Five Determinisation Algorithms

Rob van Glabbeek[1,2] and Bas Ploeger[3⋆]

[1] National ICT Australia, Locked Bag 6016, Sydney, NSW1466, Australia
[2] School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia
[3] Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. Determinisation of nondeterministic finite automata is a well-studied problem that plays an important role in compiler theory and system verification. In the latter field, one often encounters automata consisting of millions or even billions of states. On such input, the memory usage of analysis tools becomes the major bottleneck. In this paper we present several determinisation algorithms, all variants of the well-known subset construction, that aim to reduce memory usage and produce smaller output automata. One of them produces automata that are already minimal. We apply our algorithms to determinise automata that describe the possible sequences appearing after a fixed-length run of cellular automaton 110, and obtain a significant improvement in both memory and time efficiency.

1 Introduction

Finite state automata (or finite state machines) are an established and well-studied model of computation. From a theoretical point of view, they are an interesting object of study because they are expressive yet conceptually easy to understand and intuitive. They find applications in compilers, natural language processing, system verification and testing, but also in fields outside of (theoretical) computer science like switching circuits and chip design. Over the years, many flavours and variants of finite state machines have been defined and studied for a large variety of purposes.

One of the most classic and elementary type of finite state machine is the *nondeterministic finite automaton* (NFA). Typical applications of finite state automata involve checking whether some sequence of symbols meets some syntactic criterion, such as displaying a prescribed pattern or being correct input for a given program, a problem that can often be recast as checking whether that sequence is accepted by a given NFA.

A more restrictive type of automaton is the *deterministic finite automaton* (DFA). DFAs are as expressive as NFAs, in the sense that for every NFA there exists a DFA that is *language equivalent* (*i.e.* accepts the same input sequences). Contrary to NFAs, for any DFA there is a trivial linear time, constant space, online algorithm to check whether an input sequence is accepted or not. Consequently, lexical-analyser generators like LEX work on DFAs, and so do many implementations of GREP. For this reason, in many applications it pays to convert NFAs into DFAs, even though the worst-case time and space complexities of this conversion are exponential in the size of the input NFA.

⋆ This author is partially supported by the Netherlands Organisation for Scientific Research (NWO) under VoLTS grant number 612.065.410.

Once a language equivalent DFA of an NFA has been found, it is usually minimised to obtain the smallest such DFA. This minimal DFA is unique and the problem of finding it for a given NFA is called the *canonisation problem*.

Another application of NFAs is in the realm of *process theory* and *system verification* where they are used to model the behaviour of distributed systems. Typically, both a specification and an implementation of a system are represented as NFAs, and the question arises whether the execution sequences of one NFA are a subset of those of another. This is the *trace inclusion problem*. Although PSPACE-hard in general, this problem is decidable in PTIME once the NFAs are converted into equivalent DFAs.

As we see, in both the canonisation problem and the trace inclusion problem, determinisation plays an essential role. The standard determinisation algorithm is called *subset construction* (see *e.g.* [11]). Although the determinisation problem is EXPTIME-hard, this algorithm is renowned for its good performance in practice. For minimisation of DFAs a lot of algorithms have been proposed, of which Watson presents a taxonomy and performance analyses [16]. The algorithm with the best time complexity is by Hopcroft [10]: $\mathcal{O}(n \log n)$ where n is the number of states in the input DFA.

Another algorithm for canonisation is by Brzozowski [2]. It generates the minimal DFA directly from an input NFA by repeating the process of "reversing" and determinising the automaton twice. Tabakov and Vardi compare both approaches to canonisation experimentally by running them on randomly generated automata [15].

On some NFAs, the exponential blow-up by subset construction is unavoidable. However, we have encountered NFAs for which subset construction consumes a lot of memory and generates a DFA that is much larger than the minimal DFA. Therefore, our main goal is to find algorithms that are more memory efficient and produce smaller DFAs than subset construction.

In this paper we present five determinisation algorithms based on subset construction. For all of them we prove correctness. One algorithm generates the minimal DFA directly and hence is a *canonisation algorithm*. However, it calculates language inclusion as a subroutine; as deciding language inclusion is PSPACE-complete, it is unattractive to use in an implementation. The other four produce a DFA that is not necessarily minimal but is usually smaller than the DFA produced by subset construction.

We have implemented subset construction and these four new algorithms. We have benchmarked these implementations by running them on NFAs that describe patterns on the lines of a cellular automaton's evolution. We compare the implementations on the time and memory needed for the complete canonisation process (*i.e.* including minimisation) and the size of the DFA after determinisation.

