c = r \) and is 1 when \( c \neq r \). We can change the value of \((c + r) \mod 2\) by incrementing either \( c \) or \( r \) by one modulo 2, and leaving the other unchanged. We want the client to modify \( c \) and the server to modify \( r \). So, we let the first action change \( c \) and the second change \( r \). We make these changes to the actions \( \text{AltCall} \) and \( \text{AltReturn} \), calling the resulting actions \( \text{TPCall} \) and \( \text{TPReturn} \):

\[
\begin{align*}
\text{TPCall} & \triangleq \land c = r \\
\land \text{Call} & \land c' = (c + 1) \mod 2 \\
\land r' = r
\end{align*} \quad \begin{align*}
\text{TPReturn} & \triangleq \land c \neq r \\
\land \text{Return} & \land r' = (r + 1) \mod 2 \\
\land c' = c
\end{align*}
\]

This leads to the specification \( \text{TPSpec} \) in module \textit{TwoPhase} in Figure 2.1 on the next page. The module is all straightforward, except for the last two lines, which we’ll get to momentarily, in Section 2.1.2.

The system described by specification \( \text{TPSpec} \) is called the \textit{two-phased handshake protocol}. It is commonly used for communication between asynchronous hardware components.
2.1. THE TWO-PHASE HANDSHAKE

\textbf{MODULE TwoPhase}

\textbf{EXTENDS CallReturn, Naturals}
\textbf{VARIABLES} \(c, r\)

\(TPInit \triangleq \land CRInit\)
\(\land c = 0\)
\(\land r = 0\)

\(TPCall \triangleq \land c = r\)
\(\land \text{Call}\)
\(\land c' = (c + 1) \mod 2\)
\(\land r' = r\)

\(TPReturn \triangleq \land c \neq r\)
\(\land \text{Return}\)
\(\land r' = (r + 1) \mod 2\)
\(\land c' = c\)

\(TPNext \triangleq TPCall \lor TPReturn\)

\(TPSpec \triangleq TPInit \land \Box \lfloor TPNext \rfloor_{\langle \text{Face}, c, r \rangle}\)

\(TPInvariant \triangleq \land CRInvariant\)
\(\land c \in \{0, 1\}\)
\(\land r \in \{0, 1\}\)

\textbf{THEOREM} \(TPSpec \Rightarrow \Box TPInvariant\)

\(Bar \triangleq \text{INSTANCE} \ Alternation \ \text{WITH} \ x \leftarrow (c + r) \mod 2\)

\textbf{THEOREM} \(TPSpec \Rightarrow Bar!AltSpec\)

Figure 2.1: The specification of the two-phase handshake protocol.

2.1.2 Implementation of the Alternation System

We have constructed the two-phase handshake protocol from the alternation specification, where we implemented the variable \(x\) of \(AltSpec\) by \((c + r) \mod 2\). Formally, what we did was develop the specification \(TPSpec\) so it implements the specification \(AltSpec\) obtained by substituting \((c + r) \mod 2\) for \(x\) in the specification \(AltSpec\) of the alternation system. We call the substitution

\[x \leftarrow (c + r) \mod 2\]

a refinement mapping \(g\) and we say that \(TPSpec\) implements \(AltSpec\) under this refinement mapping. To express this in TLA\(^+\), we add to module \textit{TwoPhase}
the statement

\[ \text{Bar} \triangleq \text{instance Alternation with } x \leftarrow (c + r) \% 2 \]

This statement imports into module TwoPhase all the definitions (but not the declarations, like variable \(x\)) from module Alternation with two changes:

- The expression \((c + r) \% 2\) is substituted for the variable \(x\) in every definition.
- Every defined symbol \(F\) is renamed \(\text{Bar}!F\).

Thus, this statement defines \(\text{Bar}!\text{AltCall}\) and \(\text{Bar}!\text{AltReturn}\) to be the actions that we called \(\text{AltCall}\) and \(\text{AltReturn}\) in equation (2.1) on page 18.

Note that module Alternation has two additional declared variables besides the variable \(x\)—namely, the variables \(\text{arg}\) and \(\text{rtn}\), whose declarations it imported from module CallReturn. The instance statement is actually an abbreviation for

\[ \text{Bar} \triangleq \text{instance Alternation with } x \leftarrow (c + r) \% 2, \ \text{arg} \leftarrow \text{arg}, \ \text{rtn} \leftarrow \text{rtn} \]

where the variables \(\text{arg}\) and \(\text{rtn}\), imported by TwoPhase from CallReturn, are substituted for the variables of the same name of module Alternation. TLA\(^+\) allows us to eliminate from the with clause any substitution of a variable by an identifier of the same name.

Module TwoPhase ends by asserting the theorem \(\text{TPSpec} \Rightarrow \text{Bar}!\text{AltSpec}\), which states that specification TwoPhase implements specification AltSpec under the refinement mapping \(x \leftarrow (c + r) \% 2\). From our discussion in Section 1.5, we would expect to prove this by proving (i) \(\text{TPInit} \Rightarrow \text{Bar}!\text{AltInit}\) and (ii) \(\text{TPNext} \Rightarrow \text{Bar}!\text{AltNext}\). Let’s go through the exercise of doing this.

We first prove \(\text{TPInit} \Rightarrow \text{Bar}!\text{AltInit}\). Formula \(\text{Bar}!\text{AltInit}\) is the formula obtained from \(\text{CRInit} \land (x = 0)\) by substituting \((c + r) \% 2 = 0\) for \(x\). Since \(\text{CRInit}\) (defined in CallReturn) doesn’t mention \(x\), we have

\[ \text{Bar}!\text{AltInit} = \text{CRInit} \land ((c + r) \% 2 = 0) \]

It is obvious that this is implied by \(\text{TPInit}\).

We must next prove \(\text{TPNext} \Rightarrow \text{Bar}!\text{AltNext}\). For this, it suffices to prove \(\text{TPCall} \Rightarrow \text{Bar}!\text{AltCall}\) and \(\text{TPReturn} \Rightarrow \text{Bar}!\text{AltReturn}\). We first try proving \(\text{TPCall} \Rightarrow \text{Bar}!\text{AltCall}\). Since \(\text{Call}\) (defined in CallReturn) does not mention \(x\), formula \(\text{Bar}!\text{AltCall}\) is just the formula we called \(\text{AltCall}\) in equation (2.1) on page 18. Comparing the definitions of \(\text{TPCall}\) and \(\text{AltCall}\), to prove \(\text{TPCall} \Rightarrow \text{Bar}!\text{AltCall}\), we have to prove:

\[
\left( \land c = r \\
\land c' = (c + 1) \% 2 \\
\land r' = r \right) \Rightarrow \left( \land (c + r) \% 2 = 0 \\
\land (c' + r') \% 2 = 1 \right)
\]
This is easily seen to be true—if $c$ and $r$ are integers. However, formula $TPCall$ doesn’t say that $c$ and $r$ are integers. If they aren’t, we can’t deduce much from $TPCall$ about the value of $c'$. Formally, we can’t deduce $TPCall \Rightarrow Bar! AltCall$. What we can deduce is

$$TPCall \land TPInvariant \Rightarrow Bar! AltCall$$

where the invariant $TPInvariant$ asserts that $c$ and $r$ are elements of $\{0, 1\}$. We can use the invariant if we have already proved that it is an invariant—that is, if $TPSpec$ implies $\Box TPInvariant$.

We won’t belabor the details of this particular proof, which are quite simple. However, the general technique is important, so we will state very carefully what we have to do to prove that $TPSpec$ implies $Bar! AltSpec$. We must prove three things:

$$TPSpec \Rightarrow \Box TPInvariant$$

$TPInvariant$ is an invariant of the $TPSpec$.

$$TPInit \Rightarrow Bar! AltInit$$

The initial predicate of $TPSpec$ implies the initial predicate of $Bar! AltSpec$.

$$TPNext \land TPInvariant \Rightarrow Bar! AltNext$$

The next-state action of $TPSpec$ implies the next-state action of $Bar! AltSpec$.

