Case 5:
An efficient test whether a reactive system is deterministic

Antti Valmari
Tampere University of Technology

File \LaTeX ed November 19, 2009
1 Background

1.1 Labelled transition systems

In the theory of finite automata the object of the study is the set of the strings that the automaton accepts.

A finite automaton is \((Q, \Sigma, \Delta, \hat{q}, F)\)

- \(Q\) and \(\Sigma\) are finite sets
- \(\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q\)
- \(\hat{q} \in Q\)
- \(F \subseteq Q\)

\(q = a_1a_2\cdots a_n \Rightarrow q'\) means that there is a path from the state \(q\) to the state \(q'\) such that the sequence of the elements of \(\Sigma\) along the path is \(a_1a_2\cdots a_n\).

- that is, there are states \(q_0, q_1, \ldots, q_m\) and transitions \((q_0, b_1, q_1), (q_1, b_2, q_2), \ldots, (q_{m-1}, b_m, q_m)\) (where \(m \geq n\)) so that \(q = q_0, q_m = q',\) and when the symbols \(\varepsilon\) are left out of the sequence \(b_0b_1\cdots b_m,\) then \(a_1a_2\cdots a_n\) remains
The language \textit{accepted} by the automaton is \( \{ \sigma \in \Sigma^* \mid \exists q \in F : \hat{q} = \sigma \Rightarrow q \} \)

In the study of the behaviour of reactive and concurrent systems, a structure is used that is like finite automata with the following differences:

- there is no set \( F \) of final states (all states are treated as final states)
- more than just the produced string is of interest of the path taken
- the set of states \( Q \) and the alphabet \( \Sigma \) need not be finite
- usually \( \tau \) is used as the symbol of an invisible step
- symbols of (sets of) states tend to begin with \( S \) or \( s \)
- (there may be many initial states)

A \textit{labelled transition system (LTS)} is the 4-tuple \((S, \Sigma, \Delta, \hat{s})\), where

- \( \Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S \)
- \( \hat{s} \in S \)
Now

- \( s = \varepsilon \Rightarrow s' \) means that there is a path from \( s \) to \( s' \) that consists of zero or more \( \tau \)-transitions
- \( s = a \Rightarrow s' \) means that there is a path from \( s \) to \( s' \) that starts with zero or more \( \tau \)-transitions, then contains an \( a \)-transition, and finally zero or more \( \tau \)-transitions
- similarly \( s = a_1 a_2 \cdots a_n \Rightarrow s' \) allows \( \tau \)-transitions freely anywhere in the path, and other than \( \tau \)-transitions have the labels \( a_1, a_2, \ldots, a_n \) in this order

The sequence of the non-\( \tau \) labels of a path that starts at the initial state is a trace.

The set of the traces of an LTS corresponds to the language of a finite automaton

- all states are final states

\( \Rightarrow \) every prefix of a trace is a trace

- e.g., if \( abcaab \) is a trace, then also each of the following is a trace: \( abca, abc, ab, a, \) and \( \varepsilon \)

In the context of labelled transition systems, the elements of \( \Sigma \) are called visible actions.

\( \tau \) is the invisible action.
1.2 On the behaviour of labelled transition systems

The following differences in the taken path are of interest, among others (depending on the research sub-community)

Opinions vary regarding which differences are essential and which are not

- in the background of varying opinions are both “philosophical” views and requirements caused by developing a well-working mathematical theory
- the topic is extensive
- fortunately, it is not essential for our current case
However, one term is useful in the sequel: *divergence*

- a never-ending sequence (or a cycle) of transitions that are labelled with $\tau$
  - is denoted with $s \xrightarrow{\tau^\omega}$
  - $\omega$ is the infinity in discrete mathematics and theoretical computer science
- a *divergence trace* is a trace that leads to a divergence
1.3 Determinism of labelled transition systems

One may, of course, define determinism like with finite automata

- $\forall s : \forall a : \forall s_1 : \forall s_2 ; (s, a, s_1) \in \Delta \land (s, a, s_2) \in \Delta : s_1 = s_2 \land a \neq \tau$

- we call this structural determinism

Unfortunately, such a notion of determinism is not particularly useful in the context of reactive and concurrent systems

- the banishment of invisible, that is, $\tau$-transitions is too strong a requirement

- the parallel composition of two deterministic components is not necessarily deterministic, even in the absence of $\tau$-transitions

In the year 2006, a well-working notion of determinism was published

- a notion that does not allow divergences but is otherwise essentially the same had been published before a couple of times formulated in different ways
An idea: after each possible way of executing a trace, the possibilities of immediate continuation must be the same in the following sense:

- for each \( a \in \Sigma \), if at least one of them allows \( a \) as the next visible action, then each one of them allows

- if at least one of them may diverge, then each one of them may

In other words,

\[
\forall s_1 \in S : \forall s_2 \in S : \forall \sigma \in \Sigma^* : \hat{s} = \sigma \Rightarrow s_1 \land \hat{s} = \sigma \Rightarrow s_2 : (1) \land (2),
\]

where

1. \( \forall a \in \Sigma : s_1 = a \Rightarrow \rightarrow s_2 = a \Rightarrow \)

2. \( s_1 \tau^\omega \rightarrow \rightarrow s_2 \tau^\omega \rightarrow \)

*Operational determinism*
A comment on the symbols for implication and equivalence

