# Finite State Automata for Topological Sorting

Bill Mitchell

May 28, 2009

## 1 Introduction

A specification of concurrent communicating processes represents a causal relationship of the events contained in the specification. It is very simple to model this as a finite partial order over the set of events when the specification contains no iteration (e.g. for an MSC with no loop constructs). When there is iteration it is far more complex to express the causal relationships between the events within a mathematical setting.

This paper first explores, in section 2, how to describe all total extensions of a partial order as a finite state automaton (FSA). These total extensions represent the test scripts for a specification when there is no iteration. Then, in section 3 we extend these ideas to specifications which contain complex iterations. For MSC this means the inline loop construct. Section 3.2 describes the algorithm for characterising an iterative specification as a FSA.

### 1.1 Motivating Examples

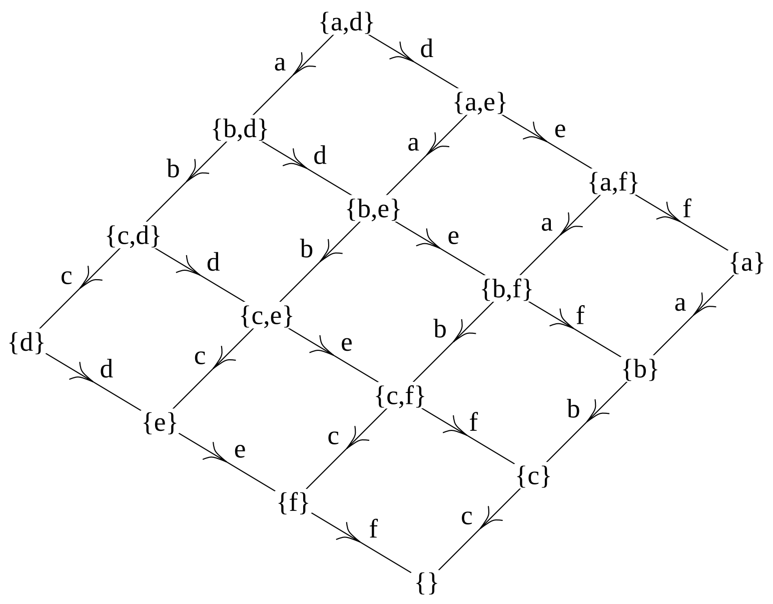
For a particular partial order how many ways are there to extend it to a total order. Within ptk we use the phrase topological sort to denote a total extension of a partial order. What is the most compact form we can represent the set of topological sorts of a partial order.

As an example consider the set $\{a, b, c\}$ which has empty partial order (i.e. completely unordered). Any total order of this set extends the empty partial order. The finite state automaton pictured below represents these total orders. The state $\{a, b, c\}$ is the start state, $\{\,\}$ is the accepting state. A total order on $\{a, b, c\}$ is given by exactly those words which are accepted by the automaton.

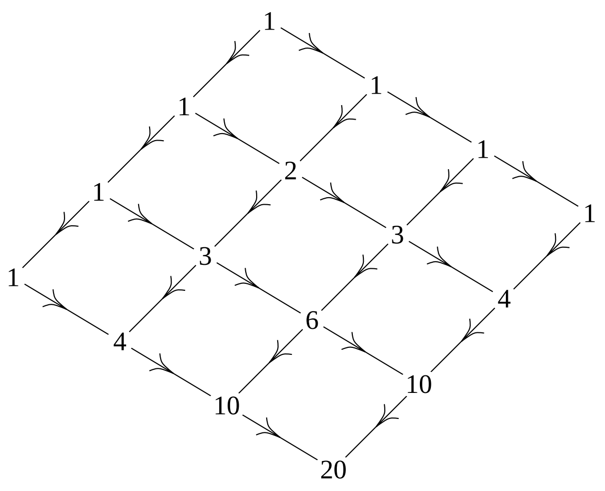
$$\begin{tikzcd}
 & \{a,b,c\} \arrow[dl, "a"'] \arrow[d, "b"] \arrow[dr, "c"] & \\
\{b,c\} \arrow[d, "b"'] \arrow[dr, "c"] & \{a,c\} \arrow[dl, "a"] \arrow[dr, "c"] & \{a,b\} \arrow[dl, "a"'] \arrow[d, "b"] \\
\{c\} \arrow[dr, "c"] & \{b\} \arrow[d, "b"] & \{a\} \arrow[dl, "a"'] \\
 & \{\} & 
\end{tikzcd}$$

This automaton of course accepts any permutation of the letters $a$, $b$ and $c$. The automaton is best visualised as a cube in three dimensional space. This automaton has the smallest number of states which will generate the topological sorts of the empty ordering on $\{a, b, c\}$.

