

Trace specifications in Alloy

Jeremy Jacob*

2008 November 26

1 Introduction

Alloy¹ is a light-weight modelling formalism whose underlying mathematics is the relational calculus; its tool is a model checker which can both refute invalid claims and find examples that satisfy consistent predicates. Jackson [2006] shows how to use Alloy to model systems in a state-based style that follows Z (and, in some respects, VDM), but where operations are not ‘first-class’. He also sketches how operations can be made first class².

The purpose of this note is to explain a style that removes state to leave nothing *but* the events. Essentially, it allows specifications in the style of Hoare [1985], as predicates describing the allowed sequences of events (or *traces*) of a system. Traces allow a complete description of the safety properties of a system; they do not allow us to discuss non-determinism and termination properties (the *failures* model is needed for those) or liveness properties. Safety properties are described pointwise: a system satisfies a specification if every one of its traces satisfies the property, and so refinement is modelled by implication. Certain properties, such as confidentiality, are only describable in terms of the entire set of traces (or failures or other observation) of the system [Jacob, 1992].

A sequence of all the events in a system’s history is (if we ignore clock time) the most detailed state of a non-terminated system we have access to. A trace may contain more information than we need to construct the current state, but it certainly contains enough. Thus we have a canonical form for specification states.

*University of York, England

¹See <http://alloy.mit.edu/>.

²This work has been extended since it was first published.

Traces can be pragmatically useful, too. I have found errors in a state-based specification by considering a trace based version, because no information is ever thrown away. The example involved an operation that adds pairs (let us say, a ticket and a time) to a relation; at most one ticket should be added for any time. The ‘delete-ticket’ operation removes all information from the database associated with the ticket; however the original specification deleted too much information, and a ticket can be recorded as issued at the same time as a ticket that has been deleted. Consideration of the trace-based specification made this problem much more obvious, because all the information about previously entered times is in the trace.

The major disadvantage of the style is that interesting traces are relatively long. In one example below, at least 9 events are necessary to get all possibilities for one cycle of a protocol communicating over a one-place buffer (and a second cycle may have started but not finished). Checks of interesting properties are expensive to analyse.

2 Encoding

The encoding strategy is to use the Alloy ordering utility to arrange the events in the analysis scope into a trace. The analyzer can then find all traces, of some fixed length³, that satisfy a given property.

We must ensure that the properties are *prefix-closed*, that is if they apply to a trace then they also apply to every initial sub-trace. In symbols, in the context of **sig** Event{} and **open** util/ordering[Event] the predicate **pred** p[t : **set** Event]{...} is prefix-closed if and only if:

all e : Event | **let** pfx=prevs[e] | p[pfx+e] **implies** p[pfx]

As a consequence a predicate is prefix-closed only if p[**none**]. As prefix-closure is a second-order predicate it cannot be programmed in Alloy, and consequently we have to program it explicitly.

The encoding is best explained by examples. The first (section 3) uses Hoare’s examples of vending machines, while the second (section 4), contains the more complex example of the triple redundancy protocol.

³Because of the way that the ordering utility is encoded, every event in scope must be in the ordering, and, moreover, the set of events must be non-empty.

```

1 module vending_machines
2
3 open util/ordering[Event]
4 fun nxt:Event→Event{{a:Event, b:nxt[a]}}
5
6 abstract sig Event{}
7
8 fun upto[e:Event]:set Event{prevs[e]+e}
9
10 sig Coin extends Event{}
11
12 pred no_vendor_loss[product : set (Event—Coin)]{
13     all e:Event | let pfx=upto[e] | #(product&pfx)<=#(Coin&pfx)
14 }
15 pred max_customer_loss[product : set (Event—Coin), max_loss : Int]{
16     all e:Event | let pfx=upto[e] | #(Coin&pfx)—max_loss<=#(product&pfx)
17 }

```

Listing 1: Simple trace specification: start of module

3 Example: vending machines

3.1 Module heading

Listing 1 shows the start of the module. It declares a signature to represent events in the system (Line 6), ensures that they are totally ordered in a displayable way (Lines 3–4), and defines a useful function to calculate prefixes (Line 8).

