8. Generalizing Abstraction Functions

In this handout, we give a number of examples of specs and implementations 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 implementation 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. An optimized implementation can simulate the spec without having enough state to generate all the state of the spec. By adding *history variables* to the implementation, 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 implementation to the spec.

Next we look at implementations that simulate 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 specification correspond to a single transition in the implementation

Finally, we look at an example in which the spec makes a non-deterministic choice sooner than the choice is exposed in the external behavior. An implementation 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 implementation, 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 a *backward simulation* from the implementation 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 an implementation. 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 implementation that actually is correct.

A statistical database

Consider the following specification 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 $\frac{\sum_i \left(db(i)-m\right)^2}{n} = \frac{\sum_i db(i)^2}{n} - m^2$. (We make the standard assumptions of commutativity, associativity, and distributivity for the arithmetic here.)

```
MODULE StatDB [ V WITH {Zero: ()->V, "+": (V,V)->V, (V,V)->V, "-": (V,V)->V,
                        "/": (V,Int)->V} ] 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 directly follows the definitions of mean and variance.

The following implementation of the `StatDB` module does not retain the entire collection of values. Instead, it keeps track of the size, sum, and sum of squares of the values in the collection. Simple algebra shows that this is enough to compute the mean and variance in the manner done below.

```
MODULE StatDBImpl                                   % implements StatDB
    [ V WITH {Zero: ()->V, "+": (V,V)->V, (V,V)->V, "-": (V,V)->V,
              "/": (V,Int)->V} ] 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
    [*]  VAR avg := Mean() | RET sumSquare/count – avg**2
    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 implementation corresponds to many states of the specification. This happens because the specification contains more information than is needed to generate the desired external behavior. In this example, the states of the specification could be partitioned into equivalence classes based on their correspondence with states of the implementation: two states are equivalent if they correspond to the same implementation state. Then any two equivalent states yield the same future behavior of the module.

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 specification states contain more information than the implementation states. A *history variable* is a variable that is added to the state of the implementation *T* in order to keep track of the extra information in the specification *S* that was left out of the implementation. Even though the implementation has been optimized *not* to retain certain information, we can put it back in to prove the implementation correct, as long as we do it in a way that does not change the behavior of the implementation. What we do is to construct a new implementation *TH* 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* ⊆ traces of *S*.

In this example, we can simply add an extra state component `db` to the implementation `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; RET >>
    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 implementation `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:


```
   count     = db.size
/\ sum       = (+ : db)
/\ sumSquare = (+ : {v :IN db | | Square(v)})
```

That is, `count`, `sum` and `sumSquare` 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`. In this proof, it is easy to show 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 is easy to see because of the invariant.

In general, we can augment the state of an implementation 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 must not depend on the value of a history variable. One important case of this is that a return value must 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 can be shown correct, it follows that the original module without the history variables is also correct, because they have the same traces.

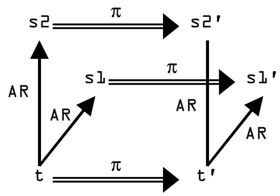
This definition is formulated in terms of the underlying state machine model. However, most people think of history variables as syntactic constructs in their own particular programming languages; in this case, the restrictions on their use must be defined in terms of the language syntax.

In the `StatDB` example, we have simply added a history variable that records the entire state of the specification. This is not necessary; sometimes there might be only a small piece of the state that is missing from the implementation. However, the brute-force strategy of using the entire specification 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 implementation 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 *states*(*T*) × *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 $(t, s) \in R$.
2. If *t* and *s* are reachable states of *T* and *S* respectively, with $(t, s) \in R$, and (t, π, t') is a step of *T*, then there is a step of *S* from *s* to some *s*', having the same trace, and with $(t', s') \in R$.

The picture illustrates the idea. It shows `t` related to `s1` and `s2`, and an action π taking each of them into a state related to `t'`.

It turns out that the same theorem holds as for abstraction functions:

Theorem 1: If there is an abstraction relation from *T* to *S*, then *T* implements *S*, that is, every trace of *T* is a trace of *S*.

