

requires that the return be executed only if no user has the lock. The input condition of input function `mysid.rel` requires that the calling thread have the lock. Its output condition is vacuous, so it is non-blocking. Note that this function updates `acqd[mytid]` in the input part rather than the output part, thereby allowing another user to acquire the lock *before* `mysid.rel` returns. A system program that implements this service can assume that its input functions will be called only under these conditions; it need not check for them.

As with any service program, the input and output parts are to be executed atomically by the platform. (The entire function `MySid.rel` is not assumed to be atomic; but it is effectively atomic because only one thread has the lock at any time.) The powerful (read-modify-write) atomicity of `mysid.acq`'s return is not a problem because the lock service program is intended for analysis and not for execution. The progress assumption is expressed with “leads-to” constructs rather than fairness constructs.

Each evolution of the service program consists of an atomic execution of the main code followed by a sequence of atomic executions of input parts and output parts. At any time, any ready input or output part can be executed. Thus the program defines *all* evolutions such that (1) each user i cycles through `acq` call, `acq` return, `rel` call, and `rel` return, (2) between any two successive `acq` returns, there is a `rel` call by the user of the first `acq` return, and (3) every `acq` call eventually returns provided no user holds the lock indefinitely. The last condition comes from the progress assumption. The service program is fault-free in isolation (otherwise it would be of no use as a standard for implementation).

This lock service is more general than the simple lock because it allows requests to be served in any order, only requiring that no request is indefinitely delayed. Thus it represents many possible implementations, including the simple lock given earlier.

2.4 Simple lock implements simple lock service

We now establish that `SimpleLock(N)` implements `SimpleLockService(N)` for any N . Henceforth, for brevity, we will refer to these as *lock* and *service*. We state the implements conditions, first in terms of evolutions and then in terms of the lock and service programs. Then we establish the conditions, first using assertional reasoning with operational proofs and then illustrating assertional proofs.

The lock implements the service if the following conditions hold. (Recall that an evolution x of the lock is *safe* wrt the service if the latter has an evolution y with the same input-output sequence as x ; if y also satisfies the service's progress assumption, then x is *complete* wrt the service.)

- Safety. For every finite evolution x of the lock that is safe wrt the service:
 - Input: If extending x with a `lck.acq` call or `lck.rel` call is safe wrt the service, then the lock is ready for this input at the end of x and its execution is fault-free.
 - Output: Any step that the lock can do at the end of x is fault-free; furthermore, if that step does a `lck.acq` return or a `lck.rel` return, then extending x with this output is safe wrt the service.
- Progress. For every evolution x of the lock that is safe wrt the service: if x satisfies the lock's progress assumption then x is complete wrt the service.

The above conditions are expressed in terms of evolutions. We now express them in terms of the lock and service *programs*. This is preferable because it is clearer, it allows the use of program verification techniques (assertional reasoning in our case), and it can be mechanically tested. The first step is to “invert” the service program, i.e., interchange its inputs and outputs, resulting in a program that provides the most general environment that the lock can expect. This program, called `SimpleLockServiceInverse`, or *service inverse* for short, is shown in figure 2.3. The next step is to define a program that executes the lock and service inverse concurrently. The program, called Z , is shown in figure 2.4. Z is a “closed” program, meaning it has no inputs or outputs with its environment; so calls to systems of Z are only made by other systems of Z . Then the lock implements the service iff the following hold:

- Safety: Z is fault-free (i.e., has no faulty evolutions).
- Progress: Z satisfies `lsi`'s progress condition.

For ease of reference, `SimpleLock` and `SimpleLockServiceInverse` are shown together in figure 2.5. Before proving the safety and progress conditions, we identify code chunks in Z that are effectively atomic; the larger the chunks, the simpler the proof. Only threads 0 through $N-1$ can execute code in `lsi`, because `lsi` has no input functions and its output functions require `mytid` to be in $[0..N-1]$. System `lck` has only one local thread, `lck`. So Z has only these $N+1$ threads.

```

service SimpleLockService(int N) {
service SimpleLockServiceInverse(Sys* lck, int N) { // lck: lock system
  ic {lck ≠ mysid, N ≥ 1}
  boolean[N] acqd ← false;          // acqd[i] is true iff i has lock
  return mysid;
  // end main

  input void mysid.acq() {
  output doAcq() {
    ie oc {(mytid in [0..N-1]) and (not acqd[mytid])}
    lck.acq(); // added call
  b1: ee ic {forall(j in [0..N-1]: not acqd[j])}
    acqd[mytid] ← true;
  }

  input void mysid.rel() {
  output doRel() {
    ie oc {(mytid in [0..N-1]) and acqd[mytid]}
    acqd[mytid] ← false;
    lck.rel(); // added call
    ee ic {true}
  }

  atomicity assumption {input parts and output parts};

  progress assumption condition {
    forall(i in [0..N-1]: (i at mysid.rel.oc) leads-to (not i in mysid.rel));
    forall(i in [0..N-1]: acqd[i] leads-to (not acqd[i]))
      ⇒ forall(i in [0..N-1]: (i at b1) leads-to acqd[i]);
  }
}