- there are—at least in principle—two different implications and equivalences in logic
  - operators in propositional logic that produce truth values, similarly with $\land$ and $\lor$
  - that express relations between claims, similarly with “therefore” in natural language

- the difference is similar with in arithmetic: $+$ and $\cdot$ produce numbers, $\geq$ and $=$ compare numbers

- the former are denoted with, e.g., $\to$ and $\leftrightarrow$ or $\Rightarrow$ and $\Leftrightarrow$

- the latter are denoted with, e.g., $\Rightarrow$ and $\Leftrightarrow$ or $\therefore$ and $\equiv$

- the choice may be affected by if the symbols already have another meanings
  - now we have rather similar-looking $-a\to$ and $=a\Rightarrow$

- the difference is often insignificant and is often ignored

- we try to use $\to$ and $\leftrightarrow$ in formulas, and $\Rightarrow$ and $\Leftrightarrow$ between formulas

- e.g., $(T \to F) \to F \iff F \to F \iff T$, but $(T \Rightarrow F) \Rightarrow F$ is misuse of notation, and $T \Rightarrow F \Rightarrow F$ does not hold
Advantages of operational determinism

• matches the intuitive idea of determinism, when invisible actions are not directly observed

• is preserved by parallel composition

• in the case of operationally deterministic systems, the differences between various schools in the notions of “equivalent behaviour” mostly disappear
  ⇒ it is appropriate to say that the theory of reactivity is largely theory of nondeterminism
1.4 The goal

An algorithm that recognizes determinism has applications in security

- nondeterminism means irregular leakage of information
- probably that information was not intended for delivery

An efficient algorithm has potential applications in the processing of labelled transition systems in general

- no need to take into account special properties of complicated equivalence notions
- it will turn out that it is cheap to make an equivalent structurally deterministic labelled transition system from an operationally deterministic one

⇒ many things can be made with cheap algorithms for deterministic finite automata

Beforehand an algorithm was known that is quadratic in the number of states (for the definition that bans divergences)

We wanted a better algorithm!
2 The basic idea of the algorithm

2.1 The notion of equivalent states

Operational determinism requires that all states that can be reached with the same trace have certain properties in common

- for every $a \in \Sigma$ it holds that if $s_1 = a \Rightarrow$, then $s_2 = a \Rightarrow$
- if $s_1 - \tau^\omega \rightarrow$, then $s_2 - \tau^\omega \rightarrow$

The requirement may extent to also states that cannot be reached with the same trace

- let $\hat{s} = \sigma \Rightarrow s_1$, $\hat{s} = \sigma \Rightarrow s_2$, $\hat{s} = \rho \Rightarrow s_2$, and $\hat{s} = \rho \Rightarrow s_3$
- if $s_1 = a \Rightarrow$, then $s_2 = a \Rightarrow$ because of $\sigma$
- since $s_2 = a \Rightarrow$, we have $s_3 = a \Rightarrow$ because of $\rho$
- therefore, $s_1 = a \Rightarrow \Rightarrow s_3 = a \Rightarrow$, even if $s_1$ and $s_3$ did not have a trace in common
- similarly $s_1 - \tau^\omega \rightarrow \Rightarrow s_3 - \tau^\omega \rightarrow$
⇒ It is worthwhile to introduce a more general relation “∼” between states that keeps track of the equivalence requirements.

Operational determinism does not state any requirements to states that cannot be reached at all.

- posing requirements to such states may lead to a wrong result.

⇒ the requirements must be in one way or another restricted to reachable states.

The restriction can be presented in the definition of the “∼” relation.

Another possibility is to leave unreachable states out beforehand or afterwards.

- unreachable states do not affect the behaviour.

⇒ leaving them out is allowed.

- “beforehand” = discussion is restricted to the reachable part of the LTS.

- “afterwards” = “∼” is only required between the reachable states.

In this case leaving them out afterwards is simple and perhaps the simplest.

- a challenge: will you see in the sequel where and how it happens?
2.2 Properties of equivalent states

We now only consider reachable states for the reason told above.

All the time let $a \in \Sigma$

These requirements have already been found necessary:

- $s_1 \sim s_2 \land s_1 = a \Rightarrow s_2 = a \Rightarrow$

- $s_1 \sim s_2 \land s_1 - \tau^0 \rightarrow \Rightarrow s_2 - \tau^0 \rightarrow$

If $s_1 - \tau \rightarrow s_2$, then $s_2$ can be reached with every trace with which $s_1$ can be reached.

- because we only consider reachable states, $s_1$ can be reached with some trace

$\Rightarrow$ they can be reached with the same trace

$\Rightarrow$ we require $s_1 - \tau \rightarrow s_2 \Rightarrow s_1 \sim s_2$

If $s_1$ and $s_2$ can be reached with the same trace $\sigma$, and if $s_1 - a \rightarrow s'_1$ and $s_2 - a \rightarrow s'_2$, then $s'_1$ and $s'_2$ can be reached with the same trace $\sigma a$

$\Rightarrow$ we require $s_1 \sim s_2 \land s_1 - a \rightarrow s'_1 \land s_2 - a \rightarrow s'_2 \Rightarrow s'_1 \sim s'_2$
To make the computations go through, we must also state explicitly the natural requirement that “∼” is an *equivalence*, that is, it must be

