Chapter 4

Multiclient-Server Systems

In this chapter, we extend the client-server paradigm to one in which multiple clients interact with a server. Logically, there is a single centralized server. However, we will see that this server is an abstraction that can be implemented in many ways—such as by a distributed "peer-to-peer" network of processors.

Before getting to the multiclient server, we introduce a client-server system with a single client that generalizes the alternation system of Chapter 2. We then describe a simple multiclient server and, finally, the general one.

4.1 The One-Client Server

We write the general specification of a one-client server system in Section 4.1.2 and describe an instance of it in Section 4.1.3. But first, we introduce a module *CSCallReturn* for describing the interaction between the clients and the servers.

4.1.1 Module *CSCallReturn*

Module *CSCallReturn* is almost the same as module module *PCCallReturn* (Figure 3.2, page 52), which describes the communication between a producer and a consumer. The only difference is that *CSCallReturn* does not have the operator parameter *RtnVal*, and it declares *Output* to be a parameter instead of defining it. The module's parameters are therefore:

Input The set of all possible *Call* arguments.

Output The set of all possible *Return* values.

iFace A variable describing the client-server communication interface.

```
┌─────────────────────── MODULE CSCallReturn ───────────────────────┐
EXTENDS Naturals
CONSTANTS Input, Output
├───────────────────────────────────────────────────────────────────┤
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 4.1: Module *CSCallReturn*.

We will use module *CSCallReturn* by instantiating it, substituting suitable sets for the parameters *Input* and *Output*

The module defines the predicates *CRInit* and *CRTypeOK* that describe the initial value and type of *iFace*, and it defines the actions $Call(v)$ and $Return(v)$. For a single-client system, these actions are:;

$Call(v)$ A client call with argument v.

$Return(v)$ A system return with value v.

For a multiclient system, the argument of *Call* and *Return* will also identify the client. The complete module is in Figure 4.1 on this page.

4.1.2 The General Specification

In the alternation system of Chapter 2, the client issues a request and the server replies with a response that is a function of the request. Our one-client server system generalizes the alternation system by allowing the server's response to be a function of the client's previous requests.

To describe the possible dependence of the server's behavior on previous requests, we let the server have a state. We describe the server's response and its new state as functions of the client's request and the server's current state. Our specification has operator parameters *ResponseVal* and *NewState* such that:

$ResponseVal(v, s)$ is the server's response to a client request v when the server's state is s.

$NewState(v, s)$ is the server's new state after responding to a client request v when its current state is s.

We have three other constant parameters:

Request The set of possible client requests.

State The set of possible server states.

InitialState The server's initial state.

There are two relations that must hold among these parameters:

- *InitialState* must be an element of *State*.
- $NewState(v, s)$ must be an element of *State*, for every v in *Request* and s in *State*.

We make these relations assumptions of our specification. We define the set *Response* of all possible server responses to be the set of all values $ResponseVal(v, s)$ with v in *Request* and s in *State*:

$$Response \triangleq \{ResponseVal(v, s) : v \in Request, s \in State\}$$

Later, we give an example of how this general specification is instantiated to give a particular example of a one-client server system. But now, let's write a module *OneClientServer* that specifies the general one-client server system. As before, we begin by writing a module *IOneClientServer* containing the internal specification.

We instantiate the *CSCallReturn* module with *Request* substituted for *Input* and *Response* substituted for *Output*. The other parameter of *CSCallReturn* is the variable *iFace*. We let it be instantiated by a variable of the same name, which we declare in module *IOneClientServer*. We can then instantiate *CSCallReturn* with the statement:

$$\text{INSTANCE } CSCallReturn \text{ WITH } Input \leftarrow Request, Output \leftarrow Response$$

To write our specification, we have to choose the state variables and decide what the individual steps (state changes) are. We introduce a variable *sstate* to describe the server's state. We will also need a variable to record whether it's the client's turn to issue a request or the server's turn to respond to the previous request. We call that variable *cstate*. These are internal variables of the system; the only visible variable is *iFace*, which describes the communication between the client and the server.

We have to decide when the system's state changes—should it be when the client issues its call or when the server issues its response? Since the variable *sstate*, which describes the system's state is internal, it makes no difference when it changes. It could be changed either by the calling action or the responding action. It could even be changed by a separate action that occurs between the call and the response. All that we are specifying are the possible sequences of values of *iFace*.

We will choose the third option, letting *sstate* be changed by a separate action. The interaction between the client and server can then be in any of the following three states, which we name from the client's point of view:

idle The client can issue a request.

calling The client has issued a request, but the server's state has not yet changed.

returning The server's state has changed, but the server has not yet issued a response to the client's request.

We let *cstate* be a record with a *ctl* field that specifies the state of the client-server interaction, and with a *val* field that records the calling argument when in the *calling* state and the response value when in the *returning* state. The type invariant for the variables *sstate* and *cstate* is then:

$$\begin{array}{l} \land\ sstate \in State \\ \land\ cstate \in [ctl : \{\text{“idle”}, \text{“calling”}, \text{“returning”}\},\ val : Request \cup Response] \end{array}$$

We can make the type invariant more precise by saying that the *cstate.val* is a request when *cstate.ctl* equals “calling” and is a response when *cstate.ctl* equals “returning”. It's convenient to let the *Respond* action not change *cstate.val*, so *cstate.val* is a response when *cstate.ctl* equals “idle”. We can then add the following conjuncts to the type invariant:

$$\begin{array}{l} \land\ (cstate.ctl \in \{\text{“calling”}\}) \Rightarrow (cstate.val \in Request) \\ \land\ (cstate.ctl \in \{\text{“returning”}, \text{“idle”}\}) \Rightarrow (cstate.val \in Response) \end{array}$$

Initially, the system's state *sstate* equals *InitialState*, *cstate.ctl* equals “idle”, *cstate.val* can be any response, and the value of *iFace* is specified by the state predicate *CRInit* from the *CSCallReturn* module. The initial predicate is therefore:

$$\begin{array}{l} \land\ sstate = InitialState \\ \land\ \exists\, v \in Response : cstate = [ctl \mapsto \text{“idle”},\ val \mapsto v] \\ \land\ CRInit \end{array}$$

Defining the next-state action is straightforward. It is the disjunction of three actions:

$\exists req \in Request : OCSIssueRequest(req)$ where $OCSIssueRequest(req)$ describes the client's issuing of request *req*.

$OCSDo$ that changes *sstate*, saving the response value in *cstate.val*.

OCSRespond that describes the issuing of the response by the server.

For liveness, we want to ensure that the server responds to every client request. Once the *OCSRespond* or *OCSNext* action becomes enabled, it remains enabled until it is "executed". Therefore, the desired liveness property is ensured by weak fairness of $OCSRespond \lor OCSNext$.

The complete internal specification is in Module *IOneClientServer* in Figure 4.1.2 on the following two pages. For convenience in writing subscripts, we define *ocsvars* to be the tuple of all the specification's variables. Formula *IOCSSpec* is the internal specification of the one-client server, with the variables *sstate* and *cstate* visible. The actual specification *OCSSpec*, with these variables hidden, is defined by

```
---------------------- MODULE OneClientServer ----------------------
CONSTANTS Request, State, InitialState, ResponseVal(_, _),
          NewState(_, _)

VARIABLE iFace

Inner(sstate, cstate) ≜ INSTANCE IOneClientServer
OCSSpec ≜ ∃ sstate, cstate : Inner(sstate, cstate)!IOCSSpec
--------------------------------------------------------------------
```

We could generalize the one-client server system to be nondeterministic. It could have a set of possible starting states, and it could allow nondeterminism in the choice of response and new state. We have deliberately made the server deterministic—the sequence of client requests determines the sequence of responses and server states.

4.1.3 A Register

As a simple example of a one-client server system, we specify a register whose value can be read and written by the client. We write the specification by defining actual values for the constant parameters of module *OneClientServer*, declaring its variables, and then instantiating that module. This substitutes, for each parameter of *OneClientServer*, the symbol of the same name.

Our specification is in module *OneClientRegister* in Figure 4.3 on page 89. It first declares the parameter *RegisterVal*, which is the set of possible register values. It next defines the set *Request* of requests. There are two types of requests, *read* and *write*. An elements of *Requests* is record with a *type* field; a write also has a *val* field, which equals the value being written. The set *State* of states is equal to *RegisterVal*. The initial state *InitialState* is defined to be an arbitrary element of *RegisterVal*.[1]

[1] Observe that we define *InitialState* to be some particular, arbitrarily chosen element of

MODULE *IOneClientServer*

CONSTANTS $Request,\ State,\ InitialState,\ ResponseVal(\_,\ \_),\ NewState(\_,\ \_)$

$$\begin{array}{ll}\textsc{assume} & \land\ InitialState \in State \\ & \land\ \forall\, v \in Request,\ s \in State : NewState(v,\ s) \in State\end{array}$$

$$Response \triangleq \{ResponseVal(v,\ s) : v \in Request,\ s \in State\}$$

VARIABLES $sstate,\ cstate,\ iFace$

INSTANCE $CSCallReturn$ WITH $Input \leftarrow Request,\ Output \leftarrow Response$

$$\begin{array}{lll}OCSInit & \triangleq & \land\ sstate = InitialState \\ & & \land\ \exists\, v \in Response : cstate = [ctl \mapsto \text{“idle”},\ val \mapsto v] \\ & & \land\ CRInit\end{array}$$

$$\begin{array}{l}OCSTypeInvariant \triangleq \\ \quad \land\ sstate \in State \\ \quad \land\ cstate \in [ctl : \{\text{“idle”},\ \text{“calling”},\ \text{“returning”}\},\ val : Request \cup Response] \\ \quad \land\ (cstate.ctl \in \{\text{“calling”}\}) \Rightarrow (cstate.val \in Request) \\ \quad \land\ (cstate.ctl \in \{\text{“returning”},\ \text{“idle”}\}) \Rightarrow (cstate.val \in Response) \\ \quad \land\ CRTypeOK\end{array}$$

$$\begin{array}{lll}OCSIssueRequest(req) & \triangleq & \land\ cstate.ctl = \text{“idle”} \\ & & \land\ Call(req) \\ & & \land\ cstate' = [ctl \mapsto \text{“calling”},\ val \mapsto req] \\ & & \land\ \textsc{unchanged}\ sstate\end{array}$$

$$\begin{array}{lll}OCSDo & \triangleq & \land\ cstate.ctl = \text{“calling”} \\ & & \land\ sstate' = NewState(cstate.val,\ sstate) \\ & & \land\ cstate' = [ctl \mapsto \text{“returning”}, \\ & & \qquad\qquad\quad val \mapsto ResponseVal(cstate.val,\ sstate)] \\ & & \land\ \textsc{unchanged}\ iFace\end{array}$$

$$\begin{array}{lll}OCSRespond & \triangleq & \land\ cstate.ctl = \text{“returning”} \\ & & \land\ Return(cstate.val) \\ & & \land\ cstate' = [cstate\ \textsc{except}\ !.ctl = \text{“idle”}] \\ & & \land\ \textsc{unchanged}\ sstate\end{array}$$

$$\begin{array}{lll}OCSNext & \triangleq & \lor\ \exists\, req \in Request : OCSIssueRequest(req) \\ & & \lor\ OCSDo \\ & & \lor\ OCSRespond\end{array}$$

$$ocsvars \triangleq \langle cstate,\ sstate,\ iFace\rangle$$

Figure 4.2a: Specification of a one-client server system (beginning).

$$\begin{aligned} IOCSSpec \triangleq\ & \land\ OCSInit \\ & \land\ \Box[OCSNext]_{ocsvars} \\ & \land\ \mathrm{WF}_{ocsvars}(OCSDo \lor OCSRespond) \end{aligned}$$

Figure 4.2b: Specification of a one-client server system (end).

The operator *ResponseVal* is defined so that the response to a read request is the current value of the register (the current state); the response to a write request is the string "OK". The operator *NewState* is defined so a read request leaves the register unchanged; a write request sets the register to the request's value. The INSTANCE statement then includes all the definitions from module *OneClientServer*, with the parameters instantiated by the defined variables and parameters of the same name from module *OneClientRegister*. Thus, it defines *OCSSpec* to be the internal specification of the one-client register system.

We can't use TLC to check this specification of the one-client register because it cannot handle the $\boldsymbol{\exists}$ operator in the definition of *OCSSpec* in module *OneClientServer*. To use TLC, we have to define the internal specification of the one-client register, obtained by instantiating the internal specification *IOCSSpec* of the one-client server. We do this by writing a module *IOneClientRegister* that is the same as the *OneClientRegister* module except that it instantiates *IOneClientServer* instead of *OneClientServer*. Since *IOneClientServer* has the additional variables *sstate* and *cstate* (which appear in module *OneClientServer*

RegisterVal. The server is completely deterministic; in every behavior allowed by the specification, its initial state has the same value—namely, the value CHOOSE $v \in RegisterVal$: TRUE.

MODULE *OneClientRegister*

CONSTANT *RegisterVal*

$Request \triangleq [type : \{\text{"read"}\}] \cup [type : \{\text{"write"}\},\ val : RegisterVal]$

$State \triangleq RegisterVal$

$InitialState \triangleq \text{CHOOSE } v \in RegisterVal : \text{TRUE}$

$$\begin{aligned} ResponseVal(req, s) &\triangleq \text{IF } req.type = \text{"read"} \text{ THEN } s \\ &\phantom{\triangleq \text{IF } req.type = \text{"read"}} \text{ ELSE } \text{"OK"} \\ NewState(req, s) &\triangleq \text{IF } req.type = \text{"read"} \text{ THEN } s \\ &\phantom{\triangleq \text{IF } req.type = \text{"read"}} \text{ ELSE } req.val \end{aligned}$$

VARIABLE *iFace*
INSTANCE *OneClientServer*

Figure 4.3: A one-client register.

as bound variables, not as parameters of the module), those variables must be declared in module *IOneClientRegister*. We therefore have to replace the VARIABLE statement in module *OneClientRegister* with

$$\text{VARIABLES } \mathit{iFace},\ \mathit{sstate},\ \mathit{cstate}$$

The specification in this module *IOneClientRegister* is perfectly correct, but TLC may not handle it properly. In particular, it may report an error if asked to check the invariant *OCSTypeInvariant*. It is instructive to understand why. The problem comes when TLC tries to check the conjunct

$$\begin{aligned}\mathit{cstate} \in [&\mathit{ctl} : \{\text{“idle”}, \text{“calling”}, \text{“returning”}\},\\ &\mathit{val} : \mathit{Request} \cup \mathit{Response}]\end{aligned}$$

which requires checking

$$\mathit{cstate.val} \in \mathit{Request} \cup \mathit{Response}$$

TLC does this by enumerating the elements in $\mathit{Request} \cup \mathit{Response}$ and seeing if it finds one that equals $\mathit{cstate.val}$. Suppose that

- $\mathit{cstate.val}$ equals “OK”, which it will immediately after processing a write request, and
- TLC enumerates $\mathit{Request} \cup \mathit{Response}$ starting with an element $[\mathit{type} \mapsto \text{“write”}, \mathit{val} \mapsto v]$ for some v.

TLC must then decide if the string “OK” equals the record $[\mathit{type} \mapsto \text{“write”}, \mathit{val} \mapsto v]$. Are those two values equal? Our first impulse would be to say that they aren't equal. But how do we know? The semantics of TLA$^+$ is based on the idea that it would be a bad idea to base the correctness of a specification on the assumption that two such disparate values as the string “OK” and the record $[\mathit{type} \mapsto \text{“write”}, \mathit{val} \mapsto v]$ are unequal. So, the semantics of TLA$^+$ does not specify whether or not those two values are equal, and therefore TLC can't decide if they are. TLC therefore reports an error.

The invariant actually is satisfied because “OK” is an element of $\mathit{Request} \cup \mathit{Response}$, so it doesn't matter whether or not “OK” equals $[\mathit{type} \mapsto \text{“write”}, \mathit{val} \mapsto v]$. However, TLC will give up and report an error without examining the remaining elements of $\mathit{Request} \cup \mathit{Response}$.

In general, when writing specifications, you should not assume that values that look different are different. Different values that intuitively have the same type are different—for example, $1 \neq 2$ and “OK” $\neq$ “NotOK”. (However, we don't know whether or not “OK” equals 1.) Other examples of different values are:

- Records with different sets of components are unequal, so $[\mathit{foo} \mapsto 1] \neq [\mathit{bar} \mapsto \text{“OK”}]$.

- Sets with different numbers of elements are unequal, so $\{1, 2\} \neq \{\text{“OK”}\}$.
- Tuples (sequences) with different numbers of elements are unequal, so $\langle 1, 2\rangle \neq \langle \text{“OK”} \rangle$.
- Functions with unequal domains are unequal, so $[i \in 1 \,..\, 2 \mapsto \text{“OK”}] \neq [i \in 1 \,..\, 3 \mapsto 1]$.

TLC also assumes that a model value is different from any other kind of value.

TLC’s model values are described on pages 61–62.

An easy way to solve this problem is to modify the specification in module *IOneClientRegiser* by replacing the string “OK” with a new value *OK*. We could either let *OK* be a parameter, or else define it by

$$OK \;\triangleq\; \text{CHOOSE } x : x \notin Request$$

We could then use TLC to check the specification by replacing *OK* with the model value `OK`. As explained on pages 61–62, we do this by adding