For this simple example, there’s no need to prove anything. We can let TLC check it for us. We can use the following configuration file for module $TwoPhase$:

$$SPECIFICATION TPSpec INARIANT TPInvariant PROPERTY Bar! AltSpec$$

It tells TLC to check the invariant and that $TPSpec$ implies $Bar! AltSpec$.  

### 2.2 A Multiprocess Program

We now consider the alternation system as a program. We can consider $AltSpec$ to describe a program in which a single process alternately performs the operations represented by actions $Call$ and $Return$. Such a program might be written in an ordinary programming language as:

$$\text{while (true)} \{ \text{Call; Return} \}$$

The variable $x$ in formula $AltSpec$ represents the control state of the program: $x = 0$ means control is at statement $Call$, $x = 1$ means control is at statement $Return$.

---

1 The current version of TLC allows only a simple identifier as a `PROPERTY`. So, we must define a simple identifier to equal $Bar! AltSpec$. This is done in module $MTwoPhase$. 
We can also view the internal alternation specification as a two-process program, one process being the client and the other the server. Actions *AltCall* and *AltReturn* are the two processes' next-state actions, and *x* is a variable shared by the two processes. However, we can’t write this program using ordinary programming language constructs, so we’ll introduce some new ones. We aren’t going to use these constructs in real programs that can be run on a computer, so we’ll just use them informally. First, we need constructs to express parallel composition and waiting:

- We introduce the parallel composition operator || that separates statements performed in parallel.

- We introduce the `await` construct, where the statement `await(cond)` waits until the Boolean-valued expression `cond` is true.

- When describing concurrent algorithms, it’s important to know what operations are atomic—meaning that we can pretend that they are performed as a single, indivisible step. We therefore introduce the convention of putting double angle brackets ⟨⟩ around a statement whose execution must be atomic.

With this notation, we can write the program represented by the internal alternation specification *AltSpec* as:

```plaintext
while (true) { ⟨⟨ await(x==0); Call; x=1 ⟩⟩ }
|| while (true) { ⟨⟨ await(x==1); Return; x=0 ⟩⟩ }
```

The semicolons inside the ⟨⟩ brackets are somewhat misleading. When executing the statement

 ⟨⟨ await(x==0); Call; x=1 ⟩⟩

the three substatements are not executed sequentially. Instead, they are executed as a single step to produce the same result as if they had been executed sequentially. Since executing the statements `Call` and `x=1` in either order produces the same result, we could just as well have written this statement as

 ⟨⟨ await(x==0); x=1; Call ⟩⟩

In fact, we could even have written it as

 ⟨⟨ Call; await(x==0); x=1 ⟩⟩

Since the entire statement must be executed atomically, the `Call` cannot be performed unless the `await` statement can be executed immediately—that is, unless *x* equals 0. We don’t worry here about how such a statement is implemented to be atomic. We simply assume that it works.
2.2. A MULTIPROCESS PROGRAM

Unfortunately, real processors cannot actually execute such a complex statement atomically. Usually, they can execute atomically only one access (a read or a write) to a single variable. A processor will require at least three steps to execute the code inside the atomic brackets: one step to read \( x \) equal to 0, one step to execute the \textit{Call} (which sets the variable \textit{arg}), and one step to set \( x \) to 1. (When executing the \texttt{await} statement, the processor might have to read \( x \) multiple times; but we can ignore the reads that find \( x \) equal to 1 because they have no effect on the state.)

In what order should those three steps be performed? It seems pretty obvious that the \texttt{await} should be executed first, but what of the other two? Suppose the client process executed statement \( x = 1 \) before executing the \textit{Call} statement. Then server process could then execute the entire body of its \texttt{while} statement, starting with the \texttt{await(x=1)} statement, between the execution of the client’s \( x = 1 \) statement and its \textit{Call} statement. In this scenario, the \texttt{Return} statement is executed before the corresponding \textit{Call} statement, which is wrong. So, the only remaining possibility is to execute the client’s \textit{Call} before its \( x = 1 \) statement. Similarly, the server’s \texttt{Return} has to be executed before its \( x = 0 \) statement.

This gives us the following program, where we have introduced labels for the statements:

\[
\text{while (true) \{c1: \langle \texttt{await(x==0)} \rangle; c2: \langle \texttt{Call} \rangle; \ c3: \langle x=1 \rangle \}} \]
\[
\text{|| while (true) \{r1: \langle \texttt{await(x==1)} \rangle; r2: \langle \texttt{Return} \rangle; r3: \langle x=0 \rangle \}}
\]

We now write a TLA\(^+\) specification \textit{AltPgmSpec} that describes this program. We introduce a new variable \( pc \) to represent the program’s control state. Its value will be a record with \( c \) and \( r \) components, whose values are strings. For example, \( pc.c = \text{“c2”} \) and \( pc.r = \text{“r1”} \) means that control in the first process is at statement \( c2 \) and control in the second process is at \( r1 \). (TLA\(^+\) uses conventional notation for describing strings, where \text{“c2”} is the string consisting of the two characters \( c \) and \( 2 \).) TLA\(^+\) allows us to write a record whose \( c \) component is \text{“c2”} and whose \( r \) component is \text{“r1”} as

\[
\left[ c \mapsto \text{“c2”}, \ r \mapsto \text{“r1”} \right]
\]
A behavior that represents a possible execution of the program is:

\[
\begin{bmatrix}
\text{arg} = 0 \\
\text{rtn} = 0 \\
x = 0 \\
pc = [c \mapsto \text{"c1"}, \\
r \mapsto \text{"r1"}]
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\text{arg} = 0 \\
\text{rtn} = 0 \\
x = 0 \\
pc = [c \mapsto \text{"c2"}, \\
r \mapsto \text{"r1"}]
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\text{arg} = 3 \\
\text{rtn} = 0 \\
x = 0 \\
pc = [c \mapsto \text{"c3"}, \\
r \mapsto \text{"r1"}]
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\text{arg} = 3 \\
\text{rtn} = 0 \\
x = 1 \\
pc = [c \mapsto \text{"c1"}, \\
r \mapsto \text{"r2"}]
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\text{arg} = 3 \\
\text{rtn} = 9 \\
x = 1 \\
pc = [c \mapsto \text{"c1"}, \\
r \mapsto \text{"r3"}]
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\text{arg} = 3 \\
\text{rtn} = 9 \\
x = 0 \\
pc = [c \mapsto \text{"c2"}, \\
r \mapsto \text{"r1"}]
\end{bmatrix}
\rightarrow
\begin{bmatrix}
\text{arg} = 6 \\
\text{rtn} = 9 \\
x = 0 \\
pc = [c \mapsto \text{"c3"}, \\
r \mapsto \text{"r1"}]
\end{bmatrix}
\rightarrow \cdots
\]

The action $\text{AltPgmDoC}$, which describes steps of the first process, is the disjunction of three actions, each describing one of the process's statements. For example, its $\text{await}$ statement is enabled—meaning a step can be taken—when $x = 0$ and $pc.c = \text{"c1"}$. Executing that statement changes $pc.c$ to $\text{"c2"}$ and leaves $pc.r$ and the other variables unchanged. The disjunct of $\text{AltPgmDoC}$ representing the $\text{await}$ statement therefore has the form:

\[(2.2) \quad \land x = 0 \\
\land pc.c = \text{"c1"} \\
\land pc' = [c \mapsto \text{"c2"}, \ r \mapsto pc.r] \\
\land (x' = x) \land (arg' = arg) \land (rtn' = rtn)\]

