T-79.231 Parallel and Distributed Digital Systems

Reachability analysis

Marko Mäkelä

July 16, 2002

Reachability analysis

Reachability analysis refers to the procedure of

- checking whether a formally described system may reach
  - a “bad” state violating the asserted safety properties or
  - a loop of events that is against the given liveness conditions
- and visualising a violating execution if one exists.

Usually, reachability analysis requires exploring all reachable states in the system. Sometimes it is possible to make use of the given conditions and forget about some states.

The basic solution: computing the reachability graph

```
REACHABILITY-GRAPH(⟨S, T, F, W, M0⟩)
G : ⟨V, E, v0⟩ ← ⟨{M0}, ∅, M0⟩;
Markings : stack ← ⟨M0⟩;
while Markings ≠ ⟨⟩
do M ← Markings.pop();
   for t ∈ enabled(M)
   do M′ ← fire(M, t);
      if M′ ∉ V
        then V ← V ∪ {M′}
             Markings.push(M′);
      E ← E ∪ {⟨M, t, M′⟩};
return G;
```

The algorithm makes use of two functions:

- enabled$(M) := \{t \mid M[t\rangle\}$
- fire$(M,t) := [M\rangle t$

When the search stack is replaced with a search queue, the graph is traversed breadth first instead of depth first. Breadth first search finds the shortest transition path from the initial marking to an erroneous marking. Model checking liveness requires depth first search, as some loops need to be detected.

Implementing reachability analysis

The worst bottle-necks of the preceding algorithm are:

- the loop in the lines 5–10
- the lines 5 and 6: constructing the set of successor states $\bigcup_{t \in \mathsf{enabled}(M)} \{\mathsf{fire}(M,t)\}$
- the lines 7 and 8: searching and adding states

The line 10 is rather harmless, since the arcs can be written to a sequential file, unless also "reverse" arcs are to be written, indexed by the target state. The arcs can omitted as well and recomputed when generating a counterexample.

The search queue or stack $Markings$ may contain either states or state numbers, which saves memory but requires that in line 4, the state M is retrieved from a "database" $DB : \mathbb{N} \to V$.

Representing the set of states V

The set V can be represented in two fundamentally different ways. While computing the set $\bigcup_{t \in \mathsf{enabled}(M)} \{\mathsf{fire}(M,t)\}$, the elements of the set $M \in V$ are best kept in an "expanded form" so that the expressions of the modelling language can be easily evaluated.

The data structure for the found states V must support the following functions:

- $M \in V$: has the state M been found already?
- $V \leftarrow V \cup \{M\}$: add M to the set of reachable states (and assign an index n_M to it)
- (Fetch a state by index: $DB(n_M) \mapsto M$)

The functions in parentheses are only needed if the search stack or queue keeps state numbers. If V is expressed as a dynamic table DB where each state is stored separately, it pays off to *compact* the states to bit strings. For instance, if it is known that the $n = 5$ clients of a system may be in $m = 3$ different states, the pairs $\langle n, m \rangle$ do not need to occupy $32 + 32$ bits or even $8 + 8$ bits, but only $\log_2 \lceil nm \rceil = \log_2 \lceil 15 \rceil = 4$ bits.

Symbolic representations of the state set (1/2)

The state set V can also be represented as a single “lump” that cannot be iterated but where states can be added or checked for existence.

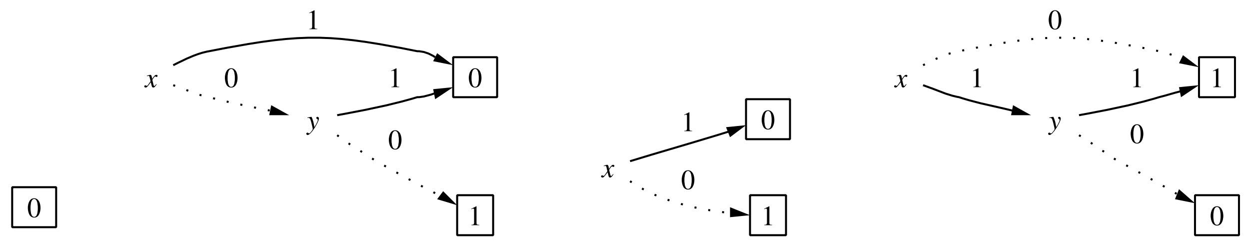
One way of implementing this kind of a structure is to build up a directed graph, interpreted as an automaton that accepts exactly those “words” (states) that belong to the set.

