Chapter 5. Using Design Abstraction

> "Seek simplicity, and distrust it."
>
> —(Alfred North Whitehead, 1861–1947)

System design is a process of discovery. By exploring possible solutions, a designer discovers the initially unknown constraints, and weeds out the designs that seemed plausible at first, but that do not survive closer scrutiny. In the process, the designer determines not only what is right and what is wrong, but also what is relevant and what is irrelevant to the basic design premises.

For a design tool to be effective in this context, it needs to be able to assist the designer in the creation and the analysis of intuitive high–level abstractions without requiring the resolution of implementation–level detail. The tool should be able to warn when the design premises are logically flawed, for instance, when the design is ambiguous, incomplete, or inconsistent, or when it does not exhibit the properties it was designed to have. To use such a tool well, the designer should be comfortable building and checking design abstractions. This task is sufficiently different from the task of building design implementations that it is worth a closer look.

What Makes a Good Design Abstraction?

The purpose of a design model is to free a designer from having to resolve implementation−level details before the main design premises can be checked. SPIN not only supports design abstractions, it requires them. At first sight, SPIN's input language may look like an imperative programming language, but it intentionally excludes a number of features that would be critical to an implementation language. It is not directly possible, for instance, to express floating point arithmetic, process scheduling, or memory management operations. Although all of these things can be important in an implementation, they can and should be abstracted from a high−level design model. Note that in much the same vein, most of today's higher−level programming languages make it impossible to express decisions on register allocation, or the management of instruction caches. These issues, though important, are best resolved and checked at a different level of abstraction.

So, what makes a good design abstraction in concurrent systems design? The focus on concurrent systems has important implications. In a sequential application, the best abstraction is typically data−oriented. In a concurrent application the preferred abstraction is control−oriented.

Data and Control

A large sequential program is typically divided into smaller modules, each performing a well−defined computational function. The interfaces between these modules are kept as small as possible, reducing the number of assumptions that one module must make about the others. The interface definitions in this case are data−oriented, not control−oriented. The modules in a sequential application typically do not maintain internal state information independently from the rest of the program.

This is different in distributed systems design. In a distributed system, the module structure is typically determined by an externally imposed system architecture (e.g., determined by a physical separation of the main components of the system). Each module then has its own independent thread of control, which necessarily carries state information. In this case control, not data, is the primary concern in the definition of module interfaces.

In a SPIN verification model the focus is on the control aspects of a distributed application, not on the computational aspects. PROMELA allows us to express the assumptions that are made within each module about the interactions with other modules. The language discourages the specification of detailed assumptions about process−internal computations.

To make sure that the models we can specify in PROMELA always have effectively verifiable properties, we impose two requirements:

- the model can only specify finite systems, even if the underlying application is potentially infinite, and
- the model must be fully specified, that is, it must be closed to its environment.

The second requirement says that the behavior that is defined in a verification model may not depend on any hidden assumptions or components. All input sources must be part of the model, at least in abstract form.

For arbitrary software applications these two requirements are not automatically satisfied. To achieve verifiability, we have to apply abstraction.

A program that allows unrestricted recursion, or that contains unbounded buffers, for instance, is not finite−state. A program that reads input from a file−descriptor or an input−stream, similarly, is not closed to its environment. These programs cannot be verified with automatic techniques unless some abstractions are made.

The need for abstraction, or modeling, is often seen as a hurdle instead of a feature in model checking applications. Choosing the right level of abstraction, though, can mean the difference between a tractable model with provable properties and an intractable model that is only amenable to simulation, testing, or manual review. Sometimes we have to choose between proving simple properties of a complex model, formalized at a low−level abstraction, and proving more complex properties of a simpler model, formalized at a higher−level abstraction.

The importance of abstraction places demands on the design of the input language of a model checking tool. If the input language is too detailed, it discourages abstractions, which in the end obstructs the verification process. We have to be careful, therefore, in choosing which features are supported in a verification system. In SPIN, for instance, a number of otherwise desirable language features have been left out of the specification language. Among them are support for memory management, floating point calculations, and numerical

analysis. Other verification systems are even more restrictive, and exclude also structured data types, message passing, and process creation. Still other systems are more permissive, and, at the price of increased complexity, include support for: real−time verification, probabilities, and procedures. All other things being equal, we should expect the most permissive system to be the easiest to build models for, but the least efficient to verify them. SPIN attempts to find a balance between ease of use and model checking efficiency.

The recommended way to develop a verification model is as follows:

- First decide which aspects of the design are important and require verification. Express these in a set of system requirements. Requirements, to be of use in the verification process, must be testable. It should be possible to state clearly under which precise conditions a requirement is violated. It should further not be possible for the system to fail without violating at least one of the requirements. The collection of an adequate set of requirements is in itself a process of discovery. What at first may be thought to be an insignificant requirement may become the dominant design issue over time. Other design requirements or design constraints may at first be completely unknown, until an initial analysis reveals their relevance.
- Next, consider the essence of the design itself, specifically those aspects that are meant to secure the behavior of interest, and that help the system meet its requirements.
- Only then construct an executable abstraction in PROMELA for the design. The abstraction should be detailed enough to capture the essence of the solution, and no more. What we are looking for is the smallest sufficient model that allows us to perform a verification.

It is sometimes helpful to think of a verification model as a system of process modules and interfaces. Each process module represents an asynchronous entity in a distributed system. Each process is only detailed enough to capture the minimal set of assumptions that this module must make about its peers. These assumptions take the form of interface definitions. The purpose of the verification is to check that all interface definitions, formalizing the assumptions that processes in the system make about each other, are logically consistent. We can perform such checks without having to detail each module's precise implementation. The modules remain, as it were, black boxes.

Phrased differently, the purpose of design verification is to find logical flaws in the reasoning that produced a design. The goal is not to find computational errors or coding errors.

