Tricks for CSP Model Checking

Gavin Lowe

October 25, 2022

This document outlines a few tricks that can be useful when model checking using CSP and FDR.

1 Efficiency

Model checking can often take a long time. We describe some techniques for making a check more efficient.

The first step is to work out where the time is being spent. A check using FDR goes through two main stages.

1. Compilation, creating explicit state machines for each of the sequential components in the system and specification, and reducing the state machine for the specification to a normal form (comparable to a *deterministic* finite automaton);

2. Checking, exploring the cross product of the specification and the system.

When running a check in FDR, clicking on the box labelled "?" next to the check will pop up a box displaying the progress. During the compilation phase, this will say "Compiling". During the checking phase, the box displays horizontal orange lines, one for each ply explored so far, with the width of the line proportional to the number of states explored in that ply.

If the compiling phase is slow, run some more checks to work out where the problem lies. For a check $Spec \sqsubseteq System$, consider the following alternative checks:

- $STOP \sqsubseteq System$: if this is slow to compile, then the compilation of $System$ is the problem;

- $STOP \sqsubseteq Spec$: if this is slow, then the compilation of $Spec$ is the problem;

- $Spec \sqsubseteq STOP$: if this is slow and the previous item is reasonably fast, then the normalisation of $Spec$ is slow.

If a process that is composed of several sub-processes is slow, then run similar checks for suitable sub-processes, to discover which are causing the problem.

1.1 Improving compilation

Suppose a particular component P is slow to compile.

A common way to improve compilation is to split P into the parallel composition of several sub-processes. This is best exemplified by a process that represents a set. The process Set(S) below represents a set with contents S, with events to add an element, test if the set contains an element, and test if the set is empty. .

```
EmptySet = S({})
Set(S) =
  add?x → Set(S ∪ {x})
  □ contains?x!(x ∈ S) → Set(S)
  □ S = {} & empty → Set(S)
```

If the elements of S are taken from a type T of size n, then this set has 2^n states. The state machine created at compilation time will contain all of these states, even if only a small proportion are explored during the check itself.

Instead, the set can be modelled by a process formed as the parallel composition of n processes, one for each element of T, as follows.

```
EmptySet = ||| x ← T • [alpha(x)] NotContains(x)
NotContains(x) =   -- x not in the set
  add.x → Contains(x)
  □ contains.x.false → NotContains(x)
  □ empty → NotContains(x)
Contains(x) =      -- x in the set
  add.x → Contains(x)
  □ contains.x.true → Contains(x)
alpha(x) = {| add.x, contains.x, empty |}
```

Note that all the components synchronise on the event empty, so that event can occur only when every component is in the NotContains state, as required.