Chapter 14. A Verification Model of a Telephone Switch

> "For when the actual facts show a thing to be impossible we are instantly convinced that it is so."
>
> —(Polybius, The Histories, Book XII)

When faced with a software verification problem, it is often tempting to build a model that is as close to reality as possible. If an implementation exists, the temptation is to duplicate its functionality as faithfully as possible within the language of the model checker used. If only a design exists, the attempt can be to build a trial implementation for verification purposes. The purpose of this chapter is to show that this is not the best approach. The proper management of computational complexity is a key issue in all but the simplest applications of formal verification, and more often than not determines the success or failure of the attempt.

General Approach

The intuitive approach to software verification sketched here should be contrasted with the standard approach that one routinely takes in physics or mathematics. When one wants to analyze, say, the structural integrity of a bridge or a building, one does not start with a description of the structure that is as close to reality as possible. The best approach is to start with the simplest possible description of the structure that can capture the essential characteristics that must be analyzed. The reason is proper management of complexity. Even when mathematics is sufficiently expressive to describe reality in its minutest details, doing so would not only be a laborious task, it would not help in the least to simplify analytical chores. Computations on highly detailed descriptions, by man or by machine, can become so complex and time−consuming that the end results, if obtainable at all, become subject to doubt.

Keep it Simple

The purpose of a model checking exercise is not to build and analyze verification models that are as detailed as possible: it is the opposite. The best we can do is to find and build the smallest sufficient model to describe the essential elements of a system design. To construct that model, we attempt to simplify the problem, eliminating elements that have no direct bearing on the characteristics we want to verify. There is no universal recipe for how this can be accomplished. What works best is almost always problem dependent. Sometimes the smallest sufficient model can be constructed by generalizing a problem, and sometimes it requires specializing a problem.

The hardest problem of a verification project is to get started. The best advice that can be given here is to make a deliberate effort to start simple, perhaps even with a coarser abstraction than may seem justified. Then slowly evolve the verification model, and the corresponding correctness requirements, until sufficient confidence in the correctness of the design has been established. It is only reasonable to invest considerable resources into a verification at the very last phase of a project—to perform a final and thorough check to make sure that nothing of importance was missed in the earlier steps.

Throughout most of a verification effort, a tool like SPIN should be used in a mode where one can get instantaneous feedback about relatively simple descriptions of the design problem. Slowly, the description can become more refined, and as our confidence in its accuracy grows, our willingness to spend a little more time on each verification task can grow.

Managing Complexity

On a reasonably modern machine SPIN verifications should not consume more than a few seconds during the initial development of a verification model, and no more than a few minutes during the latter stages of verification. In very rare cases it may be necessary to spend up to a portion of an hour on a thorough verification in a final check, but this should be a very rare exception indeed.

To summarize this approach:[1]

> [1] This approach to verification was first articulated by Prof. Jay Strother Moore from the University of Texas at Austin, when describing the proper use of interactive theorem provers.

- Start simple. Try to find the smallest sufficient model that can express something interesting about the problem you are trying to solve.

  Check the initial model thoroughly. More often than not you will be surprised that what you believed to be trivially true in the simplified world is not true at all. The typical reasons are small misjudgements in the development of the model, or subtle misunderstanding in the formulation of the properties checked for.
- Evolve the model and, if possible, its correctness properties step by step. Keep each incremental step small, and repeat the checks at each step. Stop when the complexity grows too rapidly and rethink the last change made. Try to find alternatives that can reduce the complexity. The numbers of asynchronously executing processes and the size of message buffers are the two most important sources of complexity in SPIN models, so try to keep these as small as possible at first.
- Keep the verification tool on a short leash. Do not spend more than a few seconds on initial verifications until you have developed sufficient confidence that what you ask the tool to verify is actually what you are interested in.

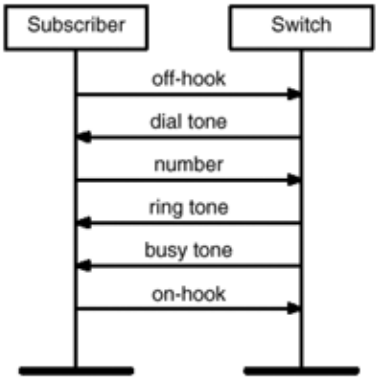
To illustrate this approach, we will discuss the development of a SPIN verification model for a significant fragment of an important and very well−known type of distributed system: a telephone switch. The problem context is familiar enough that many have attempted to build models like the ones we will discuss. Many of these attempts have ended in either a lost battle with the fundamental complexity of the underlying problem, or the adoption of simplifying but rather unrealistic assumptions about how a phone system actually works. We will try to do better here.

Modeling a Switch

The telephone system is so familiar to us that few of us realize that the underlying behavior can be phenomenally complex. Much of this complexity is due to the addition of feature behavior. Features such as three−way calling, call waiting, and call forwarding can interact in often unforeseen ways. Making sure that all such possible interactions comply with the relevant standards is a non−trivial task, even for experts. The problem is still quite non−trivial if we trim it down to its bare essence: providing support for only basic POTS (Plain Old Telephone Service) calls.

The normal dialogue for a POTS call looks simple. After taking the receiver off−hook, the subscriber hears a dial tone. This is the signal for the subscriber to dial a number. If that number is valid, the subscriber can expect to hear either a ring tone or a busy tone. If the number is invalid, an error tone or a busy tone will be the result. After a while, a ring tone can disappear when the call is answered, or it can turn into a busy tone when the maximum ring−time is exceeded. At any time during this sequence, the subscriber can abort the call and return the phone on−hook. This scenario is illustrated in Figure 14.1.

