TV Guide

David Faitelson

January 15, 2004

Contents

Chapter 1

Building and Installing TV

Chapter 2

How to use T✓

In order to use T✓ we first specify the properties we wish to test, by writing them in T✓'s language. Then we add traces to the program and run it on a set of test cases. Then we run T✓ giving it both the description of the properties and the output of the program. If the output violates one of the properties, T✓ will print the first position in the trace that caused the violation together with the state of the process (more on that later) at the point were it refused to accept the trace. See figure 2.1.

2.1 Tracing

T✓ does not work on arbitrary input. It requires the output of the program to be in a specific format. This format is an ASCII file that consists from a sequence of events (that we call a trace) each event occupying a single line.

The task of inserting traces into the program being tested, is beyond the scope of T✓. The interface between T✓ and the programs that it tests is a simple ASCII file (or named pipe, see section 2.3 below). This makes T✓ independent from the language that was chosen to implement the program, which makes it useful as a generic testing tool.

2.2 The T✓ language

In this section we illustrate how the different elements of the T✓ language can be used to verify programs. We begin with simple examples but they become more and more sophisticated as we progress.

Since T✓ works by analysing a trace file, we don't have to write a program that produce the traces. It is much easier to create the trace files by hand.

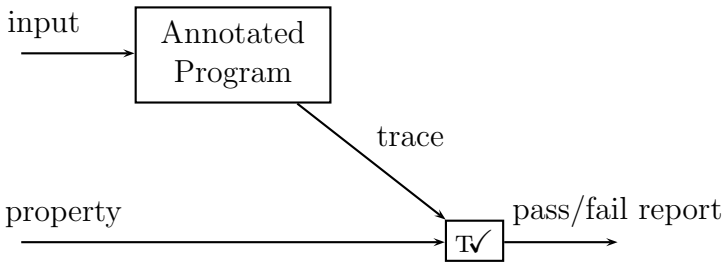

Figure 2.1: The test case input is fed to the program which is annotated to produce a trace file. The specification of the property we wish to test, and the trace file are fed to T✓ which reports if the trace is valid or not.

2.2.1 Vending Machines

Although vending machines are not popular projects among programmers, they provide a lot of insight into the kind of things we do here, and they are simple to understand, so we will play with them a bit before moving to more interesting things.

Suppose we build a software controller for a vending machine. The machine accepts a coin and then offers a chocolate bar. The T✓ description of the behaviour of the machine is:

```
VM = coin -> choc -> Stop

=] VM
```

The last line in this short model is called an *assertion*. It specifies the process that T✓ will use to check against the trace. This description will accept the trace

```
coin
```

and the trace

```
coin
choc
```

but it won't accept the trace

```
coin
coin
```

Here is what you get when you run T✓ on the model and the traces:

```
% tv vm.tv coin.trace
Trace accepted, process is now:

choc -> VM

Processed 2 lines

% tv vm.tv coincoin.trace
Trace rejected by process:

choc -> VM

At line 2
```

Note: The prefix operator `e -> P` insists that the trace contains the event e, and then checks the rest of the trace file against the description of P.

Note: Trace files contain a sequence of events, each event in a single line. Events can contain any kind of alphanumeric character but they must begin with a lower case letter or digit.

2.2.2 Choice

The vending machine is now extended to provide a choice of products. After you put a coin into the machine you can either select a chocolate or a bottle of juice:

```
VM = coin -> (choc -> Stop
              []
              juice -> Stop)

=] VM
```

This machine will accept any trace that begins with coin and follows either with choc or with juice.

2.2.3 Sharing

Imagine we want to check the validity of a relay race. There are two runners, the first must pass a stick to the second before the second can begin running. Each runner signals a start event when it begins running and a stop event when it stops.

```
R1 = start1 -> pass -> stop1 -> Stop

R2 = pass -> start2 -> stop2 -> Stop

=] R1 [| pass |] R2
```

The two runners *share* the pass event. This means that both of them must participate in it simultaneously: the first runner passes the stick to the second runner and the second runner accepts the stick.

In terms of traces it means that $R1$ can only start after the pass event, and that $R2$ can not stop before the pass event. The pass event must happen exactly once, indicating that the stick is passed from $R1$ to $R2$.

When two processes share a set of events they must perform these events together before they can go on, but they can proceed independently on all other events.

2.2.4 Interleaving

The next example is a test for a program that has five threads, each performing a different calculation. We know the result that each thread produces, but since the threads run independently of each other, the output that they produce changes every time. There are $5! = 120$ different combinations of legal outputs. Instead of writing all of them explicitly, we can use the following T✓ program:

```
P1 = 1 -> Stop
P2 = 2 -> Stop
P3 = 3 -> Stop
P4 = 4 -> Stop
P5 = 5 -> Stop

=] P1 ||| P2 ||| P3 ||| P4 ||| P5
```

Each thread is modelled as a process that emits a single event, i, and then stops. The threads are then interleaved to form the process that describes the behaviour of the complete system.

Note: The interleaving operator `|||` is equivalent to the parallel operator `[| |]` with an empty set of events.

Assuming that thread i produces output i, the T✓ program above will verify that the output trace is one of the 120 legal outputs we expect, but any other output will be flagged as invalid.