Let's write the third conjunct more compactly. Observe that $x, \text{arg},$ and $\text{rtn}$ are unchanged if the triple $\langle x, \text{arg}, \text{rtn} \rangle$ is unchanged. So, the third conjunct of (2.2) can be written

\[(2.3) \quad \langle x, \text{arg}, \text{rtn} \rangle' = \langle x, \text{arg}, \text{rtn} \rangle\]

To express this a little bit more compactly, we let $\text{UNCHANGED} \ e$ mean $e' = e$ for any expression $e$, and we write (2.3) as

\[(2.4) \quad \text{UNCHANGED} \ \langle x, \text{arg}, \text{rtn} \rangle\]

(The $\text{UNCHANGED}$ operator makes a specification easier to read, so we use it even when it doesn’t save space.) Recall that we defined $i\text{Face}$ to equal $\langle \text{arg}, \text{rtn} \rangle$, so saying that $\langle \text{arg}, \text{rtn} \rangle$ is unchanged is equivalent to is equivalent to saying that $i\text{Face}$ is unchanged. Hence, we can write (2.4) even more compactly as

\[\text{UNCHANGED} \ \langle x, i\text{Face} \rangle\]
2.2. A MULTIPROCESS PROGRAM

In fact, we are interested in replacing \( \langle \text{arg}, \text{return} \rangle \) by \( \text{iface} \) not to save space, but to remove explicit mention of \( \text{arg} \) and \( \text{return} \) in our specification. That way, we can change the variables of module \( \text{CallReturn} \) without having to change any module that imports it, as long as \( \text{CallReturn} \) defines \( \text{iface} \) to be the tuple of variables declared by \( \text{CallReturn} \).

Introducing the \texttt{UNCHANGED} operator allows us to rewrite action (2.2) as:

\[
\begin{align*}
\text{(2.5)} & \quad \land x = 0 \\
& \quad \land \text{pc}.c = \text{“c1”} \\
& \quad \land \text{pc}’ = [c \mapsto \text{“c2”}, \ r \mapsto \text{pc}.r] \\
& \quad \text{UNCHANGED} \ \langle x, \text{iface} \rangle
\end{align*}
\]

Finally, observe that the second and third conjuncts of represents program execution flow. Analogous conjuncts will appear in each of the three disjuncts of action \( \text{AltPgmDoC} \). We therefore define an operator \( \text{CGoTo} \) so these conjuncts can be written \( \text{CGoTo}(\text{“c1”, “c2”}) \). The definition is

\[
\text{CGoTo}(\text{loc}1, \text{loc}2) \triangleq \land \text{pc}.c = \text{loc}1 \\
\land \text{pc}’ = [c \mapsto \text{loc}2, \ r \mapsto \text{pc}.r]
\]

and (2.5) becomes

\[
\begin{align*}
\land x = 0 \\
& \quad \land \text{CGoTo}(\text{“c1”, “c2”}) \\
& \quad \text{UNCHANGED} \ \langle x, \text{arg}, \text{return} \rangle
\end{align*}
\]

The rest of the specification, through the definition of \( \text{AltPgmSpec} \), is straightforward; it appears in Figure 2.2 on the following two pages.

Module \( \text{AlternationPgm} \) introduces two invariants. The first, \( \text{AltPgmTypeOK} \), is a simple type-correctness invariant. Its definition uses one additional piece of TLA+ notation:

\[
\{c : \{\text{“c1”, “c2”, “c3”}\}, \ r : \{\text{“r1”, “r2”, “r3”}\}\}
\]

This expression is the set of all records having a \( c \) component that is an element of the set \{\text{“c1”, “c2”, “c3”}\} and an \( r \) component that is an element of the set \{\text{“r1”, “r2”, “r3”}\}. (It is therefore a set containing 9 records.)

In addition to this simple type invariant, specification \( \text{AltPgmSpec} \) has a more interesting invariant, called \( \text{AltPgmInvariant} \). This invariant asserts that:

- If control in the first process is not at \( \text{c1} \), then \( x = 0 \).
- If control in the second process is not at \( \text{r1} \), then \( x = 1 \).

An immediate consequence is that control cannot both be not at \( \text{c1} \) and not at \( \text{r1} \). In other words, control must always be either at \( \text{c1} \) or \( \text{r1} \) (or at both). As we will see later, this is an important property of the specification.
CHAPTER 2. ALTERNATION AS A MULTIPROCESS SYSTEM

---

**MODULE AlternationPgm**

EXTENDS CallReturn

VARIABLES $x$, $pc$

$$
\begin{align*}
\text{AltPgmInit} & \triangleq \land CRI\text{Init} \\
& \quad \land x = 0 \\
& \quad \land pc = [c \mapsto "c1", r \mapsto "r1"]
\end{align*}
$$

$$
\begin{align*}
\text{CGoTo}(loc1, loc2) & \triangleq \land pc.c = loc1 \\
& \quad \land pc' = [c \mapsto loc2, r \mapsto pc.r]
\end{align*}
$$

$$
\begin{align*}
\text{RGoTo}(loc1, loc2) & \triangleq \land pc.r = loc1 \\
& \quad \land pc' = [c \mapsto pc.c, r \mapsto loc2]
\end{align*}
$$

$$
\begin{align*}
\text{AltPgmDoC} & \triangleq \lor \land \text{CGoTo}("c1", "c2") \\
& \quad \land x = 0 \\
& \quad \land \text{UNCHANGED} \langle x, i\text{Face} \rangle \\
& \lor \land \text{CGoTo}("c2", "c3") \\
& \quad \land \text{Call} \\
& \quad \land \text{UNCHANGED} \langle x \rangle \\
& \lor \land \text{CGoTo}("c3", "c1") \\
& \quad \land x' = 1 \\
& \quad \land \text{UNCHANGED} \langle i\text{Face} \rangle 
\end{align*}
$$

$$
\begin{align*}
\text{AltPgmDoR} & \triangleq \lor \land \text{RGoTo}("r1", "r2") \\
& \quad \land x = 1 \\
& \quad \land \text{UNCHANGED} \langle x, i\text{Face} \rangle \\
& \lor \land \text{RGoTo}("r2", "r3") \\
& \quad \land \text{Return} \\
& \quad \land \text{UNCHANGED} \langle x \rangle \\
& \lor \land \text{RGoTo}("r3", "r1") \\
& \quad \land x' = 0 \\
& \quad \land \text{UNCHANGED} \langle i\text{Face} \rangle 
\end{align*}
$$

$$
\text{AltPgmNext} \triangleq \text{AltPgmDoC} \lor \text{AltPgmDoR}
$$

$$
\text{AltPgmSpec} \triangleq \text{AltPgmInit} \land \Box[\text{AltPgmNext}]_{(x, pc, i\text{Face})}
$$

$$
\text{AltPgmTypeOK} \triangleq \land CRI\text{Invariant} \\
& \quad \land x \in \{0, 1\} \\
& \quad \land pc \in \{c : \{"c1", "c2", "c3"\}, r : \{"r1", "r2", "r3"\}\}
$$

---

Figure 2.2a: A specification of the alternation program (beginning).
2.3. IMPLEMENTATION RE-EXAMINED

\[ \text{AltPgmInvariant} \triangleq \land (pc.c \neq "c1") \Rightarrow (x = 0) \]
\[ \land (pc.r \neq "r1") \Rightarrow (x = 1) \]

**THEOREM** \( \text{AltPgmSpec} \rightarrow \Box (\text{AltPgmTypeOK} \land \text{AltPgmInvariant}) \)

Figure 2.2b: A specification of the alternation program (end).