```
OK = OK
```

to the configuration file’s `CONSTANTS` statement.

If you do this, you will find that there is one other case in which $Request \cup Response$ will cause TLC to have the same problem: *ResponseVal* will evaluate to an element of *RegisterVal* if the operation type is a read. If *RegisterVal* is a natural number, then TLC will attempt to form a set containing naturals and records. In this case, we can avoid this problem by having *ResponseVal* evaluate to a record when the operation type is a read.c

4.2 A Simple Multiclient Server

We now generalize the one-client server to a simple multiclient server. The generalization consists of simply allowing multiple clients to issue requests. To generalize module *OneClientServer*, we make the following basic changes:

- We introduce a parameter *Client*, the set of clients. You should think of *Client* as the set of all possible clients. A system where the set of current clients can change is represented by recording in the system state whether an element of *Client* is a current client or merely a possible client.
- We let *cstate* be a function with domain *Client*, where $cstate[c]$ is the state of client c.
- A request argument or a response value is a pair $\langle c, v\rangle$, where c is the client issuing the call or receiving the return, and v is the argument or return value.

These changes imply all the other changes that must be made—for example, each action is now parameterized by the client that issues the command. To ensure that a response is generated for every client's request, we need a weak fairness condition for each client. Writing the specification is straightforward. The internal specification *ISMCSpec* appears in module *ISimpleMultiClientServer* in Figure 4.2 on the following two pages. We define the actual specification *SMCSpec* to equal $\exists\, sstate, cstate : ISMCSpec$ in a separate *SimpleMultiClientServer* module with the usual TLA$^+$ incantation.

Our decision to let the server's state be changed by a separate internal action (one that changes only internal variables) between the request and the response did not affect the one-client server specification. It has important consequences for the multiclient server specification, because it means that the order in which the requests are actually done—that is, the response value computed and the state changed—is not tied to the order in which the requests or the responses are issued. For example, suppose the following four events occur in order:

1. Client c_1 issues a request.
2. Client c_2 issues a request.
3. The server responds to c_1's request.
4. The server responds to c_2's request.

It is possible for c_1's request to be done *after* c_2's request. Only if the server had responded to c_1's request before client c_2 had issued its request can we be sure that c_1's request must be done before c_2's.

As an example of a simple multiclient server, we specify a toy banking system. The clients are the bank customers, and the state of the bank describes the amount of money in each client's account. A client can perform any of three operations: deposit money, withdraw money, or transfer money to the account of another client. The response to a request simply indicates whether or not the request is legal—a withdrawal or transfer being legal iff there is enough money in the account. To avoid the problem that prevents TLC from handling the *IOneClientRegister* specification, discussed on page 90, we let a response be a record whose *val* field is either "OK" or "No", depending on whether or not the request is legal. The complete specification is in Figure 4.5 on page 95. (For simplicity, we allow requests that specify an amount of $0.)

Let's now use TLC to check the specification of the banking system. Since TLC cannot handle the hiding operator $\exists$ in the simple multiclient server specification *SMCSpec*, we must define an internal specification of the banking system. We do that with a module *IBankingSystem* that is the same as module *BankingSystem*, except that declares the internal variables *sstate* and *cstate* and instantiates *ISimpleMultiClientServer* instead of *SimpleMultiClientServer*.

When we try to let TLC check the instantiated specification *ISMCSpec*, we are immediately faced by the problem that this is not a finite-state system,

```
┌──────────────── MODULE ISimpleMultiClientServer ─────────────────┐
CONSTANTS Client, Request, State, InitialState,
          NewState(_, _, _), ResponseVal(_, _, _)

ASSUME  ∧ InitialState ∈ State
        ∧ ∀ c ∈ Client, v ∈ Request, s ∈ State : NewState(c, v, s) ∈ State

Response ≜ {ResponseVal(c, v, s) : c ∈ Client, v ∈ Request, s ∈ State}
├──────────────────────────────────────────────────────────────────┤
VARIABLES sstate, cstate, iFace

INSTANCE CSCallReturn WITH Input  ← Client × Request,
                           Output ← Client × Response
├──────────────────────────────────────────────────────────────────┤
SMCInit ≜  ∧ sstate = InitialState
           ∧ ∃ v ∈ Response : cstate = [c ∈ Client ↦ [ctl ↦ "idle", val ↦ v]]
           ∧ CRInit

SMCTypeInvariant ≜
  ∧ sstate ∈ State
  ∧ cstate ∈ [Client → [ctl : {"idle", "calling", "returning"},
                        val : Request ∪ Response]]
  ∧ ∀ c ∈ Client :
      ∧ (cstate[c].ctl ∈ {"calling"}) ⇒ (cstate[c].val ∈ Request)
      ∧ (cstate[c].ctl ∈ {"returning", "idle"}) ⇒
           (cstate[c].val ∈ Response)
├──────────────────────────────────────────────────────────────────┤
SMCIssueRequest(c, req) ≜
  ∧ cstate[c].ctl = "idle"
  ∧ Call(⟨c, req⟩)
  ∧ cstate′ = [cstate EXCEPT ![c] = [ctl ↦ "calling", val ↦ req]]
  ∧ UNCHANGED sstate

SMCDo(c) ≜
  ∧    cstate[c].ctl = "calling"
  ∧    sstate′ = NewState(c, cstate[c].val, sstate)
  ∧    cstate′ = [cstate EXCEPT
                   ![c] = [ctl ↦ "returning",
                           val ↦ ResponseVal(c, cstate[c].val, sstate)]]
  ∧    UNCHANGED iFace
```

Figure 4.4a: Specification of a simple multiclient server (beginning).

$$SMCRespond(c) \triangleq \begin{array}[t]{l} \land\ cstate[c].ctl = \text{“returning”} \\ \land\ Return(\langle c,\ cstate[c].val \rangle) \\ \land\ cstate' = [cstate\ \text{EXCEPT}\ ![c].ctl = \text{“idle”}] \\ \land\ \text{UNCHANGED}\ sstate \end{array}$$

$$SMCNext \triangleq \exists\, c \in Client : \begin{array}[t]{l} \lor\ \exists\, req \in Request : SMCIssueRequest(c,\ req) \\ \lor\ SMCDo(c) \\ \lor\ SMCRespond(c) \end{array}$$

$$ocsvars \triangleq \langle cstate,\ sstate,\ iFace \rangle$$

$$ISMCSpec \triangleq \begin{array}[t]{l} \land\ SMCInit \\ \land\ \Box[SMCNext]_{ocsvars} \\ \land\ \forall\, c \in Client : \mathrm{WF}_{ocsvars}(SMCDo(c) \lor SMCRespond(c)) \end{array}$$

Figure 4.4b: Specification of a simple multiclient server (end).

since there is no bound on the amount of money in a client's account. High-level system specifications often do have unbounded states—for example, unbounded message queues. We could easily rewrite this specification to be finite-state, adding a parameter *MaxBalance* that is the upper bound on the balance of a client's account, making a request illegal if it would cause a balance to exceed that amount. However, our goal in using TLC is to check a specification without changing it.

We will have to make some modifications to the specification. Instead of modifying module *IBankingSystem*, we write a new module *MCBankingSystem* that extends *IBankingSystem* and describes the modifications. First, we must tell TLC to bound the set of states that it checks. We do this by introducing a *constraint*, which is a state predicate. TLC will stop exploring a behavior when it reaches a state that doesn't satisfy the constraint. We could tell TLC to restrict its exploration to states in which each client's balance is less than $3 by giving it the constraint

$$\forall c \in Client\ :\ sstate[c] \leq 3$$

Instead of fixing the value 3, we use a parameter *MaxBalance* so we can change the size of the model by modifying only the configuration file.

TLC still can't handle the specification because the next-state action contains a disjunct of the form $\exists req \in Request : \ldots$, and TLC can handle quantification only over finite sets. So, we tell it to replace *Request* with the set *MCRequest* of requests whose *amt* field is at most equal to a parameter *MaxTransaction*.

If we try running TLC on the specification with just this change, it will complain when it tries to evaluate the set *Response* of responses. This set is

```
┌──────────────────────── MODULE BankingSystem ────────────────────────┐
EXTENDS Naturals

CONSTANT Client

Request ≜ [type : {"deposit", "withdraw"}, amt : Nat]
              ∪
          [type : {"transfer"}, dest : Client, amt : Nat]

State ≜ [Client → Nat]

InitialState ≜ [c ∈ Client ↦ 0]

NewState(c, req, st) ≜
  CASE req.type = "deposit"   → [st EXCEPT ![c] = st[c] + req.amt]
    □ req.type    = "withdraw" →
        IF req.amt ≤ st[c] THEN [st EXCEPT ![c] = st[c] − req.amt]
                           ELSE st
    □ req.type = "transfer" →
        IF req.amt ≤ st[c]
          THEN [st EXCEPT ![c] = st[c] − req.amt,
                          ![req.dest] = st[req.dest] + req.amt]
          ELSE st

ResponseVal(c, req, st) ≜
  [type ↦ "response", val ↦ IF   ∨ req.type = "deposit"
                                 ∨ req.amt ≤ st[c]
                               THEN "OK"
                               ELSE "No"]

VARIABLES iFace
INSTANCE SimpleMultiClientServer
└──────────────────────────────────────────────────────────────────────┘
```

Figure 4.5: Specification of a simple banking system.

defined in module *SimpleMultiClientServer* to equal

$$\{ResponseVal(c,\ v,\ s) \ : \ c \in Client,\ v \in Request,\ s \in State\}$$

and TLC would have to perform an infinite computation to evaluate it because *State* is an infinite set. (We have told TLC to restrict which states it examines, but the set *State* is still infinite.) So, we tell TLC to replace *State* with a finite set *MCState*. TLC examines all states reachable in one step from a state satisfying the constraint. So, it reach can a state in which a client's balance is as large as $MaxBalance + MaxTransaction$. To prevent TLC from finding a violation in the the type invariant, we define *MCState* to be the set of states in which each

```
──────────────── MODULE MCBankingSystem ────────────────
EXTENDS IBankingSystem
CONSTANTS MaxBalance, MaxTransaction

MCConstraint ≜ ∀ c ∈ Client : sstate[c] ≤ MaxBalance

MCState      ≜ [Client → 0 .. (MaxBalance + MaxTransaction)]
MCRequest    ≜ [type : {"deposit", "withdraw"}, amt : 0 .. MaxTransaction]
                 ∪
               [type : {"transfer"}, dest : Client, amt : 0 .. MaxTransaction]

MCLiveness ≜
   ∀ c ∈ Client : (cstate[c].ctl = "calling") ⇝ (cstate[c].ctl = "idle")
────────────────────────────────────────────────────────
```

```
CONSTANTS  MaxBalance = 3
           MaxTransaction = 2
           Client = {c1,c2}
           State <- MCState
           Request <- MCRequest
SPECIFICATION ISMCSpec
INVARIANTS SMCTypeInvariant
PROPERTY MCLiveness
CONSTRAINT MCConstraint
```

Figure 4.6: A module and configuration file for checking the banking system specification.

client's balance is at most *MaxBalance* + *MaxTransaction*.

These definitions are contained in module *MCBankingSystem*, which appears along with its configuration file in Figure 4.6 on this page. It also includes a simple liveness property for TLC to check.

4.3 A Multithreaded Implementation

How would we implement the simple multiclient server as a multithreaded program? One answer is to implement the server and the clients as separate threads. However, the next-state action *SMCNext* of the specification *ISMCSpec* has the form $\exists c \in Client : \ldots$. This suggests an implementation with just one thread per client, and no separate server thread. We implement $cstate[c].ctl$ with the program control state of thread c; we implement $cstate[c].val$ as `val[c]`, for an array variable `val`; and we let *sstate* be a program variable read and written by all the threads.

To simplify the translation from TLA$^+$, we let the statement `local v = exp` introduce a new local variable `v` and assign it the value `exp`. The scope of `v` ends at the next unmatched } or ⟫. We can then write the program for client c as:

```
while (true) {
     idle: ⟪ local req = ?;
             Call(c, req);
             val[c] = req ⟫;
  calling: ⟪ local ss = sstate;
             sstate = NewState(c, val[c], ss);
             val[c] = ResponseVal(c, val[c], ss) ⟫;
returning: ⟪ Return(c, val[c]) ⟫ }
```

We need a new programming language construct to express the parallel composition of these client threads. We let the statement

```
||(i in j..k) { P(i) }
```

concurrently execute copies of the program $P(n)$, for values of i ranging from j through k, as separate threads. If we let there be N clients numbered from 1 through N, we can then write the multithreaded version of the simple multiclient server as

```
||(c in 1..N) { ... }
```

where the "`...`" is the program given above for client `c`.

The multithreaded program given above has the same grain of atomicity as the specification *SMCSpec*. To get a realistic implementation, we need to refine the grain of atomicity to one that is easy to implement in a real program.

The atomic `idle` and `returning` statements mention only variables that are local to the individual client. We can get an "equivalent" finer-grained program by breaking these statements into smaller atomic substatements. Each of these substatements represents an atomic subaction that commutes with every action of every other client.[2] Hence, by interchanging executions of commuting actions—the way we did in Section 3.5.2—we can transform any behavior of the finer-grained program into an equivalent one of the coarser-grained program.

From now on, we will let a statement not enclosed in ⟪ ⟫ brackets mean that we are leaving its grain of atomicity unspecified. We replace the program above with this finer-grained one:

[2] We could destroy that commutativity by going out of our way to use shared variables—for example, if we were to use a single global variable `req` instead of a separate local variable for each client. We assume that an implementation doesn't introduce such gratuitous variable sharing.

```
  || (c in 1..N) {
       while (true) {
            idle: { local req = ?;
                    Call(c, req);
                    val[c] = req };
         calling: ⟪ local ss = sstate;
                    sstate = NewState(c, val[c], ss);
                    val[c] = ResponseVal(c, val[c], ss) ⟫;
       returning: Return(c, val[c]) } }
```

If we want our program to implement specification *SMCSpec*, we have to make the `Call` and `Return` operations atomic. Since how this is ensured depends on the actual architecture, we won't worry about that here.

This program still has the atomic `calling` statement. That statement reads and writes the variable `sstate`, which is a shared variable—meaning that is accessed by more than one thread. If the `calling` statement were broken into subactions, those subactions would not commute with actions of other clients. Indeed, interleaving the subactions from the execution of two clients' `calling` statements could produce incorrect results. For example, suppose we broke the statement into these three actions:

