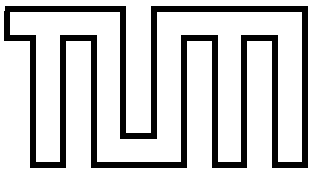
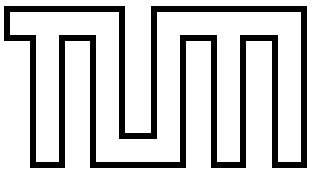


FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's thesis in Computer Science

Efficient and Verified Computation of Simulation Relations on NFAs

Manuel Eberl



FAKULTÄT FÜR INFORMATIK

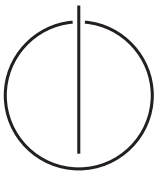
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's thesis in Computer Science

Efficient and Verified Computation of Simulation Relations on NFAs

Effiziente und verifizierte Berechnung von Simulationsrelationen auf NFAs

Author:	Manuel Eberl
Supervisor:	Prof Tobias Nipkow, PhD
Advisors:	Dr Peter Lammich
	Dr Thomas Tuerk
Date:	October 15, 2012



I assure the single-handed composition of this bachelor's thesis only supported by declared resources.

Garching, 10 October 2012 Manuel Eberl

Abstract

In this thesis, the algorithm presented by Ilie, Navarro, and Yu for computing simulation relations on *Nondeterministic Finite Automata* [INY04] is implemented and verified with the interactive theorem prover *Isabelle* using Peter Lammich's *Monadic Refinement Framework*. This is done by writing an abstract version of the algorithm and proving it correct and then successively replacing parts of it with more concrete commands, proving the correctness of each modification, until an executable algorithm is obtained. This code can then be exported to programming languages such as ML, OCaml, Haskell or Scala.

Zusammenfassung

In dieser Arbeit wird der Algorithmus von Ilie, Navarro und Yu zur Berechnung von Simulationsrelationen auf *Nichtdeterministischen Endlichen Automaten* [INY04] implementiert und mit dem interaktiven Theorembeweiser *Isabelle* unter Verwendung von Peter Lammichs *Monadischem Refinement-Framework* verifiziert. Der Beweis erfolgt über mehrere Stufen, indem zuerst eine abstrakte Version des Algorithmus geschrieben und als korrekt bewiesen wird und dann nach und nach Teile davon durch konkreteren Code ersetzt werden, wobei die Korrektheit jedes Schrittes bewiesen wird, bis ausführbarer Code entsteht. Dieser kann dann in Programmiersprachen wie ML, OCaml, Haskell oder Scala exportiert werden.

Contents

Acknowledgements

I would like to thank my advisors, Drs Peter Lammich and Thomas Tuerk, for their excellent support whenever I got stuck, and my supervisor Prof Tobias Nipkow for providing me with this most instructive topic for my thesis.

My thanks also go to Nikolai Wyderka and my friends in the #in.tum IRC channel, who were good enough to proofread this thesis or parts thereof and point out numerous typos; and to Prof Javier Esparza, who, with his expertise on automata, helped me clear up some of the confusion I accumulated whilst working through the literature on this field.

1 Introduction

In [INY04], Ilie, Navarro, and Yu give an efficient algorithm for computing simulation preorders on *Nondeterministic Finite Automata* (NFAs) and explain how to use these for reduction of NFAs. This algorithm is based on an earlier algorithm for simulation preorders on node-labelled graphs presented in [HHK95] by Henzinger et al. The topic of this thesis is to implement this algorithm as a functional programme with the interactive theorem prover Isabelle in the context of an existing formalisation of NFAs and to develop a formal correctness proof for this implementation. The resulting algorithm could then, for instance, be used to reduce the size of NFAs that are used in regular expression algorithms. Also, the same algorithm can be applied to Büchi automata, which are used in model checking.

1.1 Utilised tools

For the formal verification, the interactive theorem prover Isabelle was used, in particular Isabelle/HOL, i. e. Isabelle with Higher Order Logic. The algorithm was formalised in a monadic, functional style using the Refinement Framework by Peter Lammich (described in depth in [LT12]) and uses the finite automata implementation by Thomas Tuerk [Tue12].

The Refinement Framework provides methods for writing algorithms with precisely defined semantics in an intuitive way and proving properties of these algorithms. In particular, it allows the user to write and verify algorithms on an abstract level at first and then *refine* them successively, by replacing parts of the code with optimised or more concrete statements until an efficient, executable version is obtained, which can then be exported to other languages using Isabelle's code generator. This breaks down the proof obligations for the verification of the algorithm into smaller steps and makes correctness proofs much more modular and readable.

The Refinement Framework is still in an early stage of development and this thesis also serves as an evaluation of the current state of the framework with regard to usability and bugs.

1.2 Outline

In Section 2, we will introduce a number of definitions related to automata and simulation relations. In particular, we will introduce the notion of the *simulation preorder* of an NFA and demonstrate some of its properties and how it can be used

in order to reduce NFAs. In Section 3, we will then explain the algorithm that was implemented and verified in this thesis, a slightly modified version of the algorithm by Ilie et al. We will also describe how correctness proofs using refinement work and how the individual steps were performed in the process of verifying this particular algorithm. Reoccuring patterns (such as refining a set comprehension to a **for each** loop) are explained only once and unnecessary details are left out; for the detailed proof, refer to the commented Isabelle proof.

In Section 4, the verified algorithm, exported to ML, is tested on a small example automaton and the performance of the algorithm is compared to that of a similar Java implementation.

In the conclusion, we discuss the advantages and disadvantages of the Refinement Framework and the problems that were encountered with it in the process of writing this thesis. We also make a number of suggestions how these problems could be mitigated. Furthermore, we discuss possible improvements of the algorithm and what applications it could have in the context of the CAVA project.

2 Theory

2.1 Finite automata

A *Nondeterministic Finite Automaton* (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta : Q \times \Sigma \mapsto \mathcal{P}(Q)$ is a transition function, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states. Furthermore, we define the transition relation $\Delta \subseteq Q \times \Sigma \times Q$ where $(q, c, q') \in \Delta$ iff $q' \in \delta(q, c)$ and the reverse transition function $\delta^{-1}(q, c) = \{q' |\ (q', c, q) \in \Delta\}$.

The state $q \in Q$ is said to accept ε, the empty word, iff $q \in F$, and q is said to accept a non-empty word cw with $c \in \Sigma$ and $w \in \Sigma^*$ iff there is a successor state $q' \in \delta(q, c)$ that accepts w. The set of all $w \in \Sigma^*$ that a state $q \in Q$ accepts is called the (right) language of the state and denoted by $\mathcal{L}_\mathcal{A}(q)$ or $\vec{\mathcal{L}}_\mathcal{A}(q)$. If the automaton $\mathcal{A}$ is obvious from the context, we will sometimes omit it and simply write $\mathcal{L}(q)$ or $\vec{\mathcal{L}}(q)$. The language of the automaton $\mathcal{A}$ (denoted by $\mathcal{L}(\mathcal{A})$) is defined as the union of all $\mathcal{L}_\mathcal{A}(q)$ for $q \in I$. Furthermore, we define $\mathcal{L}_\mathcal{A}(u, v)$ as the set of all words w such that there is a path from u to v with the word w, i. e. $\mathcal{L}_\mathcal{A}(u, v)$ is the language of the automaton obtained by taking $\mathcal{A}$, but using $I = \{u\}$ and $F = \{v\}$.

Similarly, the left language $\overleftarrow{\mathcal{L}}_\mathcal{A}(q)$ is defined as the set of all $w \in \Sigma^*$ such that either $w = \epsilon$ and $q \in I$ or $w = w'c$ and there is some $q' \in \delta^{-1}(q, c)$ such that $w' \in \overleftarrow{\mathcal{L}}_\mathcal{A}(q')$.

The reverse automaton of $\mathcal{A}$ is denoted as $\mathcal{A}^{-1}$ and defined as $(Q, \Sigma, \delta^{-1}, F, I)$. It can be seen quite easily that $\mathcal{L}(\mathcal{A}^{-1}) = \mathcal{L}(\mathcal{A})^R = \{w_n \dots w_1 |\ w_1 \dots w_n \in \mathcal{L}(\mathcal{A})\}$. Obviously, we have $(\mathcal{A}^{-1})^{-1} = \mathcal{A}$.

Note that $\overleftarrow{\mathcal{L}}_\mathcal{A}(q) = \left(\vec{\mathcal{L}}_{\mathcal{A}^{-1}}(q)\right)^R$ and $\vec{\mathcal{L}}_\mathcal{A}(q) = \left(\overleftarrow{\mathcal{L}}_{\mathcal{A}^{-1}}(q)\right)^R$.

2.2 Simulation relations and preorders

All the following definitions are valid in the context of an NFA $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$.

We will now define the concept of simulation relation. A simulation relation is a relation $\mathcal{S} \subseteq Q \times Q$ that fulfils the following two conditions for all $(u, v) \in \mathcal{S}$:

1. if u is a final state, v is also a final state
2. if u can make a transition to u' with the character c, v can make a transition to some state v' with the character c such that $(u', v') \in \mathcal{S}$.