- **reflexive:** \( \forall s : s \sim s \)
- **transitive:** \( \forall s_1 : \forall s_2 : \forall s_3 : s_1 \sim s_2 \land s_2 \sim s_3 \rightarrow s_1 \sim s_3 \)
- **symmetric:** \( \forall s_1 : \forall s_2 : s_1 \sim s_2 \rightarrow s_2 \sim s_1 \)

We call a relation that satisfies these requirements a D-relation

- “D” for “determinism”

A relation “∼” \( \subseteq S \times S \) is a *D-relation*, if and only if it is an equivalence, and for every \( s_1 \in S, s_2 \in S, s'_1 \in S, s'_2 \in S, \) and \( a \in \Sigma \) it holds

- **D1** \( s_1 \sim s_2 \land s_1 =a \Rightarrow s_2 =a \Rightarrow \)
- **D2** \( s_1 \sim s_2 \land s_1 -\tau^\omega \rightarrow s_2 -\tau^\omega \rightarrow \)
- **D3** \( s_1 \sim s_2 \land s_1 -a \rightarrow s'_1 \land s_2 -a \rightarrow s'_2 \Rightarrow s'_1 \sim s'_2 \)
- **D4** \( s_1 -\tau \rightarrow s_2 \Rightarrow s_1 \sim s_2 \)
Finding and justifying this kind of concepts is not as easy in reality as our presentation may make it seem

- finding right conditions is demanding and takes time and trials
- in particular, how does one know that all the necessary conditions have been included?
- the conditions are okay if the concept facilitates proving the algorithm correct

**Theorem** An LTS is operationally deterministic, if and only if its reachable states have a D-relation

**Proof** ⇐

- if a D-relation “∼” exists, then \( \hat{s} \sim \hat{s} \) because “∼” is an equivalence
- given D4, D3, and that “∼” is an equivalence, induction over the length of the trace yields that \( \forall s_1 : \forall s_2 : \forall \sigma : \hat{s} =\sigma \Rightarrow s_1 \land \hat{s} =\sigma \Rightarrow s_2 \Rightarrow s_1 \sim s_2 \)
- from this D1 and D2 guarantee that the LTS is operationally deterministic
The proof in the opposite direction is somewhat more difficult

- it is based on demonstrating that the following relation is a D-relation:
  \[ s \sim s' \iff \exists s_0, \ldots, s_n : \exists \sigma_1, \ldots, \sigma_n : \\
  s = s_0 \land s' = s_n \land \forall i ; 1 \leq i \leq n : \hat{s} = \sigma_i \Rightarrow s_{i-1} \land \hat{s} = \sigma_i \Rightarrow s_i \]

- D1 and D2 follow by induction on \( n \) because the LTS is operationally deterministic
  - we already saw the reasoning on screen 11

- To prove D3, let \( \hat{s} = \sigma_i \Rightarrow s_{i-1} \) and \( \hat{s} = \sigma_i \Rightarrow s_i \) for \( 1 \leq i \leq n \), and let \( s_0 \xrightarrow{a} s'_0 \) and \( s_n \xrightarrow{a} s'_n \)
  - we have D3-\( s_1 = s_0 \), D3-\( s_2 = s_n \), D3-\( s'_1 = s'_0 \), and D3-\( s'_2 = s'_n \)
  - we have to prove \( s'_0 \sim s'_n \)
  - because \( s_0 \xrightarrow{a} s'_0 \), we have \( s_0 = a \Rightarrow s'_0 \)
  - by operational determinism, there are \( s'_1, \ldots, s'_{n-1} \) such that \( s_1 = a \Rightarrow s'_1, \ldots, \)
    \( s_{n-1} = a \Rightarrow s'_{n-1} \)
  - we have \( \hat{s} = \sigma_i a \Rightarrow s'_{i-1} \) and \( \hat{s} = \sigma_i a \Rightarrow s'_i \) for \( 1 \leq i \leq n \)
  - \( s'_0 \sim s'_n \)

- D4 is obvious, we may choose \( n = 1 \)
2.3 From a D-relation towards an algorithm

An algorithm that is directly based on the definition of operational determinism would require recognizing and comparing pairs (or bigger sets) of states that can be reached with the same trace

- comparison is pretty easy: \( \forall a \in \Sigma : s_1 =a \Rightarrow s_2 =a \Rightarrow \) and \( s_1 -\tau_0 \rightarrow \leftrightarrow s_2 -\tau_0 \rightarrow \)
  - “\( =a \Rightarrow \)” requires a graph search algorithm, however
  - also “\( -\tau_0 \rightarrow \)” may seem to require a graph search algorithm, but the result may be cheaply computed for each state beforehand and stored with the state
  - why is “\( -\tau_0 \rightarrow \)” different in this respect from “\( =a \Rightarrow \)”?

- it is, however, difficult to see which states must be compared to each other

- e.g., the determinization algorithm of finite automata recognizes the sets of states that can be reached with the same trace, but uses an exponential structure for that
An algorithm that is directly based on the notion of D-relation needs to

- compare states: D1 and D2, like before
- include all pairs \((s_1, s_2)\) for which D4 holds, to the relation
- grow the relation as required by D3
- take care that the relation is an equivalence

The properties D3 and D4 can be checked locally

- need not even traverse the “\(\equiv\Rightarrow\)” relation
  \(\Rightarrow\) does not seem hopeless

There are no mutually exclusive choices in building the relation

  \(\Rightarrow\) a greedy algorithm suffices

Textbooks present an efficient representation for an equivalence that is built one part at a time
The notion of D-relation seems to be much closer to an efficient algorithm than the notion of operational determinism.