The reason is that for *T* to simulate *S* it isn't necessary to have a function from *T* states to *S* states; it's sufficient to have a relation. A way to think of this is that the two modules, *T* and *S*, are running in parallel. The execution is driven by module *T*, which executes in any arbitrary way. *S* follows along, producing the same externally visible behavior. The two conditions above guarantee that there is always some way for *S* to do this. Namely, if *T* begins in any initial state *t*, we just allow *S* to begin in some related initial state *s*, as given by (1). Then as *T* performs each of its transitions, we mimic the transition with a corresponding transition of *S* having the same externally visible behavior; (2) says we can do so. In this way, we can mimic the entire execution of *T* with an execution of *S*.

Example: Consider the abstraction function given for `MajorityRegister` in handout 5. We can easily write it as an abstraction relation from `MajorityRegister` to `Register`, not depending on the invariant to make it a function.

```
FUNC AR(m, d) -> Bool = VAR seqno := {p' :IN m.rng | | p'.seqno}.max |
     RET (P{d, seqno} IN m.rng)
```

For (1), suppose that *t* is any initial state of `MajorityRegister`. Then there is some default value *d* such that all copies have value *d* and `seqno` 0 in *t*. Let *s* be the state of `Register` with value *d*; 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 `MajorityRegister` and `Register`, respectively, with $(t, s) \in AR$, and (t, π, t') a step of `MajorityRegister`. Because *t* is a reachable state, it must satisfy the invariants given for `MajorityRegister`. We consider cases, based on π. Again, the interesting cases are the procedure bodies.

An abstraction relation for `StatDB`

Recall that in the `StatDB` example we couldn't use an abstraction function to prove that the implementation satisfies the spec, because each nontrivial state of the implementation corresponds to many states of the specification. We can capture this connection with an abstraction relation. The relation that works is described in Spec[1] as:

```
TYPE T = [count: Int, sum: V, sumSquare: V]          % state of StatDBImpl
     S = [db: SEQ V]                                  % state of StatDB

FUNC AR(t, s) -> Bool =
     RET    db.size = count
         /\ (+ : db)) = sum
         /\ (+ : {v :IN db | | Square(v)}) = sumSquare
```

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.

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.

Abstraction relation to history variable	*History variable to abstraction relation*
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)`.	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) -> S` such that `TH` simulates `S`.
Define `AF((t, s)) = s`	Define `AR(t, s) =` `(EXISTS h | I(t, h) /\ AF(t, h) = s)` That is, `t` is related to `s` if there's a value for `h` in state `t` that `AF` maps to `s`.

[1] This is one of several ways to represent a relation, but it is the standard one in Spec. Earlier we described the abstraction relation as a set of pairs (*t*, *s*). In terms of `AR`, this set is `{t, s | AR(t, s) | (t, s)}` or simply `AR.set`, using one of Spec's built-in methods on predicates. Yet another way to write it is as a function `T -> SET S`. In terms of `AR`, this function is `(\ t | {s | AR(t, s)}` or simply `AR.setF`, using another built-in method. These different representations can be confusing, but different aspects of the relation are most easily described using different representations.

For each step `(t, π, t')` of `T`, and `s` such that `AR(t, s)` holds, the abstraction relation gives us `s'` such that `(t, π, t')` simulates `(s, π, s')`. Add `((t, s), p, (t', s'))` as a transition of `TH`. This maintains the invariant.	For each step `(t, π, t')` of `T`, and `h` such that the invariant `I(t, h)` holds, `TH` has a step `((t, h), π, (t', h'))` that simulates `(s, π, s')` where `s = AF(t, h)` and `s' = AF(t', h')`. So `AR(t', s')` as required.

This correspondence makes it clear that any implementation 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, π, 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 implementations that have no counterpart in the corresponding specifications. We will see examples later in the course in which single steps of implementations correspond to several steps of the specifications.