```
  calling: { c1: local ⟪ss = sstate⟫;
             c2: ⟪sstate = NewState(c, val[c], ss)⟫ ;
             c3: ⟪val[c] = ResponseVal(c, val[c], ss)⟫ }
```

Suppose client 1 executed $c1$, client 2 then executed $c1$ and $c2$, and then client 1 executed $c2$. All effects of client 2's operation on *sstate* are erased by client 1's $c2$ action. Hence, client 2 thinks it has performed an operation, but that operation is not reflected in the system's state. It's clear that this behavior doesn't satisfy specification *SMCSpec* for many choices of the *NewState* operator—for example, the one used in the banking system.

So, we face a dilemma. Our program is incorrect if we simply split statement `calling` into subactions; but the statement is too complex to be executed atomically by a real computer. Let's examine more closely why we can decompose the `idle` and `responding` statements but not the `calling` statement. To transform a behavior of the finer-grained program to an a behavior equivalent to one of the coarser-grained program, we have to repeatedly interchange actions of two different threads to group together all the substeps of a single statement by a single client. This requires that those pairs of actions of different threads commute. If we split the `calling` statement into subactions; we are prevented from doing this because subactions of the `calling` statement from two different clients do not commute.

It would solve our problem if we could transform a behavior of the finer-grained program to group together all substeps of any single operation without interchanging subactions of the `calling` statement. This would be the case if no subaction of another client's `calling` statement could be executed between the execution of two substatements of any client's `calling` statement. We could in turn achieve this if we could prevent control in two different clients from being at or within their `calling` statements at the same time.

So, we can obtain a fine-grained implementation, in which the `calling` statement is executed by an arbitrary sequence of atomic statements, by guaranteeing that no two threads are executing their `calling` statement concurrently. A portion of code that cannot be executed concurrently by two different threads is called a *critical section*. The problem of guaranteeing mutually exclusive access to a critical section is called the *mutual exclusion problem*. We now study that problem.

4.3.1 The Mutual Exclusion Problem

Implementing mutual exclusion in a multithreaded system is the most studied topic in the theory of concurrent systems. We begin our examination of it by stating it precisely, in an abstract form. We let each thread have four states:

noncs The thread does not now want to execute its critical section.

waiting The thread is waiting to enter its critical section.

cs The thread is in its critical section.

exiting The thread has finished its critical section and is exiting the synchronization code.

A thread goes through the states in the indicated order; when it leaves the *exiting* state, it enters the *noncs* state.

If our only requirement were that no two threads are ever in their critical sections at the same time, then we could implement mutual exclusion by having the threads take turns executing their critical sections. With n threads, we would first have thread 0 execute its critical section, then thread 1, . . . , then thread $n - 1$, then thread 0, and so on. This would be an example of marked-graph synchronization—the graph consisting of a cycle of n nodes (node i representing the execution of thread i's critical section) with a single token. Such a solution would require a thread to keep entering its critical section even if it had no reason to do so. Thus, when implementing a multiclient server, a client would have to participate in the mutual exclusion algorithm even when it had no request to execute.

We are interested only in mutual exclusion algorithms in which a thread that does not want to enter its critical section need not do anything. If only one thread ever wants to enter its critical section, then only that thread need ever perform any action. This form of synchronization cannot be described by a marked graph. At any point during the execution of a connected marked graph, any node can have fired only a bounded number of times more than any other node. (If there is a cycle with k tokens that contains the two nodes, then each node can have fired at most k times more than the other.) Mutual exclusion synchronization cannot be represented by a marked graph in which each thread's entering its critical section is represented by firing a separate node, since one thread could enter its critical section an unbounded number of times while another never enters its own critical section.

Writing a TLA$^+$ specification of the safety part of the mutual exclusion specification is straightforward. We let *Thread* be the set of threads and let *tstate* be a variable such that $tstate[t]$ is the state of thread t. A thread t can perform three actions:

$METry(t)$ Go from the *noncs* to the *waiting* state.

$MEEnterCS(t)$ Go from the *waiting* to the *cs* state. This action is enabled only when no thread is in the *cs* state.

$MEExitCS(t)$ Go from the *cs* to the *exiting* state.

$MEFinish(t)$ Go from the *exiting* to the *noncs* state.

Formally defining these actions is easy.

The first liveness requirement we make is that a thread can always exit its critical section and reach its noncritical section. That is, once a thread t has entered the *exiting* state, it eventually reaches the *noncs* state. This is asserted by weak fairness of the $MEExitCS(t)$ action.

There are a number of different liveness conditions that we can require about when a thread must enter its critical section. Two of the simplest and most common are:

- Each waiting thread must eventually enter its critical section. This condition is known as *starvation freedom*.

- Some waiting thread must eventually enter its critical section (but any particular waiting thread might wait forever). This condition is known as *deadlock freedom*.

As stated here, both of these conditions assume that a thread that is in its critical section eventually leaves its critical section. This assumption is most naturally regarded as a requirement on how the threads use a mutual exclusion algorithm, not on the algorithm itself. So, we restate these two liveness conditions so they imply the statements above if no thread stays in its critical section forever.

To restate the conditions, we use the observation that $MEEnterCS(t)$ is enabled iff thread t is waiting to enter its critical section and no thread is in its critical section.

For deadlock freedom, it suffices to require that if no thread is in its critical section and some thread is waiting to enter its critical section, then some thread eventually will enter its critical section. No thread is in its critical section and some thread is waiting iff the action $\exists t \in \mathit{Thread} : \mathit{MEEnterCS}(t)$ is enabled. Some thread then eventually enters its critical section iff this action is executed. Hence, deadlock freedom is expressed by weak fairness of this action.

For starvation freedom, it suffices to require that, any waiting thread t cannot be continually able to enter its critical section without eventually doing so. A waiting thread t is able to enter its critical section iff $\mathit{MEEnterCS}(t)$ is enabled. So, we require that, for any t, action $\mathit{MEEnterCS}(t)$ cannot be continually enabled without an $\mathit{MEEnterCS}(t)$ step eventually happening. *Continually* enabled means enabled in infinitely many states; it allows the possibility that the action could also be disabled in infinitely many states. This condition cannot be expressed with weak fairness of any action. To express it, we introduce the notion of *strong fairness*. For any action A, we define strong fairness of A to hold for a behavior iff any one of the following equivalent conditions holds:

1. If A is ever enabled infinitely often, then an A step must eventually occur.
2. If A is enabled infinitely often, then infinitely many A steps must occur.
3. Either A is eventually never enabled, or infinitely many A steps occurs.

Infinitely often means *in infinitely many states.*

As with weak fairness, many people find it hard to see that these three characterizations are all equivalent. If you are among them, review the discussion of weak fairness on page 34.

Similarly to what we did for weak fairness, we define $\mathrm{SF}_v(A)$ to be the formula that is true of a behavior iff any of the three conditions above hold, except with A replaced by the action $A \wedge (v' \neq v)$. (See Section 2.5.4 on page 36.) In the informal discussion of strong fairness, we usually ignore the subscripts.

We can now express starvation freedom for a mutual exclusion algorithm as strong fairness of $\mathit{MEEnterCS}(t)$, for all threads t. The complete specification of mutual exclusion is in module *MutualExclusion* of Figure 4.7 on the next page. We have defined two versions of the specification, *MEDeadlockFree* and *MEStarvationFree*, with the two liveness conditions. We also defined the invariant *MSMutexInvariant* that expresses mutual exclusion.

4.3.2 Implementing Mutual Exclusion

A multithreaded implementation of mutual exclusion consists of two procedures, `Enter` and `Exit`. A thread calls the `Enter` procedure before executing its critical section, and it calls `Exit` afterwards. It's traditional to describe mutual exclusion algorithms with the procedures `Enter` and `Exit` written as in-line code:

MODULE *MutualExclusion*

CONSTANT *Thread*
VARIABLE *tstate*

$$METypeOK \triangleq tstate \in [Thread \rightarrow \{\text{"noncs"}, \text{"waiting"}, \text{"cs"}, \text{"exiting"}\}]$$

$$\begin{aligned}&MEMutexInvariant \triangleq \\ &\quad \forall\, t1, t2 \in Thread : (tstate[t1] = \text{"cs"}) \land (tstate[t2] = \text{"cs"}) \Rightarrow (t1 = t2)\end{aligned}$$

$$MEInit \triangleq tstate = [t \in Thread \mapsto \text{"noncs"}]$$

$$\begin{aligned}METry(t) \triangleq\ &\land\, tstate[t] = \text{"noncs"} \\ &\land\, tstate' = [tstate \text{ EXCEPT } ![t] = \text{"waiting"}]\end{aligned}$$

$$\begin{aligned}MEEnterCS(t) \triangleq\ &\land\, tstate[t] = \text{"waiting"} \\ &\land\, \forall\, tt \in Thread : tstate[tt] \neq \text{"cs"} \\ &\land\, tstate' = [tstate \text{ EXCEPT } ![t] = \text{"cs"}]\end{aligned}$$

$$\begin{aligned}MEExitCS(t) \triangleq\ &\land\, tstate[t] = \text{"cs"} \\ &\land\, tstate' = [tstate \text{ EXCEPT } ![t] = \text{"exiting"}]\end{aligned}$$

$$\begin{aligned}MEFinish(t) \triangleq\ &\land\, tstate[t] = \text{"exiting"} \\ &\land\, tstate' = [tstate \text{ EXCEPT } ![t] = \text{"noncs"}]\end{aligned}$$

$$\begin{aligned}MENext \triangleq\ &\exists\, t \in Thread : \\ &\quad METry(t) \lor MEEnterCS(t) \lor MEExitCS(t) \lor MEFinish(t)\end{aligned}$$

$$\begin{aligned}&MEDeadlockFree \triangleq \\ &\quad MEInit \land \Box[MENext]_{tstate} \land (\forall\, t \in Thread : \mathrm{WF}_{tstate}(MEFinish(t))) \\ &\qquad \land \mathrm{WF}_{tstate}(\exists\, t \in Thread : MEEnterCS(t)) \\ &MEStarvationFree \triangleq \\ &\quad MEInit \land \Box[MENext]_{tstate} \land (\forall\, t \in Thread : \mathrm{WF}_{tstate}(MEFinish(t))) \\ &\qquad \land (\forall\, t \in Thread : \mathrm{SF}_{tstate}(MEEnterCS(t)))\end{aligned}$$

Figure 4.7: The specification of mutual exclusion

```
while (true) {
  noncritical section; Enter; critical section; Exit };
```

An implementation of mutual exclusion may not have an explicit *tstate* variable. Instead, *tstate*[*t*] is usually defined in terms of the control state of thread t. In other words, the algorithm usually don't implement specification *MEDeadlockFree* or *MEStarvationFree* of module *MutualExclusion*. Instead, it implements it under a refinement mapping.

Until now, implementation has meant implication. If a specification *Spec* equals $\exists\, x : \mathit{ISpec}$, then we proved that a specification *Impl* implements *Spec* by showing that *Impl* implements *ISpec* under a refinement mapping that doesn't change the visible variables of *Spec*. We are now introducing a new use of the term *implementation* saying that *Impl* implements *Spec* under a refinement mapping that can change the visible variables of *Spec*. Such a refinement mapping is sometimes called an *interface refinement*, because it changes the interface (visible) variables of the specification. The term *implementation* is used informally with both meanings—implication and implementation under an interface refinement. It's important to be aware of which term is being used when someone talks about implementing a specification.

When a mutual exclusion algorithm is described in terms of `Enter` and `Exit` procedures, the interface refinement (refinement mapping) under which it implements one of the specifications in module *MutualExclusion* is defined so that:

- The call of the `Enter` procedure implements the $\mathit{METry}(t)$ action.
- The return from `Enter` implements the $\mathit{MEEnterCS}(t)$ action.
- The call of `Exit` implements the $\mathit{MEExitCS}(t)$ action.
- The return from `Exit` implements the $\mathit{MEFinish}(t)$ action.

When the algorithm is described with in-line `Enter` and `Exit` procedures, the action that brings control in thread t to the beginning of `Enter` implements $\mathit{METry}(t)$, the action that brings control to the beginning of the critical section implements $\mathit{MEEnterCS}(t)$, and so on.

Using Semaphores

It's trivial to implement mutual exclusion using a semaphore *sem*. `Enter` is ⟪`P(sem)`⟫ and `Exit` is ⟪`V(sem)`⟫, so each thread's program is

```
while (true) {
  noncritical section;  ⟪P(sem)⟫ ;  critical section;  ⟪V(sem)⟫  };
```

If `sem` is initially equal to 1 and is not modified in the noncritical or critical section, then after one thread has executed its ⟪`P(sem)`⟫ operation, no other thread can execute ⟪`P(sem)`⟫ until the first thread has executed its ⟪`V(sem)`⟫. Hence, at most one thread can be in its critical section at a time, so this implements mutual exclusion.

What flavor of mutual exclusion this algorithm implements depends on the liveness property required of a thread's $P(\mathit{sem})$ action. The weakest requirement generally made is that, if a nonempty set of threads is waiting to perform a P operation on a semaphore whose value is positive, then at least one of them will eventually succeed. It is expressed by weak fairness of each thread's P

action, or equivalently, by weak fairness of the disjunction of all threads' P actions. A semaphore that satisfies only this requirement is called a *weak* or *unfair* semaphore. With a weak semaphore, the simple semaphore implementation of mutual exclusion is deadlock free.

A stronger requirement on a semaphore is that it be *fair*. A fair semaphore guarantees that a thread waiting to perform a P operation must eventually do so if it is infinitely often able to. Fairness of a semaphore is expressed by strong fairness of each thread's P action. The simple semaphore implementation of mutual exclusion is starvation free if the semaphore is fair.

The Simple Atomic Bakery Algorithm

Implementing mutual exclusion with a semaphore is a perfectly reasonable thing to do when programming in a language like Java. However, it is not intellectually very satisfying because it begs the question of how the semaphore itself is implemented. Moreover, we sometimes have to implement mutual exclusion when synchronization primitives like semaphores are not available. A problem that has challenged computer scientists from the earliest days of multithreaded programming is how to implement mutual exclusion using only read and write operations. Many solutions have been offered, a number of them inspired by the requirements of particular systems. Here we describe just one, called the bakery algorithm. For reasons that will be explained in Chapter 5, it is a particularly interesting solution. In this section, we give a simplified version of it we call the atomic bakery algorithm.

We begin with a very highly simplified version that works as follows. Each thread maintains a number that equals 0 when the thread is in its noncritical section. To enter its critical section, the thread first sets its number to be one greater than the largest of all the other threads' numbers. It then enters its critical section when it has the smallest non-negative number. To exit its critical section, the thread sets its number back to 0.

Let the threads be numbered from 1 through N, and let $num[t]$ be the value of thread t's number. The program of thread t in this simplified algorithm can be written as:

```
while (true) {
        noncritical section;
 enter: ⟪num[t] = 1 + Maximum(num[1], ... , num[N])⟫ ;
        || (i in 1..N) { if (i != t) {
              e_i: ⟪await((num[i]=0) or (num[t]<num[i]))⟫ } };
        critical section;
        ⟪num[t] = 0⟫ }
```

The parallel composition operator indicates that the `await` operations can be performed in any order, or concurrently by multiple subthreads. We use the informal "`...`" in the `Maximum` expression rather than defining a language construct for the purpose.

A TLA$^+$ specification of this algorithm is formula *SBSpec* of module *SimpleBakery* in Figure 4.3.2 on the following two pages. The program state of thread t is described by $tstate[t]$, except that, when control is in the "`||`" statement, substatement $\mathtt{e_i}$ has yet to be executed iff i is an element of $waitingFor[t]$.

This specification introduces a useful little feature of TLA$^+$: the @ construct in an EXCEPT clause. In the expression

$$[waitingFor \text{ EXCEPT } ![t] = @ \setminus \{i\}]$$

the @ stands for $waitingFor[t]$. If we think of this expression as constructing a "new version" of *waitingFor* by replacing the value of $waitingFor[t]$, then the @ stands for the value from the old version that's being replaced. Similarly, in the construct $[r \text{ EXCEPT } !.c = exp]$, an @ in expression *exp* stands for $r.c$.

Correctness of the Simple Atomic Bakery Algorithm

We have used the variable *tstate* to describe the control state in specification of module *SBSpec* so it assumes the same values as in the specifications of the *MutualExclusion* module. So, we expect it to actually implement the mutual exclusion specification, without the need for a refinement mapping.

The simple atomic bakery algorithm is starvation free. Proving correctness of it means proving that *SBSpec* implies *MEStarvationFree*—that is, proving the formula $SBSpec \Rightarrow MEStarvationFree$. We can use TLC to debug the algorithm, but it can't verify its correctness. If threads continually try to enter their critical section, the values of $num[t]$ can become arbitrarily large. So, TLC cannot explore all possible executions. So, let's prove that the algorithm is correct.

One of the nice things about mathematics is that we can write proofs with extreme rigor—even going down, if we wish, to the level of detail where the proof is reduced to mindless symbol pushing. When proofs are properly structured, the more detailed the proof, the more confidence we have in the correctness of what we're trying to prove. (Without proper structuring, adding more detail can make the proof harder to follow and not reduce the chance of making an error.) We want to find the degree of rigor that provides a reasonable level of confidence in the proof with the smallest amount of effort.

We begin by proving safety—that is, proving *SBSpec* implies $MEInit \land \Box[MENext]_{tstate}$. The safety part of *MEStarvationFree* essentially asserts two things: (i) that the value of $tstate[t]$ for each thread t cycles through the values "noncs", "waiting", "cs", and "exiting", and (ii) two different threads t cannot both have $tstate[t] =$ "cs". It's clear that our algorithm satisfies the first property. The only interesting part of the proof is proving the second property, which

```
┌─────────────────── MODULE SimpleBakery ───────────────────┐
EXTENDS Naturals

CONSTANT N
ASSUME N > 1

Thread ≜ 1 . . N

VARIABLE num, tstate, waitingFor

vars ≜ ⟨num, tstate, waitingFor⟩
├───────────────────────────────────────────────────────────┤
SBInit ≜ ∧ num = [t ∈ Thread ↦ 0]
         ∧ tstate = [t ∈ Thread ↦ "noncs"]
         ∧ waitingFor = [t ∈ Thread ↦ {}]

SBTypeOK ≜ ∧ num ∈ [Thread → Nat]
           ∧ tstate ∈ [Thread → {"noncs", "waiting", "cs", "exiting"}]
           ∧ waitingFor ∈ [Thread → SUBSET Thread]
├───────────────────────────────────────────────────────────┤
GoTo(t, loc1, loc2) ≜ ∧ tstate[t] = loc1
                      ∧ tstate′ = [tstate EXCEPT ![t] = loc2]

SBSetNum(t) ≜
  LET max[i ∈ Thread] ≜ IF i = 1 THEN num[1]
                                 ELSE IF num[i] > max[i − 1] THEN num[i]
                                                             ELSE max[i − 1]
      maxNum ≜ max[N]
  IN  ∧ GoTo(t, "noncs", "waiting")
      ∧ num′ = [num EXCEPT ![t] = 1 + maxNum]
      ∧ waitingFor′ = [waitingFor EXCEPT ![t] = Thread \ {t}]

