Chapter 3

Producer-Consumer Synchronization

3.1 Pipelined Alternation

In the alternation system, a client issues a sequence of calls, to which a server replies with returns. The client waits for a return before issuing the next call. We now generalize this system to one in which the client need not wait for a return before issuing the next call, but can get up to N calls ahead, for some positive integer N. Alternation is the case when $N = 1$. The process of issuing a new request before the previous one has been completed is often known as pipelining. This particular form of pipelining is usually known as a *producer-consumer* system.

With multiple calls outstanding, it becomes important to keep track of the argument of each call to make sure that, for each i, the i^{th} return provides the correct response to the i^{th} call. We therefore redefine the *Call* and *Return* actions so they take an explicit argument, where action $Call(v)$ represents issuing a call with argument v, and action $Return(v)$ represents issuing return with value v. We first write the specification *PCSpec* of the producer-consumer system, and then we write a module *PCCallReturn* that defines the new *Call* and *Return* operators.

3.1.1 The Safety Specification

To write a TLA$^+$ specification of the producer-consumer system, we need to introduce the constant parameter N. This is done with the statement

$$\textsc{constant}\ N$$

We assume that N is a positive integer. It's a good idea to make that assumption explicit, which we do in TLA$^+$ with the following statement, where *Nat* is defined by the *Naturals* module to be the set of natural numbers (non-negative integers).

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

The producer-consumer system's specification requires an internal variable to keep track of the sequence of outstanding calls for which no return has yet been issued. We use a variable q whose value is the sequence of outstanding call values. The standard way to represent a finite sequence in TLA$^+$ is as a tuple, where we regard $\langle e_1, e_2, e_3 \rangle$ to be the three-element sequence whose first element (head) is e_1. TLA$^+$ has a standard *Sequences* module that defines the following operators for finite sequences, where s is assumed to be a sequence:

Computer scientists often use the term *list* instead of *finite sequence.* They also have a bizarre habit of labeling sequence elements with cardinal numbers instead of ordinal numbers—for example, writing $\langle e_0, e_1, e_2 \rangle$, so e_1 is the *second* element of the sequence!

- $\mathit{Head}(s)$ is the first element of s, if s is nonempty.
- $\mathit{Tail}(s)$ is the sequence obtained from s by deleting the first element, if s is nonempty.
- $\mathit{Append}(s, e)$ is the sequence obtained by appending the element e to the tail of s.
- $\mathit{Len}(s)$ is the length of (number of components of) s.
- $\mathit{Seq}(E)$ is the set of all finite sequences whose components are elements of the set E.

We assume that the *PCCallReturn* module defines *Input* to be the set of possible call values, so q will be a finite sequence of elements of *Input*. In other words, the type of q will be $\mathit{Seq}(\mathit{Input})$. Moreover, since there will be at most N outstanding unanswered calls, q will have length at most N. So, the type invariant for q is

$$(q \in \mathit{Seq}(\mathit{Input})) \land (\mathit{Len}(q) \leq N)$$

Initially, q will be the empty sequence, which is represented by the 0-tuple $\langle\ \rangle$. So, the initial condition for q is:

$$q = \langle\ \rangle$$

The next-state action *PCNext* is the disjunction of two actions, one that performs the *Call* and the other the return. We define $\mathit{PCCall}(v)$ to be the action that performs a call step with argument v, and we define *PCReturn* to be the action that performs a return step. (Unlike the call value, the return value is determined; it must equal $\mathit{RtnVal}(v)$, where v is the corresponding call value.) So, we have

$$\mathit{PCNext} \triangleq (\exists\, v \in \mathit{Input} : \mathit{PCCall}(v)) \lor \mathit{PCReturn}$$

A call can be issued whenever there are fewer than N outstanding unanswered calls. Hence, action $PCCall(v)$ is enabled iff the length of q is less than N. It performs a $Call(v)$ step and appends v to the tail of q. The definition is thus:

$$\begin{array}{lcl} PCCall(v) & \triangleq & \land\ Len(q) < N \\ & & \land\ Call(v) \\ & & \land\ q' = Append(q,\ v) \end{array}$$

A *PCReturn* action is enabled whenever there is some unanswered call—that is, when q is nonempty. It returns *RtnVal* applied to the earliest unanswered call value, which is the one at the head of q; and it removes the element at the head of q. Its definition is:

The conditions $Len(q) > 0$ and $q \neq \langle\,\rangle$ are equivalent, if q is a sequence.

$$\begin{array}{lcl} PCReturn & \triangleq & \land\ Len(q) > 0 \\ & & \land\ Return(RtnVal(Head(q))) \\ & & \land\ q' = Tail(q) \end{array}$$

The complete specification of the producer-consumer system appears in module *ProducerConsumer* of Figure 3.1 on the next page. We are assuming again that function *iFace* is still the tuple of all variables declared in module *PCCallReturn* and that *CRInit* is the initial predicate for these variables. (Actually, *iFace* will be a variable—the only variable declared in *PCCallReturn*.) We now use the name *TypeOK* rather than *Invariant* for simple type-correctness invariants. The *IProducerConsumer* module thus defines the type-correctness invariant *PCTypeOK*, assuming that *CRTypeOK* is the type-correctness invariant from module *PCCallReturn*.

3.1.2 Liveness

Specification *PCSpec* does not mention liveness. As with alternation, there are two possible liveness requirements. If we view the producer-consumer system as a client-server system, then we want to require that the server issue a return for every call. In other words, a *PCReturn* should be issued if there is an outstanding unanswered call. This is the case precisely if the *PCReturn* action is enabled. Therefore, weak fairness of the *PCReturn* action is the desired fairness condition. Our internal specification of a client-server producer-consumer system is therefore:

$$CSPCSpec \triangleq PCSpec \land \mathrm{WF}_{\langle q,\, iFace \rangle}(PCReturn)$$

We might also want to specify a non-stop producer-consumer system, in which the client keeps issuing calls and the server keeps issuing returns. Weak fairness of $\exists v \in Input : PCCall(v)$ implies that, whenever q is not full (has fewer than N elements), the client must eventually issue a call and insert an element into q, ensuring that q is not empty. Weak fairness of *PCReturn* implies that, whenever

MODULE *ProducerConsumer*

EXTENDS *PCCallReturn*, *Sequences*, *Naturals*

CONSTANT N

ASSUME $(N \in Nat) \land (N > 0)$

VARIABLE q

$$PCTypeOK \triangleq \begin{array}{l} \land\ CRTypeOK \\ \land\ q \in Seq(Input) \\ \land\ Len(q) \leq N \end{array}$$

$$PCInit \triangleq CRInit \land q = \langle \rangle$$

$$PCCall(v) \triangleq \begin{array}{l} \land\ Len(q) < N \\ \land\ Call(v) \\ \land\ q' = Append(q, v) \end{array}$$

$$PCReturn \triangleq \begin{array}{l} \land\ Len(q) > 0 \\ \land\ Return(RtnVal(Head(q))) \\ \land\ q' = Tail(q) \end{array}$$

$$PCNext \triangleq (\exists\, v \in Input : PCCall(v)) \lor PCReturn$$

$$PCSpec \triangleq PCInit \land \Box[PCNext]_{\langle q,\, iFace \rangle}$$

THEOREM $PCSpec \Rightarrow \Box PCTypeOK$

Figure 3.1: The specification of the producer-consumer system.

q is nonempty, the server must eventually issue a return and remove an element from q, ensuring that q is not full. The conjunction of both weak fairness conditions implies that the client keeps issuing calls and the server keeps issuing returns. Hence, the specification of a non-stop producer-consumer system is:

$$NonstopPCSpec \triangleq \begin{array}{l} \land\ PCSpec \\ \land\ \mathrm{WF}_{\langle q,\, iFace \rangle}(\exists v \in Input : PCCall(v)) \\ \land\ \mathrm{WF}_{\langle q,\, iFace \rangle}(PCReturn) \end{array}$$

What about weak fairness of the next-state action *PCNext*? Does that imply that calls and returns keep occurring? At first glance, it appears not to. Weak fairness of *PCNext* asserts that if it is enabled, meaning that if a call or return can be performed, then a call or a return must eventually be performed. If the system could keep performing calls forever, then it would never have to perform a return. However, this can't happen because performing only calls would fill

q. The client can issue at most N calls without the server issuing a return. Likewise, the server can issue at most N returns before the q becomes empty and it must wait for the client to issue a call. Thus, an equivalent way to specify the non-stop producer-consumer system is by defining

$$\mathit{NonstopPCSpec} \;\triangleq\; \mathit{PCSpec} \land \mathrm{WF}_{\langle q,\, \mathit{iFace} \rangle}(\mathit{PCNext})$$

The two definitions of *NonstopPCSpec* are equivalent only because neither the client nor the server can perform an unbounded number of steps without waiting for the other. If we removed the enabling condition $\mathit{Len}(q) < N$ from the $\mathit{PCCall}(v)$ action, then q could be of unbounded length, and the second definition would allow behaviors in which the client performed infinitely many calls while the server performed only finitely many returns (perhaps none). The first definition would still require that there be infinitely many returns, so the two definitions would not be equivalent.

3.1.3 The *PCCallReturn* Module

We now write the *PCCallReturn* module, which defines the actions $\mathit{Call}(v)$ and $\mathit{Return}(v)$, as well *Input*, *RtnFcn*, *iFace*, *CRInit*, and *CRTypeOK*. This time, we make the definitions more general than before. Instead of defining *Input*, the set of possible call values, to be some specific set, we declare it to be a constant parameter that can denote any set. We also declare *RtnFcn* to be an operator parameter—an arbitrary constant operator that takes a single argument. This is all done with the declaration:

$$\textsc{constants}\ \mathit{Input},\ \mathit{RtnVal}(\_)$$

We define *Output*, the set of possible output values, to be the set of all elements of the form $\mathit{RtnVal}(v)$, where v is any element of *Input*. This is written in TLA$^+$ as:

$$\mathit{Output} \;\triangleq\; \{\mathit{RtnVal}(v) : v \in \mathit{Input}\}$$

Instead of defining the interface state function *iFace* to be a pair of variables $\langle \mathit{arg}, \mathit{rtn} \rangle$, we declare *iFace* to be a variable and let its value be a record with *arg* and *rtn* components. We also correct a problem with our previous definition of the *Call* and *Return* actions. Suppose the client issues two successive calls with the same value. The second *Call* and *Return* steps do not change *arg* or *rtn*; they are stuttering steps. In the specification *AltSpec*, we can tell that the call and return occur because the value of x changes. However, if we hide x, then we are just left with stuttering steps. There is no way to tell that something has happened.

```
──────────────── MODULE PCCallReturn ────────────────
EXTENDS Naturals
CONSTANTS Input, RtnVal(_)
Output ≜ {RtnVal(v) : v ∈ Input}
─────────────────────────────────────────────────────
VARIABLE iFace

CRTypeOK  ≜  iFace ∈ [arg : Input,   aBit : {0, 1},
                      rtn : Output,  rBit : {0, 1}]

CRInit    ≜  ∃ v ∈ Input, w ∈ Output : iFace = [arg ↦ v,  aBit ↦ 0,
                                                rtn ↦ w,  rBit ↦ 0]

Call(v)   ≜  iFace' = [arg ↦ v,           aBit ↦ (iFace.aBit + 1)%2,
                      rtn ↦ iFace.rtn,  rBit ↦ iFace.rBit]

Return(v) ≜  iFace' = [arg ↦ iFace.arg,  aBit ↦ iFace.aBit,
                      rtn ↦ v,          rBit ↦ (iFace.rBit + 1)%2]
─────────────────────────────────────────────────────
```

Figure 3.2: Module *PCCallReturn*.