2 Preliminaries

Finite automata. A *nondeterministic finite automaton* (NFA) $\mathcal{N}$ is a tuple $(S_\mathcal{N}, \Sigma_\mathcal{N}, \delta_\mathcal{N}, i_\mathcal{N}, F_\mathcal{N})$ where $S_\mathcal{N}$ is a finite set of states, $\Sigma_\mathcal{N}$ is a finite input alphabet, $\delta_\mathcal{N} \subseteq S_\mathcal{N} \times \Sigma_\mathcal{N} \times S_\mathcal{N}$ is a transition relation, $i_\mathcal{N} \in S_\mathcal{N}$ is the initial state and $F_\mathcal{N} \subseteq S_\mathcal{N}$ is a set of final (or accepting) states. A *deterministic finite automaton* (DFA) is an NFA $\mathcal{D}$ such that for all $p \in S_\mathcal{D}$ and $a \in \Sigma_\mathcal{D}$ there is precisely one $q \in S_\mathcal{D}$ such that $(p, a, q) \in \delta_\mathcal{D}$.

In graphical representations of DFAs we also allow states that have *at most* one outgoing a-transition for each alphabet symbol a. Formally speaking, these abbreviate the

DFA obtained by adding a non-accepting *sink* state as the target of all missing transitions. Note that adding such a state preserves language equivalence (defined below).

For any alphabet Σ, Σ^* denotes the set of all finite strings over Σ and $\varepsilon \in \Sigma^*$ denotes the empty string. Any subset of Σ^* is called a *language over* Σ. For any states $p, q \in S_\mathcal{N}$ of an NFA $\mathcal{N}$ and string $\sigma \in \Sigma_\mathcal{N}^*$ with $\sigma = \sigma_1 \cdots \sigma_n$ and $\sigma_1, \ldots, \sigma_n \in \Sigma_\mathcal{N}$ for some $n \geq 0$, we write $p \xrightarrow{\sigma}_\mathcal{N} q$ to denote the fact that:

$$\exists p_0, \ldots, p_n \in S_\mathcal{N} \,.\, p_0 = p \,\wedge\, p_n = q \,\wedge\, (p_0, \sigma_1, p_1), \ldots, (p_{n-1}, \sigma_n, p_n) \in \delta_\mathcal{N}.$$

Language semantics. The *language of a state* $p \in S_\mathcal{N}$ of an NFA $\mathcal{N}$ is defined as: $\mathcal{L}_\mathcal{N}(p) = \{\sigma \in \Sigma_\mathcal{N}^* \mid \exists q \in F_\mathcal{N} \,.\, p \xrightarrow{\sigma}_\mathcal{N} q\}$. The *language of an NFA* $\mathcal{N}$ is defined as: $\mathcal{L}(\mathcal{N}) = \mathcal{L}_\mathcal{N}(i_\mathcal{N})$. For any NFAs $\mathcal{N}$ and $\mathcal{M}$ and states $p \in S_\mathcal{N}$ and $q \in S_\mathcal{M}$, p is *language included* in q, denoted $p \sqsubseteq_L q$, iff $\mathcal{L}_\mathcal{N}(p) \subseteq \mathcal{L}_\mathcal{M}(q)$. Moreover, p and q are *language equivalent*, denoted $p \equiv_L q$, iff $p \sqsubseteq_L q \wedge q \sqsubseteq_L p$. An NFA $\mathcal{N}$ is *language included* in an NFA $\mathcal{M}$ iff $i_\mathcal{N} \sqsubseteq_L i_\mathcal{M}$ and $\mathcal{N}$ and $\mathcal{M}$ are *language equivalent* iff $i_\mathcal{N} \equiv_L i_\mathcal{M}$.

Simulation semantics. Given NFAs $\mathcal{N}$ and $\mathcal{M}$, a relation $R \subseteq S_\mathcal{N} \times S_\mathcal{M}$ is a *simulation* iff for any $p \in S_\mathcal{N}$ and $q \in S_\mathcal{M}$, $p \;R\; q$ implies:

- $p \in F_\mathcal{N} \Rightarrow q \in F_\mathcal{M}$ and
- $\forall a \in \Sigma_\mathcal{N} \,.\, \forall p' \in S_\mathcal{N} \,.\, p \xrightarrow{a}_\mathcal{N} p' \;\Rightarrow\; \exists q' \in S_\mathcal{M} \,.\, q \xrightarrow{a}_\mathcal{M} q' \,\wedge\, p' \;R\; q'$.