Intuitively, $(u, v) \in \mathcal{S}$ means that every path that can be taken from u can be matched with a "similar" path from v, where every state v_i along the path from v simulates the corresponding state u_i (i. e. $(u_i, v_i) \in \mathcal{S}$) along the path from u.

More formally, we call $\mathcal{S} \subseteq Q \times Q$ a simulation relation iff for all $u, v \in Q$ the following holds:

$$(u, v) \in \mathcal{S} \quad \Longrightarrow \quad (u \in F \Rightarrow v \in F) \ \wedge \\ (\forall c \in \Sigma.\, \forall u' \in \delta(u, c).\, \exists v' \in \delta(v, c).\, (u', v') \in \mathcal{S})$$

$\mathrm{sim}(\mathcal{A})$ denotes the set of all simulation relations of $\mathcal{A}$. Furthermore, we say "u is simulated by v" or "v simulates u" for states $u, v \in Q$ iff there is a simulation $\mathcal{S} \in \mathrm{sim}(\mathcal{A})$ such that $(u, v) \in \mathcal{S}$ and introduce the shorthand $u \leq_{\mathcal{A}} v$ for "u is simulated by v". Since $\leq_{\mathcal{A}}$ is reflexive and transitive (see next section), $\leq_{\mathcal{A}}$ is a preorder, which we call the simulation preorder.

Because of the recursion in the second condition in our definition of a simulation relation, there are several valid simulation relations for a given NFA in most cases. In particular, the empty relation $\emptyset$ is a simulation relation for all NFAs, albeit a trivial and therefore uninteresting one. However, it can be shown that for every NFA $\mathcal{A}$, there exists a largest simulation relation, which we call $\mathcal{S}_{\mathcal{A}}$, with the following properties:

1. $\mathcal{S}_{\mathcal{A}} \in \mathrm{sim}(\mathcal{A})$
2. $\forall \mathcal{S} \in \mathrm{sim}(\mathcal{A}).\, \mathcal{S} \subseteq \mathcal{S}_{\mathcal{A}}$

The uniqueness of such an $\mathcal{S}_{\mathcal{A}}$ is obvious, since $\mathcal{S}_{\mathcal{A}} \subseteq \mathcal{S}'_{\mathcal{A}}$ and $\mathcal{S}'_{\mathcal{A}} \subseteq \mathcal{S}_{\mathcal{A}}$ implies $\mathcal{S}_{\mathcal{A}} = \mathcal{S}'_{\mathcal{A}}$. The existence is also obvious, since $\mathcal{S}_{\mathcal{A}}$ can be constructed as $\mathcal{S}_{\mathcal{A}} = \bigcup_{\mathcal{S} \in \mathrm{sim}(\mathcal{A})} \mathcal{S}$.

Let us now look at how the simulation preoder $\leq$ and the largest simulation $\mathcal{S}_{\mathcal{A}}$ are related. Consider a pair of arbitrary states u and v. If $(u, v) \in \mathcal{S}_{\mathcal{A}}$, then $u \leq_{\mathcal{A}} v$ obviously holds by definition. Conversely, if $u \leq_{\mathcal{A}} v$, there is a simulation relation $\mathcal{S}$ that contains (u, v), and since $\mathcal{S}_{\mathcal{A}}$ contains all simulation relations, (u, v) is then also contained in $\mathcal{S}_{\mathcal{A}}$. Hence, $(u, v) \in \mathcal{S}_{\mathcal{A}}$ and $u \leq_{\mathcal{A}} v$ imply each other and are therefore equivalent, i. e. $\leq_{\mathcal{A}} = \mathcal{S}_{\mathcal{A}}$

Now we shall take a look at the consequences of simulation. It can easily be seen that $u \leq_{\mathcal{A}} v$ implies $\mathcal{L}(u) \subseteq \mathcal{L}(v)$: if $w \in \mathcal{L}(u)$, there is some path from u to a final state with the word w, and because v simulates u, we can then always find an equivalent path from v to a final state (proof by induction over word length, see Isabelle proof).

However, the converse does not hold in the general case: consider an arbitrary non-empty word $cw \in \mathcal{L}(u)$. If v is to simulate u, there must be one successor state $v' \in \delta(v, c)$ that fulfils all the criteria for simulation, independently of what w looks like. For cw merely to be accepted by v, on the other hand, the automaton may take separate routes from v, depending on w. Therefore, simulation is strictly stronger than language inclusion. This is also illustrated by Figure 1.

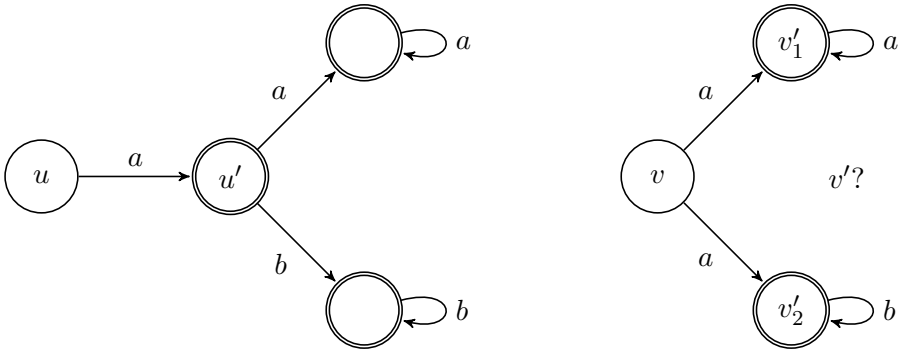


Figure 1: An automaton that illustrates that $\mathcal{L}(u) \subseteq \mathcal{L}(v)$ does not necessarily imply $u \leq_{\mathcal{A}} v$. In this automaton, $\mathcal{L}(u) = \mathcal{L}(a(a^*|b^*)) = \mathcal{L}(v)$, but neither v'_1 nor v'_2 simulate u', which implies $u \not\leq v$

In a *Deterministic Finite Automaton* (DFA), that is an NFA in which $|I| = 1$ and $\forall q\, c.\ |\delta(q, c)| = 1$, this situation does not occur – the simulation preorder and right language inclusion are equivalent for DFAs.

2.3 Properties of the simulation preorder

2.3.1 Reflexivity

Obviously, any state in an automaton can simulate itself, i. e. for all $u \in \mathcal{Q}$, we have $(u, u) \in \mathcal{S}_\mathcal{A}$ and, equivalently, $u \leq_\mathcal{A} u$. This can be shown easily by showing that $\{(u, u) \mid u \in Q\}$ is a simulation:

$$\begin{aligned} u = v \quad \Longrightarrow \quad & (u \in F \Rightarrow v \in F) \ \wedge \\ & (\forall c \in \Sigma.\ \forall u' \in \delta(u, c).\ \exists v' \in \delta(v, c).\ u' = v') \end{aligned}$$

Both conditions are trivially fulfilled. Therefore, $\{(u, u) \mid u \in Q\}$ is a simulation, which implies that it is contained in $\mathcal{S}_\mathcal{A}$, which implies that $\mathcal{S}_\mathcal{A}$ is reflexive.

2.3.2 Transitivity

The proof for transitivity requires reasoning about the transitive closure $\mathcal{S}_\mathcal{A}^+$ of $\mathcal{S}_\mathcal{A}$. It can be shown that $\mathcal{S}_\mathcal{A}^+$ is equal to $\mathcal{S}_\mathcal{A}$ by showing that $\mathcal{S}_\mathcal{A}^+$ is a simulation itself.

The condition that needs to be verified for $\mathcal{S}_\mathcal{A}^+$ to be a simulation is:

$$\begin{aligned} (u, w) \in \mathcal{S}_\mathcal{A}^+ \quad \Longrightarrow \quad & (u \in F \Rightarrow w \in F) \ \wedge \\ & (\forall c \in \Sigma.\ \forall u' \in \delta(u, c).\ \exists w' \in \delta(w, c).\ (u', w') \in \mathcal{S}_\mathcal{A}^+) \end{aligned}$$

This can be shown quite easily by induction over the assumption $(u, w) \in \mathcal{S}_\mathcal{A}^+$, using the inductive definition of the transitive closure. For details, refer to the formal proof in Isabelle.