```

Figure 2.3: SimpleLockServiceInverse, obtained from SimpleLockService after deletions and additions (underlined).

```

system Z(int N) {
  ic {N ≥ 1}
  inputs(); outputs(); // closed sytem
  Sys* lck ← startSystem(SimpleLock(N)); // lock
  Sys* lsi ← startSystem(SimpleLockServiceInverse(lck, N)); // service inverse
}

```

Figure 2.4: Program of lock and inverse lock service.

Variable `acqd[i]` is accessed only by thread `i`, so it is not shared (i.e., not accessed by multiple threads simultaneously). Variable `xreq[i]` is shared by threads `i` and `lck`. Variable `xacq` is shared by all the threads. Recall that a code chunk is effectively atomic if all its shared variable accesses appear in one atomic statement. Thus the following code chunks are atomic:

- `Z.main`: consisting of `lck`'s main and `lsi`'s main. (`Z` starts `lsi` only after `lck`'s main code returns its sid. Threads `lck.main` and `lck.t` do not access anything simultaneously.)
- In function `serve`, each of the statements `a0`, `a1`, `a2`, and `a3`-iteration accesses a shared variable and hence cannot be combined. But statement `a4` has no shared variable, so it can be combined with the preceding atomic step;

```

system SimpleLock(int N) {
  ic {N ≥ 1}
  boolean[N] xreq ← false;
  boolean xacq ← false;
  int xp ← 0;
  Thread* t ← startThread(serve());
  return mysid;

  function void serve() {
    forever do {
a0: if • xreq[xp] then {
a1:   • xacq ← true;
a2:   • xreq[xp] ← false;
a3:   while • xacq do skip;
      }
a4: if xp = N-1 then xp ← 0 else xp ← xp+1;
      }
    }

  input void mysid.acq() {
    ic {mytid in [0..N-1]}
a5: xreq[mytid] ← true;
a6: while • xreq[mytid] do skip;
    return;
  }

  input void mysid.rel() {
    ic {mytid in [0..N-1]}
a7: xacq ← false;
    return;
  }

  atomicity assumption {reads and writes
    of xacq, xreq[0], ..., xreq[N-1]}

  progress assumption {weak fairness for
    every thread}
}

service SimpleLockServiceInverse(Sys* lck, int N) {
  ic {lck ≠ mysid, N ≥ 1}
  boolean[N] acqd ← false;
  return mysid;

  output doAcq() {
    • oc {(mytid in [0..N-1]) and (not acqd[mytid])}
    lck.acq();
b1: ic {forall(j in [0..N-1]: not acqd[j])}
    acqd[mytid] ← true;
  }

  output doRel() {
    • oc {(mytid in [0..N-1]) and acqd[mytid]}
    acqd[mytid] ← false;
    lck.rel();
    ic {true}
  }

  atomicity assumption {input parts and output parts};

  progress condition {
    forall(i in [0..N-1]:
      (i at mysid.rel.oc) leads-to (not i in mysid.rel));
    forall(i in [0..N-1]: acqd[i] leads-to (not acqd[i]))
      ⇒ forall(i in [0..N-1]: i at b1 leads-to acqd[i]);
  }
}

```

Figure 2.5: Component programs of $Z(N)$; the “•”s mark the control points after accounting for effective atomicity.

i.e., a4 can be executed atomically with a0 when $xreq[xp]$ is false and with a3 when $xacq$ is false.)