SBWaitFor(t, i) ≜
  ∧ tstate[t] = "waiting"
  ∧ i ∈ waitingFor[t]
  ∧ (num[i] = 0) ∨ (num[t] < num[i])
  ∧ waitingFor′ = [waitingFor EXCEPT ![t] = @ \ {i}]
  ∧ tstate′ = IF waitingFor′[t] = {} THEN [tstate EXCEPT ![t] = "cs"]
                                     ELSE tstate
  ∧ UNCHANGED num

SBExitCS(t) ≜
  ∧ GoTo(t, "cs", "exiting")
  ∧ UNCHANGED ⟨num, waitingFor⟩
```

Figure 4.8a: A simplified version of the atomic bakery algorithm (beginning).

.

$$
\begin{array}{l}
\mathit{SBFinish}(t) \;\triangleq \\
\quad \land\; \mathit{GoTo}(t,\ \text{“exiting”},\ \text{“noncs”}) \\
\quad \land\; \mathit{num}' = [\mathit{num}\ \text{EXCEPT}\ ![t] = 0] \\
\quad \land\; \text{UNCHANGED}\ \mathit{waitingFor}
\end{array}
$$

$$
\begin{array}{l}
\mathit{SBNext} \;\triangleq \\
\quad \exists\, t \in \mathit{Thread} :\ \begin{array}[t]{l} \lor\; \mathit{SBSetNum}(t) \\ \lor\; \exists\, i \in \mathit{Thread} \setminus \{t\} : \mathit{SBWaitFor}(t,\ i) \\ \lor\; \mathit{SBExitCS}(t) \\ \lor\; \mathit{SBFinish}(t) \end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathit{SBEnterOrFinish} \;\triangleq \\
\quad \exists\, t \in \mathit{Thread} :\ \begin{array}[t]{l} \lor\; \exists\, i \in \mathit{Thread} \setminus \{t\} : \mathit{SBWaitFor}(t,\ i) \\ \lor\; \mathit{SBFinish}(t) \end{array}
\end{array}
$$

$$
\mathit{SBSpec} \;\triangleq\; \mathit{SBInit} \land \Box[\mathit{SBNext}]_{\mathit{vars}} \;\land \mathrm{WF}_{\mathit{vars}}(\mathit{SBEnterOrFinish})
$$

Figure 4.8b: A simplified version of the atomic bakery algorithm (end).

.

asserts that the following state predicate is an invariant of *SBSpec*:

$$
\begin{array}{l}
\mathit{MutexInv} \;\triangleq \\
\quad \forall t1, t2 \in \mathit{Thread} \;:\; (\mathit{tstate}[t1] = \text{“cs”}) \land (\mathit{tstate}[t2] = \text{“cs”}) \Rightarrow (t1 = t2)
\end{array}
$$

Very often, the key to correctness of an algorithm lies in the invariance of a state predicate. Indeed, the invariance of *MutexInv* is often taken to be the specification of mutual exclusion. So, it's important to learn how to prove the invariance of a state predicate.

By definition, *MutexInv* is an invariant of *SBSpec* iff *SBSpec* implies $\Box \mathit{MutexInv}$; in other words, iff *MutexInv* is true in every state of every behavior satisfying *SBSpec*. So, to prove that *MutexInv* is an invariant of *SBSpec*, we assume that an infinite sequence $s_1, s_2, s_3, \ldots$ of states satisfies *SBSpec* and prove that *MutexInv* is true on each state s_i. To prove that something is true of s_i, for every positive integer i, we use mathematical induction. This means proving:

1. *MutexInv* is true of the initial state s_1.

2. For all positive integers i, if *MutexInv* is true in a state s_i, then it is true in the next state s_{i+1}.

To prove 1, we remember that s_1 is an initial state of a behavior satisfying *SBSpec* iff it satisfies the initial predicate *SBInit*. Hence, we prove 1 by proving that *SBInit* implies *MutexInv*. This is easy, since *SBInit* implies $\mathit{tstate}[t] \neq \text{“cs”}$ for all threads t.

To prove 2, we remember that s_{i+1} can be the next state after s_i in a behavior satisfying *SBSpec* only if the step $s_i \rightarrow s_{i+1}$ satisfies the next-state action *SBNext* or else is a stuttering step (one that leaves the variables of *SBSpec* unchanged). If $s_i \rightarrow s_{i+1}$ is a stuttering step, then 2 is trivially true. Hence, we must show that:

> For any states s_i and s_{i+1}, if *MutexInv* is true in a state s_i, and $s_i \rightarrow s_{i+1}$ is a step satisfying *SBNext*, then *MutexInv* is true in state s_{i+1}.

The step $s_i \rightarrow s_{i+1}$ satisfies *SBNext* iff *SBNext* is true when unprimed variables are replaced by their values in s_i and primed variables are replaced by their values in s_{i+1}. Hence, the condition above is equivalent to $\mathit{MutexInv} \land \mathit{SBNext} \Rightarrow \mathit{MutexInv}'$, where $\mathit{MutexInv}'$ is the formula obtained by priming all the variables in *MutexInv*.

However, *MutexInv* does not satisfy condition 2 for all states s_i. For example, suppose s_i is a state in which thread $t1$ is waiting to enter its critical section and has finished executing its `await` statement for all threads except thread $t2$, thread $t2$ is in its critical section, and $\mathit{num}[t2] > \mathit{num}[t1]$. Then *MutexInv* will be true in state s_i if no other thread is in its critical section. However, $t1$ can then execute its `await` statement (because $\mathit{num}[t2] > \mathit{num}[t1]$) and enter its critical section, yielding a state s_{i+1} such that $s_i \rightarrow s_{i+1}$ satisfies *SBNext*, but s_{i+1} does not satisfy *MutexInv* because both $t1$ and $t2$ are in their critical sections.

The algorithm is correct because condition 2 is true for all *reachable* states s_i—that is, all states that can occur in a behavior satisfying *SBSpec*. To construct an example in which condition 2 did not hold, we chose a state s_i that isn't reachable.

As often happens in proofs by mathematical induction, we have to strengthen the property we're trying to prove in order to make the induction work. We have to find a state predicate *SBInv* that is stronger than (implies) *MutexInv* satisfying:

1. $\mathit{SBInit} \Rightarrow \mathit{SBInv}$
2. $\mathit{SBInv} \land \mathit{SBNext} \Rightarrow \mathit{SBInv}'$

A state predicate *SBInv* that satisfies these two conditions is called an *inductive invariant* of the specification *SBSpec*.

Finding an inductive invariant that implies the invariant we're trying to prove is a skill that is learned through practice. Satisfying 1 is usually easy; it's condition 2 that's the problem. A predicate satisfying 2 is also called an invariant of action *SBNext*. So, the problem is to find an invariant of *SBNext*.

It's important to learn how to construct inductive invariants. An invariant like *MutexInv* essentially asserts that the algorithm is correct. An inductive

In the literature, the term *invariant* is sometimes used to mean what we are calling an invariant, and sometimes to mean an inductive invariant. Confusion about the difference between the two kinds of invariant has led to "proofs" of incorrect algorithms.

invariant explains why it is correct. To understand why this is so, consider the following proof that our algorithm satisfies mutual exclusion.

> We suppose that two threads, $t1$ and $t2$, are both in their critical section, and we obtain a contradiction. Thread $t1$ read $num[t2]$ in its e_{t2} statement, and $t2$ read $num[t1]$ in its e_{t1} statement. By symmetry, we can assume $t1$'s read occurred last. Then $t1$ read the current value of $num[t2]$, so $num[t2] > num[t1]$. Thread $t2$ would not have entered its critical section had its read obtained the current value of $num[t1]$. Hence, it must have read $num[t1]$ and set $num[t2]$ before $t1$ set $num[t1]$ to its current value. But this implies that $t1$ must have seen the current value of $num[t2]$ when setting $num[t1]$ to its current value, which implies that $num[t1] > num[t2]$. We thus have the required contradiction.

This is the sort of proof that most people naturally tend to write. It explains why two processes can't both be in their critical sections because of previous steps that must have occurred. This kind of reasoning is highly error-prone; it has led to the "proofs" of many incorrect algorithms.

What the algorithm does next depends only on its current state, not on what steps have occurred in the past. Therefore, mutual exclusion is ensured because of some property of its current state. The inductive invariant expresses that property.

So, let's construct the inductive invariant. A good way to start is to write down all the simple invariants we can think of that seem relevant, and to conjoin them with the invariant we're trying to prove. This is often a good place to start. The first and simplest invariant that is relevant is the type correctness invariant. We always start with it. Next, we write relevant relations about the variables of individual threads. In this case, there are two relations that could be relevant:

- $num[t] = 0$ iff thread t is in its noncritical section.
- Thread t goes to its critical section after choosing a nonzero number and ensuring that its number is greater than any other process's number.

In other words:

$$\forall t \in Thread : \begin{array}{l} \land\ (tstate[t] \neq \text{“}\mathsf{noncs}\text{”}) \Rightarrow (num[t] > 0) \\ \land\ (tstate[t] = \text{“}\mathsf{waiting}\text{”}) \Rightarrow (waitingFor[t] \neq \{\}) \end{array} \tag{4.1}$$

Let $Inv1$, our first approximation to the inductive invariant, be the conjunction of the type invariant, (4.1), and $MutexInv$. The state predicate $Inv1$ is an invariant, but not an inductive invariant. For example, the counterexample that we used above to show that $MutexInv$ isn't an inductive invariant can also provide a counterexample for this stronger invariant.

After we've conjoined all the simple invariants to the invariant we're trying to prove, the next thing to try is to work backwards from the invariant we've constructed so far. Let's look at how *MutexInv* can be made false. In other words, how can we reach a state in which two different threads, $t1$ and $t2$, are both in their critical sections? A little thought reveals that the real problem occurs not when both threads have entered their critical sections, but when both have passed the point of waiting for the other. Let's say that $t1$ has passed $t2$ if $t1$ is either in its critical section, or else is still waiting but has already executed its `await` statement $\mathtt{e}_{t2}$. In other words, we can define:

$$\begin{aligned} Passed(t1, t2) \;\triangleq\; &\lor\ tstate[t1] = \text{“cs”} \\ &\lor\ (tstate[t1] = \text{“waiting”}) \land (t2 \notin waitingFor[t1]) \end{aligned}$$

We want to strengthen *MutexInv* by asserting that two threads cannot both have passed each other. So, let our next approximation, *Inv*2, be the conjunction of *Inv*1 and:

$$(4.2)\quad \forall t1, t2 \in Thread \;:\; Passed(t1, t2) \land Passed(t2, t1) \Rightarrow (t1 = t2)$$

Note that (4.2) implies *MutexInv*, since a thread that is in its critical section has passed all threads.

*Inv*2 is still not an inductive invariant. Let's see how predicate (4.2) can be made false. It can be made false starting in a state with $Passed(t1, t2)$ true and $Passed(t2, t1)$ false and taking a step in which $t2$ executes its statement $\mathtt{e}_{t1}$. That step is enabled only if $num[t1] > num[t2]$ or $num[t1] = 0$. Predicate (4.1) (which is a conjunct of *Inv*2) rules out the case $num[t1] = 0$. To rule out the case $num[t1] > num[t2]$, we are led to conjoining

$$(4.3)\quad \forall t1, t2 \in Thread \;:\; Passed(t1, t2) \Rightarrow (num[t2] \geq num[t1])$$

However, this isn't an invariant of the algorithm (that is, it isn't true of all reachable states) because $t1$ can pass $t2$ while $t2$ is in its noncritical section and $num[t2] = 0$. We must weaken (4.3) to:

$$\begin{aligned} (4.4)\quad \forall t1, t2 \in Thread \;:\; Passed(t1, t2) \Rightarrow &\lor\ num[t2] = 0 \\ &\lor\ num[t2] \geq num[t1] \end{aligned}$$

We let *Inv*3 be the conjunction of *Inv*2 and (4.4). It turns out that *Inv*3 is an inductive invariant, so we can let it be *SBInv*. Putting all these conjuncts together, simplifying a bit, and remembering that we can omit the conjunct *MutexInv* because it is implied by (4.2), we get:

$$\begin{aligned} SBInv \;\triangleq\; &\land\ SBTypeOK \\ &\land\ \forall\, t1 \in Thread : \\ &\qquad \land\ (tstate[t1] \neq \text{“noncs”}) \Rightarrow (num[t1] > 0) \\ &\qquad \land\ (tstate[t1] = \text{“waiting”}) \Rightarrow (waitingFor[t1] \neq \{\}) \\ &\qquad \land\ \forall\, t2 \in Thread : \\ &\qquad\qquad \land\ Passed(t1,\ t2) \land Passed(t2,\ t1) \Rightarrow (t1 = t2) \\ &\qquad\qquad \land\ Passed(t1,\ t2) \Rightarrow\ \lor\ num[t2] = 0 \\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \lor\ num[t2] \geq num[t1] \end{aligned}$$

To check that *SBInv* is an inductive invariant, we must check that it's implied by the initial predicate, and that any program step in a state in which *SBInv* is true leaves *SBInv* true. Formally, this means proving $SBInit \Rightarrow SBInv$ and $SBInv \land SBNext \Rightarrow SBInv'$. The first condition is easy to check. The second isn't hard, but it's somewhat tedious. To avoid mistakes, we have to break it into pieces and check each piece separately. We first break it up by splitting the next-state action *SBNext* into its disjuncts. It follows easily from the definition of *SBNext* that, to prove $SBInv \land SBNext \Rightarrow SBInv'$, we must prove that, for all threads t:

1. $SBInv \land SBSetNum(t) \Rightarrow SBInv'$
2. For every thread $i \neq t$: $SBInv \land SBWaitFor(t, i) \Rightarrow SBInv'$
3. $SBInv \land SBExitCS(t) \Rightarrow SBInv'$
4. $SBInv \land SBFinish(t) \Rightarrow SBInv'$

We next break each of these tasks into pieces by proving separately each of the conjuncts of $SBInv'$. For example, 1 is proved by proving

1.1. $SBInv \land SBSetNum(t) \Rightarrow SBTypeOK'$

and, for every thread $t1$:

1.2. $SBInv \land SBSetNum(t) \Rightarrow ((tstate'[t1] \neq \text{“noncs”}) \Rightarrow (num'[t1] > 0))$

1.3. $SBInv \land SBSetNum(t) \Rightarrow ((tstate[t1] = \text{“waiting”}) \Rightarrow (waitingFor[t1] \neq \{\}))$

and, for every pair of threads $t1$ and $t2$:

1.4. $SBInv \land SBSetNum(t) \Rightarrow (Passed(t1, t2)' \land Passed(t2, t1)' \Rightarrow (t1 = t2))$

1.5. $SBInv \land SBSetNum(t) \Rightarrow (Passed(t1, t2) \Rightarrow \begin{array}{l} \lor\ num[t2] = 0 \\ \lor\ num[t2] \geq num[t1]\,) \end{array}$

This gives us 20 conditions to verify. Most of them are easy. For example, consider 1.2. We must assume $SBInv \land SBSetNum(t)$ and prove

$$((tstate'[t1] \neq \text{“noncs”}) \Rightarrow (num'[t1] > 0))$$

If $t \neq t1$, it follows because $SBSetNum(t)$ implies that $tstate'[t1] = tstate[t1]$ and $num'[t1] = num[t1]$, and *SBInv* implies

$$((tstate[t1] \neq \text{“noncs”}) \Rightarrow (num[t1] > 0))$$

If $t = t1$, then it follows because $SBSetNum(t)$ and *SBTypeOK* imply that $num'[t] > 0$ (since $(1 + k) > 0$ for any natural number k).

We can do the same reasoning informally by working from the informal program description instead of the TLA$^+$ specification. For example, 1.2 can be expressed informally as the assertion that executing thread t's statement labeled `enter` in a state satisfying the invariant produces a state in which $num[t1] > 0$ for every thread $t1$ not in its noncritical section.

Whether you write the proof formally or informally, it's important that you do it carefully, breaking the proof into pieces, and writing a separate proof for each piece. (Often, there is some reasoning common to several pieces that can be separated out as lemmas.) Numbering the pieces of the proof helps. It's also useful to name or number the individual parts of the invariant and the individual conjuncts of actions so you can refer to them easily in your proofs. It doesn't matter what method you use to do this. What's important is that you be clear and methodical, so it's easy to see that you've checked everything that needs to be checked. With practice, you'll learn to dispose quickly of the trivial pieces of the proof. In Problem 4.3, you can write out the complete proof of invariance of *SBInv*.

Having proved safety, we now consider liveness. We first show that the algorithm is deadlock free. The best way to do that is to assume that it isn't and obtain a contradiction. The algorithm is not deadlock free if it is possible for there to be some thread waiting to enter its critical section and no thread is or ever will be in its critical section. So, assume that is the case. Eventually any thread that is in its exiting code will complete it, and any thread that is going to try entering its critical section will do so. By invariant *SBInv*, all waiting threads t then have $num[t] > 0$ and all other threads have $num[t] = 0$. If some thread t has the smallest value of $num[t]$ among all waiting threads, then it's easy to see that t will eventually finish executing all its `await` statements $\mathtt{e}_i$ and enter its critical section, which contradicts the assumption that no thread is ever in its critical section. So, we must show that, among all the threads t with $num[t] > 0$, there is whose value of $num[t]$ is strictly less than that of all the other threads. This follows from the observation that two different threads t cannot have the same non-zero value of $num[t]$. In other words, the state predicate

$$(4.5) \quad \forall t1, t2 \in \mathit{Thread} : (num[t1] = num[t2]) \land (num[t1] > 0) \Rightarrow (t1 = t2)$$

is an invariant. It's easy to check that (4.5) is an invariant. In fact the conjunction of it and the type-correctness invariant $num \in [\mathit{Thread} \rightarrow \mathit{Nat}]$ is inductive. This follows from the observation that the only action that sets $num[t]$ greater than 0 sets it to a value greater than that of $num[tt]$ for any other thread tt. This completes the proof that the algorithm is deadlock free.

We prove by contradiction that the algorithm is starvation free. We assume that every thread that enters its critical section eventually exits, but that some thread t waits forever without entering, and we obtain a contradiction. Once t has set $num[t] > 0$ and begun waiting, then every thread tt that afterward tries to enters the critical section will set $num[tt] > num[t]$, and hence will not be able to enter the critical section until t does. If no thread stays forever in its critical section and t never enters the critical section, then eventually no thread will be able to enter the critical section and deadlock will occur. But we have proved that the algorithm cannot deadlock, which is the required contradiction.

Our proof of starvation freedom is simple, but rather informal. We feel that it is convincing enough to give us confidence that the basic algorithm does satisfy that property, though we could easily have made some mistake in writing down the precise algorithm in module *SimpleBakery*. Such a mistake would mean that formula *SBSpec* does not represent the algorithm we think it does—that is, the algorithm we showed to be starvation free. That kind of mistake is likely to appear in an execution with just two or three processes and reasonably small values of $num[t]$. So, we can use TLC to gain confidence that we haven't made a mistake in translating our intuitive idea of what the algorithm does into TLA$^+$. There is no way to check whether we have made such a mistake in writing the informal program on page 104 because we have no formal semantics for that language. (Indeed, it would be hard to formalize the "`...`" notation.) We could try to turn our informal programming notation into a precise programming language with a rigorous semantics. But the semantics would be complicated enough that the precise meaning of the program would not be obvious, so we would find it no easier to tell if the program represented the algorithm we thought it did than if formula *SBSpec* does. In fact, it could very well be harder.

Although very rigorous proofs of liveness can be written in TLA, they tend to be even more tedious than rigorous proofs of invariance. Experience has shown that trying to write a rigorous invariance proof often reveals subtle errors in an algorithm. Writing a rigorous liveness proof seldom finds a serious error. So, we will be content with or informal proof of starvation freedom, and will increase our confidence that the algorithm really does satisfy that property by using TLC to check it on small models.

The Atomic Bakery Algorithm

In the simple atomic bakery algorithm, thread t's `enter` statement atomically reads $num[i]$ for all threads i and sets $num[t]$. The "game" of shared-memory mutual exclusion algorithms is to make a single atomic action only access (read or write) a single memory location. More precisely, it should only access a single shared memory location—one that can be accessed by other threads as well. Accesses to a thread's local state commute with actions of every other thread, so we can reduce a finer-grained program to one in which a single atomic action can access any number of memory locations local to the thread.

So, we want to replace the atomic `enter` statement by one in which each access is a separate action—a statement we can write as