2.3 Implementation Re-Examined

2.3.1 Invariance Under Stuttering

Suppose we asked someone to build us a discrete oscillator and she produced the alternation program described by specification \( \text{AltPgmSpec} \). The oscillator specification describes how \( x \) changes; it doesn’t say anything about \( \text{arg} \), \( \text{rtn} \), and \( pc \). So, to decide whether the alternation program implements the oscillator, we should just look at the changes to \( x \). If we ignore the other variables, the value of \( x \) alternates between 0 and 1, just as the discrete oscillator’s specification says it should. So, it is reasonable to say that the alternation program should implement the discrete oscillator—and hence, that specification \( \text{AltPgmSpec} \) should imply the specification \( \text{OscSpec} \) of the discrete oscillator.

Suppose we take the behavior of the alternation program on page 24 and simply remove the values of the other variables. We then get:

\[
\begin{align*}
(x = 0) & \rightarrow [x = 0] \rightarrow [x = 0] \rightarrow [x = 1] \rightarrow [x = 1] \rightarrow [x = 1] \rightarrow \\
& \rightarrow [x = 0] \rightarrow [x = 0] \rightarrow [x = 0] \rightarrow \ldots
\end{align*}
\]

This differs from the behavior (1.1) on page 2 because it contains extra steps in which \( x \) doesn’t change. We call such steps *stuttering* steps. Since our alternation program is to implement a discrete oscillator, formula \( \text{OscSpec} \) needs to allow behaviors, like (2.6), that contain stuttering steps.

The need to allow stuttering steps may seem somewhat strange, so let’s try another explanation of why they’re needed. Recall that, in Section 1.5 (page 12), we said that a state is an assignment of values to all possible variables, and that it represents a state of the entire universe. If that’s to be the case, then (1.1) can’t be the only possible behavior of the oscillator. If it were, then everything else in the universe could change only when the state of the oscillator changed. For example, it would be impossible to build a faster oscillator. This is obviously absurd. Steps that don’t change \( x \) represent steps in which something else in the universe may be changing.

The system can’t observe stuttering steps because all of the variables of the system remain unchanged. Hence, adding or removing a stuttering step from a behavior should not change whether or not the behavior satisfies the specification. A specification having that property is said to be *invariant under*
stuttering. A specification that is not invariant under stuttering would make no sense, since it would distinguish between two behaviors that should be indistinguishable. The syntax of TLA+ makes it possible to write only specifications that are invariant under stuttering.

It is the mysterious $[\cdot]_x$ that makes the specification of the discrete oscillator invariant under stuttering. Recall that we defined the specification to be

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

For any action $A$ and state function $v$, we define:

$$[A]_v \triangleq A \lor (\text{UNCHANGED } v)$$

Thus, $[x' = (x + 1) \% 2]_x$ is the action

$$(x' = (x + 1) \% 2) \lor (x' = x)$$

This action allows steps that leave $x$ unchanged as well as steps that increment $x$ by 1 modulo 2. Putting a $\Box$ in front of the action asserts that it holds for every step in the behavior. Hence, formula OscSpec allows behaviors like (2.6) that have stuttering steps.

All of our (internal) specifications have the form $\text{Init} \land \Box [\text{Next}]_v$, where $v$ is the tuple of all the specification’s variables.\(^2\) It is true of a behavior iff $\text{Init}$ is true of the first state, and every step either satisfies $\text{Next}$ or leaves all the specification’s variables unchanged. Thus, all of our specifications allow stuttering steps.

Our specification OscSpec has one problem: it allows too many stuttering steps. It permits behaviors in which $x$ changes only a finite number of times, and then remains forever unchanged. A behavior that ends in this way with an infinite sequence of stuttering steps represents an execution in which the system halts. Halting is sometimes called termination and sometimes called deadlock, depending on whether or not it’s considered a good thing. If we don’t want the discrete oscillator to halt, then we’ll have to conjoin to its specification an additional condition asserting that it never halts. We’ll see how to do that later.

So far, we have said that a behavior is a sequence of states, but haven’t said whether it can be a finite sequence. We now make precise that it can’t; a behavior is an infinite sequence of states. An execution in which the system halts is represented by a behavior that ends with an infinite sequence of stuttering steps. Just because the system halts doesn’t mean the rest of the universe stops; other variables can keep changing.

\(^2\)More generally, as in AllFymSpec, leaving $v$ unchanged implies that all the variables of the specification are left unchanged.
2.4 The Grain of Atomicity

2.4.1 Re-examining the two Representations

Let's take another look at the relation between the specifications \textit{AltSpec} and \textit{AltPgmSpec}. We can consider both to be representations of the two-process alternation program. In \textit{AltSpec}, we represented the execution of the client process's three statements

\[
c1: \text{wait}(x=0); \quad c2: \text{Call}; \quad c3: x = 1
\]

as a single step. In the more realistic specification \textit{AltPgmSpec} of this program, we represented the execution as three separate steps. Similarly, execution of the three statements in the server process's \texttt{while} loop were represented by one step in \textit{AltSpec} and by three in \textit{AltPgmSpec}. We say that \textit{AltPgmSpec} is a

\textbf{finer-grained} specification than \textit{AltSpec}.

Let's give the names \textit{AltPgmDoC1}, \textit{AltPgmDoC2}, \ldots, \textit{AltPgmDoR2}, \textit{AltPgmDoR3} to the disjuncts of actions \textit{AltPgmDoC} and \textit{AltPgmDoR}. Labeling the arrows in a behavior of the alternation program with the action that describes the step, we get:

\[
\begin{align*}
&\text{\textit{AltPgmDoC1}} \quad \text{\textit{AltPgmDoC2}} \quad \text{\textit{AltPgmDoC3}} \quad \text{\textit{AltPgmDoR1}} \quad \text{\textit{AltPgmDoR2}} \quad \ldots
\end{align*}
\]

For any such behavior, the ordinary alternation system has the corresponding behavior

\[
\begin{align*}
&\text{\textit{AltCall}} \quad \text{\textit{AltReturn}} \quad \text{\textit{AltCall}} \quad \text{\textit{AltReturn}} \quad \ldots
\end{align*}
\]

We can consider these two behaviors to be two different ways of viewing an execution of the two-process program. If we consider only the variables \texttt{arg} and \texttt{rtn}, then these behaviors are the same except that the first has extra stuttering steps.

These two views are representations of the two-process program at two different grains of atomicity. Which one we use depends on what we want it for. We would prefer to use the coarser-grained representation described by the specification \textit{AltSpec}, since it is simpler. If we are interested only in the behavior of the \texttt{Call} and \texttt{Return} actions, which affect only the variables \texttt{arg} and \texttt{rtn}, then specification \textit{AltSpec} is good enough. However, \textit{AltSpec} is too coarse-grained to help us design a program that correctly implements alternation. A processor cannot perform the \texttt{Call} or \texttt{Return} action and change \texttt{x} in a single step. Studying \textit{AltSpec} cannot show us that the program must perform the \texttt{Call} or \texttt{Return} operation before changing \texttt{x}. For that, we need a finer-grained representation such as specification \textit{AltPgmSpec}. 

2.4.2 Finer-Grained Call and Return

So far, we have been representing the Call and Return operations as atomic, meaning that their execution is described by a single step of a behavior. Is this representation realistic? We can’t answer that question without knowing how these operations are implemented. Each of these operations are described in the CallReturn module as simply setting a single variable. In a program, they might be implemented as complicated, multi-step operations. In a hardware system, they might be implemented by setting the voltage on a set of wires. Since the voltages on different wires do not necessarily change simultaneously, we might want to represent an execution of one of these operations by multiple steps.

Suppose we want to modify our specification AltPgmSpec to obtain a finer-grained specification FGAltPgmSpec in which Call and Return are not atomic operations, so their execution is represented by multiple steps. The Call operation could set $arg$ to several intermediate values before setting it to its final value; and Return could similarly change $rtn$ multiple times.

The resulting finer-grained specification would no longer implement the specification AltSpec of alternation. A specification in which $arg$ is changed multiple times when executing Call does not implement a specification which asserts that $arg$ is changed just once. The extra steps allowed by the finer-grained specification are not stuttering steps; they change the visible variables $arg$ and/or $rtn$.