Whenever a state is added to the set, the pointer jungle of the graph is modified here and there. In practice, this kind of a *symbolic* data structure must be kept in the main memory.

Symbolic data structures are very closely related to the operations and data types of the language. When each state is stored separately, the operations leading from one state to another are irrelevant; it must merely be possible to encode each state to a bit string.

Symbolic representations of the state set (2/2)

One of the best known symbolic data structures is BDD *(Binary Decision Diagram)* that was designed with Boolean circuits in mind. BDDs are directed acyclic graphs (folded trees) containing at most two leaf nodes, 0 and 1, and inner nodes corresponding to the variables of the system. Each variable node has two outgoing arcs, one for each value the variable may assume. A BDD represents a formula that holds for exactly those assignments that have been added to it. For instance, let us store the states $\langle x \mapsto \bot, y \mapsto \bot \rangle$, $\langle x \mapsto \bot, y \mapsto \top \rangle$ and $\langle x \mapsto \top, y \mapsto \top \rangle$ in said order to an initially empty set:



$\bot$ $\neg x \land \neg y$ $\neg x$ $\neg x \lor y$

Adding a state may enlarge or shrink the representation of the set. This kind of methods are very sensitive of the order in which variables are represented and states are added.

Marko Mäkelä

Shrinking the set of reachable states

Traditional representations of the state set often are the only meaningful choice if the system being explored has many high-level operations. Then the large number of states and transitions is a problem. What about trading some processing time for space?

Exploiting the property being checked makes it possible to avoid exploring all possible behaviours of the system. The major reduction methods are:

- identifying states:
  - detecting symmetries and mapping states to each other with permutations
  - initialising dead variables to default values
  - slicing (before reachability analysis, eliminate those variables and transitions that do not affect the property being checked)
- partial order reduction methods: reducing the branching degree

Symmetries (1/2)

For a place/transition net $\langle S, T, F, W \rangle$, a *symmetry* $\sigma : (S \to S) \cup (T \to T)$ is an *automorphism* (a self-isomorphism of the net) that respects the arc weights:

$$W(a,b) = W(\sigma(a), \sigma(b)).$$

The symmetry σ of the net can be augmented to a symmetry $\sigma : (S \to \mathbb{N}) \to (S \to \mathbb{N})$ of markings $M : S \to \mathbb{N}$ by defining

$$(\sigma(M))(\sigma(s)) = M(s).$$

It follows that $M' = [M\rangle t$ if and only if $\sigma(M') = [\sigma(M)\rangle\sigma(t)$.

Symmetries (2/2)

A set of symmetries Σ is a *symmetry group* if and only if it is reflexive, symmetric and transitive:

1. there is $id \in \Sigma$ such that $id(M) = M$,
2. if $\sigma \in \Sigma$ then $\sigma^{-1} \in \Sigma$, and
3. if $\sigma \in \Sigma$ and $\sigma' \in \Sigma$ then $(\sigma' \circ \sigma) \in \Sigma$.

The symmetry group defines an equivalence relation for markings (and places and transitions):

$$M \equiv M' :\Leftrightarrow (\exists \sigma \in \Sigma : M = \sigma(M')).$$

A marking M is *symmetric* if and only if

$$\forall \sigma \in \Sigma : M = \sigma(M).$$

The symmetry reduction method

SYMMETRIC$(\langle S, T, F, W, M_0, \Sigma \rangle)$

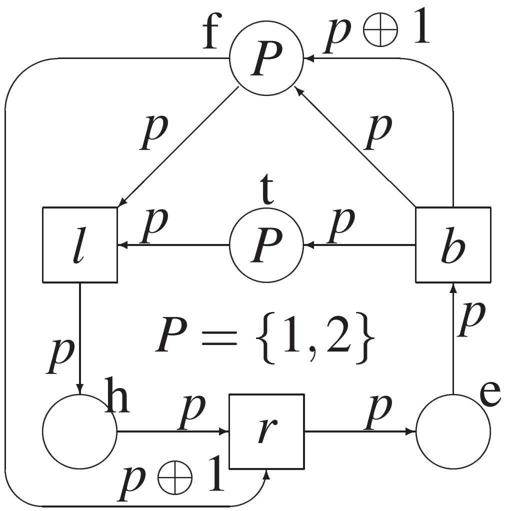
$G : \langle V, E, v_0 \rangle \leftarrow \langle \{M_0\}, \emptyset, M_0 \rangle$;
$\mathit{Markings} : \text{stack} \leftarrow \langle M_0 \rangle$;
while $\mathit{Markings} \neq \langle \rangle$
do $M \leftarrow \mathit{Markings}.\mathsf{pop}()$;
  for $t \in \mathsf{enabled}(M)$
  do $M' \leftarrow \mathsf{fire}(M, t)$;
    if $\exists M'' \in V : \exists \sigma \in \Sigma : M'' = \sigma(M')$
     then $E \leftarrow E \cup \{\langle M, t, M'' \rangle\}$
     else if $M' \notin V$
       then $V \leftarrow V \cup \{M'\}$
         $\mathit{Markings}.\mathsf{push}(M')$
       $E \leftarrow E \cup \{\langle M, t, M' \rangle\}$
return G;