In a real implementation, something changes when a call is issued with the same value as the previous call. For example, a software procedure call is performed by storing the argument on a stack and moving the stack pointer. Regardless of whether the argument is the same as before, the stack pointer is changed by the call. A return from a software procedure also involves moving a stack pointer. We don't want to describe any particular call and return mechanism. Instead, we add to *iFace* a one-bit record component *aBit* that is complemented by a $Call(v)$ step, and a component *rBit* that is complemented by a $Return(v)$ step. The definitions of $Call(v)$ and $Return(v)$ are now straightforward—for example:

$$\begin{array}{lll} Call(v) & \triangleq & iFace' = [arg \mapsto v, \quad aBit \mapsto (iFace.aBit + 1)\%2, \\ & & \qquad rtn \mapsto iFace.rtn, \; rBit \mapsto iFace.rBit] \end{array}$$

The only remaining problem is deciding what the initial value of *iFace* should be. We let the initial values of $iFace.arg$ and $iFace.rtn$ be any elements of *Input* and output, respectively. We let $iFace.aBit$ and $iFace.rBit$ initially equal 0. This initial condition is expressed mathematically by asserting that there exists a value v in *Input* and a value w in *Output* such that *iFace* equals

$$[arg \mapsto v,\ aBit \mapsto 0,\ rtn \mapsto w,\ rBit \mapsto 0]$$

The complete module *PCCallReturn* appears in Figure 3.2 on this page.

3.1.4 Running TLC on Producer-Consumer

Let's now use TLC to check that the internal producer-consumer specification *PCSpec* satisfies its type invariant *PCTypeOK*. (Note that our fairness conditions affect only eventual behavior, not safety, so they have no influence on whether or not an invariant holds.) Our producer-consumer specification is parametrized. To determine if any particular behavior satisfies *PCSpec*, we have to know the values of the constant parameters N, *Input*, and *RtnVal*.

TLC can't determine if a property holds for all possible values of these parameters. It can check the property only for a particular *model* of the system, which is obtained by choosing particular values for these constants. We can easily specify the values of N and *Input* with a `CONSTANTS` statement in the configuration file. For example, we can tell TLC to let N equal 3 and *Input* equal the set $\{2, 3, 5\}$. However, because *RtnVal* is an operator that takes an argument, we must substitute a defined operator for it. So, we define an operator *MCRtnVal* by

$$MCRtnVal(v) \triangleq v^2$$

and we put the following statement in the configuration file:

Note the use of "`=`" when telling TLC to use particular values for N and *Input*, and the use of "`<-`" when telling it to substitute a defined operator for *RtnVal*.

```
CONSTANTS N = 3
          Input = {2,3,5}
          RtnVal <- MCRtnVal
```

But where do we put the definition of *MCRtnVal*? We could simply add it to the *IProducerConsumer* module. However, it's best not to modify a specification just to check it with TLC. Instead, we create this separate module for the purpose:

MODULE *MCProducerConsumer*

EXTENDS *IProducerConsumer*, *Naturals*

$MCRtnVal(v) \triangleq v^2$

and we run TLC on this module.

3.2 A Producer-Consumer Program

3.2.1 The TLA$^+$ Specification

We now describe an implementation of the producer-consumer system as a two-process program. It's traditional to use the names *producer* and *consumer* rather than *client* and *server* for the processes. We implement the sequence q with an

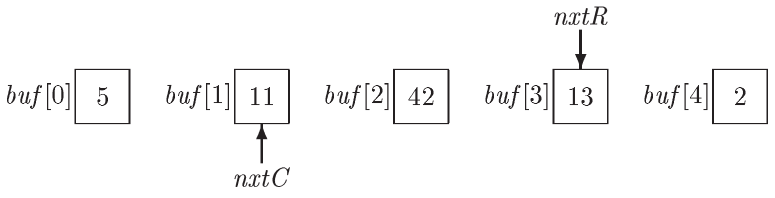
```
      while (true) { ⟪P(cSem);
                       buf[nxtC] = ?;
                       Call(buf[nxtC]);
                       nxtC = (nxtC + 1) % N;
                       V(rSem);⟫}

  ||  while (true) { ⟪P(rSem);
                       buf[nxtR] = RtnVal(buf[nxtR]);
                       Return(buf[nxtR]);
                       nxtR = (nxtR + 1) % N;
                       V(cSem);⟫}
```

Figure 3.3: A coarse-grained semaphore producer-consumer program.

N-element array *buf* and two pointers, *nxtC* and *nxtR*. We let *nxtC* point to the array element where the next call argument is to be put, and we let *nxtR* point to the array element from where the argument for the next return is stored, regarding *buf* as a ring buffer. For example, if $N = 5$, the state

It's most convenient to number the elements of *buf* from 0 through $N - 1$, so element $(i+1) \% N$ is the next one after element i. (A ring buffer has no unique first element.)



represents $q = \langle 13, 2, 5 \rangle$.

The consumer has to wait until the buffer is not empty, and the producer has to wait until it is not full. To avoid busy waiting, we have the processes wait with semaphore P operations. Our implementation generalizes the semaphore implementation of alternation, using two semaphores *cSem* and *rSem*. The value of *cSem* is the number of free buffer elements; the value of *rSem* is the number of full ones. Initially, $cSem = N$ and $rSem = 0$. We have the producer store the argument in the buffer element when it issues the call; and we have the consumer replace the argument with the return value when issuing the return. The program is written informally in Figure 3.3 on this page, where "?" denotes an arbitrarily-chosen argument value. This is a coarse-grained program, where the entire operations of the producer and of the consumer are atomic actions, each executed as a single step.

Note that the *c* in *cSem* stands for *call*, not *consumer*.

We now write a TLA$^+$ specification of this program. But first, we need to explain how to represent arrays in TLA$^+$. What a programmer calls an array f indexed by a set S, a mathematician calls a function f with domain S. We use the terminology of mathematicians when talking about specifications, so the variable *buf* in our specification will be a function with domain $0 \,..\, (N - 1)$.

Unlike ordinary mathematics, TLA$^+$ uses square brackets for function application. Thus, $buf[3]$ is the value obtained by applying the function buf to the element 3.

For any sets S and T, the TLA$^+$ expression $[S \rightarrow T]$ denotes the set of all functions with domain S and range in T. In other words, it is the set of all functions f with domain S such that $f(e) \in T$ for all $e \in S$. In every behavior satisfying our specification, the value of $buf[i]$ will be either a call argument or a return value, for $0 \leq i < N$. The set of all call values is *Input* and the set of all return values is *Output*, so buf will be an element of $[0 \,..\, (N-1) \rightarrow Input \cup Output]$.

In TLA$^+$, we represent the change to buf produced by the programming-language assignment statement `buf[i]=v` with a formula of the form $buf' = exp$, for some expression exp. This expression must be a function $\widehat{buf}$ that is the same as buf, except with $\widehat{buf}[i]$ equal to v. We write this expression in TLA$^+$ as $[buf \text{ EXCEPT } ![i] = v]$. (Don't try to make any sense of the individual pieces of this expression—except for the subexpressions buf, i, and v.)

We have now explained all the notation we need to write the TLA$^+$ specification *SemPCSpec* of the two-process producer-consumer semaphore program. It appears as module *SemProducerConsumer*, in Figure 3.4 on the next page. We have again separated the invariant into two parts: the simple type invariant *SemPCTypeOK* and the more interesting invariant *SemPCInvariant*.

To show that *SemPCSpec* satisfies the specification *PCSpec* of the producer-consumer system, we show that it implements the internal specification *PCSpec* under a refinement mapping. We use a refinement mapping that substitutes for q the state function $\overline{q}$, whose value is the sequence of unanswered call arguments. As we've observed, the length of $\overline{q}$ is value of $rSem$. The variable $nxtR$ points to the next call argument to be responded to, which is at the head of $\overline{q}$. Thus, the head (first element) of $\overline{q}$ is $buf[nxtR]$, so the i^{th} element of $\overline{q}$ is $buf[(nxtR + i - 1) \% N]$.

To define $\overline{q}$ in TLA$^+$, we need to know two things: how sequences are described by functions, and how to describe a function. We've already said that an m-element sequence is represented an m-tuple. In TLA$^+$, an m-tuple is a function whose domain is the set $1 \,..\, m$. For example, $\langle e_1, e_2, e_3 \rangle[2]$ equals e_2. So, $\overline{q}$ is the function with domain $1 \,..\, rSem$ such that $\overline{q}[i]$ equals $buf[(nxtR + i - 1) \% N]$, for all i in $1 \,..\, rSem$. This function is written in TLA$^+$ as

$$[i \in 1 \,..\, rSem \mapsto buf[(nxtR + i - 1) \% N]]$$

In general, $[id \in S \mapsto F(id)]$ is the function f with domain S such that $f[id] = F(id)$, for any $id \in S$. The identifier id is a bound symbol in this expression.