Figure 14.1. Typical Scenario for a POTS Call



Before reading on, put this book aside and attempt to build a small SPIN model that captures the interactions between a subscriber and a switch, as just sketched, restricting to outgoing calls for simplicity. Then do some verifications with SPIN to discover all the things that can go wrong with your model.

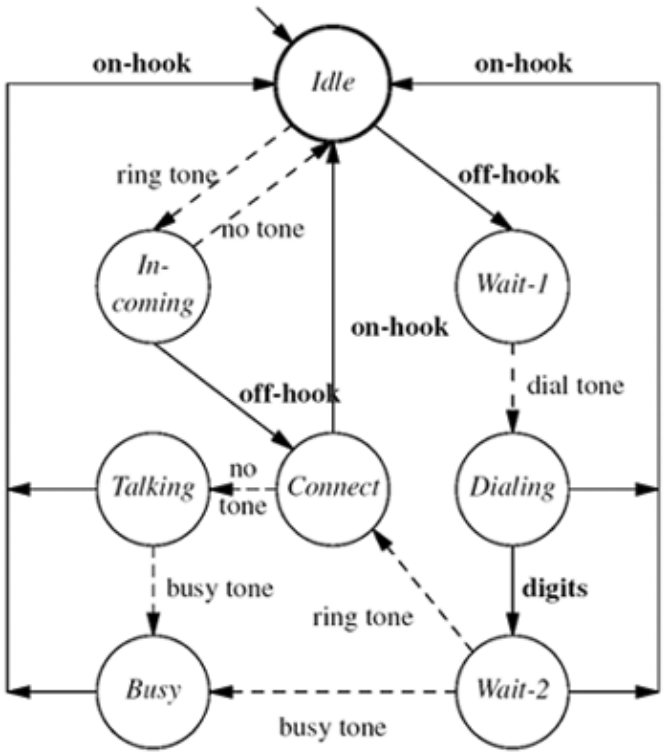
Subscriber Model

To develop a model that can reproduce the behavior from Figure 14.1, we will minimally have to model two entities: subscribers and switches. Because our focus will be on verifying properties of switch behavior, we should try to keep the number of assumptions we make about the behavior of subscribers as small as possible. We do not need to know, for instance, when or why subscribers place calls, why they hang up or why they sometimes fail to hang up. All we need to know about subscribers is what they can do that is visible to the switch. The set of things that a subscriber can do that is visible to the switch is blissfully small: the subscriber can lift the receiver off−hook, or return it on−hook. In between those two actions the subscriber can dial digits and flash the hook [2], and that is all we need to know.

> [2] A flash−hook signal can have special significance for certain call features, such as, for instance, three−way calling. We will discuss this feature later in the chapter.

Let us first consider the sample subscriber model from Figure 14.2. It tries to capture the behavior of a fairly reasonable subscriber, responding to the tones that may be generated by the switch. Some of these tones are generated in response to subscriber actions and some can be generated seemingly spontaneously by the switch, for instance, to alert the subscriber to incoming calls.

Figure 14.2. Initial Behavior Model of a POTS Subscriber (Solid arrows refer to events triggered by the subscriber, and dashed arrows refer to signals that are generated by the switch.)

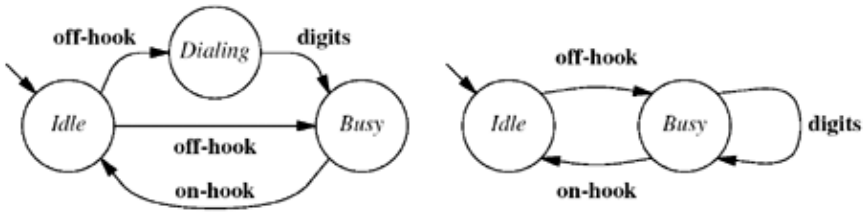


It is important to realize at this point that the subscriber model from Figure 14.2, though quite persuasive, is inadequate for our verification task. For our current purpose, the subscriber model is meant to capture the minimal set of assumptions that the switch can make about subscriber actions. In this context, then, it is unnecessary and even unwarranted to assume that the subscriber will always behave reasonably. Fortunately,

many potentially unreasonable behaviors of the subscriber are in fact physically impossible. The subscriber cannot, for instance, generate two off−hook signals in a row without an intervening on−hook, and the subscriber cannot dial digits with the phone on−hook. There is, however, no reason to assume that the subscriber will always wait for a dial tone before aborting a call attempt, as Figure 14.2 seems to indicate. In fact, a subscriber may well ignore all tones from the switch in deciding what to do next.

We can modify the model from Figure 14.2 to reflect these assumptions by combining all states that are connected by transitions that correspond to the generation of audible tones in the switch (i.e., all dashed arrows). This produces the three−state model shown on the left in Figure 14.3.

Figure 14.3. Two Alternative Subscriber Models



The subscriber can either go off−hook without dialing digits (e.g., to accept an incoming call), or the subscriber can go off−hook with the intent to dial digits (e.g., to initiate an outgoing call). But in the new model the difference between incoming and outgoing calls is no longer visible. We can therefore go one step further and combine also the two states that can be reached by the subscriber going off−hook into one single state. This leads to the two−state model shown on the right in Figure 14.3.