The verification model must allow us to make refutable statements about a design. As Imre Lakatos once phrased it

> "The purpose of analysis is not to compel belief but rather to suggest doubt."

Elements of a model that cannot contribute to its refutability can and should be deleted in the interest of enhancing the model's verifiability.

In the last two decades verification tools have evolved from simple reachability analyzers, run on relatively small machines, into powerful model checking systems, that can be run on machines that are orders of magnitude faster and larger. One thing has not changed though: computational complexity remains the single most important issue in this area. There is just one powerful weapon that can reliably defeat the dragon of computational complexity, and that weapon is abstraction.

New users of SPIN often attempt to build verification models that remain relatively close to the implementation level of an application, making only syntactic changes to accommodate the whims of the input language. These users often only seriously consider abstraction when a concept or feature is encountered that cannot be represented at all in the language of the model checker. At this point, the user is often

frustrated, and frustration is a poor motivator in the search for good design abstractions. Much of the detail included in verification models that are produced in this way is functionally irrelevant to the properties to be checked, yet all this detail can seriously limit the thoroughness of a possible check.

The Smallest Sufficient Model

It is sometimes easy to lose sight of the one real purpose of using a model checking system: it is to verify system properties that cannot be verified adequately by other means. If verification is our objective, computational complexity is our foe. The effort of finding a suitable design abstraction is therefore the effort of finding the smallest model that is sufficient to verify the properties that we are interested in. No more, and no less. A one−to−one translation of an implementation into a verification modeling language such as PROMELA may pass the standard of sufficiency, but it is certainly not the smallest such model and may cause unnecessary complexity in verification, or even render the verification intractable. To reduce verification complexity we may sometimes choose to generalize a problem, and sometimes we may choose to specialize it.

As the difference between a verification model and an implementation artifact becomes larger, one may well question if the facts that we are proving still have relevance. We take a very pragmatic view of this here. For our purposes, two models are equivalent if they have the same properties. This means that we can always simplify a verification model if its properties of interest are unaffected by the simplifications. A verification system, for that matter, is effective if it can be used to uncover real defects in real systems. There is little doubt that the verification methodology we are discussing here can do precisely that.

Avoiding Redundancy

The success of the model checking process for larger applications relies on our skill in finding and applying abstractions. For smaller applications this skill amounts mostly to avoiding simple cases of redundancy. It should be noted, for instance, that paradigms that are commonly used in simulation or testing can be counterproductive when used in verification. A few examples will suffice to make this point.

Counters

In the construction of a simulation model it is often convenient to add counters, for instance, to keep track of the number of steps performed. The counter is basically a write−only variable, used only in print statements.

The example in Figure 5.1 illustrates a typical use. The variable `cnt` is used here as if it were a natural number with infinite range. No check for overflow, which will inevitably occur, is made. The implicit assumption that in practical cases overflow is not likely to be a concern may be valid in program testing or simulation; it is false in verifications where all possible behaviors must be taken into account.

Figure 5.1 Counter Example

```
active proctype counter ()
{   int cnt = 1;

    do
    :: can_proceed ->
        /* perform a step */
        cnt++;
        printf("step: %d\n", cnt)
    od
}
```

It should be noted carefully that it is not necessarily a problem that the variable `cnt` may take up to 32 bits of storage to maintain its value. The real problem is that this variable can reach 2^{32} distinct values (over four billion). The complexity of the verification problem may be increased by that same amount. Phrased differently: Removing a redundant counter can reduce the complexity of a naive verification model by about nine orders of magnitude.

Sinks, Sources, and Filters

Another avoidable source of complexity is the definition of processes that act solely as a source, a filter, or a sink for a message stream. Such processes often add no refutation power to a verification model, and are almost always better avoided.[1]

> [1] The exception would be if a given correctness property directly depends on the process being present in the model. This should be rare.

- A sink process, for instance, merely receives and then discards messages. Since the messages are discarded in the model, they should probably not even be sent within the model. Having them sent but not processed would indicate an incomplete abstraction.
- A source process generates a set of possible messages that is then forwarded to a given destination. If the sole function of the source process is to provide the choice of messages, this choice can possibly be moved beneficially into the destination process, avoiding the sending of these messages altogether.
- A filter process passes messages from one process to another, possibly making small changes in the stream by dropping, duplicating, inserting, or altering messages. Again, if the desired effect is to generate a stream with a particular mix of messages, it is often possible to generate just such a stream directly.

Figure 5.2 shows a simple example of each of these three types of processes. To see how much can be saved by removing the sink process, for instance, consider the number of reachable states that is contributed by the storage of messages in the channel named `q`. The channel can hold between zero and eight distinct messages, and each of these is one of three possible types. This gives a total number of states equal to:

$$\sum_{i=0}^{8} 3^i = 9,841$$

Figure 5.2 A Sink, a Source, and a Filter Process

```
mtype = { one, two, three };

chan q = [8] of { mtype };
chan c = [8] of { mtype };

active proctype sink()
{
    do
    :: q?one
    :: q?two
    :: q?three
    od
}

active proctype filter()
{       mtype m;
        do
        :: c?m -> q!m
        od
}
```

```
active proctype source()
{
    do
    :: c!one
    :: c!two
    :: c!three
    od
}
```

This means that removing the process and the associated channel can decrease the complexity of the model by almost four orders of magnitude. The refutation power of the model is increased accordingly.

The temptation to include a dummy process is often given by a desire to model all existing parts of a system. The application being modeled may, for instance, contain a process or a task that performs a function that ends up being of only peripheral interest to the verification. There is an understandable uneasiness in the user to discard such processes, until it is reinforced that one is constructing a verification model, not the duplicate of an implementation. The difference in complexity can again be orders of magnitude.