Given NFAs $\mathcal{N}$ and $\mathcal{M}$, for any $p \in S_\mathcal{N}$ and $q \in S_\mathcal{M}$:

- p is *simulated by* q, denoted $p \subseteq_\rightarrow q$, iff there exists a simulation R such that $p \;R\; q$;
- p and q are *simulation equivalent*, denoted $p \rightleftarrows q$, iff $p \subseteq_\rightarrow q \wedge q \subseteq_\rightarrow p$;

Clearly $p \subseteq_\rightarrow q$ implies $p \sqsubseteq_L q$.

Subset construction. The subset construction (or powerset construction) is the standard way of determinising a given NFA. For reasons that will become apparent in the next sections, we slightly generalise the normal algorithm by augmenting it with a function f on sets of states, which is applied to every generated set. The algorithm is Algorithm 1 and shall be referred to as $\textsc{Subset}(f)$. It takes an NFA $\mathcal{N}$ and generates a DFA $\mathcal{D}$. Of course, it should be the case that $\mathcal{N} \equiv_L \mathcal{D}$, which depends strongly on the function f. For normal subset construction, $\textsc{Subset}(\mathcal{I})$, where $\mathcal{I}$ is the identity function, it is known that the language of $\mathcal{N}$ is indeed preserved. In the sequel, whenever we use the term "subset construction" we mean the normal algorithm, *i.e.* $\textsc{Subset}(\mathcal{I})$.

It is known that in the worst case, determinisation yields a DFA that is exponentially larger than the input NFA. An example of an NFA that gives rise to such an exponential blow-up is the NFA that accepts the language specified by the regular expression $\Sigma^* x \Sigma^n$ for some alphabet Σ, $x \in \Sigma$ and $n \geq 0$. Figure 1(a) shows the NFA for $\Sigma = \{a, b\}$ and $x = a$. This NFA has $n + 2$ states, whereas the corresponding DFA has 2^{n+1} states and is already minimal.

An interesting thing to note is that if the initial state were accepting (Figure 1(b)), the minimal DFA would consist of only one state with an a, b-loop: the accepted language has become Σ^*. However, subset construction still produces the exponentially larger DFA first, which should then be reduced to obtain the single-state, minimal DFA.

Algorithm 1 The SUBSET(f) determinisation algorithm

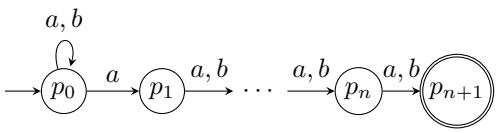
Pre: $\mathcal{N} = (S_\mathcal{N}, \Sigma_\mathcal{N}, \delta_\mathcal{N}, i_\mathcal{N}, F_\mathcal{N})$ is an NFA
Post: $\mathcal{D} = (S_\mathcal{D}, \Sigma_\mathcal{D}, \delta_\mathcal{D}, i_\mathcal{D}, F_\mathcal{D})$ is a DFA
1: $\Sigma_\mathcal{D} := \Sigma_\mathcal{N}$; $\delta_\mathcal{D} := \emptyset$; $i_\mathcal{D} := f(\{i_\mathcal{N}\})$; $F_\mathcal{D} := \emptyset$;
2: $S_\mathcal{D} := \{i_\mathcal{D}\}$; todo $:= \{i_\mathcal{D}\}$; done $:= \emptyset$;
3: **while** todo $\neq \emptyset$ **do**
4: pick a $P \in$ todo;
5: **for all** $a \in \Sigma_\mathcal{N}$ **do**
6: $P' := f(\{p' \in S_\mathcal{N} \mid \exists p \in P \,.\, p \xrightarrow{a}_\mathcal{N} p'\})$;
7: $S_\mathcal{D} := S_\mathcal{D} \cup \{P'\}$;
8: $\delta_\mathcal{D} := \delta_\mathcal{D} \cup \{(P, a, P')\}$;
9: todo $:=$ todo $\cup\, (\{P'\} \setminus$ done$)$;
10: **end for**
11: **if** $\exists p \in P \,.\, p \in F_\mathcal{N}$ **then**
12: $F_\mathcal{D} := F_\mathcal{D} \cup \{P\}$;
13: **end if**
14: todo $:=$ todo $\setminus \{P\}$;
15: done $:=$ done $\cup \{P\}$;
16: **end while**

3 Determinisation using Transition Sets

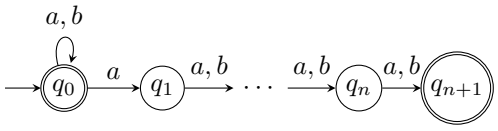
In this section we show that subset construction can just as well be done on sets of transitions as on sets of states. We observe that the contribution of an NFA state p to the behaviour of a DFA state P consists entirely of p's outgoing transitions. We no longer think of a DFA state as being a set of NFA states, but rather a set of NFA *transitions*.