This last model clearly admits more behaviors than the earlier two models. It allows, for instance, for the scenario in which the subscriber keeps dialing digits while off−hook, as indeed a real subscriber might do. In the first two models this behavior was not represented, as a result of the assumptions about reasonable behavior that was made in the creation of the first model in Figure 14.2. There are several other such assumptions in Figure 14.2 that are not present in the two−state model from Figure 14.3.

We will use the two−state subscriber model as the simplest sufficient model of subscriber behavior. This model is readily expressed in PROMELA. To do so, however, we need to decide how to model the exchange of signals between subscriber and switch. In the simple model, the information flows only from the subscriber to the switch in the form of off−hook, on−hook, and digit signals. The telephone switch is designed to be much faster than a human subscriber, so it is safe to assume that the switch will always be ready to receive any signal sent by its subscribers. The simplest way to formalize this in PROMELA is with the use of a global rendezvous port. We will call the port over which a subscriber can reach the switch tpc, as a shorthand for the phone company. This resulting subscriber model expressed in PROMELA is shown in Figure 14.4.

Figure 14.4 Two−State Model of Subscriber

```
mtype = { offhook, digits, onhook };

chan tpc = [0] of { mtype };

active proctype subscriber()
{
Idle:   tpc!offhook;

Busy: if
      :: tpc!digits -> goto Busy
      :: tpc!onhook -> goto Idle
```

```
        fi
}
```

In constructing the model we will initially restrict our attention to modeling the interactions of a single subscriber with a single local switch. At this stage, nothing of interest is gained by adding multiple subscribers into the model: the switch looks at each subscriber line independently. We can significantly reduce the complexity of verification by representing the possible interactions of subscribers in a slightly more abstract way. An outgoing call attempt of our subscriber of interest may succeed or fail, for instance, depending on the state of the network and the state of other subscribers. All we are interested in is the effects of success or failure, not in the precise circumstances of success or failure. We will illustrate these notions in more detail shortly.

Switch Model

The real complexity inevitably comes in the definition of the switch behavior, so it is again important to keep things as simple as possible at first. We will develop a switch model here for the handling of outgoing calls only, reducing the number of issues that we will have to confront somewhat. The interplay of incoming and outgoing calls can be subtle, but it can be studied separately once we have confidence in the basic model we are developing here.

A first−cut model of the switch behavior can then be formalized in PROMELA, in a simple state−oriented format, as shown in Figure 14.5. Because the audible tones are generated more for information than to restrict the subscriber actions, they appear in this model as print statements only. In particular, these signals need not be recorded in state variables.

Figure 14.5 Simple Switch Model for Outgoing Calls

```
active proctype switch() /* outgoing calls only */
{
Idle:   if
        :: tpc?offhook ->
                printf("dial tone\n"); goto Dial
        fi;
Dial:
        if
        :: tpc?digits ->
                printf("no tone\n"); goto Wait
        :: tpc?onhook ->
                printf("no tone\n"); goto Idle
        fi;
Wait:
        if
        :: printf("ring tone\n") -> goto Connect;
        :: printf("busy tone\n") -> goto Busy
        fi;
Connect:
        if
        :: printf("busy tone\n") -> goto Busy
        :: printf("no tone\n")   -> goto Busy
        fi;
Busy:
        if
        :: tpc?onhook ->
                printf("no tone\n"); goto Idle
        fi
}
```

In this model, the success or failure of an outgoing call is represented as a non−deterministic choice between the generation of a ring tone or a busy tone signal in state named Wait. The state named Connect represents the situation where call setup is completed. The call can now end either by the remote subscriber (which is not explicitly present in the model here) hanging up first, or the local subscriber hanging up first. In the first case, a busy tone will be generated; in the latter case no tone is generated. The two possibilities are again formalized with the help of a non−deterministic choice, indicating that both scenarios are possible.

There is no interaction with remote switches in the network represented in this model just yet. We will add that shortly, after we can convince ourselves that the simpler model is on track. As a first check, we can

perform some short simulation runs, limiting the run to twenty steps. Such simulations show sensible call scenarios for this model, for instance, as follows:

```
$ spin -c -u20 version1
proc 0 = subscriber
proc 1 = switch
q\p   0   1
  1   tpc!offhook
  1   .   tpc?offhook
          dialtone
  1   tpc!digits
  1   .   tpc?digits
          notone
          ringtone
          notone
  1 tpc!onhook
  1 .   tpc?onhook
        notone
  1 tpc!offhook
  1 .   tpc?offhook
        dialtone
  1 tpc!digits
  1 .   tpc?digits
        notone
        ringtone
-------------
depth-limit (-u20 steps) reached
-------------
final state:
-------------
#processes: 2
 20:    proc 1 (switch) line 29 "version1" (state 12)
 20:    proc 0 (subscriber) line 11 "version1" (state 6)
2 processes created
```

Next, we perform a verification. The verification run confirms that there are no major problems and that the behavior is still exceedingly simple, with just nine reachable, and no unreachable states. The results are as follows:

```
$ spin -a version1
$ cc -o pan pan.c
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim            - (none specified)
        assertion violations   +
        acceptance cycles      - (not selected)
        invalid end states     +

State-vector 24 byte, depth reached 11, errors: 0
       9 states, stored
       6 states, matched
```