Simple Refutation Models

Is it realistic to expect that we can build models that are of practical significance and that remain computationally tractable? To answer this, we discuss two remarkably simple models that have this property. The first model counts just twelve reachable states, which could be sketched on a napkin. The second model is not much larger, with fifty−one reachable states, yet it too has undeniable practical significance. A naive model for either of these examples could easily defeat the capabilities of the most powerful model checking system. By finding the right abstraction, though, we can demonstrate that the design represented by the first model contains a design flaw, and we can prove the other to be a reliable template for the implementation of device drivers in an operating systems kernel.

The two abstractions discussed here require less computational power to be verified than what is available on an average wristwatch today. To be sure, it is often harder to find a simple model than it is to build a complex one, but the effort to find the simplest possible expression of a design idea can provide considerably greater benefits.

Pathfinder

NASA's Pathfinder landed on the surface of Mars on July 4th, 1997, releasing a small rover to roam the surface. The mechanical and physical problems that had to be overcome to make this mission possible are phenomenal. Designing the software to control the craft may in this context seem to have been one of the simpler tasks, but designing any system that involves concurrency is challenging and requires the best minds and tools. Specifically, in this case, it was no easier to design the software than the rest of the spacecraft. And, as it turned out, it was only the control software that occasionally failed during the Pathfinder mission. A design fault caused the craft to lose contact with earth at unpredictable moments, causing valuable time to be lost in the transfer of data.

It took the designers a few days to identify the origin of the bug. To do so required an attempt to reproduce an unknown, non−deterministic execution sequence with only the tools from a standard system test environment, which can be very time−consuming.

The flaw turned out to be a conflict between a mutual exclusion rule and a priority rule used in the real−time task scheduling algorithm. The essence of the problem can be modeled in a SPIN verification model in just a few lines of code, as illustrated in Figure 5.3.

Figure 5.3 Small Model for the Pathfinder Problem

```
mtype = { free, busy, idle, waiting, running };

mtype H_state = idle;
mtype L_state = idle;
mtype mutex = free;

active proctype high_priority()
{
end:
    do
    :: H_state = waiting;
        atomic { mutex == free -> mutex = busy };
        H_state = running;

        /* produce data */

        atomic { H_state = idle; mutex = free }
    od
}

active proctype low_priority() provided (H_state == idle)
{
end:
    do
    :: L_state = waiting;
        atomic { mutex == free -> mutex = busy };
        L_state = running;

        /* consume data */

        atomic { L_state = idle; mutex = free }
    od
}
```

Two priority levels are modeled here as `active proctypes`. Both processes need access to a critical region for transferring data from one process to the other, which is protected by a mutual exclusion lock. If by chance the high priority process starts running while the low priority process holds the lock, neither process can proceed: the high priority process is locked out by the mutex rule, and the low priority process is locked out by the priority rule, which is modeled by a `provided` clause on the low priority process.

The model shown here captures the essence of this problem in as few lines as possible. A verification of this model is a routine exercise with SPIN. The verifier is generated and compiled for exhaustive search as follows:

```
$ spin -a pathfinder.pml
$ cc -o pan pan.c
```

Next, the verification run is performed:

```
$ ./pan
pan: invalid end state (at depth 4)
pan: wrote pathfinder.pml.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 20 byte, depth reached 4, errors: 1
       5 states, stored
       1 states, matched
       6 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493 memory usage (Mbyte)
```

The verifier finds an error after exploring only five system states. The full state space counts no more than twelve reachable system states, as illustrated in Figure 5.4, which should be compared to the many billions of possible states of the real memory−module in the Pathfinder controller that must be searched in exhaustive tests of the non−abstracted system. (All states can be generated by the verifier if we use option `-c0`, cf. Chapter 19, p. 536.)

Figure 5.4. Reachability Graph for Pathfinder Problem

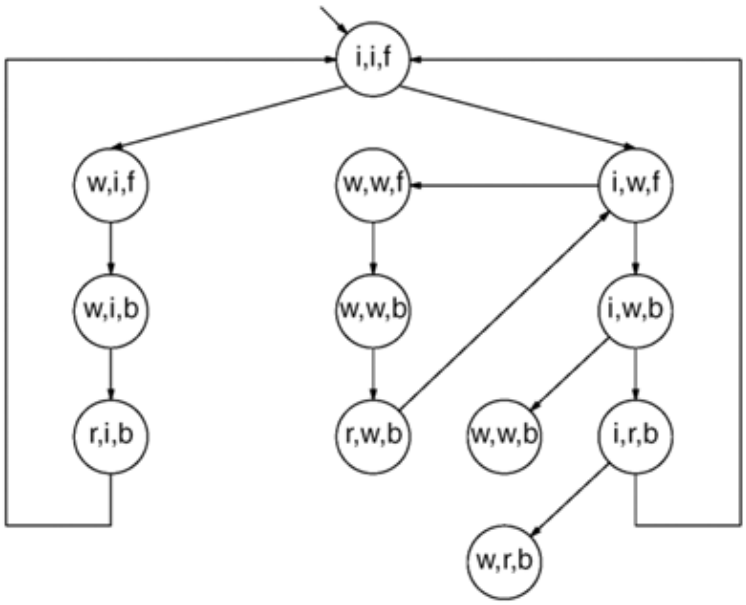
Process states: i = idle, w = waiting, r = running

Mutex lock states: f == free, b = busy

System states (x,y,z): x = high priority process, y = low priority process, z = mutex lock

Reachable Deadlock states: (w,w,b) and (w,r,b)

Starvation Cycles: {(i,i,f),(w,i,f),(w,i,b),(r,i,b)} and {(i,w,f),(w.w.f),(w,w,b),(r,w,b)}



The complete reachability graph for this system is readily built, even by a human model checker without computerized assistance. In Figure 5.4 we have marked the reachable system states with a triple. The first two elements of the triple identify the local state of each of the two processes: `i` for `idle`, `w` for `waiting`, and `r` for `running`. The third element of the triple records the state of the shared mutual exclusion lock variable: `f` for `free`, and `b` for `busy`.