As a second example consider the partial order on the set $\{a, b, c, d, e, f\}$ given by $a < b < c$ and $d < e < f$. The topological sorts of this partial order is represented by the following finite state automaton. In this example the start state is $\{a, d\}$ and the accepting state is $\{\}$. This is the smallest FSA which accepts this language.



From this automaton we can easily calculate how many topological sorts there are. Label the start state with value 1, for any state v label it with the sum of the values of its parent states. The label of the accepting state is the number of possible total extensions. Also the sum of the values of all the states is the number of nodes which would be required to express the total extensions in the form of a tree as is done in ptk. For this example we get a picture like this:



Notice that this picture is a fragment of Pascals triangle. It is precisely the fragment required to calculate the binomial coefficient $C^6_3 = 20$. The number of total extensions for this case is 20 which is the label of the accepting state, and the number of nodes which the tree representation would have is

$$1 + (1 + 1) + (1 + 2 + 1) + (1 + 3 + 3 + 1) + (4 + 6 + 4) + (10 + 10) + 20 = 69$$

## 2 The FSA algorithm for non iterating specifications

The algorithm is very simple to describe and is clearly closely related to the algorithm in ptk for generating the tree representation of the topological sorts. However this is not the case when an MSC contains the loop construct.

Let $<$ be a partial order on a set of events $S$. For convenience we think of $<$ as represented in graphical form. For an event $a$ let $\mathsf{n}(a)$ be the child events of $a$ in the graphical representation of $S$. That is $\mathsf{n}(a)$ is the set of events $b$ where $a < b$ and there is no $c \in S$ such that $a < c < b$, so $\mathsf{n}(a)$ is the set of events which are $\mathsf{n}$ext after $a$ according to $<$. For a set $X \subseteq S$ let

$$\mathsf{m}X = \{x \in X \mid \nexists y \in X : y < x\}$$

This is the set of minimal events in $X$.

For a given $X \subseteq S$ we can define a FSA $A(X)$ as follows. The initial state is $\mathsf{m}X$. The states in the automaton are certain subsets of $S$ (which we will define in a moment). Define a transition $U \xrightarrow{a} V$ whenever $a \in U$ and

$$V = \mathsf{m}((U - \{a\}) \cup \mathsf{n}(a))$$

The set of states $S_X$ and transitions $T_X$ for $A(X)$ are defined recursively as follows:

```
val $S_X = \{\mathsf{m}X\}$
val $T_X = \{\}$
val nextStates $= S_X$
val nextTrans $= \{\}$
fun trns $U = \{\ U \xrightarrow{u} \mathsf{m}((U - \{u\}) \cup \mathsf{n}(u))\ \mid u \in U\}$
fun nx $(U \xrightarrow{a} V) = V$
fun concat SetOfSets = foldl $\cup$ { } SetOfSets
val $(S_X, T_X) =$
                while (not (nextStates = { }))
                    { let
                          val nextTrans = concat (map trns nextStates)
                          val nextStates = map nx nextTrans
                          val $S_X = S_X \cup$ nextStates
                          val $T_X = T_X \cup$ nextTrans
                      in
                          $(S_X, T_X)$
                      end
                    }
```

Recall that for a function $f : A \longrightarrow B$, if $X$ is a set of elements of $A$, then $\mathsf{map}\ f\ X = \{f(x) \mid x \in A\}$. Note that $\{\,\} \in S_X$ after the while loop terminates. The start state of $A(X)$ is $\mathsf{m}X$, the final accepting state is $\{\,\}$. Define $\mathcal{A} = A(S)$, this is the FSA which characterises the topological sorts of $<$.

For a string $\alpha = a_0 \cdots a_n$ of events from $S$ define the ordering $<_\alpha$ by $a <_\alpha b$ if and only if for some $i$ and $j$, $i < j$ and $a = a_i$ and $b = b_j$. We call $\alpha$ a topological sort of $S$ if the order $<_\alpha$ is a total extension of the partial order $<$.