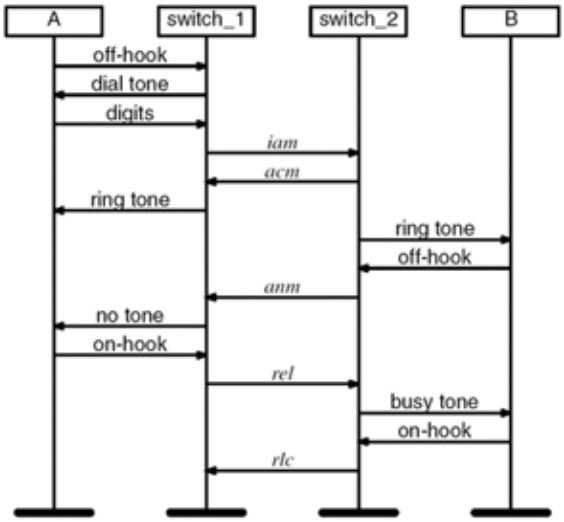
```
      15 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573   memory usage (Mbyte)
unreached in proctype subscriber
        line 15, state 8, "-end-"
        (1 of 8 states)
unreached in proctype switch
        line 40, state 29, "-end-"
        (1 of 29 states)
```

This puts us in a good position to extend our first model to a slightly more realistic one by adding the possible interactions with remote switches.

Remote Switches

So far, our switch model decides internally whether or not a call attempt failed or succeeded by making a non−deterministic decision on the generation of either a ring tone or a busy tone. We will now add a little more of the dialogue that can actually take place inside the switch during the setup of an outgoing call. In most cases the switch will have to interact with remote switches in the network to determine if the called number and the network resources that are needed to connect to it are available. The protocol for that is known as Signaling System 7, SS7 for short. A typical SS7 dialogue is shown in Figure 14.6.

Figure 14.6. SS7 Scenario for Call Setup



The first message sent by the local switch to a remote switch is called the initial address message, iam. The message triggers an address complete message, acm, in response. When the call is answered, an answer message, anm, follows. The teardown phase is started with a release, rel, request, which is acknowledged with a release confirmation, rlc.

To model this interaction we have to add a model of a remote switch. Note that we do not need to model the behavior of remote subscribers directly, because their behavior is not directly visible to the local switch. The remote subscribers are hidden behind remote switches and all negotiations about the setup and teardown of calls happen only through the intermediation of the remote switches. Also note that even though every switch acts both as a local switch to its local subscribers and as a remote switch to the rest of the network, it would be overkill to clone the local switch behavior to derive remote switch behavior. Doing so has the unintended consequence of making the detailed internal behavior of remote switches and remote subscribers visible to the verifier, which can significantly increase verification complexity.

Let us first extend the model of the local switch with the new SS7 message exchanges. This leads to the extended switch model shown in Figure 14.7.

Figure 14.7 Extended Local Switch Model

```
mtype = { iam, acm, anm, rel, rlc }; /* ss7 messages */

chan rms = [1] of { mtype }; /* channel to remote switch */
```

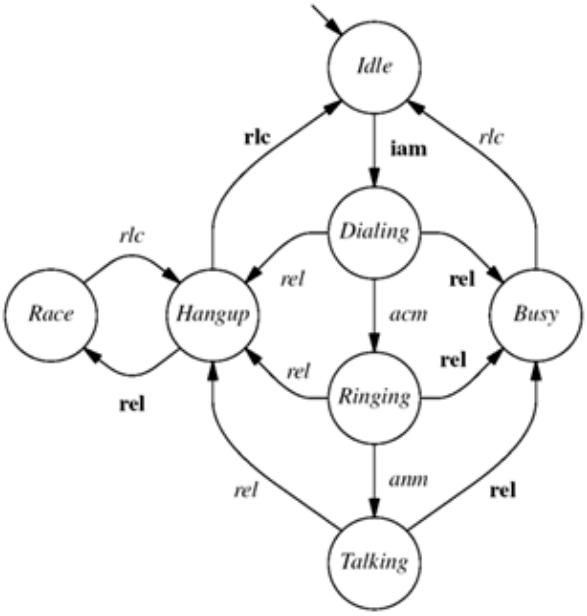
```
active proctype switch_ss7()
{
Idle:
    if
    :: tpc?offhook -> printf("dial tone\n"); goto Dial
    fi;
Dial:
    if
    :: tpc?digits -> printf("no tone\n"); rms!iam;
                     goto Wait
    :: tpc?onhook -> printf("no tone\n"); goto Idle
    fi;
Wait:
    if
    :: tpc?acm -> printf("ring tone\n"); goto Wait
    :: tpc?anm -> printf("no tone\n"); goto Connect
    :: tpc?rel -> rms!rlc; printf("busy tone\n");
                  goto Busy
    :: tpc?onhook -> rms!rel; goto Zombie1
    fi;
Connect:
    if
    :: tpc?rel -> rms!rlc; printf("busy tone\n"); goto Busy
    :: tpc?onhook -> rms!rel; goto Zombie1
    fi;
Busy:           /* off-hook, waiting for on-hook */
    if
    :: tpc?onhook -> printf("no tone\n"); goto Idle
    fi;

Zombie1:        /* on-hook, waiting for rlc */
    if
    :: tpc?rel -> rms!rlc; goto Zombie1
    :: tpc?rlc -> goto Idle
    :: tpc?offhook -> goto Zombie2
    fi;
Zombie2:        /* off-hook, waiting for rlc */
    if
    :: tpc?rel -> rms!rlc; goto Zombie2
    :: tpc?rlc -> goto Busy
    :: tpc?onhook -> goto Zombie1
    fi
}
```

We have introduced two new states, called Zombie1 and Zombie2, to represent different intermediate phases of a call teardown which now requires interaction with the remote switch.