2.4 Reduction of NFAs using simulation preorders

In [INY04] and [CC03], three criteria related to simulation preorders for reducing NFAs are given. Two states $u, v \in Q$ can be merged if one of the following holds:

1. $u \leq_\mathcal{A} v$ and $v \leq_\mathcal{A} u$
2. $u \leq_{\mathcal{A}^{-1}} v$ and $v \leq_{\mathcal{A}^{-1}} u$
3. $u \leq_\mathcal{A} v$ and $u \leq_{\mathcal{A}^{-1}} v$ and $\mathcal{L}(u, u) = \{\varepsilon\}$

The reason for this is that for arbitrary $p, q \in Q$, the property $p \leq_{\mathcal{A}} q$ implies $\vec{\mathcal{L}}(p) \subseteq \vec{\mathcal{L}}(q)$ and $p \leq_{\mathcal{A}^{-1}} q$ implies $\overleftarrow{\mathcal{L}}(p) \subseteq \overleftarrow{\mathcal{L}}(q)$. Therefore, each of the three properties above implies the corresponding following one:

1. $\vec{\mathcal{L}}(u) = \vec{\mathcal{L}}(v)$
2. $\overleftarrow{\mathcal{L}}(u) = \overleftarrow{\mathcal{L}}(v)$
3. $\vec{\mathcal{L}}(u) \subseteq \vec{\mathcal{L}}(v)$ and $\overleftarrow{\mathcal{L}}(u) \subseteq \overleftarrow{\mathcal{L}}(v)$ and $\mathcal{L}(u, u) = \{\varepsilon\}$

The intuitive explanations for why states can be merged if one of these properties applies are:

1. Once the automaton is in u or v (by reading some word w_1), it will accept w_2 if it is in u iff it accepts w_2 when it is in v. Therefore, it does not matter whether the automaton is in u or in v and we may simply contract the two states, since they behave the same.
2. Since $\overleftarrow{\mathcal{L}}(v) = \vec{\mathcal{L}}_{\mathcal{A}^{-1}}(v)$, we have right language equality of u and v in $\mathcal{A}^{-1}$. This means that by property 1, we can merge u and v in $\mathcal{A}^{-1}$, and this implies that we can merge them in $\mathcal{A}$ as well.
3. For each path that leads from an initial state to u, there is a path from an initial state to v since $\overleftarrow{\mathcal{L}}(u) \subseteq \overleftarrow{\mathcal{L}}(v)$. Also, for each path that leads from u to a final state, there is a path from v to a final state since $\vec{\mathcal{L}}(u) \subseteq \vec{\mathcal{L}}(v)$. Therefore, the state u is completely subsumed by the state v and we can therefore simply delete u – or, equivalently, merge it with v. The condition $\mathcal{L}(u, u) = \{\varepsilon\}$ means that u may not be non-trivially reachable from itself, i. e. it may only occur once on any path. This is necessary for the reduction to work [CC05].

However, it is crucial to be aware that merging states according to one of these properties can interfere with another property. If one merges two states u and v to some state z due to property 1, the $\leq_{\mathcal{A}^{-1}}$ preorder can change and if one merges them due to property 2, the $\leq_{\mathcal{A}}$ preorder may change.[1] This, of course, means that merging states due to one property may reduce one's possibilities to merge states in later steps. Therefore, some merging strategies can lead to smaller automata than others. In [ISoY05], it is shown that the problem of using these preorders to reduce an NFA optimally is in fact NP-hard.

[1] By *change* we mean that the left and right languages of a new state z may not have a certain property w. r. t. another state even though the left and right languages of u or v did.

3 The algorithm and the refinement process

3.1 Using the Refinement Framework

Before explaining our correctness proof, we will give a short introduction to how refinement in general and with the Refinement Framework in particular works. For a detailed introduction, see [LT12].

Verification by refinement works by first formalising the algorithm on a mathematical level, i. e. working with sets, set comprehensions and so on. Having proven this abstract algorithm correct, the next step is to successively replace the abstract statements with more concrete computations, e. g. set comprehensions are replaced by loops that compute the set. Optimisations such as caching can also be performed in this way; for instance, a complex expression may be replaced with a cache lookup. In each step, the refinement has to be proven correct, i. e. one has to prove that the refined algorithm still does (roughly) the same.[2]

The advantage of this approach is that most of the interesting part of the correctness proof is done on a very abstract level and the optimisations and implementation details can be handled separately, after the correctness of the basic algorithm has already been shown. If one were to prove the entire algorithm without these refinement steps, the proof would be very lengthy and difficult to understand and maintain.

The verification and refinement process is supported by the Refinement Framework with a number of tools, such as a nondeterministic[3], monadic programming language in which abstract algorithms can be implemented easily, a *verification condition generator* (VCG) to prove the correctness of algorithms in this language, a *refinement condition generator* (RCG) to guide the refinement process by extracting the relevant proof obligations that arise from replacing parts of the algorithm and a number of other tools that automate parts of the process.

[2] *roughly the same* means that, due to nondeterminism, the refined algorithm may have only a subset of the possible results of the original algorithm, and also, the result may have a different "format" if data refinement is used. (see 3.4.1)

[3] The reason why nondeterminism is often required even though the resulting algorithms are, of course, deterministic, is that on the abstract level, one often has operations such as *return some arbitrary element of the set A* and can therefore have many possible results – or indeed none. The notion *programme 1 refines programme 2* then means that programme 1 returns a (not necessarily proper) subset of the possible results of programme 2. Ideally, in the end one has an executable programme that has exactly one result and that refines the set of all valid results.

3.2 The abstract algorithm

In this section, we will now define our modified version of the algorithm given in [INY04] and explain, on an intuitive level, why it is correct.

The idea behind computing simulation preorders is that nonsimulatability[4] of pairs of states propagates backwards in the automaton: for one state v to simulate another state u, we need to be able to match every step from u with an equivalent step from v, i. e. if we know that for some $u' \in \delta(u, c)$ there is no $v' \in \delta(v, c)$ which simulates u', we can conclude that v cannot simulate u.

We can therefore start with an initial, coarse[5] estimate of our simulation preorder $\mathcal{S}_\mathcal{A}$ and successively remove pairs (u, v) that violate the simulation condition[6] until there are no such pairs anymore.

As the initial estimate, we take the set of all (u, v) with $u \in F \Rightarrow v \in F$, which is the largest set of state pairs that fulfil the first property of a simulation. The following piece of pseudocode, which is similar to the abstract algorithm for simulation preorders on node-labelled graphs given by Henzinger et al in [HHK95], illustrates this principle of successively removing pairs that violate the second condition:

Algorithm 1 compute_$\mathcal{S}_\mathcal{A}$_naive($\mathcal{A}$)

```
1 S := (Q \ F) × Q ∪ F × F
2 while ∃(u, v) ∈ S. ∃c ∈ Σ. ∃u' ∈ δ(u, c). ∄v' ∈ δ(v, c). (u', v') ∈ S do
3    obtain such a (u, v)
4    S := S \ {(u, v)}
5 end while
```

However, this algorithm is rather inefficient. In each iteration, it traverses the entire set $\mathcal{S}$, looking for a violating pair. Ilie et al. use several modifications to make the algorithm more efficient. First of all, they do not compute $\leq$ directly, but compute $\not\leq$ instead, successively adding pairs of states. The incomplete $\not\leq$ relation is called ω in the algorithm.

Furthermore, they introduce an additional set $\mathcal{C}$ which stores all those state pairs in ω that have not been processed yet, i. e. the (u', v') of which we know that v' does not simulate u', but we have not propagated that information backwards to the

[4] For lack of a better word. What we mean by *nonsimulatability* of a pair (u, v) is that (u, v) cannot be in a simulation, i. e. $u \not\leq v$.

[5] *coarse* meaning all the pairs that are in $\mathcal{S}_\mathcal{A}$ are in our relation, but there may also be pairs in our estimate that are not in $\mathcal{S}_\mathcal{A}$.

[6] the two properties in the definition of a simulation relation, although we can take care of the $u \in F \Rightarrow v \in F$ condition in our initial estimate so that we only have to check for pairs violating the second condition

predecessors of u' and v'. This way, we do not have to look for violating pairs in each iteration but can simply take a (u', v') from $\mathcal{C}$ and propagate its nonsimulatability, adding to ω and $\mathcal{C}$ all (u, v) for which v cannot simulate u any longer due to this new information.

Algorithm 2 INY_basic($\mathcal{A}$)

```
1  ω := F × (Q \ F) ∪ {(u, v) | ∃c. δ(u, c) ≠ ∅ ∧ δ(v, c) = ∅}
2  C := ω
3  while C ≠ ∅ do
4     obtain (u', v') ∈ C
5     C := C \ {(u, v)}
6     T := {(u, v) | c ∈ Σ, v ∈ δ^-1(v', c), u ∈ δ^-1(u', c).
                          (u, v) ∉ ω ∧ ∀v'' ∈ δ(v, c). (u', v'') ∈ ω \ C} [7] [8]
7     ω := ω ∪ T
8     C := C ∪ T
9  end while
10 return ω
```

For given u' and v', the set T is the set of all pairs $(u, v) \notin \omega$ where u is a predecessor of u' w. r. t. $c \in \Sigma$ and v is a predecessor of v' w. r. t. some $c \in \Sigma$ so that v can no longer simulate u. It is the set of all pairs that are affected by the fact that v' cannot be simulated by u'. Therefore, we have to add T to ω and $\mathcal{C}$.

Note that the second part of the initialisation of ω and $\mathcal{C}$ is crucial. Ilie et al. have chosen $F \times (Q \setminus F)$ as their initial value for ω and $\mathcal{C}$. This is a mistake, since it fails to recognise the nonsimulatability of states u, v where there is a $u' \in \delta(u, c)$, but $\delta(v, c) = \emptyset$, meaning a step from u to u' with the character c cannot be matched with any step from v at all. A simple example would be the following automaton:

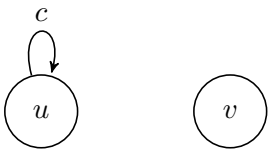


Figure 2: An example for an automaton where $u \not\leq v$ for which the algorithm given by Ilie et al. erroneously computes $\omega = \emptyset$ and therefore $(u, v) \notin \omega$.

This, of course, can only be the case in non-total automata, the algorithm works as described in [INY04].