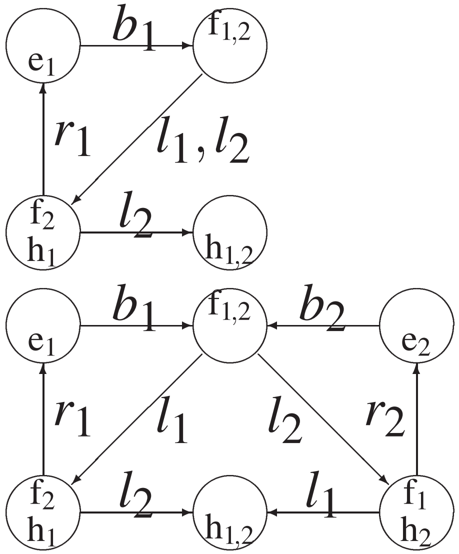
If M_0 is symmetric, the reachable markings and deadlocks are preserved. The liveness of transitions or the truth values of LTL formulae are not always preserved.

The worst bottle-necks are finding the symmetry groups and detecting symmetric states.

The method can be generalised to other modelling languages, such as high-level nets. It may even save time, as entire subtrees of the state space may remain unexplored.

The symmetry reduction method: an example





Clearly, the reachability graph of the dining philosopher system is symmetric. One symmetry rotates the philosophers and forks:

$$\sigma = \{\mathrm{f}_1 \mapsto \mathrm{f}_2, \mathrm{f}_2 \mapsto \mathrm{f}_1, \mathrm{t}_1 \mapsto \mathrm{t}_2, \mathrm{t}_2 \mapsto \mathrm{t}_1, \mathrm{h}_1 \mapsto \mathrm{h}_2, \mathrm{h}_2 \mapsto \mathrm{h}_1, \mathrm{e}_1 \mapsto \mathrm{e}_2, \mathrm{e}_2 \mapsto \mathrm{e}_1\}.$$

As a matter of fact, this σ *generates* a symmetry group: for instance, $id = \sigma \circ \sigma$ and $\sigma^{-1} = \sigma$.

A partial reduction method: stubborn sets

The stubborn set method tries to eliminate some of the enabled transitions when computing the set of successor states, which makes the reachability graph branch less and become smaller. The method preserves all deadlocks. One variant of it also preserves the truth values of all LTL formulae not containing the connective $\bigcirc$.

The method is based on splitting the transitions T into two sets T_s and $T \setminus T_s$ such that firing any transition in T_s does not affect the enabledness of the transitions in $T \setminus T_s$.

In the following commuting diagram, let there be $t \in T_s$ and a transition sequence $\tau = (T \setminus T_s)^*$. For the properties that hold in M''' it does not matter whether the marking is reached via the path $t\tau$ or the path τt.

$$\begin{tikzcd} M \arrow[r, "\tau"] \arrow[d, "t"] & M' \arrow[d, "t"] \\ M'' \arrow[r, "\tau"] & M''' \end{tikzcd}$$

Stubborn sets (1/4)

Let there be a place/transition system $\langle S, T, F, W \rangle$ with a marking M. Then $T_s \subseteq T$ is *stubborn* in M if and only if it contains M-enabled transitions and $\forall t \in T_s$:

1. If $M[t\rangle$ then $({}^\bullet t)^\bullet \subseteq T_s$.
2. If $\neg M[t\rangle$ then $\exists s \in {}^\bullet t : M(s) < W(s,t) \wedge {}^\bullet s \subseteq T_s$.

The set of all transitions T is trivially stubborn.

It is best to construct smallest possible stubborn sets, since then there will be the least number of branches in the reachability graph.

Stubborn sets (2/4)

One stubborn set T_s in M can (inefficiently) be computed as follows:

1. Choose an M-enabled transition $t \in T$: $M[t\rangle$. Let $T_s = \{t\}$.

2. Choose a transition $t' \in T_s$ (that has not been considered yet):

   (a) $M[t'\rangle$: The definition will hold by setting $T_s \leftarrow T_s \cup ({}^\bullet t')^\bullet$. Return to step 2.