Definition 1. Given an NFA $\mathcal{N}$, a *transition tuple* is a pair (T, b) where $T \in \mathcal{P}(\Sigma_\mathcal{N} \times S_\mathcal{N})$ is a set of transitions and $b \in \mathbb{B}$ is a boolean.

For every transition tuple (T, b) we define the projection functions set and fin as:



(a)



(b)

Fig. 1. Two NFAs of size $\mathcal{O}(n)$ for which subset construction produces a DFA of size $\mathcal{O}(2^n)$. Here initial states are marked by unlabelled incoming arrows, and final states by double circles. In case (a) this DFA is already minimal; in case (b) the minimal DFA has size 1.

Algorithm 2 The TRANSSET(f) determinisation algorithm

Pre: $\mathcal{N} = (S_\mathcal{N}, \Sigma_\mathcal{N}, \delta_\mathcal{N}, i_\mathcal{N}, F_\mathcal{N})$ is an NFA
Post: $\mathcal{D} = (S_\mathcal{D}, \Sigma_\mathcal{D}, \delta_\mathcal{D}, i_\mathcal{D}, F_\mathcal{D})$ is a DFA
1: $\Sigma_\mathcal{D} := \Sigma_\mathcal{N}$; $\delta_\mathcal{D} := \emptyset$; $i_\mathcal{D} := f(\mathsf{tuple}(i_\mathcal{N}))$; $F_\mathcal{D} := \emptyset$;
2: $S_\mathcal{D} := \{i_\mathcal{D}\}$; todo $:= \{i_\mathcal{D}\}$; done $:= \emptyset$;
3: **while** todo $\neq \emptyset$ **do**
4: pick a $P \in$ todo;
5: **for all** $a \in \Sigma$ **do**
6: $P' := f(\bigcup_{(a,p)\in\mathsf{set}(P)} \mathsf{trans}(p),\ \exists(a,p) \in \mathsf{set}(P)\,.\, p \in F_\mathcal{N})$;
7: $S_\mathcal{D} := S_\mathcal{D} \cup \{P'\}$;
8: $\delta_\mathcal{D} := \delta_\mathcal{D} \cup \{(P, a, P')\}$;
9: todo $:=$ todo $\cup\, (\{P'\} \setminus$ done$)$;
10: **end for**
11: **if** $\mathsf{fin}(P)$ **then**
12: $F_\mathcal{D} := F_\mathcal{D} \cup \{P\}$;
13: **end if**
14: todo $:=$ todo $\setminus\, \{P\}$;
15: done $:=$ done $\cup\, \{P\}$;
16: **end while**

$\mathsf{set}(T, b) = T$ and $\mathsf{fin}(T, b) = b$. For every state $p \in S_\mathcal{N}$ of NFA $\mathcal{N}$, $\mathsf{trans}(p)$ is the set of outgoing transitions of p and $\mathsf{tuple}(p)$ is the transition tuple belonging to p:

$$\mathsf{trans}(p) = \{(a, q) \in \Sigma_\mathcal{N} \times S_\mathcal{N} \mid p \xrightarrow{a}_\mathcal{N} q\}$$
$$\mathsf{tuple}(p) = (\mathsf{trans}(p), p \in F_\mathcal{N}).$$

The DFA state $P \subseteq S_\mathcal{N}$ now corresponds to the transition tuple (T, b) where $T = \bigcup_{p\in P} \mathsf{trans}(p)$ and $b \equiv \exists p \in P\,.\, p \in F_\mathcal{N}$. We need the boolean b to indicate whether the DFA state is final as this can no longer be determined from the elements of the set. Only the labels and target states of the transitions are stored because the source states are irrelevant and would only make the sets unnecessarily large.

Given NFA $\mathcal{N}$, the *language of a transition* $(a, p) \in \Sigma_\mathcal{N} \times S_\mathcal{N}$ is defined as: $\mathcal{L}_\mathcal{N}(a, p) = \{a\sigma \in \Sigma_\mathcal{N}^* \mid \sigma \in \mathcal{L}_\mathcal{N}(p)\}$. The *language of a set of transitions* T is defined as $\mathcal{L}_\mathcal{N}(T) = \bigcup_{t\in T} \mathcal{L}_\mathcal{N}(t)$ and the *language of a transition tuple* (T, b) is defined as:

$$\mathcal{L}_\mathcal{N}(T, b) = \mathcal{L}_\mathcal{N}(T) \cup \begin{cases} \{\varepsilon\} & \text{if } b \\ \emptyset & \text{if } \neg b. \end{cases}$$

Language inclusion and equivalence for transitions and transition tuples can now be defined in the usual way by means of set inclusion and equality.

