8. Generalizing Abstraction Functions

In this handout, we give a number of examples of specs and code for which simple abstraction functions (of the kind we studied in handout 6 on abstraction functions) don’t exist, so that the abstraction function method doesn’t work to show that the code satisfies the spec. We explain how to generalize the abstraction function method so that it always works.

We begin with an example in which the spec maintains state that doesn’t actually affect its behavior. Optimized code can simulate the spec without having enough state to generate all the state of the spec. By adding history variables to the code, we can extend its state enough to define an abstraction function, without changing its behavior. An equivalent way to get the same result is to define an abstraction relation from the code to the spec.

Next we look at code that simulates a spec without taking exactly one step for each step of the spec. As long as the external behavior is the same in each step of the simulation, an abstraction function (or relation) is still enough to show correctness, even when an arbitrary number of transitions in the spec correspond to a single transition in the code.

Finally, we look at an example in which the spec makes a non-deterministic choice earlier than the choice is exposed in the external behavior. Code may make this choice later, so that there is no abstraction relation that generates the premature choice in the spec’s state. By adding prophecy variables to the code, we can extend its state enough to define an abstraction function, without changing its behavior. An equivalent way to get the same result is to use an abstraction relation and define a backward simulation from the code to the spec.

If we avoided extra state, too few or too many transitions, and premature choices in the spec, the simple abstraction function method would always work. You might therefore think that all these problems are not worth solving, because it sounds as though they are caused by bad choices in the way the spec is written. But this is wrong. A spec should be written to be as clear as possible to the clients, not to make it easy to prove the correctness of code for. The reason for these priorities is that we expect to have many more clients for the spec than implementers. The examples below should make it clear that there are good reasons to write specs that create these problems for abstraction functions. Fortunately, with all three of these extensions we can always find an abstraction function to show the correctness of any code that actually is correct.

A statistical database

Consider the following spec of a “statistical database” module, which maintains a collection of values and allows the size, mean, and variance of the collection to be extracted. Recall that the mean \( m \) of a sequence \( db \) of size \( n > 0 \) is just the average \( \frac{\sum_{i} db(i)}{n} \), and the variance is \( \frac{\sum_{i} (db(i) - m)^2}{n} \).

\[
\sum_{i} \frac{(db(i) - m)^2}{n} = \frac{\sum_{i} db(i)^2}{n} - m^2. \quad \text{(We make the standard assumptions of commutativity, associativity, and distributivity for the arithmetic here.)}
\]

```markdown
EXPORT Add, Size, Mean, Variance =
VAR db : SEQ V := {}
APROC Add(v) = << db + := {v}; RET >>
APROC Size() -> Int = << RET db.size >>
APROC Mean() -> V RAISES {empty} = << IF db = {} => RAISE empty [*] VAR sum := (+ : db) | RET sum/Size() FI >>
APROC Variance() -> V RAISES {empty} = << IF db = {} => RAISE empty [*] VAR avg := Mean(), sum := (+ : {v :IN db | | (v - avg)**2}) | RET sum/Size() FI >>
END StatDB
```

This spec is a very natural one that follows directly from the definitions of mean and variance.

The following code for the StatDB module does not retain \( db \). Instead, it keeps track of the size, sum, and sum of squares of the values in \( db \). Simple algebra shows that this is enough to compute the mean and variance incrementally, as StatDBImpl does.

```markdown
% implements StatDB
EXPORT Add, Size, Mean, Variance =
VAR count := 0
sum := V.Zero() sumSquare := V.Zero()
APROC Add(v) = << count + := 1; sum + := v; sumSquare + := v**2; RET >>
APROC Size() -> Int = << RET count >>
APROC Mean() -> V RAISES {empty} = << IF count = 0 => RAISE empty [*] RET sum/count FI >>
APROC Variance() -> V RAISES {empty} = << IF count = 0 => RAISE empty [*] RET sumSquare/count - Mean()**2 | RET sum/Size() | RET sum/Size() FI >>
END StatDBImpl
```

StatDBImpl implements StatDB, in the sense of trace set inclusion. However we cannot prove this using an abstraction function, because each nontrivial state of the code corresponds to many
states of the spec. This happens because the spec contains more information than is needed to
generate its external behavior. In this example, the states of the spec could be partitioned into
equivalence classes based on the possible future behavior; two states are equivalent if they give
rise to the same future behavior. Then any two equivalent states yield the same future behavior
of the module. Each of these equivalence classes corresponds to a state of the code.

To get an abstraction function we must add history variables, as explained in the next section.