**Proposition 2.1**
The FSA $\mathcal{A}$ accepts exactly the set of topological sorts of $S$ with respect to $<$. Also $\mathcal{A}$ is the smallest FSA which accepts this language.

The number of states in $\mathcal{A}$ is always smaller than the number of nodes required by the tree representation of the topological sorts which is used in ptk. The only exception is when the partial order $<$ is total, then the automaton has the same number of states as the tree. Each of the two FSA in the above examples is the automaton $\mathcal{A}$ for the particular partial order of the example.

As explained in the second example above we can calculate how many total orders there are directly from $\mathcal{A}$. Label the start state with value 1, for any state v label it with the sum of the values of its parent states. The label of the accepting state is the number of possible total extensions. Also the sum of the values of all the states is the number of nodes which would be required to express the total extensions in the form of a tree as is done on ptk.

Consider the extreme case where the partial order is empty and $S$ has $n$ elements. The topological sort tree for this case will have $\mathsf{t}(n)$ nodes where $\mathsf{t}$ is defined by

$\mathsf{t}(n) = 1 + n\mathsf{t}(n-1)$
$\mathsf{t}(1) = 1$

The FSA $\mathcal{A}$ will have $2^n$ states. For example in the case $n = 12$ we have $\mathsf{t}(n) = 823059745$ nodes and $\mathcal{A}$ has 4096 states. In this case the possible number of interleavings is $12! = 479001600$.

## 2.1 Another Example

Consider the partial order $a < c$, $b < c$, $b < d$. In this case $\mathcal{A}$ is this:

$$\begin{tikzcd}
 & \{a,b\} \arrow[dl, "a"'] \arrow[dr, "b"] & & \\
\{b\} \arrow[dr, "b"] & & \{a,d\} \arrow[dl, "a"'] \arrow[dr, "d"] & \\
 & \{c,d\} \arrow[dl, "c"'] \arrow[dr, "d"] & & \{a\} \arrow[dl, "a"'] \\
\{d\} \arrow[dr, "d"] & & \{c\} \arrow[dl, "c"'] & \\
 & \{\} & &
\end{tikzcd}$$

This is a more subtle example than the first two since the graph of the partial order is connected, whereas the first two can be thought of as the disjoint union of a number of total orders.

The initial events in this partial order are $a$ and $b$ which is why the start state is $\{a, b\}$. As an example of how this is generated look at the two states which follow on from the initial state. From the definition of trns we get

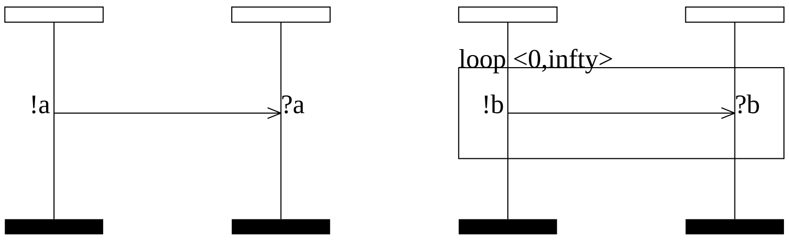
$$\mathsf{trns}\{a, b\} = \{\ \{a, b\} \xrightarrow{a} \{b\},\ \{a, b\} \xrightarrow{b} \{a, d\}\ \}$$

which comes from

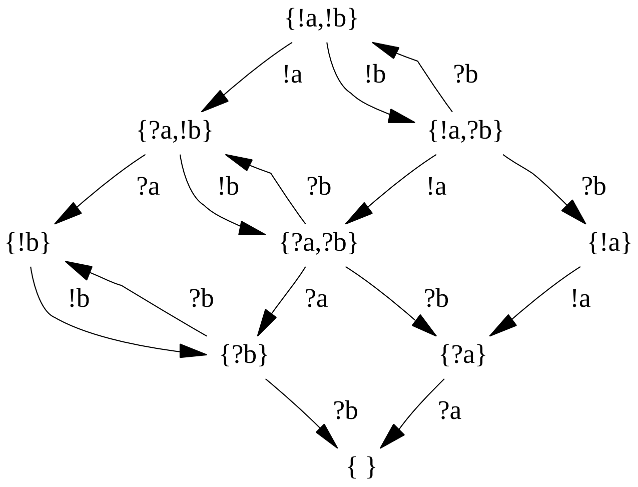
$$\begin{aligned}\mathsf{m}((\{a,b\}-\{a\})\cup\mathsf{n}(a)) &= \mathsf{m}\{c,b\} = \{b\} \text{ and}\\ \mathsf{m}((\{a,b\}-\{b\})\cup\mathsf{n}(b)) &= \mathsf{m}\{a,c,d\} = \{a,d\}\end{aligned}$$