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 implementation. The spec is:


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

TYPE M           =  A -> D
VAR  memory      :  M

FUNC BigRead() -> M = RET memory

APROC BigWrite(m: M) = << memory := m; RET >>

END RWMem
```

The implementation is:

```
MODULE RWMemImpl [A,D] EXPORT BigRead, BigWrite =

TYPE M           =  A -> D
VAR  memory      :  M
     done        :  SET A := {}

FUNC BigRead() -> M = RET memory

PROC BigWrite(m) =
     << done := {} >>;
     DO << VAR a | ~(a IN done) => memory(a) := m(a); done + := {a} >> OD;
     RET

END RWMemImpl
```

We can prove that `RWMemImpl` implements `RWMem` using an abstraction function. The state of `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 `BigWrite` occurred at the point where the *first* change occurs to the memory occurs in `RWMemImpl`. (Alternative definitions are possible.) The abstraction function is defined by:

> `RWMem.memory = RWMemImpl.memory` unless there is an active `BigWrite` and `done` is nonempty; otherwise `RWMem.memory = m`, where `BigWrite(m)`is the active `BigWrite`. `RWMem`’s pc for an active `BigRead` is the same as that for `RWMemImpl`. `RWMem`'s pc for an active `BigWrite` is before the body if the pc in `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 `BigWrite` of `RWMemImpl` except for the step that writes to memory correspond to no steps of `RWMem`. This is typical: an implementation usually has many more transitions than a spec, because the implementation 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.

In this example, it is also possible to interchange the implementation and the specification, and show that `RWMem` implements `RWMemImpl`. This can be done using an abstraction function. In the proof that this is an abstraction function, the body of a `BigWrite` in `RWMem` corresponds to the entire sequence of steps comprising the body of the `BigWrite` in `RWMemImpl`.

Exercise: Add crashes to this example. The specification should contain a component `OldStates` that keeps track of the results of partial changes that could result from a crash during the current `BigWrite`. A `Crash` during a `BigWrite` in the specification can set the memory nondeterministically to any of the states in `OldStates`. A `Crash` in the implementation simply discards any active procedure. Prove the correctness of your implementation using an abstraction function.

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 are—abstraction functions and relations aren't quite enough.

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

Example: Let `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.

```
MODULE NonDet EXPORT Out =

VAR  i := 0

APROC Out() -> Int = <<
     IF  i = 0 => BEGIN i := 2 [] i := 3 END; RET 1 [*] RET i FI >>

END NonDet
```

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

```
MODULE LateNonDet EXPORT Out =

VAR  i := 0

APROC Out() -> Int = <<
     IF i = 0 => i := 1 [*] i = 1 => BEGIN i := 2 [] i := 3 END [*] SKIP FI;
     RET i >>

END LateNonDet
```

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

Well, we can show that `NonDet` implements `LateNonDet` with an abstraction function that is just the identity. However, no abstraction relation can be used to show that `LateNonDet` implements `NonDet`. The problem is that the nondeterministic choice in `NonDet` occurs before the output of `1`, whereas the choice in `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 implementation 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 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.

```
MODULE ReliableMsg [M] EXPORT Put, Get, Crash =

VAR  q  : SEQ M := {}

APROC Put(m)     = << q + := {m} >>
APROC Get() -> M = << VAR m := q.head | q := q.tail; RET m >>

APROC Crash()    = << VAR q' | q' <<= q => q := q' >>
% Drop any of the queued messages (<<= is non-contiguous subsequence)

END ReliableMsg
```

Most practical implementations (for instance, the Internet's TCP protocol) have 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`.

```
MODULE LateReliableMsg [M] EXPORT Put, Get, Crash =

VAR  q  : SEQ [m, mark: Bool] := {}

APROC Put(m)     = << q + := {m} >>
APROC Get() -> M =
    << DO VAR x := q.head | q := q.tail; IF x.mark => SKIP [] RET x.m FI OD >>

APROC Crash()    = << q := {x :IN q | | x{mark := true}} >>
% Mark all the queued messages. This is a sequence, not a set constructor, so it doesn't reorder the messages.

END LateReliableMsg
```

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`.

Premature choice: Consensus

For another examples, consider the *consensus* problem of getting a set of process to agree on a single value chosen from some set of allowed values; we will study this problem in detail later, in handout 18 on consensus. The spec doesn't mention the processes at all:

```
MODULE Consensus [V] EXPORT Allow, Outcome =

VAR outcome     :  (V + Null) := nil                    % Data value to agree on

APROC Allow(v) = << outcome = nil => outcome := v [] SKIP >>
FUNC  Outcome() -> (V + Null) = RET outcome

END Consensus
```

This spec chooses the value to agree on as soon as the value is allowed. An implementation 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 an implementation.