**History variables**

The problem in the StatDB example is that the spec states contain more information than the
code states. A *history variable* is a variable that is added to the state of the code $T$ in order to
keep track of the extra information in the spec $S$ that was left out of the code. Even though the
code has been optimized not to retain certain information, we can put it back in to prove the
code correct, as long as we do it in a way that does not change the behavior of the code. What we do is
to construct new code $TH$ (with History) that has the same behavior as $T$, but a bigger state. If
we can show that $TH$ implements $S$, it follows that $T$ implements $S$, since traces of $T$ = traces of
$TH \subseteq$ traces of $S$.

In this example, we can simply add an extra state component $db$ (which is the entire state of
StatDB) to the code StatDBImpl, and use it to keep track of the entire collection of elements,
that is, of the entire state of StatDB. This gives the following module:

```
MODULE StatDBImplH ... = % implements StatDB
VAR count := 0 % as before
sum := V.Zero() % as before
sumSquare := V.Zero() % as before
db := SEQ V := [] % history: state of StatDB

APROC Add(v) = <<
count + := 1; sum + := v; sumSquare + := v**2;
db + := [v] RET >>

% The remaining procedures are as before
END StatDBImplH
```

All we have done here is to record some additional information in the state. We have not
changed the way existing state components are initialized or updated, or the way results of
procedures are computed. So it should be clear that this module exhibits the same external
behaviors as the code StatDBImpl given earlier. Thus, if we can prove that StatDBImplH
implements StatDB, then it follows immediately that StatDBImpl implements StatDB.

However, we can prove that StatDBImplH implements StatDB using an abstraction function.
The abstraction function, $AF$, simply discards all components of the state except $db$. The
following invariant of StatDBImplH describes how $db$ is related to the other state:

```
INVARIANT count = db.size
```

That is, count, sum and sum$Square$ contain the number of elements in $db$, the sum of the
elements in $db$, and the sum of the squares of the elements in $db$, respectively.

With this invariant, it is easy to prove that $AF$ is an abstraction function from StatDBImplH to
StatDB. This proof shows that the abstraction function is preserved by every step, because the
only variable in StatDB, $db$, is changed in exactly the same way in both modules. The interesting
thing to show is that the $Size$, $Mean$, and $Variance$ operations produce the same results in both
modules. But this follows from the invariant.

In general, we can augment the state of code with additional components, called *history
variables* (because they keep track of additional information about the history of execution),
subject to the following constraints:

1. Every initial state has at least one value for the history variables.
2. No existing step is disabled by the addition of predicates involving history variables.
3. A value assigned to an existing state component does not depend on the value of a history
   variable. One important case of this is that a return value does not depend on a history
   variable.

These constraints guarantee that the history variables simply record additional state information
and do not otherwise affect the behaviors exhibited by the module. If the module augmented with
history variables is correct, the original module without the history variables is also correct,
because they have the same traces.

This definition is formulated in terms of the underlying state machine model. However, most
people think of history variables as syntactic constructs in their own particular programming
languages; in this case, the restrictions on their use must be defined in terms of the language
syntax.

In the StatDB example, we have simply added a history variable that records the entire state of
the spec. This is not necessary; sometimes there might be only a small piece of the state that is
missing from the code. However, the brute-force strategy of using the entire spec state as a
history variable will work whenever any addition of history variables will work.

**Abstraction relations**

If you don’t like history variables, you can define an abstraction relation between the code and
the spec; it’s the same thing in different clothing.

An abstraction relation is a simple generalization of an abstraction function, allowing several
states in $S$ to correspond to the same state in $T$. An abstraction relation is a subset of
$\text{states}(T) \times \text{states}(S)$ that satisfies the following two conditions:

1. If $t$ is any initial state of $T$, then there is an initial state $s$ of $S$ such that $(t, s) \in R$. 
The proof that \( AR \) is an abstraction relation is straightforward. We must show that the two properties in the definition of an abstraction relation are satisfied. In this proof, the abstraction relation is used to show that every response to a size, mean or variance query that can be given by StatDBImpl can also be given by StatDB. The new state of StatDB is uniquely determined by the code of StatDB. Then the abstraction relation in the prior states together with the code performed by both modules shows that the abstraction relation still holds for the new states.

An abstraction relation for MajReg

Consider the abstraction function given for MajReg in handout 5. We can easily write it as an abstraction relation from MajReg \& Register, not depending on the invariant to make it a function. Recall the types:

\[
\begin{align*}
\text{TYPE} & \quad F = [V, N] & \% \text{Pair of value and sequence number} \\
M & \quad = C -> P & \% \text{Memory: a pair at each copy} \\
\text{FUNC} & \quad AR(m, v) -> \text{Bool} = \text{VAR} n := m.rng.max.n | \text{RET } (P\{v, n\} \in m.rng)
\end{align*}
\]

For (1), suppose that \( t \) is any initial state of MajReg. Then there is some default value \( v \) such that all copies have value \( v \) and \( n = 0 \) in \( t \). Let \( s \) be the state of Register with value \( v \); then \( s \) is an initial state of Register and \( (t, s) \in AR \), as needed.