- doAcq() call chunk: from doAcq()'s output part to the end of a5 in input function acq(). (All shared variable accesses are in a5.)
- acq() return chunk: from the a6-iteration with $xreq[mytid]$ false to the end of doAcq()'s input part. (All shared variable accesses are in a6.)
- doRel() chunk: from the output part to the end of the input part, including the execution of the rel() call. (All shared variable accesses are in a7.)

Thus for analysis, one can assume that the only possible control points are those marked by the “•”s in figure 2.5; i.e., the code chunks between them are executed atomically. Now back to the task of showing that Z is fault-free and satisfies $1s_i$'s progress condition.

Proof of safety condition

The goal is to show that Z has no faulty evolutions. System lck is fault-free in isolation (from A_0). System lsi is fault-free in isolation, because the lock service, and hence its inverse, are fault-free. Because lck and lsi are fault-free in isolation, it suffices to show that whenever one does an output, the other is ready for the corresponding input, i.e., the input condition holds. There are four interactions to consider: $lck.acq$ call, $lck.rel$ call, $lck.acq$ return, and $lck.rel$ return. We cover each in turn next. Also, henceforth we omit the lck and lsi prefixes when there is no ambiguity.

Input function acq is called only by $doAcq$. So we want the former's input condition to hold whenever the latter's output condition holds, i.e., $doAcq().oc \Rightarrow acq().ic$ to be invariant. Substituting for the oc and ic expressions, this predicate becomes:

$$A_1 : ((mytid \text{ in } [0..N-1]) \text{ and } (\text{not } acqd[i])) \Rightarrow (mytid \text{ in } [0..N-1])$$

$Inv A_1$ holds trivially because A_1 's consequent (the part to the right of the \Rightarrow) is the same as the first conjunct in A_1 's antecedent (the part to the left of the \Rightarrow). So A_1 holds regardless of what Z does, even in unreachable states of Z .

Input function rel is called only by $doRel$. Exactly as with acq , we want the former's ic to hold whenever the latter's oc holds, which means the following should be invariant, which it is exactly as in the case of A_1 :

$$A_2 : ((mytid \text{ in } [0..N-1]) \text{ and } acqd[i]) \Rightarrow (mytid \text{ in } [0..N-1])$$

Output function $doAcq$'s return happens only when a (guest) thread i is on statement $a6$ in lck and $xreq[i]$ is false. So we want the following to be invariant:

$$A_3 : ((i \text{ on } a6) \text{ and } (\text{not } xreq[i])) \Rightarrow \text{forall}(j \text{ in } [0..N-1]: \text{not } acqd[j])$$

Output function $doRel$'s return is fault-free because its input condition is vacuous.

What remains is to show that A_3 is invariant. Let's take a closer look at the evolution of Z . After the initial step (i.e., $Z.main$), $xacq$ and the entries of $xreq$ and $acqd$ are false, and thread $lck.t$ starts repeatedly executing $a0;a4$. When a thread i executes $doAcq$ call chunk, it sets $xreq[i]$ to true and waits on $a6$. When thread $lck.t$ finds $xreq[xp]$ true (in $a0$), it executes $a1$ and $a2$ and waits on $a3$. Until $lck.t$ executes $a2$, every $acqd$ entry is false. Thread xp , which would be waiting on $a6$, finds $xreq[xp]$ false and executes acq return chunk, which sets $acqd[xp]$ to true (in $doAcq$'s input part). Thread $lck.t$ remains on $a3$ until thread xp executes $doRel$ chunk, setting $xacq$ and $acqd[xp]$ to false. At this point, $lck.t$ executes $a3$ and $a4$ and returns to $a0$.

So when $lck.t$ is not on $a3$, nobody has the lock. And when $lck.t$ is on $a3$, thread xp has the lock or is about to get it, and nobody else has the lock. In terms of assertions, the following are invariant:

$$A_4 : (\text{not } lck.t \text{ on } a3) \Rightarrow \text{forall}(j \text{ in } [0..N-1]: \text{not } acqd[j])$$

$$A_5 : (lck.t \text{ on } a3) \Rightarrow (acqd[xp] \text{ or } ((\text{not } acqd[xp]) \text{ and } (xp \text{ on } a6) \text{ and } (\text{not } xreq[xp]))) \\ \text{and } \text{forall}(j \text{ in } [0..N-1], j \neq xp: \text{not } acqd[j])$$

Now let's get back to $Inv A_3$. Initially A_3 holds (vacuously, because i is not on $a6$). Thread i comes to $a6$ by executing $doAcq$ call chunk; this sets $xreq[i]$ to true and hence preserves A_3 vacuously. Variable $xreq[i]$ becomes false when $lck.t$ executes $a2$ with xp equal to i ; at this point no one has the lock (because A_4 holds non-vacuously before this step) and hence A_3 's consequent holds. While thread i remains on $a6$, no other thread can set its $acqd$ entry to true (because that would violate $Inv A_5$). When thread i leaves $a6$, A_3 holds vacuously. So $Inv A_3$ holds and we are done.