So, let’s begin by writing a new, finer-grained specification of alternation that allows the Call and Return operations to change $arg$ and $rtn$ multiple times. There are a number of different ways to do this. A simple way is to define an action FGCall and a state predicate FGCallDone such that a Call operation is represented by a sequence of FGCall steps, ending in a state in which FGCallDone is true. We replace the action AltCall of the alternation specification (defined in module Alternation of Figure 1.1 on page 8) with the following action

$$\text{FGAltCall} \triangleq \begin{array}{c}
\wedge x = 0 \\
\wedge \text{FGCall} \\
\wedge x' = \text{IF \ FGCallDone' THEN \ 1 \ ELSE \ x}
\end{array}$$

An FGAltCall step changes $arg$ as specified by the FGCall action; and it sets $x$ to 1 iff the FGCall actions makes FGCallDone true, leaving the value of $x$ unchanged if the new value of FGCallDone is FALSE.

We do the same thing for the Return operation, defining an action FGReturn and a state predicate FGReturnDone. We replace action AltReturn by an action FGAltReturn that changes $x$ only when FGReturnDone becomes true. We then get a new internal specification IFGAltSpec of alternation with nonatomic Call and Return operations. We then hide the internal variable $x$ to obtain the finer-grained specification FGAltSpec.
In the same way, we can modify the alternation program’s specification $\text{AltPgmSpec}$ (defined in module $\text{AlternationPgm}$) to allow nonatomic $\text{Call}$ and $\text{Return}$ operations. We need to change the two disjuncts of the next-state action that contain the $\text{Call}$ and $\text{Return}$ actions. For example, we replace the third disjunct of action $\text{AltPgmDoC}$ by one that performs an $\text{FGCall}$ step instead of a $\text{Call}$ step, and that changes $pc.c$ only if $\text{FGCallDone}$ is true in the new state. The new disjunct is:

$$\forall \langle \cdot \rangle. pc.c = "c2"$$
$$\land pc' = \begin{cases} [c \mapsto \text{if } \text{FGCallDone} \text{ then } "c3" \text{ else } "c2", \\ r \mapsto pc.r] \end{cases}$$
$$\land \text{FGCall}$$
$$\land \text{UNCHANGED } \langle x \rangle$$

The other two conjuncts of $\text{AltPgmDoC}$ remain the same. We similarly define the action $\text{FGReturn}$ and the state predicate $\text{FGReturnDone}$.

We still have to define the actions $\text{FGCall}$ and $\text{FGReturn}$ and the state predicates $\text{FGCallDone}$ and $\text{FGReturnDone}$. As in module $\text{CallReturn}$, we let $arg$ be a four-bit number and $rtn$ be an eight-bit number, and we let the function computed by the call be the square. We let the state functions $\text{FGCallDone}$ and $\text{FGReturnDone}$ be new variables.

An execution of the $\text{Return}$ operation is represented by a sequence of $\text{FGReturn}$ steps, ending in one that sets $\text{FGReturnDone}$ true. The $\text{Return}$ operation ends with $rtn$ equal to $arg^2$, so an $\text{FGReturn}$ step that sets $\text{FGReturnDone}$ true must set $rtn$ to $arg^2$. The other $\text{FGReturn}$ steps, which set $\text{FGReturnDone}$ false, can set $rtn$ to any eight-bit number (including $arg^2$). A straightforward definition of $\text{FGReturn}$ appears in module $\text{FGCallReturn}$ in Figure 2.3 on the next page. Here’s an equivalent way of writing the definition:

$$\text{FGReturn} \triangleq \begin{cases} \land rtn' \in \text{Input} \\
\land \text{FGReturnDone}' \in \{\text{TRUE, FALSE}\} \\
\land \text{FGReturnDone}' \Rightarrow (rtn' = arg^2) \\
\land \text{UNCHANGED } \langle \text{arg, FGCallDone} \rangle \end{cases}$$

An execution of the $\text{Call}$ operation is represented by a sequence of $\text{FGReturn}$ steps, ending with one that sets $\text{FGCallDone}$ true. The operation can end with $arg$ equal to any four-bit number. So an $\text{FGReturn}$ step simply sets $arg$ to any four-bit number and $\text{FGCallDone}$ to any Boolean.

Module $\text{FGCallReturn}$ contains all the relevant definitions, including the initial predicate $\text{FGCRInit}$, the tuple $iFace$ of interface variables, and the type invariant $\text{FGCRIInvariant}$. Using it, we can write modules to define the fine-grained versions of the alternation specification and the alternation program specification. Writing those modules will be left as an exercise.
EXTENDS Naturals

\begin{align*}
Input & \triangleq 0 \ldots 15 \\
Output & \triangleq 0 \ldots 255 \\
Rtn Val(n) & \triangleq n^2
\end{align*}

VARIABLES arg, rtn, FGCallDone, FGR return Done

\begin{align*}
FGCRInit & \triangleq \land arg = 0 \\
& \land \land arg = 0 \\
& \land \land FGCallDone = \text{TRUE} \\
& \land \land FGR return Done = \text{TRUE}
\end{align*}

\begin{align*}
FGCall & \triangleq \land arg' \in Input \\
& \land \land FGR return Done' = \text{TRUE} \\
& \land \land \text{UNCHANGED } (rtn', \text{FGR return Done'})
\end{align*}

\begin{align*}
FGReturn & \triangleq \land \lor \land \land arg' \in Output \\
& \land \land FGR return Done' = \text{TRUE} \\
& \land \land \land arg' = \text{arg}^2 \\
& \land \land \land FGR return Done' = \text{TRUE} \\
& \land \land \text{UNCHANGED } (arg, \text{FGR CallDone})
\end{align*}

iFace \triangleq \langle \text{arg, rtn, FGR CallDone, FGR return Done} \rangle

\begin{align*}
FGCR Invariant & \triangleq \land arg \in Input \\
& \land \land \land \rho \in Output \\
& \land \land \land \land FGR CallDone \in \{\text{TRUE, FALSE}\} \\
& \land \land \land \land \text{FGR return Done} \in \{\text{TRUE, FALSE}\}
\end{align*}

Figure 2.3: Definitions of the fine-grained call and return operations.

2.4.3 What Grain of Atomicity?

The grain of atomicity we use when representing a system depends on our goal. Let’s suppose we want to check whether the alternation program is really correct. We’ve seen that we need a finer-grained representation than the one provided by specification ActSpec of module Alternation, in which the entire operation of reading \( x \), performing the Call or Return operation, and changing \( x \) is done atomically. Can we use the specification ActPgmSpec, in which the operation takes three steps, or do we need the finer-grained specification sketched above, in which executing Call and Return takes multiple steps? Might we need an even finer-grained specification in which reading or writing \( x \) is not atomic?
2.4. THE GRAIN OF ATOMICITY

To answer this question, let’s compare the alternation program specification \( \text{AltPgmSpec} \) with the finer-grained specification \( \text{FGAltPgmSpec} \). Let’s name the disjunctions of the alternation program’s next-state action by the label of the statement it describes; for example, letting \( c2 \) be the second disjunct of action \( \text{AltPgmDoC} \). A behavior satisfying the alternation program’s specification looks like this:

\[
\begin{align*}
(2.7) \quad & s_1 \xrightarrow{c_1} s_2 \xrightarrow{c_2} s_3 \xrightarrow{c_3} s_4 \xrightarrow{r_1} s_5 \xrightarrow{r_2} s_6 \xrightarrow{r_3} s_7 \xrightarrow{c_1} s_8 \xrightarrow{c_2} s_9 \xrightarrow{r_1} \ldots
\end{align*}
\]

for states \( s_i \). The finer-grained specification \( \text{FGAltPgmSpec} \) contains additional variables not present in \( \text{AltPgmSpec} \)—namely, \( \text{FGCallDone} \), and \( \text{FGRetrunDone} \). Let’s choose the states \( s_i \) so that those variables always equal \( \text{TRUE} \). (Since \( \text{AltPgmSpec} \) doesn’t mention the variables \( \text{FGCallDone} \), and \( \text{FGRetrunDone} \), changing their values doesn’t affect whether or not a behavior satisfies that specification.) Behavior \( (2.7) \) then satisfies specification \( \text{FGAltPgmSpec} \), since that specification allows behaviors in which an execution of \( \text{Call} \) and \( \text{Return} \) consists of a single step. Specification \( \text{FGAltPgmSpec} \) also allows additional behaviors in which the \( c2 \) and \( r2 \) steps of \( (2.7) \) are replaced by multiple steps. For example, the step \( s_8 \xrightarrow{c_2} s_9 \) might be replaced by

\[
s_8 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow t_3 \longrightarrow s_9
\]

for some intermediate states \( t_1 \), \( t_2 \), and \( t_3 \).