For (2), suppose that \( t \) and \( s \) are reachable states of MajReg and Register, respectively, with \( (t, s) \in AR \), and \( (t, \pi, t') \) a step of MajReg. Because \( t \) is a reachable state, it must satisfy the invariants given for MajReg. We consider cases, based on \( \pi \). Again, the interesting cases are the procedure bodies.

Abstraction relations vs. history variables

Notice that the invariant for the history variable \( db \) above bears an uncanny resemblance to the abstraction relation \( AR \). This is not an accident—the same ideas are used in both proofs, only they appear in slightly different places. The following table makes the correspondence explicit.

<table>
<thead>
<tr>
<th>Abstraction relation to history variable</th>
<th>History variable to abstraction relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given an abstraction relation ( AR ), define ( TH ) by adding the abstract state ( s ) as a state variable to ( T ). ( AR ) defines an invariant on the state of ( TH ): ( AR(t, s) ).</td>
<td>Given ( TH ), ( T ) extended with a history variable ( h ), there’s an invariant ( I(t, h) ) relating ( h ) to the state of ( T ), and an abstraction function ( AF(t, h) -&gt; S ) such that ( TH ) simulates ( S ).</td>
</tr>
<tr>
<td>Define ( AF((t, s)) = s )</td>
<td>Define ( AR(t, s) = (\exists h :: I(t, h) / / \ AF(t, h) = s) )</td>
</tr>
</tbody>
</table>

That is, \( t \) is related to \( s \) if there’s a value for \( h \) in state \( t \) that \( AF \) maps to \( s \).
For each step \((t, \pi, t')\) of \(T\), and \(s\) such that 
\(AR(t, s)\) holds, the abstraction relation gives 
us \(s'\) such that \((t, \pi, t')\) simulates \((s, s', \pi')\). Add \((t, s), (t', s')\) as a 
transition of \(\mathcal{H}\). This maintains the invariant.

This correspondence makes it clear that any code that can be proved correct using history 
variables can also be proved correct using an abstraction relation, and vice-versa. Some people 
prefer using history variables because it allows them to use an abstraction function, which may 
be simpler (especially in terms of notation) to work with than an abstraction relation. Others 
prefer using an abstraction relation because it allows them to avoid introducing extra state 
components and explaining how and when those components are updated. Which you use is just 
a matter of taste.

**Taking several steps in the spec**

A simple generalization of the definition of an abstraction relation (or function) allows for 
the possibility that a particular step of \(T\) may correspond to more or less than one step of \(S\). This is 
fine, as long as the externally-visible actions are the same in both cases. Thus this distinction is 
only interesting when there are internal actions.

Formally, a (generalized) abstraction relation \(R\) satisfies the following two conditions:

1. If \(t\) is any initial state of \(T\), then there is an initial state \(s\) of \(S\) such that \((t, s) \in R\).
2. If \(t\) and \(s\) are reachable states of \(T\) and \(S\) respectively, with \((t, s) \in R\), and \((t, \pi, t')\) is a step of 
   \(T\), then there is an execution fragment of \(S\) from \(s\) to some \(s'\), having the same trace, and with 
   \((t', s') \in R\).

Only the second condition has changed, and the only difference is that an execution fragment 
(of any number of steps, including zero) is allowed instead of just one step, as long as it has 
the same trace, that is, as long as it looks the same from the outside. We generalize the definition of 
an abstraction function in the same way. The same theorem still holds:

**Theorem 2:** If there is a generalized abstraction function or relation from \(T\) to \(S\), then \(T\) 
implements \(S\), that is, every trace of \(T\) is a trace of \(S\).

From now on in the course, when we say “abstraction function” or “abstraction relation”, we will 
mean the generalized versions.

Some examples of the use of these generalized definitions appear in handout 7 on file systems, 
where there are internal transitions of code that have no counterpart in the corresponding specs. 
We will see examples later in the course in which single steps of code correspond to several 
steps of the specs.