Our semaphore producer-consumer specification *SemPCSpec* implements the producer-consumer specification *PCSpec* under the substitution (refinement map-

MODULE $\mathit{SemProducerConsumer}$

EXTENDS $\mathit{PCCallReturn}, \mathit{Semaphores}, \mathit{Naturals}$

CONSTANT N

ASSUME $(N \in \mathit{Nat}) \land (N > 0)$

VARIABLES $\mathit{buf}, \mathit{nxtC}, \mathit{nxtR}, \mathit{cSem}, \mathit{rSem}$

$$\begin{aligned}\mathit{SemPCTypeOK} \triangleq\ &\land \mathit{CRTypeOK}\\ &\land \mathit{buf} \in [0 \mathrel{..} (N \quad - 1) \rightarrow \mathit{Input} \cup \mathit{Output}]\\ &\land \mathit{nxtC}\ \in 0 \mathrel{..} (N-1)\\ &\land \mathit{nxtR}\ \in 0 \mathrel{..} (N-1)\\ &\land \mathit{cSem} \in 0 \mathrel{..} N\\ &\land \mathit{rSem} \in 0 \mathrel{..} N\end{aligned}$$

$$\mathit{SemPCInvariant} \triangleq \mathit{cSem} + \mathit{rSem} = N$$

$$\begin{aligned}\mathit{SemPCInit} \triangleq\ &\land \mathit{CRInit}\\ &\land \mathit{buf} \quad \in [0 \mathrel{..} (N-1) \rightarrow \mathit{Output}]\\ &\land \mathit{nxtC}\ = 0\\ &\land \mathit{nxtR}\ = 0\\ &\land \mathit{cSem} = N\\ &\land \mathit{rSem} = 0\end{aligned}$$

$$\begin{aligned}\mathit{SemPCCall}(v) \triangleq\ &\land P(\mathit{cSem})\\ &\land \mathit{buf}' = [\mathit{buf}\ \text{EXCEPT}\ ![\mathit{nxtC}] = v]\\ &\land \mathit{Call}(v)\\ &\land \mathit{nxtC}' = (\mathit{nxtC} + 1)\%N\\ &\land V(\mathit{rSem})\\ &\land \text{UNCHANGED}\ \mathit{nxtR}\end{aligned}$$

$$\begin{aligned}\mathit{SemPCReturn} \triangleq\ &\land P(\mathit{rSem})\\ &\land \mathit{buf}' = [\mathit{buf}\ \text{EXCEPT}\ ![\mathit{nxtR}] = \mathit{RtnVal}(\mathit{buf}[\mathit{nxtR}])]\\ &\land \mathit{Return}(\mathit{buf}'[\mathit{nxtR}])\\ &\land \mathit{nxtR}' = (\mathit{nxtR} + 1)\%N\\ &\land V(\mathit{cSem})\\ &\land \text{UNCHANGED}\ \mathit{nxtC}\end{aligned}$$

$$\mathit{SemPCNext} \triangleq (\exists\, v \in \mathit{Input} : \mathit{SemPCCall}(v)) \lor \mathit{SemPCReturn}$$

$$\mathit{SemPCSpec} \triangleq \mathit{SemPCInit} \ \land\ \Box[\mathit{SemPCNext}]_{\langle \mathit{buf}, \mathit{nxtC}, \mathit{nxtR}, \mathit{cSem}, \mathit{rSem}, \mathit{iFace}\rangle}$$

THEOREM $\mathit{SemPCSpec} \Rightarrow \Box(\mathit{SemPCTypeOK} \land \mathit{SemPCInvariant})$

Figure 3.4: The producer-consumer semaphore program.

ping)

(3.1) $q \leftarrow [i \in 1 \,..\, rSem \mapsto buf[(nxtR + i - 1) \,\%\, N]\,]$

In Problem 3.6, you will verify that *SemPCSpec* does implement *PCSpec* under this refinement mapping.

Formula *SemPCSpec* is just a safety specification. As with the producer-consumer specification *PCSpec*, there are two reasonable liveness properties we can add. We can specify a non-stop system, which performs infinitely many calls and returns, by conjoining one of the following two conditions: (i) weak fairness of each of the two actions $\exists v \in Input : SemPCCall(v)$ and *SemPCReturn*, or (ii) weak fairness of the next-state action *SemPCNext*.

3.2.2 A Java Implementation

It is a simple exercise to translate the `SemProducerConsumer` specification into Java. In Figure 3.5 is a Java public class definition of a bounded buffer object.

A bounded buffer is one of the most frequently used abstractions in concurrent systems because it decouples somewhat the rate that one process calls *Insert* from the rate that the other process calls *Remove*. For example, a virtual memory system uses this abstraction to decouple the rate that a process consumes page frames (the removing process) from the rate that a cleaner process produces clean page frames by writing dirty frames back to backing store (the inserting process). When one instantiates a bounded buffer, though, one has to decide on a value of N. This may not be easy to do.

If the producer and consumer run at constant rates, then the value of N should be two. For example, suppose that the producer runs a bit faster than the consumer. Eventually, the producer will fill all N buffers and will wait for the consumer to consume a value. Thus, the producer will eventually run at the rate of the consumer, and the only parallelism we will get is the overlap between the consumer accessing one buffer and the producer filling another. Hence, two buffers (often called “double buffering”) is sufficient. A similar argument holds if the producer runs more slowly than the consumer. If the processes don’t create or use buffers at a constant rate, however, then a larger value of N could help. For example, in the virtual memory system, there should be enough buffers to accommodate the change in the working set of a process, since these requests will arrive in a burst. If there are not enough, then the each page fault will require a page frame to be first cleaned, which would greatly slow down the page fault processing.

```
public class SemProducerConsumer {
    private int N, buf[], nxtC, nxtR;
    private Semaphore cSem, rSem;

    public SemProducerConsumer (int N) {
        this.N = N;
        buf = new int[N];
        nxtC = 0;
        nxtR = 0;
        cSem = new Semaphore(N);
        rSem = new Semaphore(0);
    }

    public void Insert (int i) {
        cSem.P();
        buf[nxtC] = i;
        nxtC = (nxtC + 1) % N;
        rSem.V();
    }

    public int Remove () {
        int i;
        rSem.P();
        i = buf[nxtR];
        nxtR = (nxtR + 1) % N;
        cSem.V();
        return i;
    }
}
```

Figure 3.5: Java bounded-buffer class.

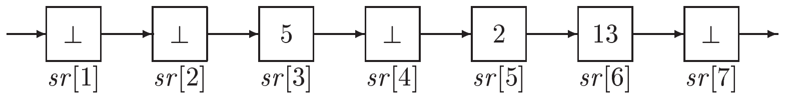
3.3 A Shift Register

3.3.1 The Safety Specification

We have been thinking of the producer-consumer system as one in which a sequence of arguments v_1, v_2, ... are sent by the client to the server, and the server responds with the sequence of results $\mathit{RtnVal}(v_1)$, $\mathit{RtnVal}(v_2)$, ... At any time, the client can have sent at most N more values than the server.

An important instance of this system is one in which *RtnVal* is the identity operator, so $\mathit{RtnVal}(v) = v$ for all v. There is obviously no point in a client calling a server to compute the identity operator. In this case, the producer-consumer system models a system in which a *sender* process sends values to a *receiver* process. The call operation represents the sending of a value by the sender, and the return operation represents the receipt of a value by the receiver. The system is then usually called a bounded FIFO (first-in-first-out) queue.

We will implement a FIFO with a shift register, consisting of an N-element array sr of cells. Each cell contains either a value (an argument/return-value) or a special value indicating that the cell is empty. A call puts a value in $sr[1]$ and a return removes the value from $sr[N]$. A value can be shifted to the next cell when that cell is empty. For example, here's a possible state in which the sequence of values that have been sent but not received is $\langle 13, 2, 5 \rangle$, where $\perp$ marks an empty cell.



The TLA$^+$ specification is straightforward, except for defining the value that represents an empty cell. We call the value *Empty* (TLA$^+$ does not let us use symbols like $\perp$ as identifiers), and we define it by:

$$Empty \triangleq \text{CHOOSE } v : v \notin Input$$

This defines *Empty* to be an arbitrary value that is not an element of the set *Input*. The next-state action is the disjunction of the following actions:

- $\exists v \in Input : SRCall(v)$
  Action $SRCall(v)$ does a call with argument v and puts v in $sr[1]$; it is enabled iff $sr[1]$ is empty.
- $\exists i \in 1..(N-1) : SRShift(i)$
  Action $SRShift(i)$ moves a value from $sr[i]$ to $sr[i+1]$; it is enabled iff $sr[i]$ is nonempty and $sr[i+1]$ is empty.
- *SRReturn*
  This action does a return with value $sr[N]$; it is enabled iff $sr[N]$ is nonempty.

The $SRShift(i)$ action sets the new value of $s[i]$ to *Empty* and the new value of $sr[i+1]$ to the old value of $sr[i]$. We could express this by

$$sr' = [[sr \text{ EXCEPT } ![i] = Empty] \text{ EXCEPT } ![i+1] = sr[i]]$$

Fortunately, TLA$^+$ allows us to abbreviate this as

$$sr' = [sr \text{ EXCEPT } ![i] = Empty, ![i+1] = sr[i]]$$

The rest of the specification is straightforward and appears in Module *ShiftRegister* of Figure 3.6 on the next page.

```
──────────────── MODULE ShiftRegister ────────────────
EXTENDS PCCallReturn, Naturals
──────────────────────────────────────────────────────
CONSTANT N
ASSUME (N ∈ Nat) ∧ (N > 0)

Empty ≜ CHOOSE v : v ∉ Input
──────────────────────────────────────────────────────
VARIABLE sr

SRInit ≜ ∧ CRInit
         ∧ sr = [i ∈ 1 .. N ↦ Empty]

SRTypeOK ≜ ∧ CRTypeOK
           ∧ sr ∈ [1 .. N → Input ∪ {Empty}]
──────────────────────────────────────────────────────
SRCall(v) ≜ ∧ sr[1] = Empty
            ∧ Call(v)
            ∧ sr' = [sr EXCEPT ![1] = v]

SRReturn ≜ ∧ sr[N] ≠ Empty
           ∧ Return(sr[N])
           ∧ sr' = [sr EXCEPT ![N] = Empty]

SRShift(i) ≜ ∧ sr[i]     ≠ Empty
             ∧ sr[i + 1] = Empty
             ∧ sr' = [sr EXCEPT ![i] = Empty, ![i + 1] = sr[i]]
             ∧ UNCHANGED iFace

SRNext ≜ ∨ ∃ v ∈ Input : SRCall(v)
         ∨ ∃ i ∈ 1 .. (N − 1) : SRShift(i)
         ∨ SRReturn
──────────────────────────────────────────────────────
SRSpec ≜ SRInit ∧ □[SRNext]_⟨sr, iFace⟩

THEOREM SRSpec ⇒ □SRTypeOK
──────────────────────────────────────────────────────
```

Figure 3.6: Specification of a shift register.

3.3.2 The Refinement Mapping

It should be obvious that *SRSpec* implements the internal producer-consumer specification *PCSpec* under the refinement mapping in which $\overline{q}$ is the sequence obtained by reversing the elements of *sr* and then removing the empty cells. However, you don't yet know enough TLA$^+$ to define $\overline{q}$ formally. We now remedy that deficiency.