A signature defines a class of events that represent payment (Choc, Line 10) and two utility predicates capture properties a designer may wish to assert of all behaviours of a vending machine system. The first is that the vendor never makes a loss (`no_vendor_loss`, Lines 12–14): the number of products delivered never exceeds the number of coins paid. The second (`max_customer_loss`, Lines 15–17) states that the customer may make a loss when buying products, but no larger than some constant. Note that both of these predicates are written to be prefix-closed.

```

1 sig Choc extends Event{}
2
3 pred choc_vm[max_loss : Int]{
4     no_vendor_loss[Choc] and max_customer_loss[Choc, max_loss]
5 }
6
7 choc_vm0: run{choc_vm[0]} for 6 but 0 Drink_Event
8 choc_vm1: run{choc_vm[1]} for 6 but 0 Drink_Event
9 choc_vm2: run{choc_vm[2]} for 6 but 0 Drink_Event
10 choc_vm2a: run{choc_vm[2] and #Choc<#Coin} for 6 but 0 Drink_Event
11 vm1_is_vm2: check{
12     choc_vm[1] implies choc_vm[2]
13 } for 8 but 0 Drink_Event
14
15 pred alternate_coin_choc{
16     first in Coin
17     next in Coin->Choc + Choc->Coin
18 }
19
20 choc_alt_is_vm1: check{
21     alternate_coin_choc iff choc_vm[1]
22 } for 6 but 0 Drink_Event

```

Listing 2: Simple trace specification (cont.): simple vending machines

3.2 Very simple vending machines

The next part of the module (Listing 2) is inspired by the ‘chocolate vending machine’ specification of Hoare [1985, §1.10].

We add a signature to model delivery of a chocolate bar (Choc, Line 1), and a predicate to model a machine that satisfies the vendor of chocolate bars, but may only satisfy the customer up to some possible maximum loss (choc_vm, Lines 3–5). This predicate, when instantiated, defines a class of acceptable behaviours for a vending machine.

Five commands are given in Lines 7–13. The first four generate example traces for chocolate vending machines in which the customer may make no, one or two bars loss, and in the fourth case we look for examples in which the vendor makes a profit over-all (but not necessarily in any prefix). (The scopes restrict the signature ‘Drink_Event to 0; this signature is introduced in subsection 3.3) The first command, choc_vm0, finds no instance because the

ordering utility insists that there is at least one element in the ordering; it would be nicer if it allowed the empty ordering over an empty sequence as there is exactly one consistent trace that satisfies no loss to either party: the empty trace. The fifth command, `vm1_is_vm2`, is a refinement check: that every observable behaviour of `choc_vm[1]` is a possible observable behaviour of `choc_vm[2]`.

Lines 15–18 define a property, `alternate_coin_choc`, over `coin-choc` traces that says payments and deliveries strictly alternate, and that the first event must be a payment. The command on Lines 20–22 checks that (within the scope) alternation, starting with payment is an identical behaviour to that of no vendor loss plus a maximum loss to the customer of one bar. Thus, in this style, with one command we can compare the entire behaviour of two systems.

The vending machines described in this subsection are very simple. In the next section we describe more complex behaviours.

3.3 More complex vending machines

In this subsection we describe a family of vending machines that can deliver products from two classes, and allows the user to switch between them. This example is also derived from Hoare [1985, §1.1.4]. The code can be found in Listing 3.

The two products are drinks of orange and lemon, declared on Line 1; these signatures represent values and so are not extensions of `Event`. The events are: selecting which beverage the machine will deliver next (`Select`), and delivering it (`Drink`). These events are declared on Lines 2–3; note that they have a value field to say which beverage is involved. A different modeling strategy for the events is to use four separate signatures, but that is less convenient.

The property, `default`, on Lines 5–8 defines what it means for a selection event to affect later deliveries, and also describes what happens before a selection is made. If there is a latest selection, then the value selected is that delivered, otherwise a value from the default is delivered.

On Lines 10–14 a behaviour, `drink_orange_1_flex`, is defined in which: the vendor makes no loss on drinks, the customer may make a loss of one coin and the customer may select between orange and lemon, with the default being orange.

The behaviour `drink_orange_1_flex` allows a customer to insert a coin and then make any number of selections before taking a drink. We may wish to restrict selection to between transactions. The property `no_choice_within_transactions` (Lines 16–18) describes exactly this, and the behaviour described on Lines 20–25 is an example system that has this behaviour.