Here, we give a simple example involving a large write to a memory, which is done in one step 
in the spec but in individual steps in the code. The spec is:

```
MODULE RWMem [A, V] EXPORT BigRead, BigWrite =

TYPE M = A -> V

VAR memory : M

PROC BigWrite(m: M) = << memory := m; RET >> END RWMem
```

We can prove that \(\text{RWMemImpl}\) implements \(\text{RWMem}\) using an abstraction function. The state of 
\(\text{RWMemImpl}\) includes program counter values to indicate intermediate positions in the code, as 
well as the values of the ordinary state components. The abstraction function cannot yield partial 
changes to memory; therefore, we define the function as if an entire abstract \(\text{BigWrite}\) occurred 
at the point where the first change occurs to the memory occurs in \(\text{RWMemImpl}\). (Alternative 
definitions are possible; for instance, we could have chosen the last change.) The abstraction 
function is defined by:

\[
\text{RWMemImpl}.\text{memory} = \text{RWMemImpl}.\text{memory} \text{ unless pending is nonempty. In this case}
\]

\[
\text{RWMemImpl}.\text{memory} = \text{m}, \text{ where \text{BigWrite}(m) is the active \text{BigWrite} that made pending nonempty. \text{RWMemImpl}.pc for an active \text{BigRead} is the same as that for \text{RWMemImpl}. \text{RWMemImpl}.pc for an active \text{BigWrite} is before the body if the pc in \text{RWMemImpl} is at the beginning of the body; otherwise it is after the body.}
\]

In the proof that this is an abstraction function, all the atomic steps in a \(\text{BigWrite}\) of \(\text{RWMemImpl}\) 
except for the step that writes to memory correspond to no steps of \(\text{RWMem}\). This is typical: code 
for usually has many more transitions than a spec, because the code is limited to the atomic 
actions of the machine it runs on, but the spec has the biggest atomic actions possible because 
that is the simplest to understand.

Note that the guard in \(\text{RWMemImpl}.\text{BigRead}\) prevents a \(\text{BigRead}\) from returning an intermediate 
state of memory, which would be a transition not allowed by the spec. Of course this can’t happen 
unless there is concurrency.
In this example, it is also possible to interchange the code and the spec, and show that \texttt{RWMem} implements \texttt{RWMemImpl}. This can be done using an abstraction function. In the proof that this is an abstraction function, the body of a \texttt{BigWrite} in \texttt{RWMem} corresponds to the entire sequence of steps comprising the body of the \texttt{BigWrite} in \texttt{RWMemImpl}.

**Exercise:** Add crashes to this example. The spec should contain a component \texttt{OldStates} that keeps track of the results of partial changes that could result from a crash during the current \texttt{BigWrite}. A crash during a \texttt{BigWrite} in the spec can set the memory nondeterministically to any of the states in \texttt{OldStates}. A crash in the code simply discards any active procedure. Prove the correctness of your code using an abstraction function. Compare this to the specs for file system crashes in handout 7.

**Premature choice**

In all the examples we have done so far, whenever we have wanted to prove that one module implements another (in the sense of trace inclusion), we have been able to do this using either an abstraction function or else its slightly generalized version, an abstraction relation. Will this always work? That is, do there exist modules \(T\) and \(S\) such that the traces of \(T\) are all included among the traces of \(S\), yet there is no abstraction function or relation from \(T\) to \(S\)? It turns out that there do—abstraction functions and relations aren’t quite enough.

To illustrate the problem, we give a very simple example. It is trivial, since its only point is to illustrate the limitations of the previous proof methods.

Example: Let \texttt{NonDet} be a state machine that makes a nondeterministic choice of 2 or 3. Then it outputs 1, and subsequently it outputs whatever it chose.

\[
\begin{align*}
\text{MODULE NonDet EXPORT Out = } & \\
& \text{VAR i := 0} \\
& \text{APROC Out() -> Int = } << \\
& \quad \text{IF i = 0 => BEGIN i := 2 \[\] i := 3 END; RET 1 } \\
& \quad \text{[\star]} \text{ RET i FI >>} \\
& \text{END NonDet}
\end{align*}
\]

Let \texttt{LateNonDet} be a state machine that outputs 1 and then nondeterministically chooses whether to output 2 or 3 thereafter.

\[
\begin{align*}
\text{MODULE LateNonDet EXPORT Out = } & \\
& \text{VAR i := 0} \\
& \text{APROC Out() -> Int = } << \\
& \quad \text{IF i = 0 => i := 1 \[\star] i := 1 END; RET 1 } \\
& \quad \text{[\star]} \text{ SKIP FI >>} \\
& \text{END LateNonDet}
\end{align*}
\]

Clearly \texttt{NonDet} and \texttt{LateNonDet} have the same traces: Out() = 1; Out() = 2; ... and Out() = 1; Out() = 3; ... Can we show the implements relationships in both directions using abstraction relations?

Well, we can show that \texttt{NonDet} implements \texttt{LateNonDet} with an abstraction function that is just the identity. However, no abstraction relation can be used to show that \texttt{LateNonDet} implements \texttt{NonDet}. The problem is that the nondeterministic choice in \texttt{NonDet} occurs before the output of 1, whereas the choice in \texttt{LateNonDet} occurs later, after the output of 1. It is impossible to use an abstraction relation to simulate an early choice with a later choice. If you think of constructing an abstract execution to correspond to a concrete execution, this would mean that the abstract execution would have to make a choice before it knows what the code is going to choose.