The determinisation algorithm that uses transition tuples is Algorithm 2. We shall refer to it as TRANSSET(f) where f is a function on transition tuples. Again, language preservation depends on the specific function f being used. For $f = \mathcal{I}$ this is indeed the case, which we prove in [7], the full version of this paper. Using TRANSSET($\mathcal{I}$) for determinisation can give a smaller DFA than SUBSET($\mathcal{I}$) as is shown by the example in Figure 2. Here, TRANSSET($\mathcal{I}$) happens to produce the minimal DFA directly. This is generally not the case: on the NFA of Figure 1(b), TRANSSET($\mathcal{I}$) generates a DFA of size 2^{n+1}, while the minimal DFA has size 1.

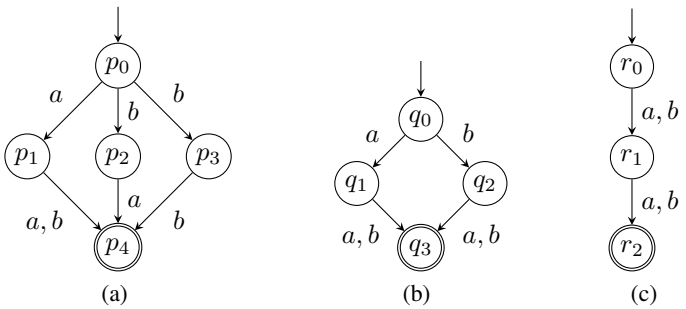


Fig. 2. NFA (a) for which the DFA produced by SUBSET($\mathcal{I}$) (b) is larger than the (minimal) DFA produced by TRANSSET($\mathcal{I}$) (c).

4 Determinisation using Closures

We introduce a *closure* operation that can be used in the SUBSET algorithm instead of the identity function $\mathcal{I}$. It aims to add NFA states to a given DFA state (*i.e.* a set of NFA states) without affecting its language. This results in an algorithm that generates smaller DFAs. In particular, we show that if the criterion to add a state is chosen suitably, SUBSET with closure is an algorithm that produces the minimal DFA directly.

Definition 2. For any set of states $P \subseteq S_{\mathcal{N}}$ of an NFA $\mathcal{N}$ and relation $\sqsubseteq \subseteq S_{\mathcal{N}} \times \mathcal{P}(S_{\mathcal{N}})$, the *closure* of P under $\sqsubseteq$, $\mathsf{close}_{\sqsubseteq}(P)$, is defined as:

$$\mathsf{close}_{\sqsubseteq}(P) = \{p \in S_{\mathcal{N}} \mid p \sqsubseteq P\}.$$

The language preorder $\sqsubseteq_L$ can be lifted to operate on states and sets of states in the following way. Define the *language of a set of states* P of an NFA $\mathcal{N}$ as: $\mathcal{L}_{\mathcal{N}}(P) = \bigcup_{p \in P} \mathcal{L}_{\mathcal{N}}(p)$. Language equivalence and inclusion can now be defined on any combination of states and sets of states, in terms of set equivalence and inclusion. For instance, for a state $p \in S_{\mathcal{N}}$ and a set of states $P \subseteq S_{\mathcal{N}}$, $p \sqsubseteq_L P$ holds if $\mathcal{L}_{\mathcal{N}}(p) \subseteq \mathcal{L}_{\mathcal{N}}(P)$.

Applying this, the algorithm SUBSET($\mathsf{close}_{\sqsubseteq_L}$) generates the minimal DFA that is language equivalent to the input NFA. This statement is proven in [7].

5 Simulation Preorder

Although it ensures that the output DFA of SUBSET($\mathsf{close}_{\sqsubseteq_L}$) is minimal, language inclusion is an unattractive preorder to use. Deciding language inclusion is PSPACE-complete [13] which implies that known algorithms have an exponential time complexity. Moreover, most algorithms involve a determinisation step which would render our optimisation useless.

The simulation preorder $\subseteq$ [12] is finer than language inclusion on NFAs, meaning it relates fewer NFAs. However, considering its PTIME complexity (see *e.g.* [1, 9]), it is an attractive way to "approximate" language inclusion (see also [4]). Hence, as a more