[7] Note that the syntax $\{t(a, b, c) \mid a \in A,\ b \in B,\ c \in C.\ \ P(a, b, c)\}$ means $\{t' \mid \exists a \in A,\ b \in B,\ c \in C.\ \ t' = t(a, b, c) \wedge P(a, b, c)\}$.

[8] You may wonder why it says $\omega \setminus \mathcal{C}$ and not ω. The reason for that is that with $\omega \setminus \mathcal{C}$, this expensive condition can later be optimised easily, but with ω, this would not be possible (see p. 17).

3.3 Verification

In order to prove the correctness of the algorithm, we first need to give a loop invariant and a termination proof for the **while** loop.

3.3.1 Termination

Termination can be proven by giving a well-founded relation over our variables $\omega, \mathcal{C}$ and showing that the $\omega', \mathcal{C}'$ obtained at the end of one iteration are smaller w. r. t. that relation than the original $\omega, \mathcal{C}$. A good choice for such a relation is a strict order based on the measure function $\phi(\omega, \mathcal{C}) = |(Q \times Q) \setminus (\omega \setminus \mathcal{C})|$. In each iteration, this measure decreases by 1, since we only add elements to both ω and $\mathcal{C}$ and remove exactly one element from $\mathcal{C}$.

3.3.2 Loop invariant

The loop invariant we chose contains the following five facts:

1. $\omega \subseteq Q \times Q$
2. $F \times (Q \setminus F) \subseteq \omega$
   (ω contains all (u, v) that violate simulation condition 1)
3. $\mathcal{C} \subseteq \omega$
4. $\omega \cap \mathcal{S}_{\mathcal{A}} = \emptyset$
   (i. e. ω never becomes too large)
5. $(u, v) \notin \omega \wedge u' \in \delta(u, c) \Longrightarrow \exists v' \in \delta(v, c).\ (u', v') \notin \omega \setminus \mathcal{C}$
   (i. e. after each iteration, ω is "large enough"; each $(u, v) \notin \omega$ must "justify" its absence from ω by providing a suitable successor v' to match any step from u to some u')

It can be seen quite easily that our initial values for ω and $\mathcal{C}$ fulfil these conditions. We will now proceed to prove the conservation of these properties after a single iteration of the loop.

Properties 1-3: The first three properties of the invariant are trivially preserved by an iteration

Property 4: In essence, we have to show that none of the pairs in T can be in a simulation relation. This is because the condition for simulating pairs is a direct contradiction to the definition of T.

Property 5: This reduces to showing that for any $(u, v) \in T$ and $u'' \in \delta(u, c)$, there is a $v'' \in \delta(v, c)$ such that $(u'', v'') \in \omega \setminus \mathcal{C}$ (which implies $(u'', v'') \neq (u', v')$, as (u', v') was just removed from $\mathcal{C}$).

This means that if the simulation of u by v relied on the simulation of u' by v' so far and (u, v) is still not in ω after the iteration, there has to be another suitable $v'' \in \delta(v, c)$ for any step from u to u' with c that previously relied on simulation of u' by v'. This is the case because of the definition of T: if there were no such (u'', v''), the pair (u, v) would be in T.

At this point, we have shown that the loop terminates and preserves the invariant. What remains to be shown is that if the invariant still holds for $\mathcal{C} = \emptyset$, this implies the correctness of the algorithm, i.e. $\mathcal{S}_\mathcal{A} = (Q \times Q) \setminus \omega$:

$\subseteq$**:** because $\omega \subseteq Q \times Q$ and $\omega \cap \mathcal{S}_\mathcal{A} = \emptyset$ (from invariant) and $\mathcal{S}_\mathcal{A} \subseteq Q \times Q$ (by definition)

$\supseteq$**:** because $(Q \times Q) \setminus \omega$ is a simulation and $\mathcal{S}_\mathcal{A}$ contains all simulation relations. The fact that it is a simulation is a direct consequence of invariant properties 1, 2, and 5, which correspond directly to the definition of a simulation relation.

Therefore, the algorithm correctly computes the correct value for ω. This concludes the essential part of the verification; what remains is the concretisation of the abstract operations in the algorithm, such as the set comprehensions and the $\forall$ condition.

3.3.3 Refinement

We will now refine the set comprehension in the **while** loop with nested **for each** loops. Recall that the set T was defined as:

$$T := \{(u, v) \mid c \in \Sigma,\ v \in \delta^{-1}(v', c),\ u \in \delta^{-1}(u', c).\\ (u, v) \notin \omega \wedge \forall v'' \in \delta(v, c).\ (u', v'') \in \omega \setminus \mathcal{C}\}$$

Each of the loops effectively removes one of the three quantified variables c, v, and u in the set comprehension by iterating over all possible values for the variable. We will proceed in this order, i.e. the outermost loop iterates over all $c \in \Sigma$, the next over all $v \in \delta^{-1}(v', c)$ and so on. The conditions $(u, v) \notin \omega$ and $\forall v'' \in \delta(v, c).\ (u', v'') \in \omega \setminus \mathcal{C}$ will be transformed to **if** statements in the loops. Since the second condition depends only on v and c, but not on u, the corresponding **if** statement can be moved outwards to the loop over all v.

A subtle complication in the refinement of the algorithm is that it works with nested loops over all $c \in \Sigma$, all $v \in \delta^{-1}(v', c)$ and so on, updating ω and $\mathcal{C}$ with every iteration. Therefore, later loop iterations do not operate on the original ω – they already work with an updated version of ω that may contain more elements than the original ω. However, since any elements added to ω are also added to $\mathcal{C}$, the set $\omega \setminus \mathcal{C}$ stays invariant, which means that the changing ω and $\mathcal{C}$ do not change the overall result, as the loop iterations only work with $\omega \setminus \mathcal{C}$.

It became apparent that due to complications with the nested loops and changing ω and $\mathcal{C}$, it is easier to use not $\omega' = \omega \cup T$ and $\mathcal{C}' = \mathcal{C} \cup T$, but a generalised version of this. In this generalised condition, we allow an arbitrary set T' to be added to ω and $\mathcal{C}$ as long as it has the following two properties:

1. $T \subseteq T'$ (T' must be *large enough*)
2. $T' \cap \mathcal{S}_\mathcal{A} = \emptyset$ (T' must not be *too large*)

The abstract algorithm with the modified **obtain** statement looks like this:

Algorithm 3 INY_abstract1($\mathcal{A}$)

```
1  ω := F × (Q \ F) ∪ {(u, v) | ∃c. δ(u, c) ≠ ∅ ∧ δ(v, c) = ∅}
2  C := ω
3  while C ≠ ∅ do
4     obtain (u', v') ∈ C
5     C := C \ {(u', v')}
6     obtain (ω', C') ∈ {(ω', C') | T' = ω' − ω = C' − C and T' is valid}
7     ω := ω'
8     C := C'
9  end while
10 return ω
```

T' is valid $\iff$ $T \subseteq T'$ and $T' \cap \mathcal{S}_\mathcal{A} = \emptyset$

(where T is defined as before)

For our refinement, we now have to replace the statement in line 6 with a **for each** loop that iterates over all $c \in \Sigma$ and computes such a pair $(\omega', \mathcal{C}')$. This loop will contain a similar **obtain** statement, but without the quantification over c. We need to replace this statement with another for loop that iterates over all $v \in \delta^{-1}(v', c)$ and so on, until no **obtain** statements remain.

The three loops look like this:

Algorithm 4 INY_abstract2_loopc($\mathcal{A}, \omega, \mathcal{C}, u', v'$)

```
1 for each c ∈ Σ do
2   obtain (ω', C') ∈ {(ω', C') | T'_1 = ω' − ω = C' − C and T'_1 is valid}
3   ω := ω'
4   C := C'
5 end for
6 return (ω, C)
```

$$T'_1 \text{ is valid} \iff T_1(c) \subseteq T'_1 \text{ and } T'_1 \cap \mathcal{S}_\mathcal{A} = \emptyset$$
$$\text{where } T_1(c) = \{(u,v) \mid v \in \delta^{-1}(v', c),\ u \in \delta^{-1}(u', c).\ (u,v) \notin \omega \wedge \forall v'' \in \delta(v,c).\ (u', v'') \in \omega \setminus \mathcal{C}\}$$

Algorithm 5 INY_abstract3_loopv($\mathcal{A}, \omega, \mathcal{C}, u', v', c$)

```
1 for each v ∈ δ^{-1}(v', c) do
2   if ∀v'' ∈ δ(v, c). (u', v'') ∈ ω \ C then
3     obtain (ω', C') ∈ {(ω', C') | T'_2 = ω' − ω = C' − C and T'_2 is valid}
4     ω := ω'
5     C := C'
6   end if
7 end for
8 return (ω, C)
```

$$T'_2 \text{ is valid} \iff T_2(c,v) \subseteq T'_2 \text{ and } T'_2 \cap \mathcal{S}_\mathcal{A} = \emptyset$$
$$\text{where } T_2(c,v) = \{(u,v) \mid u \in \delta^{-1}(u', c).\ (u,v) \notin \omega\}$$

Algorithm 6 INY_abstract4_loopu($\mathcal{A}, \omega, \mathcal{C}, u', v', c, v$)

```
1 for each u ∈ δ^{-1}(u', c) do
2   if (u, v) ∉ ω then
3     ω := ω ∪ {(u, v)}
4     C := C ∪ {(u, v)}
5   end if
6 end for
7 return (ω, C)
```