You might think that this example is unrealistic, and that this kind of thing never happens in real life. The following three examples show that this is wrong: we will study code for all of these examples later in the course. We go into a lot of detail here because most people find these situations very unfamiliar and hard to understand.

**Premature choice: Reliable messages**

Here is a realistic example (somewhat simplified) that illustrates the same problem: two specs for reliable channels, which we will study in detail later, in handout 26 on reliable messages. A reliable channel accepts messages and delivers them in FIFO order, except that if there is a crash, it may lose some messages. The straightforward spec drops some queued messages during the crash.

\[
\begin{align*}
\text{MODULE ReliableMsg}[M] \text{ EXPORT Put, Get, Crash = } & \\
& \text{VAR q : SEQ M := {} } \\
& \text{APROC Put(m) = } << \text{ q + := {m} >>} \\
& \text{APROC Get() \rightarrow M = } << \text{ VAR m := q.head | q := q.tail; RET m >>} \\
& \text{APROC Crash() = } << \text{ VAR q' | q' <<= q => q := q' >>} \\
& \text{\% Drop any of the queued messages (\[\star\] is non-contiguous subsequence) } \\
& \text{END ReliableMsg}
\end{align*}
\]

Most practical code (for instance, the Internet’s TCP protocol) has cases in which it isn’t known whether a message will be lost until long after the crash. This is because they ensure FIFO delivery, and get rid of retransmitted duplicates, by numbering messages sequentially and discarding any received message with an earlier sequence number than the largest one already received. If the underlying message transport is not FIFO (like the Internet) and there are two undelivered messages outstanding (which can happen after a crash), the earlier one will be lost if and only if the later one overtakes it. You don’t know until the overtaking happens whether the first message will be lost. By this time the crash and subsequent recovery may be long since over.

The following spec models this situation by ‘marking’ the messages that are queued at the time of a crash, and optionally dropping any marked messages in Get.

\[
\begin{align*}
\text{MODULE LateReliableMsg}[M] \text{ EXPORT Put, Get, Crash = } & \\
& \text{END LateReliableMsg}
\end{align*}
\]
Like the simple NonDet example, these two specs are equivalent, but we cannot prove that LateReliableMsg implements ReliableMsg with an abstraction relation, because ReliableMsg makes the decision about what messages to drop sooner, in Crash. LateReliableMsg makes this decision later, in Get, and so does the standard code.

Premature choice: Multi-word clock

Here is a third example of premature choice in a spec: reading a clock. The spec is simple:

```
MODULE Clock
VAR t : Int  % the current time
THREAD Tick() = DO << t + := 1 >> OD  % demon thread advances
PROC Read() -> Int = VAR t1: Int | << t1 := t >>; << VAR t2 | t1 <= t2 /
                  t2 <= t => RET t2 >>
END Clock
```

This is in a concurrent world, in which several threads can invoke Read concurrently, and Tick is a demon thread that is entirely internal. In that world there are three transitions associated with each invocation of Read: entry, body, and exit. The entry and exit transitions are external because Read is exported.

We may want code for that allows the clock to have more precision than can be carried in a single memory location that can be read and written atomically. We could easily achieve this by locking the clock representation, but then a slow process holding the lock (for instance, one that gets pre-empted) could block other processes for a long time. A clever 'wait-free' code for Read (which appears in handout 17 on formal concurrency) reads the various parts of the clock representation one at a time and puts them together deftly to come up with a result which is guaranteed to be one of the values that \( t \) took on during this process. The following spec chooses the value to agree on as soon as the value is allowed. Outcome may return \( \text{nil} \) even after the choice is made because in distributed code it's possible that not all the participants have heard what the outcome is. Code for almost certainly saves up the allowed values and does a lot of communication among the processes to come to an agreement. The following spec has that form. It is more complicated than the first one (more state and more operations), and closer to code, using an internal Agree action to model what the processes do in order to choose a value.

```
MODULE LateConsensus [V] EXPORT Allow, Outcome =
VAR outcome : (V + Null) := nil  % Data value to agree on
VAR allowed : SET V := {}
APROC Allow(v) = << allowed / := {v} >>   % the current time
PROC Read() -> Int = VAR t1: Int | VAR t2 | t1 <= t2 /
                  t2 <= t => RET t2 >>
END LateConsensus
```

Again both specs have the same traces: a sequence of invocations and responses from Read, such that for any two Reads that don't overlap, the earlier one returns a smaller value \( t \). In the choice of \( t \) depends on when the body of Read runs relative to the various Ticks. In LateClock argument of some preceding Allow. But there is no abstraction relation from LateConsensus to Consensus, because there is no way for LateConsensus to come up with the outcome before it does its internal Agree action.