The obvious way to define $\overline{q}$ is by induction. Figuring out what to induct on is less obvious. The trick is to define $F(i)$ inductively to be the result of reversing the elements and removing the empty cells from the first i cells of *sr*, and then to let $\overline{q}$ equal $F(N)$. A mathematician would write something like this, where $\circ$ is the concatenation operator on sequences, defined in the *Sequences* module.

$$\begin{aligned}
F(i) &\triangleq \textsc{if } i = 0 \textsc{ then } \langle\rangle \\
&\qquad\qquad \textsc{else if } sr[i] = Empty \textsc{ then } F(i-1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textsc{else } \langle sr[i]\rangle \circ F(i-1) \\
\overline{q} &\triangleq F(N)
\end{aligned}$$

TLA$^+$ does not allow you to write this kind of recursive operator definitions. However, you can write a recursive definition of a *function f*, using the syntax

$$f[id \in S] \;\triangleq\; exp$$

where S is the domain of function. Thus, we can change the definition above of the operator F into the following definition of the function F. Replacing $\overline{q}$ by the legal TLA$^+$ identifier *qBar*, we can rewrite the definition above as follows.

$$\begin{aligned}
F[i \in 0\,..\,N] &\triangleq \textsc{if } i = 0 \textsc{ then } \langle\rangle \\
&\qquad\qquad \textsc{else if } sr[i] = Empty \textsc{ then } F[i-1] \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textsc{else } \langle sr[i]\rangle \circ F[i-1] \\
qBar &\triangleq F[N]
\end{aligned}$$

The function F is of no interest in itself; it's defined only to be used in the definition of *qBar*. TLA$^+$ has a LET/IN construct that allows us to make a definition local to an expression. Using it, we can rewrite the definition of *qBar* as

$$\begin{aligned}
qBar \triangleq\; & \textsc{let } F[i \in 0\,..\,N] \triangleq \\
&\qquad\qquad \textsc{if } i = 0 \textsc{ then } \langle\rangle \\
&\qquad\qquad\qquad \textsc{else if } sr[i] = Empty \textsc{ then } F[i-1] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{else } \langle sr[i]\rangle \circ F[i-1] \\
&\quad \textsc{in } \; F[N]
\end{aligned}$$

We can use TLC to check that *SRSpec* implements *PCSpec* under this refinement mapping, when *RtnVal* is instantiated with the identity operator. However, TLC needs some help because it cannot evaluate the CHOOSE expression in the definition of *Empty*. We need to tell TLC what value to use instead of *Empty*. The trick is to tell TLC to use a *model value*—a special kind of value that is different from any other value around. We can tell TLC to use the model value `foo` for *Empty* by adding the following clause to the `CONSTANTS` statement in the configuration file:

```
Empty = foo
```

However, a better choice of model value than `foo` is the model value `Empty`, since that will make TLC's error messages more perspicuous. So, it's better to add the following clause to the `CONSTANTS` statement.

```
Empty = Empty
```

3.3.3 Liveness

We can specify the usual two flavors of producer-consumer liveness for the shift register. For client-server liveness, we need weak fairness not only of the *SRReturn* action, but also on the $SRShift(i)$ actions that are needed to push the values to the end of the register, from where they can be returned. Conjoining either of the following fairness conditions to *SRSpec* yields equivalent specifications of a shift register with client-server fairness.

$$\begin{array}{l} \mathrm{WF}_{\langle sr,\, iFace \rangle}(SRReturn) \;\wedge\; \mathrm{WF}_{\langle sr,\, iFace \rangle}(\exists i \in 1 \,..\, (N-1) : SRShift(i)) \\ \mathrm{WF}_{\langle sr,\, iFace \rangle}(SRReturn \vee (\exists i \in 1 \,..\, (N-1) : SRShift(i))) \end{array}$$

As with all our producer-consumer implementations, non-stop execution can be specified with either weak fairness of the next-state action or with the conjunction of weak fairness properties for each of its disjoints.

3.4 Processes Reconsidered

We have taken for granted that a producer-consumer system consists of two processes, a producer and a consumer. We will now examine that assumption more closely.

3.4.1 Producer-Consumer in N Processes

We start by examining the basic pattern of producer-consumer synchronization. For simplicity, let's assume a non-stop execution, so infinitely many call and return operations are executed. Let C_1, C_2, etc. be the executions of call operations, and let R_1, R_2, etc. be the executions of return operations. Let's form a directed graph from these operation executions by drawing an arc between operation execution A and operation execution B if the execution of A must precede the execution of B. To avoid clutter, we will eliminate arcs that are deducible from other arcs by transitivity. Here's the graph we get for the producer-consumer system when $N = 3$.

(3.2)
$$\begin{tikzcd}
C_1 \arrow[r] \arrow[d] & C_2 \arrow[r] \arrow[d] & C_3 \arrow[r] \arrow[d] & C_4 \arrow[r] \arrow[d] & C_5 \arrow[r] \arrow[d] & \cdots \\
R_1 \arrow[r] \arrow[urrr] & R_2 \arrow[r] \arrow[urrr] & R_3 \arrow[r] \arrow[urrr] & R_4 \arrow[r] \arrow[urr, no head] & R_5 \arrow[r] & \cdots
\end{tikzcd}$$

In general, the graph has the following arcs, for each i:

- Arcs from C_i to C_{i+1} and from R_i to R_{i+1}. Calls and returns are each issued sequentially.

- An arc from C_i to R_i. The i^{th} call must precede the i^{th} return.

- An arc from R_i to C_{i+N}. The producer/caller can get at most N calls ahead of the consumer/returner.

The two horizontal rows of operation executions in (3.2) represent the producer and consumer processes.

Let's now redraw the graph (3.2) in a different way, keeping the same nodes and arcs:

(3.3)
$$\begin{tikzcd}
C_1 \arrow[r] \arrow[d] & R_1 \arrow[r] \arrow[d] & C_4 \arrow[r] \arrow[d] & R_4 \arrow[r] \arrow[d] & C_7 \arrow[r] \arrow[d] & \cdots \\
C_2 \arrow[r] \arrow[d] & R_2 \arrow[r] \arrow[d] & C_5 \arrow[r] \arrow[d] & R_5 \arrow[r] \arrow[d] & C_8 \arrow[r] \arrow[d] & \cdots \\
C_3 \arrow[r] \arrow[uurr, bend right] & R_3 \arrow[r] \arrow[uurr, bend right] & C_6 \arrow[r] \arrow[uurr, bend right] & R_6 \arrow[r] \arrow[uurr, bend right, no head] & C_9 \arrow[r] & \cdots
\end{tikzcd}$$

The procedure we used to redraw the graph is to let the i^{th} row contain all the operations with numbers congruent to i modulo N. If we think of the semaphore program implementation, we see that each row contains the operations performed to a single buffer element. The i^{th} row contains the calls and returns whose values are written to $buf[i-1]$.

Comparing the two graphs, we see that thinking of producer-consumer synchronization as an N-process system is just as natural as thinking of it as a two-process system. An implementation with an array of buffers can just as easily be written with one process per buffer. Let's implement the producer-consumer specification *PCSpec* with an N-process semaphore program.

3.4.2 An *N*-Process Semaphore Program

We use the same array *buf* of buffers as before, and let process i read and write $buf[i]$, for $i \in 0\,..\,(N-1)$. Each process i alternately does a *call* operation, putting the argument in $buf[i]$, and then a return for that argument, putting the return value in $buf[i]$. Before doing the call, it must wait until process $(i-1)\,\%\,N$ has done its call; and before doing the return, it must wait until process $(i-1)\,\%\,N$ has done its return. In a semaphore program, waiting is done on semaphore P operations. We use two arrays of semaphores, *cSemA* and *rSemA*. Process i waits on a $P(cSemA[i])$ operation before performing a call, and it waits on a $P(rSemA[i])$ operation before performing a return. When process i performs a call or return operation, it does a V to the appropriate semaphore of process $(i+1)\,\%\,N$, enabling that process's next call or return. The program of process i is then:

```
while (true) { c: ⟪ P(cSemA[i]);
                    buf[i] = ?;
                    Call(buf[i]);
                    V(cSemA[(i+1)%N]) ⟫;
               r: ⟪ P(rSemA[i]);
                    buf[i] = RtnVal(buf[i]);
                    Return(buf[i]);
                    V(rSemA[(i+1)%N)] ⟫ }
```

The body of the `while` loop is broken into two separate actions. The labels mark the two control points. This program maintains the same grain of atomicity as the one in Figure 3.3 (page 54), where each step performs either a call or a return.

Observe that $cSemA[i]$ and $rSemA[i]$ will always equal either 0 or 1, for each $i \in 0\,..\,(N-1)$, so the variables *cSemA* and *rSemA* have type $[1\,..\,(N-1) \rightarrow 0\,..\,1]$. These variables also satisfy a stronger invariant—namely, that $cSemA[i]$ equals 1 for exactly one process i, and $cSemA[j]$ equals 1 for exactly one process j. To express this invariant concisely, we use the TLA$^+$ subset expression $\{i \in S : exp\}$, which is the subset of S containing all elements i that satisfy expression *exp*. The invariant then asserts that each of the following sets contains a single element:

$$\{i \in 1\,..\,(N-1) \,:\, cSemA[i] = 1\} \qquad \{i \in 1\,..\,(N-1) \,:\, rSemA[i] = 1\}$$

The standard module *FiniteSets* defines the *Cardinality* operator so that $Cardinality(S)$ is the number of elements of S, if S is a finite set. The invariant can then be written:

$$\begin{aligned}&\land\ Cardinality(\{i \in 0\,..\,(N-1) : cSemA[i] = 1\}) = 1\\ &\land\ Cardinality(\{i \in 0\,..\,(N-1) : rSemA[i] = 1\}) = 1\end{aligned}$$

Because we have an array of semaphores rather than just individual semaphore variables, we can't use the *Semaphore* module. We could define P and V operations for an array sA of semaphores, where $P(sA, i)$ and $V(sA, i)$ are the operations to $sA[i]$. They would be defined by

$$\begin{aligned} P(sA,\ i) &\triangleq (sA[i] > 0) \land (sA' = [sA \text{ EXCEPT } ![i] = sA[i] - 1]) \\ V(sA,\ i) &\triangleq sA' = [sA \text{ EXCEPT } ![i] = sA[i] + 1] \end{aligned}$$

However, each action of process i performs a P and a V operation to different elements of the array of semaphores, and with these definitions, the conjunction $P(sA,\ j) \land V(sA,\ k)$ equals FALSE for any j and k. So, we will just write the actions directly. For example, the simultaneous execution of `P(cSemA[i])` and `V(cSemA[(i+1)%N])` is represented by:

$$\begin{aligned} &\land\ cSemA[i] > 0 \\ &\land\ cSemA' = [cSemA \text{ EXCEPT } ![i] = cSemA[i] - 1, \\ &\phantom{\land\ cSemA' = [cSemA \text{ EXCEPT }} ![(i+1)\%N] = cSemA[(i+1)\%N] + 1] \end{aligned}$$

In addition to the variables buf, $cSemA$, and $rSemA$, we will need a variable pc to represent the control state. We let $pc[i]$ be the control state of process i, equal to either "c" or "r". We define the operator $GoTo$ so that, for example

$$GoTo(2, \text{"r"}, \text{"c"}) = (pc[2] = \text{"r"}) \land (pc' = [pc \text{ EXCEPT } ![2] = \text{"c"}])$$

The complete specification is in Module *NProcProducerConsumer*, in Figure 3.4.2 on the following two pages. As usual, we can conjoin weak fairness properties to obtain client-server or non-stop liveness.