The next step is to add a matching model for a remote switch, handling incoming request for connections. The switch can reject calls by immediately sending the rel message in response to the initial iam message, to signify that it is busy or otherwise unable to handle the request. The rel message is always acknowledged with an rlc confirmation.

An outline for the behavior of a remote switch is shown in Figure 14.8. Message names in bold indicate incoming messages; the remaining messages are the responses. Note that there can be a race between the two subscribers for the teardown of the call. Messages between local and remote switches travel over the network and will generally incur some latency, so rather than a rendezvous port we have used a buffered message channel, though with a very small buffer capacity of one message to keep things simple.

Figure 14.8. POTS Interface Model for a Remote Switch



The outline from Figure 14.8 is represented in PROMELA in Figure 14.9.

Figure 14.9 PROMELA Model of Visible Behavior of Remote Switch

```
active proctype remote_ss7()
{
Idle:
        if
        :: rms?iam -> goto Dialing
        fi;
 Dialing:
        if
        :: tpc!acm -> goto Ringing
        :: tpc!rel -> goto Hangup
        :: rms?rel -> goto Busy
        fi;
Ringing:
        if
        :: tpc!anm -> goto Talking
        :: tpc!rel -> goto Hangup
        :: rms?rel -> goto Busy
        fi;
Talking:
        if
        :: tpc!rel -> goto Hangup
        :: rms?rel -> goto Busy
        fi;
Hangup:
        if
        :: rms?rlc -> goto Idle
        :: rms?rel -> goto Race
        fi;
Busy:
        if
        :: rms?rlc -> goto Idle
        fi;
```

```
Race:
        if
        :: tpc!rlc -> goto Busy
        fi
}
```

The verifier is content with these extensions, reporting the following result:

```
$ spin -a version2
$ cc -o pan.c
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)

        assertion violations    +
        acceptance cycles       - (not selected)
        invalid end states      +

State-vector 32 byte, depth reached 30, errors: 0
      52 states, stored
      36 states, matched
      88 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573  memory usage  (Mbyte)

unreached in proctype subscriber
        line 15, state 8, "-end-"
        (1 of 8 states)
unreached in proctype switch_ss7
        line 87, state 62, "-end-"
        (1 of 62 states)
unreached in proctype remote_ss7
        line 125, state 41, "-end-"
        (1 of 41 states)
```

The number of reachable states has increased, but we have succeeded in keeping the model small. We may also be curious to see what the effect is of a more generous allotment of buffer capacity in the message channel between local and remote switches. If we change the number of slots in the buffer from one to two, the number of reachable system states increases to 60, implying that this change does indeed allow for some new behaviors. A further increase to three slots increases the number of states to 64, which remains unaffected by any further increases.

Adding Features

At this point we can improve the model by adding a treatment for incoming calls that originate at remote switches. We could also consider extending the model to handle multiple subscribers or end−to−end connections. Instead, we will try extend the switch behavior in a slightly more interesting way—by adding a call processing feature.

Three–Way Calling

We would like to add the capability for a subscriber to flash the hook after a call has been set up (i.e., quickly going on–hook and back off–hook) to place the currently connected party on hold and get a new dial tone. The subscriber should then be able to dial a new call, and establish a three–way connection by flashing the hook a second time. A third flash of the hook should terminate the three–way connection by dropping the last dialed party. We will assume that an on–hook from the originating subscriber during the call terminates all connections, independent of the current state of the call.

The addition of feature behavior like this to an existing call model often introduces unexpected types of interaction between the existing, trusted behavior and the newly added behavior. Being able to check these types of extensions with small verification models can therefore be of considerable value.

The switch must now be able to manage two connections for the same subscriber, so we will need to extend the model to have at least two instantiations of the model for a remote switch. We want to keep the control of the different connections separate, to make sure that we do not unnecessarily complicate the behavior of the switch. We can accomplish this by introducing a subscriber line session manager process that can interact with multiple session handlers. The manager keeps track of which session is active and shuttles the messages between sessions and subscriber. The various sessions are unaware of each other's existence and can behave just like in the single connection model from before.

A first change that we have to make to accomplish all this in the last model is to change the role of the switch process into that of a session manager. Before making any other changes to support the three–way calling feature directly, we will make and check this change. Figure 14.10 shows the new version of the switch process.

Figure 14.10 Switch Session Management Structure

```
chan sess = [0] of { mtype };

mtype = { idle, busy }; /* call states */
mtype s_state = idle;

active proctype switch()
{       mtype x;

        atomic
        {       run session_ss7(sess, rms);
                run remote_ss7(rms, sess)
        };
end:    do
        :: tpc?x ->
                if
                :: x == offhook ->
                        assert(s_state == idle);
                        s_state = busy
                :: x == onhook ->
                        assert(s_state == busy);
                        s_state = idle
                :: else
                fi;
                sess!x /* forward message */
        od
```

```
}
```

In this version of the switch process we have used a slightly different approach to the representation of the call states. Instead of using labeled control–flow points (as in Figure 14.9), we use `mtype` variables to store the state information.