Proof of progress condition

System lsi has two progress conditions. The first condition is that every $doRel$ call terminates, which holds trivially because the $doRel$ chunk has no loops and is executed with weak fairness (from lck 's progress assumption). The second progress condition is an implication. Its antecedent is B_0 below. Its consequent is B_1 below, where "i at $b1$ " has been replaced by "i on $a6$ " because acq 's return chunk is effectively atomic.

$$B_0 : \text{forall}(i \text{ in } [0..N-1]: acqd[i] \text{ leads-to } (\text{not } acqd[i]))$$

$$B_1 : \text{forall}(i \text{ in } [0..N-1]: (i \text{ on } a6) \text{ leads-to } acqd[i])$$

We have to establish that Z satisfies B_1 assuming B_0 . Suppose thread $lck.t$ is at a_0 with $xp = j$ and $xreq[j]$ true. At this point, thread j is waiting on a_6 and $xreq[j]$ stays true (because only $lck.t$ can make it false). Thus $lck.t$ eventually executes statements $a_0..a_2$ (because of weak fairness), setting $xreq[j]$ to false and $xacq$ to true, and starts waiting on a_3 . Hence thread j , which is still on a_6 , eventually executes the acq return chunk, at which point $acqd[j]$ becomes true. Thus the following holds:

$$B_2 : ((lck.t \text{ on } a_0) \text{ and } (xp = j) \text{ and } xreq[j]) \text{ leads-to } ((lck.t \text{ on } a_3) \text{ and } (xp = j) \text{ and } acqd[j])$$

If B_2 's rhs (the part to the right of the *leads-to*) holds, then thread j eventually executes $doRel$ (because of assumption B_0), making $xacq$ false. It stays false (because only $lck.t$ can make it true), and so $lck.t$ eventually leaves a_3 . Thus the following holds (recall that a_4 's execution is combined with a_3 's):

$$B_3 : ((lck.t \text{ on } a_3) \text{ and } (xp = j) \text{ and } acqd[j]) \text{ leads-to } ((lck.t \text{ on } a_0) \text{ and } (xp = (j+1) \bmod N))$$

B_2 and B_3 imply the following:

$$B_4 : ((lck.t \text{ on } a_0) \text{ and } (xp = j)) \text{ leads-to } ((lck.t \text{ on } a_0) \text{ and } (xp = (j+1) \bmod N))$$

Thus xp keeps increasing modulo- N . Hence for every thread j such that j on a_6 , thread lck eventually comes to a_0 with $xp = j$, where it finds $xreq[j]$ to be true, and so eventually $acqd[j]$ becomes true (from B_2). This establishes B_1 , and we are done.

Assertional proof of $Inv A_4$

The previous analysis illustrated assertional reasoning: assertions were used to express desired and intermediate properties, and they were shown to hold by operational arguments. We now illustrate an assertional proof for $Inv A_4$, that is, a proof consisting of applications of proof rules. We use only one proof rule:

- **Invariance induction rule:** Z satisfies $Inv A_4$ if there is a predicate C satisfying the following:

- (1) Z 's initial atomic step (i.e., $Z.main$) establishes C .
- (2) For every atomic step e of Z : executing e in a state satisfying C results in a state that satisfies C .
- (3) $C \Rightarrow A_4$ holds.

Given a predicate C , it is easy to check whether the conditions of the rule hold. Finding a suitable C may not be easy. The natural way to do this is to consider atomic steps that do not preserve A_4 , and then identify what more needs to hold so that they preserve A_4 .