The two systems on Lines 27–37 illustrate two different types of choice. In the

```

1  enum Beverage{Orange, Lemon}
2  abstract sig Drink_Event extends Event{drink : Beverage }
3  sig Select, Drink extends Drink_Event{}
4
5  pred default[d : set Beverage]{
6      all e : Drink | let s=max[Select&prevs[e]] |
7          e.drink in (some s implies s.drink else d)
8  }
9
10 pred drink_orange_1_flex{
11     no_vendor_loss[Drink]
12     max_customer_loss[Drink, 1]
13     default[Orange]
14 }
15
16 pred no_choice_within_transactions{
17     all e : Select | let pfx=prevs[e] | #(Coin&pfx)=#(Drink&pfx)
18 }
19
20 pred drink_orange_2_inflex{
21     no_vendor_loss[Drink]
22     max_customer_loss[Drink, 2]
23     default[Orange]
24     no_choice_within_transactions
25 }
26
27 pred drink_random_a{
28     no_vendor_loss[Drink]
29     max_customer_loss[Drink, 1]
30     default[Orange] or default[Lemon]
31 }
32
33 pred drink_random_b{
34     no_vendor_loss[Drink]
35     max_customer_loss[Drink, 1]
36     default[Orange+Lemon]
37 }
38
39 rand_a_refines_rand_b: check {
40     drink_random_a implies drink_random_b
41 } for 8 but 0 Choc

```

Listing 3: Simple trace specification (cont.): less simple vending machines

```

1 module trp
2
3 open util/ordering[Event]
4 fun nxt:Event→Event{{a:Event, b:next[a]}}
5
6 fun upto[xs:set Event, x:Event]:set xs{prevs[x]&xs}
7 fun prefix[xs:set Event, x:Event]:set xs{upto[xs,x]+x&xs}
8 fun subseq[xs:set Event]:xs→xs{{disj a,b:xs | lt[a,b] and no nexts[a]&upto[xs,b]}}
9 pred show_match[disj ins, outs : set Event]{
10   some c : ins→outs | c={i : ins, o : outs | #(upto[ins, i])=#(upto[outs, o])}
11 }
12
13 abstract sig Event{value : Value}
14
15 enum Value{T, F}
16
17 sig In, Out, Mid, MidI, MidO extends Event{}

```

Listing 4: The triple redundancy protocol: header

first system, `drink_random_a`, the system decides on start-up whether the default is orange or lemon; however in `drink_random_b` a choice is made at each delivery before the first selection event. The check on Lines 39–41 tests whether the first system refines the second (it does).

4 Example: triple redundancy protocol

A more complex example is a little system composed of concurrent subsystems. Among the simplest of these are the simple protocols: we describe one in which the transmitter repeats each input bit three times over the network, and the receiver reads triples of bits from the network, passing on the majority value. The network will be represented by a one-place buffer, and a one-place corrupting buffer. Also, we describe an unbounded buffer and check that the sender-network-receiver system refines a buffer, for network implemented as a one-place buffer and a corrupting buffer.

4.1 The header

Listing 4 contains the header.

```

1 pred buffer[disj ins, outs : set Event]{
2   all e : outs | one {i : ins&prevs[e] | #(upto[ins,i])=#(upto[outs,e]) and i.value=e.value}
3 }
4
5 buffer: run{buffer[In,Out] and show_match[In, Out]} for 9 but 0 Mid, 0 Midl, 0 MidO
6
7 pred buffer_1[disj ins, outs : set Event]{
8   buffer[ins, outs]
9   subseq[ins+outs] in ins—>outs + outs—>ins
10 }
11
12 buffer_1: run {
13   buffer_1[In, Out] and show_match[In, Out]
14 } for 9 but 0 Mid, 0 Midl, 0 MidO
15
16 buffer_1_is_buffer: check {
17   buffer_1[In, Out] implies buffer[In, Out]
18 } for 5 but 0 Midl, 0 MidO

```

Listing 5: The triple redundancy protocol: buffers

As for the vending machines, we declare a carrier signature `Event`, but this time every event refers to some value (either `T` or `F`). Observable classes of events are communications on the five synchronisation channels `In`, `Out`, `Mid`, `Midl` and `MidO`.

Useful functions calculate the initial sub-sequence of some subclass of events `upto`, but excluding any event; the prefix, which is similar but includes the event; and the ordering relation restricted to a class of events (`subseq`). The fourth function (`show_match`) is a utility to persuade the visualiser to show how inputs and outputs in different sub-systems match; it is not a part of the modelling.

4.2 Buffers

Listing 5 contains the specification of two buffers.

A buffer (`buffer`) has the property that if an n th output exists, then the n th input is earlier and has the same value. Note that this places no constraints on the capacity of the buffer.