3.4.3 Non-Determinism in Producer-Consumer

Alternation synchronization is deterministic; it consists of an alternating sequence of calls and returns. The only nondeterminism is in the choice of call values. When $N > 1$, producer-consumer synchronization allows nondeterminism. For example, after executing the first call, the system can then execute either the next call or the first return. The operation-execution graph, which we drew for $N = 3$ in (3.2) and (3.3), describes all possible executions. Any sequence of calls and returns obtainable by linearizing that graph—that is, by totally ordering the nodes so each arc points forwards—is a possible execution order.

Thinking of systems only in terms of behaviors (totally ordered sequences of states) can blind us to the fact that the operation execution graph of a producer-consumer system is completely determined (once N is chosen). The apparent nondeterminism comes because the graph determines a partial order, rather than a total order, of the operations. If we consider the operation-execution graph itself to represent an execution, then the nondeterminism disappears. If we don't

```
┌─────────────────── MODULE NProcProducerConsumer ───────────────────┐
EXTENDS PCCallReturn, FiniteSets, Naturals
├────────────────────────────────────────────────────────────────────┤
CONSTANT N
ASSUME (N ∈ Nat) ∧ (N > 0)
├────────────────────────────────────────────────────────────────────┤
VARIABLES buf, cSemA, rSemA, pc

NProcPCTypeOK ≜ ∧ CRTypeOK
                ∧ buf   ∈ [0 .. (N − 1) → Input ∪ Output]
                ∧ cSemA ∈ [0 .. (N − 1) → 0 .. 1]
                ∧ rSemA ∈ [0 .. (N − 1) → 0 .. 1]
                ∧ pc    ∈ [0 .. (N − 1) → {"c", "r"}]

NProcPCInvariant ≜ ∧ Cardinality({i ∈ 0 .. (N − 1) : cSemA[i] = 1}) = 1
                   ∧ Cardinality({i ∈ 0 .. (N − 1) : rSemA[i] = 1}) = 1

NProcPCInit ≜ ∧ CRInit
              ∧ buf   ∈ [0 .. (N − 1) → Output]
              ∧ cSemA = [i ∈ 0 .. (N − 1) ↦ IF i = 0 THEN 1 ELSE 0]
              ∧ rSemA = [i ∈ 0 .. (N − 1) ↦ IF i = 0 THEN 1 ELSE 0]
              ∧ pc    = [i ∈ 0 .. (N − 1) ↦ "c"]
├────────────────────────────────────────────────────────────────────┤
GoTo(i, loc1, loc2) ≜ ∧ pc[i] = loc1
                      ∧ pc′ = [pc EXCEPT ![i] = loc2]

NProcPCCall(i, v) ≜ ∧ GoTo(i, "c", "r")
                    ∧ cSemA[i] > 0
                    ∧ cSemA′ = [cSemA EXCEPT
                                   ![i]          = cSemA[i] − 1,
                                   ![(i + 1)%N] = cSemA[(i + 1)%N] + 1]
                    ∧ buf′ = [buf EXCEPT ![i]    = v]
                    ∧ Call(v)
                    ∧ UNCHANGED rSemA

NProcPCReturn(i) ≜ ∧ GoTo(i, "r", "c")
                   ∧ rSemA[i] > 0
                   ∧ rSemA′ = [rSemA EXCEPT
                                  ![i]          = rSemA[i] − 1,
                                  ![(i + 1)%N] = rSemA[(i + 1)%N] + 1]
                   ∧ buf′ = [buf EXCEPT ![i]    = RtnVal(buf[i])]
                   ∧ Return(buf′[i])
                   ∧ UNCHANGED cSemA
```

Figure 3.7a: The N-process producer-consumer semaphore program (beginning).

$$NProcPCNext \triangleq \exists\, i \in 0 \,..\, (N-1) : \begin{array}{l} \lor\ \exists\, v \in Input : NProcPCCall(i,\ v) \\ \lor\ NProcPCReturn(i) \end{array}$$

$$NProcPCSpec \triangleq NProcPCInit \land \Box[NProcPCNext]_{\langle buf,\, cSemA,\, rSemA,\, pc,\, iFace\rangle}$$

THEOREM $NProcPCSpec \Rightarrow \Box(NProcPCTypeOK \land NProcPCInvariant)$

Figure 3.7b: The N-process producer-consumer semaphore program (end).

require non-stop liveness, then we have some choice, because we can execute only an initial subgraph of the graph—that is, a subgraph that, if it contains a node, also contains all the nodes that precede it in the partial order. The fact that it has a uniquely determined operation-execution graph is an important characteristic of the producer-consumer problem—one that distinguishes it from other systems that we will study later.

There are good reasons why we view systems primarily in terms of behaviors. Experience has shown that this approach works best in practice for describing and reasoning about complex systems. However, viewing a system execution as an operation-execution graph can provide useful insight. We have seen how it can lead us to an N-process implementation of producer-consumer synchronization. We will use this way of viewing systems again.

3.5 The Grain of Atomicity

3.5.1 A Fine-Grained Semaphore Program

Let's now write a finer-grained version of the semaphore program in Figure 3.3 (page 54), this time representing the execution of each statement within the `while` loops as a separate step. This finer-grained program is shown in Figure 3.8 on the next page.

We can also write a TLA$^+$ specification $FGSemPCSpec$ of this program. We use the statement label as the name of the corresponding action. So, the next-state action is

$$FGSemPCNext \triangleq c1 \lor c2 \lor \ldots \lor r4 \lor r5$$

The definitions of actions $c1$, ..., $r5$ are straightforward. We introduce a variable pc and define the operators $CGoTo$ and $RGoTo$ as in module $AlternationPgm$ on page 26. We then have

$$c1 \triangleq \begin{array}{l} \land\ CGoTo(\text{“c1”},\ \text{“c2”}) \\ \land\ P(cSem) \\ \land\ \text{UNCHANGED}\ \langle buf,\ nxtC,\ nxtR,\ rSem,\ iFace\rangle \end{array}$$

```
    while (true) { c1:《 P(cSem) 》;
                   c2:《 buf[nxtC] = ? 》;
                   c3:《 Call(buf[nxtC]) 》;
                   c4:《 nxtC = (nxtC + 1)%N 》;
                   c5:《 V(rSem) 》; }
||  while (true) { r1:《 P(rSem) 》;
                   r2:《 buf[nxtR] = RtnVal(buf[nxtR]) 》;
                   r3:《 Return(buf[nxtR]) 》;
                   r4:《 nxtR = (nxtR + 1)%N 》;
                   r5:《 V(cSem) 》; }
```

Figure 3.8: A fine-grained semaphore producer-consumer program.

$$\begin{array}{lll} c2 & \triangleq & \land\ CGoTo(\text{“c2”}, \text{“c3”}) \\ & & \land\ \exists\, v \in Input : buf' = [buf \text{ EXCEPT } ![nxtC] = v] \\ & & \land\ \text{UNCHANGED } \langle nxtC, nxtR, cSem, rSem, iFace \rangle \end{array}$$

...

$$\begin{array}{lll} r5 & \triangleq & \land\ RGoTo(\text{“r5”}, \text{“r1”}) \\ & & \land\ V(cSem) \\ & & \land\ \text{UNCHANGED } \langle buf, nxtC, nxtR, rSem, iFace \rangle \end{array}$$

Although rather long-winded, this is a direct representation of the program of Figure 3.8.

Let's show that this fine-grained specification *FGSemPCSpec* implements the coarse-grained one *SemPCSpec* defined in module *SemProducerConsumer* on page 56. We must find state functions $\overline{buf}$, $\overline{nxtC}$, $\overline{nxtR}$, $\overline{cSem}$, and $\overline{rSem}$ to substitute for the variables *buf*, *nxtC*, *nxtR*, *cSem*, and *rSem* so that *FGSemPCSpec* implies $\overline{SemPCSpec}$. The refinement mapping will leave the interface variable *iFace* unchanged.

The trick to defining the refinement mapping is to define the barred variables so they are changed when *iFace* is—that is, they are changed by the actions $c3$ and $r3$ that perform the call and return. For example, *nxtC* is incremented by action $c4$, but we want $\overline{nxtC}$ to be incremented when $c3$ is executed. Thus, $\overline{nxtC}$ should equal $(nxtC+1)\,\%\,N$ after $c3$ is executed but before $c4$ is executed. Hence, we want to define:

$$\overline{nxtC} \triangleq \text{IF } pc.c = \text{“c4” THEN } (nxtC+1)\%N \text{ ELSE } nxtC$$

Similar reasoning leads to

$$\overline{nxtR} \triangleq \text{IF } pc.r = \text{“r4” THEN } (nxtR+1)\%N \text{ ELSE } nxtR$$

$$\overline{cSem} \triangleq cSem + (\text{IF } pc.c \in \{\text{“c2”}, \text{“c3”}\} \text{ THEN } 1 \text{ ELSE } 0)$$

$$\begin{array}{rcl} & & + (\textsc{if}\ pc.r \in \{\text{"r4"}, \text{"r5"}\}\ \textsc{then}\ 1\ \textsc{else}\ 0) \\ \overline{rSem} & \triangleq & rSem + (\textsc{if}\ pc.r \in \{\text{"r2"}, \text{"r3"}\}\ \textsc{then}\ 1\ \textsc{else}\ 0) \\ & & + (\textsc{if}\ pc.c \in \{\text{"c4"}, \text{"c5"}\}\ \textsc{then}\ 1\ \textsc{else}\ 0) \end{array}$$

However, we can't define $\overline{buf}$ in this way. For example, we want $\overline{buf}$ to be changed when $c3$ is executed, not when $c2$ is executed. Hence, after $c2$ is executed, we want $\overline{buf}[nxtC]$ to equal the value of $buf[nxtC]$ before $c2$ was executed. But that value has been overwritten.

The solution to this problem is to introduce an *auxiliary variable.* An auxiliary variable is one that is added to a specification without changing the behavior of the actual specification variables. Formally, adding an auxiliary variable *aux* to a specification *Spec* means defining a new specification *ASpec*, whose variables are *aux* together with the variables of *Spec*, such that hiding *aux* in *ASpec* yields a specification that is equivalent to *Spec*. In other words, *ASpec* is obtained by adding the auxiliary variable *aux* to *Spec* iff $\exists\, aux : ASpec$ is equivalent to *Spec*. In general, a formula $(\exists\, x : F)$ implies a formula G iff F implies G, assuming that x is not a free variable of G. Hence, *Spec* satisfies a property P in which variable *aux* does not occur iff *ASpec* satisfies P. So, if *ASpec* is obtained from *Spec* by adding an auxiliary variable, we can that *Spec* satisfies a property by proving that *ASpec* does.

