Getting started with SMV

K. L. McMillan
Cadence Berkeley Labs
2001 Addison St.
Berkeley, CA 94704
USA
mcmillan@cadence.com

March 23, 1999

Abstract

This tutorial introduces the SMV verification system. It includes examples of temporal logic model checking, and refinement verification, including techniques of circular compositional proof, temporal case splitting, symmetry reduction, data type reduction and induction.

1 Introduction

This is a short tutorial introduction to SMV, a verification system for hardware designs. SMV is a formal verification tool, which means that when you write a specification for a given system, it verifies that every possible behavior of the system satisfies the specification. This is in contrast to a simulator, which can only verify the system's behavior for the particular stimulus that you provide.

A specification for SMV is a collection of properties. A property can be as simple as a statement that a particular pair of signals are never asserted at the same time, or it might state some complex relationship in the values or timing of the signals. Properties are specified in a notation called *temporal logic*. This allows concise specifications about temporal relationships between signals. Temporal logic specifications about finite state systems can be automatically formally verified by a technique called *model checking*.

SMV is quite effective in automatically verifying properties of combinational logic and interacting finite state machines. Sometimes, when checking properties of complex control logic, the verifier will produce a counterexample. This is a behavioral trace that violates the specified property. This makes SMV a very effective debugging tool, as well as a formal verification system.

Model checking by itself is limited to fairly small designs, because it must search every possible state that a system can reach. For large designs, especially those including substantial data path components, the user must break the correctness proof down into parts small enough for SMV to verify. This is known as *compositional* verification. SMV provides a number of tools to help the user reduce the verification of large, complex systems to small finite state problems. These techniques include *refinement verification*, *symmetry reduction*, *temporal case splitting*, *data type reduction*, and *induction*.

This tutorial will introduce all of the above techniques by example.

2 Modeling, specifying and verifying

We will start with some very simple examples, to illustrate the process of entering a model, specifying properties, and running SMV to verifying them. You can enter the examples yourself, using a text editor (and thus become acquainted with SMV's response to syntax errors). Or, if you are reading this tutorial on-line, you can follow the hyperlinks to the corresponding files.

Consider, for example, the following description of a very simple combinational circuit, with some assertions added. This example is written in SMV's native language. Use a text editor to enter the following program into a file called "prio.smv".

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;

  ack1 := req1;
  ack2 := req2 & ~req1;
```

```
  mutex : assert ~(ack1 & ack2);
  serve : assert (req1 | req2) -> (ack1 | ack2);
  waste1 : assert ack1 -> req1;
  waste2 : assert ack2 -> req2;
}
```

This example shows most of the basic elements of an SMV module. The module has four *parameters*, `req1`, req2, `ack1` and `ack2`, of which the former two are inputs, and the latter two outputs. It contains:

- *Type declarations.* In this case the signals `req1`, req2, `ack1` and `ack2` are declared to be of type `boolean`.
- *Signal assignments.* These give logic functions for outputs `ack1` and `ack2` in terms of inputs `req1` and `req2`.
- *Assertions.* These are properties to be proved.

The program models a (highly trivial) two bit priority-based arbiter, which could be implemented with a two-gate circuit. The `assert` statements specify a number of properties that we would like to prove about this circuit. For example, the property called `mutex` says that outputs `ack1` and `ack2` are not true at the same time. Note that `&` stands for logical "and" while `~` stands for logicol "not". The property `serve` says that if either input `req1` or `req2` is true, then one of the two outputs `ack1` or `ack2` is true. Note that `|` stands for logical "or", while `->` stands for "implies". Logically, `a -> b` is equivalent to `~a | b`, and can be read "`a` implies `b`" or "if `a` then `b`".

We would like to verify these specifications formally, that is, for all possible input patterns (of which in this case there are only four). To do this under Unix, enter the following shell command:

```
vw prio.smv
```

On a PC under Windows, double-click the icon for the file "prio.smv". This will start the SMV viewer, called "vw", with the file "prio.smv". This interface has a number of tabbed pages, which can be accessed by clicking an the appropriate tab. When you start the interface, you see the browser, which is a tree representation of all the signals and assertions in your source file, and the source page, which shows the source file. If you made a syntax error in the source file, this error will be pointed out on the source page. Correct the error, and then choose "Reopen" from the "File" menu.

If you have no syntax errors, expand `top level` in the browser by double-clicking it, or by clicking the `+` icon. The `+` indicates that `top level` has children which are not currently visible. You should see under `top level` the names of all the signals and properties in your source file. Since none of these has children, they will not be marked with a `+`. Select one of these, and notice the highlight in the source page moves to the location in the program where that signal or property is declared. Select the signal `ack2`, and then in the source page, select "Where assigned" in the "Show" menu. The souce line where `ack2` is assigned will now be highlighted.

Now select "Verify all" from the "Prop" menu. SMV verifies the four properties in our program. The results page now shows the results of this verification run. In this case, all the properties are true.

Now let's modify the design so that one of the specifications is false. For example, change the line

```
ack1 := req1;
```

to

```
ack1 := req1 & ~req2;
```

Save the modified text file and choose "Reopen" from the "File" menu (or, if you are on-line, just click here to save typing). Then select "Prop—Verify all" again. Notice that this time the property `serve` is false. Also note, not all of the properties appear in the results pane. This is because SMV stops when it reaches the first property that is false. Thus, not all the properties were checked.

When a property is false, SMV produces a counterexample that shows a case when it doesn't hold. To see the counterexample for `serve`, select it in the results page by clicking on it. The trace page will appear, showing a counterexample – a truth assignment to all the signals that shows that our property is false. The counterexample shows the case when both inputs are true and both outputs are false.

The verifier keeps track of which properties have been verified since the most recent source file change. You can see which properties have been verified thus far, by selecting the properties page. Currently only `mutex` is verified. To verify `waste1`, for example, click on it in the properties page, and then choose "Verify waste1" from the "Prop" menu. Notice that only the property you select is verified in this case. The name of the property that is currently selected appears at the bottom of the window.

2.1 Sequential circuits and temporal properties

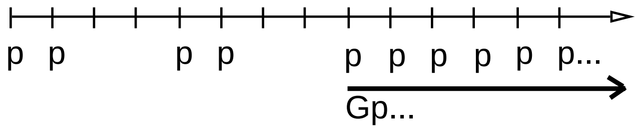
To specify sequential circuits, we need to be able to make statements about how signals evolve over time. SMV uses a notation called *temporal logic* for this purpose. Temporal logic formulas are like formulas in ordinary boolean logic, except that truth value of a formula in temporal logic is a function of time. Some new operators are added to the traditional boolean operators "and", "or", "not" and "implies", in order to specify relationships in time.

For example, the `F` operator is used to express a condition that must hold true at some time in the future. The formula `F p` is true at a given time if `p` is true at some later time. On the other hand, `G p` means that `p` is true at all times in the future. Usually, we read `F p` as "eventually `p`" and `G p` as "henceforth `p`".

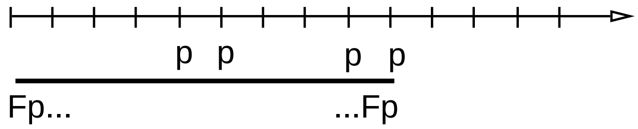
In addition, we have the "until" operator and the "next time" operator. The formula `p U q`, which is read "`p` until `q`" means that `q` is eventually true, and until then, `p` must always be true. The formula `X p` means that `p` is true at the next time.

Here are the exact definitions of the temporal logic operators, with example time lines showing the states when they hold true:

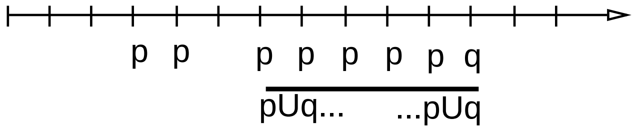
- The "globally" operator: `G p` is true at time t if `p` is true at *all* $t' \geq t$.



- The "future" operator: `F p` is true at time t if `p` is true at *some* $t' \geq t$.



- The "until" operator: `p U q` is true at time t iff
  - `q` is true at *some* $t' \geq t$, and
  - `p` is true in the range $[t, t')$



- The "next time" operator: `X p` is true at time t if `p` is true at time $t + 1$.

As an example, suppose we are designing a bus arbiter with two "grant" signals, `ack1` and `ack2`. Among other things, we want to specify that the two grant signals are never asserted at the same time. In temporal logic, we would write `G ~(ack1 & ack2)`. The `G` operator is used to say that our specification should hold true at all times. This is needed because SMV interprets "assert" statements to hold at the initial state of execution of the program. If we wrote only `~(ack1 & ack2)`, SMV would interpret this only to mean that both grants may not be asserted at time $t = 0$.

Return to our original example, and edit the four properties we specified so that they begin with the `G` operator. Make sure to undo the error we introduced in the previous section. You should have something like this:

```
module main(req1,req2,ack1,ack2)
{
  input req1,req2 : boolean;
  output ack1,ack2 : boolean;

  ack1 := req1;
  ack2 := req2 & ~req1;

  mutex  : assert G ~(ack1 & ack2);
  serve  : assert G ((req1 | req2) -> (ack1 | ack2));
  waste1 : assert G (ack1 -> req1);
```

```
  waste2 : assert G (ack2 -> req2);
}
```

Open the file and choose "Prop—Verify all" again to confirm that the properties we specified in fact hold true for all time. This is because the two logic equations we wrote for `ack1` and `ack2` hold implicitly for all time. Now let's write a more interesting temporal specification. Suppose we want to use our priority circuit as a bus arbiter. In addition to the above properties, we would like to avoid "starvation" of the low priority requester. That is, we don't want `req2` to be asserted forever while `ack2` is never asserted. Put another way, we want it to always eventually be true that either `req2` is negated or `ack2` is asserted. In temporal logic, we write "always eventually" by combining `G` and `F`. In this case we assert: `G F (~req2 | ack2)`. Therefore, add the following specification to the program:

```
  no_starve : assert G F (~req2 | ack2);
```

Now open the new version and verify the property `no_starve`. The property should be false, and a counterexample trace with one state should appear in the trace page. Notice that the state number is marked with "repeat" signs, thus: `|:  1 :|`. This is to indicate that the first state repeats forever. In this state, both `req1` and `req2` are asserted. Since `req1` has priority, `ack2` is never asserted, hence requester 2 "starves".

As an aside, you might also have observed that the signal `ack1` doesn't appear in the trace. This is because SMV noticed that the property `no_starve` doesn't actually depend on this signal, so it left `ack1` out of its analysis. The set of signals that a property depends on is referred to as the *cone* of that property. When you have selected a given property to verify, you can view the cone of that property by clicking the "Cone" tab. In this case, you'll notice that the signals `req1` and `req2` are listed as "free". This is because they are unconstrained inputs to the circuit, and thus are free to take on any values in their type. These signals each contribute one "combinational" variable to the verification problem. SMV must verify the property you specified for all possible combinations of these variables. Thus, it is generally best to keep the number of variables in the cone small, when possible.

Now, to prevent this starvation case, let's add a latch to the circuit that remembers whether `ack1` was asserted on the previous cycle. In this case we'll give priority to requester 2 instead. To do this, add the following code to the program:

```
  bit : boolean;
  next(bit) := ack1;
```

The above means that `bit` is a boolean variable, and that the value of `bit` at time $t+1$ is equal to the value of `ack1` at time t. This is how a state variable (or a register, if you like) is represented to SMV – as an equation involving one time unit of delay. Now, replace the definitions of `ack1` and `ack2` with the following:

```
  if (bit) {
    ack1 := req1 & ~req2;
    ack2 := req2;
  }
```

```
  else {
    ack1 := req1;
    ack2 := req2 & ~req1;
  }
```

That is, when `bit` is set, we reverse the priority order. Note that even though this may look like a sequential program, it really represents two simultaneous equations. If you like, you can write the same thing instead like this:

```
    ack1 := bit ? req1 & ~req2 : req1;
    ack2 := bit ? req2 : req2 & ~req1;
```

Now open the new version and verify property `no_starve`. It should be true. By the way, you might have noticed that we didn't specify an initial (*i.e.* reset) value for the register `bit`. In fact, SMV verified `no_starve` for both possible initial values. If you check the "Cone" panel, you'll notice that there are now two combinational variables (the inputs) and one state variable (the signal `bit`).

2.2 A three-way arbiter

Now let's try to apply the same idea to a three-way bus arbiter. In this version, we will have one latched bit for each requester. This bit holds a one when the corresponding requester was granted the bus on the previous cycle. We'll still use a fixed priority scheme, but if a given request was granted on the previous cycle, we'll give it lowest priority on the current cycle. Thus, if the bit for a given requester is set, its request is served only if no others are requesting. Further, the requester with its bit set does not inhibit lower priority requesters. Here is one attempt at such an arbiter:

```
module main(req1,req2,req3,ack1,ack2,ack3)
{
  input req1,req2,req3 : boolean;
  output ack1,ack2,ack3 : boolean;

  bit1,bit2,bit3 : boolean;

  next(bit1) := ack1;
  ack1 := req1 & (bit1 ? ~(req2 | req3) : 1);

  next(bit2) := ack2;
  ack2 := req2 & (bit2 ? ~(req1 | req3) : ~(req1 & ~ bit1));

  next(bit3) := ack3;
  ack3 := req3 & (bit3 ? ~(req2 | req3) :
                          ~(req2 & ~bit2 | req1 & ~bit1));
}
```

The specifications for the three-way arbiter are as follows:

```
mutex : assert G ~(ack1 & ack2 | ack1 & ack3 | ack2 & ack3);
serve : assert G ((req1 | req2 | req3) -> (ack1 | ack2 | ack3));
waste1 : assert G (ack1 -> req1);
waste2 : assert G (ack2 -> req2);
waste3 : assert G (ack3 -> req3);

no_starve1 : assert G F (~req1 | ack1);
no_starve2 : assert G F (~req2 | ack2);
no_starve3 : assert G F (~req3 | ack3);
```

They are similar to the two-way case, but note that in `mutex` we consider all pairs. Also, we've specified non-starvation for all of the requesters, just in case. Save this program in a file (you can put the specifications anywhere inside the `module` declaration – statement order is irrelevant in SMV). Then open the file and choose "Verify all". You should get a false result for `no_starve3`. Click on `no_starve3` and observe the counterexample trace. This is an example of a "livelock". The last two states in the counterexample repeat forever. Notice that requesters 1 and 2 are served alternately while requester 3 starves.

In fact, there is another error in the design. If you select the `serve` property and try to verify it, you'll find that `serve` can be false in the initial state. This occurs if more than one of the `bits` are true initially. We could rule this out by specifying initial values for these bits, as follows:

```
init(bit1) := 0;
init(bit2) := 0;
init(bit3) := 0;
```

Alternatively, if we don't care if no one gets served in the initial state, we can change the specification. In temporal logic `X p` means that `p` is true at the "next" time. Thus, for example `X G p` means that `p` holds from the second state onward. Thus, we could change the specification to:

```
serve : assert X G ((req1 | req2 | req3) -> (ack1 | ack2 | ack3));
```

As an exercise, you might want to try designing and verifying a three-way arbiter that satisfies all the specifications above.

2.3 A traffic light controller

Now we'll consider a slightly more complex example that uses some additional features of SMV's language. The example is a controller that operates the traffic lights at an intersection where two-way street running north and south intersects a one-way street running east. The goals are to design the controller so that collisions are avoided, and no traffic waits at a red light forever.

The controller has three traffic sensor inputs, `N_Sense`, `S_Sense` and `E_Sense`, indicating when a car is present at the intersection traveling in the north, south and east directions respectively. There are three outputs, `N_Go`, `S_Go` and `E_Go`, indicating that a green light should be given to traffic in each of the three directions.

```
module main(N_Sense,S_Sense,E_Sense,N_Go,S_Go,E_Go){
  input N_Sense,S_Sense,E_Sense : boolean;
  output N_Go,S_Go,E_Go : boolean;
```

In addition, there are four internal registers. The register `NS_Lock` is set when traffic is enabled in the north or south directions, and prevents east-going traffic from being enabled. The three bits `N_Req`, `S_Req`, `E_Req` are used to latch the traffic sensor inputs.

```
  NS_Lock, N_Req, S_Req, E_Req : boolean;
```

The registers are initialized as follows:

```
  init(N_Go) := 0;
  init(S_Go) := 0;
  init(E_Go) := 0;
  init(NS_Lock) := 0;
  init(N_Req) := 0;
  init(S_Req) := 0;
  init(E_Req) := 0;
```

In modeling the traffic light controller's behavior, we will use two new SMV statements. The `case` statement is a conditional form. The sequence:

```
  case{
    cond1 : {block1}
    cond2 : {block2}
    cond3 : {block3}
  }
```

is equivalent to

```
  if (cond1) {block1}
  else if (cond2) {block2}
  else if (cond3) {block3}
```

In addition, we will use the `default` construct to indicate that certain assignments are to be used as defaults when the given signals are not assigned in the code that follows. In a sequence like this:

```
  default {block1}
  in {block2}
```

assignments in `block2` take precedence over assignments in `block1`. SMV enforces a "single assignment rule", meaning that only one assignment to a given signal can be active at any time. Thus, if we have more than one assignment to a signal, we must indicate which of the two takes precedence in case both apply.

Now, returning to the traffic controller, if any of the sense bits are true, we set the corresponding request bit:

```
default{
  if(N_Sense) next(N_Req) := 1;
  if(S_Sense) next(S_Req) := 1;
  if(E_Sense) next(E_Req) := 1;
}
```

The code to operate the north-going light is then as follows:

```
in default case{
  N_Req & ~N_Go & ~E_Req : {
    next(NS_Lock) := 1;
    next(N_Go) := 1;
  }
  N_Go & ~N_Sense : {
    next(N_Go) := 0;
    next(N_Req) := 0;
    if(~S_Go) next(NS_Lock) := 0;
  }
}
```

This says that if a north request is latched, and the north light is not green and there is no east request, then switch on the north light and set the lock (in effect, we give priority to the east traffic). If the north light is on, and there is no more north traffic, switch off the light, clear the request, and switch off the lock. Note however, that if the south light is on, we don't switch the lock off. This is to prevent south and east traffic from colliding. The south light code is similar:

```
in default case{
  S_Req & ~S_Go & ~E_Req : {
    next(NS_Lock) := 1;
    next(S_Go) := 1;
  }
  S_Go & ~S_Sense : {
    next(S_Go) := 0;
    next(S_Req) := 0;
    if(~N_Go) next(NS_Lock) := 0;
  }
}
```

Finally, the east light is switched on whenever there is an east request, and the lock is off. When the east sense input goes off, we switch off the east light and reset the request bit:

```
in case{
  E_Req & ~NS_Lock & ~E_Go : next(E_Go) := 1;
  E_Go & ~E_Sense : {
    next(E_Go) := 0;
    next(E_Req) := 0;
```

```
    }
  }
```

There are two kinds of specification we would like to make about the traffic light controller. The first is a “safety” specification that say that lights in cross directions are never on at the same time:

```
  safety: assert G ~(E_Go & (N_Go | S_Go));
```

The second is a “liveness” specification, for each direction, which says that is the traffic sensor is on for a given direction, then the corresponding light is eventually on, thus no traffic waits forever at a read light:

```
  N_live: assert G (N_Sense -> F N_Go);
  S_live: assert G (S_Sense -> F S_Go);
  E_live: assert G (E_Sense -> F E_Go);
```

Note, however, that our traffic light controller is designed so that it depends on drivers not waiting forever at a green light. We want to verify the above properties given that this assumption holds. To do this, we write some “fairness constraints”, as follows:

```
  N_fair: assert G F ~(N_Sense & N_Go);
  S_fair: assert G F ~(S_Sense & S_Go);
  E_fair: assert G F ~(E_Sense & E_Go);
```

Each of these assertions states that, always eventually, it is not the case that a car is at a green light. To tell SMV to assume these “fairness” properties when proving the “liveness” properties, we say:

```
  using N_fair, S_fair, E_fair prove N_live, S_live, E_live;
  assume E_fair, S_fair, N_fair;
}
```

Because of the `assume` statement, the fairness constraints themselves will simply be left unproved. Now, open this file and try to verify the property `safety`. The result should be “false”, and in the “Trace” panel, you should see a counterexample trace in which the south light goes off exactly at the time when the north light goes on. The result is that the lock bit is cleared. This is because the code for the south light takes precedence over the code for the north light, due to our use of `default`. With the north light on and the lock cleared, the east light can now go on, violating the safety property.

To fix this problem, let’s change the south light code so that it tests to see whether that north light is about to go on before clearing the lock. Here is the revised code for the south light:

```
  in default case{
    S_Req & ~S_Go & ~E_Req : {
      next(NS_Lock) := 1;
      next(S_Go) := 1;
```

```
    }
    S_Go & ~S_Sense : {
      next(S_Go) := 0;
      next(S_Req) := 0;
      if(~(N_Go | N_Req & ~N_Go & ~E_Req)) next(NS_Lock) := 0;
    }
  }
```

Open this new version and verify the property `safety`. It should be true. Now try to verify `N_live`. It should come up false, with a counterexample showing a case where both the north and south lights are going off at exactly the same time. In this case neither the north code nor the south code clears the lock, because each thinks that the other light is still on. As a result, the lock remains on, which prevents an east request from being served. Since the east request takes priority over north and south requests, the controller is deadlocked, and remains in the same state indefinitely (note the "repeat signs" on the last state).

To fix this problem, we'll give the north light controller the responsibility to turn off the lock when both lights are going off. Here's the new north light code:

```
  in default case{
    N_Req & ~N_Go & ~E_Req : {
      next(NS_Lock) := 1;
      next(N_Go) := 1;
    }
    N_Go & ~N_Sense : {
      next(N_Go) := 0;
      next(N_Req) := 0;
      if(~S_Go | ~S_Sense) next(NS_Lock) := 0;
    }
  }
```

Open this new version and verify the properties `safety`, `N_live`, `S_live` and `E_live`. They should all be true. Note that if you try to verify the fairness constraints `N_fair`, `S_fair` and `E_fair`, they will come up false. These are unprovable assumptions that we made in designing the controller. However, if we used the controller module in a larger circuit, we could (and should) verify that the environment we put the controller into actually satisfies these properties. In general, it's best to avoid unproved assumptions if possible, since if any of these assumptions is actually false, all the properties we "proved" are invalid.

3 Symbolic model checking

A model checker verifies a property by building a graph of all of the states in the model. In SMV, the number of states in the model is 2^n, where n is the number of state variables in the cone of the property. In fact, it is only necessary for the model checker to consider the states that are "reachable" from an initial state. However, as you might expect, the amount of computational effort required to verify a property still tends to grow very rapidly with the number of state variables. This is known as the "state explosion problem".

To address this problem, SMV uses a structure called a “Binary Decision Diagram” (BDD) to implicitly represent the state graph of the model, and sets of states satisfying given properties. For some models and properties, the use of BDD’s (implicit enumeration) allows SMV to handle models with many orders of magnitude more states than could be handled by considering individual states (explicit enumeration). First, we see a simple example of a circuit with a very large number of states that can still be handled efficiently using BDD’s. Later we’ll consider what to do when a direct approach using BDD’s doesn’t work.

3.1 A buffer allocation controller

This example is designed to control the allocation and freeing of buffers in, for example, a packet router. The controller keeps an array of “busy” bits, one for each available data buffer. The busy bit is true when the buffer is in use, and false otherwise. An input `alloc` indicates a request to allocate a new buffer for use. If there is a buffer available, the controller outputs the index of this buffer on a signal `alloc_addr`. If there is no buffer available, it asserts an output `nack`. To make the circuit a little more interesting, we’ll add a counter that keeps track of the number of busy bits that are set. Thus `nack` is asserted when the count is equal to the total number of buffers. To begin with, we’ll define the number of buffers to be 32, using a C-style macro definition:

```
#define SIZE 32
module main(alloc,nack,alloc_addr,free,free_addr)
{
  input alloc : boolean;
  output nack : boolean;
  output alloc_addr : 0..(SIZE - 1);
  input free : boolean;
  input free_addr : 0..(SIZE - 1);

  busy : array 0..(SIZE - 1) of boolean;
  count : 0..(SIZE);

  init(busy) := [0 : i = 0..(SIZE-1)];
  init(count) := 0;
```

Note that we initialized `busy` to a vector of 32 zeros using an iterator expression. Here is the logic for the counter and the `nack` signal. Notice, we add one to the counter when there is an allocation request and `nack` is not asserted. We subtract one from the counter when there is a free request, and the buffer being freed is actually busy. Note, if we didn’t check to see that the freed buffer is actually busy, the counter could get out of sync with the busy bits.

```
  nack := alloc & (count = SIZE);
  next(count) := count + (alloc & ~nack) - (free & busy[free_addr]);
```

Next we handle the setting and clearing of the busy bits. We use a `default` statement to indicate that, if a given buffer is being both freed and allocated at the same time, the allocation request takes precedence.

```
  default{
    if(free) next(busy[free_addr]) := 0;
  } in {
    if(alloc & ~nack) next(busy[alloc_addr]) := 1;
  }
```

Finally, we choose a buffer to allocate using a priority encoder. This is most easily generated using the `chain` constructor. This repeats a given block of statements for a range of index values, given precedence to later iterations. So, for example

```
chain (i = 0; i < 3; i = i + 1) block(i)
```

is equivalent to

```
default block(0) in default block(1) in default block(2)
```

Our priority encoder is defined as follows:

```
  chain(i = (SIZE - 1); i >= 0; i = i - 1){
    if(~busy[i]) alloc_addr := i;
  }
```

Since the last statement in the chain is the case `i = 0`, we effectively give highest priority to buffer 0. Note, in the case when all buffers are busy, `alloc_addr` in not assigned, and thus remains undefined.

Now, we consider the problem of specifying the buffer allocator. We will write a separate specification for each buffer, stating that the given buffer is never allocated twice without being freed in the interim. This is a technique known as "decomposition", that is, breaking a complex specification of a system into smaller parts that can be verified separately. To make it simpler to state the specification, it helps to define some additional signals: a bit `allocd[i]` to indicate that buffer `i` is currently being allocated, and a bit `freed[i]` to indicate that buffer `i` is currently being freed:

```
  for(i = 0; i < SIZE; i = i +1){
    allocd[i], freed[i] : boolean;

    allocd[i] := alloc & ~nack & alloc_addr = i;
    freed[i] := free & free_addr = i;
  }
```

Note, we used a `for` constructor to make an instance of these definitions for each buffer `i`. To write the specification that a buffer is not allocated twice, we can use "until" operator of temporal logic. Recall that the formula `p U q` in temporal logic means that `q` is eventually true, and until then, `p` must always be true.

```
  for(i = 0; i < SIZE; i = i +1){
    safe[i] : assert G (allocd[i] -> ~ X ((~freed[i]) U allocd[i]));
  }
}
```

Here we state that, if buffer `i` is allocated, then it is not the case that, starting at the next time, it remains unfreed until it is allocated a second time.

Now, let's verify this specification. Open the file and verify the property `safety[0]`. This should take something under a minute. If you watch the log output during the verification process, you'll notice that it is reporting a sequence of "iterations". These are the steps of a breadth-first search of the model's state space, starting from the initial states. The numbers reported are the sizes of the BDD's representing the set of states reached thus far in the search. The size of the BDD's can be much smaller than the number of states in the set. To see this, select "Prop—State count". This will rerun the verification and report the number of states reached at each iteration. The final number of states reached in this case is something over two billion.

Now let's increase the number of buffers from 32 to 64. Change the definition of `SIZE` at the beginning of the program to

```
#define SIZE 64
```

Open the new version, select the property `safety[0]`, and then select "Prop—State count". This will verify the property, and also compute the number of states reached. You might want to go make a cup of coffee at this point, since the computation will take ten or twenty minutes. The only point to be made here is that the number of states reached is on the order of 10^{19}, while the BDD representing this set of state has about 4000 "nodes". This shows that the BDD's can be a very compact representation for large sate sets. Sometimes, this makes it possible to verify a model, even though the number of states is much too large to be searched "explicitly" (*i.e.* larger than the number of atoms in the universe).

There is no guarantee, however, that SMV's BDD-based algorithms will be able solve a given verification problem. This is because the problem is SMV is trying to solve is fundamentally hard (PSPACE complete, to be precise). On the other hand, when SMV fails to solve a verification problem (or when we run out of patience waiting for it to solve the problem), there are usually many ways to make the problem simpler for SMV to solve. This usually involves decomposition – breaking big problems into small problems, and then localizing the verification of each subproblem to a small part of the overall model. This technique is described in the following section.

4 Refinement verification

Refinement verification is methodology of verifying that the functionality of an abstract system model is correctly implemented by a low-level implementation. It can be used, for example, to verify that a packet router or bus protocol, modeled at the clock-cycle level, correctly implements a given abstract model of end-to-end data transfer. Similarly one can verify that a clock-accurate model of a pipelined, out-of-order processor correctly implements a given instruct-set architecture (*i.e.*, a programmer's model of a machine).

By breaking a large verification problem into small, manageable parts, the refinement methodology makes it possible to verify designs that are much too large to be handled directly by model checking. This decomposition of the verification problem is enabled by specifying *refinement maps* that translate the behavior of the abstract model into the behavior of given

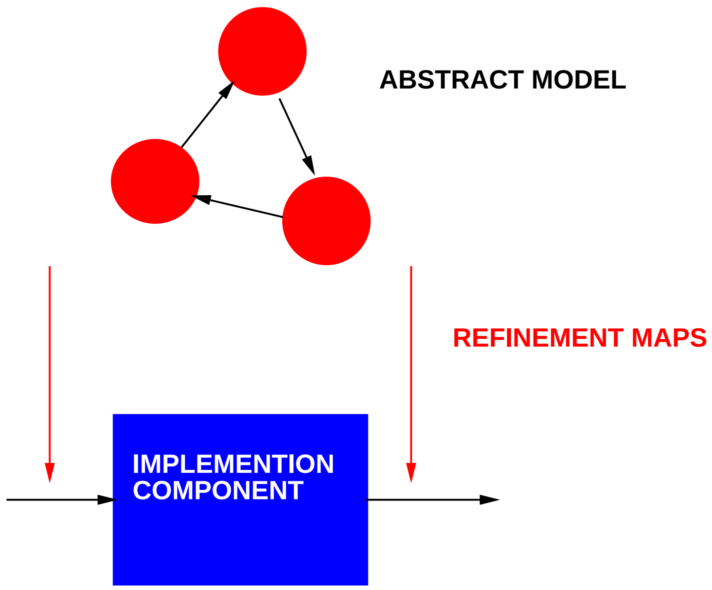


Figure 1: Refinement maps

interfaces and structures in the low-level design. This makes it possible to verify small parts of the low-level design in the context of the abstract model. Thus, the proof obligations can be reduced to a small enough scale to be verified by model checking.

SMV supports this methodology by allowing one to specify many abstract definitions for the same signal. A new construct called a "layer" is introduced for this purpose. A layer is a collection of abstract signal definitions. A layer can, for example, define low-level implementation signals as a function of abstract model signals, and thus provide a refinement map (*i.e.*, a translation between abstraction levels). The low-level implementation of a signal must be simultaneously consistent with all of its abstract definitions. Thus, each abstract definition entails a verification task – to show that every implementation behavior is allowed by this definition. For the purpose of this verification task, one may use whichever abstract definition is most convenient for defining of the other signals. Suppose, for example, that we have abstract definitions of both the inputs and outputs of a given low-level block as a function of a high-level model, as depicted in figure 1. We can use the abstract definitions of the inputs to drive the inputs of the block from the high-level model when verifying that the outputs are consistent with their abstract definitions. Thus, the abstract model provides the context (or environment) for verifying the block, and we do not need to consider the remainder of the low-level model.

SMV also supports design by a successive refinement. One can define a sequence of layers, each of which is more detailed than the previous layer. The implementation of each signal is given by the lowest-level definition in the hierarchy.

4.1 Layers

A layer is a collection of abstract signal definitions. These are expressed as assignments in exactly the same way that the implementation is defined, except that they are bracketed by a `layer` statement, as follows:

```
layer <layer_name> : {
      assignment1;
      assignment2;
      ...
      assignmentn;
}
```

where each assignment is of the form

```
<signal> := <expression>;
```

or

```
next(<signal>) := <expression>;
```

or

```
init(<signal>) := <expression>;
```

High level control structures, such as `if`, `switch` and `for` can also be used inside a `layer` construct, since these are simply “syntactic sugar” for assignments of the above form.

The layer declaration is actually a formal specification, which states that every implementation behavior must be consistent with all of the given assignments. If this is the case, we say the implementation refines the specification.

As an example, let’s consider a very simple example of a specification and implementation of a finite state machine:

```
module main(){
  x : boolean;

  /* the specification */

  layer spec: {
    init(x) := 0;
    if(x=0) next(x) := 1;
    else next(x) := {0,1};
  }

  /* the implementation */

  init(x) := 0;
  next(x) := ~x;
}
```

Note that `spec` is not a keyword here – it is just an arbitrary name given to our specification. This specification is nondeterministic, in that at state 1, it may transition to either state 0 or state 1. The implementation on the other hand has only one behavior, which alternates between state 0 and state 1. Since this is one possible behavior of `spec`, the specification `spec` is satisfied.

If you enter this example into a file, and open the file with `vw`, you will find in the Properties page a single entry named `x//spec`. This is a notation for "the definition of signal `x` in layer `spec`". It appears in the Properties page because it is an obligation to be verified, rather than a part of the implementation. You can verify it by selecting "Prop—Verify all". SMV does this by translating the assignment into an initial condition and transition invariant. The former states that `x` is 0 at time $t = 0$, while the latter states that the value of `x` at time $t + 1$ is 1 if `x` is 0 at time t, and else is either 0 or 1. The implementation must satisfy these two conditions, which are verified by exhaustive search of the state space of the implementation.

If more than one signal is assigned in a layer, then the two definitions are verified separately. This is known as `decomposition`. The reason for using decomposition is that we may be able to use a different abstraction of the implementation to prove each component of the specification. As a very simple example, consider the following program:

```
module main(){
  x,y : boolean;

  /* the specification */

  layer spec: {
    x := 1;
    y := 1;
  }

  /* the implementation */

  init(x) := 1;
  next(x) := y;
  init(y) := 1;
  next(y) := x;
}
```

Both state bits in the implementation start at 1, and at each time they swap values. Thus, the specification is easily seen to be satisfied – both `x` and `y` are always equal to 1. If you open this example with `vw`, you will find two entries in the Properties page: `x//spec` and `y//spec`. Each of these can be verified separately (i.e., we can verify separately that `x` is always equal to 1 and that `y` is always equal to 1). Suppose we want to verify `x//spec` (select it in the Properties page). We now have two choices: we can use either the specification definition of `y` or the implementation definition `y`. Note, however, that if we use the specification definition of `y`, we eliminate one state variable from the model, since `y` is defined to be identically 1.

Thus, by decomposing a specification into parts, and using one part as the "environment" for another, we have reduced the number of state variables in the model, and thus reduced the verification cost (though it is in any event trivial in this case). In fact, if you click on the Cone tab in `vw`, you will see that SMV has selected layer `spec` to define `y`, and that as a result, `y` is not a state variable. This is because SMV assumes by default that it is better to use an abstract definition of a signal than a detailed one. Select "Prop—Verify `x//spec`" to verify the property using this abstraction.

Note that `y//spec` can now be verified using `x//spec` to define `x`. This might at first seem to be a circular argument. However, SMV avoids the potential circularity by only assuming `y//spec` holds up to time $t-1$ when verifying `x//spec` at time t, and *vice versa*. Because of this behavior, we need not be concerned about circularities when choosing an abstract definition to drive a signal. SMV does the bookkeeping to insure that when all components of the specification are declared "verified", then in fact the implementation refines the specification.

4.2 Refinement maps

The most effective way to decompose the specification and verification of a system into manageable parts is to define an abstract model as a specification, and then to specify "refinement maps" that relate abstract model behaviors to implementation behaviors. Generally, abstract models specify "what" is being done, without specifying the "how", "where" or "when". The "where" and "when" are given by the refinement maps, while the implementation determines the "how". In the simplest case the abstract model does nothing at all. For example, in the case of a link-layer protocol that simply transfers a stream of data from point A to point B without modifying it, there is no "what" and the only important information is the "where" and "when". The abstract model in this case might consist only of the stream of data itself. In the case of a microprocessor, the abstract model might determine the sequence of instructions that are executed according to the ISA (instruction set architecture). The refinement map would determine what instruction appears at each stage of the pipeline at any given time.

4.2.1 A very simple example

We will consider first a very simple example of specifying abstractions and refinement maps. Suppose that we would like to design a circuit to transmit an array of 32 bytes from its input to its output, without modifying the array. The abstract model in this case is just an unchanging array of bytes, since no actual operations are performed on the array. The refinement maps specify the protocol by which the array is transferred at the input and output. We'll assume the the input consists of three components: a bit `valid` indication the the input currently holds valid data, an index `idx` that tells which element of the array is currently being transferred, and a byte `data` that gives the value of this element. Assume the output uses a similar protocol. Thus far, we have the following specification:

```
typedef BIT 0..7;
typedef INDEX 0..31;
```

```
typedef BYTE array BIT of boolean;

module main(){

  /* the abstract model */

  bytes : array INDEX of BYTE;
  next(bytes) := bytes;

  /* the input and output signals */

  inp, out : struct{
    valid : boolean;
    idx : INDEX;
    data : BYTE;
  }

  /* the refinement maps */

  layer spec: {
    if(inp.valid) inp.data := bytes[inp.idx];
    if(out.valid) out.data := bytes[out.idx];
  }
```

Note that the abstract model simply states that nothing happens to the array of bytes. The refinement map is partially specified. For example, if `inp.valid` is 0, then `inp.data` is allowed to have any value, since there is no `else` clause in the conditional. You can think of this as a "don't care" case in the specification.

Now let's add a very trivial implementation:

```
  init(out.valid) := 0;
  next(out) := inp;
}
```

That is, the output is just the input delayed by one time unit. Note, at time $t = 0$ we have to signal that the output is not valid, but we don't have to specify initial values for `idx` and `data` since they are "don't cares" in this case.

Save this program in a file and open it with `vw`. Note that there are eight properties in the file, of the form `out.data[i]//spec`, where `i = 0..7`. Select property `out.data[0]//spec`, for example. If you click on the Cone tab, you'll notice that only signals with bit index 0 appear. This is because SMV has detected the property you selected doesn't depend on the other bit indices. Also notice that the data input signal `inp.data[0]` has used layer `spec` for its definition (since this is in fact the only available definition at this point). Thus, we are driving the input of our implementation from the abstract model (through a refinement map) and verifying the output with respect to the abstract model (again through a refinement map). Now, select "Prop—Verify `out.data[0]//spec`". It should take less than 2 seconds to verify this property. You can select "Prop—Verify All" to verify the remainder of the refinement maps. SMV will quickly recognize that the 7 remaining verification problems are isomorphic to the one we just solved, and report "true" for all of them. Note that although we have reduced the number of state bits by a factor of eight by using decomposition (since we only deal with one bit index at a time) we are still using 32 bits out of the data array for each verification. This gives us 39 state bits, which is a fairly large number and guarantees us at least 4 billion states. In this case, the large state space is easily handled by the BDD-based model checker, so we do not have to do any further decomposition. In general however, we cannot rely on this effect. Later we'll see how to decompose the problem further, so that we only use one bit from the data array.

4.2.2 End-to-end verification

Now we'll consider a more complex (though still trivial) implementation with multiple stages of delay. The goal is to verify the end-to-end delivery of data by considering each stage in turn, specifying a refinement map for each stage. The refinement map for each stage drives the input of the next. Suppose we replace the above implementation with the following implementation that has three time units of delay:

```
  stage1, stage2 : struct{
    valid : boolean;
    idx : INDEX;
    data : BYTE;
  }

  init(stage1.valid) := 0;
  next(stage1) := inp;
  init(stage2.valid) := 0;
  next(stage2) := stage1;
  init(out.valid) := 0;
  next(out) := stage2;
```

We'll include a refinement map for each intermediate delay stage, similar to the maps for the input and output:

```
  layer spec: {
    if(stage1.valid) stage1.data := bytes[stage1.idx];
    if(stage2.valid) stage2.data := bytes[stage2.idx];
  }
}
```

When verifying the output of one stage, we can drive the output of the previous stage from the abstract model, via the refinement map, thus decomposing the verification of each stage into a separate problem. Open this version in `vw` and select, for example, the property `out.data[0]//spec`. That is, we want to verify the final output against the refinement map. Select the Cone page, and notice that to define the data outputs of the `stage2`, SMV has chosen the layer `spec`, rather that the implementation definition. The number of state bits remaining (51) is still larger than in the previous case, however, because `spec` doesn't give any definition of the signals `valid` and `idx`, hence these are still driven by the implementation.

If you select "Prop—Verify `out.data[0]//spec`", you'll observe that we can still quickly verify this property, even thought the number of state variables is larger. Nonetheless, we would like to make the verification of the last stage independent of the previous stages, to be sure we can still verify it if the previous stages are made more complex. We can do this by explicitly "freeing" the signals `stage2.valid` and stage2.idx, that is, allowing these signals to range over any possible values of their types. This is the most abstract possible definition of a signal, and is provided by a built-in layer called `free`. To tell SMV explicitly to use the `free` layer for these signals, we add the following declaration:

```
using
  stage2.valid//free, stage2.idx//free
prove
  out.data//spec;
```

Open this new version, and select property `out.data[0]//spec`. Note the the number of state bits (in the Cone page) is now 39, as in our original problem. In fact, if you select "Prop—Verify `out.data[0]//spec`" you will probably get a very fast answer, since SMV will notice that the verification problem you are trying to solve is isomorphic to that of the one-stage implementation we started with. This information was saved in a file for future use when that property was verified.

To verify `stage2`, in the same way, we need to make similar `using...prove` declaration, as follows:

```
using stage1.valid//free, stage1.idx//free prove stage2.data//spec;
```

Note that we don't need a corresponding declaration for stage1, since the input signals `inp.valid` and `inp.idx` have been left undefined, and are thus free in any event. With this addition, chose "Prop—Verify all", and observe that all the properties are verified very quickly, since they are all isomorphic.

4.2.3 Refinement maps as types

You may have observed that it is getting a bit tedious to refinement maps for each stage of the implementation, when they are actually all the same. SMV provides a way to avoid this by specifying abstract definitions of a signal as part of its data type. We can also give a type a parameter, so that we can specify in the type declaration which abstract object an implementation object corresponds to. A parameterized type in SMV is otherwise known as a *module*. Let's declare a type with a refinement map as follows:

```
module byte_intf(bytes){

  bytes : array INDEX of BYTE;
```

```
    valid : boolean;
    idx : INDEX;
    data : BYTE;

    layer spec:
      if(valid) data := bytes[idx];
}
```

This defines an interface type that transfers an array `bytes` of bytes according to a specific protocol. This protocol is defined by layer `spec`. Now, lets rewrite our example using this type:

```
module main(){

  /* the abstract model */

  bytes : array INDEX of BYTE;
  next(bytes) := bytes;

  /* the input and output signals */

  inp, out : byte_intf(bytes);

  /* the implementation */

  stage1, stage2 : byte_intf(bytes);

  init(stage1.valid) := 0;
  next(stage1) := inp;
  init(stage2.valid) := 0;
  next(stage2) := stage1;
  init(out.valid) := 0;
  next(out) := stage2;

  /* abstraction choices */

  using stage2.valid//free, stage2.idx//free prove out.data//spec;
  using stage1.valid//free, stage1.idx//free prove stage2.data//spec;
}
```

Notice that there's no need to write the intermediate refinement maps. They are part of the data type.

4.2.4 The effect of decomposition

To see the effect of using refinement maps let's make two versions of our simple example, one with and one without intermediate refinement maps. We can easily do this by changing the types of the intermediate stages. To make it interesting, we'll use 32 delay stages. Here is the version with intermediate refinement maps:

```
/* the implementation */

stages : array 1..31 of byte_intf(bytes);


init(stages[1].valid) := 0;
next(stages[1]) := inp;

for(i = 2; i <= 31; i = i + 1){
  init(stages[i].valid) := 0;
  next(stages[i]) := stages[i-1];
}

init(out.valid) := 0;
next(out) := stages[31];

/* abstraction choices */


for(i = 2; i <= 31; i = i + 1)
  using stages[i-1].valid//free, stages[i-1].idx//free
  prove stages[i].data//spec;

using stages[31].valid//free, stages[31].idx//free prove out.data//spec;
```

Here is the version without intermediate refinement maps:

```
/* the implementation */

stages : array 1..31 of
  struct{
    valid : boolean;
    idx : INDEX;
    data : BYTE;
  }

init(stages[1].valid) := 0;
next(stages[1]) := inp;

for(i = 2; i <= 31; i = i + 1){
  init(stages[i].valid) := 0;
  next(stages[i]) := stages[i-1];
}
```

```
init(out.valid) := 0;
next(out) := stages[31];
```

Note, we don't want to free any of the intermediate signals in this version. Now, open the first version, and select "Props—Verify all". It should verify all 256 properties in something like 15 seconds (depending on your machine). Now, open the second version (without refinement maps). There are only 8 properties to verify in this case (one for each output bit), bit SMV cannot verify these properties, as you may observe by select "Prop—Verify all". When you get bored of watching SMV do nothing, select "Prop—Kill Verification" (note, this may not work under Windows), and click the Cone tab. Observe that the cone contains 256 state variables, which is usually to large for SMV to handle (though occasionally SMV will solve a problem of this size, if the structure of the problem is appropriate for BDD's). Note that it is possible to construct even a fairly trivial example which cannot be verified directly by model checking, but can be verified by decomposition and model checking. Generally, when a direct model checking approach fails, it's best to look for a decomposition of the problem using refinement maps, rather than to try to determine why the BDD's exploded.

4.3 Decomposing large data structures

In our trivial example, we are sending an array of 32 bytes. Because we only need to consider one bit out of each byte at a time, we were able to verify the implementation without explicitly decomposing this data structure. However, cases often arise when it is necessary to consider only one element at a time of a large structure. For example, we might increase the size of our array to 1 million bytes. As we will see later, sometimes even small arrays must be decomposed in this way. One one of decomposing a large array in the abstract model is to write an array of refinement maps (we'll see a more elegant way later, in section 4.6). Each element of this array defines a given low-level signal *only* when it contains the value of the corresponding element in the abstract array. For example, let's rewrite our interface data type to use a decomposed refinement map of this kind:

```
  module byte_intf(bytes){

    bytes : array INDEX of BYTE;

    valid : boolean;
    idx : INDEX;
    data : BYTE;

    forall(i in INDEX)
      layer spec[i]:
        if(valid & idx = i) data := bytes[i];
}
```

Notice that layer `spec` is now an array, with one element for each element of the array `bytes`. The layer `spec[i]` specifies the value of `data` only when `idx` is equal to `i`, and otherwise leaves `data` undefined. The advantage of this refinement map is that `spec[i]` refers to only one element of the array `bytes`. Thus, the other elements will not appear in the cone when verifying it, and we have reduced the number of state variables that the model checker must handle.

Let's go back to our 3-stage delay example, and use this new definition of `byte_intf`. Because we have changed the layer declarations, we also have to change the corresponding `using...prove` declarations. Replace these with the following:

```
forall(i in INDEX){
  using stage2.valid//free, stage2.idx//free prove out.data//spec[i];
  using stage1.valid//free, stage1.idx//free prove stage2.data//spec[i];
}
```

Now, when you try to open this file, you'll get an error message, something like this:

```
The implementation layer inherits two definitions of inp.data[5]
...in layer spec[31], "map7.smv", line 15
...in layer spec[30], "map7.smv", line 15
Perhaps there is a missing "refines" declaration?
```

This is because we have given many abstract definitions for `inp.data` without providing an implementation. By default, if there is only one abstract definition, SMV takes this as the implementation. However, if there are many abstract definitions, it is possible that these definitions are contradictory, and hence there is no possible implementation. There are several possible ways to make SMV stop complaining about this. One is to provide an actual implementation. For example, we could simply implement `inp.data` by a nondeterministic choice among all possible data values. This would mean, of course, that we could not then prove consistency with the maps `inp.data//spec[i]`. On the other hand, we don't really want to prove these, since they are actually assumptions about the inputs to our design, and not properties to be proved. One way to tell SMV this is to declare `inp` explicitly as an input to the design. SMV does not attempt to verify refinement maps driving global inputs. It just takes them as assumptions. If our main module is later used as a submodule in a later design, we'll have to verify these maps in the context of the larger design. Meanwhile, let's change the header of our main module to look like the following:

```
module main(bytes,inp,out){
  bytes : array INDEX of BYTE;
  input  inp : byte_intf(bytes);
  output out : byte_intf(bytes);
```

Notice we've also make `bytes` a parameter to the module. If we later use this module in a larger design, we can then specify what abstract data array we want the module instance to transmit. Now, open this file, and select, for example, property `out.data[0]//spec[0]`. You'll notice that there are now only 8 state variables in the cone, since 31 of data bits have been eliminated. Also, notice that SMV chose the layer `spec[0]` to define `stage2.data[0]`, out of the 32 possible abstract definitions. This is a heuristic choice, which was made on the basis of the fact that we are verifying an abstraction in layer `spec[0]`. If you'd like to see the reasoning SMV went through to arrive at this choice, select the signal `stage2.data[0]` and pull down "Abstraction—Explain Layer".

If you now select "Prop—Verify `out.data[0]//spec[0]`", you can observe that the verification is in fact faster than in the previous case. However, you'll also notice that the number of properties to prove is now very large. In fact, it is 32 times greater than before, since every property has now been decomposed into 32 cases! Select "Prop—Verify All", and you will find that the total verification time for this long list of properties is about 15 seconds, actually longer than before. Surely it is unnecessary to verify all of the 32 cases for each refinement map, since each is in effect symmetric to all the others. In fact, if we simply tell SMV where the symmetry is, we can convince it to prove only one case out of 32.

4.4 Exploiting Symmetry

Change the type declaration for `INDEX` from

```
typedef INDEX 0..31;
```

to

```
scalarset INDEX 0..31;
```

This is exactly the same as an ordinary type declaration, except it tells SMV that the given scalar type is *symmetric*, in the sense that exchanging the roles of any two values of the type has no effect on the semantics of the program. In order to ensure that this symmetry exists, there are a number of rules placed on the use of variables of a scalarset type. For example, we can't use constants of a scalarset type, and the only operation allowed on scalarset quantities is equality comparison. In addition, we can't mix scalarset values with values of any other type. We can, however, declare an array whose index type is a scalarset. This makes it legal for us to make the type `INDEX` into a scalarset. Now, when SMV encounters an array of properties whose index is of scalarset type, it chooses only one case to prove, since if it can prove one case, then by symmetry it can prove all of them.

Let's see the effect of this on our example. Open the new file (with `INDEX` changed to a scalarset), and look in the Properties page. You'll see that there are now only properties from layer `spec[0]`. Pull down "Prop—Verify All", and you'll find the total verification time reduced to about a half second (a savings of a factor 32!).

We can go a step further than this, and make the type `BIT` a scalarset as well. This is because all of the bits within a byte are symmetric to each other. So change

```
typedef BIT 0..7;
```

to

```
scalarset BIT 0..7;
```

and open the new file. Now, in the Properties pane, there are only three properties, one for each stage! Thus, using symmetry, we have reduced the number of properties, by a factor of $32 \times 8 = 256$.

4.5 Decomposing large structures in the implementation

Thus far, we've seen how we can decompose a large structure in the abstract model (such as the byte array in our example), and verify properties relating only to one small component of the structure. Now, we'll consider the case where we have a large structure in the implementation, and wish to consider only one component at a time. Let's keep the specification from our previous example, but design an implementation that has a large buffer that can store data bytes in transit. To make the problem more interesting, we'll put flow control in the protocol, so that our implementation can stop the flow of incoming data when its buffer is full. To implement flow control, we'll use two signals, one to indicate the sender is ready (`srdy`) and one to indicate the receiver is ready (`rrdy`). A byte is transferred when both of these signals are true. Here's the definition of this protocol as an interface data type:

```
module byte_intf(bytes){

  bytes : array INDEX of BYTE;

  srdy,rrdy : boolean;
  idx : INDEX;
  data : BYTE;

  valid : boolean;
  valid := srdy & rrdy;

  forall(i in INDEX)
    layer spec[i]:
    if(valid & idx = i) data := bytes[i];
}
```

Note that the refinement map only specifies the value of the data when both `srdy` and `rrdy` are true. Our system specification is exactly the same as before:

```
module main(bytes,inp,out){
  bytes : array INDEX of BYTE;
  input  inp : byte_intf(bytes);
  output out : byte_intf(bytes);

  /* the abstract model */

  next(bytes) := bytes;
```

For the implementation, we'll define an array of 8 cells. Since all of the cells are symmetric, we'll define a scalarset type to index the cells:

```
scalarset CELL 0..7;
```

Each cell holds an index and a data byte. Each cell also needs a bit to say when the data in the cell are valid:

```
cells : array CELL of struct{
  valid : boolean;
  idx : INDEX;
  data : BYTE;
}
```

We also need pointers to tell us which cell is to receive the incoming byte and which cell is to send the outgoing byte:

```
recv_cell, send_cell : CELL;
```

The implementation is ready to receive a byte when the cell pointed to by `recv_cell` is empty (*i.e.*, not `valid`). On the other hand, it is ready to send a byte when the cell pointed to by `send_cell` is full (*i.e.*, `valid`):

```
inp.rrdy := ~cells[recv_cell].valid;
out.srdy := cells[send_cell].valid;
```

Here is the code that implements the reading and writing of cells:

```
forall(i in CELL)init(cells[i].valid) := 0;

default{
  if(inp.valid){
    next(cells[recv_cell].valid) := 1;
    next(cells[recv_cell].idx) := inp.idx;
    next(cells[recv_cell].data) := inp.data;
  }
} in {
  if(out.valid){
    next(cells[send_cell].valid) := 0;
  }
}

out.idx := cells[send_cell].idx;
out.data := cells[send_cell].data;
```

For the moment, we will leave the pointers `recv_<cell` and `send_cell` undefined, and thus completely nondeterministic. This will allow us to cover all possible policies for choosing cells. Later, we can refine these signals to use a particular policy (*e.g.*, round-robin) without invalidating our previous work.

Finally, having defined our implementation, we will define a refinement map for the structure `cells` so that we do not have to consider the entire array at once. In fact, this refinement map almost defines itself, given the way the data structure `cells` is encoded. We want to say that if a cell `i` is valid, then its data is equal to the element of `bytes` pointed to by its index `idx`. Here is the refinement map:

```
forall(i in INDEX)
  layer spec[i]:
    forall(j in CELL)
      if(cells[j].valid & cells[j].idx = i) cells[j].data := bytes[i];
```

Note that once again, we have decomposed the map into separate indices. If cell `j`'s index is `i`, then cell `j` contains byte `i` from the abstract array.

Now that we have defined each cell's contents in terms of the abstract model, we can verify each cell separately. We can then assume that all the cells are correct when we verify the implementation output. Open this file, and notice that in the properties pane, there are just two properties: `cells[0].data[0]//spec[0]` and `out.data[0]//spec[0]`. All the other properties are equivalent to one of these by symmetry. Try "Prop—Verify All" to check that in fact our refinement is correct. Now select `cells[0].data[0]//spec[0]` in the Properties pane, and the click on the Cone tab. There are 15 state variables in total for this property. Notice that once again SMV has chosen layer `spec[0]` to drive `inp.data[0]`, since this is the layer we are verifying. Because of the decomposition we have used, data bits from only one cell and one element of the `bytes` array appear in the cone. In fact, most of the state bits come from the `valid` bits of the cells. These are included in the cone because the bit `inp.rrdy` depends on them. However, it is reasonable to hypothesize that the correctness of cell 0 does not actually depend on the `valid` bits of the other cells. We should be able to free them and still verify the property. To do this, add the following declaration to the program:

```
forall(i in INDEX) forall(j in CELL) forall(k in BIT)
  using cells//free, cells[j]
  prove cells[j].data[k]//spec[i];
```

This declaration probably requires some explanation. First, even though we are only interested in proving one property, `cells[0].data[0]//spec[0]`, we give a `prove` declaration for `cells[j].data[k]//spec[i]`, for all `i,j,k`. This is because we are not allowed to use constants of a scalarset type in the program. Second, in order to free the signals in all the cells *except* cell `j`, we specify `cells//free`, indicating that all components of `cells` should use the `free` layer, and then specify `cells[j]` to override this choice for the specific case of cell `j`. In a `using` declaration, a signal name without a layer indicates the implementation definition of that signal.

Open this version and select the property `cells[0].data[0]//spec[0]`. The number of state variables should now be 8 rather than 15, since the `valid` bits for the other cells are now free variables. Select "Prop—Verify `cells[0].data[0]//spec[0]`" and observe that our hypothesis is confirmed – the correctness of cell 0 is preserved, even when we free the state of the other cells. Also note that verification time is reduced.

Note, that by freeing some signals, we have decreased the number of state variables in the cone, we have also increased the number of "combinational" variables. These are variables that act as free or constrained inputs to the model. We can go a step further and substitute the "undefined" value for these bits. This is very much like an "X" value in a logic simulator. For example:

```
0 & undefined  = 0
1 & undefined  = undefined
0 | undefined  = undefined
1 | undefined  = 1
```

Using the undefined value has the advantage that no combinational variables will introduced, since these signals are given the constant value "undefined". The difficulty is that, as in a logic simulator, these undefined values can propagate widely, giving a pessimistic result – we may find that a counterexample is produced to the property using undefined values, even though the property is actually true. However, we can never "prove" a false property by introducing undefined values.

We can set signals to the undefined value using a predefined layer called `undefined`. For example, replace `cells//free` in the `using ... prove` declaration above with

```
cells//undefined
```

This will cause the signals that were previously freed to be given the undefined value instead. Open the new file and select the property `cells[0].data[0]//spec[0]`. Notice in the Cone pane that the other `valid` bits are no longer combinational variables. Thus we have eliminated 7 combinational variables from the cone. On the other hand, you can observe by selecting "Prop—Verify `cells[0].data[0]//spec[0]`" that the property is still provable under this weaker assumption about the environment.

Finally, let's go back to the other property we need to prove in this example, which is that the outputs are correct with respect to the specification (`out.data[0]//spec[0]`). Select this property in the Properties pane, and observe that there are still 49 state variables in the cone. This is because, although our refinement map drives the data value for each cell from the abstract model, the control bits `idx` and `valid` for each cell are still driven from the implementation. This is not a problem for us, since BDD's come to our rescue in this case. You can confirm this by selecting "Prop—Verify `out.data[0]//spec[0]`". This verification should take less than 2 seconds. Nonetheless, if this were not the case, we could reduce the number of state bits by freeing the cells' control bits. That is, our refinement map provides that the data in a cell are correct, for any values of the control bits `valid` and `idx`. So let's add the following declaration to the program:

```
forall(i in INDEX) forall(j in CELL) forall(k in BIT)
  using cells[j].idx//free, cells[j].valid//free
  prove out.data[k]//spec[i];
```

Open the new version and select the property `out.data[0]//spec[0]`. Notice that the number of state bits is now reduced to 1, a single bit of the abstract array. The verification time is also reduced, as you can observe by selecting "Prop—Verify `out.data[0]//spec[0]`".

4.6 Case analysis

Suppose that we have a condition `p`, and we would like to show that `p` holds true at all times. For any particular variable `x`, we could break the problem into cases. For each possible value of v of x, we could show that condition p is true at those times when x = v. Since at all times x must have one of these values, we can infer that p must be true at all times.

SMV has a special construct to support this kind of case analysis. It is especially useful for compositional verification, since for each case we can use a different abstraction of the system, including different components in the verification. This allows us to break large verification problems into smaller ones.

The above described case analysis is expressed in SMV in the following way:

```
forall (v in TYPE)
  subcase q[v] of p for x = v;
```

Now suppose that p is some temporal assertion G cond, where cond is any boolean condition. The above declaration effectively defines a collection of properties q[v], as if we had written

```
forall (x in TYPE)
  q[v] assert G (x=v -> cond);
```

That is, each q[v] asserts that p holds at those times when x = v. Clearly, if q[v] holds for all values of v, then p holds. Thus, SMV is relieved of the obligation of proving p, and instead separately proves all the cases of q[v]. Note that if TYPE is a scalarset type, we may in fact have to prove only one case, since all the other cases are symmetric.

4.6.1 A very simple example

Now, let's look at a trivial example of this. Let's return to our very simple example of transmitting a sequence of bytes. Here is the specification again:

```
scalarset BIT 0..7;
scalarset INDEX 0..31;
typedef BYTE array BIT of boolean;

module main(){

  /* the abstract model */

  bytes : array INDEX of BYTE;
  next(bytes) := bytes;

  /* the input and output signals */

  inp, out : struct{
    valid : boolean;
    idx : INDEX;
    data : BYTE;
  }
```

```
  /* the refinement maps */

  layer spec: {
    if(inp.valid) inp.data := bytes[inp.idx];
    if(out.valid) out.data := bytes[out.idx];
  }
```

And let's use our original very trivial implementation:

```
  init(out.valid) := 0;
  next(out) := inp;
}
```

That is, the output is just the input delayed by one time unit.

Note that our specification (layer `spec`) says that at all times the output value must be equal to the element of array `bytes` indicated by the index value `out.idx`. We would like to break this specification into cases and consider only one index value at a time. To do this, we add the following declaration:

```
  forall (i in INDEX)
    subcase spec_case[i] of out.data//spec for out.idx = i;
```

In this case, the property we are splitting into cases is `out.data//spec`, the assignment to `out.data` in layer `spec`. The resulting cases are `out.data//spec_case[i]`. Note, however, that in the subcase declaration, we only give a layer name for the new cases, since the signal name is redundant. This declaration is exactly as if we had written

```
  forall (i in INDEX)
    layer spec_case[i]:
      if (out.idx = i)
        out.data := bytes[out.idx];
```

except that SMV recognizes that if we prove `out.data//spec_case[i]` for all `i`, we don't have to prove `out.data//spec`. Run this example, and look in the properties pane. You'll see that `out.data//spec` does not appear, but instead we have `out.data//spec_case[0]`. Note that only the case `i = 0` appears, since INDEX is a scalarset type, and SMV knows that all the other cases are symmetric to this one. Now look in the cone pane. You'll notice that all of the elements of the array `bytes` are in the cone. This is because the definition of `inp.data` in layer `spec` references all of them. However, all of them except element 0 are in the `undefined` layer. This is a heuristic used by SMV: if a property references some specific value or values of a given scalarset type, then only the corresponding elements of arrays over that type are used. The rest are given the undefined value. You might try clicking on element `bytes[1]` and choosing `Abstraction|Explain Layer` to get an explanation of why this signal was left undefined. You can, of course, override this heuristic by explicitly specifying a layer for the other elements. In this case, however, the heuristic works, since property `out.data//spec_case[0]` verifies correctly.

4.6.2 Using case analysis over data paths

Now we'll look at a slightly more complex example, to show how case ananlysis can be used to reduce a verification problem to a smaller one, by considering only one path that a given data item might take from input to output. This technique is quite useful in reasoning about data path circuitry.

We'll use essentially the same specification as before, but in this case our implementation will be the array of cells that we used previously when discussing refinement maps (section 4.5). We have an array of cells, and each incoming byte is stored in an arbitrarily chosen cell. Recall that the specification in this case has to take into account the handshake signals. That is, the data are only valid when both `sdry` and `rrdy` are true:

```
/* the abstract model */

bytes : array INDEX of BYTE;
next(bytes) := bytes;