The switch process now creates instantiations for a single session handler process and a remote switch, passing the proper message channels for input and output as parameters to these processes. We have added a channel named sess to be used by the switch process to pass call control messages from the subscriber to the local session handler. Since this is a local interaction within the switch, we can safely make this a rendezvous port again. We have also added a global variable s_state to record the call state of the session process, in so far as it is known by the session manager.

The remote switch process remains as it was in Figure 14.9, except that it is now instantiated dynamically by the switch process and sends to and receives from message channels that are passed to it by the switch process via parameters. Similarly, the session handler process remains as it was in Figure 14.7, except for the the name change from switch to session_ss7 and the use of message channels that are passed via parameters.

In checking this version of the model, we would expect the number of reachable states to increase somewhat, because of the addition of the session management process, but there should be no change in functionality. Some initial simulation runs appear to confirm the latter. Running the verification produces the following result:

```
$ spin -a version3
$ cc -o pan pan.c
$ ./pan
pan: invalid end state (at depth 41)
pan: wrote version3.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction

Full statespace search for:
        never claim           - (none specified)
        assertion violations  +
        acceptance cycles     - (not selected)
        invalid end states    +

State-vector 52 byte, depth reached 42, errors: 1
      30 states, stored
       0 states, matched
      30 transitions (= stored+matched)
       1 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573 memory usage (Mbyte)
```

The error reported here reveals one of a series of problems with the new model. Seven of these problems are cases of incompleteness. We find that in the new session handler, based on the version in Figure 14.7, the digits message can be received in states Wait, Connect, Busy, and Zombie2, where the message is currently not expected. Similarly, the acm and anm messages can arrive in state Zombie1. Further, we now also detect

an incompleteness in the remote switch process, which we based on Figure 14.9. Here the message rlc can now arrive in the Race state.

The sample executions that are generated by SPIN leave little room for doubt that these errors are real and must be corrected. Why is it that these errors did not show up before we introduced the session manager process? The reason is simple, but only in retrospect. In the earlier version of the model, there was a direct communication channel between switch and subscriber, based on rendezvous. The subscriber could offer to perform a rendezvous handshake on a digits message while the switch process was in state Wait, but because the switch process had no matching receive operation at that state, the offer could be declined without causing an error. In the new model the communication is in two steps, introducing an asynchronous decoupling of behaviors. Offers from the subscriber on any messages are now always accepted by the session management process, independent of the session state, and they must now be passed on successfully to the new session handler. We could have found the same problems in the earlier model if we switched from rendezvous communication to a buffered communication channel between subscriber and switch.

If we repair the newly found omissions in the session handler process we obtain the new model in Figure 14.11. Similarly, the corrected model for the remote switch is shown in Figure 14.12.

Figure 14.11 New Model for the Session Handler

```
proctype session_ss7(chan inp; chan out)
{
Idle:
    if
    :: inp?offhook -> printf("dial tone\n"); goto Dial
    fi;
Dial:
    if
    :: inp?digits -> printf("no tone\n");
                     out!iam; goto Wait
    :: inp?onhook -> printf("no tone\n"); goto Idle
    fi;
 Wait:
    if
    :: inp?acm -> printf("ring tone\n"); goto Wait
    :: inp?anm -> printf("no tone\n"); goto Connect
    :: inp?rel -> out!rlc; printf("busy tone\n"); goto Busy
    :: inp?onhook -> out!rel; goto Zombie1
    :: inp?digits -> goto Wait /* added */
    fi;
Connect:
    if
    :: inp?rel -> out!rlc; printf("busy tone\n"); goto Busy
    :: inp?onhook -> out!rel; goto Zombie1
    :: inp?digits -> goto Connect /* added */
    fi;
Busy:           /* off-hook, waiting for on-hook */
    if
    :: inp?onhook -> printf("no tone\n"); goto Idle
    :: inp?digits -> goto Busy /* added */
    fi;
Zombie1:        /* on-hook, waiting for rlc */
    if
    :: inp?rel -> out!rlc; goto Zombie1
    :: inp?rlc -> goto Idle
    :: inp?offhook -> goto Zombie2
    :: inp?acm -> goto Zombie1 /* added */
```

```
    :: inp?anm -> goto Zombie1 /* added */
    fi;
Zombie2:        /* off-hook, waiting for rlc */
    if
    :: inp?rel -> out!rlc; goto Zombie2
    :: inp?rlc -> goto Busy
    :: inp?onhook -> goto Zombie1
    :: inp?digits -> goto Zombie2 /* added */
    fi
}
```

Figure 14.12 New Model for the Remote Switch

```
proctype remote_ss7(chan inp; chan out)
{
Idle:
        if
        :: inp?iam -> goto Dialing
        fi;
Dialing:
        if
        :: out!acm -> goto Ringing
        :: out!rel -> goto Hangup
        :: inp?rel -> goto Busy
        fi;
Ringing:
        if
        :: out!anm -> goto Talking
        :: out!rel -> goto Hangup
        :: inp?rel -> goto Busy
        fi;
Talking:
        if
        :: out!rel -> goto Hangup
        :: inp?rel -> goto Busy
        fi;
Hangup:
        if
        :: inp?rlc -> goto Idle
        :: inp?rel -> goto Race
        fi;
Busy:
        if
        :: inp?rlc -> goto Idle
        fi;
Race:
        if
        :: out!rlc -> goto Busy
        :: inp?rlc ->   /* added */
                out!rlc; goto Idle
        fi
}
```

If we repeat the verification with this model, all error reports disappear. As expected, the number of reachable states increases a little further, to 263 states, due to the addition of the session management process. We are now set to add support for handling an extra call session to support the three–way calling feature. First, we add the flash message, and some extra call states, with the following declarations:

```
/* Revised Declarations for Three-Way Calling */

#define NS  2  /* nr of sessions in 3way call */

mtype = { offhook, digits, flash, onhook }
mtype = { iam, acm, anm, rel, rlc };
mtype = { idle, busy, setup, threeway };

chan tpc       = [0] of { mtype };
chan rms[NS]   = [1] of { mtype };   /* added */
chan sess[NS]  = [0] of { mtype };  /* added */

mtype s_state = idle;
```

The new switch process is shown in Figure 14.13. It has to handle the additional scenarios, but it is still fairly close to the last version.