The basic idea of the algorithm we will develop is to gradually build a D-relation between the reachable states until it is ready or it becomes certain that it does not exist.
3 The processing of the equivalence

3.1 Recall: equivalence and partition

Processing an equivalence in a computer may seem difficult

- when \( s_1 \sim s_2 \) starts to hold, at the same time \( s'_1 \sim s'_2 \) must start to hold for every \( s'_1 \) and \( s'_2 \) for which \( s'_1 \sim s_1 \) and \( s_2 \sim s'_2 \) held

- what does it cost to take care of this after every change?

Fortunately, there is another point of view that takes the problem away

Every equivalence “\( \sim \)” on the set \( A \) corresponds to a collection \( \{A_1, A_2, \ldots, A_k\} \) of pairwise disjoint non-empty sets that together cover all elements of \( A \); and vice versa

- a partition
“Pairwise disjoint” means that no two different sets in the collection have elements in common

- that is, $\forall i : \forall j ; 1 \leq i < j \leq k : A_i \cap A_j = \emptyset$

- holds because of symmetry and transitivity

“Cover all elements of $A$” means that every element of $A$ belongs to some set $A_i$

- that is, $A_1 \cup A_2 \cup \cdots \cup A_k = A$

- holds because of reflexivity

$s_1$ and $s_2$ belong to the same set if and only if $s_1 \sim s_2$

The set $A_i$ that contains $x$

- is called the *equivalence class* of $x$

- is often denoted with $[x]$
That is,

- \([\lfloor x \rfloor] = \{y \mid x \sim y\} = \) the equivalence class of \(x\)
- the partition is obtained from the equivalence as the collection of sets
  \(\{\lfloor x \rfloor \mid x \in A\}\)
- the equivalence is obtained from the partition with the formula
  \(x \sim y \iff \exists i; 1 \leq i \leq k : x \in A_i \land y \in A_i\)
  or with either of the formulas
  \(x \sim y \iff x \in \lfloor y \rfloor\)
  \(x \sim y \iff \lfloor x \rfloor = \lfloor y \rfloor\)

Processing a partition in a computer does not seem extremely difficult
3.2 The representation of a partition as an efficient data structure

Our task is made easier by the fact that the relation is only grown, not made smaller

- that is, new $s_1 \sim s_2$ start to hold, but old do not cease to hold
- in the language of partitions: sets in the partition are joined but not split

⇒ The following operations, expressed in the language of partitions, suffice:
  - finding out whether two elements belong to the same set
  - creating a set that consists of a single element
  - joining two sets

Textbooks know a very efficient data structure that is ideal for this purpose

  - *disjoint-set structure* or *disjoint-set forest*

Every element has a record that contains, in addition to the content of the element,

- a link to another element record or to nowhere
- a number that will be used as an “upper approximation to the height”, as will be described later
The links have been set so that following them eventually leads to an element whose link points to nowhere

- a *representative*

- (in the CLR book the link of the representative does not point to nowhere, but back to the representative)

Two elements belong to the same set (that is, $s_1 \sim s_2$), if and only if they have the same representative

$\Rightarrow$ finding the representative is a basic operation that is used in the implementation of other operations
Two sets are joined by re-directing the link of the representative of one set to point to the representative of the other set.

The operation of the data structure is sped up with two tricks:

*Path compression*: always when a representative of an element has been found, the links of that element and all the elements that were met along the path are re-directed to point directly to the representative.

- constant time additional cost to the finding of the representative
- reduces later finding work significantly by making the paths shorter
Union by rank: when joining sets, the original representative whose upper approximation to the height is bigger is chosen as the common representative

- if both original representatives have the same upper approximation to the height, either one may be chosen as the common representative
- in that case, the new upper approximation to the height is the old one incremented by one
- in the opposite case, the new upper approximation to the height is the same as the old one
- this trick, too, shortens paths
- if path compression were not used, the upper approximation to the height would be the precise height
Let $n$ be the number of times a singleton set is created (that is, the total number of elements) and $m$ the total number of operations

- obviously $m \geq n$
- two sets may be joined at most $n - 1$ times

Without speed-up tricks all elements could join into one chain

$\Rightarrow$ total execution time may be $\Theta(mn)$

The use of union by rank reduces the worst case run time to the order of $\Theta(m \log n)$

The effect of path compression alone to the asymptotic running time is difficult to express

Together both tricks improve the asymptotic running time to $O(m\alpha(m,n))$, where $\alpha$ is a function that grows very slowly indeed

- in practice as good as linear running time
If many states gather into the same equivalence class, comparing each pair of them (to check D2, for instance) requires a quadratic number of comparisons:

- fortunately, it is not necessary to compare each pair!

