

example, if a user calls release lock when it does not have the lock, this can result in two users having the lock at the same time (how?). We will establish later that simple lock works correctly provided the users behave properly.

For now, we establish some basic properties of simple lock. First, simple lock is fault-free *in isolation*, that is, assuming its inputs are called only when ready. Its only possible undefined operations are out-of-bound references to `xreq` in functions `mysid.acq` and `serve`. The former is not possible because of `mysid.acq`'s input condition. The latter is not possible because `xp` is initially in $[0..N-1]$ and is changed only by `a4`, which keeps `xp` in this range. Second, the main code is effectively atomic, because during its execution there is no other guest thread in the system (no input can be called because the environment does not yet know this system's `sid`) and thread `t`, if it has started, does only reads until an input occurs.

Summarizing, the following holds:

A_0 : (SimpleLock is fault-free and its main code is effectively atomic)

2.3 Simple lock service

```

service SimpleLockService(int N) { // lock service for users 0, ..., N-1
  ic {N ≥ 1} // program input condition; for analysis only
  boolean[N] acqd ← false; // acqd[i] is true iff i has lock
  return mysid;
  // end main

  input void mysid.acq() {
    ic {(mytid in [0..N-1]) and (not acqd[mytid])}
  b1: oc {forall(j in [0..N-1]: not acqd[j])}
    acqd[mytid] ← true;
    return;
  }

  input void mysid.rel() {
    ic {(mytid in [0..N-1]) and acqd[mytid]}
    acqd[mytid] ← false;
    oc {true}
    return;
  }

  atomicity assumption {input parts and output parts}

  progress assumption {
    forall(i in [0..N-1]: (i at mysid.rel.oc) leads-to (not i in mysid.rel)); // release call returns
    forall(i in [0..N-1]: acqd[i] leads-to (not acqd[i])) // if no one holds the lock forever
    ⇒ forall(i in [0..N-1]: (i at b1) leads-to acqd[i]); // then every request is satisfied
  }
}

```

Figure 2.2: Simple lock service program.

The lock service program, called `SimpleLockService`, is shown in figure 2.2. The program parameters, program input condition, and main code are as in `SimpleLock`, except that the main code defines an array `acqd` that indicates the user (if any) that has acquired the lock.

The input condition of input function `mysid.acq` is more constrained than in the simple lock. In addition to requiring that the calling thread's `tid`, `mytid`, is in $[0..N-1]$, it requires that the calling thread not have the lock. Also, the calling thread would not have a request already pending (otherwise `mytid` would be at `b1`). The output condition of `mysid.acq`

```

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.

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. The last is obviously fault-free because its input condition (in `doRel`) is just true. We cover each of the remaining three 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., $\text{doAcq}().\text{oc} \Rightarrow \text{acq}().\text{ic}$ to be invariant. Substituting for the `oc` and `Ic` expressions, this predicate becomes:

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

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