Figure 14.13 New Version of the Session Manager

```
active proctype switch()
{       mtype x;

        atomic
        {       run session_ss7(sess[0], rms[0]);
                run session_ss7(sess[1], rms[1]);
                run remote_ss7(rms[0], sess[0]);
                run remote_ss7(rms[1], sess[1])
        };
 end:   do
        :: tpc?x ->
                if
                :: x == offhook ->
                        assert(s_state == idle);
                        s_state = busy;
                        sess[0]!x
                :: x == onhook ->
                        assert(s_state != idle);
                        if
                        :: s_state == busy ->
                                sess[0]!x
                        :: else ->
                                sess[0]!x; sess[1]!x
                        fi;
                        s_state = idle
                :: x == flash ->
                        assert(s_state != idle);
                        if
                        :: s_state == busy ->
                                sess[1]!offhook;
                                s_state = setup
                        :: s_state == setup ->
                                s_state = threeway
                        :: s_state == threeway ->
                                sess[1]!onhook;
                                s_state = busy
                        fi
                :: else ->
                        if
                        :: s_state == idle
```

```
                        /* ignored */
                    :: s_state == busy ->
                        sess[0]!x
                    :: else ->
                        sess[1]!x
                    fi
                fi
        od
}
```

The handling of the messages arriving at the session manager now depends on the state of the call, in a fairly straightforward way. Running a verification of this extended model produces the following result:

```
  $ spin -a version4
  $ cc -o pan pan.c
  $ ./pan (Spin Version 4.0.7 -- 1 August 2003)
                + Partial Order Reduction

  Full statespace search for:
         never claim               - (none specified)
         assertion violations      +
         acceptance cycles         - (not selected)
         invalid end states        +

 State-vector 80 byte, depth reached 5531, errors: 0
    30479 states, stored
    55947 states, matched
    86426 transitions (= stored+matched)
        3 atomic steps
 hash conflicts: 1836 (resolved)
 (max size 2^18 states)

 Stats on memory usage (in Megabytes):
 2.682   equivalent memory usage for states ...
 1.827   actual memory usage for states (compression: 68.13%)
         State-vector as stored = 52 byte + 8 byte overhead
 1.049   memory used for hash table (-w18)
 0.320   memory used for DFS stack (-m10000)
 3.109   total actual memory usage

 unreached in proctype subscriber
         line 22, state 10, "-end-"
         (1 of 10 states)
 unreached in proctype session_ss7
         line 106, state 74, "-end-"
         (1 of 74 states)
 unreached in proctype switch
         line 156, state 46, "-end-"
         (1 of 46 states)
 unreached in proctype remote_ss7
         line 196, state 44, "-end-"
         (1 of 44 states)
```

The number of reachable states has now increased from 263 to 30,479, but no new errors, and no unreachable states, are reported.

A Three−Way Calling Scenario

Did we actually succeed in reproducing the three−way calling behavior we had in mind? We can make sure of this by formalizing and checking some properties that can together establish compliance with the required feature behavior. As one of those checks, we can check that the intended three−way calling behavior is at least possible, simply by claiming that it cannot occur and allowing SPIN to generate a counterexample. We can, for instance, check the behavior that results if the subscriber generates the sequence of off−hook, digit, and flash signals that corresponds to the correct setup of a three−way call. The problem we have to solve now is to detect the occurrence of these events with a system state property. The interaction between subscriber and the switch currently takes place via a rendezvous port, which cannot be polled for the presence of messages. We can get around this problem in two different ways. The more obvious method is perhaps to change the rendezvous port named tpc into a one−slot buffered message channel, so that the contents of the single slot can be polled. This change is effective, but it also increases the complexity of the verification, and it may introduce new behaviors. A quick check with SPIN tells us that the number of system states roughly triples (reaching 97,791 states), but no error behaviors are introduced.

Another method, which in this case incurs lower overhead, is to add a global state variable called last_sent, and to change the subscriber process in such a way that it always assigns the value of the last sent message to that variable, where it can be checked with a simple state property. The updated version of the subscriber process would then look as shown in Figure 14.14.

Figure 14.14 Revised Subscriber Process

```
mtype last_sent;

active proctype subscriber()
{
Idle:   tpc!offhook;
        last_sent = offhook;

Busy:   if
        :: atomic { tpc!digits ->
                last_sent = digits;
                goto Busy
           }
        :: atomic { tpc!flash ->
                last_sent = flash;
                goto Busy
           }
        :: atomic { tpc!onhook ->
                last_sent = onhook;
                goto Idle
           }

        fi
}
```

With this change, the number of reachable states increases from 30,479 to 35,449 system states, a far smaller penalty.