```
⟪ num[t] ⟫ = 1 + Maximum(⟪ num[1] ⟫ , ... , ⟪ num[N] ⟫ )
```

However, the algorithm no longer works with this finer-grained statement. In fact, just separating the evaluation of the Maximum and the setting of num[t] into two separate actions makes the algorithm incorrect. Suppose a thread t reads all the $num[i]$ and finds them equal to 0, and then "sleeps". Other threads could then continue executing, and some other thread tt could eventually enter wind up in its critical section with $num[tt] = 2$. Thread t could then "wakes up", set $num[t]$ to 1, and enter its critical section, since $num[t] < num[tt]$, violating mutual exclusion.

To solve that, we have thread t set a flag $choosing[t]$ while it is executing the nonatomic assignment statement that sets $num[t]$. Another thread waits until the $choosing[t]$ flag is reset before executing the await statement that reads $num[t]$.

There remains one additional problem. When computing the Maximum of all the other thread's numbers and setting $num[t]$ are made separate steps, it becomes possible for two different threads that concurrently try to enter their critical sections to choose the same value of $num[t]$. If this happens, we need to "break the tie" and allow one of the two threads to enter its critical section. We decide in favor of the lowest-numbered thread. So, we let thread t enter its critical section if, for every other thread i, we have:

$$\begin{array}{ll} (4.6) & \lor\ num[i] = 0 \\ & \lor\ num[t] < num[i] \\ & \lor\ (num[t] = num[i]) \land (t < i) \end{array}$$

We define $t \prec i$ to be the state predicate (4.6).

We make one more minor change to the algorthm. There's no particular reason to set $num[t]$ to be one greater than the maximum of the other threads' numbers. We could set it to be any number greater than that maximum. When describing an algorithm, it's usually a good idea to make it as general as possible—especially when the generality doesn't add any extra complexity. The extra generality may turn out to be useful when implementing the algorithm. So, we allow $num[t]$ to be set to any number greater than the maximum of the other numbers. For this purpose, we introduce the program notation that the statement x :> exp means set x to any value greater than exp. Here's the informal program that describes the atomic bakery algorithm.

```
while (true) {
     noncritical section;
 w1: ⟪choosing[t] = true⟫;
 w2: ⟪num[t]⟫ :> Maximum(⟪num[1]⟫, ... , ⟪num[N]⟫) ;
 w3: ⟪choosing[t] = false⟫;
 w4: || (i in 1..N) { if (i != t) {
          $d_i$: ⟪await(!choosing[i])⟫ ;
          $e_i$: ⟪await(t ≺ i)⟫ } };
     critical section;
```

```
        ex: ⟪ num[t] = 0 ⟫  }
```

The formal TLA$^+$ specification is in Figure 4.3.2 on pages 116–118. The following variables capture program state not represented by program variables:

pc The control state, where $pc[t]$ = "w4" means that thread t is executing statement `w4`.

pcw4 For each i different from t, $pcw4[t][i]$ indicates where control is in the i^{th} "iteration" of the `||` statement's body, where $pcw4[t][i] =$ "f" means that statements $\mathtt{d}_i$ and $\mathtt{e}_i$ have been executed.

unRead The set of threads i such that $num[i]$ has not yet been read during execution of statement `w2`.

maxRead The maximum number read so far during execution of statement `w2`.

It is not obvious that this algorithm actually ensures mutual exclusion. We won't try to give a behavioral argument, but will instead construct an inductive invariant. As before, we start with the type invariant and simple assertions relating the values of variables to the control state—in this case, when $num[t]$ is greater than 0 and when $choosing[t]$ is true.

$$(4.7)\quad \begin{array}{l} \land\ ABTypeOK \\ \land\ \forall t \in Thread\ :\ \begin{array}[t]{l} \land\ (num[t] > 0) \equiv (pc[t] \in \{\text{"w3"}, \text{"w4"}, \text{"cs"}, \text{"ex"}\}) \\ \land\ choosing[t] \equiv (pc[t] \in \{\text{"w2"}, \text{"w3"}\}) \end{array} \end{array}$$

We'll use the invariant *SBInv* of the simple algorithm as a guide for the remaining conjuncts. We can redefine *Passed* for this algorithm as:

$$Passed(t1, t2)\ \triangleq\ \begin{array}[t]{l} \lor\ pc[t1] \in \{\text{"cs"}, \text{"ex"}\} \\ \lor\ (pc[t1] = \text{"w4"}) \land (pcw4[t1][t2] = \text{"f"}) \end{array}$$

From the last conjunct of *SBInv*, we would guess that we would need the following conjunct.

$$(4.8)\quad \forall t1, t2 \in Thread\ :\ Passed(t1, t2) \Rightarrow (t1 \prec t2)$$

This is actually not an invariant because it doesn't hold for $t1 = t2$. (The corresponding conjunct of *SBInv* does work for $t1 = t2$.) However, there's a more interesting problem with (4.8): it's not strong enough to make the invariant inductive. To see why not, suppose $t1$ is in its critical section, and thread $t2$ is executing statement `w2` and is about to set $num[t2]$ to a value less than $num[t1]$. Then $Passed(t1, t2)$ and $t1 \prec t2$ hold initially, but $t1 \prec t2$ becomes false when $t2$ sets $num[t2]$. To rule that out, we must conjoin to $t1 \prec t2$ in (4.8) the

```
┌──────────────────── MODULE AtomicBakery ────────────────────┐
EXTENDS Naturals

CONSTANT N
 ASSUME (N  ∈ Nat) / (N ¿ 1)

Thread ≜ 1 . . N

VARIABLES num, choosing, pc, pcw4, unRead, maxRead

vars ≜ ⟨num, choosing, pc, pcw4, unRead, maxRead⟩
├─────────────────────────────────────────────────────────────┤
ABInit ≜ ∧ num      = [t ∈ Thread ↦ 0]
         ∧ choosing = [t ∈ Thread ↦ FALSE]
         ∧ pc       = [t ∈ Thread ↦ "noncs"]
         ∧ pcw4     = [t ∈ Thread ↦ [i ∈ Thread ↦ "f"]]
         ∧ unRead   = [t ∈ Thread ↦ {}]
         ∧ maxRead  = [t ∈ Thread ↦ 0]

ABTypeOK ≜ ∧ num ∈ [Thread → Nat]
           ∧ choosing ∈ [Thread → {TRUE, FALSE}]
           ∧ pc       ∈ [Thread → {"noncs", "w1", "w2", "w3",
                                   "w4", "cs", "ex"}]
           ∧ pcw4     ∈ [Thread → [Thread → {"d", "e", "f"}]]
           ∧ unRead   ∈ [Thread → SUBSET Thread]
           ∧ maxRead  ∈ [Thread → Nat]
├─────────────────────────────────────────────────────────────┤
GoTo(t, loc1, loc2) ≜ ∧ pc[t] = loc1
                      ∧ pc′ = [pc EXCEPT ![t] = loc2]

t1 ≺ t2 ≜ ∨ num[t2] = 0
          ∨ num[t1] < num[t2]
          ∨ (num[t1] = num[t2]) ∧ (t1 < t2)

ABTry(t) ≜
  ∧    GoTo(t, "noncs", "w1")
  ∧    UNCHANGED ⟨num, choosing, pcw4, unRead, maxRead⟩

ABW1(t) ≜
  ∧ GoTo(t, "w1", "w2")
  ∧ choosing′ = [choosing EXCEPT ![t] = TRUE]
  ∧ unRead′   = [unRead EXCEPT ![t] = Thread \ {t}]
  ∧ maxRead′  = [maxRead EXCEPT ![t] = 0]
  ∧ UNCHANGED ⟨num, pcw4⟩
```

Figure 4.9a: The atomic bakery algorithm (beginning).

```
$ABW2Rd(t) \triangleq$
  $\land\ pc[t] = \text{“w2”}$
  $\land\ \exists\, tt \in unRead[t] :$
      $\land\ maxRead' = [maxRead\ \textsc{except}$
                     $![t] = \textsc{if}\ num[tt] > @\ \textsc{then}\ num[tt]\ \textsc{else}\ @]$
      $\land\ unRead' = [unRead\ \textsc{except}\ ![t] = @ \setminus \{tt\}]$
  $\land\ \textsc{unchanged}\ \langle num, choosing, pc, pcw4 \rangle$

$ABW2Wr(t) \triangleq$
  $\land\ GoTo(t, \text{“w2”}, \text{“w3”})$
  $\land\ unRead[t] = \{\}$
  $\land\ \exists\, i \in Nat : \land\ i > maxRead[t]$
                   $\land\ num' = [num\ \textsc{except}\ ![t] = i]$
  $\land\ \textsc{unchanged}\ \langle choosing, pcw4, unRead, maxRead \rangle$

$ABW3(t) \triangleq$
  $\land\ GoTo(t, \text{“w3”}, \text{“w4”})$
  $\land\ choosing' = [choosing\ \textsc{except}\ ![t] = \textsc{false}]$
  $\land\ pcw4' = [pcw4\ \textsc{except}\ ![t] = [tt \in Thread \mapsto \text{“d”}]]$
  $\land\ \textsc{unchanged}\ \langle num, unRead, maxRead \rangle$

$ABW4d(t, i) \triangleq$
  $\land$    $pc[t] = \text{“w4”}$
  $\land$    $pcw4[t][i] = \text{“d”}$
  $\land$    $\lnot choosing[i]$
  $\land$    $pcw4' = [pcw4\ \textsc{except}\ ![t][i] = \text{“e”}]$
  $\land$    $\textsc{unchanged}\ \langle num, choosing, pc, unRead, maxRead \rangle$

$ABW4e(t, i) \triangleq$
  $\land$    $pc[t] = \text{“w4”}$
  $\land$    $pcw4[t][i] = \text{“e”}$
  $\land$    $t \prec i$
  $\land$    $pcw4' = [pcw4\ \textsc{except}\ ![t][i] = \text{“f”}]$
  $\land$    $\textsc{unchanged}\ \langle num, choosing, pc, unRead, maxRead \rangle$

$ABEnter(t) \triangleq$
  $\land\ GoTo(t, \text{“w4”}, \text{“cs”})$
  $\land\ \forall\, i \in Thread \setminus \{t\} : pcw4[t][i] = \text{“f”}$
  $\land\ \textsc{unchanged}\ \langle num, choosing, pcw4, unRead, maxRead \rangle$