A one-place buffer (`buffer_1`) is a buffer that strictly alternates inputs and outputs (compare with `alternate_coin_choc` above, Listing 2). It includes the constraint that the capacity is one.

```

1 pred transmitter[disj ins, outs : set Event]{
2   let next=subseq[ins+outs] |
3   all e : ins+outs |
4     let prev_3=next.e+next.(next.e)+next.(next.(next.e)) {
5       e in outs implies one prev_3&ins and (prev_3&ins).value=e.value
6       e in ins implies no prev_3 or prev_3 in outs
7     }
8 }
9
10 transmitter: run {transmitter[In, Mid]} for 9 but 0 Out, 0 MidI, 0 MidO

```

Listing 6: The triple redundancy protocol: transmitter

The first two commands show how the utility `show_match` can be used to make the visualisation more helpful. The third allows us to have some confidence that a one-place buffer is a buffer (although this is a consequence of the way `buffer_1` is defined).

4.3 Transmitter

Listing 6 contains the specification of the transmitter.

An output of some value is allowed only if that value has been input in the previous three interactions, and no other input has occurred in that triple.

An input is allowed either at the start of the protocol or when there has not been an input for three events.

4.4 Receiver

Listing 6 contains the specification of the receiver.

An output is allowed if it follows exactly three inputs, and its value is the majority value of those inputs.

An input is allowed if there are fewer than three immediately preceding inputs.

4.5 The transmitter/receiver pair is a protocol

The definition of a protocol is that a pipeline composed of the transmitter and receiver refines a buffer. This is easily checked (in some finite scope): see Listing 8.

```

1 pred receiver[disj ins, outs : set Event]{
2   let next = subseq[ins+outs] |
3   all e : ins+outs |
4     let prev_3=next.e+next.(next.e)+next.(next.(next.e)) {
5       e in outs implies {
6         #prev_3&ins=3
7         e.value = (#(prev_3&value.T)>1 => T else F)
8       }
9       e in ins implies #prev_3&ins<3
10    }
11 }
12
13 receiver: run {receiver[Mid, Out]} for 9 but 0 In, 0 MidI, 0 MidO

```

Listing 7: The triple redundancy protocol: receiver

```

1 trp_traces: run{
2   transmitter[In, Mid]
3   receiver[Mid,Out]
4   show_match[In, Out]
5 } for 11 but 0 MidI, 0 MidO, 8 int
6
7 trp_correct: check{
8   transmitter[In, Mid] and receiver[Mid,Out] implies buffer[In, Out]
9 } for 11 but 0 MidI, 0 MidO, 8 int

```

Listing 8: The triple redundancy protocol: correctness of protocol

```

1 trp_1_traces: run{
2     transmitter[In, MidI]
3     buffer_1[MidI, MidO]
4     receiver[MidO, Out]
5     show_match[In, Out]
6     show_match[MidI, MidO]
7 } for 11 but 0 Mid, 5 int
8
9 trp_oo_traces: run{
10     transmitter[In, MidI]
11     buffer[MidI, MidO]
12     receiver[MidO, Out]
13     show_match[In, Out]
14     show_match[MidI, MidO]
15 } for 11 but 0 Mid, 8 int

```

Listing 9: The triple redundancy protocol: good networks

The first command generates examples of the system (again using the `show_match` utility to improve the output of the visualiser). Note how communication between the two sub-systems arises from synchronisation on the vents in the class `Mid`. The second command checks the correctness of the pair as a protocol.

4.6 The triple-redundancy protocol connected via good networks

In Listing 9 we see two commands to generate examples of three systems, that differ in their networks. The first is a one-place buffer and the second an unbounded buffer. Again, notice the use of `show_match` to improve the value of the visualisations.

We could easily write commands to check the correctness of these systems, but do not do so here.

4.7 The triple-redundancy protocol connected via a bad network

Last of all we can construct a network that corrupts, but not too often, and see if the protocol mitigates it. A corrupting network is defined and used in Listing 10.

The sub-system corrupter describes a buffer, except that an output may deliver a corrupt value, unless neither of the previous two are corrupt.

```

1 pred corrupter[disj ins,outs : set Event]{
2   all e : outs | let pos=#(upto[outs, e]) {
3     some i : ins&prevs[e] | #(upto[ins, i])=pos
4     ((val[outs, pos-1]!=val[ins, pos-1] or val[outs, pos-2]!=val[ins, pos-2])
5     implies
6     val[outs, pos]=val[ins, pos])
7   }
8   subseq[ins+outs] in ins->outs + outs->ins
9 }
10
11 pred show_corruption[disj ins, outs : set Event]{
12   some b : ins->outs |
13   b={i : ins, o : outs | #(upto[ins, i])=#(upto[outs, o]) and i.value!=o.value}
14 }
15
16 corrupter: run {
17   corrupter[In, Out]
18   show_match[In,Out]
19   show_corruption[In, Out]
20 } for 11 but 0 Mid, 0 MidI, 0 MidO, 5 int
21
22 trp_corrupter_traces: run{
23   transmitter[In, MidI]
24   corrupter[MidI, MidO]
25   receiver[MidO, Out]
26   show_match[In, Out]
27   show_match[MidI, MidO]
28   show_corruption[MidI, MidO]
29 } for 11 but 0 Mid, 5 int
30
31 trp_useful: check{
32   transmitter[In, MidI] and corrupter[MidI, MidO] and receiver[MidO, Out]
33   implies
34   buffer[In, Out]
35 } for 11 but 0 Mid, 5 int