The claim we are interested in can be formalized as shown in Figure 14.15. In this claim, we need to refer to the process instantiation numbers of the processes of type `remote_ss7` and `session_ss7`. A simple way to find out what these `pid` numbers are is to print them in a verbose simulation run of the system.

Figure 14.15 Never Claim to Trigger Three–Way Calling Scenario

```
#define Final \
        subscriber@Idle && switch@end \
        && remote_ss7[4]@Idle && remote_ss7[5]@Idle \
        && session_ss7[2]@Idle && session_ss7[3]@Idle

#define Event(x) \
        do \
        :: last_sent == x -> break \
        :: else \
        od

never { /* sample of a 3way call: */
        Event(offhook);
        Event(digits);
        Event(flash);
        Event(digits);
        Event(flash);
        Event(digits);
        Event(flash);
        Event(onhook);
        do
        :: Final -> break
        :: else
        od
}
```

A sample three–way calling scenario is now quickly generated. We first compile and run the model checker in a mode that allows us to generate the shortest possible scenario that matches the claim. Because the claim expresses a safety property (there are no accept–state or progress–state labels), we can find such a scenario with a breadth–first search.

```
$ spin -a version6
$ cc -DBFS -o pan pan.c
$ ./pan
...
pan: claim violated! (at depth 79)
pan: wrote version6.trail
...
```

We can reproduce the scenario with SPIN's guided simulation option, as follows. Because the claim successfully matched the scenario we are interested in, we do not need to scrutinize it too much further. If, however, no matching execution would have been found, we would have had to consider more carefully if, for instance, the claim structure is compatible with the requirements of SPIN's partial order reduction algorithm, i.e., that it expresses a stutter–invariant property (it does), and if not, we would have repeated the check with partial order reduction disabled.

```
$ spin -t -c version6
proc 0 = subscriber
proc 1 = switch
proc 2 = session_ss7
proc 3 = session_ss7
```

```
proc 4 = remote_ss7
proc 5 = remote_ss7
q\p   0   1   2   3  4  5
  5   tpc!offhook
  5   .   tpc?offhook
  1   .   sess[0]!offhook
  1   .   .   .   inp?offhook
                  MSC: dial tone
  5   tpc!digits
  5   .   tpc?digits
  1   .   sess[0]!digits
  1   .   .   .   inp?digits
                  MSC: no tone
  3   .   .   .   out!iam
  3   .   .   .   .   .   inp?iam
  1   .   .   .   .   .   out!rel
  1   .   .   .   inp?rel
  3   .   .   .   out!rlc
                  MSC: busy tone
  3   .   .   .   .   . inp?rlc
  5   tpc!flash
  5   .   tpc?flash
  2   .   sess[1]!offhook
  2   .   .   .   . inp?offhook
                    MSC: dial tone
  5   tpc!digits
  5   .   tpc?digits
  2   .   sess[1]!digits
  2   .   .  .    . inp?digits
                    MSC: no tone
  4   .   .  .    . out!iam
  4   .   .  .    . .    . inp?iam
  2   .   .  .    . .    . out!rel
  2   .   .  .    . inp?rel
  4   .   .  .    . out!rlc
                    MSC: busy tone
  4   .   .  .    . .    . inp?rlc
  5   tpc!flash
  5   .   tpc?flash
  5   tpc!digits
  5   .   tpc?digits
  2   .   sess[1]!digits
  2   .   .  .     .  inp?digits
  5   tpc!flash
  5   .   tpc?flash
  2   .   sess[1]!onhook
  2   .   .  .     .  inp?onhook
                     MSC: no tone
  5   tpc!onhook
  5   .   tpc?onhook
  1   .   sess[0]!onhook
  1   .   .  .    inp?onhook
                  MSC: no tone
  spin: trail ends after 79 steps
  -------------
  final state:
  -------------
  #processes: 7

              queue 3 (rms[0]):
              queue 4 (rms[1]):
```

```
            s_state = idle
            last_sent = onhook
79: proc 5 (remote_ss7) line 172 "vertsion6" (state 3)
79: proc 4 (remote_ss7) line 172 "vertsion6" (state 3)
79: proc 3 (session_ss7) line 38 "vertsion6" (state 4)
79: proc 2 (session_ss7) line 38 "vertsion6" (state 4)
79: proc 1 (switch) line 128 "vertsion6" (state 43) ...
79: proc 0 (subscriber) line 16 "vertsion6" (state 1)
79: proc - (:never:) line 227 "vertsion6" (state 52)
7 processes created
```

In Summary

In this chapter we have developed a relatively simple model of a telephone switch that represents an interesting fragment of its behavior for handling outgoing calls.

By starting with a very simple model that was revised in small and easily understood increments, we can catch errors at an early stage and avoid large blowups in the complexity of verification. After each small incremental step, we can check our intuition about the behavior of the model with short simulation and verification runs. Despite a few obvious limitations (e.g., the absence of a treatment for incoming calls), the model already includes some feature behavior that can be very challenging to implement correctly. The hard part of an exercise like this is to keep the model and its state space small, so that we can continue to verify it rigorously. This is an exercise in restriction and judicious abstraction. The target of this exercise is always to find the smallest sufficient model that allows us to verify all properties of interest.

Perhaps one of the nicer things about the use of a model checker such as SPIN is that the tool does not expect us to get things right on the first attempt. The tool can help us find both sources of complexity and sources of error. A model checking tool is often conveniently used as an exploratory tool: allowing the user to answer quick what−if questions about possible directions that might be taken to solve complex software design problems.