The most common type of auxiliary variable, and the one we will use here, is a *history* variable. A history variable is an auxiliary variable whose value records information from the past. Suppose *Spec* has initial predicate *Init* and next-state action *Next*. The simplest way to add a history variable h to *Spec* is to define the following initial predicate and next-state action

$$HInit \triangleq \begin{array}{l} \land\ Init \\ \land\ h = exp \end{array} \qquad HNext \triangleq \begin{array}{l} \land\ Next \\ \land\ h' = Pexp \end{array}$$

where *exp* is any expression containing the (unprimed) variables of *Spec* and *Pexp* is any expression containing unprimed and/or primed variables of *Spec*. Let *vbl* be the tuple of variables of *Spec*. If h is not a variable of *Spec*, then the two formulas

$$Init \land \Box[Next]_{vbl} \qquad HInit \land \Box[HNext]_{\langle vbl, h \rangle}$$

describe the exactly the same behaviors, if we ignore the value of h. If *Spec* equals $Init \land \Box[Next]_{vbl} \land L$ for some liveness condition L, then $HInit \land \Box[HNext]_{\langle vbl, h \rangle} \land L$ is obtained by adding the history variable h to *Spec*.

We can now complete the definition of our refinement mapping by defining $\overline{buf}$ to equal the history variable *bufBar*, added to the specification *FGSemPCSpec* by defining the following initial predicate and next-state action.

$$HFGSemPCInit \triangleq \begin{array}{l} \land\ FGSemPCInit \\ \land\ bufBar = buf \end{array}$$

```
HFGSemPCNext ≜
  ∧ FGSemPCNext
  ∧ bufBar' =
        IF c3 THEN [bufBar EXCEPT ![nxtC] = buf[nxtC]]
              ELSE IF r3 THEN [bufBar EXCEPT ![nxtR] = buf[nxtR]]
                          ELSE bufBar
```

TLA$^+$ has a CASE construct that allows us to write the last conjunct in the definition of *HFGSemPCNext* more symmetrically as follows:

```
bufBar' = CASE c3     → [bufBar EXCEPT ![nxtC] = buf[nxtC]]
            □  r3     → [bufBar EXCEPT ![nxtR] = buf[nxtR]]
            □  OTHER  → bufBar
```

We often say that *FGSemPCSpec* implements *SemPCSpec* under this refinement mapping, even though it's actually *HFGSemPCSpec* that does.

With this refinement mapping, we have proved that *FGSemPCSpec* implies (implements) $\exists\, buf, nxtC, nxtR, cSem, rSem : SemPCSpec$. We have already shown that *SemPCSpec* implies $\exists\, q : PCSpec$. Since implication is transitive, this implies that *FGSemPCSpec* implies $\exists\, q : PCSpec$. Another way of saying this is that, since *FGSemPCSpec* implements *SemPCSpec* under a refinement mapping that leaves the variable *iFace* unchanged, then any property depending only on that variable that is satisfied by *SemPCSpec* is also satisfied by *FGSemPCSpec*.

3.5.2 Another Road to Refinement

We obtained the refinement mapping under which *FGSemPCSpec* implements *SemPCSpec* as follows. We defined the barred variables so that they were changed by the steps that performed the call and return. If we look at the barred variables (and *iFace*), rather than the original variables, then all the steps of *FGSemPCSpec* were stuttering steps, except for $c3$ and $r3$ steps. Those steps performed, on the barred variables, the effects of all the steps $c1$–$c5$ and $r1$–$r5$, respectively. We now describe how we could have seen that such a refinement mapping was possible without actually constructing it.

Commutativity of Actions

First, let's introduce a bit of notation. As before, we write $s \xrightarrow{\alpha} t$ to mean that $s \rightarrow t$ is an α step—in other words, step $s \rightarrow t$ satisfies action α. We define the composition $\alpha \cdot \beta$ of actions α and β to be the action obtained by performing α and then β. More precisely, $\alpha \cdot \beta$ is defined to be the action such that $s \xrightarrow{\alpha \cdot \beta} t$ holds iff there is a state u such that $s \xrightarrow{\alpha} u$ and $u \xrightarrow{\beta} t$ hold, for any states s and t. The actions of the fine-grained and coarse-grained semaphore producer-consumer programs are related by:

$$\begin{aligned} c1 \cdot c2 \cdot c3 \cdot c4 \cdot c5 &= \land\ \exists v \in \mathit{Input} : \mathit{SemPCCall}(v) \\ &\ \land\ (pc.c = \text{“}\mathtt{c1}\text{”}) \land (pc' = pc) \\ r1 \cdot r2 \cdot r3 \cdot r4 \cdot r5 &= \land\ \mathit{SemPCReturn} \\ &\ \land\ (pc.r = \text{“}\mathtt{r1}\text{”}) \land (pc' = pc) \end{aligned}$$

We say that an action α *right-commutes* with an action β, or that β *left-commutes* with α, iff $s \xrightarrow{\alpha\cdot\beta} t$ implies $s \xrightarrow{\beta\cdot\alpha} t$, for any states s and t. By the definition of action composition, this is true iff, for every triple of states s, t, and u such that $s \xrightarrow{\alpha} u \xrightarrow{\beta} t$, there is a state v such that $s \xrightarrow{\beta} v \xrightarrow{\alpha} t$. In other words, if we can get from state s to state t by first taking an α step and then a β step, then we can also get from s to t by first taking a β step and then an α step. We say that actions α and β *commute* if each right- and left-commutes with the other.

Constructing Pretend Variables

We want to pretend that the fine-grained program, described by *FGSemPCSpec*, is actually the simpler coarse-grained program, described by *SemPCSpec*. We do this by finding pretend versions of the coarse-grained program's variables such that, when we look at the values of the pretend variables in a behavior of the fine-grained program, they change the way the actual variables are changed by the coarse-grained program.

A behavior of the fine-grained program might look something like this

$$(3.4) \qquad \cdots\ s_{41} \xrightarrow{c1} s_{42} \xrightarrow{c2} s_{43} \xrightarrow{r1} s_{44} \xrightarrow{c3} s_{45} \xrightarrow{r2} s_{46} \xrightarrow{c4} s_{47} \xrightarrow{r3} s_{48} \xrightarrow{r4} s_{49} \xrightarrow{c5} \cdots$$

with steps of the two process's next-state action interleaved. We want to show that there exist pretend variables so that, viewing the pretend variables instead of the ordinary variables, behavior (3.4) that looks like this:

$$\begin{aligned}(3.5) \qquad \cdots\ & \widetilde{s_{41}} \xrightarrow{=} \widetilde{s_{42}} \xrightarrow{=} \widetilde{s_{43}} \xrightarrow{=} \widetilde{s_{44}} \xrightarrow{c1\cdot c2\cdot c3\cdot c4\cdot c5} \widetilde{s_{45}} \xrightarrow{=} \widetilde{s_{46}} \xrightarrow{=} \\ & \widetilde{s_{47}} \xrightarrow{r1\cdot r2\cdot r3\cdot r4\cdot r5} \widetilde{s_{48}} \xrightarrow{=} \widetilde{s_{49}} \xrightarrow{=} \cdots\end{aligned}$$

where $\widetilde{s_i}$ is state s_i, except with the pretend variables' values substituted for the actual variables' values, and $\xrightarrow{=}$ is a stuttering step. The actual behavior (3.4) yields a pretend behavior that, after removing uninteresting stuttering steps, looks like this:

$$\widetilde{s_{41}} \xrightarrow{c1\cdot c2\cdot c3\cdot c4\cdot c5} \widetilde{s_{46}} \xrightarrow{r1\cdot r2\cdot r3\cdot r4\cdot r5} \cdots$$

To get from (3.4) to (3.5), we will use commutativity, interchanging pairs of adjacent actions, to move all the caller steps adjacent to the $c3$ steps, and all

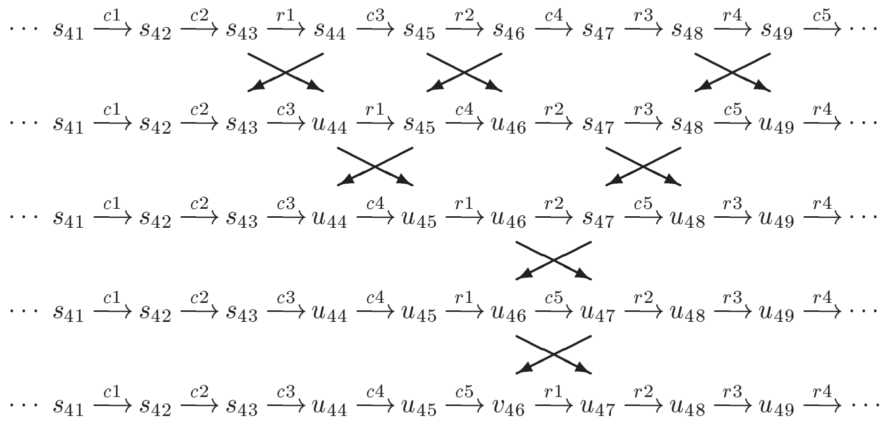


Figure 3.9: The transformation from (3.4) to (3.6).

the returner steps adjacent to the $r3$ steps. Thus, $c1$, $c2$, $r1$, and $r2$ steps are moved to the right, and $c4$, $c5$, $r4$, and $r5$ steps are moved to the left. For example, we assume for now that $r1$ right-commutes with $c3$, so we can interchange $r1$ and $c3$ in

$$s_{43} \xrightarrow{r1} s_{44} \xrightarrow{c3} s_{45}$$

to get

$$s_{43} \xrightarrow{c3} u_{44} \xrightarrow{r1} s_{45}$$

for some state u_{44}. Doing this by the sequence of transformations shown in Figure 3.9 on this page yields

$$\cdots\ s_{41} \xrightarrow{c1} s_{42} \xrightarrow{c2} s_{43} \xrightarrow{c3} u_{44} \xrightarrow{c4} u_{45} \xrightarrow{c5} v_{46} \xrightarrow{r1} u_{47} \xrightarrow{r2} u_{48} \xrightarrow{r3} u_{49} \xrightarrow{r4} \cdots \tag{3.6}$$

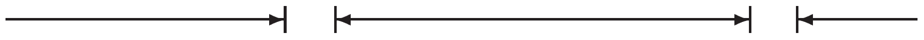
We will show later that the necessary commutativity relations do hold to justify this construction.