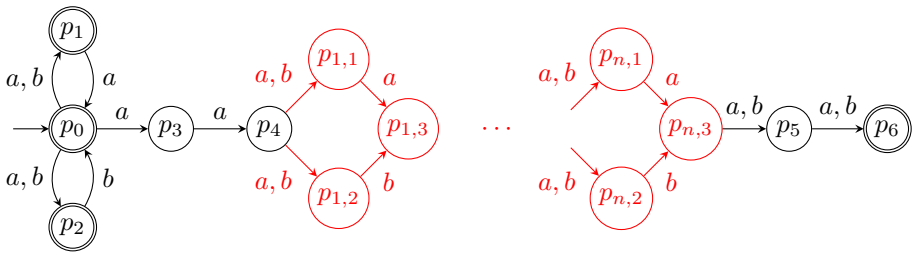


Fig. 3. NFA of size $\mathcal{O}(n)$ for which SUBSET($\mathsf{close}_{\subseteq}$) generates a DFA of size $\mathcal{O}(2^n)$ for any $n \geq 1$. The minimal DFA has 1 state.

practical alternative to SUBSET($\mathsf{close}_{\sqsubseteq_L}$) we define the algorithm SUBSET($\mathsf{close}_{\subseteq}$). The required lifting of $\subseteq$ to states and sets of states is as follows. For any state $p \in S_\mathcal{N}$ and set of states $P \subseteq S_\mathcal{N}$ of an NFA $\mathcal{N}$, we have $p \subseteq P$ iff:

- $p \in F_\mathcal{N} \Rightarrow \exists q \in P \,.\, q \in F_\mathcal{N}$ and
- there exists a simulation $R \subseteq S_\mathcal{N} \times S_\mathcal{N}$ such that:

  $\forall a \in \Sigma_\mathcal{N} \,.\, \forall p' \in S_\mathcal{N} \,.\, p \xrightarrow{a}_\mathcal{N} p' \;\Rightarrow\; \exists q, q' \in S_\mathcal{N} \,.\, q \in P \wedge q \xrightarrow{a}_\mathcal{N} q' \;\wedge\; p' \; R \; q'.$

The correctness of SUBSET($\mathsf{close}_{\subseteq}$) is established in [7]. The example in Figure 3 shows not only that the resulting DFA is no longer minimal, but moreover that it can be exponentially larger than the minimal DFA. This NFA contains a pattern that repeats itself n times for any $n \geq 1$. It is based on the NFA of Figure 1(b) interwoven with a pattern that prevents SUBSET($\mathsf{close}_{\subseteq}$) from merging states that will later turn out to be equivalent. The NFA accepts the language given by the regular expression $(a \mid b)^*$.

6 Determinisation using Compressions

Algorithm SUBSET($\mathsf{close}_{\subseteq}$) adds all simulated states to a generated set of states. Another option would be to remove all redundant states from such a set. More specifically, we remove every state that is simulated by another state in the set. For this operation to be well-defined, it is essential that no two different states in the set are simulation equivalent. This can be achieved by minimising the input NFA using simulation equivalence prior to determinisation. In turn, this amounts to computing the simulation preorder that was already necessary in the first place.

Definition 3. Given a set P such that $\neg\exists p, q \in P \,.\, p \neq q \wedge p \rightleftarrows q$. Then $\mathsf{compress}_{\subseteq}(P)$ denotes the *compression* of P under $\subseteq$ and is defined as:

$$\mathsf{compress}_{\subseteq}(P) = \{p \in P \mid \forall q \in P \,.\, p \neq q \Rightarrow p \not\subseteq q\}.$$

The function $\mathsf{compress}_{\subseteq}$ can be used not only for sets of states but also for transition tuples. For that, we first define $\subseteq$ on the transitions of an NFA $\mathcal{N}$ as follows. For any $(a, p), (b, q) \in \Sigma_\mathcal{N} \times S_\mathcal{N}$, we have $(a, p) \subseteq (b, q)$ iff $a = b$ and $p \subseteq q$. By Definition 3 $\mathsf{compress}_{\subseteq}$ is now properly defined on sets of transitions and it can be extended to transition tuples in a straightforward manner: $\mathsf{compress}_{\subseteq}(T, b) = (\mathsf{compress}_{\subseteq}(T), b)$.

This way, we obtain two more determinisation algorithms: $\textsc{Subset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ and $\textsc{Transset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$. Their correctness proofs can be found in [7].

7 Lattice of Algorithms

Figure 4 orders the algorithms described in the previous sections in a lattice: we draw an arrow from algorithm A to algorithm B iff for every input NFA, A produces a DFA that is at most as large as the one produced by B. The algorithms $\textsc{Subset}(\mathsf{close}_{\subseteq}^{\rightarrow})$ and $\textsc{Transset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ are in the same class of the lattice, because they always yield isomorphic DFAs. The relations of Figure 4 are substantiated in [7]; Figures 1(b), 2 and 3 provide counterexamples against further inclusions.