   (b) $\neg M[t'\rangle$: According to the definition, find a "culprit" place $s \in {}^\bullet t'$ that prevents t' from firing, and let $T_s = T_s \cup {}^\bullet s$. Return to step 2.

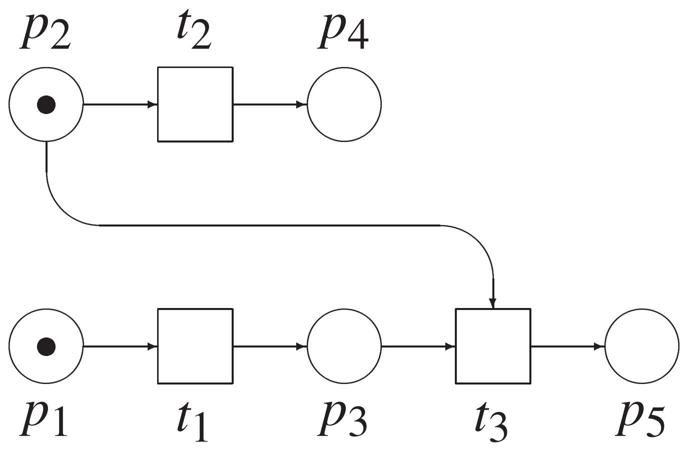
The algorithm terminates when the set T_s does not change during an iteration.

Stubborn sets (3/4)

STUBBORN($\langle S,T,F,W,M_0\rangle$)

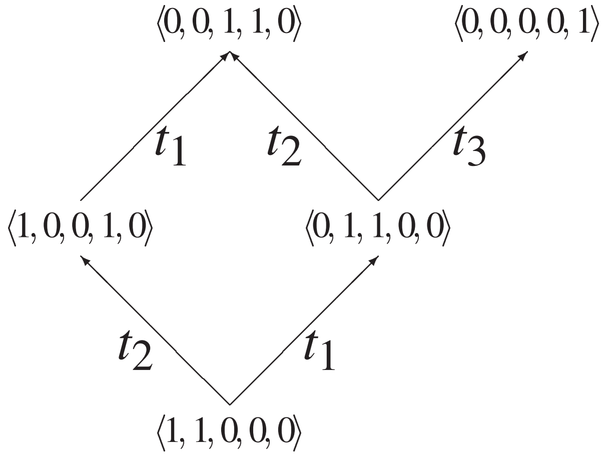
$G : \langle V, E, v_0\rangle \leftarrow \langle \{M_0\}, \emptyset, M_0\rangle$;
$Markings$: stack $\leftarrow \langle M_0 \rangle$;
while $Markings \neq \langle\rangle$
do $M \leftarrow Markings$.pop();
  for $t \in$ stubborn(M)
  do $M' \leftarrow$ fire(M,t);
    if $M' \notin V$
      then $V \leftarrow V \cup \{M'\}$
        $Markings$.push(M');
    $E \leftarrow E \cup \{\langle M, t, M'\rangle\}$;
return G;

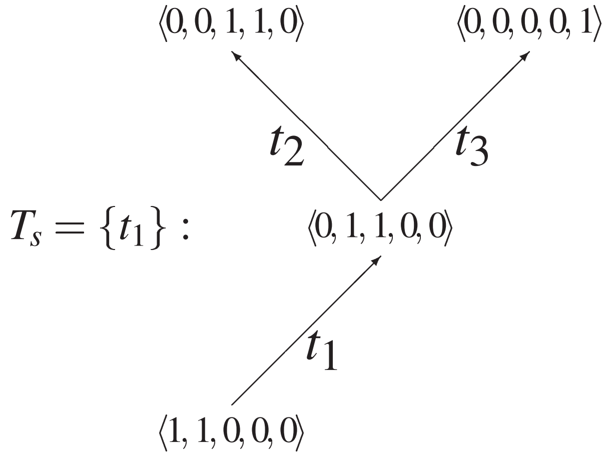
Let us examine the following system:



If initially $T_s \supseteq \{t_2\}$, $({}^\bullet t_2)^\bullet = \{t_3\}$ and ${}^\bullet p_3 = \{t_1\}$ must be added to the set, because $p_3 = {}^\bullet t_3$. We obtain $T_s = T$.
If the initial choice is $T_s = \{t_1\}$, the set will be stubborn as such.