/* the input and output signals */

inp, out : struct{
  srdy,rrdy : boolean;
  idx : INDEX;
  data : BYTE;
}

/* the refinement maps */

layer spec: {
  if(inp.srdy & inp.rrdy) inp.data := bytes[inp.idx];
  if(out.srdy & out.rrdy) out.data := bytes[out.idx];
}
```

For reference, here is the implementation again:

```
/* the implementation */

cells : array CELL of struct{
  valid : boolean;
  idx : INDEX;
  data : BYTE;
}

recv_cell, send_cell : CELL;

inp.rrdy := ~cells[recv_cell].valid;
out.srdy := cells[send_cell].valid;
```

```
  forall(i in CELL)init(cells[i].valid) := 0;

  default{
    if(inp.srdy & inp.rrdy){
      next(cells[recv_cell].valid) := 1;
      next(cells[recv_cell].idx) := inp.idx;
      next(cells[recv_cell].data) := inp.data;
    }
  } in {
    if(out.srdy & out.rrdy){
      next(cells[send_cell].valid) := 0;
    }
  }
o
  out.idx := cells[send_cell].idx;
  out.data := cells[send_cell].data;
```

Recall that in the previous example, we wrote refinement maps for the data in the individual cells, in order to break the verification problem into two pieces: one to show that cells get correct data, and the other to show that data in cells are correctly transfered to the output. Now, we will use case analysis to get a simpler decomposition, with only one property to prove.

Our case analysis in this example will be a little finer. That is because we have two arrays we would like to decompose. One is the array of bytes to transfer, and the other is the array of cells. We would like to consider separately each case where `byte[i]` gets transfered through `cell[j]`. In this way, we can consider only one byte and one cell at a time. This is done with the following declaration:

```
  forall (i in INDEX) forall (j in CELL)
    subcase spec_case[i][j] of out.data//spec
      for out.idx = i & send_cell = j;
```

Notice that our case analysis now has two parameters. Each case is of the form `out.idx = i & send_cell = j` where `i` is an `INDEX` and `k` is a `CELL`. We can, in fact, have as many parameters in the case analysis as we like, provided we write the condition in the above form. SMV recognizes by the form of the expression that the cases are exhaustive.

Now run this example, and observe that once again, we have a single property to prove: `out.data//spec_case[0][0]`. The other cases are symmetric. If you look in the cone, you'll see that, while all elements of `bytes` and `cells` are referenced, all except element 0 of these arrays is left undefined, according to SMV's default heuristic. This makes the verification problem small enough that we can handle it directly, without resorting to an intermediate refinement map. You can confirm this by verifying `out.data//spec_case[0][0]`.

This technique of breaking into cases as a function of the specific path taken by a data item through a system is the most important reduction in using SMV to verify data path circuitry. Notice that symmetry is crucial to this reduction, since without it we would have a potential explosion in the numer of different paths.

4.7 Data type reductions

Now suppose that we would like to verify the correct transmission of a very large array of bytes, or even an array of unknown size. SMV provides a way to do this by reducing a type with a large or unknown number of values to an abstract type, with a small fixed number of values. This type has one additional abstract value to represent all the remaining values in the original type.

For example, when verifying the correct transmission of byte `i`, we might reduce the index type to just two values – `i` and a value representing all numbers not equal to `i`, (which SMV denotes `NaN`). This is an abstraction, since `NaN`, when compared for equality against itself, produces an undetermined value. In fact, here is a truth table of the equality operator for the reduced type:

=	i	NaN
i	1	0
NaN	0	{0,1}

The program with the reduced index data type is an abstraction of the original program, such that any property that is true of the abstract program is true of the original (though the converse is not true).

4.7.1 A very simple example

Let's return to our very simple example of transmitting a sequence of bytes (section 4.6.1). For reference, here is the specification again:

```
scalarset BIT 0..7;
scalarset INDEX 0..31;
typedef BYTE array BIT of boolean;

module main(){

  /* the abstract model */

  bytes : array INDEX of BYTE;
  next(bytes) := bytes;

  /* the input and output signals */

  inp, out : struct{
    valid : boolean;
    idx : INDEX;
    data : BYTE;
```

```
  }

  /* the refinement maps */

  layer spec: {
    if(inp.valid) inp.data := bytes[inp.idx];
    if(out.valid) out.data := bytes[out.idx];
  }
```

And let's use our original very trivial implementation:

```
  init(out.valid) := 0;
  next(out) := inp;
}
```

That is, the output is just the input delayed by one time unit.

As before, let's break the specification up into cases, one for each index value:

```
  forall (i in INDEX)
    subcase spec_case[i] of out.data//spec for out.idx = i;
```

If you run this example, and look in the cone pane, you'll see that there are five state variables in the cone for both `inp.idx` and `out.idx`. This is expected, since five bits are needed to encode 32 values. However, notice that for case `i`, if the index value at the output is not equal to `i`, we don't care what the output is. Our property `spec_case[i]` only specifies the output at those times when `out.idx = i`. We can therefore group all of the index values not equal to `i` into a class, represented by a single abstract value (`NaN`), and expect that the specification might still be true. To do this, add the following declaration:

```
  forall (i in INDEX)
    using INDEX->{i} prove out.data//spec_case[i];
```

This tells SMV to reduce the data type `INDEX` to an astract type consisting of just the value `i` and `NaN` (note, we don't specify `NaN` explicitly). Now, open the new version, and observe the cone. You'll notice the state variables `inp.idx` and `out.idx` now require ony one boolean variable each to encode them, since their type has been reduced to two values. Now try verifying the property `out.data//spec_case[0]`. The result is true, since the values we reduced to the abstract value don't actually matter for the particular case of the specification we are verifying.

Now, let's suppose that we don't know in advance what the size of the array of bytes will be. Using data type reductions, we can prove the correctness of our implemenation for any size array (including an infinite array). To do this, change the declaration

```
scalarset INDEX 0..31;
```

to the following:

```
scalarset INDEX undefined;
```

This tells SMV that `INDEX` is a symmetric type, but doesn't say exactly what the values in the type are. In such a case, SMV *must* have a data type reduction for `INDEX` to prove any properties, because it can only verify properties of finite state systems. Now run the new version. You'll notice that the result is exactly the same as in the previous case. One boolean variable is used to encode values of tye `INDEX`, and the specification is found to be true. In fact, in the previous case, SMV didn't in any way use the fact that type `INDEX` was declared to have the specific range `0..31`. Thus it's not surprising that when we remove this information the result is the same. By using finite state verification techniques, we have proved a property of a system with an infinite number of states (and an infinite number of systems with finite state spaces).

One might ask what would happen if, using a scalarset of undefined range, we ommitted the data type reduction. Wouldn't that give us an infinite state verification problem? Try removing the declaration

```
forall (i in INDEX)
  using INDEX->{i} prove out.data//spec_case[i];
```

from the problem and run the resulting file. You'll observe that nothing has changed from the previous case. Since SMV can't handle undefined scalarsets without a data type reduction, it guesses a reduction. It simply includes in the reduced type all the specific values of the given type that appear in the property. In this case, there is only one, the index `i`.

4.7.2 A slightly larger example

Now, let's reconsider the example from the previous section of an implementation with an array of cells (section 4.6.2). For reference, here are the specification and implementation:

```
/* the specification */

layer spec: {
  if(inp.srdy & inp.rrdy) inp.data := bytes[inp.idx];
  if(out.srdy & out.rrdy) out.data := bytes[out.idx];
}

/* the implementation */

cells : array CELL of struct{
  valid : boolean;
  idx : INDEX;
  data : BYTE;
}

recv_cell, send_cell : CELL;
```

```
  inp.rrdy := ~cells[recv_cell].valid;
  out.srdy := cells[send_cell].valid;

  forall(i in CELL)init(cells[i].valid) := 0;

  default{
    if(inp.srdy & inp.rrdy){
      next(cells[recv_cell].valid) := 1;
      next(cells[recv_cell].idx) := inp.idx;
      next(cells[recv_cell].data) := inp.data;
    }
  } in {
    if(out.srdy & out.rrdy){
      next(cells[send_cell].valid) := 0;
    }
  }

  out.idx := cells[send_cell].idx;
  out.data := cells[send_cell].data;
```

Let's make just one change to the source: we'll redefine the scalarset types INDEX and CELL to have undefined range:

```
scalarset INDEX undefined;
scalarset CELL undefined;
```

Since these types have undefined ranges, SMV will choose a data type reduction for use (though, of course, we could specify one if we wanted to). Now, run this modified version. You'll notice that in the properties pane, we have just one property to prove, as before: `out.data//spec_case[0][0]`. In the cone pane, obverve that the variables of type `INDEX` and `CELL` have only one boolean variable encoding them (representing the value 0 and `NaN`). In addition, only `cell[0]` and `byte[0]` appear. This is because SMV chose to reduce the types `INDEX` and `CELL` to contain only those values appearing in the property being verified, which in this case are just the value 0 for both types. Confirm that in fact the specification can be verified using this reduction.

Note that the proof reduction that we used for the case of a fixed number of cells and a fixed number of bytes, worked with no modification when we switched to an arbitrary number of bytes and cells!

These very simple examples provide a paradigm of the verification of complex hardware systems using SMV. One begins by writing refinement maps. They specifiy the inputs and outputs of the system in terms of a more abstract model, and possibly specify internal points as well, to break the verification problem into parts. The resulting properties are then broken into cases, generally as a function of the different paths that a data item may take from one refinement map to another. These cases are then reduced to a tractable number by symmetry considerations. Finally, for each case, a data type reduction is chosen which reduces the large (or even infinite) data types to a small fixed number of values. The resulting verification subproblems are then handled by symbolic model checking.

4.8 Proof by induction

Suppose now that we want to verify some property of a long sequence. For example, we may have a counter in our design that counts up to a very large number. Such counters can lead to inefficient verification in SMV because the state space is very deep, and as a result, SMV's breadth first search technique requires a large number of iterations to exhaustively search the state space. However, the usual mathematic proof technique when dealing with long sequences is *proof by induction*. For example, we might prove that a property holds for 0 (the base case), and further that if it holds fr some arbitrary value `i`, then it holds for `i + 1`. We then conclude by induction that the property holds for all `i`.

Data type reductions provide a mechanism for inductive reasoning in SMV. To do this, however, we need a data symmetric data type that allows adding and subtracting constants. In SMV, such data types are called *ordsets*. An ordset is just like a scalarset, except the restrictions on ordsets are slightly relaxed. If we delcare a type as follows:

```
ordset TYPE 0..1000;
```

then, in addition to the operations allowable on scalarset types, the following are also legal:

1. `x + 1` and `x - 1`,
2. `x = 0` and `x = 1000`

where `x` is of type `TYPE`. That is, we can increment and decrement values of ordset types, and also compare them with the extremal values of the type.

Induction is done in the following way: suppose we want to prove property `p[i]`, where i is the induction paremeter, ranging over type `TYPE`. We use a data type reduction that maps `TYPE` onto a set of four values: X,i-1,i,Y. Here the symbolic value `X` abstracts all the values less that `i-1`, and `Y` abstracts all the values greater than `i`. Incrementing a value in this reduced type is defined as follows:

```
X     + 1  =  {X,i-1}
(i-1) + 1  =  i
i     + 1  =  Y
Y     + 1  =  Y
```

That is, adding one to a value less than `i-1` will result in either `i-1` or a value less that `i-1`. Decrementing is similary defined. Any property provable in this abstract interpretation is provable in the original. In addition, we can show that all the cases from `i = 2` up to `i = 999` are isomorphic. Thus it is sufficient to prove oly the cases `i = 0, 1, 2, 1000`.

As an example, suppose we hae a counter that starts from zero and increments once per clock cycle, up to 1000. We'd like to show that for any value `i` from 0 to 1000, the counter eventually reaches `i`. Here's how we might set this up:

```
ordset TYPE 0..1000;

module main()
{
  x : TYPE;

  /* the counter */

  init(x) := 0;
  next(x) := x + 1;

  /* the property */

  forall(i in TYPE)
    p[i] : assert F (x = i);

  /* the proof */
  forall(i in TYPE)
    using p[i-1] prove p[i];

}
```

We prove each case `p[i]` using `p[i-1]`. That is, when proving the counter eventually reaches `i`, we assume that it eventually reaches `i-1`. (Note that technically, for the case `i = 0`, we are asking SMV to use `p[-1]`, but since this doesn't exist, it is ignored).

SMV can verify that this proof is noncircular. Further, using its induction rule, it automatically generates a data type reduction using the values `i` and `i-1`, and it generates the four cases we need to prove: `p[0], p[1], [2], p[1000]`. To confirm this, run the example, and look in the properties ane. You should see the four aforementioned properties. Now choose `Verify All` to verify that in fact the induction works, and that `p[i]` holds for all `i`.

4.8.1 Induction over infinite sequences

Now, suppose we have a counter that ranges from zero to infinity. We can still prove by induction that any value `i` is eventually reached. To do this, we declare `TYPE` to be an `ordset` without an upper bound:

```
ordset TYPE 0..;
```

With this change, run the example, and notice that in the properties pane there are now only three cases to prove: `p[0], p[1], [2]`. We don't have to prove the maximum value as a special case, because there is no maximum value. Now choose `Verify All` to verify that in fact the induction works, and that `p[i]` holds for all `i`. We've just proved a property of an infinite-state system by model checking.

4.8.2 A simple example

To see how we can use induction in practice, let's return to our example of transmitting an array of bytes. This time, however, we will assume that the bytes are in an infintie sequence. They are received at the input in the order `0, 1, 2, ...` and they must be transmitted to the output in that order.

To begin with, let's define our types:

```
scalarset BIT 0..7;
typedef BYTE array BIT of boolean;

ordset INDEX 0..;
```

Note that we defined `INDEX` as an ordset type, so we can prove properties by induction over indices.

We begin with the original refinement specification. As in section 4.2.3, we encapsulate it in a module, so we can reuse it for both input and output:

```
module byte_intf(bytes){

  bytes : array INDEX of BYTE;

  valid : boolean;
  idx : INDEX;
  data : BYTE;

  layer spec:
    if(valid) data := bytes[idx];
}
```

To specify ordering we simply introduce a counter `cnt` that counts the number of bytes received thus far. If there is valid data at the interface, we specify that the index of that data is equal to `cnt`. Thus, add the following declarations to module `byte_intf`:

```
  cnt : INDEX;

  init(cnt) := 0;
  if(valid) next(cnt)  := cnt + 1;

  ordered: assert G (valid -> idx = cnt);
```

Note, we can include temporal properties, like the above property `ordered` inside modules. Thus, for each instance of the interface definition, we'll get one instance of this property. As our first implementation, we'll just use the trivial implementation that delays the input by one clock cycle. Here's what the main module looks like:

```
module main(bytes,inp,out){
  bytes : array INDEX of BYTE;
  input  inp : byte_intf(bytes);
  output out : byte_intf(bytes);

  /* the abstract model */

  next(bytes) := bytes;

  /* the implementation */

  init(out.valid) := 0;
  next(out.valid) := inp.valid;
  next(out.data)  := inp.data;
  next(out.idx)   := inp.idx;
}
```

To prove the correctness of the data output (with respect to the refinement specification), we use the same proof as before – we split into cases based on the index of the output:

```
  forall(i in INDEX)
    subcase spec_case[i] of out.data//spec
      for out.idx = i;
```

Note that anything that can be done with a scalarset can also be done with an ordset.

So much for the data correctness – the interesting part is the correct ordering. For the proof of the ordering property, we're going to use induction over the value of the counter `cnt`. The intuition here is that, if the output index equals the counter when the counter is `i`, then at the next valid output the counter and index will both be one greater, and hence they will be equal for `cnt = i + 1`. This assumes, of course, that the input values are ordered correctly. To verify this, we must first break the output ordering property into cases based on the value of cnt:

```
  forall(i in INDEX)
    subcase ord_case[i] of out.ordered for out.cnt = i;
```

Then, we prove case `i` using case `i-1` and the input ordering property. We leave `inp.ordering` as an assumption:

```
  forall(i in INDEX){
    using ord_case[i-1], inp.ordered prove ord_case[i];
    assume inp.ordered;
  }
```

Now, run this example, and observe the properties pane. You'll notice that we now have three cases of the property `out.data//spec_case[i]` to prove: `i = 0, 1, 2`. In fact, all of these cases are isomorphic, but since `INDEX` is defined as an ordset rather than a scalarset, SMV's type checking rules don't guarantee this. Thus, SMV will effectively prove the same property three times. Fortunately, each case takes only a fraction of a second.

Now observe that we also have three cases of `ord_case[i]` to prove. Select, for example, property `ord_case[2]` from the properties pane and observe the cone. You'll notice that each value of type `INDEX` requires two boolean variables to encode it. This is because there are four values in the reduced type: `i-1`, `i` and two abstract values to represent the ranges `0..i-1` and `i+1..infinity`. Notice also that there are no data values in the cone, since the indices do not depend on the data. Thus, we have effectively separated the problem of correct ordering from correct delivery of data.

Now, try `Prop|Verify all`. All the cases should be verified in less than a second.

A note: for ordsets, a data type reduction may be specified, in lieu of SMV's default. The general form of the data type reduction for ordset types is:

```
TYPE -> { min..min+a,  i-b..i+c,  max-d..max};
```

where `min` is the minimum value of `TYPE`, `i` is the induction parameter, and `max` is the maximum value of `TYPE`. Thus, SMV allows us to use any finite number of values around the induction parameter `i` and the extremal values. In this case, the number of cases that need to proved will be larger, however.

4.8.3 A circular buffer

Now let's consider transmission of an infinite sequence of bytes again, but this time using our array of cells as a circular buffer (an implementation of a FIFO queue).

To begin with, we need to add handshaking to our interface definition, so add the following to module `byte_intf`:

```
srdy, rrdy : boolean;
valid := srdy & rrdy;
```

The signal `srdy` indicates that the sender is ready, while `rrdy` indicates the the receiver is ready. The data are valid, by definition, when both are ready.

Now, as in section 4.5, we'll use an array of 32 cells, to hold our data items. So define the type CELL as:

```
ordset CELL 0..31;
```

The reason for making it an `ordset` type will become apparent later. Now, replace the previous "trivial" implementation with the following:

```
cells : array CELL of struct{
  valid : boolean;
  idx : INDEX;
  data : BYTE;
}
```

```
recv_cell, send_cell : CELL;

inp.rrdy := ~cells[recv_cell].valid;
out.srdy := cells[send_cell].valid;

forall(i in CELL)init(cells[i].valid) := 0;

default{
  if(inp.valid){
    next(cells[recv_cell].valid) := 1;
    next(cells[recv_cell].idx) := inp.idx;
    next(cells[recv_cell].data) := inp.data;
  }
} in {
  if(out.valid){
    next(cells[send_cell].valid) := 0;
  }
}

out.idx := cells[send_cell].idx;
out.data := cells[send_cell].data;
```

Note, `recv_cell` is the cell we are receiving a byte into, and `send_cell` is the cell we are sending a byte from. We block our input (setting `inp.rrdy` to zero) when the cell wwe are receiving into is full, and block our output (setting `out.srdy` to zero) when the cell we are sending from is empty. When we receive into a cell, we set its `valid` bit to true, and when we send from the cell, we clear its `valid` bit.

Up to this point, we haven't said what policy is used to choose `recv_cell` and `send_cell`. To make our buffer ordered, we can use a round-robin policy. This means that each time we receive a byte, we increment `recv_cell` by one, and each time we send a byte, we increment `send_cell` by one. When either of these reaches its maximum value, it returns to zero. To accomplish this, add to following code to the implementation:

```
init(recv_cell) := 0;
if(inp.srdy & inp.rrdy)
  next(recv_cell) := (recv_cell = 31) ? 0 : recv_cell +1;

init(send_cell) := 0;
if(out.srdy & out.rrdy)
  next(send_cell) := (send_cell = 31) ? 0 : send_cell +1;
```

Note that, since `CELL` is an ordset type, rather than a `scalarset`, it's legal to compare it against the maximum value (31) and set it back to the minumum value (0). If `CELL` were a scalarset, it wouldn't be legal to introduce any constants of the type.

Now that we have our implementation, lets prove both the correctness of the data output and correctness of the ordering. The case splitting statement for data correctness is the same as when we did this example in section 4.6.2, where we weren't concerned with data ordering:

```
forall(i in INDEX) forall(j in CELL)
  subcase spec_case[i][j] of out.data//spec
    for out.idx = i & send_cell = j;
```

That is, we consider separately the case of each byte index `i`, and the cell `j` that it is stored in. That way, we only need to consider one cell in the aray at a time. Notice that adding ordering does not change the proof of data correctness in any way.

Now for the ordering question. Again, we are going to use induction. The ordering property says that when the output data are valid, the output index must be equal to are count of the number of previous values. We'll do the proof by induction over the value of the counter. That is, we'll assume that the index was correct when the count was `i-1`, and then prove that the index is correct when the count is `i`. This means that, as before, we have to split cases based on `cnt`. However, in this case we also have to split cases on the cell in which the current output value stored. Thus, we use the following case splitting declaration:

```
forall(i in INDEX) forall(j in CELL)
  subcase ord_case[i][j] of out.ordered
    for out.cnt = i & send_cell = j;
```

Now, the question is, what data type reduction to use for type `CELL`. We know we need to use cell `j`, since that is the one holding the data item we are interested in. However, in addition, we need to use the previous cell. The intuition behind this is as follows. We are assuming that the output index is correct for byte `i-1`. If byte `i` is stored in cell `j`, then byte `i-1` is stored in cell `j-1` (which one exception). This means we need to inclde cell `j-1`. Then, if cell `j-1` contains index i-1, and the inputs are ordered, it follows that cell `j` will contain index `i`, which is what we are trying to prove. Thus, we might use the data type reduction:

```
CELL -> {j-1..j}
```

However, note that the exception to the above reasoning is the case `j = 0`. In this case, the "previous" cell is cell 31. Since there's no way (yet) to write a special data type reduction for this case, we'll just include the value `31` in our data type reduction for all the cases. Thus, we write:

```
forall(i in INDEX) forall(j in CELL)
  using CELL -> {j-1..j,31} prove ord_case[i][j];
```

Now comes the actual inductive step: we use the case `cnt = i-1` to prove the case `cnt = i`:

```
forall(i in INDEX} forall(j in CELL)
  using ord_case[i-1], inp.ordered prove ord_case[i][j];

assume inp.ordered;
```

Notice that we use the entire array `ord_case[i-1]` (for all cells) in this verification. This isn't really necessary, since only the "previous cell" (`j-1` or `31`) is needed in any give case, but its harmless. Note that we aren't doing induction over the cell number. In fact, we can't do this, since the cells are used in a circular manner. This would result in a cycle in the proof.

Now, run this example, and note the properties that appear in the properties pane. You'll observe that the property `ord_case[i][j]` has to be proved for all the combinations of `i = 0,1,2` and `j = 0,1,2,30,31`. The reason we have extra cases to prove for the cell index `j`, is that we included the maximum value `31` in the data type reduction. SMV reasons that the case `j=30` might not be isomorphic to the case ĵ=31, since we might compare `j` in some way with the value 31. However, as you can observe by selecting "Prop—Verify All", all of these cases can be verified quickly. This is because the number of state variables is small after data type reductions.

Thus, we've proved that a circular buffer implementation correctly transmits an infinite sequence of bytes using a given handshake protocol.

4.8.4 Abstract variables

Notice that the case of the circular buffer, we don't really have to send the byte indices, since they can be inferred from the ordering property of the interface. The data output doesn't depend on them. Thus, in the actual implementation, we would leave out the `idx` output of the buffer, considering it only an "auxiliary" variable used in the verification. This use of "auxiliary state" added to the implementation gives us a convenient way to specify interfaces as a function of abstract models. The auxiliary information tells us which object in the abstract model is currently appearing at the interface. This in turn allows us to specify what data should be appear at the interface as a function of the abstract model. In the next section, we'll see a slightly different way to do this.

We can tell SMV that a given variable is part of the proof only, and not part of the actual implementation, by declaring it as `abstract`. For example, in the `byte_intf` module, we would declare the `idx` component as:

```
abstract idx : INDEX;
```

SMV will verify for us that no actual implementation logic depends on this variable. The abstract variables can thus be excised from the implementation while retaining all the properties we've proved.

4.9 Instruction processors

Up to now, when discussing refinement verification, we've considered only the transfer of data from one place to another, without actually operating on the data. Now we'll have a look at how to verify instruction set processors, that is, machines that input a sequence of operations to be performed on some data structure, such as a register file or a memory. In this case, our abstract model is usually an "instruction set architecture" (ISA). This is represented by a simple sequential machine the processes instructions on at a time, in the order they are received. The implementation is usually a more complex machine that works on more than one instruction at a time. This can be done, for example, by pipelining, or out-of-order execution techniques.

The key to verification of such designs in SMV is to break the problem up into individual instructions. Usually, we break an instruction up into two parts, which correspond to two lemmas int the proof. The first lemma is that all the operands fed to the function unit(s) are correct, according to the abstract model. The second is that all results produced by the functional unit(s) are correct (again, with respect to the abstract model). Needless to say, we use lemma 1 to prove lemma 2, and *vice versa*. The reason for breaking the probelm into two lemmas is that the operand fetching operation and the functional unit operation are somewhat different in nature, so it's convenient to separate the two issuues, so we can apply a different proof approach to each (much as we separated the issues of data correctness and ordering in the circular buffer).

Now, in order to specify that the operands and results are correct with respct to the abstract model, we usually have to add some auxiliary information to the implementation (see the previous section). In this case, we add to each instruction moving through the implementation a few extra fields to store the correct operands and results for that instruction, as computed by the abstract model.

4.9.1 A very simple example

As a very simple example, let's define an instruction set architecture with just one instruction, performed on values in a register file. Each instruction has two source operands and a destination operand. Thus, an opcode consists of three fields – `srca`, `srcb` and `dst`. For simplicity, we'll make the operation addition. Here's what the ISA model might look like:

```
scalarset REG undefined;
typedef WORD array 0..31 of boolean;

module main()
{
  r : array REG of WORD;
  srca, srcb, dst : REG;

  opra, oprb, res : WORD;

  opra := r[srca];
  oprb := r[srcb];
  res  := opra + oprb;
  next(r[dst]) := res;
}
```

We've declared a type `REG` to represent a register index, a type `WORD` to represent a data word (in this case a 32 bit word). Notice that `REG` is a undefiend scalarset. That is, we don't say, for the moment, how many registers there are.

Notice, also, that we've given names to the operand values `opra` and `oprb`, and to the operation result `res`. It wasn't necessary to do this. That is, we could have written:

```
next(res[dst]) := r[srca] + r[srcb];
```

This would have been more concise. However, it's convenient to give the intermediate quantities names, since we will use these later in writing refinement relations. Now let's implement this abstract model with a simple 3 stage pipeline, where the fisrt stage fecthes the operands, the second stage does the addition, and the third stage stores the result into the register file. The implementation has a reguster bypass path that forwards the results directly from later stages of pipe to the operand fetch stage.

```
/* the implementation */

/* implementation register file */

ir : array REG of WORD;

/* pipe registers */

stage1 : struct {
  valid : boolean;
  dst : REG;
  opra, oprb : WORD;
}

stage2 : struct{
  valid : boolean;
  dst : REG;
  res : WORD;
}

/* read stage : fetch operands with bypass */

next(stage1.opra) :=
  case{
  stage1.valid & srca = stage1.dst : alu_output;
  stage2.valid & srca = stage2.dst : stage2.res;
  default : ir[srca];
};
next(stage1.oprb) :=
  case{
  stage1.valid & srcb = stage1.dst : alu_output;
  stage2.valid & srcb = stage2.dst : stage2.res;
  default : ir[srcb];
```

```
};

next(stage1.dst) := dst;
init(stage1.valid) := 0;
next(stage1.valid) := 1;


/* alu stage: add operands */

alu_output : WORD;
alu_output := stage1.opra + stage1.oprb;
next(stage2.res) := alu_output;

next(stage2.dst) := stage1.dst;
init(stage2.valid) := 0;
next(stage2.valid) := stage1.valid;


/* writeback stage: store result in r */

if(stage2.valid)
  next(ir[stage2.dst]) := stage2.res;
```

Note that each stage has a `valid` bit, which says whether there is an instruction in it. Initially, these bits are zero.

Now, we would like to write two refinement maps – one which defines the correct operand values in `stage1` and the other which defines the correct result at the adder output. To do this, we add some auxiliary state information to each stage tat remembers the correct operand and result values for the given stage, as computed by the abstract model. Let's add the following component to `stage1` :

```
stage1.aux : struct{
   opra, oprb, res : WORD;
}
```

Now, let's add some code to record the correct operand and result values for the first stage:

```
next(stage1.aux.opra) := opra;
next(stage1.aux.oprb) := oprb;
next(stage1.aux.res)  := res;
```

That is, we simply record the abstract model's values for `opra`, `oprb` and `res`. Note, this is why we gave them explicit names in the abstract model. This is all the auxiliary information we'll need to state our refinement relations. However, for e deeper pipeline, we could just pass the auxiliary information down the pipe along with the instructions, as follows:

```
next(stage2.aux) := stage1.aux;
...
```

Now, we can state the two refinement maps in terms of the auxiliary state information. For the operands, we specify that, if stage 1 has a valid instruction, then its operands are equal to the correct operand values:

```
layer lemma1: {
  if(stage1.valid) stage1.opra := stage1.aux.opra;
  if(stage1.valid) stage1.oprb := stage1.aux.oprb;
}
```

For the ALU results, we specify that, if stage1 has a valid instruction, then the ALU output is equal to the correct result value:

```
layer lemma2:
  if(stage1.valid) alu_output := stage1.aux.res;
```

We would like to show, of course, the correct operands imply correct results, and conversely, correct results imply correct operands. However, since we have an arbitrary number of registers to deal with, we'll need to break `lemma1` into cases as a function of which register is being read. The only problem we have in doing this is that we don't know which registers were the source operands for the instruction in stage one, because our implementation does not store this information. This problem is easily solved, however, since we can store the information in our auxiliary state. So let's add two components to the auxiliary state:

```
next(stage1.aux.srca) := srca;
next(stage1.aux.srcb) := srcb;
```

Of course, we have to remember to declare these components in our auxiliary structure (their type is `REG`). Now, we split the operand refinement maps into cases based on which are the actual source registers of the instruction in stage 1. For the `srca` operand, we have:

```
forall(i in REG)
  subcase lemma1[i] of stage1.opra//lemma1 for stage1.aux.srca = i;
```

Similarly, for `srcb`, we have:

```
forall(i in REG)
  subcase lemma1[i] of stage1.oprb//lemma1 for stage1.aux.srcb = i;
```

This way, we only have to consider one register at a time, so we can reduce an arbitrary number of registers to just one, for each case. Note, we don't need to do this for `lemma2`, the result refinement maps, since it doesn't depend on the register file. It depends only on the operands.

Now we're ready to prove the various cases of our lemmas. For `lemma1`, we say:

```
forall(i in REG)
  using res//free, alu_output//lemma2 prove stage1//lemma1[i];
```

That is, we assume that the ALU ouput is correct, and show that (future) operands we obtain are correct. Notice that there are several paths that an ALU result might take to get back to the operand registers in stage 1. It might follow the bypass path, or it might get stored in register `i`. Either way, it should agree with what the abstract model gets. Notice also that the correct storage and forwarding of a result deosn't depend on what the result actually is. For this reason, we free the abstract model's result `res`. This eliminates the abstract model's ALU from the cone.

To prove the result lemma (`lemma2`), we assume that operands entering the ALU are correct:

```
using opra//free, oprb//free, stage1//lemma1
  prove alu_output//lemma2;
```

Note, in this case, we don't care what the correct operands actually are – we only care that the abstract model and the implementation agree on them (`lemma1`). Thus, we free `opra` and `oprb`, and eliminate the abstract model register file from the cone. This is important, since this register file is of unbounded size, and in this case we have no single register index to which we can reduce the type `REG`.

Now, run this example. You'll notice that there are 32 instances to prove for each of

```
stage1.opra[i]//lemma1[0]
stage1.oprb[i]//lemma1[0]
alu_output[i]//lemma2
```

where `i` is a bit index within a word. This is because SMV proves the refinement maps for each of the 32 bits of the data path separately. Later we'll see how to reduce this rather large number of properties. For the moment, however, select property

```
stage1.opra[0]//lemma1[0]
```

and try to verify it. You should get a counterexample. In this counterexemple, the initial value of `r[0][0]` (a bit in the abstract register file) is zero, while the initial value of `ir[0]0]` (the corresponding bit in the implementation register file) is one. The problem here is that the abstract model is underspecified. Because we have specified the initial state of the register file, it is nondeterministic. As a result of this, the abstract model and implementation have diverged.

When there is a nondeterministic choice in an abstract model, we sometimes have to provide a "witness function" for this choice. That is, as a function of the implementation behavior, we plug in a suitable value in the abstract model. In this case, since the initial value in the specification is complete undefined, we are free to plug in any value we like. So let's write the following:

```
init(r) := ir;
```

That is, we just set the initial value of the abstract model register file to be the same as the initial value of the implementation register file. You might be wondering why we have to do this. That is, why can't SMV figure out what the correct initial value of the register file is. The answer is that it could, for any given property. However, it might use different intial values to prove different properties. As a result, even though we would have "verified" all the properties, there would be no single choice that makes all the properties true. Thus, for reasons of soundness, SMV requires you to fix the choice once and for all, and then verifies all the properties for the particular choice you make.

In any event, let's open the new version, with the witness function, and try again to verify `stage1.opra[0]//lemma1[0]`. You should find that the property is true. Look in the Cone pane, and observe that it contains only 11 boolean state variables. This is bacause we are considering only registers `r[0]` and `ir[0]`, and only bit 0 of the data path. We obtain only bit 0 of the data path since neither the abstract model ALU nor the implementation ALU is in the cone. The former was eliminated by freeing `res`, while the latter was eliminated by using `lemma2` to drive the ALU output in the implementation.

Now select property `alu_output[0]//lemma2`. The cone is rather large in this case (66 state variables) because bit 0 depends in this case on all the other bits of the data path through the ALU. (This is because bit 0 is the most significant bit,and depeds on all the others through the carry chain). However, notice the register files are not in the cone in this case, because we have freed `opra` and `oprb`, and we have driven the implementation operand registers using `lemma1`.

Go ahead and verify property `alu_output[0]//lemma2`. You should find that it checks fairy quickly in spite of the large number of state variables. This is because the ALU operation is addition, and SMV succeeds in finding an ordering of the BDD variables that maes the addition function compact. In fact, select `Prop|Verify All` to verify all the remaining properties. On my machine, this takes a little under eight seconds.

On the other hand, if we had had a multiplier in the ALU the story would have been different. This is because there is no BDD variable ordering that makes this function compact. The verification of multipliers is beyond the scope of this tutorial. There is, however, a way of separating the problem of airthmentic verification from the processor verification problem. In this way, we can verify the processor design independent of the ALU function. Then we can plug in any ALU function we like.

4.9.2 Uninterpreted functions

Suppose that instead of specifying the exact function of the ALU in our abstract model, we simply use a symbol `f` to denote this function. Suppose further that we use the same function symbol in our implementation, and we are able to prove a refinement relation between the two. It would then follow that the refinement holds for any concrete function we might want to plug in place of `f`.

To represent such an uninterpreted function symbol in SMV, we simply introduce an array to represent its lookup table. For example, if we have a function `f` that takes two `WORD` arguments and produces a `WORD` result, we might write:

```
forall (a in WORD) forall (b in WORD)
```

```
f[a][b] : WORD;
```

or equivalently

```
f : array WORD of array WORD of WORD;
```

The only thing we need to know about function `f` is that it doesn't change over time. To declare this in SMV, we can simply write:

```
next(f) := f;
```

Now, to evaluate function `f` over two arguments `a` and `b`, we just look up the result in the table. For example:

```
res := f[opra][oprb];
```

The trick here is that, without a data type reduction for type `WORD`, the lookup table for `f` will be of astronomical size. However, by case splitting, we can consider only the case when the arguments are some fixed values, and the result of the function is some fixed value. By doing this, we then have to consider only one element of the table for `f` at a time. This is a good thing, but it requires that `WORD` be a symmetric type (a scalarset or an ordset), so that we can reduced the very large numer of cases (here $2^{32} \times 2^{32} \times 2^{32}$) to a tractable number (for example, 6).

So now let's rewrite our example using an uninterpreted function symbol `f` for the ALU function. First, let's redefine type `WORD` to be a scalarset:

```
scalarset WORD undefined;
```

We don't have to say what the range of the type is. Instead, we'll verify our design for any word size. Now, in the main module, let's define an uninterpreted function `f`:

```
f : array WORD of array WORD of WORD;
next(f) := f;
```

Finally, we'll replace the ALU functions in both abstract model and implementation with function `f`. In the abstract model, change

```
res := opra + oprab;
```

to

```
res := f[opra][oprb];
```

In the implementation, change

```
alu_output := stage1.opra + stage1.oprb;
```

to

```
alu_output := f[stage1.opra][stage1.oprb];
```

Now that we've modeled our problem with an uninterpreted function, we need to do a little further case splitting, so that we only have to think about a few values of `WORD` at a time.

For the operand lemma, we'll split cases on the cirrect operand value. That is, we'll prove that the operands we obtain are correct when the correct value is some fixe number `j`:

```
forall(i in REG) forall(j in WORD)
  subcase lemma1[i][j] of stage1.opra//lemma1
    for stage1.aux.srca = i & stage1.aux.opra = j;
```

(and similarly for `oprb`). For the results lemma, we want to consider only one entry in the lookup table for `f` at a time. We'll split our result refinement map (`lemma2`) into cases based on the values of the two operands, and the value of function `f` for those two particular values. Thus for example, we might verify that the `alu_output` signal is correct only in the particular case when `opra = 0` and `oprb = 1` and when `f[0][1] = 2`. Here is a suitable case splitting declaration:

```
forall (a in WORD) forall(b in WORD) forall(c in WORD)
  subcase lemma2[a][b][c] of alu_output//lemma2
    for stage1.aux.opra = a
      & stage1.aux.oprb = b
      & f[a][b] = c;
```

Our `using...prove` declarations are exactly the same as before, except that they have added parameters for the additional case splits:

```
forall(i in REG) forall(j in WORD)
  using res//free, alu_output//lemma2 prove stage1//lemma1[i][j];

forall (a in WORD) forall(b in WORD) forall(c in WORD)
  using opra//free, oprb//free, stage1//lemma1
    prove alu_output//lemma2[a][b][c];
```

Now, open the new version. For `alu_output//lemma2[a][b][c]`, there are six cases to prove:

```
alu_output//lemma2[0][0][0]
alu_output//lemma2[1][0][0]
alu_output//lemma2[2][0][0]
alu_output//lemma2[0][1][0]
alu_output//lemma2[1][1][0]
alu_output//lemma2[2][1][0]
```

That is, SMV generates enough cases so that we see all the possible equality relationships between `a`, `b` and `c`, of which there are 3 factorial. For lemma 1, we now have just one case each for `opra` and `oprb`, since there is only one parameter of type `WORD`.

Select property `alu_output//lemma2[0][0][0]` and look at the cone. You'll notice that only one element of the lookup table for `f` appears in the cone: `f[0][0]`. This is because 0 is the only specific valued in the reduced type `WORD`. (SMV automatically chose a reduction for us, including just those values that specifically appear in the property we're proving). Verify this property. It's not surprising that the verification is rather fast, since there are only 5 state variables.

Now select property `alu_output//lemma2[2][1][0]`. Notice that in this case we have nine cases of `f[a][b]` in the cone (all the combinations of `a,b = 0,1,2`). This is because SMV isn't smart enough to figure out that the only element that actually matters is `f[2][1]`. We could, if we wanted to, include a declaration to make the remaining values undefined:

```
forall (a in WORD) forall(b in WORD) forall(c in WORD)
  using f//undefined, f[a][b] prove alu_output//lemma1[a][b][c];
```

This would reduce the number of state variables quite a bit, but it isn't really necessary. Even with the extraneous variables, the verification is quite fast, as you may observe.

Finally, select `Prop|Verify All` to verify the remaining cases. We have now verified our trivial pipeline design for an arbitrary number of registers, an arbitrary word size, and an arbitrary ALU function.

4.9.3 What about outputs?

Up to now, we've proved a certain relationship between the abstract model and the implementation, but we haven't really proved that the circuit `observably` implements its specification. This is because the pipeline has no outputs. We could easily, however, give the processor and output instruction (perhaps one that outputs the sum of two registers). In this case the output of our pipeline would likely appear with some delay, relative to the specification. This means we would need to write a refiement map for the pipeline output that delays the abstract model output by some fixed amount. In this case, since the delay is finitely bounded, writing such a map is straightforward (we'll leave it as an "exercise for the reader"). If there isn't a known fixed bound on the output delay, we might, for example, borrow a technique from a previous section. That is, we could attach in index to each instruction, so that we know which instruction's value is appearing at any given time at the output. We could then use induction, as before, to show that the output values appear in the correct order.

In any event, in the next section, we'll see an example of a more interesting implementation, with an output.

4.10 An out-of-order instruction processor

The above may have seemed like a great deal of effort to verify such a simple design. However, we will find that the proof becomes only incrementally more complex when we move to a much more complex implementation – an instruction processor using Tomasulo's algorithm.

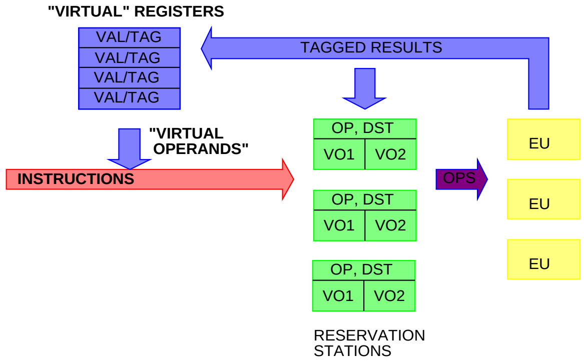


Figure 2: Flow of instructions in Tomasulo's algorithm

4.10.1 Tomasulo's algorithm

Tomasulo's algorithm allows execution of instructions in data-flow order, rather than sequential order. This can increase the throughput of the unit, by avoiding pipeline stalls. Each pending instruction is held in a "reservation station" until the values of its operands become available, then issued "out-of-order".

The flow of instructions is pictured in figure 2. Each instruction, as it arrives, fetches its operands from a special register file. Each register in this file holds either an actual value, or a "tag" indicating the reservation station that will produce the register value when it completes. The instruction and its operands (either values or tags) are stored in a reservation station (RS). The RS watches the results returning from the execution pipelines, and when a result's tag matches one of its operands, it records the value in place of the tag. When the station has the values of both of its operands, it may issue its instruction to an execution pipeline. When the tagged result returns from the pipeline, the RS is cleared, and the result value, if needed, is stored in the destination register. However, if a subsequent instruction has modified the register tag, the result is discarded. This is because its value in a sequential execution would be overwritten.

In addition to an ALU instruction, we include instructions that read register values to an external output and write values from an external input. There is also a "stall" output, indicating that an instruction cannot be received either because there is no available RSto store it, or because the value of the register to be read to an output is not yet available.

4.10.2 The abstract model

As before, our abstract model is a simple machine that executes instructions in order as they arrive. Additionally, in this case, it has the ability to stall. The choice of whether to stall or not is nondeterministic.

As before, we make the register index values and data values undefined scalarsets:

```
scalarset WORD undefined;
scalarset REG  undefined;

module main()
```

```
{
  ...
}
```

We define an uninterpreted function `f` for the ALU:

```
f : array WORD of array WORD of WORD;
next(f) := f;
```

Here is the abstract model:

```
opin : {ALU,RD,WR,NOP};       /* opcode input */
srca,srcb,dst : REG;          /* source and dest indices input */
din,dout : WORD;              /* data input and output */
r : array REG of WORD;        /* the register file */
opra,oprb,res : WORD;         /* operands and result */
stallout : boolean;           /* stall output (nondeterministic) */

/* the abstract model */

layer arch:
  if(~stallout)
    switch(opin){
      ALU : {
        opra := r[srca];
        oprb := r[srcb];
        res := f[opra][oprb];
        next(r[dst]) := res;
      }
      RD : {
        dout := r[srca];
      }
      WR : {
        next(r[dst]) := din;
      }
    }
```

Note that we've put our specification inside a layer called `arch`, so that we can refine the data output signal `dout` in the implementation. Also note that since we haven't specified a value for `stallout` it remains nondeterministic. In case of an `ALU` operation, our behavior is as before: apply the ALU operation `f` to the two source operands, and store the result in the register file. In case of a `RD` operation, we read the `srca` operand from the register file and assign it to `dout`, the data output. In case of a `WR` opration, we store the value of the data input, `din`, into the destination register. (Finally, in case of a `NOP` operation, we do nothing).

4.10.3 Implementation

In the implemenation, we have two main data structures: the register file and the array of reservation stations. We define these as follows:

```
ir : array REG of
  struct{
    resvd : boolean;
    tag : TAG;
    val : WORD;
  }

st : array TAG of
  struct{
    valid : boolean;
    opra, oprb : st_opr;
    dst : REG;
    issued : boolean;
  }
```

Each register has a bit `resvd`, which is true when it is holding a tag (we say it is "reserved") and false when it is holding a value. Each reservation station has a bit `valid` to indicate is is holding a valid instruction, a bit `issued` to indicate its instruction has been issued to an execution unit, and two operand fields, `opra` and `oprb`. The operand type is defined as follows:

```
typedef st_opr struct{
  valid : boolean;
  tag : TAG;
  val : WORD;
}
```

Each operand has a bit `valid`. When `valid` is true, it holds a value, otherwise it holds a tag. The type `TAG` is an index into the reservation station array, and is declared as follows:

```
scalarset TAG undefined;
```

The result bus is called `pout` and is declared as follows:

```
pout : struct{
  valid : boolean;
  tag : TAG;
  val : WORD;
}
```

We also need arbitrary choices for the reservation station to store a new instruction into, and the reservation to issue to an execution unit at any given time:

```
st_choice : TAG;
issue_choice : TAG;
```

Now, we begin with the implementation behavior. Initially, all the reservation stations are empty, and all the registers are unreserved:

```
forall(i in TAG)
  init(st[i].valid) := 0;
forall(i in REG)
  init(ir[i].resvd) := 0;
```

There are three basic operations that occur on the register file and reservation stations:

- incoming instructions stored in a RS,
- instruction issue to execution unit and
- instruction completion (writeback to register file).

These three operations appear in the following `default...in` structure:

```
default
  {...instruction completion logic...}
in default
  {...incoming instruction logic...}
in
  {...instruction issue logic...}
```

This is done to specify the relative priority of the three operations in case they write to the same register at the same time. However, in principle they shouldn't interfere with eachother, except in one case where we need a register bypass.

Now, here is the implementation of instruction completion:

```
  if(pout.valid){
    forall(i in REG)
      if(ir[i].resvd & ir[i].tag = pout.tag){
        next(ir[i].resvd) := 0;
        next(ir[i].val) := pout.val;
      }

    forall(i in TAG){
      if(~st[i].opra.valid & st[i].opra.tag = pout.tag){
        next(st[i].opra.valid) := 1;
        next(st[i].opra.val) := pout.val;
      }
      if(~st[i].oprb.valid & st[i].oprb.tag = pout.tag){
        next(st[i].oprb.valid) := 1;
        next(st[i].oprb.val) := pout.val;
```

```
      }
      if(st[i].issued && pout.tag = i)
        next(st[i].valid) := 0;
    }
  }
```

The signal `pout.tag` tells us which instruction the returning result is for. We match it against the tags in the register file – if any reserved register has this tag, we store the returning value in it, and mark it unreserved. Similarly, we match the tag against any reservation stations that are valid – if one of the operands has this tag, we store the result in it, and mark it valid. Finally, the reservation station whose index is `pout.tag` has now completed, so we mark it invalid.

Now, here's the code for incoming instructions. Note, we have to consider a special case where an operand of the incoming instruction is returning on the result bus at precisely this moment. In this case, we bypass the register file and send the result dirctly to the reservation station:

```
  if(~stallout)
    switch(opin){
      ALU : {

        /* store the instruction in an RS */

        next(ir[dst].resvd) := 1;
        next(ir[dst].tag) := st_choice;
        next(st[st_choice].valid) := 1;
        next(st[st_choice].issued) := 0;

        /* fetch the a operand (with bypass) */

        if(pout.valid & ir[srca].resvd & pout.tag = ir[srca].tag){
          next(st[st_choice].opra.valid) := 1;
          next(st[st_choice].opra.tag) := ir[srca].tag;
          next(st[st_choice].opra.val) := pout.val;
        } else {
          next(st[st_choice].opra.valid) := ~ir[srca].resvd;
          next(st[st_choice].opra.tag) := ir[srca].tag;
          next(st[st_choice].opra.val) := ir[srca].val;
        }

        /* fetch the a operand (with bypass) */

        if(pout.valid & ir[srcb].resvd & pout.tag = ir[srcb].tag){
          next(st[st_choice].oprb.valid) := 1;
          next(st[st_choice].oprb.tag) := ir[srcb].tag;
```

```
          next(st[st_choice].oprb.val) := pout.val;
        } else {
          next(st[st_choice].oprb.valid) := ~ir[srcb].resvd;
          next(st[st_choice].oprb.tag) := ir[srcb].tag;
          next(st[st_choice].oprb.val) := ir[srcb].val;
        }
      }

      RD : dout := ir[srca].val;

      WR : {
        next(ir[dst].val) := din;
        next(ir[dst].resvd) := 0;
      }
    }
```

Note that when when fetching an operand from a reserved register, if the tag matches the returning result on pout, we directly move the pout data into the operand field of the reservation station. Otherwise, we move the contents of the register (whether a tag or a value).

Finally, here is the code for instruction issue:

```
  if(st[issue_choice].valid
     & st[issue_choice].opra.valid
     & st[issue_choice].oprb.valid
     & ~st[issue_choice].issued
     & exe_rdy)
    {
      exe_valid := 1;
      next(st[issue_choice].issued) := 1;
    }
  else exe_valid := 0;

  exe_tag  := issue_choice;
  exe_opra :=  st[issue_choice].opra.val;
  exe_oprb :=  st[issue_choice].oprb.val;
}
```

If the RS chosen for issue has a valid instruction, and if both its operands are valid, and if it is not already issued, and if an execution unit is available, we send an instruction to the execution units, and mark the RS as issued.

There are two reasons why the above operations might result in a stall: the reservation station chosen for an incoming instruction might be full, or the register chosen for reading out might be reserved. Thus, here is the definition of stallout:

```
ASSIGN stallout :=
```

```
 opin = ALU & st[st_choice].valid
 | opin = RD & ir[srca].resvd;
```

Now, for the execution units, we will use a fairly abstract model. Each execution unit computes its result, and stores it for an arbitrary length of time, before signaling that it is ready. Here is our data structure for an execution unit:

```
eu : array EU of struct{
  valid, ready : boolean;
  res : WORD;
  tag : TAG;
}
```

We also need two arbitrary choices for execution units to receive the issued instruction, and to send completed results to the result bus:

```
issue_eu, complete_eu : EU;
```

Initially, let's use only one execution unit, to simplify the proof. Later, we'll see how to handle multiple execution units.

```
scalarset EU 0..0;
```

Here is the rest of the code for the execution unit(s):

```
exe_rdy,exe_valid : boolean;
exe_tag : TAG;
exe_opra, exe_oprb : WORD;

forall(i in EU)
  init(eu[i].valid) := 0;

default{
  if(~eu[issue_eu].valid){
    next(eu[issue_eu].valid) := exe_valid;
    next(eu[issue_eu].res) := f[exe_opra][exe_oprb];
    next(eu[issue_eu].tag) := exe_tag;
  }
} in {
  pout.valid := eu[complete_eu].valid & eu[complete_eu].ready;
  pout.val := eu[complete_eu].res;
  pout.tag := eu[complete_eu].tag;
  if(pout.valid)
    next(eu[complete_eu].valid) := 0;
}
```

Initially, all the execution units are invalid. If the unit chosen for issue is not valid, we mark it valid, and store in it the result of applying the function `f` to the two operands. We also store the tag of the issuing instruction.

If the unit chosen for completion is valid and ready, we pass its result on to the result bus (`pout`) and mark it invalid. Note that ready is a completely nondeterministic bit here, modeling an unknown delay in the execution unit. Also note that in practice, we would define some policy for choosing a unit to issue to and a unit to complete (presumably we do not want to choose to issue to an already valid unit, for example). This would likely involve introducing a priority encoder or round-robin policy, which would break the symmetry of the `EU` type. Symmetry breaking is a topic for another section, however.

The last part of the implementation is the witness function for the initial state of the abstract model register file:

```
layer arch:
  forall(i in REG)
    init(r[i]) := ir[i].val;
```

4.10.4 Refinement maps

As before, now that we have an abstract model and an implementation, we will write refinement maps that relate the two, and then break these into cases that are small enough problems to verify with model checking. Surprisingly, the refinement maps that we will use are almost identical to the ones we used for the simple pipeline. That is, we have one lemma that states that operands obtained by the reservation stations are correct, and one that states that results returning from the execution units are correct.

Also as before, to write these specifications, we will add some auxiliary state to the implementation, to remember what the correct values of the operands and results are. Each reservation station will have an auxiliary structure containing values for `opra`, `oprb` and `res`. In addition, we'll include the source register indices `srca` and `srcb` (recall that last time we used these values for case splitting):

```
aux : array TAG of struct {
  opra, oprb, res : WORD;
  srca, srcb : REG;
}
```

Now, when we store an instruction in a reservation station, we want to record the correct values from the abstract model into the auxiliary structure:

```
if(~stallout & opin = ALU){
  next(aux[st_choice].opra) := opra;
  next(aux[st_choice].oprb) := oprb;
  next(aux[st_choice].res)  := res;
  next(aux[st_choice].srca) := srca;
  next(aux[st_choice].srcb) := srcb;
}
```

Now that we've recorded the correct values, we can specify our refinement maps. For the operands (lemma1) we state that if a given RS holds a valid operand, its value must match the correct value. For the "a" operand, we have:

```
forall(k in TAG)
  layer lemma1 :
   if(st[k].valid & st[k].opra.valid)
     st[k].opra.val := aux[k].opra;
```

The "b" operand is similar. Now, for the result lemma (lemma2) we state that, if a result is returning on the result bus, tagged for a given reservation station, then its value is the correct result for that reservation station:

```
forall (i in TAG)
  layer lemma2[i] :
   if(pout.tag = i & pout.valid)
     pout.val := aux[i].res;
```

4.10.5 Case splitting

Now, let's split our lemmas into cases, so that we only have to think about one possible path for data to follow from one refinement map to the other. We begin with the operand lemma.

Consider a result returning on the result bus. That result is the result value of a given reservation station i. It then (possibly) gets stored in a register j. Finally it gets read as an operand for reservation station k. This suggests a case split which will reduce the size of the verification problem to just two reservation stations and one register. For each operand arriving at reservation station k, we split cases based on the reservation station i that it came from (this is the "tag" of the operand) and the register j that it passed through (this is the source operand index, srca or srcb, that we store in the auxiliary state for just this purpose). We also want to split cases on the correct data value, since WORD is an undefined scalarset type. Thus, here is the case splitting declaration for the "a" operand:

```
forall (i in TAG) forall (j in REG) forall (k in TAG) forall(c in WORD)
   subcase lemma1[i][j][c]
   of st[k].opra.val//lemma1
   for st[k].opra.tag = i & aux[k].srca = j & aux[k].opra = c;
```

That is, we consider only the case where the tag (*i.e.* the producing reservationstation) is i, and source register is j and the correct value is c. The "b" operand is similar.

For the result lemma (lemma2), we consider a pair of operands that start in some reservation station i and pass through execution unit j. Since i is a parameter of the lemma already, we are left with just j to split cases on (this is the value of the signal complete_eu). However, we now also have three data values to split cases on: the two operands a and b, and the result, c = f[a][b]. As before, this will reduce the data type WORD and the table f down to a tractable size. Thus, here is our case splitting declaration for lemma2:

```
forall(i in TAG) forall(j in EU)
forall(a in WORD) forall(b in WORD) forall(c in WORD)
   subcase lemma2[i][j][a][b][c]
   of pout.val//lemma2[i]
   for aux[i].opra = a & aux[i].oprb = b & f[a][b] = c
       &  complete_eu = j;
```

Finally, we have one last thing to prove, which is that the data output is correct according to the architrectural model (layer arch). This is quite similar to the operand lemma. That is, every data output value was produced as a result by some instruction and then stored in the source register for the RD instruction. Therefore, when proving that data output values are correct, we will split cases on the producing reservation station (this is obtained from the tag of the source register) and the source register index. In addition, as before, we consider only the case where the correct value is some arbitary constant c:

```
forall (i in TAG) forall (j in REG) forall (k in TAG) forall(c in WORD)
   subcase arch[i][j][c]
   of dout//arch
   for srca = j & ir[j].tag = i & r[j] = c;
```

4.10.6 The proof

Now we procede to define the abstractions used to prove the cases of the two lemmas. As before, when proving lemma1 we use lemma2 and *vice versa*. Also as before, we free the results in the abstract model when verifying operands, and free the operands when verifying the results.

Here is the proof for the operand lemma lemma1 and the data output (both of these assume lemma2):

```
forall (i in TAG) forall (j in REG) forall (k in TAG) forall(c in WORD)
  using res//free, pout//free, pout.val//lemma2[i]
  prove st[k]//lemma1[i][j][c], dout//arch[i][j][c];
```

Notice the we also freed the signals in the pout bus (other than the value itself, which is given by lemma2), so that none of the execution unit logic appears in the cone.

For the results lemma (lemma2), we take a similar tack: we use lemma2 for the operands, and otherwise free them in order to eliminate the operand fetch logic from the cone:

```
forall(i in TAG) forall(j in EU)
  forall(a in WORD) forall(b in WORD) forall(c in WORD)
    using opra//free, oprb//free, st[i]//lemma1, f//undefined, f[a][b]
      prove lemma2[i][j][a][b][c];
```

Notice we've set all the elements of the lookup table for f to undefined except for f[a][b] since this is the only element of the table that matters to our particular case.

Now, open the file. For st[k].opra.val//lemma1[i][j][c], the "a" operand correctness lemma, you'll notice we have two cases to prove:

```
st[0].opra.val//lemma1[0][0][0]
st[1].opra.val//lemma1[0][0][0]
```

This is because both `i` (the producer RS) and `k` (the consumer RS) are both of type `TAG`. Thus SMV must verify one case where `i = k` and one case where $\texttt{i} \neq \texttt{k}$. All the other cases are equivalent to one of these by permuting values of type `TAG`. Now, select property `st[1].opra.val//lemma1[0][0][0]` (this is the more interesting of the two cases, since it involves two reservation stations). Now, look in the Cone pane. You should observe that all of the state variables of type `TAG` (such as `st[1].opra.tag`) require two bits to encode them. This is because the type `TAG` has been reduced to three values: 0, 1, and an abstract value representing all the other tags. On the other hand, register indices (such as `srca`) have been reduced to just two values, and hence are represented by a single boolean value. These reductions were made by default, because we didn't specify data type reductions for the undefined scalarsets.

Notice also that we have freed signals in such a way as to cut off any connection to the exectution units in the abstract model and the implementation. Thus, for example, the function `f` does not appear in the cone. Finally, as a result of the data type reductions, we have only register zero and RS's zero and one in the Cone. Accesses to any other elements of these arrays will yield the undefined value. The result of all these reductions is that the cone contains only 25 state bits. Try verifying the property. Because of the smal number of state bits, it verifies on my machine in a little under one second.

Now let's consider the results lemma (`lemma2`). This appears as a collection of cases of the form:

```
pout.val//lemma2[i][j][a][b][c]
```

which states that results for RS `i` on the result bus `pout` are correct, in the case where execution unit `j` is returning a result, the "a" operand is `a`, the "b" operand is `b` and the `f[a][b]` (the correct result) is `c`. Since `a`, `b` and `c` are all of te same type, we have $3! = 6$ cases to prove:

```
pout.val//lemma2[0][0][0][0][0]
pout.val//lemma2[0][0][0][1][0]
pout.val//lemma2[0][0][1][0][0]
pout.val//lemma2[0][0][1][1][0]
pout.val//lemma2[0][0][2][0][0]
pout.val//lemma2[0][0][2][1][0]
```

This is enough to represent all the possible equality relatiopnships between `a`, `b`, and `c`. The most difficult case should be the last one, since it hase three different values of type `WORD`. In fact, if you select this property and look in the cone pane, you should observe that the values of type `WORD` are reprsented by two boolean variables (enough to encode the values 0, 1,and 2, plus an abstract value). In addition, because the index data types are reduced to only those values occurring in the property, we have only one reservation station in the cone. If we access any RS's other than zero, we'll get an undefined value. However, this should not affect the truth of our property, since it only tests returning results that derive from resrevation station zero. The other results will, of course, be incorrect in the reduced model, but our property ignores them. You can validate this argument by selecting “Prop—Verify pout.val//lemma2[0][0][2][1][0]”. The property verifies quite quickly, because there are only 18 state variables in the cone (it takes less than half a second on my machine).

Now choose “Prop—Verify All” to verify the remaining cases. It should take on the order of five seconds to do this. We have verified an out-of-order execution unit with an arbitrary number of registers and reservation stations, an arbitrary size data word and an arbitrary function. The basic strategy we used to do this was the same as for the simpler pipelined unit:

1. **Refinement maps and auxiliary state**. We broke the problem into two parts, by writing refinement maps that specify the correct values for the operands and results obtained in the implementation. To do this, the correct values are obtained from the abstract model, and stored in auxiliary state.

2. **Path splitting**. We broke the large data structures (the register file and RS array) down into just a few components by splitting cases on the path taken by a data item from one refinement map to another.

3. **Symmetry**. The large number of cases produced by the above two steps are reduced to a small finite number by considerations of symmetry.

4. **Data type reductions**. After case splitting, we can reduce the large (or infinite) types, such as data words, to small finite types by grouping all the irrelevant values into a single abstract value. A special case of this is the uninterpreted function abstraction, in which we use a large table to represent and arbitrary function, but then split cases in such a way that we use only one element of the table for each case, after the data type reduction.

As a result of this strategy, the problem has been reduced to 11 rather small finite-state lemmas.

4.10.7 Abstract counterexamples

Veriifcation runs that succeed are not generally very interesting. To convince yourself that the above proof strategy actually works, you might wat to try introducing a bug into the implementation to see what happens. For example, let’s remove the bypass logic from the operand fetch to see what happens. Replace the following code:

```
/* fetch the a operand (with bypass) */

if(pout.valid & ir[srca].resvd & pout.tag = ir[srca].tag){
  next(st[st_choice].opra.valid) := 1;
  next(st[st_choice].opra.tag) := ir[srca].tag;
  next(st[st_choice].opra.val) := pout.val;
} else {
  next(st[st_choice].opra.valid) := ~ir[srca].resvd;
```

```
    next(st[st_choice].opra.tag) := ir[srca].tag;
    next(st[st_choice].opra.val) := ir[srca].val;
  }
```

with this:

```
  /* fetch the a operand (without bypass) */

  next(st[st_choice].opra.valid) := ~ir[srca].resvd;
  next(st[st_choice].opra.tag) := ir[srca].tag;
  next(st[st_choice].opra.val) := ir[srca].val;
```

Now, open the modified version and select "Prop—Verify All". You should get a counterexample for property