We first note that if we don't want $acqd[j]$ to hold in a situation, then we also don't want j to be about to get the lock, i.e., $((j \text{ on } a_6) \text{ and } (\text{not } xreq[j]))$ to hold, otherwise thread j can execute the acq return chunk, making $acqd[j]$ true. Let $z(j)$ denote that j does not have the lock nor is about to get it. Formally,

$$z(j) : acqd[j] \text{ or } ((j \text{ on } a_6) \text{ and } (\text{not } xreq[j]))$$

We will establish the invariance of the following stronger version of A_4 :

$$C_1 : (\text{not } lck.t \text{ on } a_3) \Rightarrow \text{forall}(j \text{ in } [0..N-1] : \text{not } z(j))$$

In order for C_1 to hold when $lck.t$ leaves a_3 , the following should be invariant:

$$C_2 : ((lck.t \text{ on } a_3) \text{ and } (\text{not } xacq)) \Rightarrow \text{forall}(j \text{ in } [0..N-1] : \text{not } z[j])$$

When $lck.t$ is on a_3 and $xacq$ is true, we expect that thread xp , and no one else, has the lock or is about to get it.

$$C_3 : ((lck.t \text{ on } a_3) \text{ and } xacq) \Rightarrow (z(xp) \text{ and } \text{forall}(j \text{ in } [0..N-1], j \neq xp : \text{not } z(j)))$$

Conversely, when thread i has the lock or is about to get it, we expect thread $lck.t$ to be waiting on a_3 for i :

$$C_4 : z(i) \Rightarrow ((lck.t \text{ on } a_3) \text{ and } (xp = i) \text{ and } xacq)$$

The following are also needed (e.g., C_5 is needed to preserve C_4 after $lck.t$ does a_2):

$$C_5 : (lck.t \text{ on } a_2) \Rightarrow xacq$$

$C_6 : (\text{lck.t on a1..a2}) \Rightarrow (\text{xreq[Exp]} \text{ and } (\text{xp on a6}))$
 $C_7 : \text{forall}(j \text{ in } [0..N-1]: \text{xreq}[j] \Rightarrow (j \text{ on a6}))$

It's easy to check that the conjunction of $C_1 - C_7$ is a suitable predicate C for the invariance induction rule. Consider the initial step, $Z.\text{main}$: it establishes the consequent of C_1 and C_2 , and falsifies the antecedent of the others. Consider step $a0$ by lck.t : it preserves C_1 (i.e., C_1 before the step ensures that C_1 holds after the step); it preserves C_4 and C_7 ; it establishes C_2, C_3, C_5 vacuously; it establishes C_6 (from C_7 holding prior to the step). Each of the remaining atomic steps can be similarly shown to preserve $C_1 - C_7$. Note that one can *check* this without having any global understanding of program Z . All that is needed is to understand the individual constructs of program Z and predicate C .

Assertional proof of B_2

We now illustrate an assertional proof for the progress assertion B_2 . The following proof rules are used:

- **Leads-to via thread x rule:** Z satisfies P leads-to Q if the following hold:
 - (1) The execution of any atomic step in any state satisfying P preserves P or establishes Q .
 - (2) Starting from any state satisfying P , the execution of the next atomic step of thread x establishes Q .
- **Leads-to closure rules:**
 - P leads-to Q holds if P leads-to R and R leads-to Q hold.
 - P leads-to Q holds if $\text{Inv}(P \Rightarrow Q)$ holds.

The following lines establish that Z satisfies B_2 . Each line below states an assertion (at the left) and a proof rule application (at right) that establishes the assertion. B_2 follows from the closure of D_0 through D_5 .

$D_0 : ((\text{lck.t on a0}) \text{ and } (\text{xp} = j) \text{ and } \text{xreq}[j]) \text{ leads-to } ((\text{lck.t on a1}) \text{ and } (\text{xp} = j))$ [via lck.t]
 $D_1 : ((\text{lck.t on a1}) \text{ and } (\text{xp} = j)) \text{ leads-to } ((\text{lck.t on a2}) \text{ and } (\text{xp} = j))$ [via lck.t]
 $D_2 : ((\text{lck.t on a2}) \text{ and } (\text{xp} = j)) \text{ leads-to } ((\text{lck.t on a3}) \text{ and } (\text{xp} = j))$ [via lck.t]
 $D_3 : ((\text{lck.t on a3}) \text{ and } (\text{xp} = j)) \text{ leads-to } ((\text{lck.t on a3}) \text{ and } (\text{xp} = j))$ [D_1, D_2, D_3 , closure]
 $D_4 : ((\text{lck.t on a3}) \text{ and } (\text{xp} = j)) \text{ leads-to } ((\text{lck.t on a3}) \text{ and } (\text{xp} = j) \text{ and } (j \text{ on a6}))$ [$\text{Inv } C_6, D_3$, closure]
 $D_5 : ((\text{lck.t on a3}) \text{ and } (\text{xp} = j) \text{ and } (j \text{ on a6})) \text{ leads-to } ((\text{lck.t on a3}) \text{ and } (\text{xp} = j) \text{ and } \text{acq}[j])$ [via j]