Clearly, with two processes and one boolean lock, this level of abstraction gives rise to a maximum of eighteen system states (3x3x2), only twelve of which are effectively reachable from the initial system state. A verifier such as SPIN can efficiently build and analyze reachability graphs with billions of reachable system states, so a problem of this size is hardly challenging.

In this case, it is also not difficult to identify the two possible deadlock trajectories in the reachability graph, even by hand. The two system deadlock states in the graph are the states without any outgoing arrows. There is no possible exit from either state (`w`,`w`,`b`) or state (`w`,`r`,`b`) because in both cases the low priority process holds the lock and thereby blocks the high priority process, but the process holding the lock cannot proceed because of the priority rule that blocks it when the high priority process is not idle.

To analyze the behavior of this model we can look for paths leading into deadlock states. The verifier found the first such path, ending in state (`w`,`r`,`b`) (some information is elided for layout purposes):

```
$ spin -t -p pathfinder.pml
```

```
1: proc 1 (low)  line 40 ...   [l_state = waiting]
2: proc 1 (low)  line 41 ...   [((mutex==free))]
2: proc 1 (low)  line 41 ...   [mutex = busy]
3: proc 1 (low)  line 42 ...   [l_state = running]
4: proc 0 (high) line 27 ...   [h_state = waiting]
spin: trail ends after 4 steps
#processes: 2
               h_state = waiting
               l_state = running
               mutex = busy
4: proc 1 (low)  line 46 "pathfinder.pml" (state 8)
4: proc 0 (high) line 28 "pathfinder.pml" (state 4)
2 processes created
```

We can also use the verifier to look for more subtle types of properties. We may ask, for instance, if there is any way for one process to delay the execution of the other process indefinitely. Of course, because of the priority rule, there is such a possibility here. The use of the term indefinitely means that we are looking for possibly infinite executions with a special property.

The only type of infinite execution that can be performed in a finite reachability graph is, of course, a cyclic one. It is not hard to identify three basic cycles in the graph from Figure 5.4 with the property that only one of the processes repeatedly gets to its running state, while the other remains in its idle, or waiting state.

There is only effective denial of service if a process is indefinitely waiting to execute, that is, if the denied process has at least reached its waiting state. This rules out one of the three cycles, but leaves the other two as candidates for further inspection. As we shall see in more detail in Chapters 8 and 9, SPIN can reveal the existence of executions like this in even very large reachability graphs. One way to do so, in this case, is to mark the running state of either process as a progress state, and then ask SPIN to report on the existence of non−progress cycles in the reachability graph.

We will mark the running state in the low priority process as a progress state, as follows:

```
progress:      l_state = running;
```

The check for the absence of non−progress cycles is now performed as follows:

```
$ spin -a pathfinder.pml
$ cc -DNP -o pan pan.c # enable NP algorithm
$ ./pan -l -f # search for fair non-progress cycles
pan: non-progress cycle (at depth 24)
pan: wrote pathfinder.pml.trail
(Spin Version 4.0.7 -- 1 August 2003)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim           +
        assertion violations  + (if within scope of claim)
        non-progress cycles   + (fairness enabled)
        invalid end states    - (disabled by never claim)
```

```
State-vector 24 byte, depth reached 31, errors: 1
      11 states, stored (23 visited)
       4 states, matched
      27 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493 memory usage (Mbyte)
```

Inspecting the error trail reveals the path leading into a potentially infinite cycle where the high priority process starves its low priority counterpart:

```
$  spin -t -p pathfinder.pml
 2:  proc  1  (low)   line 40 ...  [l_state = waiting]
 4:  proc  1  (low)   line 41 ...  [((mutex==free))]
 4:  proc  1  (low)   line 41 ...  [mutex = busy]
 6:  proc  1  (low)   line 42 ...  [l_state = running]
 8:  proc  1  (low)   line 46 ...  [l_state = idle]
 8:  proc  1  (low)   line 46 ...  [mutex = free]
10:  proc  1  (low)   line 40 ...  [l_state = waiting]
12:  proc 0   (high)  line 27 ...  [h_state = waiting]
14:  proc 0   (high)  line 28 ...  [((mutex==free))]
14:  proc 0   (high)  line 28 ...  [mutex = busy]
16:  proc 0   (high)  line 29 ...  [h_state = running]
18:  proc 0   (high)  line 33 ...  [h_state = idle]
18:  proc 0   (high)  line 33 ...  [mutex = free]
20:  proc 0   (high)  line 27 ...  [h_state = waiting]
22:  proc 0   (high)  line 28 ...  [((mutex==free))]
22:  proc 0   (high)  line 28 ...  [mutex = busy]
24:  proc 0   (high)  line 29 ...  [h_state = running]
 <<<<<START OF CYCLE>>>>>
26:  proc 0   (high)  line 33 ...  [h_state = idle]
26:  proc 0   (high)  line 33 ...  [mutex = free]
28:  proc 0   (high)  line 27 ...  [h_state = waiting]
30:  proc 0   (high)  line 28 ...  [((mutex==free))]
30:  proc 0   (high)  line 28 ...  [mutex = busy]
32:  proc 0   (high)  line 29 ...  [h_state = running]
spin: trail ends after 32 steps
#processes: 2
               h_state = running
               l_state = waiting
               mutex = busy
32:  proc 1   (low)   line 41 "pathfinder.pml" (state 4)
32:  proc 0   (high)  line 33 "pathfinder.pml" (state 8)
2 processes created
```