After successively expanding all the **obtain** statements in the algorithm with the corresponding loops defined above, we get the following pseudocode:

Algorithm 7 INY_abstract4($\mathcal{A}$)

```
1  ω := F × (Q \ F) ∪ {(u, v) | ∃c. δ(u, c) ≠ ∅ ∧ δ(v, c) = ∅}
2  C := ω
3  while C ≠ ∅ do
4     obtain (u', v') ∈ C
5     C := C \ {(u', v')}
6     for each c ∈ Σ do
7        for each v ∈ δ^-1(v', c) do
8           if ∀v'' ∈ δ(v, c). (u', v'') ∈ ω \ C then
9              for each u ∈ δ^-1(u', c) do
10                if (u, v) ∉ ω then
11                   ω := ω ∪ {(u, v)}
12                   C := C ∪ {(u, v)}
13                end if
14             end for
15          end if
16       end for
17    end for
18 end while
19 return ω
```

In summary: in each iteration of the **while** loop, one state pair (u', v') is removed from $\mathcal{C}$ and the nonsimulatability of (u', v') is propagated to the predecessors of u' and v', i. e. for all predecessors u of u' and v of v' (w. r. t. to some $c \in \Sigma$), we check whether (u, v) is affected by the fact that u' cannot be simulated by v'. If, taking this new information into account, there is no successor v'' of v left that can simulate u', we now know that v cannot simulate u either, since u can make a step to u' with c, but v cannot match this step. Therefore, (u, v) must be added to ω and $\mathcal{C}$.

Note that the condition of the first conditional, which contains a $\forall$, was not expanded to a **for each** loop. The reason for this is that there is a less expensive way to evaluate this condition without iterating over the entire set $\delta(v, c)$: we introduce a counter $N(c, u', v)$ that keeps track of the number of the $v'' \in \delta(v, c)$ that are not in $\omega \setminus \mathcal{C}$, i. e. the successors of v w. r. t. c that may – according to the knowledge at that point of the computation – be used to simulate u', or which are already known not to simulate u' but where this information has not been propagated to the predecessors yet.

In the beginning, this number is equal to $|\delta(v, c)|$ for each triple c, u', v, since $\omega = \mathcal{C}$. Intuitively: in the beginning, we have not propagated any information, so all successors of v w. r. t. c fulfil the conditions stated in the previous paragraph, which is why $N(c, u', v)$ is simply the number of all successors of v w. r. t. c. During the execution of the algorithm, counter values will decrease due to backpropagation of nonsimulatability. The minimum value is, of course, 0. When a counter value reaches 0, this means no successor of v can be used to simulate u', which, of course, means that v cannot simulate any predecessor w. r. t. c of u'. We can therefore replace the condition in the conditional with $N(c, u', v) = 0$, but also need to decrease $N(c, u', v)$ for all $c \in \Sigma$, $v \in \delta^{-1}(v', c)$ whenever we remove some (u', v') from $\mathcal{C}$, since this means that another successor of v w. r. t. c is now known to be unable to simulate u'.

For efficiency, we also store $|\delta(v, c)|$ for all v and c in a counter called d (these values are required in the initialisation of ω, $\mathcal{C}$ and N) and we store $\delta^{-1}(v, c)$ for all v and c in a map δ^r.

In [INY04], N increases from 0 to a maximum of $|\delta(v, c)|$. We chose to implement a decreasing counter instead in order to eliminate the lookups in the map d. This has led to an increase in performance of roughly five per cent compared to the increasing counter version of the algorithm that was first implemented.

The optimised algorithm using N, d, and δ^r looks like this:

Algorithm 8 INY_abstract5($\mathcal{A}$)

```
1  for each v, c do d(v, c) := |δ(v, c)|
2  for each v, c: do δ^r(v, c) := δ^{-1}(v, c)
3  for each c, u', v: do N(c, u', v) := d(v, c)
4  ω := F × (Q \ F) ∪ {(u, v) | ∃c. d(u, c) ≠ 0 ∧ d(v, c) = 0}
5  C := ω
6  while C ≠ ∅ do
7     obtain (u', v') ∈ C
8     C := C \ {(u', v')}
9     for each c ∈ Σ do
10       for each v ∈ δ^{-1}(v', c) do
11          N(c, u', v) := N(c, u', v) − 1
12          if N(c, u', v) = 0 then
13             for each u ∈ δ^{-1}(u', c) do
14                if (u, v) ∉ ω then
15                   ω := ω ∪ {(u, v)}
16                   C := C ∪ {(u, v)}
17                end if
18             end for
19          end if
20       end for
21    end for
22 end while
23 return ω
```

The algorithm still contains a set comprehension in the initialisation. However, since the implementation is trivial – two nested **for each** loops with conditionals that test whether $d(u)$ (resp. $d(v)$) are non-zero (resp. zero) – we will not write it out in full. In fact, the initialisation of N, d, and δ^r is also done somewhat differently in the algorithm that was defined and verified in Isabelle for reasons of efficiency, and it is implemented in a separate refinement step, but these are minor details, which is why we will not describe them in detail here.

3.3.4 Refinement proofs

The correctness proofs for these refinements are relatively straightforward: with each consecutive replacement of a set comprehension with a **for each** loop, the invariant is generalised; for instance, in the loop over all $c \in \Sigma$ the invariant states that the nonsimulatability of (u', v') has been propagated to all predecessors w. r. t.

some $c \in \Sigma \setminus \Sigma'$, where Σ' is the set of characters left in the iteration. For technical reasons, the information $(u', v') \in \mathcal{C}$ is also carried in the invariants. For $\Sigma' = \Sigma$, the new invariant is directly implied by the invariant above it, and conversely, for $\Sigma' = \emptyset$, the new invariant implies the invariant above it for the next iteration.

After three such refinement steps, the only set comprehensions left are the ones in the initialisation. The next step is to introduce the counter. This requires augmentation of all the loop invariants with the condition that

$$\forall c, u', v.\ \ N(c, u', v) = |\{v'' \in \delta(v, c) \mid (u', v') \notin \omega \setminus \mathcal{C}\}|$$

and similar generalisations of this for the inner loop invariants.

Another small number of refinement steps are then performed to introduce d and δ^r and to compute the initial values of ω, $\mathcal{C}$, N, d, and δ^r. After all of this, the algorithm would already be executable were it not for the lack of concrete data structures. At this stage, we still operate on abstract mathematical sets (ω, $\mathcal{C}$, ...) and (partial) functions (N, d, δ^r). In the next step of the refinement process, these need to be replaced with data types that the code generator can handle.

3.4 The concrete algorithm and code generation

3.4.1 Data refinement and concretisation

In general, *data refinement* is the process of replacing data types in an algorithm with other data types that refine the original ones. "Refine" means that throughout the execution of the algorithm, the original *abstract* value and the new *concrete* value must be in a *refinement relation*. This relation relates two values if the concrete value is an implementation of the abstract one. The refinement relation can often be written in the form $\{(s, s') \mid \mathit{invar}(s) \ \wedge \ s' = \alpha(s)\}$, where *invar* is some invariant for the concrete value and α is an abstraction function that transforms the concrete value to an abstract value. This is the most common case, in which the refinement relation is *single-valued*, i. e. any well-formed concrete value represents one (and only one) abstract value, but an abstract value may have several corresponding concrete values – or indeed none at all.

A simple example would be the representation of an set with a distinct list. The refinement relation then is $\{(s, s') \mid s \text{ is a distinct list } \wedge \ s' = \mathit{set}(s)\}$.[9] Note that the concrete values $[23, 42]$ and $[42, 23]$ correspond to the same abstract value $\{23, 42\}$,

[9] A distinct list is a list in which no element occurs more than once, $\mathit{set}(s)$ denotes the set of all values that occur in the list s.

whereas the concrete value $[42, 42]$ corresponds to no abstract value at all, since it does not fulfil the invariant, and the abstract value $\{1, 2, 3, \ldots\}$ does not have a corresponding concrete value either, since lists are always finite.

Another possible use for data refinement is optimisation, e. g. one might augment the abstract value with some kind of cache in order to avoid superfluous evaluations of an expensive expression. For example, the introduction of the counter N was data refinement: the abstract value is $(\omega, \mathcal{C})$, the concrete value is $(\omega', \mathcal{C}', N)$ and the two are related iff $\omega' = \omega$, $\mathcal{C}' = \mathcal{C}$ (the abstraction function) and N has the correct value for all c, u', v (the invariant for the concrete value).

Since the optimisations have already been performed, the only remaining data refinement is the replacement of abstract data types with concrete implementations. The remaining steps from here on are simple, but can become quite tedious, especially in relatively complex algorithms such as this. The three remaining steps that are necessary in order to obtain executable code for our algorithm with the Refinement Framework are:

1. Replacing the abstract mathematical data types (such as HOL sets and (partial) HOL functions) with generic *interfaces* for data structures provided by the Collection Framework [LL10] that offer the same functionality. (*StdSet* and *StdMap*, respectively) These correspond to abstract data structures such as the *Set* and *Map* interfaces in Java. The connection between the abstract HOL data types and the concrete Collection Framework data types is also done with an abstraction function α and an invariant *invar*. α takes a concrete value, e. g. an *StdSet* and returns the corresponding abstract HOL set. The invariant ensures that the data structure is *well-formed* (recall the earlier example of the distinct list).