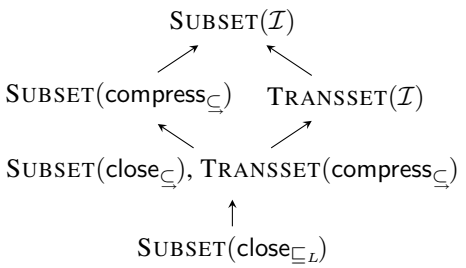


Fig. 4. The lattice of algorithms presented in the previous sections.

8 Implementation and Benchmarks

We have implemented the algorithms $\textsc{Subset}(\mathcal{I})$, $\textsc{Transset}(\mathcal{I})$, $\textsc{Subset}(\mathsf{close}_{\subseteq}^{\rightarrow})$, $\textsc{Subset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ and $\textsc{Transset}(\mathsf{compress}_{\subseteq}^{\rightarrow})$ in the C++ programming language. A set of states or transitions is stored as a tree with the elements in the leaves. All subtrees are shared among the sets to improve memory efficiency. A hash table provides fast and efficient lookup of existing subtrees.

The benchmarks are performed on a 32-bits architecture computer having two Intel Xeon 3.06 GHz CPUs and 4 GB of RAM. It runs Fedora Core 8 Linux, kernel 2.6.23. The code is compiled using the GNU C++ compiler (version 4.1.2).

Every benchmark starts off by minimising the NFA using simulation equivalence. For this we have implemented our partitioning algorithm [6] which is based on [5] and also computes the simulation preorder on the states of the resulting NFA. Every determinisation algorithm is applied to this minimised NFA, after which the resulting DFA is minimised by the tool ltsmin of the μCRL toolset [3, 8] (version 2.18.1).

For the benchmarks we consider a one-dimensional cellular automaton (CA) (see *e.g.* [18]), which is represented by a function $\rho : \Sigma^w \to \Sigma$ called the *rule* where Σ is an alphabet and w is the *width* of the automaton. Given an infinite sequence $\sigma \in \Sigma^\infty$, a *step* of a CA is an application of ρ to every w-length subsequence of σ, which produces a new sequence. The possible finite sequences appearing as a continuous subsequence

	STEP 4					STEP 5				
	DT	**MT**	**DS**	**MS**	$\lvert S_{\mathcal{D}}\rvert$	**DT**	**MT**	**DS**	**MS**	$\lvert S_{\mathcal{D}}\rvert$
SUBSET($\mathcal{I}$)	0.6	0.4	5.4	2.0	58 370	212.5	76.7	688.2	267.2	7 663 165
TRANSSET($\mathcal{I}$)	1.0	0.4	9.0	2.0	58 094	257.3	79.1	1 146.9	263.0	7 541 248
SUBSET($\mathsf{close}_{\subseteq}$)	1.6	< 0.1	2.1	0.2	4 720	2 739.7	1.61	123.2	6.3	176 008
SUBSET($\mathsf{compress}_{\subseteq}$)	< 0.1	< 0.1	0.6	0.2	4 745	4.3	1.4	16.7	6.4	179 146
TRANSSET($\mathsf{compress}_{\subseteq}$)	< 0.1	< 0.1	0.7	0.2	4 720	4.1	1.6	22.9	6.3	176 008

Table 1. Benchmark results for canonising NFAs of steps 4 and 5 of CA 110.
Legend: **D** = Determinisation, **M** = Minimisation, **T** = Time (sec), **S** = Space (peak memory use, MB), $|S_{\mathcal{D}}|$ = Size of DFA after determinisation.

of the infinite sequence obtained after n steps of a given CA (starting from a random input sequence) constitute a language that can be described by a DFA [17]. It is known that for some CA rules, the size of these DFAs increases exponentially in n (*cf.* [14]).

For $\Sigma = \{0, 1\}$ and $w = 3$, the CA with number 110 has the following rule:

$$\rho = \{\ 000 \mapsto 0,\ 001 \mapsto 1,\ 010 \mapsto 1,\ 011 \mapsto 1,\\ 100 \mapsto 0,\ 101 \mapsto 1,\ 110 \mapsto 1,\ 111 \mapsto 0\ \}.$$

It is known to be computationally universal and to exhibit the exponential blow-up phenomenon described above. We have generated the minimal DFAs for steps 1 through 5 of this CA using the various algorithms presented here. The most interesting results are those for steps 4 and 5, which are shown in Table 1. The input NFA has 228 states for step 4 and 1 421 for step 5; the minimal DFAs have sizes 1 357 and 18 824 respectively. The algorithms that use $\mathsf{compress}_{\subseteq}$ clearly outperform the others, in both memory and time efficiency. Every algorithm that uses a function other than $\mathcal{I}$, generates a DFA that is an order of magnitude smaller than that of its $\mathcal{I}$-counterpart.