The fine-grained program specification \( \text{FGAltPgmSpec} \) executes the non-atomic \( \text{Call} \) and \( \text{Return} \) operations alternately, an execution of \( \text{Call} \) finishing before the next execution of \( \text{Return} \) starts, and vice-versa. We can see this from the invariant \( \text{AltPgmInvariant} \) (defined on page 27 in Figure 2.2) of \( \text{AltPgmSpec} \). This invariant implies that \( pc.c = “c2” \) and \( pc.r = “r2” \) cannot both be true. This implies that the client cannot be ready to perform a \( \text{Call} \) when the server is ready to perform a \( \text{Return} \). Hence, if we expand the one-step execution of those operations by multi-step executions, executions of \( \text{Call} \) steps cannot be interleaved with executions of \( \text{Return} \) steps.

The fine-grained program specification therefore implements the specification \( \text{FGAltSpec} \) of finer-grained alternation. As a problem, you will show this by showing that \( \text{FGAltPgmSpec} \) implements \( \text{FGAltSpec} \), using the same refinement mapping under which \( \text{AltPgmSpec} \) implements \( \text{AltSpec} \), defined above:

\[
\tau \triangleq \begin{cases} 
\text{IF} \ (pc.c = “c3”) \lor (pc.r = “r3”) \ \text{THEN} \ (x + 1) \ % \ 2 \\
\text{ELSE} \ x
\end{cases}
\]

We will have more to say about the grain of atomicity in Section 3.5.
2.5 Fairness and Liveness

2.5.1 Client-Server Liveness

Our internal specification $AltSpec$ of the alternation system allows behaviors that halt in any state—that is, behaviors that take only stuttering steps from any point onwards. Formula $AltSpec$ is an example of a safety property. Intuitively, a safety property asserts that the system does not take wrong step. But it doesn't require that anything eventually happen. A property that asserts that something must eventually happen is called a liveness property. We now strengthen $AltSpec$ by adding a liveness property.

We consider the alternation system to be a client-server system in which call operations are requests and return operations are responses. We want the server to respond to every client request, but we don't want to require the client to keep sending requests. Thus, we allow the system to terminate after a return operation, but not after a call operation.

To express this liveness requirement, observe that, in any behavior satisfying the safety specification $AltSpec$, action $AltReturn$ is enabled in a state iff an $AltCall$ step has occurred and the corresponding $AltReturn$ step has not. We can therefore require that, if $AltReturn$ is enabled, then an $AltReturn$ step must eventually occur. The TLA+ formula that asserts this requirement is written $WF_{\langle x, iFace \rangle}(AltReturn)$. The WF stands for weak fairness; for now, ignore the subscript $\langle x, iFace \rangle$.

In general, ignoring the subscript, the formula $WF_v(A)$ is true of a behavior iff any one of the following equivalent conditions holds:

1. If $A$ ever becomes enabled forever, then an $A$ step must eventually occur.

2. If $A$ ever becomes enabled forever, then infinitely many $A$ steps must occur.

3. Either $A$ is infinitely often not enabled, or infinitely many $A$ steps occurs.

Many people find it hard to see that these are all equivalent. It is perhaps hardest to see that the first implies the second. To show that it does, we assume (1) holds and that $A$ does become enabled forever, and we show that infinitely many $A$ steps occur. Suppose $A$ becomes enabled from the $i^{th}$ state onwards. By (1), this implies that step $j_1$ is an $A$ step, for some $j_1 > i$. After that step, $A$ is still enabled forever. Therefore, by (1), step $j_2$ must also be an $A$ step for some $j_2 > j_1$. After that step, $A$ is still enabled forever, so step $j_3$ must also be an $A$ step for some $j_3 > j_2$. And so on.

In any behavior satisfying $AltSpec$, once action $AltReturn$ becomes enabled, it must stay enabled until an $AltReturn$ step occurs. Hence, the formula

$$CSAltSpec \triangleq AltSpec \land WF_{\langle x, iFace \rangle}(AltReturn)$$
implies that if AltReturn ever becomes enabled, then an AltReturn step must eventually occur. (That step happens to disable the AltReturn action, since it sets $x$ to 0.)

### 2.5.2 Non-stop Alternation

Instead of thinking of the alternation system as a client-server system, we can consider it to be a system that is supposed to alternate forever, without ever stopping. Let’s specify such a system.

We’ve seen that $AltSpec$ conjoined with weak fairness of the AltReturn action implies that the system can’t terminate in a state in which AltReturn is enabled. Analogously, $AltSpec$ conjoined with weak fairness of $AltCall$ and $AltSpec$ implies that the system can’t terminate in a state in which AltCall is enabled. Since $AltCall$ or AltReturn is enabled in any reachable state of the system, conjoining both weak fairness conditions yields a specification of an alternation system that never halts. We can therefore define

$$NonstopAltSpec \triangleq AltSpec \land WF_{\langle x, iFace \rangle}(AltCall) \land WF_{\langle x, iFace \rangle}(AltReturn)$$

to be the specification of an alternation system that never halts.

There’s an easier way to define $NonstopAltSpec$. In any behavior satisfying $AltSpec$, the next-state action $AltNext$ is always enabled. Therefore, weak fairness of $AltNext$ implies that infinitely many $AltNext$ actions must occur, which means that the system never stops. An equivalent definition of $NonstopAltSpec$ is therefore:

$$NonstopAltSpec \triangleq AltSpec \land WF_{\langle x, iFace \rangle}(AltNext)$$

### 2.5.3 Leads-To

Specification $CSAltSpec$ implies that, whenever the client sets $arg$ by executing a call operation, the server will eventually respond by setting $rtn$ to the value $RtnVal(arg)$. Thus, if $arg$ is ever equal to 5, then $rtn$ will then or at some later time be equal to $RtnVal(5)$ (which equals 25). The temporal formula $P > Q$ asserts that, if formula $P$ ever becomes true, then formula $Q$ will be true then or at some later time. The operator $>$ is read leads-to. Thus, $(arg = 5) > (rtn = RtnVal(5))$ is true for every behavior satisfying $CSAltSpec$. This is true if 5 is replaced by any other value. Hence, the client-server alternation specification satisfies the following liveness property:

$$CSLiveness \triangleq \forall n \in 0..4 : (arg = n) > (rtn = RtnVal(n))$$

It is easy to see that specification $CSAltSpec$ satisfies (implies) CSLiveness.
It is useful to check that a specification implies the leads-to properties that you think it should. Trying to do so can catch errors in the specification. However, it’s generally best not to use leads-to formulas as part of the specification, but to use fairness conditions on the specification’s actions instead. (See Section 8.9.2 of Specifying Systems for an explanation of why.)