D2 requires that all states in each equivalence class agree on whether $s \xrightarrow{\omega} \tau$ holds:

- it suffices to ensure that each state is combined with a chain of comparisons to each other state in the same equivalence class
- if two states disagree, then somewhere in the chain that combines them are two adjacent states (= states that are compared against each other) that disagree
- immediately when such a disagreement is found, it is certain that there is no D-relation, and the algorithm may terminate with the result “no”

The equivalence class is represented by a set in the disjoint set data structure, and the elements in the set have a common representative that only represents elements in that set.
It suffices to ensure that

- always when a state is put for the first time into some set in the disjoint set data structure, it is compared to the representative of the set
- always when union is applied to two sets, their representatives are compared to each other

⇒ every state will be combined to its final representative via a chain of comparisons that goes through its earlier representatives

⇒ all states in the same equivalence class will be combined to the final representative of the class

Less comparisons are needed than there are states

- one state in any comparison is and the other is not the representative (of the union)
- each state is at most once in a comparison without being the representative
The same holds for D1

- D1 requires that the states in the equivalence class agree, for each \( a \in \Sigma \), whether \( s = a \) holds

A similar thing even holds for D3!

- D3 requires that 
  \[
  \forall s_1 : \forall s_2 : \forall a : \ s_1 \sim s_2 \land s_1 \overset{a}{\rightarrow} s'_1 \land s_2 \overset{a}{\rightarrow} s'_2 \rightarrow s'_1 \sim s'_2
  \]

  \( \Rightarrow \) it requires that those states in the equivalence class that have output \( a \)-transitions agree on the equivalence class to which the \( a \)-transitions lead
  - furthermore, if a state has many output \( a \)-transitions, they must lead to the same equivalence class

  \( \Rightarrow \) if two states that have been combined with a chain of comparisons disagree on the equivalence class where their outgoing \( a \)-transitions end, then disagreement arises in some comparison in the chain

In the case of D3, disagreement does not mean, however, that the result is “no”, but that the equivalence classes must be combined
It may, however, indirectly lead to “no”

- the D1- or D2-comparison made during the combining may yield “no”
- because of the combining, a D3-comparison must be made to the representatives

⇒ need for more combining may arise, which may eventually yield “no”

D4 can be taken care of by, for each $\tau$-transition, D1-, D2-, and D3-comparing once the states at the ends of the transition

Here is the definition of D-relation from screen 14 again

A relation $\sim \subseteq S \times S$ is a **D-relation**, if and only if it is an equivalence, and for every $s_1 \in S$, $s_2 \in S$, $s'_1 \in S$, $s'_2 \in S$, and $a \in \Sigma$ it holds

**D1** $s_1 \sim s_2 \land s_1 = a \Rightarrow s_2 = a \Rightarrow$

**D2** $s_1 \sim s_2 \land s_1 \xrightarrow{\tau^0} \Rightarrow s_2 \xrightarrow{\tau^0}$

**D3** $s_1 \sim s_2 \land s_1 \xrightarrow{a} s'_1 \land s_2 \xrightarrow{a} s'_2 \Rightarrow s'_1 \sim s'_2$

**D4** $s_1 \xrightarrow{\tau} s_2 \Rightarrow s_1 \sim s_2$
3.4 Comparison and union of equivalence classes

On the basis of the above, we can sketch a subroutine \( \text{Check}(s_1, s_2) \) that compares two states and either takes the union of their equivalence classes (unless they are the same to start with) or detects that it is impossible to construct a \( D \)-relation.

\[
\begin{align*}
s_1 &:= \text{representative}(s_1) ; s_2 := \text{representative}(s_2) \\
\text{if } s_1 = s_2 \text{ then return} \\
\text{if } \neg (s_1 \to \tau^\omega \leftrightarrow s_2 \to \tau^\omega) \text{ then terminate "no"} \\
\text{for } a \in \Sigma \text{ do} \\
\quad \text{if } \neg (s_1 = a \Rightarrow \leftrightarrow s_2 = a \Rightarrow) \text{ then terminate "no"} \\
\quad \text{if } s_1 \to a \land s_2 \to a \text{ then } \text{Unready}.add(s_1', s_2') \\
\text{Union}(s_1, s_2)
\end{align*}
\]

"representative"

- finds the representative of the state in the disjoint-set structure

- re-directs the links according to path compression

The implementation of the tests \( s \to \tau^\omega \) and \( s = a \Rightarrow \) is described later.
Unready is a set in which pairs of states are stored for later comparison and union of equivalence classes

- an ordinary array is okay

The states $s'_1$ and $s'_2$ are those for which $s_1 - a \rightarrow s'_1$ and $s_2 - a \rightarrow s'_2$ hold

- if there are many such states, one is chosen in a way that is described later

“Union” computes the union of the equivalence classes to which $s_1$ and $s_2$ belong
4  Efficient processing of $\tau$-paths

4.1  What is the problem?

The tests “does $s = a \Rightarrow$ hold” and “does $s - \tau^\omega \rightarrow$ hold” may require traversing arbitrarily long $\tau$-paths

- linear time, when done for one state
- quadratic, when done for every state

⇒ too expensive!