Note that if Outcome didn't have the option to return \( \text{nil} \) even after outcome # \( \text{nil} \), these modules would not be equivalent, because LateConsensus would allow the behavior

```
Allow(1); Outcome()=\text{nil}, Allow(2), Outcome()=1
```

and Consensus would not.

Premature choice: Multi-word clock

Here is a third example of premature choice in a spec: reading a clock. The spec is simple:

```
MODULE Clock
VAR t : Int  % the current time
THREAD Tick() = DO << t + := 1 >> OD  % demon thread advances t
PROC Read() -> Int = VAR t1: Int | VAR t2 |
                  t1 <= t2 /
                  t2 <= t => RET t2 >>
END Clock
```

This is in a concurrent world, in which several threads can invoke Read concurrently, and Tick is a demon thread that is entirely internal. In that world there are three transitions associated with each invocation of Read: entry, body, and exit. The entry and exit transitions are external because Read is exported.

We may want code for that allows the clock to have more precision than can be carried in a single memory location that can be read and written atomically. We could easily achieve this by locking the clock representation, but then a slow process holding the lock (for instance, one that gets pre-empted) could block other processes for a long time. A clever 'wait-free' code for Read (which appears in handout 17 on formal concurrency) reads the various parts of the clock representation one at a time and puts them together deftly to come up with a result which is guaranteed to be one of the values that \( t \) took on during this process. The following spec chooses the value to agree on as soon as the value is allowed. Outcome may return \( \text{nil} \) even after the choice is made because in distributed code it's possible that not all the participants have heard what the outcome is. Code for almost certainly saves up the allowed values and does a lot of communication among the processes to come to an agreement. The following spec has that form. It is more complicated than the first one (more state and more operations), and closer to code, using an internal Agree action to model what the processes do in order to choose a value.

```
MODULE LateConsensus [V] EXPORT Allow, Outcome =
VAR outcome : (V + Null) := nil  % Data value to agree on
VAR allowed : SET V := {}
APROC Allow(v) = << allowed / := {v} >>  % the current time
PROC Read() -> Int = VAR t1: Int | VAR t2 |
                  t1 <= t2 /
                  t2 <= t => RET t2 >>
END LateConsensus
```

Again both specs have the same traces: a sequence of invocations and responses from Read, such that for any two Reads that don't overlap, the earlier one returns a smaller value \( t \). In the choice of \( t \) depends on when the body of Read runs relative to the various Ticks. In LateClock argument of some preceding Allow. But there is no abstraction relation from LateConsensus to Consensus, because there is no way for LateConsensus to come up with the outcome before it does its internal Agree action.

Note that if Outcome didn’t have the option to return nil even after outcome # nil, these modules would not be equivalent, because LateConsensus would allow the behavior

```
Allow(1); Outcome()=nil, Allow(2), Outcome()=1
```

and Consensus would not.
the \texttt{VAR} \texttt{t2} makes the choice of \texttt{tr}, and it may choose a value of \texttt{t} some time ago. Any abstraction relation from \texttt{LateClock} to \texttt{Clock} has to preserve \texttt{t}, because a thread that does a complete \texttt{read} exposes the value of \texttt{t}, and this can happen between any two other transitions. But \texttt{LateClock} doesn’t decide its return value until its last atomic command, and when it does, it may choose an earlier value than the current \texttt{t}; no abstraction relation can explain this.

**Prophecy variables**

One way to cope with these examples and others like them is to use ad hoc reasoning to show that \texttt{LateSpec} implements \texttt{Spec}; we did this informally in each example above. This strategy is much easier if we make the transition from premise choice to late choice at the highest level possible, as we did in these examples. It’s usually too hard to show directly that a complicated module that makes a late choice implements a spec that makes a premature choice.

But it isn’t necessary to resort to ad hoc reasoning. Our trusty method of abstraction functions can also do the job. However, we have to use a different sort of auxiliary variable, one that can look into the future just as a history variable looks into the past. Just as we did with history variables, we will show that a module \texttt{TP} (\texttt{T} with \texttt{Prophecy}) augmented with a \textbf{prophecy variable} has the same traces as the original module \texttt{T}. Actually, we can show that it has the same \textit{finite} traces, which is enough to take care of safety properties. It also has the same infinite traces provided certain technical conditions are satisfied, but we won’t worry about this because we are not interested in liveness. To show that the traces are the same, however, we have to work backward from the end of the trace instead of forward from the beginning.