9 Conclusions

We have presented a schematic generalisation of the well-known subset construction algorithm that allows for a function to be applied to every generated set of states. We have given a similar scheme for a variant of subset construction that operates on sets of transitions rather than states. Next, we instantiated these schemes with several set-expanding or -reducing functions to obtain various determinisation algorithms. One of these algorithms even produces the minimal DFA directly, but its use of the PSPACE-hard language preorder renders it impractical. As our aim is to reduce the average-case workload in practice, we instead use the PTIME-decidable simulation preorder in the other algorithms. We have classified all presented algorithms in a lattice, based on the sizes of the DFAs they produce. This is a natural criterion, as the worst-case complexities are the same for all algorithms. To assess their performance, we have implemented and benchmarked them. The case study comprised NFAs describing patterns in the elementary cellular automaton with rule number 110. On these examples, the algorithms that use a function to reduce the computed sets, convincingly outperformed the others.

Based on our algorithm schemes, many more algorithms can be constructed by substituting various functions, depending on the specific needs and applications. Moreover, the functions we defined here could be equipped with any suitable preorder or partial order, *e.g.* from the linear time – branching time spectrum. We also remark that our optimisations to subset construction are particularly beneficial in cases where normal subset construction leaves a large gap between the generated DFA and the minimal one.

Acknowledgements. We would like to thank Jan Friso Groote, Tim Willemse and Sebastian Maneth for valuable ideas, discussions and/or comments.

References

1. B. Bloom & R. Paige (1995): *Transformational design and implementation of a new efficient solution to the ready simulation problem.* Science of Computer Programming 24(3), pp. 189–220.
2. J.A. Brzozowski (1963): *Canonical regular expressions and minimal state graphs for definite events.* In Proceedings of the Symposium on Mathematical Theory of Automata, MRI Symposia Series, vol. 12, Polytechnic Press, Polytechnic Institute of Brooklyn, pp. 529–561.
3. CWI: *μCRL Toolset Home Page.* `http://www.cwi.nl/~mcrl/`.
4. D.L. Dill, A.J. Hu & H. Wong-Toi (1992): *Checking for language inclusion using simulation preorders.* In Proceedings of the Third International Workshop on Computer-Aided Verification, LNCS 575, Springer, pp. 255–265.
5. R. Gentilini, C. Piazza & A. Policriti (2003): *From bisimulation to simulation: Coarsest partition problems.* Journal of Automated Reasoning 31(1), pp. 73–103.
6. R.J. van Glabbeek & B. Ploeger (2008): *Correcting a space-efficient simulation algorithm.* To appear in Proc. 20th Int. Conf. on Computer Aided Verification, LNCS, Springer.
7. R.J. van Glabbeek & B. Ploeger (2008): *Five Determinisation Algorithms*. CS-Report 08-14, Eindhoven University of Technology.
8. J.F. Groote & M.A. Reniers (2001): *Algebraic process verification.* In J.A. Bergstra, A. Ponse & S.A. Smolka, editors: Handbook of Process Algebra, Elsevier, pp. 1151–1208.
9. M.R. Henzinger, T.A. Henzinger & P.W. Kopke (1995): *Computing simulations on finite and infinite graphs.* In 36th Annual Symposium on Foundations of Computer Science (FOCS'95), IEEE Computer Society Press, pp. 453–462.
10. J.E. Hopcroft (1971): *An $n \log n$ algorithm for minimizing states in a finite automaton.* In Z. Kohavi, editor: Theory of Machines and Computations, Academic Press, pp. 189–196.
11. J.E. Hopcroft & J.D. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.
12. D.M.R. Park (1981): *Concurrency and automata on infinite sequences.* In Proceedings 5th GI-Conference on Theoretical Computer Science, LNCS 104, Springer, pp. 167–183.
13. L.J. Stockmeyer & A.R. Meyer (1973): *Word problems requiring exponential time.* In Proc. 5th Annual ACM Symposium on Theory of Computing (STOC'73), ACM, pp. 1–9.
14. K. Sutner (2003): *The size of power automata.* Theor. Comput. Sci. 295(1-3), pp. 371–386.
15. D. Tabakov & M.Y. Vardi (2005): *Experimental evaluation of classical automata constructions.* In Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LNCS 3835, Springer, pp. 396–411.
16. B.W. Watson (1995): *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Technische Universiteit Eindhoven.
17. S. Wolfram (1984): *Computation theory of cellular automata.* Communications in Mathematical Physics 96(1), pp. 15–57.
18. S. Wolfram (2002): *A New Kind of Science*. Wolfram Media, Inc.