To get from (3.6) to (3.5), we single out those states in which neither process is in the middle of the coarse-grained operation:

$$\cdots\ \boxed{s_{41}} \xrightarrow{c1} s_{42} \xrightarrow{c2} s_{43} \xrightarrow{c3} u_{44} \xrightarrow{c4} u_{45} \xrightarrow{c5} \boxed{v_{46}} \xrightarrow{r1} u_{47} \xrightarrow{r2} u_{48} \xrightarrow{r3} u_{49} \xrightarrow{r4} \cdots$$

We let the pretend variables equal the regular variables in those states, and we determine their values in the other states by letting all steps except $c3$ and $r3$ steps leave the pretend variables unchanged, propagating their values as shown here:

$$\cdots\ \boxed{s_{41}} \xrightarrow{c1} s_{42} \xrightarrow{c2} s_{43} \xrightarrow{c3} u_{44} \xrightarrow{c4} u_{45} \xrightarrow{c5} \boxed{v_{46}} \xrightarrow{r1} u_{47} \xrightarrow{r2} u_{48} \xrightarrow{r3} u_{49} \xrightarrow{r4} \cdots$$



Assuming that the commutativity relations we assumed to do the interchanging illustrated in Figure 3.9 actually do hold, then it should be intuitively clear that pretend variables exist so that every behavior of the fine-grained program changes the pretend variables the way the coarse-grained program changes the actual variables. In other words, the fine-grained program implements the specification obtained from the coarse-grained program by substituting the pretend variables for the actual variables.

What is also true in this example is that the pretend *iFace* variable and the actual *iFace* variable are always equal. To see why this is the case, observe that only $c3$ and $r3$ steps change *iFace*, and we never interchanged those steps. Therefore, the sequences of values assumed by the actual variable *iFace* in the two behaviors ((3.4) and (3.6) are the same; they just change at different states. However, the $c3$ and $r3$ steps that change *iFace* are the only steps that change the pretend variables. So, the pretend version and actual versions of *iFace* have the same sequences of values and they change at the same time, so they are always equal.

In fact, the pretend variables one obtains by this procedure are precisely the barred variables we constructed explicitly in Section 3.5.1 above. Using commutativity relations, we showed that the refinement mapping exists without having to construct it.

Verifying the Commutativity Relations

We now have to show that the commutativity relations we needed for the construction shown in Figure 3.9 actually hold. Recall that we had to interchange caller steps and returner steps—for example, $c2$ and $r4$ steps. Since we never interchanged a $c3$ and an $r3$ step, there are 40 commutativity relations to check—each of the five actions of one process is interchanged with all but the third action of the other. Fortunately, there are only a few distinct cases to consider.

Observe that if two program statements access different parts of the state, then their corresponding actions commute. For example, program statement `c4` accesses only variable `nxtC` and the control state of the caller process, while `r4` accesses only `nxtR` and the control state of the returner process, so actions $c4$ and $r4$ commute. If executing $c4$ then $r4$ takes the system from state s to state t, then so does executing $r4$ then $c4$; and vice-versa. Similarly, if $nxtC \neq nxtR$, then statements `c3` and `r2` access different parts of the state, so actions $c3$ and $r2$ commute.

The only pairs actions that we interchanged whose corresponding statements can access a common part of the state are:

1. $c1$ and $r5$, the two operations on the semaphore *cSem*.

2. $r1$ and $c5$, the two operations on the semaphore *rSem*.

3. The pairs $\langle c2, r2 \rangle$, $\langle c2, r3 \rangle$, and $\langle c3, r2 \rangle$, when $nxtC = nxtR$.

We start with case (1). Because we moved caller steps adjacent to $c3$ steps and returner steps adjacent to $r3$ steps, we had to move $c1$ to the right of $r5$ steps. Thus, we have to show that $c1$, a $P(cSem)$ action, right-commutes with $r5$, a $V(cSem)$ action. Right commutativity of these actions means that, if starting in any state s, one reach a state t by doing a $P(cSem)$ step followed by a $V(cSem)$ step, then one can also reach t by doing a $V(cSem)$ step followed by a $P(cSem)$ step. This is obvious. Case (2) is the same as case (1).

Note that a $P(cSem)$ action does not left-commute with a $V(cSem)$ action if $cSem = 0$, because then a $V(cSem)$ followed by a $P(cSem)$ can be performed, but not vice-versa.

We now consider case (3). When interchanging any of the pairs $\langle c2, r2\rangle$, $\langle c2, r3\rangle$, and $\langle c3, r2\rangle$, we start from a state in which both processes are in the middle of their `while` body; that is, a state in which the caller is at statement `c2` or `c3` and the returner is at statement `r2` or `r3`. We never have to perform such an interchange when $nxtC = nxtR$ because the following state predicate is an invariant of the fine-grained program:

$$(3.7)\quad (pc.c \in \{\text{“c2”}, \text{“c3”}\}) \land (pc.r \in \{\text{“r2”}, \text{“r3”}\}) \;\Rightarrow\; (nxtC \neq nxtR)$$

Here's an informal argument that (3.7) is an invariant. When $nxtC = nxtR$, the buffer is either empty or full. If it's empty, then the caller cannot execute its `P(cSem)` statement; if it's full, then the returner cannot execute its `P(rSem)` statement. In either case, it's impossible for both processes to get past their `P` operation and for the caller to be at `c2` or `c3` and the returner to be at `r2` or `r3`. Later, we'll show how to reliably verify the correctness of an invariant. Meanwhile, you can use TLC to check the invariance of (3.7).

3.5.3 Extra-Fine Grained Atomicity

Our method of constructing the pretend variables for the fine-grained semaphore producer-consumer program works for an even finer-grained version. For example, suppose statement `r4` were split into two actions like this:

```
r4a:  《temp = (nxtR + 1) % N》;
r4b:  《nxtR = temp》
```

The TLA$^+$ specification would be modified in the obvious way, with the introduction of the new variable *temp* and the control point “`r4`” of the returner process replaced by the two control points “`r4a`” and “`r4b`”. Actions $r4a$ and $r4b$ would satisfy the same commutativity relations with the caller's actions as action $r4$ does—assuming that variable *temp* is not accessed by any of the caller's actions. Hence, we could construct pretend variables for this finer-grained program by exactly the same procedure.

We can see that this observation applies not just to representing `r4` by two actions, but by any number of actions. For example, if $nxtR$ is a 64-bit quantity, then we can set its left- and right-hand parts separately, as:

```
    r4a:  ⟪temp = (nxtR + 1) % N⟫;
    r4b:  ⟪nxtR = temp % 2^32⟫;
    r4c:  ⟪nxtR = temp⟫
```

Moreover, the same observation applies to all of the statements `c2`–`c4` and `r2`–`r4` by any number of actions. We can construct an extra-fine-grained specification by representing these operations with multiple actions. As long as the semaphore operations are atomic, we can construct the pretend variables as above.

For the fine-grained specification *FGSemPCSpec*, the pretend version and the actual version of the variable *iFace* were equal. This implied that any property satisfied by *SemPCSpec* that depends only on *iFace* is also satisfied by *FGSemPCSpec*. In particular, *FGSemPCSpec* implements (satisfies) *PCSpec*. However, if we represent an execution of statement `c3` and/or `r3` by multiple steps, so *iFace* is not changed atomically, then the pretend version of *iFace* will not necessarily equal the actual variable *iFace*. Hence, the extra-fine-grained specification does not necessarily implement *PCSpec*.

3.5.4 Reduction in General

We started with the coarse-grained specification *SemPCSpec*; we then wrote the fine-grained specification *FGSemPCSpec*; and finally, we described an extra-fine-grained specification. In real life, things often go the other way around. We start with an actual system, such as the program of Figure 3.4, and we must determine if that system does what we want it to. To understand the system, we must describe an execution of it as a behavior—a sequence of states. To do this, we must choose the grain of atomicity. It's clear that, by using a sufficiently fine-grained representation, we can accurately describe the system's execution. We know that a program is executed by individual computer operations that are executed atomically. However, at that level of granularity, execution of

```
    nxtR = (nxtR + 1) % N
```

may be represented by hundreds of steps. We want to find the coarsest-grained representation possible that is still sufficiently accurate for our purposes.

One way to do this is to start with a representation that we are confident is sufficiently fine-grained to be accurate enough for our purposes. We then try to show that we can obtain an "equivalent" specification by combining multiple actions $\alpha_1, \ldots, \alpha_n$ into their composition $\alpha_1 \cdot \cdots \cdot \alpha_n$. (We'll discuss later what "equivalent" means.) We do this by applying commutativity relations to show that, to every behavior of the system, there corresponds a behavior in which α_i steps are contiguous. For the semaphore producer-consumer program, we did this simultaneously for the α_i equal to $c_1, \ldots, c_5$ and equal to $r_1, \ldots, r_5$. We showed that, for any behavior like (3.4) of the fine-grained program, there is a corresponding behavior like (3.6). This implies that there is an "equivalent" behavior of the coarse-grained program—the one in which the n actions $\alpha_1, \ldots, \alpha_n$ are replaced by the single action $\alpha_1 \cdot \cdots \cdot \alpha_n$. (In the semaphore program example, this was behavior (3.5).)

A coarse-grained program obtained in this way is called a *reduced* version of the finer-grained one, and this procedure is called *reduction*. In most cases, the commutativity relations used by this procedure follow because the actions being commuted access different parts of the system state. For example, in the semaphore producer-consumer program, suppose we believe that storing a value into the program variable `nxtR` can be taken to be an atomic operation. That is, we believed that a finer-grained specification in which the actual assignment of the value to `nxtR` during the execution of

```
nxtR = (nxtR + 1) % N
```

is a single step is sufficiently accurate. We can then argue as follows that we can take the entire assignment statement to be an atomic operation. The actions that compute `(nxtR + 1) % N` only read the value of `nxtR`, and they commute with any action that does not change `nxtR`. Hence, they commute with all the actions of the caller process. Therefore, for any behavior of the finer-grained representation, there is a corresponding behavior in which all the steps in the computation of `(nxtR + 1) % N` occur immediately before the step that assigns the new value to `nxtR`. We can therefore reduce the finer-grained program to one in which executing the entire assignment state is represented as a single step. This same procedure allows us to reduce any finer-grained representation in which storing a value is an atomic action into the "equivalent" fine-grained specification *FGSemPCSpec*.

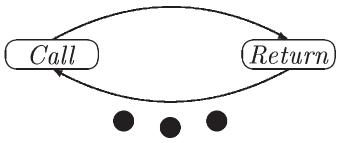
We now consider in what sense the original and the reduced system are equivalent. Let S be a fine-grained specification and let RS be the coarser-grained, specification obtained from S by reduction. The reduction process shows that we can define pretend variables such that S implements RS under the refinement mapping obtained by replacing each actual variable by its pretend version. Whether the reduced program is a satisfactory representation of the system depends on what the relationship is between the actual and the pretend variables. For the semaphore producer-consumer program, the important relation between *FGSemPCSpec* and its reduced version, *SemPCSpec*, is implied by the fact that the pretend and actual variables *iFace* are equal.