Although the sample verification model for the Pathfinder problem is trivial to analyze, it should be added here that if a model of the mutual exclusion and priority scheduling rules had been constructed before the launch of the mission, before the design flaw manifested itself, the verification model would likely have contained more detail. In the real system, for instance, a third intermediate priority level was responsible for keeping the low priority process from releasing the lock, which prevented the high priority process from completing its task. Similarly, there were also other tasks in the system that manipulated the mutual exclusion locks that would likely have been included into the model and thereby increased the complexity of the

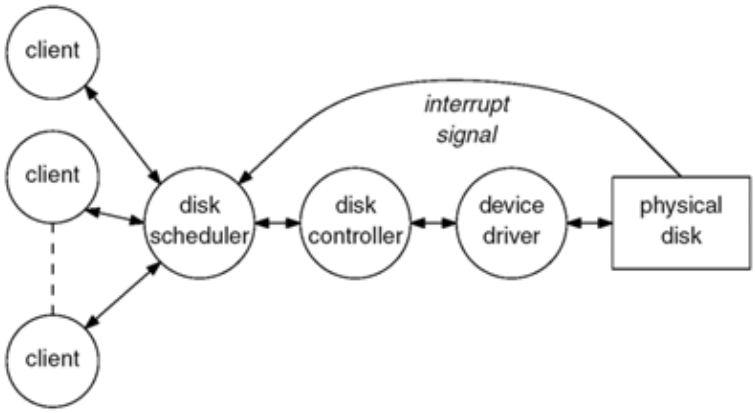
verification beyond mere napkin size. The verification itself, though, should also in these cases have been able to reveal the problem.

A Disk–Head Scheduler

The next example illustrates how an abstract model can be constructed, again in just a few lines of high–level code, to confirm that a design has the properties it is intended to have. The example is a standard problem in operating system design: scheduling access of multiple client processes to a single server, which in this case is a disk scheduler. Only one client request is served by the disk–scheduler at a time. If multiple requests arrive, they are to be queued and served in order of arrival.

In a first attempt to build a verification model for this problem, it is natural to introduce separate processes to model the disk scheduler, the disk controller, the disk device driver, and the individual client processes that submit requests. To initiate a disk operation, a client process submits a request to the scheduler, where it may get queued. When it is ready to be serviced, the scheduler sends a start command to the controller, which then initiates the requested operation through the device driver. Completion of the operation is signaled by a hardware interrupt, which is intercepted by the scheduler. This basic architecture is illustrated in Figure 5.5.

Figure 5.5. Disk Scheduler Context



Our objective is to check the basic design of the interaction between the disk scheduler and its clients. This means that the internal details of the device driver (e.g., mapping disk blocks to cylinders, sending commands to move the disk heads, etc.) are not directly relevant to this aspect of the design. To focus the verification model on the area of primary interest, we can and should abstract from the internals of the device driver process and the physical disk. The possible interactions between the device driver and the scheduler are, for instance, already captured in the device driver model from Figure 5.6.

Figure 5.6 Minimal Device Driver Interface Model

```
proctype Contr(chan req, signal)
{
    do
    ::  req?IO ->
         /* perform IO operations */
         signal!Interrupt
    od
}
```

The only assumption we are making here about the behavior of the device driver is that it will respond to every IO request with an interrupt signal within a finite, but otherwise arbitrary, amount of time. The relevant

behavior of a client process can be modeled very similarly. It suffices to assume that each client can submit one request at a time, and that it will then wait for the matching response from the scheduler.

Before looking at the remainder of the verification model, though, we can already see that these initial models for the device driver and the client have all the characteristics of filter processes. The same is true for a minimal model of the disk controller. This is not too surprising, since we deliberately placed the focus on the verification of request queuing at the disk scheduler, not on the surrounding processes.

The temptation is to include the extra processes in the abstract model anyway, simply because they represent artifacts in the application. For the actual verifi–cation job, however, their presence serves no purpose and will increase the complexity of the model. These types of processes can be removed readily by applying the relatively simple abstractions of the type we have discussed. Doing so leads to the model shown in Figure 5.7.

Figure 5.7 Disk Scheduler Model

```
#define Nclients       3

inline disk_io() {
    activeproc = curproc;
    assert(Interrupt_set == false);
    Interrupt_set = true;
}

inline Serve_client(x) {
    client_busy[x] = true;
    curproc = x+1;
   if        /* check disk status */
   :: activeproc == 0 -> disk_io()
   :: else /* Busy */ -> req_q!curproc
   fi
}

inline Handle() {
    Interrupt_set = false;
    client_busy[activeproc-1] = false;
    if
    :: req_q?curproc -> disk_io()
    :: empty(req_q) -> activeproc = 0
    fi
}

active proctype disk_sched()
{   chan req_q = [Nclients] of { byte };
    bool client_busy[Nclients] = false;
    bool Interrupt_set = false;
    byte activeproc, curproc;

    do
    :: !client_busy[0] -> progress_0: Serve_client(0)
    :: !client_busy[1] -> progress_1: Serve_client(1)
    :: !client_busy[2] -> progress_2: Serve_client(2)
    :: Interrupt_set == true -> Handle()
    od
}
```

A client process should be unable to submit a new request until the last one was completed. In the model from Figure 5.7, the client's busy status is recorded in a boolean array (rather than recorded as a process state, as in

the equivalent of Figure 5.6).

We have added two correctness properties to this model. The first property is an assertion, claiming that no new interrupt can be generated before the last one was handled. The second property is expressed with a `progress` label which appears at the point in the code where a new client request is submitted. SPIN can check that neither assertion violations nor non−progress cycles are possible for this design. The validity of the second property implies that there cannot be any infinite execution without infinite progress for at least some client.

With two clients, the reachability graph for the model in Figure 5.7 has no more than 35 states. With three clients, as shown, the number of states increases to 142. In both cases this poses no challenge to any verifier.