On the other hand, if $s_1 - \tau \rightarrow s_2$, then

- $s_2 = a \Rightarrow$ implies $s_1 = a \Rightarrow$
- $s_2 - \tau^\omega \rightarrow$ implies $s_1 - \tau^\omega \rightarrow$

⇒ It suffices to test, whether the following hold:

- $s_1 = a \Rightarrow \Rightarrow s_2 = a \Rightarrow$
- $s_1 - \tau^\omega \Rightarrow \Rightarrow s_2 - \tau^\omega \Rightarrow$
Furthermore, if $\neg(s_2 = a \Rightarrow)$, then $\neg(s = a \Rightarrow)$ holds for every state along every $\tau$-path that starts at $s_2$

- similarly for $\neg(s_2 - \tau^0 \rightarrow)$

In particular, if $\tau$-transitions lead to a state $s$ from which there are no $\tau$-transitions except perhaps a self-loop back to $s$, then

- $s = a \Rightarrow \iff s - a \rightarrow$
- $s - \tau^0 \rightarrow$ if and only if $s - \tau \rightarrow s$

$\Rightarrow$ the tests can be made for $s$ without traversing $\tau$-paths

Perhaps we can take advantage of this?
4.2 Collapsing \( \tau \)-components and removal of unreachable states

If one traverses \( \tau \)-transitions starting at an arbitrary state in a finite LTS, sooner or later one of the following happens:

- one ends up in a state that does not have outgoing \( \tau \)-transitions
- one comes to a state that has already been visited

The LTS in question is, of course, finite, because it is the input to a program.

If one comes to a state that has already been visited, a cycle that consists of \( \tau \)-transitions has been found.

All states of a \( \tau \)-cycle

- agree with each other, whether \( s \equiv a \Rightarrow \) holds
- satisfy \( s \rightarrow \tau^0 \)
- have the same continuations in the sense of the \( =\sigma \Rightarrow \)-relation

\( \Rightarrow \) Merging all states of the \( \tau \)-cycle to one state does not change the answer to the question “is the LTS operationally deterministic.”
More generally, all states of a maximal strongly connected component induced by τ-transitions can be merged

- linear time algorithms are known for finding maximal strongly connected components

⇒ τ-components can be collapsed in a separate cheap preprocessing stage

- the re-directing and merging of transitions caused by the collapsing can be done in linear time on the average by putting all transitions into a hash table
  - the only problem is to prevent two or more transitions with the same label from being created between same states in the same direction

Earlier we pointed out that

- unreachable states and transitions have no significance regarding operational determinism (or almost any other interesting property of an LTS)
  - the existence of a D-relation is only required for reachable states

We get rid of this detail by adding another stage to preprocessing, where unreachable states and transitions are removed

- a linear time graph algorithm
4.3 Bottom states

Thanks to collapsing $\tau$-components, traversing $\tau$-transitions leads sooner or later to a state, from which

- there are no outgoing $\tau$-transitions, or
- there is precisely one outgoing $\tau$-transition, and it is a self-loop

We call such states *bottom states*

As was noticed above, in the case of bottom states the tests $s \overset{a}{\Rightarrow}$ and $s \overset{\tau^0}{\rightarrow}$ reduce to the easy tests $s \overset{a}{\rightarrow}$ and $s \overset{\tau}{\rightarrow} s$

Because of D4, each state must be in the same equivalence class with every state that can be reached from it via $\tau$-transitions

⇒ we can decide that as the representative of an equivalence class we always choose a bottom state

⇒ at least one partner in a comparison is always a bottom state

⇒ effort is saved in the D1- and D2-tests
4.4 Avoiding the “$\equiv a \Rightarrow$”- and “$-\tau^0 \rightarrow$”-relation with other than bottom states

Thinking reveals that also the other member of the comparison can use the “$-a \rightarrow$”-relation instead of the “$\equiv a \Rightarrow$”-relation

- namely, if $s \equiv a \Rightarrow$, $s_B$ is the representative of $s$, and $\neg(s_B \equiv a \Rightarrow)$, then there is some $s'$ so that $s \equiv \varepsilon \Rightarrow s'$ and $s' -a \rightarrow$

- let $s'_B$ be the representative of $s'$

If $\neg(s'_B -a \rightarrow)$, this will be noticed when $s'$ is compared to its representative

In the opposite case, $s'_B -a \rightarrow$ and $\neg(s_B -a \rightarrow)$

- when D4 is applied to the transitions on the path $s \equiv \varepsilon \Rightarrow s'$, a chain of comparisons between $s_B$ and $s'_B$ is completed
  - the chain potentially goes via new representatives that are employed later
- the ends of the chain disagree on whether $-a \rightarrow$ holds

$\Rightarrow$ the situation is detected at some point in the chain
The use of the relation “$=a\Rightarrow$” can be fully replaced with the relation “$-a\rightarrow$”.

A similar reasoning shows that the test $s -\tau^0\rightarrow$ can be replaced with the test $s -\tau\rightarrow s$ also at the other end of the chain.

- because $\tau$-components have been collapsed and the LTS is finite, every infinite $\tau$-path is of the form $s_1 -\tau\rightarrow s_2 -\tau\rightarrow \cdots -\tau\rightarrow s_n -\tau\rightarrow s_n$.
4.5 Representative transitions

The processing of D3 requires that if \( s \xrightarrow{a} \), then the following must be found:

- \( s' \) such that \( s \xrightarrow{a} s' \)
- the equivalence class that contains \( s' \)

Unfortunately, a state may have many outgoing transitions with the same label

A solution

- one of them is chosen to represent all of them
  - \textit{representative transition}
- it is taken separately care of that the others lead to the same equivalence class

We denote the state at the end of the representative transition with \( s[a] \)