2.5.4 The Subscripts

The subscript $v$ appears in a weak fairness formula $\text{WF}_v(A)$ to make sure that the formula is invariant under stuttering. If we could assert that a step of a stuttering action like $x' = x$ must eventually occur, then we would have a formula that can be made true by adding a stuttering step. The formula would then not make sense as a statement about a system, because it would not be invariant under stuttering.

The formula $\text{WF}_v(A)$ actually asserts that if an $A$ step that changes $v$ ever becomes forever enabled, then such a step must eventually occur. A step that changes a state function $v$ cannot be a stuttering step. In practice, we usually take $v$ to be the tuple of all the specification’s variables. A step changes this tuple if it is not a stuttering step of the system. The subscript $v$ is redundant if action $A$ does not allow stuttering steps. This is the case in the examples above, since neither $\text{AltCall}$, $\text{AltReturn}$, nor $\text{AltNext}$ allow stuttering steps. In fact, it will usually be the case for the actions we write. (More precisely, in any behavior satisfying the safety part of the specification, no step of these actions will be a stuttering step.) However, the subscript is required by the syntax of TLA$^+$. 

In our informal discussion, we usually ignore the subscripts and simply call $\text{WF}_v(A)$ weak fairness on action $A$.

2.6 The Two-Process Program in Java

We now consider the two-process alternation system in a real programming language. In Section 2.2 we introduced the parallel composition operator $||$ and the $\text{await}(\text{cond})$ statement to cause a process to wait until the condition $\text{cond}$ is true. In most programming languages, a naive method of expressing $\text{await}(\text{cond})$ is with the loop $\text{while}(\neg \text{cond})$. There is, however, no simple way to express parallel composition that is common across most programming languages. So, let’s look at the way parallel composition is done in Java.

In Java, a $\text{thread}$ is what we have been calling a process: a concurrent activity that shares an address space with other concurrent activities. To create a thread, you extend the $\text{Thread}$ class and place the thread’s code in the class’s $\text{run}$ method. A program can start a thread by creating an instance of this new class.
and invoking its `start` method. For example, the parallel composition `A || B` can be written in Java as:

```java
public class foo {

    static class CA extends Thread {
        public void run () {
            A;
        }
    }
    
    static class CB extends Thread {
        public void run () {
            B;
        }
    }

    public static void run(String [] args) {
        CA ca = new CA();
        CB cb = new CB();
        ca.start();
        cb.start();
    }
}
```

Figure 2.4 gives the Java equivalent of the two-line program at the beginning of Section 2.2. We’ve specified `Call` and `Return` to be statements that print “Call” and “Return” respectively so that the program is executable by itself.

There are at least two troubling aspects about this program, though. The first (and less obvious) one is that it may not actually implement `AltSpec` when run on some multiprocessors. We will discuss later why this might occur. The more obvious reason is that on uniprocessors, there will be periods during which neither process is doing anything productive.

Operating system schedulers run processes for a long time (that is, tens of milliseconds) before the running process is preempted in favor of a process that is ready to run. Consider the Caller thread. Once it sets `x` to 1, it immediately tests the value of `x` to see if it is equal to 0. The value of `x` will remain 1 until the Returner thread sets it to 0. That won’t happen until the Caller thread is preempted, which most likely won’t happen for a long time.

It would be more efficient if the Caller thread could inform the operating system that, having set `x` to 1, it won’t be ready to run again until `x` is 0. One common operating system primitive that gives this kind of ability is the `semaphore`. 
public class Alternation {
    static int x = 0;

    static class Caller extends Thread {
        public void run () {
            while (true) {
                while (x != 0) {
                    System.out.println("Call");
                    x = 1;
                }
            }
        }
    }

    static class Returner extends Thread {
        public void run () {
            while (true) {
                while (x != 1) {
                    System.out.println("Return");
                    x = 0;
                }
            }
        }
    }

    public static void main(String[] args) {
        Caller c = new Caller();
        Returner r = new Returner();
        c.start();
        r.start();
    }
}

Figure 2.4: The Java version of the program from Section 2.2.

2.6.1 Semaphores

A semaphore is an abstract data type that has a nonnegative integer value. There are two methods provided by a semaphore: P that atomically decrements its value and V that atomically increments its value. Since semaphores have nonnegative values, the P() method will block until the semaphore has a positive value. Figure 2.5 gives a simple TLA+ specification for semaphores.

Semaphores are useful because they are implemented by the operating system. If a process executes P(s) when s is zero, then the operating system yields the processor to another process that is ready to run. The blocked process made
ready to run only when another process executes \( V(s) \).

The \texttt{SemAlternation} module in Figure 2.6 defines the specification \texttt{SemAltSpec} of the semaphore implementation of alternation. To implement alternation, we use two semaphores: \( c\text{Sem} \) which is used to enable a \texttt{Call} and \( r\text{Sem} \) which is used to enable a \texttt{Return}. Since \texttt{Call} happens first, the initial value of \( c\text{Sem} \) is 1 and the initial value of \( r\text{Sem} \) is 0. Executing \texttt{Call} enables \texttt{Return}, and so the \texttt{SemAltCall} action asserts \( V(r\text{Sem}) \). Similarly, the \texttt{SemAltReturn} action asserts \( V(c\text{Sem}) \).

The \texttt{SemAlternation} module also gives an invariant \texttt{SemAltInvariant} that states that both \( c\text{Sem} \) and \( r\text{Sem} \) are either 0 or 1, and that they never have the same value (hence \( c\text{Sem} + r\text{Sem} = 1 \)). Thus, at any time exactly one of the actions \texttt{SemAltCall} and \texttt{SemAltReturn} is enabled. We can use TLC to show that \texttt{SemAltInvariant} is indeed an invariant, but it is easy to verify yourself:

- It holds in the initial state: \texttt{SemAltInit} \( \Rightarrow \texttt{SemAltInvariant} \);
- Each action leaves it true: \texttt{SemAltInit} \( \land \) \texttt{SemAltNext} \( \Rightarrow \texttt{SemAltInvariant} \).

To show \texttt{SemAltSpec} implements \texttt{AltSpec} under the refinement mapping \( x \leftarrow r\text{Sem} \), we can show:

- \texttt{SemAltInit} \( \Rightarrow \texttt{AltInit} \).
- \texttt{SemAltInvariant} \( \land \) \texttt{SemAltNext} \( \| c\text{Sem}, r\text{Sem}, i\text{Face} \) \( \Rightarrow \) \texttt{AltNext} \( \| i\text{Face}, \neg \).

The first implication is obvious. If we expand the second implication using the definition of \( \square[A] \), we will eventually find ourselves wanting to show:

- \texttt{SemAltInvariant} \( \land \) \texttt{SemAltCall} \( \Rightarrow (r\text{Sem} = 0) \land \texttt{Call} \land (r\text{Sem}' = 1) \)
- \texttt{SemAltInvariant} \( \land \) \texttt{SemAltReturn} \( \Rightarrow (r\text{Sem} = 1) \land \texttt{Call} \land (r\text{Sem}' = 0) \)

These are both straightforward to show. For example, from \texttt{SemAltInvariant}, \( c\text{Sem} + r\text{Sem} = 1 \), and since \texttt{SemAltCall} implies \( c\text{Sem} = 1 \), we have \( r\text{Sem} = 0 \). The other two conjuncts in the consequent are also in the antecedent.