Unfortunately, the model above will also pass as valid any subtrace of the 120 legal traces. Thus, if any of the threads won't produce output at all, but the others still behave properly (or indeed if no thread produces output) this won't be detected by the model.

Insisting that an event happens is a liveness property. In general, T✓ can't check liveness properties because it can only check what a program has done, not what it can in principle do.

However, in this case we can convert the liveness property into a safety property, by changing the program to emit a done event when all the threads have finished, and changing the model to accept the done event only if all the processes have responded correctly:

```
P1 = 1 -> done -> Stop
P2 = 2 -> done -> Stop
P3 = 3 -> done -> Stop
P4 = 4 -> done -> Stop
P5 = 5 -> done -> Stop

=] P1 [| done |] P2 [| done |] P3 [| done |] P4 [| done |] P5
```

Since all the processes synchornize on the event `done`, this event can't happen unless all of them have printed their calculation.

2.2.5 Process functions

Instead of having to write a separate process for each thread, we can create a single parameterised process. The examples above can be written more concisely thus:

```
P n = !n -> done -> Stop

=] P 1 [|done|] P 2 [|done|] P 3 [|done|] P 4 [|done|] P 5
```

This examples illustrates another feature of T✓. The event pattern `!n` means that T✓ expects to find an event that matches the value of the variable `n` instead of matching the literal string `n`. Such event variables can only take integer values.

2.2.6 Replicated processes

Instead of writing the long line of processes above, we can use the replicated version of the parallel operator to achieve the same effect:

```
P n = !n -> done -> Stop

=] [| done |] i:1..5 @ P i
```

2.2.7 Maintaining state

Parametrised processes help us maintain state. For example, suppose we want to model a counter. Each time it receives an `inc` event it goes up by one. Each time it receives a `dec` event it goes down by one.

```
Counter n = inc -> Counter (n+1)
            []
            dec -> Counter (n-1)

=] Counter 0
```

The state of the counter is kept in the parameter `n`. Every time we get an event, the state is modified. This is reflected by calling the process function with the modified value of `n`. When T✓ reports a failure, it also displays the state of the process at the point of the failure. This gives valuable information as to why the event was rejected. We demonstrate this in the next example,

2.2.8 Restricting events

What if we want to restrict our counter to go only from zero to five ? In order to do this we need some way to reject `inc` events when `n` is equal to five, and to reject `dec` events when `n` is zero. This can be done by providing a boolean expression right after the event :

```
Counter n = inc ^ n < 5 -> Counter (n+1)
            []
            dec ^ n > 0 -> Counter (n-1)

=] Counter 0
```

Suppose that we use the following trace with the model describe above:

```
inc
dec
dec
```

The second `dec` event will try to set the state of the counter to a negative value, but `dec` events are rejected when the current state of the counter is 0, therefore T✓ will report a failure and display the state of the `Counter` process at the point of the failure:

```
Trace rejected by process:

inc -> Counter 1
[]
dec ^ false -> Counter -1

At line 3
```

Although we don't see that the Counter's value is currently 0, we can easily infer this fact by observing that if the counter will get an `inc` event its value will become 1. We also see that it is impossible for the counter to accept a `dec` event at this point because the `dec` event is guarded by a condition that is always false.

2.2.9 Event patterns

Boolean predicates can also act on the event itself. For example, suppose we want to model a task that must complete in less than ten seconds. Assuming that the trace file contains the beginning and ending times of the task in seconds, the following model will do the job:

```
Task = ?s -> ?e ^ (e - s) < 10 -> Stop

=] Task
```

The `?` operator instructs T✓ to accept any numeric event, and will bind it to the name that follows the `?` sign. Note that the boolean predicate is part of the criteria for T✓ to chose or reject the event. In this example, the event that is bound to `e` will be rejected unless the difference between `e` and `s` is less than ten.

2.2.10 Compound patterns

The problem with the previous example is that nothing in the trace indicates that the first event occurs when the task starts, and the second event occurs when the task ends. We can require that this would be indicated explicitly by creating a compound pattern:

```
Task = start?s -> end?e ^ (e - s) < 10 -> Stop

=] Task
```

A compound pattern matches events that consists from strings separated by dots. A pattern such as `start?s` will match events like

```
start.4
start.5
```

2.3 Using TV online

Up to now we were using TV with a file of events. We now describe how it is possible to use TV on the trace that a program emits while it is running. The basic idea is to let the program write into a named pipe (FIFO) and set TV to read the trace from this FIFO.

```
% mkfifo trace
% program > trace &
% tv spec.tv trace
```

This approach has two problems. First, there is no copy of the trace. The events go from the program to TV and are never collected so when TV stops and displays an error message, we can't look at the trace. Second, if TV won't be fast enough to process the events the pipe will become full and the program will block. To avoid these two problems, we let the program write into a file, and set up the FIFO between `tail`[1] and TV. This way we solve both problems: the trace is recorded in a file, and the program never blocks.

```
% mkfifo tvin
% program > trace &
% tail +1lf trace > tvin &
% tv spec.tv tvin
```

[1] `tail +1lf` samples the file a few times a second, and copies to standard output the new data that was added since the last time it sampled the file

Chapter 3

Reference