- if there is no \( a \)-transition from \( s \), we write \( s[a] = \perp \)
  - it is common in theoretical computer science that \( \perp \) represents “nothing”
- \( s[a] \) could be implemented by adding to each state an array whose size is \(|\Sigma|\)
  - we will later show an even more efficient implementation
Now \( \text{Check}(s_1, s_2) \) looks like the following:

\[
\begin{align*}
s_1 & := \text{representative}(s_1) \ ; \ s_2 := \text{representative}(s_2) \\
\textbf{if} \ s_1 = s_2 \textbf{ then return} \\
\textbf{if} \ \neg(s_1 \rightarrow \tau \rightarrow s_1 \leftrightarrow s_2 \rightarrow \tau \rightarrow s_2) \textbf{ then terminate} \ \text{“no”} \\
\textbf{for} \ a \in \Sigma \ \textbf{ do} \\
\quad \textbf{if} \ \neg(s_1[a] = \perp \leftrightarrow s_2[a] = \perp) \textbf{ then terminate} \ \text{“no”} \\
\quad \textbf{if} \ s_1[a] \neq \perp \textbf{ then Unready.add}(s_1[a], s_2[a]) \\
\text{Union}(s_1, s_2)
\end{align*}
\]

- everything except the \textbf{for}-loop is easy to implement efficiently
- really scanning every \( a \in \Sigma \) consumes \( \Theta(|\Sigma|) \) time, which may be much greater than the number of output transitions

\( \Rightarrow \) we will later develop a better way
5 The algorithm as a whole

5.1 Pseudocode

We still need a subroutine Set_representative($s_1, s_2$)

- sets the representative link of $s_1$ to point to $s_2$, and the upper approximation to the height of $s_2$ to 1
- it will only be called when equivalence classes have not otherwise been combined

⇒ produces the same result as would be obtained by
  - first making each state a singleton set
  - then calling Union($s_1, s_2$) in the direction where, if the upper approximations to the height of the representatives are the same, the common representative will be the representative of $s_2$

⇒ does not disturb the operation of the disjoint set data structure

- Union would not be reliable as such, because the representative must be a bottom state
It is possible to find a reachable bottom state for every state in linear time

- keyword: topological sorting
- can be done simultaneously with collapsing $\tau$-components

*The algorithm as a whole* looks like this:

- remove unreachable states and transitions
- collapse the $\tau$-components to single states

```plaintext
for $s \in S$ do
  find such a bottom state $s_B$ that $s = \varepsilon \Rightarrow s_B$
  Set_representative($s, s_B$)

for $s_B \in Bottom\_states$ do
  for $a \in \Sigma$ and $s' \in S$ such that $s_B \rightarrow a \rightarrow s'$ do
    Check($s_B[a], s'$)

for $s \in S - Bottom\_states$ do
  $s_B :=$ representative($s$)
  if $s \rightarrow \tau \rightarrow s \land \neg (s_B \rightarrow \tau \rightarrow s_B)$ then terminate “no”
```
for $a \in \Sigma$ do
  for $s \in S - \text{Bottom\_states}$ and $s' \in S$ such that $s - a \rightarrow s'$ do
    $s_B := \text{representative}(s)$
    if $s_B[a] = \bot$ then terminate "no"
    Check($s_B[a], s'$)
  for $s \in S$ and $s' \in S$ such that $s - \tau \rightarrow s'$ do
    Check($s, s'$)

while $\text{Unready} \neq \emptyset$ do
  choose any $(s_1, s_2) \in \text{Unready}$
  $\text{Unready} := \text{Unready} - \{(s_1, s_2)\}$
  Check($s_1, s_2$)

terminate "yes"

It is easy to convince oneself that all operations in the algorithm are legal or even necessary

It is more difficult to convince oneself that everything necessary is there
The dense mathematical correctness proof of the algorithm in the original publication is slightly more than two pages.

About one page is spent on proving that the algorithm does not give false “no”-answers.

- in the heart of the proof is an invariant that says that if the LTS has a D-relation “∼”, then, after executing the Set_representative-part, each of the following guarantees that $s_1 ∼ s_2$:
  
  - $\text{representative}(s_1) = \text{representative}(s_2)$
  - $(s_1, s_2) \in Unready$
More than one page is spent on proving that the algorithm does not yield false “yes”-answers

• in the heart of the proof is a relation that is otherwise like a D-relation, but the “∼”-relations have been replaced as follows:
  – the last ∼ of D3 has been replaced by a relation that allows chaining the presence of a pair or its inverse pair in $Unready$ and/or having the same representatives
  – the others have been replaced by having the same representatives

• the relation is made to hold by the code before the while-loop

• such a relation becomes a D-relation when $Unready = \emptyset$

$\Rightarrow$ if the while-loop terminates successfully, a D-relation has been constructed

• this is a typical pattern in proofs that something is computed long enough

The “no false ‘yes’-answers” proof also uses invariants of the type “the truth value of the claim ‘representative$(s)$ $\rightarrow a \rightarrow$’ does not ever change”
5.2 Fast scanning of output transitions

We have not yet told how the output transitions are scanned quickly enough in Check

\[
\text{for } a \in \Sigma \text{ do}
\]
\[
\text{if } \neg(s_1[a] = \bot \leftrightarrow s_2[a] = \bot) \text{ then terminate "no"}
\]
\[
\text{if } s_1[a] \neq \bot \text{ then } \text{Unready}.add(s_1[a], s_2[a])
\]