## 3 Loops

MSC can have inline expressions which represent loops. The current algorithm can not handle this case. Consider this example of an extremely simple loop:



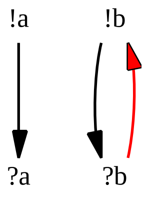
By extending the FSA algorithm to work over a set of relations in stead of a single partial order we can extend the algorithm to work for loop MSC. In this case we would get this FSA:



As before the start state is $\{!a, !b\}$ and the accepting state is $\{\ \}$.

### 3.1 Graph Representation of MSC with Loops

To generate the FSA we need first to represent the MSC as a combination of binary relations. One of these is the partial order which represents the causal relationships between the events in the MSC with the loop inline expressions removed. The other is a binary relation which represents the loop structure.



The black arrows in this graph represent the causal relationship in the underlying basic MSC (with loop removed) of the previous example. The red arrow represents the binary relation given by the loop. In general if an MSC contains $n$ loops their are $n$ loop relation $L_i$, defined by $xL_iy$ whenever $y$ is any event which is minimal with respect to $<$ within the region bounded by the loop, $y < x$, and $x$ is maximal with respect to $<$ within the region bounded by the loop.

Next extend the definitions of functions for basic MSC to the loop case. For a binary relation $R$ on events define

- For an event $a$ the next events according to $R$ is the set
  $\mathsf{n}(R)(a) = \{b \mid aRb \text{ and there is no } c \text{ where } aRc \text{ and } cRb\}$
- For a set of events $X$ the set of minimal events according to $R$ is the set
  $\mathsf{m}(R)(X) = \{a \in X \mid \nexists b \in X \text{ such that } bRa\}$
- For a set of events $U$
  $\mathsf{trns}(R)(U) = \{\, U \xrightarrow{u} \mathsf{m}(R)((U - \{u\}) \cup \mathsf{n}(R)(u)) \mid u \in U\}$
  and for a set $\mathcal{R}$ of binary relations the complete set of transitions is

$$\mathsf{trns}(\mathcal{R})(U) = \bigcup_{R\in\mathcal{R}} \mathsf{trns}(R)(U)$$

## 3.2 The FSA algorithm

The algorithm for generating the FSA for an MSC with loops is a straightforward extension of the earlier algorithm. For an MSC $M$, let $\mathcal{R}(M) = \{<\} \cup \{L_i \mid 1 \le i \le n\}$ where $<$ is the partial order representing the causal relationship of the underlying basic MSC (where the loop constructs are removed, but the underlying events in side the loop are kept), and $L_i$ are the loop binary relations defined above.

```
fun $A$ $(X)$ =
let
  val $S_X = \{\mathsf{m}(<)(X)\}$
  val $T_X = \{\}$
  val nextStates = $S_X$
  val nextTrans = $\{\}$
  fun nx $(U \xrightarrow{a} V) = V$
  fun concat SetOfSets = foldl $\cup$ { } SetOfSets
  val $(S_X, T_X)$ =
                while (not (nextStates = { }))
                    { let
                        val nextTrans = concat (map (trns $\mathcal{R}(M)$) nextStates)
                        val nextStates = map nx nextTrans
                        val $S_X = S_X \cup$ nextStates
                        val $T_X = T_X \cup$ nextTrans
                      in
                        $(S_X, T_X)$
                      end
                    }
in
  $(S_X, T_X)$
end
```

Let $\mathcal{A}$ be the automaton $A(S)$ where $S$ is all the events in $M$, where the set $\mathsf{m}(<)(S)$ is the start state, and $\{\,\}$ is the only accepting state. It is no longer the case that $\mathcal{A}$ accepts exactly the total extensions of $<$, because of the loops. It does accept all the possible interleavings of the events in $M$, and when the names are all distinct it is still the smallest automaton to do so.