$ABExitCS(t) \triangleq$
  $\land\ GoTo(t, \text{“cs”}, \text{“ex”})$
  $\land\ \textsc{unchanged}\ \langle num, choosing, pcw4, unRead, maxRead \rangle$
```

Figure 4.9b: The atomic bakery algorithm (continued).

$$\begin{array}{l}
ABEx(t) \;\triangleq \\
\quad \land\ GoTo(t,\ \text{“ex”},\ \text{“noncs”}) \\
\quad \land\ num' = [num \text{ EXCEPT } ![t] = 0] \\
\quad \land\ \text{UNCHANGED } \langle choosing,\ pcw4,\ unRead,\ maxRead \rangle
\end{array}$$

$$\begin{array}{l}
ABNext \;\triangleq \\
\quad \exists\, t \in Thread : \\
\qquad \lor\ ABTry(t) \lor ABW1(t) \lor ABW2Rd(t) \lor ABW2Wr(t) \lor ABW3(t) \\
\qquad\quad \lor\ ABEnter(t) \lor ABExitCS(t) \lor ABEx(t) \\
\qquad \lor\ \exists\, i \in Thread \setminus \{t\} : ABW4d(t,\ i) \lor ABW4e(t,\ i)
\end{array}$$

$$\begin{array}{l}
ABEnterOrFinish \;\triangleq \\
\quad \exists\, t \in Thread : \\
\qquad \lor\ ABW1(t) \lor ABW2Rd(t) \lor ABW2Wr(t) \lor ABW3(t) \lor ABEnter(t) \lor ABEx(t) \\
\qquad \lor\ \exists\, i \in Thread \setminus \{t\} : ABW4d(t,\ i) \lor ABW4e(t,\ i)
\end{array}$$

$$ABSpec \;\triangleq\; ABInit \land \Box[ABNext]_{vars} \;\land \mathrm{WF}_{vars}(ABEnterOrFinish)$$

Figure 4.9c: The atomic bakery algorithm (end).

requirement that, if $t2$ is executing `w2`, then it has either not yet read $num[t1]$ or else has read some value at least as large as $num[t1]$. Let's define

$$\begin{array}{lcl}
WillBeSeen(t1, t2) & \triangleq & (pc[t2] = \text{“w2”}) \Rightarrow \lor\ t1 \in unRead[t2] \\
 & & \qquad\qquad\qquad\quad \lor\ maxRead[t2] \geq num[t1]
\end{array}$$

Then we replace (4.8) with

$$\begin{array}{ll}
(4.9) & \forall t1 \in Thread\ : \forall t2 \in Thread \setminus \{t1\}\ : \\
 & \quad Passed(t1, t2) \Rightarrow (t1 \prec t2) \land WillBeSeen(t1, t2)
\end{array}$$

We're getting closer, but our invariant is still not strong enough to be inductive. The problem is that (4.9) isn't strong enough to ensure that it holds after $t1$ executes its statement e_{t2}. This step changes $Passed(t1, t2)$ from false to true, but (4.9) doesn't imply that the step makes $WillBeSeen(t1, t2)$ true. In any reachable state, $WillBeSeen(t1, t2)$ will be true whenever control in $t1$ is at statement e_{t2}. So, we strenghten our invariant to assert that $WillBeSeen(t1, t2)$ is true when $pc[t1] = \text{“w4”}$ and $pcw4[t1][t2] = \text{“f”}$. This strengthening does yield an inductive invariant. Putting it all together, we have the inductive invariant:

$$\begin{array}{l}
ABInv \;\triangleq \\
\quad \land\ ABTypeOK \\
\quad \land\ \forall\, t1 \in Thread : \\
\qquad\quad \land\ (pc[t1] \in \{\text{“w3”},\ \text{“w4”},\ \text{“cs”},\ \text{“ex”}\}) \equiv (num[t1] > 0) \\
\qquad\quad \land\ choosing[t1] \equiv (pc[t1] \in \{\text{“w2”},\ \text{“w3”}\})
\end{array}$$

$$\begin{array}{l} \land\ \forall\, t2 \in \mathit{Thread} \setminus \{t1\} : \\ \quad \land\ \mathit{Passed}(t1,\ t2) \Rightarrow (t1 \prec t2) \land \mathit{WillBeSeen}(t1,\ t2) \\ \quad \land\ (pc[t1] = \text{``}\mathsf{w4}\text{''}) \land (pcw4[t1][t2] = \text{``}\mathsf{e}\text{''}) \Rightarrow \mathit{WillBeSeen}(t1,\ t2) \end{array}$$

It's not hard to see that ABInv implies the mutual exclusion condition

$$(4.10)\ \ \forall t1, t2 \in \mathit{Thread}\ :\ (pc[t1] = \text{``}\mathsf{cs}\text{''}) \land (pc[t2] = \text{``}\mathsf{cs}\text{''}) \Rightarrow (t1 = t2)$$

It's easy to see that the algorithm is deadlock free, since if some thread is waiting, then there must be one thread t among the waiting threads such that $t \prec i$ for all other threads i. The atomic bakery algorithm is starvation free for essentially the same reason as the simple atomic bakery algorithm. You can provide the details in Problem 4.4.

4.4 The General Multiclient Server

4.4.1 The Specification

In the simple multiclient server specified above, the system issues an immediate response to each client request. To see why that's not general enough, consider the problem of implementing mutual exclusion with a multiclient server system. A client c that wants to enter its critical section issues an *enter* request. If no process is in its critical section, the system can immediately perform the *enter* operation and issue an *OK* response, allowing c to enter its critical section. If a process is in its critical section, the system cannot perform the *enter*. Instead, it puts c on a waiting queue. When the client currently executing its critical section is finished, it issues an *exit* request. The system performs that operation, generating with an *OK* response. It can then execute the *enter* request of some waiting process.

To handle this kind of multiclient server system, we generalize the specification of module *SimpleMultiClientServer* to allow requests that that may not always be enabled (ready for execution). We add a new operator, *RequestEnabled*, to indicate whether a client's request is enabled in a state. We would also like to be able to schedule the execution of pending enabled requests. We do this by letting whether or not a request is enabled depend on what other requests are pending. We let *pendingQ* be the queue of pending requests—ones for which the *Call* operation has been executed but the request itself, which changes the state, has not been performed. We then let *RequestEnabled* also take the queue of pending requests as an argument.

It seems natural to let the action that performs the *Call* operation also append the request on *pendingQ*. However, if we do this, then we can specify that the order in which two requests are performed depends on the order in which their *Call* operations occur—for example, by letting a request be enabled only if it is at the head of *pendingQ*. This could be difficult to implement. For example, suppose a client's call operation were implemented by changing the voltages on a set of wires. In a distributed system, two clients' wires could be attached to different computers, making it very difficult for the system to determine which one's voltage changed first. We therefore add an extra internal system action that enqueues a client's request on *pendingQ*, which must be performed before the request is performed.

The complete internal specification of the multiclient server is formula *IMCSSpec* of module *IMultiClientServer* in Figure 4.4.1 on the following two pages. The specification differs from that of the simple multiclient server in Figure 4.2 on pages 93–94 in the following significant ways:

- It defines $Remove(i, q)$ to be the sequence obtained from a sequence q by removing the i^{th} element.

- It introduces the variable *pendingQ* and the operator parameter *RequestEnabled*, where $RequestEnabled(c, v, s, q)$ is true iff client c's request v is enabled in state s when *pendingQ* equals q.

- It adds a fourth parameter q to *NewState* and *ResponseVal* that represents the value of *pendingQ*. This allows the state machine to change its state and compute a new value based on the pending requests as well as the current request.

- It introduces an assumption about *RequestEnabled*, and modifies the assumption on *NewState* so $NewState(c, v, s, q)$ must be a state only if client c' request v is enabled in state s when the queue of pending requests is q.

- The control state "pending" and the action $MCSEnqueue(c)$ have been added. That action appends $\langle c, v \rangle$ to *pendingQ*, where v is client c's current request.

- The $MCSDo(c)$ action has the additional enabling condition specified by *RequestEnabled*.

We let the liveness condition of *IMCSSpec* be weak fairness on each client's actions—other than that of issuing a request. Unlike action $SMCDo(c)$ of the simple multiclient server, the $MCSDo(c)$ action of the general multiclient server can be disabled by an action of another client. (Such an action could make the *RequestEnabled* conjunct false.) So, strong fairness of a client's action would give a different specification. As we shall see, we can effectively specify strong fairness by a suitable choice of the *RequestEnabled* operator.

We define the specification multiclient server specification *MCSSpec* to equal $\exists\, sstate, cstate, pendingQ : IMCSSpec$, with the usual separate module *MultiClientServer* that instantiates module *IMultiClientServer*.

MODULE *IMultiClientServer*

EXTENDS *Naturals*, *Sequences*

$$Remove(i,\ q) \triangleq [k \in 1 \,..\, (Len(q) - 1) \mapsto \text{IF } k < i \text{ THEN } q[k] \text{ ELSE } q[k+1]]$$

$$\begin{array}{ll}\text{CONSTANTS} & Client,\ Request,\ State,\ InitialState,\ NewState(\_,\ \_,\ \_,\ \_), \\ & ResponseVal(\_,\ \_,\ \_,\ \_),\ RequestEnabled(\_,\ \_,\ \_,\ \_)\end{array}$$

$$\begin{array}{l}Response \triangleq \{ResponseVal(c,\ v,\ s,\ q) : \\ \qquad\qquad c \in Client,\ v \in Request,\ s \in State,\ q \in Seq(Client \times Request)\}\end{array}$$

$$\begin{array}{l}\text{ASSUME } \land\ InitialState \in State \\ \qquad\quad\ \land\ \forall\, c \in Client,\ v \in Request,\ s \in State : \\ \qquad\qquad\quad \land\ \forall\, q \in Seq(Client \times Request) : \\ \qquad\qquad\qquad \land\ RequestEnabled(c,\ v,\ s,\ q) \in \{\text{TRUE},\ \text{FALSE}\} \\ \qquad\qquad\qquad \land\ RequestEnabled(c,\ v,\ s,\ q) \Rightarrow NewState(c,\ v,\ s,\ q) \in State\end{array}$$

VARIABLES *sstate*, *cstate*, *pendingQ*, *iFace*

$$\begin{array}{l}\text{INSTANCE } CSCallReturn \text{ WITH } Input \leftarrow Client \times Request, \\ \qquad\qquad\qquad\qquad\qquad\quad\ \, Output \leftarrow Client \times Response\end{array}$$

$$\begin{array}{l}MCSInit \triangleq\ \land\ sstate = InitialState \\ \qquad\qquad\quad\ \land\ \exists\, v \in Response : \\ \qquad\qquad\qquad\quad cstate = [c \in Client \mapsto [ctl \mapsto \text{“idle”},\ val \mapsto v]] \\ \qquad\qquad\quad\ \land\ pendingQ = \langle\rangle \\ \qquad\qquad\quad\ \land\ CRInit\end{array}$$

$$\begin{array}{l}MCSTypeInvariant \triangleq \\ \quad \land\ sstate \in State \\ \quad \land\ cstate \in [Client \rightarrow [ctl : \{\text{“idle”},\ \text{“calling”},\ \text{“pending”},\ \text{“returning”}\}, \\ \qquad\qquad\qquad\qquad\qquad val : Request \cup Response]] \\ \quad \land\ \forall\, c \in Client : \\ \qquad\quad \land\ (cstate[c].ctl \in \{\text{“calling”},\ \text{“pending”}\}) \Rightarrow \\ \qquad\qquad\quad (cstate[c].val \in Request) \\ \qquad\quad \land\ (cstate[c].ctl \in \{\text{“returning”},\ \text{“idle”}\}) \Rightarrow \\ \qquad\qquad\quad (cstate[c].val \in Response) \\ \quad \land\ pendingQ \in Seq(Client \times Request) \\ \quad \land\ \forall\, i \in 1 \,..\, Len(pendingQ) : \\ \qquad\quad \land\ cstate[pendingQ[i][1]].ctl\ = \text{“pending”} \\ \qquad\quad \land\ cstate[pendingQ[i][1]].val = pendingQ[i][2] \\ \quad \land\ CRTypeOK\end{array}$$

Figure 4.10a: Specification of a multiclient server (beginning).

$$
\begin{array}{l}
\mathit{MCSIssueRequest}(c,\ \mathit{req})\ \triangleq \\
\quad \land\ \mathit{cstate}[c].\mathit{ctl} = \textsf{``idle''} \\
\quad \land\ \mathit{Call}(\langle c,\ \mathit{req}\rangle) \\
\quad \land\ \mathit{cstate}' = [\mathit{cstate}\ \textsc{except}\ ![c] = [\mathit{ctl} \mapsto \textsf{``calling''},\ \mathit{val} \mapsto \mathit{req}]] \\
\quad \land\ \textsc{unchanged}\ \langle \mathit{sstate},\ \mathit{pendingQ}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{MCSEnqueue}(c)\ \triangleq \\
\quad \land\ \mathit{cstate}[c].\mathit{ctl} = \textsf{``calling''} \\
\quad \land\ \mathit{cstate}' = [\mathit{cstate}\ \textsc{except}\ ![c].\mathit{ctl} = \textsf{``pending''}] \\
\quad \land\ \mathit{pendingQ}' = \mathit{Append}(\mathit{pendingQ},\ \langle c,\ \mathit{cstate}[c].\mathit{val}\rangle) \\
\quad \land\ \textsc{unchanged}\ \langle \mathit{sstate},\ \mathit{iFace}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{MCSDo}(c)\ \triangleq \\
\quad \land\qquad \mathit{cstate}[c].\mathit{ctl} = \textsf{``pending''} \\
\quad \land\qquad \mathit{RequestEnabled}(c,\ \mathit{cstate}[c].\mathit{val},\ \mathit{sstate},\ \mathit{pendingQ}) \\
\quad \land\qquad \mathit{sstate}' = \mathit{NewState}(c,\ \mathit{cstate}[c].\mathit{val},\ \mathit{sstate},\ \mathit{pendingQ}) \\
\quad \land\qquad \mathit{cstate}' = [\mathit{cstate}\ \textsc{except} \\
\qquad\qquad\qquad\qquad ![c] = [\mathit{ctl} \mapsto \textsf{``returning''}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{val} \mapsto \mathit{ResponseVal}(c,\ \mathit{cstate}[c].\mathit{val},\ \mathit{sstate},\ \mathit{pendingQ})]] \\
\quad \land\qquad \mathit{pendingQ}' = \mathit{Remove}(\textsc{choose}\ i \in 1 \mathrel{..} \mathit{Len}(\mathit{pendingQ}) : \mathit{pendingQ}[i][1] = c, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{pendingQ}) \\
\quad \land\qquad \textsc{unchanged}\ \mathit{iFace}
\end{array}
$$

$$
\begin{array}{l}
\mathit{MCSRespond}(c)\ \triangleq \\
\quad \land\ \mathit{cstate}[c].\mathit{ctl} = \textsf{``returning''} \\
\quad \land\ \mathit{Return}(\langle c,\ \mathit{cstate}[c].\mathit{val}\rangle) \\
\quad \land\ \mathit{cstate}' = [\mathit{cstate}\ \textsc{except}\ ![c].\mathit{ctl} = \textsf{``idle''}] \\
\quad \land\ \textsc{unchanged}\ \langle \mathit{sstate},\ \mathit{pendingQ}\rangle
\end{array}
$$

$$
\begin{array}{ll}
\mathit{MCSNext}\ \triangleq\ \exists\, c \in \mathit{Client} : & \lor\ \exists\, \mathit{req} \in \mathit{Request} : \mathit{MCSIssueRequest}(c,\ \mathit{req}) \\
& \lor\ \mathit{MCSEnqueue}(c) \\
& \lor\ \mathit{MCSDo}(c) \\
& \lor\ \mathit{MCSRespond}(c)
\end{array}
$$

$$
\mathit{mcsvars}\ \triangleq\ \langle \mathit{cstate},\ \mathit{sstate},\ \mathit{pendingQ},\ \mathit{iFace}\rangle
$$

$$
\begin{array}{ll}
\mathit{IMCSSpec}\ \triangleq & \land\ \mathit{MCSInit} \\
& \land\ \Box[\mathit{MCSNext}]_{\mathit{mcsvars}} \\
& \land\ \forall\, c \in \mathit{Client} : \\
& \qquad \mathrm{WF}_{\mathit{mcsvars}}(\mathit{MCSEnqueue}(c) \lor \mathit{MCSDo}(c) \lor \mathit{MCSRespond}(c))
\end{array}
$$

Figure 4.10b: Specification of a multiclient server (end).

4.4.2 A Mutual Exclusion Server

The Specification

As an example of the use of the multiclient server specification, we now specify mutual exclusion as a multiclient server. Our specification differs from the one in the *MutualExclusion* module on page 102 in two ways. The uninteresting difference is that we use the call/return interface of module *CSCallReturn* instead of simply setting the variable *tstate*. The more interesting difference is that our earlier specification asserts that threads alternately try to enter and exit their critical sections. A corresponding multiclient server specification would require that a thread can issue an "exit" request only if its previous request was "enter", and it cannot issue two successive "enter" requests. However, our multiclient server specification allows a client to issue any request whenever it has received a response from any previous request. It has no mechanism for asserting that a client should not issue two consecutive "exit" requests.

We can write a more general multiclient server specification that permits us to constrain the conditions under which a client can issue a request. However we have chosen not to do that, and to leave such a specification as Problem 4.11 on page 140. When designing a service such as one to implement mutual exclusion, it is usually best to make it handle any request that the client might issue, returning an error value if the client should not have issued the request. We must therefore specify our mutual exclusion server to handle illegal requests.

We want to specify a server that provides starvation free mutual exclusion, so a waiting thread eventually enters its critical section if every thread that enters the critical section eventually exits. We do this by defining *RequestEnabled* so an "enter" request is enabled only if it is at the head of *pendingQ*. This implies that "enter" requests from different threads are executed in the order in which they are put on the pending queue. Since fairness of each client's action implies that an issued request is eventually placed on the pending queue, every "enter" request eventually reaches the head of the queue and is executed, unless some thread fails to exit its critical section. Thus, we have used the definition of *RequestEnabled* to transform the weak fairness guarantee of the multiclient server specification into the strong fairness guarantee of starvation freedom. The specification is in Figure 4.11 on the next page.

Relation to Our Other Specification

The specification in module *MCSMutex* is more liberal than that in the *MutualExclusion* module, since it allows behaviors in which threads try to issue illegal requests. We would therefore expect that the specification *MEStarvationFree* of starvation-free mutual exclusion in module *MutualExclusion* would implement specification *MCSSpec* of module *MCSMutex* (obtained by instantiating formula *MCSSpec* of the *MulticlientServer* module), except that the two specifications have dif-

```
┌──────────────────────── MODULE MCSMutex ────────────────────────┐
EXTENDS Sequences, Naturals

CONSTANT Thread
NotAThread ≜ CHOOSE n : n ∉ Thread

Request ≜ {"enter", "exit"}

State ≜ Thread ∪ {NotAThread}

InitialState ≜ NotAThread

NewState(t, req, st, q) ≜
  CASE req = "enter" → t  □
       req = "exit"  → IF st = t THEN NotAThread ELSE st

ResponseVal(t, req, st, q) ≜
  CASE req = "enter" → IF st = t THEN "error" ELSE "OK"    □
       req = "exit"  → IF st = t THEN "OK"    ELSE "error"