Recall the procedure by which we found the values of the pretend variables in an actual behavior of the system. We transformed the actual behavior into one in which the actions that we want to compose into a single action all occur in adjacent steps. The value of the pretend version of a variable in the i^{th} state of the actual behavior is the value of the actual variable in the i^{th} state of the transformed behavior. The transformed behavior is obtained from the actual one by commuting actions. We can therefore determine the relation between the values of the actual and the pretend variables when we understand how commuting actions changes the values of variables. In other words, when we replace a step $s \xrightarrow{\alpha} u \xrightarrow{\beta} t$ by a step $s \xrightarrow{\beta} v \xrightarrow{\alpha} t$, we must understand the relation between the values of the variables in u and in v. In general, the relation can be complicated. However, there is one important case in which it is simple. If neither α nor β change the value of a variable y, then the value of y is the same in states u and v. For example, in reducing *FGSemPCSpec* to *SemPCSpec*, none of the actions we commuted change the value of *iFace*, so the pretend value of *iFace* equals its actual value.

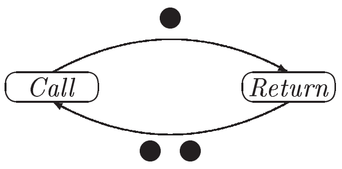
When we decide whether a representation of a system is fine-grained enough to be sufficiently accurate for our purposes, we are implicitly applying reduction to our informal understanding of how the real system behaves. Making the procedure explicit allows us to examine what we're doing more closely. This has two benefits: (i) it can catch errors, preventing us from using too coarse-grained a representation, and (ii) it can allow us to use a coarser-grained, and hence simpler, representation than we would have realized. The reduction of *FGSemPCSpec* to *SemPCSpec* is not intuitively obvious, and depends on the fact that a $P(s)$ action right-commutes with a $V(s)$ action—a fact that is easy to overlook, since the two actions don't commute.

3.6 Marked Graphs

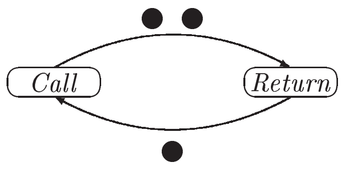
A producer-consumer system is one in which a behavior consists of a sequence of *Call* and *Return* steps, such that, at any point in the behavior, the number of *Call* steps is at least as great as the number of *Return* steps, and at most N greater than the number of *Return* steps. All such behaviors can be generated by a simple token game, which we illustrate for $N = 3$. The game starts with the following position:



A step of the game consists of "firing" a node by removing a token from its input edge, putting a token on its output edge of the node, and then adding the step indicated by the nodes' label to the behavior. The first step must consist of firing the *Call* node, which produces this configuration:



and starts the behavior with a *Call* step. The next step of the game can consist of firing either the *Call* node or the *Return* node. Firing the *Return* node produces the initial configuration; firing the *Call* node produces:



And so on. In general, behaviors of the producer-consumer system are generated by this game, starting with N tokens on the bottom arc.

We can generalize this game to an arbitrary directed graph, whose nodes are labeled with action names, having an arbitrary initial assignment of tokens to the arcs. Such a graph is called a *marked graph*, and an assignment of tokens to its arcs is called a *marking*. Execution of a marked graph consists of a sequence of steps, each firing a single node. A node can be fired iff there is at least one token on each of its input arcs. The node is fired by removing one token from each of its input arc and adding one token to each of its output arcs. This generates a set of possible sequences of steps.

Of greatest interest are strongly connected marked graphs. (A directed graph is strongly connected iff there is a path from any node to any other node.) It is easy to see that, throughout an execution of a marked graph. The number of tokens on any cycle remains constant. This implies that, for any strongly connected marked graph, there is a constant d such that the number of times any two nodes have been fired always differs by less than d.

Marked graph synchronization is the generalization of producer-consumer synchronization. As such, it is of theoretical interest. However, producer-consumer synchronization is the most important example of marked graph synchronization.

3.7 Problems

Problem 3.1 Write a stronger version of the non-stop producer-consumer system in which a call is issued infinitely often with each call value in *Input*. Use TLC to show that this implies the non-stop producer-consumer specification, for a particular choice of N and *Input*, but that the converse is not true if *Input* has more than one value.

Problem 3.2 Use TLC to show, for a particular choice of N and *Input*, that the two definitions of *INonstopPCSpec* in Section 3.1.1 are equivalent.

Note: We observed that the two definitions are not equivalent if *PCSpec* is modified to allow q to be of unbounded length. However, you can't use TLC to demonstrate this because TLC can check only systems that have (or are constrained to have) only a finite number of reachable states.

Problem 3.3 Explain why, for the semaphore producer-consumer program, weak fairness of the action $\exists v \in \mathit{Input} : \mathit{SemPCCall}(v)$ is equivalent to the conjunction of weak fairness of all $\mathit{SemPCCall}(v)$, for all v in Input. In other words, $\mathit{SemPCSpec}$ implies that these two formulas are equivalent:

$$\mathrm{WF}_{vbl}(\exists v \in \mathit{Input} : \mathit{SemPCCall}(v))$$
$$\forall v \in \mathit{Input} : \mathrm{WF}_{vbl}(\mathit{SemPCCall}(v))$$

where vbl is the tuple of all relevant variables. Use TLC to check this equivalence. Do the same for the equivalence of the two formulas

$$\mathrm{WF}_{\langle sr,\, iFace \rangle}(\exists i \in 1 \,..\, (N-1) : \mathit{SRShift}(i))$$
$$\forall i \in 1 \,..\, (N-1) : \mathrm{WF}_{\langle sr,\, iFace \rangle}(\mathit{SRShift}(i))$$

for the shift register specification.

Note: TLC Version 1 doesn't check equivalence of temporal formulas; to use it to check that F is equivalent to G, you must let it check that F implies G and that G implies F.

Problem 3.4 Write a busy-waiting version of the two-process producer-consumer program, without semaphores, using only the variables buf, $nxtC$, and $nxtR$. In the semaphore implementation, $nxtR = nxtC$ when the buffer is full and when it is empty. Your implementation will have to distinguish between these two cases. Do it in two ways: (a) letting buf have more than N elements (while still never allowing the producer to produce more than N elements than the consumer has consumed), and (b) letting $nxtC$ and $nxtR$ have values in $0 \,..\, (2N-1)$ (while still keeping only N elements in the buffer). In each case define the refinement mapping under which your specification implements PCSpec, and use TLC to check that it does. Use models on which TLC runs for less than 10 minutes.

Problem 3.5 Implement, in Java, one of the solution to Problem 3.4. Using a constant rate producer and a constant rate consumer that runs slower than the producer, measure the throughput, in buffers per second, that your program can sustain as a function of N. Choose a production rate that is something like a value every few milliseconds. Contrast this with the rate that you would get with a semaphore-based implementation of the two-process producer-consumer program. Explain the differences you find in the performances of the two solutions. Note that the behavior you see will depend on whether you run your solutions on a multiprocessor or a uniprocessor.

Problem 3.6 Sketch a proof that $\mathit{SemPCSpec}$ implements PCSpec under the refinement mapping (3.1) on page 57. Use TLC to check this for a small model. Also check this for the versions with client-server fairness and with non-stop fairness.

Problem 3.7 Figure 3.10 on the next page defines a specification of an implementation of the producer-consumer system in which q is implemented with a list rather than an array. Read and understand the specification. (The operator $\setminus$ is set difference, where $S \setminus T$ is the set of elements in S that are not in T.) Write the refinement mapping and the definition of N under which *ListPCSpec* implements *PCSpec*. Write the invariant needed to prove correctness of the refinement mapping. Use TLC to check your invariant and your refinement mapping. (Be sure to start running TLC with very small models.)

Problem 3.8 Show that specification *NProcPCSpec* of the N-process semaphore program, defined in module *NProcProducerConsumer* on pages 66–67, implements *PCSpec* under a suitable refinement mapping. Check your refinement mapping with TLC.

Also check that *NProcPCSpec* implements the two-process semaphore program specified by formula *SemPCSpec* under a suitable refinement mapping, and check this with TLC.

Problem 3.9 Write out the complete definition of the specifications *FGSemPCSpec* and *HFGSemPCSpec*, described in Section 3.5.1. Use TLC to check that *HFGSemPCSpec* implies *SemPCSpec* under the refinement mapping, and to check that (3.7) on page 74 is an invariant of *FGSemPCSpec*.

Problem 3.10 Write a specification of an arbitrary marked graph. (The graph and the initial token placement should be parameters.) Show that when instantiated with the marked graph that describes producer-consumer synchronization, this specification implements the specification *PCSpec*.

———— MODULE *ListProducerConsumer* ————

EXTENDS *PCCallReturn*

CONSTANT *CellAddress*

$$Cell \triangleq [val : Input,\ next : CellAddress]$$

VARIABLE *mem*, *freeCells*, *nxtC*, *nxtR*

$$\begin{aligned} ListPCTypeOK \triangleq\ & \land\ CRTypeOK \\ & \land\ mem \in [CellAddress \rightarrow Cell] \\ & \land\ nxtC \in CellAddress \\ & \land\ nxtR \in CellAddress \\ & \land\ freeCells \subseteq CellAddress \end{aligned}$$

$$\begin{aligned} ListPCInit \triangleq\ & \land\ CRInit \\ & \land\ mem \in [CellAddress \rightarrow Cell] \\ & \land\ nxtC \in CellAddress \\ & \land\ nxtR = nxtC \\ & \land\ freeCells = CellAddress \setminus \{nxtR\} \end{aligned}$$

$$\begin{aligned} ListPCCall(v) \triangleq\ & \exists\, fc \in freeCells : \\ & \quad \land\ Call(v) \\ & \quad \land\ nxtC' = fc \\ & \quad \land\ mem' = [mem \text{ EXCEPT } ![nxtC] = [val \mapsto v, \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad next \mapsto fc]] \\ & \quad \land\ freeCells' = freeCells \setminus \{fc\} \\ & \quad \land\ \text{UNCHANGED } nxtR \end{aligned}$$

$$\begin{aligned} ListPCReturn \triangleq\ & \land\ nxtR \neq nxtC \\ & \land\ Return(RtnVal(mem[nxtR].val)) \\ & \land\ nxtR' = mem[nxtR].next \\ & \land\ freeCells' = freeCells \cup \{nxtR\} \\ & \land\ \text{UNCHANGED } \langle nxtC,\ mem \rangle \end{aligned}$$

$$ListPCNext \triangleq (\exists\, v \in Input : ListPCCall(v)) \lor ListPCReturn$$

$$ListPCSpec \triangleq ListPCInit \land \Box[ListPCNext]_{\langle mem,\, freeCells,\, nxtC,\, nxtR,\, iFace \rangle}$$

THEOREM $ListPCSpec \Rightarrow \Box ListPCTypeOK$

Figure 3.10: A list-processing producer-consumer system.