   In order for the implementation to be correct, the operations on the concrete data types must satisfy certain correctness properties that state that the operations are compatible with the abstraction, e. g. for a well-formed value x, the abstract operation on $\alpha(x)$ yields the same result as the abstraction of the concrete operation on x and the result must again be well-formed. For the *set insertion* operation, for instance, the correctness property is:

$$\mathit{invar}(A) \implies \alpha(\mathit{set\_ins}(a, A)) = \alpha(A) \cup \{a\} \wedge \mathit{invar}(\mathit{set\_ins}(a, A))$$

2. Replacing **for each** loops with so-called iterators. These iterators are generalised versions of the fold functions in functional programming languages; an iterator is obtained by using an iterator function on a set data structure[10] A. The iterator is then a function that takes as arguments an initial state σ_0 (some value) and a function f that takes a state and an element of the set and produces a new state. The iterator then takes some element a_0 from A and computes $\sigma_1 = f(\sigma_0, a_0)$, takes another element a_1 from A and computes $\sigma_2 = f(\sigma_1, a_1)$ and so on. Basically, the iterator computes $f(\ldots f(f(\sigma_0, a_0), a_1) \ldots, a_{|A|-1})$. There are different iterator functions with different properties, but we will only require the most basic one, *set_iteratei*.[11]

3. Replacing the interfaces (e. g. *StdSet* and *StdMap*) with concrete implementations (e. g. red-black tree-based set, distinct list, and hash set (resp. red-black tree-based map and hash map). These concrete implementations correspond to collection classes like *TreeSet* and *HashSet* (resp. *TreeMap* and *HashMap*) in Java.

In order to perform these three steps, we first need some environment in which we have all the required iterator functions and data structures and the required operations on them. This is done by declaring a *locale* and assuming the existence of these operations. For example, our algorithm computes a relation as a result, i. e. a set of pairs of automaton states. As a HOL type, this would be written as a *'q×'q set*. Therefore, the algorithm requires some data type *'qq_set* that serves as a concrete implementation of a *'q×'q set*. In order to provide such a data structure, the locale fixes a type *'qq_set* and a set of operations *qq_set_ops :: "('q×'q, 'qq_set, _) set_ops_scheme"* and assumes its correctness, i. e. *StdSet qq_set_ops*.[12]

Within this locale, we can then reformulate statements such as $\omega'' \cup \{(u, v)\}$ as *qq_set.ins*$((u, v),\ \omega'')$.[13] [14] The correctness proof for this is mostly automatic, the conditions produced by the *refinement condition generator* can be proven immediately

[10] i. e. some implementation of *StdSet*

[11] For instance, there is also *set_iterateoi*, which returns an iterator that traverses the set in ascending order. *set_iteratei* does not provide any information about the order of traversal, it can be thought of as fully nondeterministic.

[12] *StdSet* is a locale provided by the Collection Framework that contains the correctness lemmas (see example with *set_ins* on previous page) for all the operations in the operation record.

[13] Note that the ω'' in the first expression is a HOL set of type *'q×'q set*, whereas the ω'' in the second expression is a Collection Framework set of type *'qq_set*.

[14] We use the mathematical notation $f(x, y)$ for function application here instead of the functional programming notation $f\ x\ y$ for reasons of consistency. In Isabelle, the latter is, of course, used.

with the correctness assumptions about the set/map operations. For instance, if we replace a $\emptyset$ in the algorithm with a *qq_set.empty* (), the RCG will generate a condition $(qq\_set\_ops.empty\ (), \emptyset) \in br\ qq\_set\_ops.\alpha\ qq\_set\_ops.invar$, i. e. *qq_set.empty* () returns a valid data structure and this data structure corresponds to an empty set.

The *refine_autoref* proof tactic provides automation for this entire process by replacing sets and maps and the operations on them with *StdSet* and *StdMap* and the corresponding operations and proving the correctness of these modifications automatically. However, it requires a rather complicated setup and tends to get stuck halfway through the process for subtle reasons that only a highly experienced user can pinpoint and fix. Therefore, only two refinement steps were performed with this method, with considerable assistance from Peter Lammich.

A minor complication in this particular algorithm is that we have nested data structures, namely the function δ^r, that returns a set. Therefore, we cannot simply refine this to an *StdMap* that maps to a HOL set, nor can we refine it to a HOL function that returns an *StdSet*, but we must refine it to an *StdMap* that returns an *StdSet*. In order to do this, we must also define a special invariant and abstraction function for this *StdMap*. This invariant needs to ensure that not only does the invariant of the map itself hold, but also the invariant of all the sets it contains; similarly, the abstraction does not only apply abstraction to the map, but also to the sets that are contained in it. It would be interesting to investigate the possibility of automating these nested cases in future versions of the framework.

Once we have eliminated all the abstract HOL datatypes and **for each** loops from the algorithm, we have a working and executable version that computes the complement of the simulation preorder. From this, we can easily compute the actual simulation preorder by subtracting ω from $Q \times Q$, which we do in a function called *compute_*$\mathcal{S}_\mathcal{A}$. This function is now defined and proven correct inside the locale, i. e. under the assumption that suitable data types and a valid NFA $\mathcal{A}$ exist. We can then do the next concretisation step in another locale by deriving an algorithm that works on a specific implementation of an NFA, namely *NFA_by_LTS*, which is an NFA that uses a so-called *labelled transition system* (LTS) internally. For convenience, we also collect all the required data structure operations in a record type. This locale can then be interpreted by giving such a record of concrete implementations for the data types, e. g. for *qq_ops*, we can use *rs_ops* for red-black trees or *lsi_ops* for distinct lists. We then have a completely executable algorithm and a theorem that states that, when given a valid NFA $\mathcal{A}$, the algorithm correctly computes its simulation preorder $\mathcal{S}_\mathcal{A}$.

The problem now is that we have an algorithm that is defined inside our locale and a lemma inside the locale that states that it always works. However, what we want is an algorithm that is defined outside of our locale (i. e. without any assumptions) and a lemma that states that this algorithm works if it is given a valid NFA as a parameter.[15] In order to do this, we have to make a number of boilerplate definitions, and in the end, we have a constant *rs_nfa_simulation_preorder* and the following corollary:

---

corollary rs_nfa_simulation_preorder_correct:

nfa_by_lts_defs.nfa_invar rs_ops rs_ops rs_lts_dlts_ops A_impl $\Longrightarrow$

ahs_invar (rs_nfa_simulation_preorder A_impl) $\wedge$

ahs_α (rs_nfa_simulation_preorder A_impl) $=$

NFA.$\mathcal{S}_\mathcal{A}$ (nfa_by_lts_defs.nfa_α rs_ops rs_ops rs_lts_dlts_ops A_impl)

The meaning of this corollary is: if *A_impl* encodes a valid NFA $\mathcal{A}$, the function *rs_nfa_simulation_preorder_correct* returns an Array Hash Set (note the *ahs_*α) that contains $\mathcal{S}_\mathcal{A}$, i. e. the algorithm is now proven correct. Using Isabelle's *export_code* command, we can now generate verified ML code for *compute_*$\mathcal{S}_\mathcal{A}$.

3.4.2 Complexity

In contrast to correctness, the complexity of the algorithm was not proven formally in this thesis due to the lack of a framework for such proofs. Instead, we will give a short, informal complexity analysis. In [INY04], it is stated that the complexity of the algorithm is $\mathcal{O}(mn)$ where m is the number of transitions in the automaton and n is the number of states. This is not immediately obvious from the four nested loops, and can be explained as follows:

The initialisation can obviously be performed in $\mathcal{O}(m + n^2)$. For the main loop, the complexity is not quite that obvious. Before we can analyse the main loop, we need the following lemma for the initial value of N:

$$\sum_{c,u',v} N(c,u',v) = \sum_{c,u',v} |\delta(v,c)| = n \sum_{c,v} |\delta(v,c)| = mn$$

This means that the sum of all entries in N initially is mn. This will be important in the following analysis. We will now consider blocks of commands in algorithm 8 on page 18 separately and determine upper bounds for how often they are executed.

[15] This assumption is, at this state, still part of the assumptions of the locale as well.

Lines 7–8: reached at most n^2 times, since every $(u', v') \in Q \times Q$ can be taken from $\mathcal{C}$ at most once.

Lines 11–12: reached at most mn times, since one entry in N decreases every time while the others remain the same, and the sum of all values in N is initially mn and always non-negative.

Line 13: reached at most once for a given triple c, u', v, since $N(c, u', v)$ can only become zero once. Thus, this line is reached at most n times for given c, u'.

Line 14–16: the number of iterations per loop for c, u' is $|\delta^{-1}(u', c)|$. Summed over all c, u', this is exactly m (because all transitions are of the form (u, c, u') for some $c \in \Sigma$, $u' \in Q$ and $u \in \delta^{-1}(u', c)$). We also know that the loop in line 13 is reached at most n times, therefore the commands in lines 14–16 are reached at most mn times.