A prophecy variable guesses in advance some non-deterministic choice that \texttt{T} is going to make later. The guess gives enough information to construct an abstraction function to the spec that is making a premature choice. When execution reaches the choice that \texttt{T} makes non-deterministically, \texttt{TP} makes it deterministically according to the guess in the prophecy variable. \texttt{TP} has to choose enough different values for the prophecy variable to keep from ruling out any executions of \texttt{T}.

The conditions for an added variable to be a prophecy variable are closely related to the ones for a history variable, as the following table shows.

<table>
<thead>
<tr>
<th>History variable</th>
<th>Prophecy variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Every initial state has at least one value for the history variable.</td>
<td>1. Every state has at least one value for the prophecy variable.</td>
</tr>
<tr>
<td>2. No existing step is disabled by new guards involving a history variable.</td>
<td>2. No existing step is disabled \textit{in the backward direction} by new guards involving a prophecy variable. More precisely, for each step ((t, \pi, t')) and state ((t', p')) there must be a \texttt{p} such that there is a step ((t, \pi, t', p')).</td>
</tr>
<tr>
<td>3. A value assigned to an existing state</td>
<td>3. Same condition</td>
</tr>
</tbody>
</table>

If these conditions are satisfied, the state machine \texttt{TP} with the prophecy variable will have the same traces as the state machine \texttt{T} without it. You can see this intuitively by considering any finite execution of \texttt{T} and constructing a corresponding execution of \texttt{TP}, starting from the end. Condition (1) ensures that we can find a last state for \texttt{TP}. Condition (2) says that for each backward step of \texttt{T} there is a corresponding backward step of \texttt{TP}, and condition (3) says that in this step \texttt{p} doesn’t affect what happens to \texttt{t}. Finally, condition (4) ensures that we end up in an initial state of \texttt{TP}.

Let’s review our examples and see how to add prophecy variables (that all start with \texttt{p}), marking the additions with boxes. For \texttt{LateNonDetP} we add \texttt{pI} that guesses the choice between 2 and 3. The abstraction function is just \texttt{NonDet.i = LateNonDetP.pI}.

\begin{verbatim}
VAR i := 0
VAR pI := 0
APROC Out() -> Int = << IF i = 0 => i := 1; BEGIN pI := 2; END >>[
*] i = 1 => BEGIN pI = 2 => i := 2; END
[
*] i = 1 => BEGIN pI = 3 => i := 3; END
[*] SKIP FI;
RET i >>

For \texttt{LateReliableMsgP} we add a \texttt{pDead} flag to each marked message that forces \texttt{Get} to discard it. Choose crashes which \texttt{pDead} flags to set. The abstraction function just discards the marks and the dead messages.

\begin{verbatim}
VAR q : SEQ [m, mark: Bool, pDead: Bool] := {}
% ABSTRACTION FUNCTION ReliableMsg.q = {x :IN LateReliableMsg.q | ~ x.dead | x.m}
% INVARIANT (ALL i :IN q.dom | q(i).dead --> q(i).mark)
APROC Get() -> M = << DO VAR x := q.head |
q := q.tail; IF x.mark => SKIP |
RET x.m FI OD >>
APROC Crash() = << VAR pDeads; SEQ Bool | pDeads.size = q.size >>
q := {x :IN q; pDeads.in pDeads | x.mark := true, pDead := pD} |

Alternatively, we can prophesy the entire state of \texttt{ReliableMsg} as did with \texttt{db} in \texttt{StatDB}, which is a little less natural in this case:

\begin{verbatim}
VAR qD : SEQ M := {}
% INVARIANT (ALL i :IN q | m) <= pD /
qD << {x :IN q | m} |
APROC Get() -> M = << DO VAR x := q.head |
\end{verbatim}
\end{verbatim}

q := q.tail;
IF x.mark \[ \{pQ = [\_] / x.m \# pQ.head\} \] => SKIP
[\_] q := pQ.tail; RET x.m
FI OD >

Then we can propagate this set back through earlier commands until we get to the one that satisfies \( SE \); that will be too confusing. Instead, write another spec \( SL \) that makes the choice later, and use prophecy variables to show that \( SL \) implements \( SE \). Then show that \( T \) implements \( SL \); this shouldn’t require prophecy. We have given three examples of such \( SE \) and \( SL \) specs; the implementations are given in later handouts.

### Backward simulation

Just as we could use abstraction relations instead of adding history variables, we can use a different kind of relation, satisfying different start and step conditions, instead of prophecy variables. This new sort of relation also guarantees trace inclusion. Like an ordinary abstraction relation, it allows construction of an execution of the spec, working from an execution of the code. Not surprisingly, however, the construction works backwards in the execution of the code instead of forwards. (Recall the inductive proof for abstraction relations.) Therefore, it is called a **backward simulation**.