For a complete verification, we will have to do two separate verification runs: one run to prove absence of assertion violations and deadlock states (safety properties), and a second run to prove the absence of non−progress cycles (a liveness property). The first check, for assertion violations and deadlock states proceeds as follows:

```
$ spin -a diskhead.pml
$ 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 25 byte, depth reached 67, errors: 0
     142 states, stored
      27 states, matched
     169 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493 memory usage (Mbyte)

unreached in proctype disk_sched
        line 41, state 56, "-end-"
        (1 of 56 states)
```

The run shows that assertion violations or deadlocks (invalid end states) are not possible, and that all local process states in the `disk_sched` process are effectively reachable.

To check for the presence of non−progress cycles requires a slightly larger search, where each state can be visited up to two times. The run confirms that also non−progress cycles are not possible in this model:

```
$ spin -a diskhead.pml
$ cc -DNP -o pan pan.c
```

```
$ ./pan -l
(Spin Version 4.0.7 -- 1 August 2003)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        non-progress cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 29 byte, depth reached 146, errors: 0
     268 states, stored (391 visited)
     252 states, matched
     643 transitions (= visited+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.493 memory usage (Mbyte)

unreached in proctype disk_sched
        line 41, state 56, "-end-"
        (1 of 56 states)
```

But are three clients also sufficient to prove the required properties for an arbitrary number of clients? Every increase in the number of clients naturally increases the number of reachable states to be inspected due in large part to the increased number of permutations of distinct client requests in the request queue. Since all clients behave the same, it should suffice to prove each client–specific property for one arbitrarily chosen client.

We can do this by showing that a client–specific property holds no matter at what point in the disk scheduler's operation the client's request may arrive. Assuming that the request queue is large enough to hold the maximum number of simultaneous requests from clients, there are just three boolean conditions that together completely determine the scheduler's state and its actions upon accepting a new client request. They are:

- `empty(req_q)`
- `(activeproc == 0)`
- `Interrupt_set == true`

This gives us maximally eight relevant combinations (i.e., local scheduler states). With just one client process the truth assignment to the three conditions necessarily defaults to true, true, and false when a new request is initiated, so this definitely does not suffice to produce all eight combinations. With two clients we reach more, but we still cannot have both `Interrupt_set` and `empty(req_q)` be true at the same time. Three clients are the minimum needed to cover all eight combinations. Adding more client processes can increase the complexity of the verification further, but it cannot cover more cases or reveal anything new about this model. Admittedly, this is a very simple example of a system where processes communicate exclusively through message passing and do not access any globally shared data. Nonetheless, the original model of this system was successfully used as a guideline for the correct design and implementation of device driver modules in a commercial operating system, so even simple verification models can have realistic practical significance.

In practice, the need to produce simple verification models is not as strict as what may be suggested by the examples from this chapter. SPIN can analyze models at a rate of 10,000 to 100,000 states per second, depending on the size of the state descriptors, and the speed of the CPU. This much power should in most cases suffice to tackle even the most challenging design problems.

Controlling Complexity

We will see in Chapter 8 that the worst−case computational expense of verifying any type of correctness property with a model checker increases with the number of reachable system states R of a model. By reducing the size of R, therefore, we can try to reduce the complexity of a verification. Abstraction is the key tool we can use to keep the size of R small, but there are also other factors that we could exploit in model building.

Let n be the number of concurrent components, and let m be the total number of data objects they access. If we represent the number of control states of the i−th component by T_i, and the number of possible values for the j−th data object by D_j, then in the worst case the size of R could be the product of all values T_1 to T_n and all values D_1 to x D_m. That is, the value of R itself may well be exponential in the number of concurrent components and the number of data objects. As a general rule, therefore, it is always good to search for a model with the fewest number of components (processes, channels, data objects), that is, to construct the smallest sufficient model.

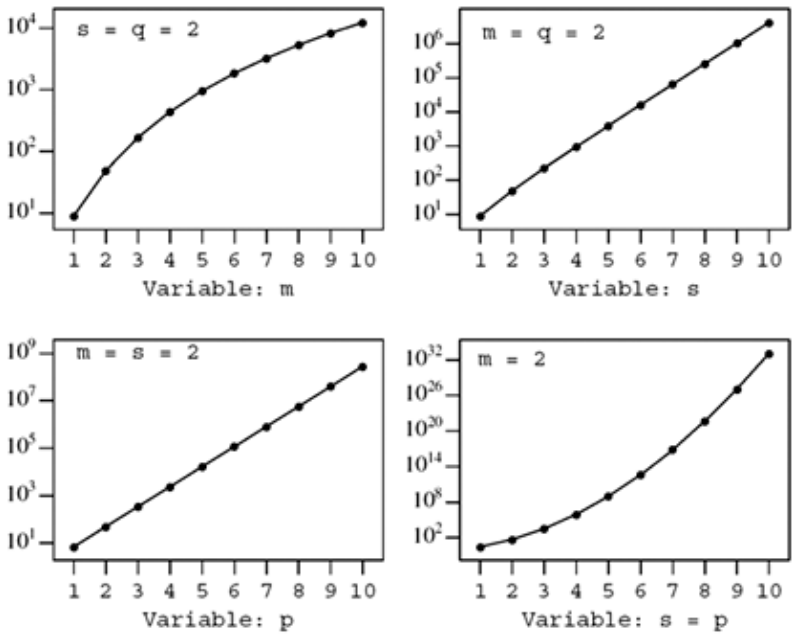
Example

Let us consider the complexity that can be introduced by a single useful type of abstract data object that is commonly used in distributed systems: a message channel or buffer. Let q be the number of buffers we have, let s be the maximum number of messages we can store in each buffer, and let m be the number of different message types that can be used. In how many different states can this set of data objects be? Each buffer can hold between zero and s messages, with each message being a choice of one out of m, therefore, the number of states RQ is:

$$R_Q = \left(\sum_{i=0}^{s} m^i\right)^q.$$

Figure 5.8 shows how the number of states varies for different choices of the parameters q, s, and m. In the top–left graph of Figure 5.8, the parameters s and q are fixed to a value of 2, and the number of message types is varied from 1 to 10. There is a geometric increase in the number of states, but clearly not an exponential one. In the top–right graph, the parameters m and q are fixed to a value of 2, and the number of queue slots s is varied. This time there is an exponential increase in the number of states. Similarly, in the bottom–left graph, the parameters m and s are fixed, and the number of queues is varied. Again, we see an exponential increase in the number of states. Worse still, in the bottom–right graph of the figure, only the number of message types is fixed and the parameters s and q are equal and varied from 1 to 10. As can be expected, the increase is now doubly exponential. The number of possible states quickly reaches astronomical values.

Figure 5.8. Number of Possible States for q Message Buffers with s Buffer Slots and m Message Types



Exponential effects work both ways. They can quickly make simple correctness properties of an uncarefully constructed model computationally intractable, but they can also help the model builder to prove subtle properties of complex systems by controlling just a few carefully chosen parameters.

The size of the available memory on a computer unavoidably restricts the size of the largest problems we can verify. We can try clever encoding and storage options for state space information, but at some point either the machine will run out of available memory or the user will run out of time, waiting for a verification run to complete. If the system is too complex to be analyzed exhaustively, we have no choice but to model it with a system that has fewer states. The tools that are used to accomplish this are: reduction, abstraction, modularity, and structure.

The existence of the state explosion phenomenon we have sketched above should never be used to excuse a designer from proving that a concurrent system fulfills its correctness requirements. It may well be considered to be the very objective of design verification to construct a tractable model and to formalize its properties. After all, since the final verification is a purely mechanical task, model building is the only real problem that the human designer must tackle.

A Formal Basis for Reduction

The SPIN verification algorithms work by detecting the presence of a counterexample to a correctness claim. If we want to prove that p holds, we can use SPIN to try to find a counterexample where the negation of p holds. For temporal logic formulae the same principle is applied. Instead of proving that there exist behaviors for which a given temporal logic formula is valid, SPIN tries to do the opposite: it attempts to find at least one behavior for which the negation of the formula is satisfied. If no counterexample can be found in an exhaustive verification, the formula is proven valid for all possible behaviors.

Call E the set of all possible runs of a system. The verification algorithms we have discussed demonstrate that either E does not contain any runs that violate a correctness requirement, or they provide positive proof that at least one such run exists. The verifier need not search for all possible runs in which the correctness requirement is satisfied, and indeed it often cannot do so.

This means that it is possible to add runs to E without affecting the validity of the proof, provided of course that we do not remove or alter any of the existing runs in E. We will use an example to demonstrate a number of strategies that can be used to reduce the complexity of a verification task by adding runs to E, such that in all cases $E \subseteq E'$. This principle is formalized in the following property.[2]

> [2] The formal definitions for the terms "finite state automaton," "run," and "correctness property" are given in Chapter 6.

Property 5.1 (Reduction Property)

> Given two finite state automata T and T', with sets of runs E and E', respectively. If $E \subseteq E'$ then any correctness property proven for T' necessarily also holds for T.

Proof: The violation of a correctness property for T is not possible without the existence of a run in E that demonstrates it. If no such run exists in E' then no such run can exist in E either, since E' includes E. □

We will see that abstractions of this type can dramatically reduce the number of reachable states of a system. Note that we can generalize a problem by removing constraints from it. The behavior of a model that is less specific often can be represented with fewer states. The least specific model would be one that imposes no constraints whatsoever on, for instance, the messages it can send. It can be represented by a one−state demon that randomly generates messages within its vocabulary, one by one, in an infinite sequence.

Example – A File Server

Assume our task is to verify the correctness of a transfer protocol that is used to access a remote file server. Our first obligation is to determine precisely which correctness properties the transfer protocol must have, and what may be assumed about the behavior of the file server and of the transmission channel.

Consider first the transmission channel. Assume the channel is an optical fiber link. The protocol verifier's job is not to reproduce the behavior of the fiber link at the finest level of detail. The quality of a verification does not improve with a more detailed model of the link. Just the opposite is the case. This is worth stating explicitly:

> A less detailed verification model is often more tractable, and allows for more general, and thus stronger, proofs.

A verification model should represent only those behaviors that are relevant to the verification task at hand. It need not contain information about the causes of those behaviors. If, in the file server example, the fiber link has a non−zero probability of errors, then the possibility of errors must be present in our model, but little more. The types of errors modeled could include disconnection, message−loss, duplication, insertion, or distortion. If all these types of errors are present, and relevant to the verification task at hand, it should suffice to model the link as a one−state demon that can randomly disconnect, lose, duplicate, insert, or distort messages.

A fully detailed model of the link could require several thousand states, representing, for instance, the clustering of errors, or the nature of distortions. For a design verification of the transfer protocol, however, it not only suffices to represent the link by a one−state demon: doing so guarantees a stronger verifi−cation result that is independent of clustering or distortion effects. Clearly, a model that randomly produces all relevant events that can be part of the real link behavior satisfies the requirements of Property 5.1. Of course, the random model of a link can contribute artificial behaviors where specific types of errors are repeated without bound. Our verification algorithms, however, provide the means to prune out the uninteresting subsets of these behaviors. If, for instance, we mark message loss as a pseudo progress event and start a search for non−progress cycles, we can secure that every cyclic execution that is reported by the verifier contains only finitely many message loss events.