Therefore, if we assume constant time for all operations (which is possible with efficient data strucures), the overall complexity of the algorithm is $\mathcal{O}(mn+n^2)$. If we further assume $m \geq n - 1$ (which is always the case if the automaton is connected, i. e. in most cases), this is equivalent to $\mathcal{O}(mn)$.

4 Testing and Evaluation

4.1 Example results

The verified ML code can now be used directly. In the context of his Isabelle automata library and the Hopcroft minimisation algorithm, Thomas Tuerk has created excellent ML testing functions for parsing, printing, and drawing automata and generating random NFAs. We used these in order to test the exported ML code on a simple example.

For instance, consider the following automaton:

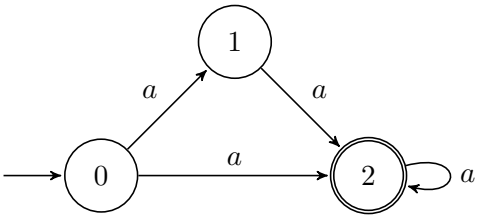


Figure 3: A simple example NFA that is used to test the algorithm.

The following Standard ML code creates a representation of this automaton and applies the simulation preorder algorithm we have exported to it:[16]

```
use "Automata_RBT.sml";
open NFA_string;
val automaton1 = NFA_string.nfa_construct ([], [], [(0, ["a"
   ], 1), (0, ["a"], 2), (1, ["a"], 2), (2, ["a"], 2)],
   [0], [2]);
val sim = nfa_simulation_preorder automaton1;
ArrayHashSet.ahs_to_list hashable_natprod sim;
```

The result is:

[16] A few lines of ML code were written as a wrapper around the *rs_nfa_simulation_preorder* function, since it expects not only an automaton, but also, among other things, a linear ordering on the label type. *nfa_simulation_preorder* merely calls *rs_nfa_simulation_preorder* with appropriate orderings/hash functions.

```
val it = [(2, 2), (1, 2), (1, 1), (0, 2), (1, 0), (0, 1),
   (0, 0)]: (int * int) list
```

This means that $0 \leq_{\mathcal{A}} 1 \leq_{\mathcal{A}} 2$ and $1 \leq_{\mathcal{A}} 0$, which can easily be checked by looking at Figure 3. Every path beginning at 0 can be matched with a path beginning at 1 and so on. Moreover, since $0 \leq_{\mathcal{A}} 1$ and $1 \leq_{\mathcal{A}} 0$, we can see directly that 0 and 1 are equivalent, i. e. they have the same right language and could therefore be merged.

4.2 Performance benchmark

Extensive evaluation has shown that the best choice of data structures for the algorithm in its current form is using array hash sets for ω, a distinct list-based set implementation for $\mathcal{C}$ and array hash maps for N, d, and δ^r. These data structures provide (almost) constant-time implementations of all the operations used in the algorithm, the exception being the addition used in calculating hash codes, which is of logarithmic complexity. The sets of the automaton itself, Q, Σ, Δ, and F are also of type *RBTSet*, but since the algorithm does not perform any operations on them save iteration, their internal representation is virtually irrelevant for its performance.

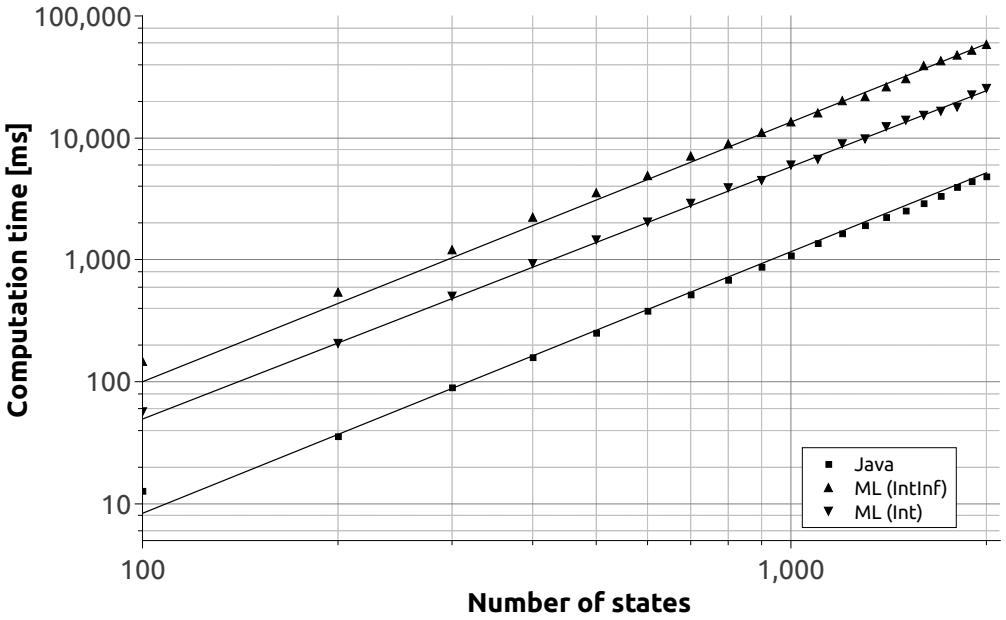
In order to see how the performance of the generated ML code compares to unverified, imperative code, we implemented the same algorithm and the same random NFA generation functions in Java and compared the average computation time of the algorithm for different parameters. Both in ML and in Java, an appropriate number of random NFAs with n states (i. e. $|Q| = n$), 4 labels (i. e. $|\Sigma| = 4$), a transition density $\frac{|\Delta|}{|Q|\cdot|\Sigma|}$ of 0.8 and a final state density $\frac{|F|}{|Q|}$ of 0.2 were created. The algorithm was then run on each of the generated NFAs. n was varied in order to observe its effect on average computation time per automaton, and the number of NFAs generated was adjusted so that it was large enough for the results to be reproducible, but also small enough to still finish within a reasonable amount of time.

The evaluation was done on a 3.50 GHz Intel Core i7-2700K with 32 GiB of memory. The ML code was compiled with MLton and the Java code was executed with the Oracle Java SE Runtime Environment 1.7.0_06-b24. However, the Java code uses the native data type *int*, which is a signed 32 bit integer, whereas the ML code uses *IntInf*, an arbitrary-precision integer data type. If one requires mathematically

correct code, arbitrary-precision integers are unavoidable, since fixed-size data types will be unable to handle very large automata. An automaton with 3 billion states would – theoretically – lead to incorrect behaviour in the Java code. In order to see the effect of arbitrary-precision integers versus native, fixed-size integers, we created a second version of the ML code, manually replacing *IntInf* in the exported code with *Int*, which is also a 32 bit signed integer, comparable to *int* in Java.

The following figures show average computation time for n ranging from 100 to 2000 in steps of 100.

n	$t_{\text{Java}}[\text{s}]$	$t_{\text{ML (IntInf)}}[\text{s}]$	$t_{\text{ML (Int)}}[\text{s}]$	$t_{\text{ML (IntInf)}}/t_{\text{Java}}$	$t_{\text{ML (Int)}}/t_{\text{Java}}$
100	0.01	0.15	0.06	11.46	4.44
200	0.04	0.55	0.21	15.24	5.74
300	0.09	1.21	0.50	13.31	5.53
400	0.16	2.24	0.94	14.14	5.91
500	0.25	3.58	1.47	14.26	5.85
600	0.38	4.92	2.04	12.90	5.36
700	0.52	7.06	2.93	13.51	5.60
800	0.69	8.97	3.91	13.08	5.69
900	0.88	11.04	4.48	12.55	5.09
1000	1.08	13.53	5.99	12.48	5.53
1100	1.38	16.13	6.68	11.71	4.85
1200	1.64	20.13	8.93	12.25	5.43
1300	1.92	21.66	9.78	11.31	5.11
1400	2.22	26.34	12.34	11.84	5.55
1500	2.55	30.77	13.96	12.06	5.47
1600	2.93	39.39	15.32	13.44	5.23
1700	3.33	42.97	16.67	12.89	5.00
1800	3.97	47.71	17.86	12.03	4.50
1900	4.40	52.42	22.61	11.92	5.14
2000	4.86	58.74	25.27	12.08	5.20



The above figure shows the data points of the three programmes in a log/log plot. The three straight lines are functions $f_i(n)$ of the form $a_i \cdot n^{b_i}$ that have been fitted onto the data points. The exponents b_i were 2.15, 2.13 and 2.07, which is slightly above the quadratic[17] complexity that we deduced theoretically earlier. This deviation is most likely due to effects from memory allocation and fluctuations in computation time, but it should be noted that the exponents are remarkably close to the prediction of 2 in any case. In fact, if we count the steps of the algorithm, i. e. how often the innermost **for each** loop is executed, we get an increase in steps of almost exactly 4 when doubling the amount of states (e. g. factor 3.97 from 500 states to 1000 states).

The factor by which the verified ML code is slower than the Java code, shown in the two rightmost columns of the table, is – averaged over all data points – 12.7 for the ML code using *IntInf* and 5.3 for the ML code using *Int*. We believe that this is a good result considering that the Isabelle algorithm is verified and was not heavily optimised before exporting to ML. Profiling the ML code shows that $17.0\,\%$ in the *IntInf* version is spent with garbage collection, $36.9\,\%$ with arithmetic of *IntInf* values and $21.2\,\%$ with hash map operations. In the *Int* version, garbage collection makes up $27.0\,\%$, hash map operations $32.6\,\%$.