RequestEnabled(t, req, st, q) ≜
  CASE req = "enter" → ∨ (st = NotAThread) ∧ (t = Head(q)[1]) □
                       ∨ st = t
       req = "exit"  → TRUE
├─────────────────────────────────────────────────────────────────┤
VARIABLES iFace
INSTANCE MultiClientServer WITH Client ← Thread
└─────────────────────────────────────────────────────────────────┘
```

Figure 4.11: A multi-thread mutual exclusion server specification.

ferent visible variables. Specification *MEStarvationFree* has the visible (and only) variable *tstate*, while *MCSSpec* has the visible variable *iFace*. Instead, we would expect *MEStarvationFree* to implement *MCSSpec* under an interface refinement—a refinement mapping that can change *iFace*.

To show that *MEStarvationFree* implements *MCSSpec* under a refinement mapping, we must show that it implements the internal specification *IMCSSpec* under a refinement mapping that substitutes expressions for the internal variables *sstate*, *cstate*, and *pendingQ* as well as for the visible variable *iFace*. However, there is a problem in trying to define such a refinement mapping. In a behavior described by the specification *MEStarvationFree*, a thread t enters its critical section by performing an *METry*(t) step followed by an *MEEnterCS*(t) step. It would be natural for those steps to implement the *MCSIssueRequest*(t, "enter") and *MCSRespond*(t) steps, respectively, of specification *IMCSSpec*. However, between those two steps in a behavior of *IMCSSpec*, there are an *MCSEnqueue*(t) step and an *MCSDo*(t) step. These two steps have no counterparts in specification *MEStarvationFree*; they cannot be described in terms of changes to the variable *tstate* of specification *MEStarvationFree*. Hence, we cannot define the refinement mapping in terms of the variable *tstate* alone. We must add one or more auxiliary variables.

We introduced auxiliary variables in Section 3.5.1. Recall that adding an auxiliary variables a to a specification *Spec* means writing a specification *ASpec* such that *Spec* is equivalent to $\boldsymbol{\exists}\, a : ASpec$. The auxiliary variables we have considered thus far are history variables, which remember information from previous states. Here, we need another kind of auxiliary variable called a *stuttering* variable that adds stuttering steps—extra steps that don't change any of the specification's actual variables.

In general, suppose our specification *Spec* equals $Init \land \Box[Next]_{vbl} \land L$, where L is the liveness condition. We add a stuttering variable s to *Spec* to form a new specification *SSpec* as follows. The value of s is a natural number. When $s = 0$, specification *SSpec* allows *Next* steps that can set s to any natural number. When $s > 0$, it allows only *Stutter* steps that decrement s and leave the variables of *SSpec* unchanged. (These are the extra stuttering steps.) We also require weak fairness of the *Stutter* action to rule out behaviors that halt with $s > 0$. The specification is then defined as follows, where *sInitVal* is any expression containing the (unprimed) variables of *Spec* and *sNextVal* is any expression containing unprimed and/or primed variables of *Spec*:

$$\begin{array}{lcl}
SInit & \triangleq & Init \land (s = sInitVal) \\
Stutter & \triangleq & (s > 0) \land (s' = s - 1) \land \text{UNCHANGED } vbl \\
SNext & \triangleq & \lor\ (s = 0) \land Next \land (s' = sNextVal) \\
 & & \lor\ Stutter \\
SSpec & \triangleq & SInit \land \Box[SNext]_{\langle vbl, s\rangle} \land \mathrm{WF}_{\langle vbl, s\rangle}(Stutter) \land L
\end{array}$$

Specification *SSpec* allows steps satisfying *Next* and *Stutter* steps that leave the variables of *Spec* unchanged. If we hide s in *SSpec*, these steps are just steps allowed by *Next* and stuttering steps, which are also allowed by *Spec*. It should be clear that the resulting specification is equivalent to *Spec*. In other words, $\boldsymbol{\exists}\, s : SSpec$ is equivalent to *Spec*.

We can modify this construction in a couple of ways. If *Next* is a disjunction of several subactions A, we can construct *SNext* by replacing each A with

$$(s = 0) \land A \land (s' = exp_A)$$

for some expression exp_A. For a multiprocess specification, where the next-state action has the form $\exists p \in Proc : Next(p)$ and each subaction is parametrized by p, we can use an array *sArray* of stuttering variables. We replace $A(p)$ by

$$\begin{array}{l}
\land\ sArray[p] = 0 \\
\land\ A(p) \\
\land\ sArray' = [sArray \text{ EXCEPT } ![p] = exp_A]
\end{array}$$

and we replace the single *Stutter* action by

$$\begin{array}{rcl} \mathit{Stutter}(p) & \triangleq & \land\ \mathit{sArray}[p] > 0 \\ & & \land\ \mathit{sArray}' = [\mathit{sArray}\ \text{EXCEPT}\ ![p] = @ - 1] \\ & & \land\ \text{UNCHANGED}\ \mathit{vbl} \end{array}$$

The additional fairness condition is $\forall p \in \mathit{Proc} : \mathrm{WF}_{\langle \mathit{vbl}, s \rangle}(\mathit{Stutter}(p))$.

We now sketch the construction of a refinement mapping under which the specification *MEStarvationFree* of module *MutualExclusion* implements specification *IMCSSpec* of module *MCSMutex*. We add an array *sArray* that adds one stuttering step to each of the steps of the next-state action of *MEStarvationFree*. (In other words, exp_A equals 1.) The steps of *IMCSSpec* are implemented by steps of *MEStarvationFree* as follows:

MCSIssueRequest	by *MeTry* or *MEExitCS*
MCSEnqueue	by the stuttering step after *MeTry* or *MEExitCS*
MCSDo	by *MEEnterCS* or *MEFinish*
MCSRespond	by the stuttering step after *MEEnterCS* or *MEFinish*

After adding the stuttering variable *sArray*, we have to add history variables to record the following information:

- The order in which requests are enqueued, contained in variable *pendingQ* (of specification *IMCSSpec*).
- The total number of calls and returns, modulo two, recorded in the *abit* and *rbit* components of variable *iFace*.

The easiest way to do this is to add two history variables that equal the *pendingQ* and *iFace*. (In fact, *pendingQ* and *iFace* would be good names for those variables.) Adding these auxiliary variables to specification *MEStarvationFree* and constructing the refinement mapping under which it implements *IMCSSpec* is left as Problem 4.15 on page 141.

A Closer Look at the Pending Queue

The specification of mutual exclusion in module *MCSMutex* seems to guarantee not just starvation freedom, but the stronger requirement of first-come, first-served (FCFS) entry to the critical section. Starvation freedom just means that every waiting thread eventually enter its critical section. FCFS entry means that any thread t must enter before another thread that requested entry after t did.

The specification in module *MCSMutex* does not really guarantee FCFS entry. FCFS entry means that processes enter in the order in which they requested entry. The specification ensures that they enter in the order in which their requests were appended to the queue *pendingQ* of pending requests. The order in which requests are appended to *pendingQ* need not be the same as the order in which those requests were issued.

The true specification of mutual exclusion specifies only the sequence of calls and returns, described by the sequence of values of the variable *iFace*. The other variables, including *pendingQ* are hidden. The order in which requests are appended to the pending queue is not externally visible. In Problem 4.10 on page 140, you will show that the specification in module *MCSMutex* allows any starvation free behavior.

4.4.3 Readers/Writers

A generalization of mutual exclusion is a readers/writers system in which clients perform two kinds of operations: reading and writing of some data. Multiple reader operations can be performed concurrently, but a write must have exclusive access to the data. There are four client requests: *enter read* and *enter write*, with which the client requests permission to begin its operation, and *exit read* and *exit write*, with which the client signals the completion of its operation.

We now have a choice of assigning priorities among pending client requests. There are three natural conditions:

Reader Priority A pending *enter read* request takes priority over a pending *enter write* request.

Writer Priority A pending *enter write* request takes priority over a pending *enter read* request.

FIFO The *enter read* or *enter write* operations are performed in the order that they appear in the pending queue.

The specification in module *ReadersWriters* in Figure 4.12 on the next page specifies a readers/writers system with reader priority.

4.5 Monitors

Some programming languages, such as Java, provide a programming abstraction that allows one to implement a multiclient server using the client threads themselves. That is, each client thread determines when its operation is enabled, changes the state of the state machine and computes the result that is returned to the client module. This programming abstraction is traditionally called a *monitor*, although in Java it is called a *synchronized object*. A Java monitor behaves somewhat differently from other monitors (for example, from the original monitors idea as presented by C. A. R. Hoare, and from the monitors provided in the Modula 3 and Mesa programming languages). We will talk only about the kind of monitor that Java provides.

```
──────────────────── MODULE ReadersWriters ────────────────────
EXTENDS FiniteSets, Sequences, Naturals
CONSTANTS Client

Request ≜ {“enter_read”, “exit_read”, “enter_write”, “exit_write”}

State ≜ [rdrs : SUBSET Client, wrtrs : SUBSET Client]

InitialState ≜ [rdrs ↦ {}, wrtrs ↦ {}]

NewState(c, v, s, q) ≜
  CASE v = “enter_read”  → [s EXCEPT !.rdrs  = @ ∪ {c}]  □
       v = “exit_read”   → [s EXCEPT !.rdrs  = @ \ {c}]  □
       v = “enter_write” → [s EXCEPT !.wrtrs = @ ∪ {c}]  □
       v = “exit_write”  → [s EXCEPT !.wrtrs = @ \ {c}]

RequestEnabled(c, v, s, q) ≜
  CASE v = “enter_read”  → s.wrtrs = {}                □
       v = “exit_read”   → TRUE                        □
       v = “enter_write” → ∨ ∧ s.wrtrs = {}
                             ∧ ∀ i ∈ 1 . . Len(q) :
                                  q[i][2] ≠ “enter_read”    □
       v = “exit_write”  → TRUE

ResponseVal(c, v, s, q) ≜
  CASE v ∈ {“enter_read”, “enter_write”}
          → IF c ∉ (s.rdrs ∪ s.wrtrs) THEN “OK” ELSE “ERROR”     □
       v = “exit_read”  → IF c ∈ s.rdrs THEN “OK” ELSE “ERROR”  □
       v = “exit_write” → IF c ∈ s.wrtrs THEN “OK” ELSE “ERROR”

VARIABLES sstate, cstate, pendingQ, iFace

INSTANCE MultiClientServer
────────────────────────────────────────────────────────────────
```

Figure 4.12: A reader-priority readers/writers system.

A monitor is a critical section associated with an object that is accessed by the object's methods. A method that has the `synchronized` modifier acquires mutual exclusion to the critical section of its object and releases mutual excusion when it returns. A method can also release mutual excusion by executing the `wait()` method, which causes the thread to block after releasing mutual excusion. A thread waiting in a method call to object x can be restarted with a `notify()` or `notifyAll()` method call on object x. The first method will unblock exactly one thread blocked in x if there are any waiting, and the second unblocks all threads blocked in x. Each unblocked thread needs to acquire mutual exclusion again before proceeding with its method call.

For example, consider the Java semaphore class given in Figure 2.7. This class has two methods with the synchronized modifier, each one corresponding to a semaphore operation. A `s.P()` method first acquires mutual excusion to the semaphore object s. If the value of the semaphore is zero, then the method relinquishes mutual excusion and blocks. A thread t blocked in such a way is unblocked only when another thread executes a `s.V()` method, but t will not resume execution until it also reacquires mutual excusion to s.

There are a few fine points that complicate this simple abstraction:

- Consider a thread t that blocks by executing a `wait()` method. When t is unblocked (via a call to either a `notify()` or `notifyAll()` method), t needs to reacquire mutual exclusion. It has no particular priority to do so. For example, another thread that is already blocked waiting to acquire mutual exclusion (either because it has called a synchronized method or has also been unblocked from a `wait()`) can obtain mutual excusion before t does. This explains why the Java Semaphores class has `wait()` executed in a `while (x == 0)` loop: another thread may call `P()` and obtain mutual exclusion before t does, in which case `x` would be zero by the time t reenters the monitor.

- Java provides no way to determine which threads are blocked waiting to acquire mutual exclusion on an object. The ability to do so would not be very useful. Suppose a thread t_1 were to determine that the set of threads T are waiting to acquire mutual exclusion. Having determined this, a new thread t_2 could call a synchronized method, which would make T out of date. Luckily, the state machine model we use does not require us to know this set of threads.

- A thread that is waiting can be interrupted by the `interrupt()` method. This method raises the `InterruptedException` exception. Hence, any invocation of `wait` needs to be wrapped in a `try` phrase.

- There is no direct way to determine which threads are blocked having done a `wait()`, but it's easy to add logic to keep track of this information. Usually, you only keep some summary information about the blocked processes.

- A synchronized method of x can call a synchronized method of y. If $x = y$, then no ill effect occurs except that mutual excusion of x is not relinquished until the outermost synchronized method is invoked. If $x \neq y$, then the invoking method will hold mutual excusion on both x and y. Holding multiple locks can lead to deadlock. For example, let thread t_1 calls first a synchronized method of object x and thread t_2 call a synchronized method of object y. If t_1 then calls a synchronized method of y and t_2 calls a synchronized method of x, then the two threads deadlock. A third thread could be used to detect such a deadlock and use the `interrupt()` method to break the deadlock.

4.5.1 A Methodology for writing Java Monitors

Since we can determine, with a bit of programming, which threads are blocked because they executed a `wait()`, we will use `wait()`, `notify()`, and `notifyAll()` to provide the function of *pendingQ* and the *RequestEnabled* operator.

Figure 4.13 gives a template for how we represent a multiclient state machine as a monitor. There is a synchronized method for each request type. Once a thread makes it into the monitor, it executes the code associated with *MCSEnqueue*. It then tests whether its own request is enabled, using an implementation of *RequestEnabled* based on the encoding of the monitor state and pending queue. If its request is not enabled, then it waits using `wait()`. The test is done as a `while` loop because, as in the Semaphores class, a thread may wake up to find that its request is not enabled.

Once out of the `while` loop, *MCSDo* is implemented using the encoding of the monitor state. In doing so, $\mathit{RequestEnabled}(c, \mathit{cstate}[c].\mathit{val}, \mathit{sstate}, \mathit{pendingQ})$ may become true for one or more clients c. If so, then the client thread uses `notifyAll()` to unblock the blocked threads. In fact, checking to see whether there are indeed blocked clients that have become enabled is not really necessary: the client thread can always call `notifyAll()`, and any newly-enabled threads will discover that they are enabled when they re-execute the condition in their `while` loop. Checking before notifying, though, can reduce the number of needlessly unblocked threads.

You can replace `notifyAll()` with `notify()` as long as you know that the thread that will become unblocked is one whose request was enabled. Furthermore, if the action that the client executes disables any newly-enabled clients, then using `notify()` reduces the amount of needless unblocking. These two conditions hold for semaphores: only processes blocked on $P(s)$ call `wait()`. All blocked processes are enabled by a $V(s)$, but once one executes the others are no longer enabled. Hence, we use `notify()`.

4.5.2 Some Examples

We now apply the methodology to three different synchronization problems.

Readers Priority

Figure 4.14 shows the monitor written referencing the specification in Figure 4.12.

```
public class MultiClientStateMachine {

    private variable representing state machine
    state and information about requests on
    pending queue

    public synchronized Request(args) {
        update information about pending requests
        while (not my RequestEnabled) {
            try { wait(); }
            catch (java.lang.InterruptedException e) { };
            }
        compute response
        compute new state
        adjust information about pending requests
        if (there is a waiting client whose request is enabled)
            notifyAll():
        }
}
```

Figure 4.13: Java multiclient state machine.

The state machine has two components to its state: *rdrs* which is the set of readers and *wrtrs* which is the set of writers. These are sets of threads, and the value of these sets are used to determine whether a request is valid. For example, if a client requests to stop reading when the client is not in *rdrs*, then the state machine replies with an error. We could use a Java Sets class to implement sets of thread identifiers, but for simplicity we just have our monitor maintain the sizes of these two sets. The sizes are stored in `activeReaders` and `activeWriters`. This means that our monitor will consider some requests as valid that the specification will not.

The variable *pendingQ* is used only to give priority to waiting readers over waiting writers: if any reader is waiting, then *enter_write* is not enabled. So, rather than maintaining *pendingQ* explicity, we record in `waitingReaders` the number of readers on *pendingQ*.

Since we keep track only of the number of waiting readers, the only methods that need to update the pending request summary are `EnterRead` and `ExitRead`. We obtain the conditions in each `while` statement from *RequestEnabled* using the monitor's representation: for example, an *enter_read* is enabled if `activeWriters` is zero rather than $s.wrtrs$ is the empty set.