For that purpose, the outgoing non-\(\tau\)-transitions of each state \(s\) are represented with a two-level list as follows:

- a record of the upper-level list has three fields
  - such an \(a\) that \(s \xrightarrow{-a} \)
  - \(s[a]\)
  - a link to a lower-level list
- the upper-level list has been ordered in increasing order according to \(a\)
- the lower-level list contains the remaining \(s'\), for which \(s \xrightarrow{-a} s'\)
The loop is executed by scanning simultaneously the upper-level lists of $s_1$ and $s_2$

- the test $\neg(s_1[a] = \bot \leftrightarrow s_2[a] = \bot)$ triggers when a different label is met in one list from the other, or one list ends before the other

- the lower-level lists make it possible to avoid consuming time for useless scanning of other output transitions with the same label

⇒ time is not spent on scanning

- labels that do not have outgoing transitions
- other outgoing $a$-transitions than the one represented by $s[a]$
We also have to discuss this loop of the main program:

```
for a ∈ Σ do
    for s ∈ S − Bottom_states and s' ∈ S such that s −a→ s' do
        s_B := representative(s)
        if s_B[a] = ⊥ then terminate “no”
        Check(s_B[a], s')
```

- for each representative, a pointer is maintained to the current point in its upper list
  ⇒ the upper-level lists of the representatives need not be scanned except perhaps forward
  ⇒ the total cost of finding the $s_B[a]$ is at most $|\{a \mid s_B −a→\}|$ summed over all bottom states $s_B$
  ⇒ $O(|\Delta|)$

- each transition $s −a→ s'$ is found in constant time, if the transitions have been sorted in a common array or list according to $a$
Sorting the transitions according to their labels in the usual way takes $O(|\Delta| \log |\Delta|)$ time

- more than the promised running time of the algorithm as a whole
- transitions need not be sorted, if they are in a suitable order already in the input
- if $|\Sigma| \leq |\Delta|$, the transitions may be sorted according to their labels with counting-sort in $\Theta(|\Delta|)$ time
- $|\Sigma| \leq |\Delta|$ is not a particularly strongly restrictive assumption
  - if it is violated, then some labels are not used to label any transition

The two-level lists can be easily created from the sorted list of all transitions

- perhaps the lower-level lists are not needed, but the transitions that would be in them can be processed by scanning the list of all non-$\tau$-transitions
Collapsing $\tau$-components may introduce many transitions with the same name from the same state to the same state

- finding and removing double transitions in less than $O(|\Delta| \log |\Delta|)$ time is difficult

Solution: we will not even try to find and remove them!

- other than representative transitions are processed in the algorithm only once each
- at most one copy of a transition may become a representative transition
- processing a duplicate of an already processed transition causes little extra work
  - a call of Check caused by it terminates almost immediately
  - the rest of the work caused by processing an other than a representative transition is small
- where only upper-level lists are scanned, extra transitions in the lower-level lists do not matter

$\Rightarrow$ Duplicate transitions cause so little harm that it is not necessary to find and remove them

- this is good news, because we do not know how to remove them efficiently enough
5.3 Running time

The main part of Check is only entered if \( s_1 \) and \( s_2 \) had different representatives before the call

- after the call they have permanently the same representative

\( \Rightarrow \) the main part may be entered at most \( |S| - 1 \) times

In the main part of Check, one of the representatives of \( s_1 \) and \( s_2 \) before the call stops from being a representative

\( \Rightarrow \) the corresponding transition \( s \rightarrow s[a] \) is not any more an output transition of a representative, and thus cannot any more be processed in the main part of Check

\( \Rightarrow \) the operation \( \text{Unready}.\text{add} \) can be done at most \( |\Delta| \) times

\( \Rightarrow O(|\Delta|) \) operations are applied to \( \text{Unready} \), and Check is called at most \( 2|\Delta| \) times

The finding of representatives towards the beginning of Check causes at most \( \alpha(|\Delta|, |S|) \) amortized work per call
In the main program

- we have described an implementation of \texttt{for } $a \in \Sigma$ loops, where the work is proportional to the number of scanned transitions
- the rest is simple

$\Rightarrow$ The running time is $O(|S| + |\Delta| \alpha(|\Delta|, |S|))$
### 5.4 Producing a structurally deterministic LTS

After the algorithm has answered “yes”, a structurally deterministic LTS that is equivalent to the original LTS is obtained by collapsing each equivalence class to a single state:

- computation can be simplified by taking advantage of that the representatives are bottom states, and thus lack $\tau$-transitions that lead elsewhere

- it suffices to compute as follows:
  
  $S' = \{ \text{representative}(s) \mid s \in S \}$
  
  $\Delta' = \{ (s, a, s') \mid s \in S' \land a \neq \tau \land \exists s'' \in S : (s, a, s'') \in \Delta \land s' = \text{representative}(s'') \} \cup \{ (s, \tau, s) \mid s \in S' \land (s, \tau, s) \in \Delta \}$
  
  $\hat{s}' = \text{representative}(\hat{s})$
6 Final remarks


The following were essential for developing the algorithm:

- a good understanding of the application area
  - e.g., inventing D-relation

- knowledge of many algorithms in textbooks
  - e.g., disjoint-set data structure, finding strong components

- lots of programming skills

- sufficient mathematical skills

—— End of case 5 ——