```
MODULE LateConsensus [V] EXPORT Allow, Outcome =

VAR outcome     :  (V + Null) := nil                    % Data value to agree on
    allowed     :  SET V := {}

APROC Allow(v) = << allowed + := {v} >>

FUNC  Outcome() -> (V + Null) = RET outcome

APROC Agree() = << VAR v | v IN allowed /\ outcome = nil => outcome := v >>

END LateConsensus
```

It should be clear that these two modules have the same traces: a sequence of `Allow(x)` and `Outcome() = y` actions in which every `y` is either `nil` or the same value, and that value is an 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.

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 EXPORT Read =

VAR t           :  Int                                   % the current time

THREAD Tick() = DO << t := t + 1 >> OD                   % demon thread advances t

PROC Read() -> Int = << RET t >>

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 an implementation 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' implementation of `Read` (which will appear 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 abstracts this strategy; it breaks `Read` down into two atomic actions and returns some value, non-deterministically chosen, between the values of `t` at these two actions.

```
MODULE LateClock EXPORT Read =

VAR t           :  Int                                   % the current time

THREAD Tick() = DO << t := t + 1 >> OD                   % demon thread advances t

PROC Read() -> Int = VAR t1: Int |
    << t1 := t >>; << VAR t2 | t1 <= t2 /\ t2 <= t => RET t2 >>

END LateClock
```

Again both specs have the same traces: a sequence of invocations and responses from `Read`, such that for any two `Read`s that don't overlap, the earlier one returns a smaller value `tr`. In `Clock` the choice of `tr` depends on when the body of `Read` runs relative to the various `Tick`s. In `LateClock` the `VAR t2` makes the choice of `tr`, and it may choose a value of `t` some time ago. Any abstraction relation from `LateClock` to `Clock` has to preserve `t`, because a thread that does a complete `Read` exposes the value of `t`, and this can happen between any two other transitions. But `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 `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 `LateX` implements `X`; we did this informally in each example above. This strategy is much easier if we make the transition from premature 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 *TP* augmented with a *prophecy variable* has the same traces as the original module *T*. Actually, we can show that it has the same *finite* traces, which is enough to take care of safety properties. It also has the same infinite traces provided certain technical conditions are satisfied. 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 *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 *T* makes non-deterministically, *TP* makes it deterministically according to the value of the prophecy variable. *TP* has to choose enough different values for the prophecy variable to keep from ruling out any executions of *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.

History variable	*Prophecy variable*
1. Every initial state has at least one value for the history variable.	1. Every state has at least one value for the prophecy variable.
2. No existing step is disabled by the addition of predicates involving a history variable.	2. No existing step is disabled *in the backward direction* by new guards involving a prophecy variable. More precisely, for each step (t, π, t') and state (t', p') there must be a p such that there is a step $((t, p), \pi, (t', p'))$.
3. A value assigned to an existing state component must not depend on the value of a history variable. One important case of this is that a return value must not depend on a history variable.	3. Same condition
	4. If t is an initial state of T and (t, p) is a state of *TP*, it must be an initial state.

If these conditions are satisfied, the state machine *TP* with the prophecy variable will have the same traces as the state machine *T* without it. You can see this intuitively by considering any finite execution of *T* and constructing a corresponding execution of *TP*, starting from the end. Condition (1) ensures that we can find a last state for *TP*. Condition (2) says that for each backward step of *T* there is a corresponding backward step of *TP*, and condition (3) says that in

this step *p* doesn't affect what happens to *t*. Finally, condition (4) ensures that we end up in an initial state of *TP*.

Let's review our examples and see how to add prophecy variables, marking the additions with boxes. For `LateNonDet` we add `p` which guesses the choice between 2 and 3. The abstraction function is just `NonDet.i = LateNonDetP.p`.