```

Listing 10: The triple redundancy protocol: a bad network

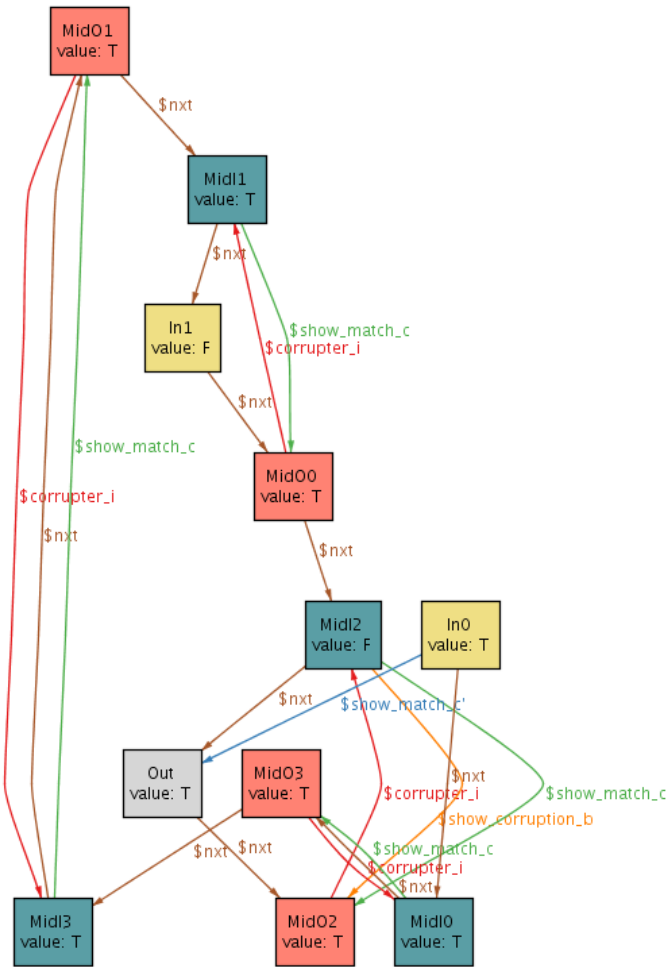


Figure 1: A trace of the Triple Redundancy Protocol over a corrupting network (using 'Magic layout').

Predicate `show_corruption` is another utility predicate, that picks out corruptions in the visualiser.

The command `trp_corrupter_traces` lets us investigate examples of the behaviour of the triple-redundancy protocol in the presence of (mild) corruption. The final check gives confidence that the triple-redundancy protocol is useful, as well as correct, by showing that the whole system corrects for certain types of corruption in the network.

Figure 1 shows a trace obtained from running the protocol with a corrupting network.

5 Conclusions

Properties of traces —with traces encoded as a total ordering of the events in scope— is a good way to think about safety properties of a variety of systems. It is flexible, allows simple statements of refinement and shows examples in a useful way through the Alloy analyzer. A library of useful predicates and functions could ease the process of writing trace specifications, although it is too early to say exactly what would be in it.

Encoding failures, in order to also specify maximal degrees of non-determinism, would require quite a different approach as a failure is a pair of a trace and a set that may have empty intersection. Whether the gains of such an encoding are worth the effort (especially considering the loading on the model checker would mean only very small examples could be checked). Other model checkers, such as FDR, are optimised for such checking (although the language is that of processes rather than individual observations).

References

- C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall, Hemel Hempstead, 1985. ISBN 0131532715.
- Daniel Jackson. *Software Abstractions: Logic, language and analysis*. MIT Press, Cambridge, MA, 2006. ISBN 0-262-10114-9.
- Jeremy L. Jacob. Basic theorems about security. *Journal of Computer Security*, 1 (4):385–411, 1992. ISSN 0926-227X.