**Proposition 3.1**
Let $\mathcal{R} = \{<\} \cup \{L_i \mid 1 \leq i \leq n\}$ be a set of binary relations on $S$ where $<$ is a partial order on $S$. Write $(u \,\mathsf{sib}\, v)$ to denote that there is some $w$ such that $w < u$ and $w < v$, that is $u$ and $v$ are siblings.

Suppose each $L_i$ has the property

$$\forall a : (u\, L_i\, a) \Rightarrow \forall v : (v \,\mathsf{sib}\, u) \Rightarrow (v\, L_i\, a)$$

Then the algorithm to construct $\mathcal{A}$ will always terminate, and the language accepted by $\mathcal{A}$ is the set of interleavings generated by the graph (which are formally defined in the next section).

The loop graph generated by any MSC always satisfies the properties of this proposition, so $\mathcal{A}$ is always finite. The exception being when there are gates present within a loop construct. In this case there are examples of MSC where the interleavings can not be described by a finite state automaton, but must be described by a context free grammar.

## 3.3 Interleavings of a looped Graph

In this section we define formally what an interleaving is for a graph which contains loops. This definition directly relates an interleaving to the structure of the graph without having to refer to unwinding the graph, or reference to moving in and out of loops. Underlying the definition is the idea of viewing the looped graph as a set of production rules in a grammar.

A graph with loops $G$ is a set $\mathcal{R}$ of binary relations over a set of event nodes $S$. For a relation $R$ over $S$ define

- $\mathsf{out}(R)(a) = \{b \mid aRb\}_m$ where $\{\,\}_m$ denotes a multi-set.
- $\mathsf{in}(R)(a) = \{b \mid bRa\}_m$

For a mutli-set $X$ let $\mathsf{map}$ denote the obvious mapping function, so that $\mathsf{map}\ f\ X = \{f(x) \mid x \in X\}_m$.

Let $\alpha = a_0 \cdots a_k$ be a string over $S$. Let $\mathbb{N}_k = \{i \mid 0 \leq i \leq k\}$, note we can think of $\alpha$ as a map $\alpha : \mathbb{N}_k \longrightarrow S$. Define a relation $W$ on $\mathbb{N}_k$ by

$$iWj = (a_i(\bigcup_{R \in \mathcal{R}} R)a_j) \wedge (i \leq j)$$

Define $\alpha$ to be an interleaving of $G$ if there is some relation $W' \subseteq W$ such that

$$\forall i \in \mathbb{N}_k\ \exists\, U, V \in \mathcal{R} :$$

$$\begin{array}{lll} \mathsf{map}\ \alpha\ \mathsf{out}(W')(i) & = \mathsf{out}(U)(a_i) & \text{when } i < k \\ \mathsf{map}\ \alpha\ \mathsf{in}(W')(i) & = \mathsf{in}(V)(a_i) & \text{when } i > 0 \end{array}$$

## 4 Conclusion

In the non-iterating case proposition 2.1 states that $\mathcal{A}$ is the smallest automaton which generates the total extensions of $<$. In ptk we start with a partial order of distinct events, but these events may not have distinct string labels. In this case we dont just want to generate all total extensions of the partial order on the events. We only want to get the total orders which give different interleavings of the labels. In this case it may be possible to construct a FSA automaton which is smaller than $\mathcal{A}$ which only generates the different interleavings of the string labels.

Suppose in the previous example we give each event the string label `msg`. Then there is only one interleaving, namely

```
msg msg msg msg
```

Clearly this can be generated by a FSA with five states, not the eight states of $\mathcal{A}$.

The algorithm for $\mathcal{A}$ generates the states and transitions on the fly so that only the necessary states and transitions are generated. Thus the complexity of the algorithm is linear in the number of states of $\mathcal{A}$. Thinking of the automaton as a compression of the topological sort tree we gain by going from a representation which is almost doubly exponential in the worst case to one which is singly exponential in the worst case. It might also be the case that we can perform useful analysis of the automaton with respect to feature interaction.

Proposition 3.1 states that the extended algorithm always terminates on loop graphs which are generated by MSC provided the loops do not contain gates (a construction which allows arbitrary message passing between parts of an MSC). In such cases there is no FSA which can describe the interleavings of the MSC. Thus the algorithm works for those cases when a FSA exists. Moreover it is again the minimal such automata when the events have distinct names.