The following table gives the conditions for a backward simulation using relation \( R \) to show that \( T \) implements \( S \), aligning each condition with the corresponding one for an ordinary abstraction relation. To highlight the relationship between the two kinds of abstraction mappings, an ordinary abstraction relation is also called a **forward simulation**.

<table>
<thead>
<tr>
<th>Forward simulation</th>
<th>Backward simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. If ( t ) is any initial state of ( T ), then there is an initial state ( s ) of ( S ) such that ((t, s) \in R).</td>
<td>1. If ( t ) is any reachable state of ( T ), then there a state ( s ) of ( S ) such that ((t, s) \in R).</td>
</tr>
<tr>
<td>2. If ( t ) and ( s ) are reachable states of ( T ) and ( S ) respectively, with ((t, s) \in R), and ((t, \pi, t')) is a step of ( T ), then there is an execution fragment of ( S ) from ( s ) to some ( s' ), having the same trace, and with ((t', s') \in R).</td>
<td>2. If ( t ) and ( s' ) are states of ( T ) and ( S ) respectively, with ((t', s') \in R), ((t, \pi, t')) is a step of ( T ), and ( t ) is reachable, then there is an execution fragment of ( S ) from some ( s ) to ( s' ), having the same trace, and with ((t, s) \in R).</td>
</tr>
<tr>
<td>3. If ( t ) is an initial state of ( T ) and ((t, s) \in R ) then ( s ) is an initial state of ( S ).</td>
<td>3. If ( t ) is an initial state of ( T ) and ((t, s) \in R ) then ( s ) is an initial state of ( S ).</td>
</tr>
</tbody>
</table>

(1) applies to any reachable state \( t \) rather than any initial state, since running backwards we can start in any reachable state, while running forwards we start in an initial state. (2) requires that every backward (instead of forward) step of \( T \) be a simulation of a step of \( S \). (3) is a new condition ensuring that a backward run of \( T \) ending in an initial state simulates a backward run of \( S \) ending in an initial state; since a forward simulation never ends, it has no analogous condition.

**Theorem 3:** If there exists a backward simulation from \( T \) to \( S \) then every finite trace of \( T \) is also a trace of \( S \).
**Proof:** Start at the end of a finite execution and work backward, exactly as we did for forward simulations.

Notice that Theorem 3 only yields finite trace inclusion. That’s different from the forward case, where we get infinite trace inclusion as well. Can we use backward simulations to help us prove general trace inclusion? It turns out that this doesn’t always work, for technical reasons, but it works in two situations that cover all the cases you are likely to encounter:

- The infinite traces are exactly the limits of finite traces. Formally, we have the condition that for every sequence of successively extended finite traces of $S$, the limit is also a trace of $S$.
- The correspondence relation relates only finitely many states of $S$ to each state of $T$.

In the NonDet example above, a backward simulation can be used to show that $LateNonDet$ implements $NonDet$. In fact, the inverse of the relation used to show that $NonDet$ implements $LateNonDet$ will work. You should check that the three conditions are satisfied.

**Backward simulations vs. prophecy variables**

The same equivalence that holds between abstraction relations and history variables also holds between backward simulations and prophecy variables. The invariant on the prophecy variable becomes the abstraction relation for the backward simulation.

**Completeness**

Earlier we asked whether forward simulations always work to show trace inclusion. Now we can ask whether it is always possible to use either a forward or a backward simulation to show trace inclusion. The satisfying answer is that a combination of a forward and a backward simulation, one after the other, will always work, at least to show finite trace inclusion. (Technicalities again arise in the infinite case.) For proofs of this result and discussion of the technicalities, see the papers by Abadi and Lamport and by Lynch and Vondrager cited below.

**History and further reading**

The idea of abstraction functions has been around since the early 1970’s. Tony Hoare introduced it in a classic paper (C.A.R. Hoare, Proof of correctness of data representations. *Acta Informatica* 1 (1972), pp 271-281). It was not until the early 1980’s that Lamport (L. Lamport, Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems* 5, 2 (Apr. 1983), pp 190-222) and Lam and Shankar (S. Lam and A. Shankar, Protocol verification via projections. *IEEE Transactions on Software Engineering* SE-10, 4 (July 1984), pp 325-342) pointed out that abstraction functions can also be used for concurrent systems.

People call abstraction functions and relations by various names. ‘Refinement mapping’ is popular, especially among European writers. Some people say ‘abstraction mapping’.

History variables are an old idea. They were first formalized (as far as I know), in Abadi and Lamport, The existence of refinement mappings. *Theoretical Computer Science* 2, 82 (1991), pp 253-284. The same paper introduced prophecy variables and proved the first completeness result. For more on backward and forward simulations see N. Lynch and F. Vondrager, Forward and backward simulations—Part I: Untimed systems. *Information and Computation* 121, 2 (Sep. 1995), pp 214-233.