The Java class in Figure 2.7 implements semaphores. We will ignore for now how it does so. Figure 2.8 shows a Java program that implements alternation using this semaphore class.
CHAPTER 2. ALTERNATION AS A MULTIPROCESS SYSTEM

MODULE SemAlternation
EXTENDS CallReturn, Semaphores
VARIABLES cSem, rSem

SemAltInit $\triangleq \land CRInit$
  $\land cSem = 1$
  $\land rSem = 0$

SemAltCall $\triangleq \land P(cSem)$
  $\land Call$
  $\land V(rSem)$

SemAltReturn $\triangleq \land P(rSem)$
  $\land Return$
  $\land V(cSem)$

SemAltNext $\triangleq$ SemAltCall $\lor$ SemAltReturn

SemAltSpec $\triangleq$ SemAltInit $\land [\square[SemAltNext]_{\{Face, cSem, rSem\}}]$

SemAltInvariant $\triangleq \land CRInvariant$
  $\land cSem \in \{0, 1\}$
  $\land rSem \in \{0, 1\}$
  $\land cSem + rSem = 1$

THEOREM SemAltSpec $\Rightarrow \square SemAltInvariant$

Figure 2.6: The semaphore implementation of alternation.

2.7 Summary

Here are some of the concepts we introduced in this chapter:

- An action is enabled in a state $s$ if there is some state $t$ so that $s \rightarrow t$ is a step that satisfies the action.

- A stuttering step is one in which the system’s variables are unchanged. Our specifications are invariant under stuttering, meaning that adding or removing a stuttering step from a behavior does not affect whether or not the behavior satisfies the specification. A behavior ending in an infinite sequence of stuttering steps represents a history of the universe in which the system halts.

- $[A]_v \triangleq A \lor (v' = v)$, for any action $A$ and state function $v$. A $[A]_v$ step is either an $A$ step or one that leaves $v$ unchanged.
2.7. SUMMARY

public class Semaphore {
    int s;

    public Semaphore (int s) { this.s = s; }

    public synchronized void P () {
        while (s == 0)
            try { wait(); }
            catch (java.lang.InterruptedException e) { }
        s = s - 1;
    }

    public synchronized void V () {
        s = s + 1;
        notify();
    }
}

Figure 2.7: A Java Semaphore class.

- A refinement mapping is a substitution

\[
x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n
\]

of expressions \( e_i \) for some variables \( x_i \) of a specification. A specification \( S_1 \) implements a specification \( S_2 \) under this refinement mapping iff \( S_1 \Rightarrow S_2 \), where \( S_2 \) is the formula obtained from \( S_2 \) by substituting the \( e_i \) for the \( x_i \). If \( S_1 \) and \( S_2 \) are defined by

\[
S_1 \triangleq Init_1 \land \Box [Next_1]_{x_1}
\]
\[
S_2 \triangleq Init_2 \land \Box [Next_2]_{x_2}
\]

we prove \( S_1 \Rightarrow S_2 \) by finding an invariant \( Inv \) of \( S_1 \) and proving:

- \( S_1 \Rightarrow \Box Inv \)
- \( Init_1 \Rightarrow \Box Init_2 \)
- \( [Next_1]_{x_1} \land Inv \Rightarrow [\overline{Next_2}]_{x_2} \)

If \( S_1 \) implements \( S_2 \) under this refinement mapping, then \( S_1 \) implements (implies) \( \exists x_1, \ldots, x_2 : S_n \).

- We can represent a system at different grains of atomicity. We want to choose the coarsest-grained representation that captures the aspects of the system that concern us.
public class semalt {
    static Semaphore c = new Semaphore(1);
    static Semaphore r = new Semaphore(0);

    static class Caller extends Thread {
        public void run () {
            while (true) {
                c.p();
                System.out.println("Call");
                r.v();
            }
        }
    }

    static class Returner extends Thread {
        public void run () {
            while (true) {
                r.p();
                System.out.println("Return");
                c.v();
            }
        }
    }

    public static void main(String[] args) {
        Caller c = new Caller();
        Returner r = new Returner();
        c.start();
        r.start();
    }
}

Figure 2.8: Alternation with semaphores in Java.
2.8 Problems

Problem 2.1 Use TLC to check that the alternation program implements the discrete oscillator.

Problem 2.2 Section 2.4.2 sketches specifications FGAltSpec, a finer-grained specification of alternation, and FGAltPgmSpec, a finer-grained specification of the alternation program. Write modules FGAlternation and FGAlternationPgm that define these specifications. Use TLC to show that FGAltPgmSpec implements FGAltSpec.

Problem 2.3 (a) Use TLC to check that specification CSAltSpec of Section 2.5.1 satisfies formula CSLiveness of Section 2.5.3. (b) Use TLC to show that the two versions of specification NonstopAltSpec in Section 2.5.2 are equivalent.

Hint: For part (b), take AltSpec as the specification and the equivalence of the two fairness conditions as the property to be checked. TLC Version 1 does not handle a property asserting the equivalence of two formulas; you must instead have it check that each formula implies the other.

Problem 2.4 Write a specification of a discrete oscillator that never stops. Use TLC to show that it is implemented by NonstopAltSpec, but not by CSAltSpec.

Problem 2.5 Write a specification of the alternation program with client-server fairness. Check that it satisfies the same leads-to property as CSAltSpec.

Problem 2.6 Write a fine-grained version of the two-phase handshake protocol of Section 2.1 that uses module FGCallReturn, and show that it implements the fine-grained alternation program specification FGAltPgmSpec that you wrote in Problem 2.2 above the refinement mapping $x \leftarrow (c + r) \% 2$. 

- Safety properties assert what the system is allowed to do; liveness properties assert what it must do. We can express liveness properties with the weak fairness operator $\langle \rangle$. A thread is created in Java as a side effect of creating an object of a class that extends Thread. The code that is executed is defined by the run method of the class.

- Busy waiting is, in general, not an efficient way to use a processor. If a process is waiting for some other process to take a step, then the operating system should switch execution contexts to let the other process run.

- Semaphores are a convenient primitive to use in building concurrent programs. They are relatively easy to use, and they do not use busy waiting.
Problem 2.7 Add liveness to the fine-grained specification of Problem 2.2 and use TLC to show that $FGAltPgmSpec$ still implements $FGAltSpec$ under the refinement mapping.

Problem 2.8 Figure 1.1 specifies alternation with a variable $x$: if $x = 0$ then it is time for a Call action, and if $x = 1$ it is time for a Return action. In this problem, you will extend this system to a ring of $N$ processes, for an arbitrary positive $N$. Figure 2.9 shows such a system for $N = 8$. The processes in the ring have no intrinsic first element, but we will number them $\{0, 1, \ldots, N-1\}$. The processes perform their operations in round-robin order:

$$Relay0 \rightarrow Relay1 \rightarrow \cdots RelayN-1 \rightarrow Relay0 \rightarrow \cdots$$

First, write a specification $RelayRing$, similar to $Alternation$, of this relay ring system for $N = 3$ (do not let $N$ be a constant that is defined outside of the TLA$^+$ specification. If you did, then you would need to know how to use TLA$^+$ functions and recursion, which we haven’t covered yet). Let the action executed by each process $i$ be $Relay(i)$. Then, come up with a specification $TokenPass$ that implements $RelayRing$ by using a single wire between each pair of processes in the ring. The wire joining processes $i$ and $(i + 1) \% N$ will be set by process $i$ alternately to 1 and 0. For $i > 0$, process $i$’s action will be enabled when the value it has for wire $i$ is different from that of wire $i - 1$. Process 0’s action will be enabled when the bit on wire 0 equals that on wire $N - 1$. Verify, using TLC, that $TokenPass$ implements $RelayRing$ under an appropriate refinement mapping.

Problem 2.9 Give a semaphore-based specification $SemRelayRing$ that implements $RelayRing$ under an appropriate refinement mapping. Use one semaphore for each pair of processes in the ring. Again, assume that $N = 3$. Then, implement this specification in Java using the $Semaphore$ class.
2.8. PROBLEMS

**Problem 2.10** In Section 2.6.1 we noted that operating system schedulers run processes for a long time before the running process is preempted in favor of a process that is ready to run. This period of time is called the *scheduling quantum*. Write a Java program that empirically determines the value of the scheduling quantum. You can use the method `System.nanoTime()` to measure elapsed time.