2.5 Producer and consumer using lock service

So far we have seen one role of the lock service, that is, to define the correctness of any valid lock implementation. We now illustrate the other role of the lock service, specifically, to serve as a lock in the design of a larger distributed system that makes use of locks. We revisit the producer-consumer-lock program, but this time using the lock service in place of the lock, as shown in Figure 2.6. This has two advantages. First, the modified program is easier to analyze because the lock service program is simpler than the lock program. Second, any correctness property satisfied by the modified program is preserved when the lock service is replaced by any valid implementation, including the simple lock program.

```

system ProdCons() {
  ic { true }
  Thread* tProd ← Tid(0);
  Thread* tCons ← Tid(1);
  Sys* lck ← startSystem(SimpleLockService(2));
  Sys* cons ← startSystem(Consumer(lck, tCons)); // sid = cons
  Sys* prod ← startSystem(Producer(lck, cons, tProd)); // sid = prod
}

```

Figure 2.6: Producer-consumer program.

```

system Producer(Sys* lck, Sys* cons, Thread* tProd) {
  ic {lck ≠ cons ≠ tProd ≠ mysid ≠ null}
  integer np ← 0; // # items produced
  startThread(tProd, produce());
  return mysid;
  // end main

  function void produce() {
    forever do {
      np ← np+1;
      lck.acq(); // acquire lock
      cons.deliver(); // item to consumer
      lck.rel(); // release lock
    }
  }

  atomicity assumption [ ]; // none; uses lck instead

  progress assumption {weak fairness for threads};
}

```

Figure 2.7: Producer program.

```

system Consumer(Sys* lck, Thread* tCons) {
  ic {lck ≠ tCons ≠ mysid ≠ null}
  integer nc ← 0; // # items consumed
  integer cache ← 0; // # items in cache
  startThread(tCons, consume());
  return mysid;
  // end main

  function void consume() {
    forever do {
      lck.acq(); // acquire lock
      while cache > 0 do { // consume cache
        nc ← nc+1;
        cache ← cache - 1;
      };
      lck.rel(); // release lock
    }
  }

  input void mysid.deliver() {
    ic {true}
    cache ← cache+1;
    return;
  }

  atomicity assumption [ ]; // none; uses lck instead

  progress assumption {weak fairness for threads};
}

```

Figure 2.8: Consumer program.

In keeping with the constraints of the simple lock service, the threads using the lock get tids 0 and 1 (as indicated by constructs `Tid(0)` and `Tid(1)` in program `ProdCons`, and by the first parameter in the `startThread(...)` calls in the producer and consumer programs). The producer and consumer programs are shown in Figures 2.7 and 2.8. The producer repeatedly produces an item and gives it to the consumer by calling the consumer's input function `deliver`. The consumer repeatedly empties its cache of items and consumes them. The producer and consumer use the lock service to ensure that items are not being delivered while the cache is being consumed. Because of the lock, neither program needs any atomicity assumption.

We expect `ProdCons` to satisfy various desired properties, for example, no faulty evolutions, no buffer underflow ($Inv\ cons.nc \leq prod.np$), and every item produced to be eventually consumed ($prod.np \geq k$ *leads-to* $cons.nc \geq k$). It is easy to prove that `ProdCons` satisfies these properties.

Here, we outline an operational proof of the first desired property, i.e., that every possible evolution of `ProdCons` is fault-free. Each component system is fault-free in isolation. The `prod-lck` interactions happen only when the input side is ready because `prod` alternates requests with releases, starting with a request. The same holds for the `cons-lck` interactions. The same holds for the `prod-cons` interactions because `cons.deliver` is always ready. The proof of the other two properties is left as an exercise.