[17] We say *quadratic complexity* here since we had a complexity of $O(mn)$ and here we have $m \propto n \cdot |\Sigma|$ with the transition density as the constant of proportionality. Since $|\Sigma|$ is constant, we therefore have $O(mn) = O(n^2)$ in this particular case.

5 Conclusion

We proved the correctness of an algorithm for computation of simulation preorders on NFAs and, in the process, uncovered a minor mistake in the original version from [INY04] that affects non-total automata; perhaps the authors implicitly assumed a total automaton without stating that this was the case. Formal verification is a powerful tool to uncover such implicit assumptions that might otherwise be missed by others who implement and use the algorithm.

The verified Isabelle algorithm is only about one order of magnitude slower than a comparable, unverified Java implementation; further optimisation may make the difference even smaller. This shows the feasability of proving reasonably complex algorithms correct with the Isabelle Refinement Framework and deriving executable code with good performance from it, even compared to unverified imperative code.

5.1 Working with the Refinement Framework

Peter Lammich's Refinement Framework with its step-by-step approach to verification made the initial proofs significantly more manageable than a single correctness proof of the entire algorithm. It allows the user to reason on an abstract level, without optimisations such as, in this case, the counter that replaces the lengthy condition in the **if** statement, and with abstract mathematical objects such as sets and functions instead of unnecessarily concrete data structures. Proofs on such an abstract level are easier to write, easier to understand and easier to maintain, since changing the data structures or other implementation details does not require changing the important part, i. e. the abstract correctness proof, at all.

Optimisations can be introduced gradually and separately from the algorithm itself, i. e. *local* changes in the algorithm only require *local* correctness proofs. For instance, when we replaced the condition in the **if** statement with a zero check on the counter, we did not have to prove the correctness of the algorithm again, we only had to prove that the counter always contains the correct information and that this information is equivalent to the original **if** statement. If one were to remove this optimisation or replace it with something else, one would only have to prove the correctness of these minor changes and the steps afterwards, but nothing about the algorithm itself, since its correctness has been proven already. It is obvious that without this framework, the verification of an algorithm like this would probably not have been possible in the scope of this thesis.

However, the framework is still in a very early stage of development and it is still very difficult to use for people who are not deeply involved in its development. Also, throughout the course of writing this thesis, a small number of bugs[18] have been discovered by us. The major problems with the Refinement Framework at this stage are:

Lack of automation, especially in data refinement

Parts of data refinement, such as replacing sets and maps with *StdSet* and *StdMap* are a mere technicality, but the user still has to do much of it by hand in many cases since the automation fails without a careful and subtle setup. In fact, for all but the most experienced users, setting up the automation for data refinement currently takes longer than doing the process by hand. Better automation would contribute significantly to the usability of the framework.

Considerable amount of boilerplate code

For every type of set and map occurring in the algorithm, a corresponding *StdSet* / *StdMap* type has to be fixed in the locale and its correctness assumed. Similarly, for every kind of **for each** loop[19], an iterator function has to be fixed and assumed correct. In algorithms with many different data structures and **for each** loops, this leads to a huge number of fixed variables and assumptions in the locale, all of which can be considered boilerplate code. Automation could also help in this case.

A possible solution for the iterator problem is already on the way; the introduction of polymorphic iterator functions will enable moving them to the other set operations in *StdSet*. This way, the user only has to fix the set operation record and assume its correctness, but no iterator functions anymore. However, the boilerplate from the *StdSet* and *StdMap* operations remains. Automation of this is difficult, since the user may want to implement the same abstract type with different concrete ones in different parts of their code, which the system cannot know, of course.

[18] Of course, none of these bugs compromised soundness, which is ensured by Isabelle; everything that could be proven correct with the Refinement Framework *was* correct. The bugs were mainly related to missing rules, which made it impossible to continue refinement of a **for each** loop in some complicated cases, and auxiliary commands not doing what they are supposed to do in some edge cases.

[19] two **for each** loops are of a different kind if they iterate over a different type of set or have a different kind of state (the value they return)

Locales

At the moment, several locales have to be defined for each algorithm, and considerable effort is required to then extract the executable algorithm and its correctness lemma from these locales. Again, automation would be useful for this process; we suggest some command that generates these locales internally and hides them from the user. However, it may be difficult to implement such a mechanism, as a large number of proof obligations arise in this process – trivial ones, but not all of them can be proven automatically. It is difficult to say if automation of this is possible.

5.2 Future work

The algorithm verified in this thesis can be integrated with the CAVA project. As described in chapter 2, simulation preorders can be used to decrease the size of NFAs by merging states, and in contrast to NFA minimisation, which is PSPACE-complete and for which there is therefore no known efficient algorithm, NFA reduction using simulation preorders can be done in $O(mn)$, i. e. polynomial time. Functions could therefore be created that perform reduction of NFAs, encoded with the NFA data structures from the CAVA project, in polynomial time.

It should be noted that in this thesis, there was no particular focus on performance. It is difficult to match the performance of unverified, hand-optimised imperative code with verified algorithms using only verified data structures, especially when the code is exported to another language, as it was in this case. Heavy performance optimisation would exceed the scope of this thesis, but minor modifications that could be made to the algorithm in order to increase performance are:

- Instead of ω and $\mathcal{C}$, use variables that contain $\omega \setminus \mathcal{C}$ and $\mathcal{C}$. This would reduce memory requirements by eliminating redundant data, but would also necessitate the use of a more complex data structure for $\mathcal{C}$ (as opposed to the currently used distinct list), as member checks on $\mathcal{C}$ would have to be performed often.
- Further improve the Isabelle Collection Framework, especially the hashing-based data structures, as their performance is exceedingly bad in some cases, probably due to poor behaviour of the utilised hash functions for product types.

- The hash maps could be nested instead of using tuples as keys, which would eliminate the difficulty of finding a good hash function for product types. The Java implementation already does this very successfully and we expect this improvement to lead to a significant increase in performance in the Isabelle version as well.

As an interesting side note, Büchi automata, which operate on infinite words, also use designated final states – like NFAs; their acceptance condition is that the automaton reaches a final state infinitely often as it processes the word. This is very similar to the acceptance condition on NFAs and it should therefore be possible to use the same algorithm on Büchi automata without any modifications. Since Büchi automata are widely used in model checking, an efficient algorithm to compute simulation preorders on them could be useful to deduce certain properties about systems modelled by Büchi automata and to reduce the size of these automata. With some modifications to the initialisation, the algorithm could perhaps be modified to work for other ω automata as well.

The importance of verified algorithms, despite the tremendous effort[20] required by the verification process, can be illustrated by a bug in an ML automata library [Lei00] that remained undiscovered for ten years, and by the issue with non-total automata in the algorithm from [INY04], which exists despite an informal proof of correctness of the algorithm given in the paper. Also, [CC03] and [INY04] forgot the property $\mathcal{L}(u, u) = \{\varepsilon\}$ in the third condition for merging states (see p. 7), which, of course, results in incorrect reductions in some cases. Ultimately, only a formal, computer-checked proof can provide (virtually) absolute certainty that an algorithm is free of mistakes like these.

[20] It is sometimes said that *rigour* comes from *rigor mortis*. After 2859 lines of proof code for what is essentially a simple algorithm, we are tempted to agree.

References

[CC03] Jean-Marc Champarnaud and Fabien Coulon, *NFA Reduction Algorithms by Means of Regular Inequalities*, Developments in Language Theory (Szeged, 2003), Proceedings, Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2003.

[CC05] Jean-Marc Champarnaud and Fabien Coulon, *NFA Reduction Algorithms by Means of Regular Inequalities – correction*, `http://jmc.feydakins.org/pdf/R19tcs04corr.pdf`, 2005.

[HHK95] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke, *Computing Simulations on Finite and Infinite Graphs*, Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, 1995, pp. 453–462.

[INY04] Lucian Ilie, Gonzalo Navarro, and Sheng Yu, *On NFA reductions*, Theory is Forever, Springer, 2004, pp. 112–124.

[ISoY05] Lucian Ilie, Roberto Solis-oba, and Sheng Yu, *Reducing the size of NFAs by using equivalences and preorders*, Combinatorial Pattern Matching, Jeju Island, South Korea, 2005.

[Lei00] Hans Leiß, *A Finite Automata Library for Standard ML*, `http://www.cis.uni-muenchen.de/~leiss/sml-automata.html`, 2000, See changelog of `fm.sml`.

[LL10] Peter Lammich and Andreas Lochbihler, *The Isabelle Collections Framework*, Interactive Theorem Proving, 2010, `http://www4.in.tum.de/~lammich/pub/itp10.pdf`.

[LT12] Peter Lammich and Thomas Tuerk, *Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm*, Interactive Theorem Proving, Lecture Notes in Computer Science, vol. 7406, Springer Berlin / Heidelberg, 2012, pp. 166–182, `http://cava.in.tum.de/templates/publications/LamTue12.pdf`.

[Tue12] Thomas Tuerk, *Finite Automata Library* (*proof outline*), `http://cava.in.tum.de/templates/downloads/automata-outline.pdf`, 2012, CAVA.