Stubborn sets (4/4)





The magnitude of a stubborn set greatly depends on which enabled transition is chosen first and how the "culprit" places s are chosen.

The smallest possible stubborn set can be obtained with the *deletion algorithm*. It starts from the set $T_s = T$. A transition is removed, and further transitions will be removed until the set is stubborn again. The *insertion algorithm* described earlier can be improved by utilising Tarjan's algorithm for computing strongly connected components.

Translating modelling formalisms

It appears easy to implement reachability analysis for any computing system, but in practice, many systems do not allow their state to be stored efficiently and restored later. It is best not to reinvent the wheel and to use dedicated reachability analysis tools.

If the systems being explored have already been described in some computer-interpreted language and if there are many such descriptions, it does not make sense to translate them by hand. The difficulty of writing an automatic translator depends on how complex the source language is. Often the translator can be restricted to a subset of the language.

When translating descriptions, it is useful to simplify them and omit everything that is irrelevant for checking the desired properties. Some constructs can be expressed more compactly with Petri nets than with a programming language. The quality of the translation has great impact on the size of the reachability graph.

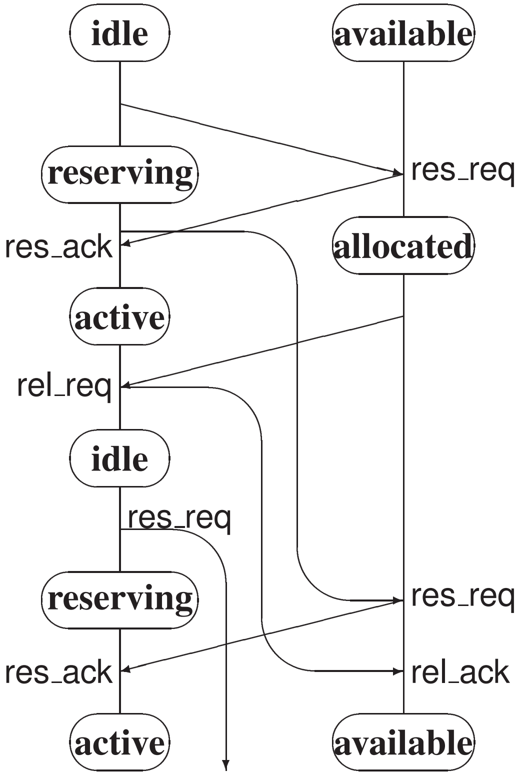
Representing counterexamples (1/2)

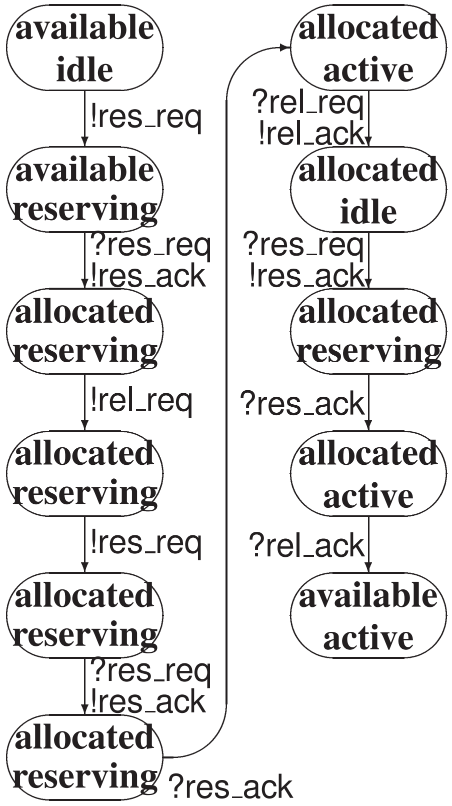
When a tool detects that the system violates its requirements, it must somehow report it to the user. It would be frustrating for the user to know that there is an error somewhere in the system. It is much more useful to represent a *counterexample*, a violating execution.

An execution can be represented as a sequence of states and events where the states are labelled with the values of variables and the events are labelled with the names of the transitions. If the input of the reachability analyser has been translated from some other description, the labels and the notation would better follow the original conventions. In other words, also the counterexamples must be translated!

Representing the executions with such chains is not always natural. Sometimes it is more convenient to explore the execution in a debugger-like tool, in an animation where one source code line or structure is highlighted at a time. Message sequence charts are suitable for representing executions of message-passing systems.

Representing counterexamples (2/2)





A message sequence chart conveys the error better than a transition sequence. A chart represents a higher-level language, since it corresponds to several transition sequences.