Because of compositionality, any correctness property that holds for program `ProdCons` also holds if the lock service is replaced by any system that implements the service, including `SimpleLock(2)`.

2.6 Concluding remarks

This section has illustrated all the main pieces of SESF. We presented a simple lock and a lock service. We defined what it means for the simple lock to implement the lock service, and showed that the appropriate conditions hold. We also showed how the lock service adequately represents the lock implementation in any environment.

As mentioned earlier, to keep the example simple our lock service assumed user ids from $[0..N-1]$ where N is fixed at lock creation. Whereas the usual lock service provided in a programming language is not constrained in the ids of the users. This extension is left as an exercise. The lock implementation here has a local thread, namely t , that constantly checks for a request. In later chapters we will see lock implementations without any local threads.

The implementation here also makes users do busy waiting; i.e., when a user thread requests the lock, it busy waits until it acquires the lock. This is unavoidable given an underlying platform without any built-in synchronization constructs (e.g., a typical bare hardware platform). Most programming languages that provide locks execute on platforms that provide non-busy waiting (i.e., without consuming processor cycles), and their lock implementation would, of course, exploit that. These languages also usually provide synchronization constructs that allow threads to sleep and be awakened. We will see two such constructs, namely condition variables and semaphores, in chapter 4.

2.7 Exercises

1. Change the order of `a1` and `a2` in `SimpleLock`, that is, do `xacq ← true` after `xreq[xp] ← false`. Does this revised program implement the simple lock service. If you answer no, give a counter-example evolution.
2. Show that program `ProdCons` (in figure 2.6) satisfies the following assertions (the third one says that `np` keeps increasing):
 - a. $Inv \text{ cons.nc} \leq \text{prod.np}$
 - b. $\text{prod.np} \geq k \text{ leads-to } \text{cons.nc} \geq k$
 - c. $\text{prod.np} \geq k \text{ leads-to } \text{prod.np} > k$

Remember to first identify the effectively atomic steps of `ProdCons`. (Hint: Is `cons.deliver` effectively atomic?)

3. Here is a modified `ProdCons` program in which the lock service has other users in addition to the producer and consumer.

```
system ProdCons2(int N) {
  ic {N > 2}
  inputs{lck.acq(i), lck.rel(i): i in [2..N-1]};
  Sys* prod ← Sid(0);
  Sys* cons ← Sid(1);
  Sys* lck ← startSystem(SimpleLockService(N));
  startSystem(cons, Consumer(lck));
  startSystem(prod, Producer(lck, cons));
}
```

Note that the environment of this modified program has access to the inputs and outputs of the lock service corresponding to users other than the producer and consumer.

- a. Under what conditions will any safety property of `prod` and `cons` in `ProdCons` also hold in `ProdCons2`?
 - b. Under what conditions will any progress property of `prod` and `cons` in `ProdCons` also hold in this `ProdCons2`?
4. Define a lock service in which there are no constraints on when a user can request the lock or release the lock. In particular, a user can request the lock even if it already has it, and a user can release the lock even if it does not have the lock. The lock service should ignore invalid requests and otherwise provide the usual lock service.
 5. Define a lock service in which the lock service can take away the lock from a user. Specifically: (1) add an output function that does a “forced release” output `s.fRel(lck)`, where `s` is the *system id* of the user with the lock; (2) add a parameter to the lock request input by which the user provides its system id when it requests the lock; and (3) make all the modifications you think necessary. (Such a lock service would be relevant in situations where conflicts between users are settled by aborting one or more users. Consider a database of objects, each with its own lock. A database transaction updates a sequence of objects, obtaining their locks as needed. If two transactions conflict, i.e., both are half-way and each has done updates inconsistent with the other’s, then at least one has to be aborted, in which case the locks it currently owns have to be forceably released.)
 6. Obtain a lock that implements the lock service in problem 5. Your lock program should assume the same platform as that of simple lock.
 7. Obtain an implementation of `SimpleLockService(N)` that runs on the same platform as `SimpleLock` but ensures that the storage required is proportional to the number of requesting users rather than the total number of potential users. Such an implementation would handle an unlimited number of potential users with general ids. (Hint: A natural approach is to keep the status of the requesting users in a list (instead of in array `xreq[N]`). The tricky part is ensuring that the list is consistently updated when requests arrive simultaneously, because adding a request to the list may not be atomic since it now involves more than just updating an array location.)