```
VAR  i := 0
     p := 0

APROC Out() -> Int = <<
     IF  i = 0 => i := 1; BEGIN p := 2 [] p := 3 END
     [*] i = 1 => BEGIN p = 2 => i := 2 [] p = 3 => i := 3 END [*] SKIP FI;
     RET i >>
```

For `LateReliableMsg` we add a `dead` flag to each marked message that forces `Get` to discard it. Crash chooses which `dead` flags to set. The abstraction function just discards the marks and the dead messages: `ReliableMsg.q = {x :IN LateReliableMsgP.q | ~ x.dead | x.m}`

```
VAR  q  : SEQ [m, mark: Bool, dead: Bool] := {}

APROC Get() -> M =
     << DO VAR x := q.head |
          q := q.tail; IF x.mark => SKIP [] ~ x.dead => RET x.m FI OD >>

APROC Crash()    = << VAR deads: SEQ Bool | deads.size = q.size =>
     q := {x :IN q, d :IN deads | | x{mark := true, dead := d}
```

For `LateConsensus` we follow the example of `NonDet` and just prophesy the outcome in `Allow`. The abstraction function is `Consensus.outcome = LateConsensus.p`

```
VAR  outcome     :  (V + Null) := nil                 % Data value to agree on
     p           :  (V + Null) := nil
     allowed     :  SET V := {}

APROC Allow(v) = << allowed + := {v}; IF p = nil => p := v [] SKIP FI >>

APROC Agree() =
     << VAR v | v IN allowed /\ outcome = nil /\ v = p => outcome := v >>
```

For `LateClock` we choose the result at the beginning of `Read`. The second command of `Read` has to choose this value, which means it has to wait until `Tick` has advanced `t` far enough. The transition of `LateClockP` that corresponds to the body of `Clock.Read` is the `Tick` that gives `t` the pre-chosen value. This seems odd, but since all these transitions are internal, they all have empty external traces, so it is perfectly OK.

```
VAR  t           :  Int                               % the current time
     p           :  Int

PROC Read() -> Int = VAR t1: Int |
     << t1 := t; VAR t': Int | p := t' >>;
     << VAR t2 | t1 <= t2 /\ t2 <= t /\ t2 = p => RET t2 >>
```

Most people find it much harder to think about prophecy variables than to think about history variables, because thinking about backward execution does not come naturally. It's easy to see that it's harmless to carry on extra information in the history variables that isn't allowed to affect the main computation. A prophecy variable, however, *is* allowed to affect the main computation, by forcing a choice that was non-deterministic to be taken in a particular way. Condition (2) ensures that in spite of this, no traces of *T* are ruled out in *TP*. It requires us to use a prophecy variable in such a way that for any possible choice that *T* could make later, there's some choice that *TP* can make for the prophecy variable's value that allows *TP* to later do what *T* does.

Here is another way of looking at this. Condition (2) requires enough different values for the prophecy variables p_i to be carried forward from the points where they are set to the points where they are used to ensure that as they are used, any set of choices that *T* could have made is possible for some execution of *TP*. So for each command that uses a p_i to make a choice, we can calculate the set of different values of the p_i that are needed to allow all the possible choices. Then we can propagate this set back through earlier commands until we get to the one that chooses p_i, and check that it makes enough different choices.

Because prophecy variables are confusing, it's important to use them only at the highest possible level. If you write a spec *SE* that makes an early choice, and implement it with a module *T*, don't try to show that T 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 this strategy.

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 specification, working from an execution of the implementation. Not surprisingly, however, the construction works *backwards* in the execution of the implementation 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*.

Forward simulation	*Backward simulation*
1. If *t* is any initial state of *T*, then there is an initial state *s* of *S* such that $(t, s) \in R$.	1. If *t* is any reachable state of *T*, then there a 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, π, 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 $(t', s') \in R$.	2. If *t*' and *s*' are states of *T* and *S* respectively, with $(t', s') \in R$, (t, π, 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 $(t, s) \in R$.
	3. If *t* is an initial state of *T* and $(t, s) \in R$ then *s* is an initial state of *S*.

(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.

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 interesting 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.

6.826—Principles of Computer Systems 1999

.