```
st[1].opra.val//lemma1[0][0][0]
```

This is what happens inthe counterexample. At step 1, an instruction with destination register 0, and result 0 is loaded into RS 0. At step 2 this is issued to an exectution unit, and at step 3, the result returns on the result bus (`pout.val` is true). At the same time, a new instruction with "a" source register 0 is store in RS 1. However, because we have removed the bypass path, this instruction doesn't notice that its operand is currently returnin on the result bus. Thus, it gets a tag 0 instead of a value for its "a" operand. Now, in step 4, a new instruction is loaded into RS 0, again with destination 0, but this time with some value other than zero as its result. Notice the the value of `res` (the abstract model result) at step 4 is `NaN`. In the reduced model, this represents any value other than zero. Then in step 5, this result returns on the result bus, with tag 0, and thus gets fowarded to RS 1, which is waiting for tag 0. Unfortunately, RS 1 is expecting a value of zero (see `aux[0].opra`, since it is really waiting for the result of an earlier instruction with tag 0. Thus our property is violated at step 6: the expected operand was zero, but the actual obtained operand (`st[1].opra.val`) is non-zero (represented by `NaN`). Even though the counterexample is abstract (*i.e.*, it contains the abstract value `NaN`), it represents a class of real counterexamples (where, for example, the value 1 is obtained instead of 0).

In fact, the counterexample is abstract in another way. Notice that at step 5, a result returns on the the result bus `pout.valid` is true, even though the reservation station (`st[0]`) is not yet in the issued state. This is because the result bus is being driven by the refinement map `lemma2` rather than by the real execution unit. Our refinement map didn't specify that a result would not return from an execution unit before it was issued. Interestingly, our design for the reservation stations and register file is sufficiently robust that a result arriving early in this way does not cause us to obtain correct operands output incorrect values. Thus, we are able to verify the implementation with rather "loose" refinement maps. This is a case of a more general phenomenon: the more robust the individual components of a design are, the simpler are the refinement maps.

4.10.8 Multiple execution units

Note finally, that we have only verified our implementation of Tomasulo's algorithm for one execution. We could easily enough verify our design for some small finite number of units as well. However, with multiple execution units, we can't abstract away all the execution units except the one we're interest in. This is because one of these abstracted units might return an incorrect tag, which would reset the state of our reservation station prematurely. You can see observe this phenomen by changing the declaration of the type `EU` to the following:

```
scalarset EU 0..7;
```

Thus, we now have 8 execution units. If you open this modified version, and try verifying all the properties, you should obtain a counterexample for `lemma2`, in which reservation station 0 issues an instruction to execution unit zero, but then some other execution unit (which is abstracted by SMV's default heuristics) returns an undefined value as its tag, causing the state of reservation station 0 to be corrupted.

We can fix this problem by forcing SMV not to abstract the control information in the other execution units (though the data can still be left abstract, since we don't care abut it). To see this, change the proof declarion for `lemma2` to the following:

```
  forall(i in TAG) forall(j in EU) forall(k in EU)
  forall(a in WORD) forall(b in WORD) forall(c in WORD)
    using opra//free, oprb//free, st[i]//lemma1, f//undefined, f[a][b],
      eu[k].tag, eu[k].ready, eu[k].valid
    prove pout//lemma2[i][j][a][b][c];
```

The only change here is that we have said, for all execution units `k`, to include the implementation definitions of `tag`, `ready` and `valid`. The overrides the daufult behavior, which is to abstract these to `undefined`. Now, open this modified version, and try verifying all the properties. This should succeed, but take about a minute, instead of the five seconds required for the one-EU version. The reason is that we have greatly increased the number of state variables. If you select property `pout//lemma2[0][0][2][1][0]`, you'll notice that the control bits of all the execution units are present in the cone, and as a result, the number of state bits is increased to 32.

As we have observed, the problem with eight execution units is still within the realm that can be solved with BDD's. However, if we want to verify the design for an arbitrary number of execution units, we'll need to deal with the problem of `interference`, the subject of the next section.

4.10.9 Proving non-interference

Our problem in verifying Tomasulo's algorithm with an arbitrary number of execution units is that we are forced consider only one execution unit at a time, in order to obtain a finite-state verification problem. Thus, we consider the correctness only of the results of one particular execution unit. When we perform this verification, the other execution units are abstracted to an undefined value Thus, although we are not concerned with the correctness of the data values they produce, they may still upset the control state in the given reservation station by returning spurious tags.

To rule out this possibility, we add a non-interference lemma. This states that while a reservation station is expecting a result from a given execution unit, no other unit returns a result for that particular RS. In general, such a lemma is needed whenever the state of one system component might be corrupted by a spurious message from other components that have been abstracted away.

In order to state the condition that a given reservation station does not receive an unexpected result, we first have to add some auxiliary state information to tell us which execution unit the reservation station is actually expecting a result from. To do this, we add an additional field to the auxiliary state array:

```
forall (i in TAG)
  aux[i].eu : TAG;
```

Now, each time that a given reservation station issues an instruction to an execution unit, we record the index of that execution unit in the auxiliary state:

```
if(exe_valid)next(aux[issue_choice].eu) := issue_eu;
```

We can now state the non-interference lemma as follows:

```
lemma3 : assert G (pout.valid -> (complete_eu = aux[pout.tag].eu));
```

That is, `lemma3` states that, at all times, if a result is returning on the `pout` bus, with a given tag `pout.tag`, then the unit returning the result (`complete_eu`) must be the unit that the indicated reservation station is waiting for (`aux[pout.tag].eu`).

Now, lets see if we can prove `lemma3`. The first thing we'll have to do is to break the lemma into cases, so we only have to consider one reservation station and one execution unit. So let's add the following case splitting declaration:

```
forall(i in TAG) forall(j in EU)
  subcase lemma3[i][j] of lemma3 for pout.tag = i & complete_eu = j;
```

That is, we only consider the case where the returning result is for reservation station `i` and the execution unit returning the result is `j`. With the above additions, open the file, and select property `lemma3[0][0]`. If you look in the cone pane, you should see that SMV has automatically perfomed data type reductions, reducing TAG to just {0,NaN} and EU to `0,NaN`. As a result, there are only 9 state variables (notice also that no data variables appear in the cone, because `lemma3` is a control property, and does depend on any data variables. However, if you try to verify the property, you'll find that it's false. The counterexample shows a case where reservation station 0 is waiting for a result from execution unit 0, but instead, at state 3, a result returns from some other execution unit (that is `complete_eu = NaN`. In other words, in trying to prove non-interference, we've run into an interference problem. You might think that we are cought in an infintie regression here. However, in fact all is not lost. This is because when proving a particular case of `lemma3` at time t, SMV will allow us to assume the full lemma holds up to time $t-1$. In other words, we only have to prove that execution unit 0 is not the `em first` execution unit to interfere. If this is true for all execution units, we can then safely infer that no execution unit interferes. To tell SMV to assume the full lemma up to time $t-1$, add the following declaration:

```
forall(i in TAG) forall(j in EU)
  using (lemma3) prove lemma3[i][j];
```

The parentheses around `lemma3` tell SMV to make the weaker assumption that `lemma3` only holds up to $t-1$. If we leave them out, SMV will complain that the proof is circular. With this addition, open the file, and try to prove `lemma3[0][0]`. This time it should be true.

Now that we've proved that other executions can't interfere, let's return to the proof of `lemma2` (correctness of returning results). We want to prove that a result coming back on the result bus is correct, *assuming* that no previous interference has occured. To do this, add `(lemma3)` to the assumptions used to prove `lemma2`. You should get a declaration like the following:

```
forall(i in TAG) forall(j in EU)
forall(a in WORD) forall(b in WORD) forall(c in WORD)
  using opra//free, oprb//free, st[i]//lemma1,
    f//undefined, f[a][b], (lemma3)
  prove pout//lemma2[i][j][a][b][c];
```

Notice that we only assume the non-interference lemma up to time $t-1$ when proving `lemma2` up to time t. In fact, SMV won't allow us to use `lemma3` up to time t. This is because `lemma2` is a refinement map. Thus, we might well choose to use it use it when proving `lemma3`, which would result in a circularity. Fortunately, the weaker assumption is sufficient to prove the lemma. To confirm this, open the new version, and choose "Prop—Verify all".

With the addition of a non-interference lemma, we have now proved that our implementation of Tomasulo's algorithm works for any word size, any ALU function, any number of registers, any number of reservation stations, and any number of execution units.

4.11 Adding a reorder buffer

Now, let's modifiy the design to use a "reorder buffer". This means that instead of writing results to the register file when they are produced by an execution unit, we store them in a buffer, and write them back to the register file in program order. This is usually done so that the processor can be returned to a consistent state after an "exceptional" condition occurs, such as an arithmetic overflow. The simplest way to do this in the present implementation is to store the result in an extra field `res` of the reservation station, and then modify the allocation algorithm so that reservation stations are allocated and freed in round-robin order. The result of an instruction is written to the register file when its reservation station is freed.

To effect this change, add the following fields to the reservation stat structure `st`:

```
completed : boolean;
res : WORD;
```

Also add a variable `complete_st` to indicate which reservation station should be deallocated:

```
complete_st : TAG;
```

Now, change the instruction completion logic, so that, when a result appears on the the bus `pout`, instead of storing it in the register file, we store it in the `res` field of the reservation station and set the `completed` bit. If the reservation station indicated by `complete_st` has its `completed` bit set, we store its result from the `res` field into the register file. Thus, we replace the instruction completeion logic with the following:

```
default {
  /* result writeback logic */

  if(st[complete_st].valid & st[complete_st].completed){
    forall(i in REG)
      if(ir[i].resvd & ir[i].tag = complete_st){
        next(ir[i].resvd) := 0;
        next(ir[i].val) := st[complete_st].res;
      }
    next(st[complete_st].valid) := 0;
  }
} in default {
  /* instruction completion logic */

  if(pout.valid){
    forall(i in TAG){
      if(~st[i].opra.valid & st[i].opra.tag = pout.tag){
        next(st[i].opra.valid) := 1;
        next(st[i].opra.val) := pout.val;
      }
      if(~st[i].oprb.valid & st[i].oprb.tag = pout.tag){
        next(st[i].oprb.valid) := 1;
        next(st[i].oprb.val) := pout.val;
      }
      if(st[i].issued && pout.tag = i){
        next(st[i].completed) := 1;
        next(st[i].res) := pout.val;
      }
    }
  }

} in ...
```

Finally, we have to make sure that a result sitting in the `res` field of a completed instruction, but not yet written back to the register file, gets forwarded to any new instructions that might need it. Thus, for example, we change the operand fetch logic for the `opra` operand to the following:

```
/* fetch the a operand (with bypass) */

if(pout.valid & ir[srca].resvd & pout.tag = ir[srca].tag){
  next(st[st_choice].opra.valid) := 1;
  next(st[st_choice].opra.tag) := ir[srca].tag;
  next(st[st_choice].opra.val) := pout.val;
} else if(ir[srca].resvd & st[ir[srca].tag].completed){
  next(st[st_choice].opra.valid) := 1;
  next(st[st_choice].opra.tag) := ir[srca].tag;
  next(st[st_choice].opra.val) := st[ir[srca].tag].res;
} else {
  next(st[st_choice].opra.valid) := ~ir[srca].resvd;
  next(st[st_choice].opra.tag) := ir[srca].tag;
  next(st[st_choice].opra.val) := ir[srca].val;
}
```

Here, we have inserted a clause so that, if register `srca` is holding a tag, pointing to a completed reservation station, then we forward the `res` field of that reservation station as `opra` operand. Change the `oprb` logic correspondingly.

Finally, we introduce logic for choosing the reservation station to allocate (`st_choice`) and free (`complete_st`), so that reservation stations are used in round-robin order:

```
#define NUM_RS 32
  breaking(TAG){
    init(st_choice) := 0;
    init(complete_st) := 0;

    if(~stallout & opin = ALU)
      next(st_choice) := st_choice + 1 mod NUM_RS;
    if(st[complete_st].valid & st[complete_st].completed)
      next(complete_st) := complete_st + 1 mod NUM_RS;
```

Note, we chose here, arbitrarily, to use reservations stations numbered from 0 to 31 in the round-robin. Also note that since this logic breaks the symmetry of the type `TAG`, we have to put it in a `breaking` clause. If a new instruction is stored in a reservation station, we increment `st_choice` modulo 32. Similarly, if a reservation station is freed (*i.e.*, the station chosen to be freed is marked completed), then we increment `complete_st` module 32. This is done so that results of instructions are written to the register file in the same order that the instructions are received.

Now, open the new version and choose "Prop—Verify all". You should find that all of the properties are still true. That is, after this design change, the processor can be verified without modifying one line of the proof! This is because our three lemmas (for operands, results and noninterference) are not affected by the design change. Now, select an instance of `lemma1`, and look in the cone pane. You will notice that the signals `st_choice` and `complete_st` are free. This is because the assignments to these signals break the symmetry of type `TAG`, and thus cannot be used to verify this property, as we are using a symmetry reduction on type `TAG`. Thus, we have in fact verified the correctness of our design's data output independent of the definition of these signals, and have not used in any way the fact that these signals obey a round-robin policy. This should not be too surprising, as in the previous version of the design, no particular ordering was used. If we were to introduce some form of "exception", however, that interrupts the instruction stream, we would presumably need to use the round-robin policy to show that a consistent state is obtained after an exception.

Nonetheless, the fact that our proof was unaffected by the design change illustrates an important general point about compositional proofs. That is, our proof has the virtue that it only specifies the values of three key signals: the source operands in the reservation stations (lemma1), the value on the result bus (lemma2) and the tag on the result bus (lemma3). Since the function of these signals was not changed in adding the reorder buffer, our proof remained valid. In general, when designing a proof decomposition, it is best to do it in such a way that as few signals as possible are referenced. In this way, the proof will be less likely to be invalidated by localized design changes.

4.12 Proving liveness

Up to now, we've proved that our implementation of Tomasulo's algorithm is a refinement of an abstract model (in this case a sequential implementation of the same instruction set). However, we should note that a circuit that simply asserted the stall signal at every time unit would also satisfy this specification. Thus, we have shown that every behavior of the implementation is correct, in the sense that no bad outputs are produced, but we haven't shown that the circuit necessarily does any actual work. To do this, we also need to prove a *liveness* property.

The most obvious specification for liveness of the implementation is that it always eventually does not stall. We will begin, however, by proving something stronger: that every instruction eventually completes. Notice that this is a sufficient but not necessary condition for liveness. That is, if an instruction's result is never used as the source operand of a later instruction, then that instruction's failure to terminate would not cause any future stalls of the machine. However, we would also like to make sure that no reservation station is permanently lost as a resource, even if its result is never needed. Thus, we will prove that whenever a reservation station is full, it eventually becomes empty.

The proof of liveness follows the same basic lines as the refinement proof. That is, we break the liveness problem into two lemmas: one for operands, and one for results. The first lemma states that the operands of any given valid reservation station are always eventually valid. The second lemma states that a result for a given valid reservation station always eventually returns. As before, we construct a circular compositional proof, using operand liveness to prove result liveness, and *vice versa*. We will also use the same path splitting approach and data type reductions as in the refinement proof.

The main difference from the refinement proof is that we will need to fill in more detail about the resource allocation policies in order for the implemention liveness to be guaranteed. Up to now, we have left a number of choices completely nondeterministic, for example, the choice of which reservation station issue to an execution unit. However, in order to ensure that every instruction eventually executes, we will require that this choice be made in a fair way. Also, we will have to gauranteee that execution units always eventually finish. On the other hand, liveness does not depend in this case on data values, thus we will find that the data path logic does not enter into the proof.

4.13 Liveness lemmas

To begin with, let's take our implementation from the previous section and add two liveness lemmas. The first states (in temporal logic) that if a given reservation station holds a valid instruction, then its operands (`opra` or `oprb`) are eventually valid. Here is the lemma for `opra`:

```
forall (i in TAG)
  live1a[i] : assert G (st[i].valid -> F st[i].opra.valid);
```

In other words, at all times, if `rs[i]` is valid, then eventually the `opra` operand of `rs[i]` is valid. Write a similar lemma for the `oprb` operand.

Now, for the result liveness, lemma, we have:

```
forall (i in TAG)
  live2[i] : assert G (st[i].valid -> F ~st[i].valid);
```

That is, if `rs[i]` has a vaild instruction, then eventually the instruction completes, resulting in `rs[i]` being invalid. Note, we could have stated that eventually the result bus has a valid result with tag `pout.tag = i`. The two are equivalent, since the reservation station goes to the invalid state if and only if a corresponding result returns on the bus.

4.14 Path splitting

Now we consider the problem of proving the operand liveness lemma. As in the refinement proof, we observe that every operand consumed by a given reservation station `i` was produced by some reservation station `j`and stored in some source register `k`. If we split cases on the producer reservation station and the source register, we can show that the operand eventually arrives in any one case, using just two reservation stations and one register in the proof. Thus, add the following case splitting declaration for the `opra` operand:

```
forall(i in TAG) forall(j in TAG) forall(k in REG)
  subcase live1a[i][j][k] of live1a[i]
    for st[i].opra.tag = j & aux[i].srca = k;
```

Recall that `st[i].opra.tag` is the producer reservation station for the `opra` operand of reservation station `i`, and `aux[i].srca` is the source register of the `opra` operand, which we previously recorded in an auxiliary variable. Thus, the subcase `live1a[i][j][k]` states that (at all times), if reservation station `rs[i]` is holding an instruction, whose `opra` operand is to be produced by `rs[j]`, and stored in source register `ir[k]`, then eventually the `opra` operand will become valid.

Note that in the refinement proof, we also had to split cases on the data value. This is unnecessary in the livenes proof, however, since liveness does not depend on data. Note, also that we will have to assume that the producer reservation station eventually produces a valid result. However, this is allowed by the circular compositional rule, as we will see in the next section.

Now, for the results liveness lemma, we would like to prove that if a reservation station holds an instruction, it will eventually terminate. As before, we would like to split cases on the execution unit that produces the result, so that we can deal with an arbitrary number of execution units. This presents a slight problem, however, since at the time the reservation station becomes valid, the execution unit has not yet been chosen. In order to split cases, we therefore need to refer to a future value of a variable, in particular, the value of the execution unit choice at the time the instruction is issued. Fortunately, we can do this using a temporal logic operator.

The temporal logic formula `p when q` is true at a given time if `p` holds at the first occasion when `q` holds (and is taken to be true if `q` never holds). It is simply an abbreviation for `( q U (q & p))`. SMV recognizes that at any given time, for any given variable `v`,

```
    (v = i) when q
```

must be true for some value of `i` in the range of `v`. This allows us to split cases on a future value of a variable instead of the current. In this case, we can split the results lemma into cases based on the the future choice of execution unit in the following way:

```
forall(i in TAG) forall(j in EU)
  subcase live2[i][j] of live2[i]
    for (aux[i].eu = j) when st[i].issued;
```

That is, we split cases on the value of the variable `aux[i].eu` (the auxiliary variable that records execution unit choice) when the instruction is issued.

4.15 The circular compositional proof

Now, in order to prove that an operand eventually arrives at a consumer reservation station, we have to assume that the producer reservation station eventually yields a result. Similarly, to prove the result of a reservation station is eventually produced, we must assume that its operands eventually arrive.

While this argument is circular on its face, we can eliminate the circularity by introducing a time delay. Thus, to prove that operands are live at time t, we assume that results are live up to time $t-1$. This is sufficient, since if the consumer reservation station is valid at time t, the producer reservation must have been valid at some time $t-1$ or earlier (that is, the producer instruction must have arrived at an eariler time than the consumer instruction). In essence, we show that an operand of an instruction must eventually arrive assuming that all instructions arriving at earlier times eventually terminate.

To implement this argument, use the following declarations:

```
forall (i in TAG) forall(j in TAG) forall(k in REG)
  using pout//free, (live2[j]) prove live1a[i][j][k], live1b[i][j][k];

forall (i in TAG) forall(j in EU)
  using opra//free, oprb//free, live1a[i], live1b[i], prove live2[i][j];
```

That is, we assume that the producer reservation station `j` is live up to $t-1$ when proving the operands eventually arrive at the consumer. The time delay is indicated by putting the assumption `live2[j]` in parentheses. Then we can assume that operands are live up to time t when proving results are live up to t. SMV will detect the cirularity, but notice that it is broken by the time delay.

Note that, as in the refinement proof, we free the result bus when verifying the operands and free the operands when verifying the results. This breaks the system into two separate parts for verification.

4.16 Fairness

Now, open the new version. You should see several new properties in the properties pane: instances of `live1a`, `live1b` and `live2`. Select, for example, `live1a[1][0][0]`. This says that operands are always eventually forwarded from producer 0, via source register 0, to consumer 1. It should verify correctly.

On the other hand, try to verify `live2[0][0]`, which states that results for reservation station 0 always eventually arrive when using execution unit 0. For this property you should get a counterexample, where the reservation station is loaded with an instruction, obtains both its operands, and then waits forever to be issued to an execution unit. Note that many reasons are possible for this. For example, we have not specified `issue_choice`, which indicates the reservation station chosen for issue to an execution unit. Thus it is possible that reservation 0 is never chosen (a failure of fairness of the arbiter). Or, it is possible that reservation station 0 is chosen, but never at a moment that there is an available execution unit. Or, it is possible that `issue_eu`, which chooses an execution unit never choses an available, or that there is never an available unit because no execution unit ever terminates. Or, because we are using an abstraction where all execution units except for `eu[0]` are abstracted away (because of the default data type reduction), it is possible that `issue_eu` always choose a unit other than zero, and this unit, being abstracted away, always claims to be busy (in fact, this is the counterexample that I got).

For the moment, let's rule out all these possibilities by simply assuming that an instruction does not remain unissued forever with its operands ready. Later, when we actually implement a policy for `issue_choice` and `issue_eu`, we'll discharge this assumption. Here is one way to state this assumption:

```
forall (i in TAG) {
  issue_fair[i] : assert G F (st_ready[i] -> st_issue[i]);
  assume issue_fair[i];
}
```

That is, it is not possible that a reservation station remains ready and not issued. We define these terms as follows:

```
forall(i in TAG) {
  st_ready[i], st_issue[i] : boolean;
  st_ready[i] := st[i].valid & st[i].opra.valid & st[i].oprb.valid & ~st[i].issued;
  st_issue[i] := issue_choice = i & exe_rdy;
```

```
}
```

Now, add `issue_fair[i]` to the assumptions used to prove `live2[i][j]`. With this addition, try again to verify `live2[0][0]`. You should get another counterexample, this time where an instruction does get issued to execution unit 0, but the execution unit never completes. To correct this problem, let's add the assumption that execution units always eventually complete:

```
forall(i in EU){
  eu_fair[i] : assert G (eu[i].valid -> F ~eu[i].valid);
  assume eu_fair[i];
}
```

That is, we assume that if an execution unit becomes valid (contains an instruction), it eventually becomes invalid (completes). We'll have to discharge this assumption later when we fill in the details of the execution units and the completion arbitration. Add the assumption `eu_fair[j]` to those used to prove `live2[i][j]`. Now, try again to verify `live2[0][0]`. You should find the property true. Now try "Prop—Verify all". All the properties should be true, although the system will warn that there are unproved assumptions (the properties `issue_fair` and `eu_fair`).

4.17 Implementing the issue arbiter

Now we come to the problem of implementing an issue arbiter that guarantees the property `issue_fair`. That is, we want to choose `issue_choice` in such a way that every ready instruction is eventually issued. One way to do this is by using a *rotating priority* scheme. In this scheme, one requester (reservation station) is assigned highest priority. If this requester is rejected (*i.e.*, requests but is not acknowledged), it retains the highest priority. Otherwise, priority rotates to the next requester. In this way, we can guarantee that, if a resource (execution unit) always eventually becomes available, then all requesters will eventually be served (or withdraw their request). Here is an implementation of the issue arbiter (we leave the choice nondeterministic in the case where the high priority requester is not requesting):

```
issue_prio : TAG;

if(st_ready[issue_prio])
  issue_choice := issue_prio;
else issue_choice := {i : i in TAG};

breaking(TAG)
   if(~(st_ready[issue_prio] & ~exe_rdy))
     next(issue_prio) := issue_prio + 1 mod TAGS;
```

Note that by incrementing `issue_prio`, we break the symmetry of the type `TAG`. This means we have to enclose the assignment within a `breaking(TAG)` declaration, so disable type checking of type `TAG`. Further, we now have to explicitly declare the number `TAGS` of reservation stations. So let's change the declaration of type `TAG` to the following:

```
scalarset TAG 0..(TAGS-1);
```

Define TAGS to be some reasonable value (say 32). Similarly, set some reasonable number of execution units (say 4). Now, we need also to define a policy for choosing an available execution unit for issue. The simplest way to do this is to specify a nondeterministic choice among all the available (non-valid) execution units:

```
issue_eu := {i ?? ~eu[i].valid : i in EU};
```

Now, remove the statement

```
assume issue_fair[i];
```

and add instead:

```
breaking(TAG) breaking(EU) forall(i in TAG)
  using
    st_ready//free, exe_rdy//free, eu//free
  prove issue_fair[i];
```

Note, the `breaking` statements are used so that we can use assignments in the proof that break they symmetry of these types. Note also that we free the input signals of the arbiter; the arbiter should satisfy the fairness property for all possible inputs.

5 Synchronous Verilog

Those familiar with the Verilog modeling language may find it easier to write models for SMV in *Synchronous Verilog* (SV). This language is syntactically only a slight variation of the Verilog language. However its semantics is not based on an event queue model, as in Verilog. Rather, SV is a synchronous language, in the same family as Esterel, Lustre, and SMV. Because SV provides a functional description of a design rather than an operational description of how to simulate it, SV is better suited than Verilog to such applications as hardware synthesis, cycle-based (functional) simulation and model checking. Nonetheless, the meaning of most SV programs should be readily apparent to one familiar with modeling in Verilog.

5.1 Basic concepts

5.1.1 Synchrony

SV is a synchronous language. This means that all statements in SV (except the `wait` statement) execute in exactly zero time. For example, consider the following simple program:

```
module main();

wire x,y,z;
```

```
always
  begin
    x = y;
  end

always
  begin
    y = z;
  end

endmodule
```

In SV, the two `always` blocks execute exactly simultaneously, in zero time. As a result, the assignments `x = y` and `y = z` can be viewed as simultaneous equations. Therefore, it is true at all times that `x = z`. Because values on wires propagate in exactly zero time, there is no need for a notion of a triggering "event". That is, we need not (and may not) write

```
always @(y)
  begin
    x = y;
  end
```

In SV, any change in `y` is always reflected instantaneously in `x`.

As in other synchronous languages, the instantaneous propagation of signals can lead to "paradoxes". For example, if we write

```
wire x,y;

always
  begin
    x = y;
  end

always
  begin
    y = !x;
  end
```

then we have two simultaneous equations with no solution. On the other hand, in this case:

```
wire x,y;

always
  begin
    x = y;
  end
```

```
always
  begin
    y = x;
  end
```

we have simultaneous equations with two solutions: `x = 0, y = 0` and `x = 1, y = 1`. In a hardware implementation, these cases would correspond to combinational cycles in the logic. There are a number of ways of dealing with such cycles. However, we will leave the behavior in such cases undefined. The SMV system simply disallows combinational cycles.

5.1.2 Wires and registers

There are two distinct classes of signals in SV: wires and signals. These differ in two respects. First, a wire has no memory. It does not maintain its previous state in the case it is not assigned. Rather, the value of an unassigned wire is undefined. A register on the other hand will maintain its previous state when unassigned. Second, a value assigned to a wire propagates in exactly zero time. On the other hand, a register entails exactly one unit of delay: a value assigned to a register becomes visible exactly one time unit later.

For example, suppose we have:

```
wire x;
reg y;

always
  begin
    x = y;
  end

always
  begin
    y = z;
  end
```

The net result of this code is that the value of `x` lags the value of `z` by exactly one time unit. Note that although the result of an assignment to a register becomes visible one time unit later, the assignment statement itself executes in zero time. For example, consider the following block of code:

```
wire x,z;
reg y;

always
  begin
    y = z;
    x = y;
  end
```

The effect of this code is that at all times `x = z`, whereas the register `y` lags `x` and `z` by one time unit. That is, within the always block, all statements except `wait` statements appear to execute in zero time. Thus, the assignment `y = z` executes in zero time, setting the value of `y` and then this value is assigned to `x`, again in zero time. However, an observer outside the always block sees the value of `y` with one time unit of delay. Another example:

```
reg [31:0] y;

initial y = 0;

always
  begin
    y = y + 1;
    y = y + 1;
  end
```

In this case, the observed sequence of values of `y` is 0,2,4,6,.... That is, the always block executes both assignment statements in exactly zero time, in effect adding 2 to `y`. This effect is seen outside the block one time unit later.

5.1.3 Wait statements

The only statement that takes time in SV is the `wait` statement. A statement of the form

```
  wait(cond)
```

causes a delay until the condition `cond` is true, but always delays at least one time unit. Thus, `wait(1)` always waits exactly one time unit. For example,

```
wire x;

always
  begin
    x = 0;
    wait(1);
    x = 1;
  end
```

results in the observed sequence of values 0,1,0,1,... for `x`. Note that a new iteration of an always block begins exactly one time unit after the previous iteration terminates.

5.1.4 Loops

A loop of the form

```
  while(cond)
    block
```

executes `block` as long as the condition `cond` is true. If `cond` is false, it falls through to the next statement in exactly zero time. The last statement of `block` must be a wait statement. As an example,

```
reg [1:0] x;

initial x = 0;

always
  begin
    while(x < 3)
      begin
        x = x + 1;
        wait(1);
      end
    x = 0;
  end
```

results in the sequence 0,1,2,3,0,1,... for `x`.

A `for` loop, on the other hand, must have static upper and lower bounds, and is unrolled at compile time. Thus, for example,

```
  for(i = 0; i < 4; i = i + 1)
    block(i)
```

is exactly equivalent to

```
    block(0);
    block(1);
    block(2);
    block(3);
```

The block in this case need not contain a wait statement.

5.1.5 Conditionals

The conditional statement of the form

```
  if(cond)
    block1
  else
    block2
```

executes `block1` if `cond` is true, and `block2` if cond is false. The evaluation of the condition takes exactly zero time.

5.1.6 Resolution

In a case where more than one value is assigned to a signal at exactly the same time, then the following resolution rule applies:

- If all assigned values are equal, then the signal is assigned the common value.
- If any assigned values are unequal, then the signal is assigned `X` (the undefined value).

This rule can be used, for example, to model a tristate bus. For example:

```
always
  begin
    if(enable1) bus = data1;
  end

always
  begin
    if(enable2) bus = data2;
  end
```

In this case, when only one of the two enable signals is true, then the bus is equal to the corresponding data signal. If both enables are true and the data values are the same, then `bus = data1 = data2`. Else `bus = X`.

5.1.7 Embedded assertions

An assertion of the form

```
  assert label: cond;
```

will evaluate `cond` whenever it executes (in zero time). If `cond` is ever false, the property named `label` is reported to be false. These assertions can be verified formally by SMV.

5.2 Example – traffic light controller

This example is a controller that operates the traffic lights at an intersection where two-way street running north and south intersects a one-way street running east. The goal is to design the controller so that collisions are avoided, and no traffic waits at a red light forever.

The controller has three traffic sensor inputs, `N_Sense`, `S_Sense` and `E_Sense`, indicating when a car is present at the intersection traveling in the north, south and east directions respectively. There are three outputs, `N_Go`, `S_Go` and `E_Go`, indicating that a green light should be given to traffic in each of the three directions.

```
module main(N_SENSE,S_SENSE,E_SENSE,N_GO,S_GO,E_GO);

  input  N_SENSE, S_SENSE, E_SENSE;
  output N_GO, S_GO, E_GO;
```

```
wire   N_SENSE, S_SENSE, E_SENSE;
reg    N_GO, S_GO, E_GO;
```

In addition, there are five internal registers. The register NS_Lock is set when traffic is enabled in the north or south directions, and prevents east-going traffic from being enabled. Similarly EW_LOCK is set when traffic is enabled in the east direction, and prevents north or south–going traffic from being enabled. The three bits N_Req, S_Req, E_Req are used to latch the traffic sensor inputs.

```
reg    NS_LOCK, EW_LOCK, N_REQ, S_REQ, E_REQ;
```

The registers are initialized as follows:

```
initial begin
  N_REQ = 0; S_REQ = 0; E_REQ = 0;
  N_GO = 0; S_GO = 0; E_GO = 0;
  NS_LOCK = 0; EW_LOCK = 0;
end
```

Always, if any of the sense bits are true, we set the corresponding request bit:

```
always begin if (!N_REQ & N_SENSE) N_REQ = 1; end
always begin if (!S_REQ & S_SENSE) S_REQ = 1; end
always begin if (!E_REQ & E_SENSE) E_REQ = 1; end
```

The code to operate the north-going light is then as follows:

```
always begin
  if (N_REQ)
    begin
      wait (!EW_LOCK);
      NS_LOCK = 1;
      N_GO = 1;
      wait (!N_SENSE);
      if (!S_GO) NS_LOCK = 0;
      N_GO = 0;
      N_REQ = 0;
    end
end
```

That is, when a north request is detected, we wait for the EW lock to be cleared, then set the NS lock, and switch on the north light. Note, these last two assignments occur simultaneously, since they execute in zero time. Then we wait for the north sensor to be off, indicating there is no more traffic in the north direction. We then clear the NS lock, but only if the south light is currently off. Otherwise, we might cause a collision of south and east traffic. Finally, we switch off the north light and clear the north request flag. Note, the last two actions occur simultaneously with switching off the lock, so there is no danger of having the lock off but the light on.

The code for the south light is similar.

```
  always begin
    if (S_REQ)
      begin
        wait (!EW_LOCK);
        NS_LOCK = 1; S_GO = 1;
        wait (!S_SENSE);
        if (!N_GO) NS_LOCK = 0;
        S_GO = 0; S_REQ = 0;
      end
  end
```

Finally, here is the code for the east light:

```
  always begin
    if (E_REQ)
      begin
        EW_LOCK = 1;
        wait (!NS_LOCK);
        E_GO = 1;
        wait (!E_SENSE);
        EW_LOCK = 0; E_GO = 0; E_REQ = 0;
      end
  end
```

This differs slightly from the north and south cases. When an east request is detected, we set the EW lock, and then wait for the NS lock to be cleared, turn on the light, wait for the traffic sensor to clear, and finally, clear lock, light and request.

There are two kinds of specification we would like to make about the traffic light controller. The first is called "mutex", and states that lights in cross directions are never on at the same time:

```
always begin
  assert mutex: !(E_GO & (S_GO | N_GO));
end
```

This assert statement executes at every time unit, and fails if the east light is on at the same time as either the north or the south lights.

Second, we have "liveness" specifications. For each direction, we specify that if the traffic sensor is on for a given direction, then the corresponding light is eventually on, thus no traffic waits forever at a red light:

```
always begin
  if (E_SENSE) assert E_live: eventually E_GO;
  if (S_SENSE) assert S_live: eventually S_GO;
  if (N_SENSE) assert N_live: eventually N_GO;
end
```

Notice that since assert statements execute in zero time, each of these statements executes once every time unit. Further, this shows the use if the "eventually" operator in an assertion. This is equivalent to the temporal logic operator `F`. For example, if at any time the assertion `E_live` executes, then `E_GO` must eventually be true.

Our traffic light controller is designed so that it depends on drivers not waiting forever at a green light. We want to verify the above properties given that this assumption holds. To do this, we write some "fairness constraints", as follows:

```
always begin
  assert E_fair: eventually !(E_GO & E_SENSE);
  assert S_fair: eventually !(S_GO & S_SENSE);
  assert N_fair: eventually !(N_GO & N_SENSE);
end
```

Each of these assertions states that, always eventually, it is not the case that a car is at a green light. To tell SMV to assume these "fairness" properties when proving the "liveness" properties, we say:

```
  using N_fair, S_fair, E_fair prove N_live, S_live, E_live;
  assume E_fair, S_fair, N_fair;

endmodule
```

In effect, we are telling SMV to ignore any execution traces where one of these assumptions is false. The fairness constraints themselves will simply be left unproved. Now, open this file and try to verify the property `mutex`. The result should be "false", and in the "Trace" panel, you should see a counterexample trace in which the north light goes off exactly at the time when the south light goes on. In this case, the north light controller is trying to set the NS lock bit at exactly the same time that the south light is trying to clear it. The result of this is undefined, hence SMV attempts to verify both cases. It reports the case where the NS lock bit is cleared, which allows the east light to go on, violating the mutex property.

To fix this problem, let's insure that this situation doesn't arise by making the south light wait to go on if the north light is currently going off. Change the code for the north light controller to the following (and make the corresponding change in the south light controller):

```
  always begin
    if (N_REQ)
      begin
        wait (!EW_LOCK & !(S_GO & !S_SENSE));
        NS_LOCK = 1;
        N_GO = 1;
        wait (!N_SENSE);
        if (!S_GO) NS_LOCK = 0;
        N_GO = 0;
        N_REQ = 0;
      end
  end
```

Open this new version and verify the property `mutex`. It should be true. Now try to verify `N_live`. It should come up false, with a counterexample showing a case where both the north and south lights are going off at exactly the same time. In this case neither the north code nor the south code clears the lock, because each thinks that the other light is still on. As a result, the lock remains on, which prevents an east request from being served. This leaves the EW lock set forever, hence the controller is deadlocked, and remains in the same state indefinitely (note the "repeat signs" on the last state).

To fix this problem, we'll have the north controller switch off the lock when the south light is either off, or going off (and make the corresponding change to the south light controller). Here is the new code for the north controller:

```
always begin
  if (N_REQ)
    begin
      wait (!EW_LOCK & !(S_GO & !S_SENSE));
      NS_LOCK = 1; N_GO = 1;
      wait (!N_SENSE);
      if (!S_GO | !S_SENSE) NS_LOCK = 0;
      N_GO = 0; N_REQ = 0;
    end
end
```

Open this new version and verify the properties `mutex`, `N_live`, `S_live` and `E_live`. They should all be true. Note that if you try to verify the fairness constraints `N_fair`, `S_fair` and `E_fair`, they will come up false. These are unprovable assumptions that we made in designing the controller. However, if we used the controller module in a larger circuit, we could (and should) verify that the environment we put the controller into actually satisfies these properties. In general, it's best to avoid unproved assumptions if possible, since if any of these assumptions is actually false, all the properties we "proved" are invalid.

5.3 Example – buffer allocation controller

This example is designed to control the allocation and freeing of buffers in, for example, a packet router. It will demonstrate how to embed assertions within Synchronous Verilog control constructs, such as `if` and `while` in order to specify temporal properties, without using temporal logic.

The controller keeps an array of "busy" bits, one for each available data buffer. The busy bit is true when the buffer is in use, and false otherwise. An input `alloc` indicates a request to allocate a new buffer for use. If there is a buffer available, the controller outputs the index of this buffer on a signal `alloc_addr`. If there is no buffer available, it asserts an output `nack`. To make the circuit a little more interesting, we'll add a counter that keeps track of the number of busy bits that are set. Thus `nack` is asserted when the count is equal to the total number of buffers. To begin with, we'll define the number of buffers to be 16, using a macro definition. We also need to define the log of this number, to indicate the number of bits in the buffer addresses.

```
‘define SIZE 16
‘define LOG_SIZE 4
module main(alloc,nack,alloc_addr,free,free_addr);
  input alloc;
  output nack;
  output [(‘LOG_SIZE-1):0] alloc_addr;
  input free;
  input [(‘LOG_SIZE-1):0] free_addr;

  reg busy[0:(‘SIZE - 1)];
  reg count[‘LOG_SIZE:0];

  initial begin
    busy = 0;
    count = 0;
  end
```

Here is the logic for the counter and the `nack` signal. Notice, we add one to the counter when there is an allocation request and `nack` is not asserted. We subtract one from the counter when there is a free request, and the buffer being freed is actually busy. Note, if we didn't check to see that the freed buffer is actually busy, the counter could get out of sync with the busy bits.

```
  always begin
    nack = alloc & (count == ‘SIZE);
    count = count + (alloc & ~nack) - (free & busy[free_addr]);
  end
```

Next we handle the setting and clearing of the busy bits:

```
  always begin
    if(free) busy[free_addr] = 0;
    if(alloc & ~nack) busy[alloc_addr] = 1;
  end
```

Note, that if a buffer is both freed and allocated at the same time, the net result is that its busy bit is set. Finally, we choose a buffer to allocate using a priority encoder. Our priority encoder is implemented as follows:

```
  always begin
    for(i = (‘SIZE - 1); i >= 0; i = i - 1)
      if (~busy[i]) alloc_addr = i;
  end
```

Note, the entire `for` loop executes in zero time. Also, in the case when all buffers are busy, `alloc_addr` is not assigned, and thus remains undefined (since it is a wire, not a register).

Now, we consider the problem of specifying the buffer allocator. We will write a separate specification for each buffer, stating that the given buffer is never allocated twice without being freed in the interim. This is a technique known as "decomposition", that is, breaking a complex specification of a system into smaller parts that can be verified separately. To make it simpler to state the specification, it helps to define some additional signals: a bit `allocd[i]` to indicate that buffer `i` is currently being allocated, and a bit `freed[i]` to indicate that buffer `i` is currently being freed:

```
wire [0:(`SIZE - 1)] allocd, freed;
for(i = 0; i < `SIZE; i = i +1)
 always
    begin
      allocd[i] = alloc & ~nack & alloc_addr == i;
      freed[i] = free & free_addr == i;
    end
```

Note, we used a `for` constructor to make an instance of these definitions for each buffer `i`. To write the specification that a buffer is not allocated twice, we simply write a block of code that waits for the given buffer to be asserted, then while it is not freed, asserts that it must not be allocated again. At the end, when the buffer is freed, we also assert that it is not simultaneously allocated again. Note that we have given both these assertions the same label `safe[i]`. Thus, a failure in either case will cause a failure of `safe[i]`.

```
for(i = 0; i < `SIZE; i = i + 1)
  always begin
    if (allocd[i]) begin
      wait(1);
      while(~freed[i]) begin
        assert safe[i]: ~allocd[i];
        wait(1);
      end
      assert safe[i]: ~allocd[i];
    end
  end
```

Now, let's verify this specification. Open the file and verify the property `safety[0]`. It should be true. You might want to modify the code so that the counter is decremented whenever `free` is asserted (whether or not the busy bit is set for the freed buffer). If you try to verify this version you will find that in fact the property `safety[0]` false, and get a counterexample showing a case where the counter gets out of sync.