Next, consider the file server. It can receive requests to create and delete, open and close, or read and write distinct files. Each such request can either succeed or fail. A read request on a closed file, for instance, will fail. Similarly, a create or write request will fail if the file server runs out of space. Again, for the verification of the interactions with the file server, we need not model in detail under what circumstances each request may succeed or fail. Our model of the server could again be a simple one−state demon that randomly accepts or rejects requests for service, without even looking at the specifics of the request.

Our one−state server would be able to exhibit behaviors that the real system would not allow, for instance, by rejecting valid requests.[3] All behaviors of the real server, however, are represented in the abstract model. If the transfer protocol can be proven correct, despite the fact that our model server may behave worse than the real one, the result is stronger than it would have been if we had represented the server in more detail. By generalizing the model of the file server, we separate, or shield, the correctness of the transfer protocol from the correctness requirements of the server. Again, the model that randomly produces all relevant events satisfies the requirements of Property 5.1.

[3] Remember that it is not the file server's behavior we are verifying, but the behavior of the transfer protocol. If the file server would have been the target of our verification, we would try to model it in more detail and generalize the transfer protocol that accesses it.

Finally, let us consider the number of message types and message buffers that are needed to represent the interaction of user processes with the remote file server. If no single user can ever have more than one request outstanding, we need minimally three distinct types of messages, independent of how many distinct services the remote system actually offers. The three message types are request, accept, and reject.

If there are q users and only one server, the server must of course know which response corresponds to which request. Suppose that we use a single buffer for incoming requests at the server, and mark each request with a parameter that identifies the user. This gives q distinct types of messages that could arrive at the server. If q x s is the total number of slots in that buffer, the number of distinct states will be:

$$\sum_{i=0}^{q \times s} q^i.$$

What if we replaced the single buffer with q distinct buffers, each of s slots, one for each user? Now we need only one type of request, and the number of buffer states is now $(s + 1)^q$. Which is better?

Note that every feasible state of the multiple buffers can be mapped to a specific state of the single buffer, for instance, by simply concatenating all s slots of all q buffers, in numerical order, into the q x s slots of the single buffer. But the single buffer has many more states, that is, all those states that correspond to arbitrary interleavings of the contents of the multiple buffers. With these parameters, then, it can make a large difference in complexity if we replace a single buffer with a set of buffers.

To get an idea of the difference, assume s = 5 and q = 3; then the total number of states of all multiple buffers combined is $(s + 1)^q = 6^3 = 216$, and the total number of states of the single buffer is or about five orders of magnitude larger. Of course, in all these cases it remains the responsibility of the model builder to make certain that only abstractions are made that are independent of the property to be proven, and that satisfy the requirements of Property 5.1.

$$\sum_{i=0}^{q \times s} q^i = \sum_{i=0}^{15} 3^i = 21,523,360$$

Assuming that we have the smallest possible model that still captures the essential features of a system, is there anything more we can do to reduce the complexity of the verification task still further? Fortunately, the answer is yes. In the next two sections, we will discuss two such techniques. The first technique modifies the verification algorithms we have developed in the previous chapter in such a way that redundant work can be avoided without diminishing the accuracy or the validity of the verification itself. The second technique is meant for cases where all the other reduction methods have failed, and we are faced with a verification problem that is still too complex to handle within the confines of available computational resources. This technique attempts to maximize the accuracy of a verification for those cases where no exact proofs are possible.

In Summary

The goal of this chapter is to show that applying model checking tools in a focused and targeted manner can be far more effective than blindly applying them as brute force reachability analyzers.

In a nutshell, the application of model checking in a design project typically consists of the following four steps:

- First, the designer chooses the properties (the correctness requirements) that are critical to the design.
- Second, the correctness properties are used as a guideline in the construction of a verification model. Following the principle of the smallest sufficient model, the verification model is designed to capture everything that is relevant to the properties to be verified, and little else. The power of the model checking approach comes in large part from our ability to define and use abstractions. Much of that power may be lost if we allow the verification model to come too close to the specifics of an implementation.
- Third, the model and the properties are used to select the appropriate verification method. If the model is very large, this could mean the choice between a precise verification of basic system properties (such as a check for absence of deadlock and the correctness of all process and system assertions), or a more approximate check of more complex logical and temporal properties.
- Fourth, the result of the verification is used to refine the verification model and the correctness requirements until all correctness concerns are adequately satisfied.

In the construction of a verifiable model it is good to be aware of the main causes of combinatorial complexity: the number and size of buffered channels, and the number of asynchronously executing processes. We can often bring the complexity of a verification task under control by carefully monitoring and adjusting these few parameters.

We return to the topic of abstraction in Chapter 10, where we consider it in the context of automated model extraction methods from implementation level code.

Bibliographic Notes

The solution to the disk scheduler problem discussed in this chapter is based on Villiers [1979].

The importance of abstraction in verification is generally recognized and features prominently in many papers in this area. Foundational work goes back to the early work of Patrick and Radhia Cousot on abstract interpretation, e.g., Cousot and Cousot [1976].

A detailed discussion of the theoretical background for abstraction is well beyond the scope of this book, but a good starting point for such a discussion can be found in, for instance, the work of Abadi and Lamport [1991], Kurshan [1993], Clarke, Grumberg, and Long [1994], Graf and Saidi [1997], Dams [1996], Dams, Gerth, and Grumberg [1997], Kesten and Pnueli [1998], Das, Dill, and Park [1999], and in Chechik and Ding [2002]. A good overview can also be found in Shankar [2002]. A general discussion of the role of abstraction in applications of SPIN is also given in Holzmann [1998b], from which we have also derived some of the examples that were used in this chapter.

An interesting discussion of the use of abstraction techniques that exploit symmetry in systems models to achieve a reduction in the complexity of verifications can be found in Ip and Dill [1996].