```
public class ReadPriority {
    int activeReaders, waitingReaders, activeWriters;

    public ReadPriority (int N) {
        activeReaders = 0;     // |s.rdrs|
        waitingReaders = 0;    // |{e in q: e[2] = "enter_read"}|
        activeWriters = 0;     // |s.wrtrs|
    }

    public synchronized void EnterRead () {
        waitingReaders++;
        while (activeWriters > 0)
            try { wait(); }
            catch (java.lang.InterruptedException e) { };
        waitingReaders--;
        activeReaders++;
    }

    public synchronized boolean ExitRead () {
        if (activeReaders == 0) return false;
        activeReaders--;
        if (activeReaders == 0) notifyAll();
        return true;
    }

    public synchronized void EnterWrite () {
        while (activeReaders > 0 || activeWriters > 0 || waitingReaders > 0)
            try { wait(); }
            catch (java.lang.InterruptedException e) { };
        waitingReaders--;
        activeReaders++;
    }

    public synchronized boolean ExitWrite () {
        if (activeWriters == 0) return false;
        activeWriters--;
        notifyAll();
        return true;
    }
}
```

Figure 4.14: Java readers priority class.

Election

Consider a class that represents simple elections. N threads can vote, where a vote is a boolean. The state machine has one method, `Vote(boolean)` with which a thread can cast a vote (an argument of *true* means that the thread votes *yes*; otherwise it votes *no*). When a majority of threads have voted either *true* or *false*, then the outcome of the election is determined by the majority vote. This election system is shown in Figure 4.15

We represent the clients that have voted *yes* with the variable $st.yes$ and those that have voted *no* with the variable $st.no$. Doing so allows us to ensure that a client votes no more than once. In our multiclient server, once a request is enabled it can execute and generate a reply, but a reply can't be generated until the election outcome is known. Hence, we define the operator s which is what the server state would be if all of the pending requests were executed. Thus, a request is enabled if one of $s.yes$ and $s.no$ contains a majority of the clients.

Figure 4.16 shows the Java monitor for this server. Again, for simplicity we don't maintain the set of client threads that have voted, and so in this monitor a client can vote more than once. Since the outcome is determined only by the number of *yes* and *no* votes, we combine the server state with the pending queue state and record a client's vote when it is logically added to the pending queue. With this design decision, the rest of the monitor is quite simple.

A change one might consider making to this monitor is to replace the `notifyAll()` with a `notify()`. This could be done because all of the pending client threads become enabled simultaneously and they remain enabled until they execute. Hence, any thread that is unblocked by a `notify()` will find the predicate in the `while` loop true. But, since all the threads are enabled, by using a `notifyAll()` they all unblock and will fall through the `while` loop. Hence, there is probably no reason to use a `notify()`.

Another change one might consider making is based on the same observation. Since all threads become enabled simultaneously and they remain enabled until they execute, one might be tempted to replace the `while` with an `if`. Doing so would be a bad idea: if a client thread were interrupted before there was an outcome, then the client thread (and all other blocked threads) would return an outcome of *no*. Eventually, other client threads might learn that the (correct) outcome was *yes*.

Barrier Synchronization

Imagine a politically correct sporting event in which a group of runners run around a track. Political correctness requires that no runner gets left behind. So, there is a barrier across the track. No runner may pass the barrier until every other runner has reached the barrier.

In the programming version, there is a collection of processes, each of which contains a region of code called the barrier. The rule is that, after entering its

```
──────────────────────── MODULE MCSElection ────────────────────────
EXTENDS FiniteSets

CONSTANT Thread

Request ≜ [type : "vote", voteYes : BOOLEAN ]        vote for or against

State ≜ [yes : SUBSET Thread, no : SUBSET Thread]  votes

 The next two operators count the votes in the pending queue
PendingYes(q) ≜ {q[i][1] : i ∈ DOMAIN q ∧ q[i][2].voteYes}

PendingNo(q) ≜ {q[i][1] : i ∈ DOMAIN q ∧ ¬q[i][2].voteYes}

 the combined votes: those that have voted and pending votes
 such that a thread can vote only once.
s(st, q) ≜ [yes ↦ st.yes ∪ (PendingYes(q) \ (st.yes ∪ st.no)),
            no  ↦ st.no  ∪ (PendingNo(q)  \ (st.yes ∪ st.no))]

Outcome(s) ≜ IF 2 ∗ Cardinality(s.yes) > Cardinality(Thread) THEN "yes"
               ELSE IF 2 ∗ Cardinality(s.no) > Cardinality(Thread) THEN "no"
               ELSE "undecided"

InitialState ≜ [yes ↦ {}, no ↦ {}]

NewState(t, req, st, q) ≜
  IF t ∈ (st.yes ∪ st.no) THEN st
   ELSE IF req.voteYes THEN [yes ↦ st.yes ∪ {t}, no ↦ st.no]
   ELSE [yes ↦ st.yes, no ↦ st.no ∪ {t}]

ResponseVal(t, req, st, q) ≜ [outcome ↦ Outcome(s(st, q))]

RequestEnabled(t, req, st, q) ≜ Outcome(s(st, q)) ≠ "undecided"
├──────────────────────────────────────────────────────────────────┤
VARIABLES sstate, cstate, pendingQ, iFace
INSTANCE MultiClientServer WITH Client ← Thread
└──────────────────────────────────────────────────────────────────┘
```

Figure 4.15: An election system.

barrier region for the i^{th} time, a process may not exit the region until every other process has also entered its barrier for the i^{th} time. This system is shown in Figure 4.17.

Note that a request is enabled (that is, a client c can pass the barrier) if either all the clients had arrived but c has not left (hence $c \in st$) or all the clients are at the barrier ($Length(q) = N$) and all that were to leave from the last barrier synchronization have indeed left ($st = \{\}$).

```
public class Election {
    private int N, yes, no;

    public Election (int N) {
        this.N = N;
        yes = 0;   // |s(st,pendingQ).yes|
        no = 0;    // |s(st,pendingQ).no|
    }

    public synchronized boolean Vote (boolean voteYes) {
        if (voteYes) yes++;
        else no++;
        while (2*yes <= N && 2*no <= N) {
          try { wait(); }
          catch (java.lang.InterruptedException e) { };
          }
        notifyAll();
        if (2*yes > N) return true;
        else return false;
    }
}
```

Figure 4.16: Java election class.

The Java implementation of this monitor is shown in Figure 4.18. Note that a request is enabled if its thread is in the set of thread *sstate*. If we only keep the size of *sstate* in our monitor, then we won't be able to determine if a client thread is in *sstate*. So, in this monitor we will maintain sets of client identifiers. Our monitor assigns to a client a client identifer between 0 and $N-1$.

The set of clients in the pending queue is kept in the variable `arrived`, and the set of clients in *sstate* are kept in the variable `leaving`.

Most of the monitor code consists of routines that implement the set operations. The synchronized method follows directly from the multiclient server specification. For example, the predicate in the `while` loop is the negation of the *RequentsEnabled* predicate.

If one knew that the thread enqueueing for the `wait()` was FIFO, then one might be tempted to avoid the overhead of using sets, and instead keep track of the lengths of the pending queue and the number of threads in *sstate*. If thread queueing is FIFO, then a thread is in *sstate* if it is one of the first N threads to re-enter the monitor after the `notifyAll()`. The queueing, however, may not be FIFO. For example, suppose that the implementation of `wait()` generated the following instructions:

```
──────────────── MODULE MCSBarrier ────────────────
EXTENDS FiniteSets

CONSTANT Thread

Range(f) ≜ {f[i] : i ∈ DOMAIN f}

Request ≜ {"waitfor"}

State ≜ SUBSET Thread  The threads that are leaving the barrier

InitialState ≜ {}

NewState(t, req, st, q) ≜
  IF Cardinality(st) > 0 THEN st \ {t}
   ELSE  Thread \ {t}

ResponseVal(t, req, st, q) ≜ "OK"

RequestEnabled(t, req, st, q) ≜ ∨ t ∈ st              t needs to leave
                                ∨ ∧ (st = {})         all previous left
                                  ∧ (Length(q) = N)   all have arrived
├───────────────────────────────────────────────────┤
VARIABLES sstate, cstate, pendingQ, iFace
INSTANCE MultiClientServer WITH Client ← Thread
└───────────────────────────────────────────────────┘
```

Figure 4.17: A barrier synchronization system.

```
w1: V(this.mutex);     // Release object mutual exclusion
w2: this.waitCount++;  // Increment number of waiting threads
w3: P(this.waiting);   // Wait to be notified
w4: this.waitCount--;  // Decrement number of waiting threads
w5: P(this.mutex);     // Reacquire object mutual exclusion
```

Even if semaphores are implemented with a FIFO queue, the waiting thread could take a context switch after executing $w1$ and before executing $w3$. This could lead to the threads not being enqueued in FIFO order as defined by the sequence of calls to `wait()`.

4.6 Other Synchronization Problems

We can describe any synchronization problem as a multiclient server, but the general method described above for implementing a multiclient server with a monitor may not be efficient enough. So, we want *ad hoc* solutions for those problems. Here are two examples.

```
public class Barrier {
    // An N-process barrier synchronization object.
    private int N;
    boolean arrived[], leaving[];

    private int MyID() {
        int i;
        for (i = 0; i < N; i++) if (!arrived[i]) return i;
        return -1;  // no free id, so return id
                    // that will cause problems later
    }

    private void LeavingGetsArrived () {
        int i;
        for (i = 0; i < N; i++) leaving[i] = arrived[i];
    }

    private int Size(boolean a[]) {
        int i, ct = 0;
        for (i = 0; i < N; i++) if (a[i]) ct++;
        return ct;
    }

    public Barrier (int N) {
        this.N = N;
        arrived = new boolean[N];    // clients in pendingQ
        leaving = new boolean[N];    // sstate
    }

    public synchronized void WaitAt () {
        int id = MyID();
        arrived[id] = true;
        while ( !leaving[id] && (Size(leaving) > 0) || (Size(arrived) < N ))
            try { wait(); }
            catch (java.lang.InterruptedException e) { };
        if (Size(leaving) == 0) LeavingGetsArrived();
        leaving[id] = false;
        arrived[id] = false;
        if (Size(leaving) > 0) notifyAll();
    }

}
```

Figure 4.18: Java barrier synchronization class.

4.6.1 Resource Allocation

A simple example of a resource allocation problem is the *dining philosophers* problem. A group of philosophers are sitting at a round table. Each philosopher occasionally wants to eat a bowl of rice, which requires that she obtain a pair of chopsticks. There is a single chopstick between each adjacent pair of philosphers. So, to eat, a philosopher must pick up the chopsticks on either side of her. When she is finished, she cleans each chopstick and puts it back where she found it. A hungry philosopher must wait until both of her neighbors have put down their chopsticks before she can eat.

A problem arises if all philosophers become hungry at the same time and pick up one chopstick—say, the one to their right. If each of them holds onto that one chopstick waiting for her left-hand neighbor to put down her chopsticks, then they will all starve.

In general, we consider a multiprocess system having a collection of resources. To perform a task, a process requires exclusive access to each of some subset of the resources.

To ensure mutually exclusive access to an individual resource r, we assign a semaphore $sem[r]$ to that resource. We let $sem[r]$ initially equal 1, and require each process to perform a $P(sem[r])$ operation before accessing resource r and to perform a $V(sem[r])$ operation when it is finished using r. We call such a semaphore $sem[r]$ a *lock* on resource r. The $P(sem[r])$ operation is called *acquiring* the lock, and the $V(sem[r])$ operation is called *releasing* the lock. A process that has acquired the lock but not yet released it is said to *hold* the lock.

In the dining philosophers problem, the philosophers are the processes and the chopsticks are the resources. Picking up the chopstick is acquiring the chopstick's lock; and putting down the chopstick is releasing its lock.

In general, the challenge is to avoid deadlock, which occurs if there is a cycle of processes, each waiting to acquire a lock held by the next. There is a simple general recipe for doing this, called the *locking-level* protocol. We define a partial order $\prec$ on the set of locks and require that each process must acquire locks in the order determined by $\prec$. This means that, if a process has acquired lock r, then it can attempt to acquire lock s only if $r \prec s$.

To see that this prevents deadlock, suppose we had a situation in which a set of processes were deadlocked, each waiting to acquire a lock owned by the next. Let's number the processes numbered 0 through $n-1$, and let $i \oplus 1$ equal $(i+1)\,\%\,n$. Then each process i is owns lock r_i and is waiting to acquire lock $r_{i\oplus 1}$. If the processes obey the locking-level protocol, then $r_i \prec r_{i\oplus 1}$ for each i, so we have a cycle $r_0 \prec r_1 \prec \cdots r_{n-1} \prec r_0$. This contradicts the requirement that $\prec$ be a partial order.

As an example of this protocol, let's define an order relation on the chopsticks as follows. Starting from some arbitrary chopstick, move in a counter-clockwise direction around the table numbering the chopsticks consecutively from 1 to n.

Thus, each philosopher has chopstick i to her left and chopstick $i+1$ to her right, for some i, except for one distinguished philosopher who has chopstick n to her left and chopstick 1 to her right. Let $\prec$ be the ordinary $<$ relation. The locking-level protocol requires that each philsopher first pick up the chopstick to her left and then the chopstick to her right—except for the distinguished philsopher, who picks up her chopsticks in the opposite order. Deadlock is impossible, and the philsophers are in no danger of starvation.

4.7 Problems

Problem 4.1 (a) Modify the *OneClientServer* specification to allow the server to start with its initial state an arbitrary element of some set of possible initial states. (b) Further modify the specification to make it nondeterministic. (c) Instantiate your specification to obtain a variant of the *OneClientRegister* in which the register initially has some special value that is not in *RegisterVal*. A read of the initial value returns an arbitrary value in *RegisterVal* and sets the register to that value.

Problem 4.2 Estimate the number of reachable states in the resource allocator specification as a function of the number of clients. Test your answer with TLC.

Problem 4.3 Write a complete proof that *SBInv* is an inductive invariant of the simple atomic bakery algorithm specification.

Problem 4.4 Write a rigorous correctness proof for the atomic bakery algorithm, including a proof that formula *ABInv* (defined on page 118) is an inductive invariant that implies (4.10) and a careful proof of liveness.

Problem 4.5 Show if there are at least three threads, then given any two integers i and j, there exists an execution of the bakery algorithm in which $num[t_1] = i$ and $num[t_2] = j$, for two threads t_1 and t_2.

Problem 4.6 Explain how we can, in principle, use TLC to check that a state predicate is an inductive invariant. Try using it to check that *SBInv* is an inductive invariant of the simple atomic bakery algorithm and, based on your experience, explain why it's not a practical approach.

Problem 4.7 Specify an alternation system as a multiclient server. Each client issues requests to perform an operation, and another request to signal that it has completed its operation. The system responds to a request to perform an operation when it is the requesting client's turn.

Problem 4.8 Specify a writer's priority version of *ReadersWriters*. Translate this into a Java monitor. If you decide to simplify the implementation (for example, by maintaining the count of requests in the pending queue rather than the pending queue itself), describe how the behaviors of your monitor differs from the specification.

Problem 4.9 Write two + specifications of the trivial semaphore mutual exclusion algorithm on page 103, one with a weak semaphore and one with a fair semaphore. Use TLC to show that it implements the mutual exclusion specifications of module *MutualExclusion*.

Problem 4.10 Explain (informally) why any sequence of requests and responses to a mutual exclusion server that provides starvation-free entry to the critical section can be produced by a behavior satisfying the specification in module *MCSMutex*.

Problem 4.11 Generalize the internal multiclient server specification of module *IMultiClientServer* to allow restrictions on when clients can issue requests. You will have to introduce one or more parameters that specify the restrictions. Instantiate it to obtain an internal specification of mutual exclusion that implements the specification in module *MutualExclusion* under a suitable refinement mapping.

Problem 4.12 Specify a version of the readers-writers problem, in which a reader does not begin reading if there is a waiting writer ahead of it in the waiting queue, and a writer does not begin writing if there is a waiting reader ahead of it in the waiting queue. Implement a readers-writers solution in Java using a monitor.

Problem 4.13 Other kinds of monitors differ in the priorities they give to threads that are unblocked from waiting. For example, Hoare's original monitors ensured that an unblocked thread runs immediately after being woken up (thereby temporarily forcing the notifying thread out of the object's critical section). Mesa monitors ensure that a thread unblocked from waiting will obtain the object's critical section before another thread obtains the critical section by calling a synchronized method of the object. Java, instead, gives no priority to the unblocked thread.

Write a Java monitor and program that uses this monitor to demonstrate this lack of priority. It should generate behaviors in which threads that are unblocked from executing a `wait()` contend to regain mutual excusion to the object with threads that are calling a synchronized method of the object. Your program should detect the situation when a thread calling the synchronized method wins the contention.

Problem 4.14 Give a semaphore implementation of a Java monitor. You should give the code that a thread calls before and after invoking a synchronized method, the code for a `wait()`, the code for a `notify()`, and the code for a `notifyAll()`.

Problem 4.15 Add auxiliary variables to specification *MEStarvationFree* and construct the refinement mapping under which it implements *IMCSSpec*, as sketched on page 126. Use TLC to check your solution.

Problem 4.16 In Section **??** (page **??**), we left open the problem of showing that *AltSpec* implies $\exists\, x, pc : \mathit{AltPgmSpec}$. Do this now by adding a stuttering variable to *AltSpec* and showing that the resulting specification implements *AltPgmSpec* under a refinement mapping.