Formal Verification of Executable Complementation and Equivalence Checking for Büchi Automata

Julian Brunner(✉)

Technische Universität München, Munich, Germany
brunnerj@in.tum.de

Abstract. We develop a complementation procedure and an equivalence checker for nondeterministic Büchi automata. Both are formally verified using the proof assistant Isabelle/Hol. The verification covers everything from the abstract correctness proof down to the generated Sml code.

The complementation follows the rank-based approach. We formalize the abstract algorithm and use refinement to derive an executable implementation. In conjunction with a product operation and an emptiness check, this enables deciding language-wise equivalence between nondeterministic Büchi automata. We also improve and extend our library for transition systems and automata presented in previous research.

Finally, we develop a command-line executable providing complementation and equivalence checking as a verified reference tool. It can be used to test the output of other, unverified tools. We also include some tests that demonstrate that its performance is sufficient to do this in practice.

Keywords: Formal verification · Omega automata · Complementation

1 Introduction

Büchi complementation is the process of taking a Büchi automaton, and constructing another Büchi automaton which accepts the complementary language. It is a much-researched topic [10,15,20,37,38]. In fact, it has been so popular that there are now several meta-papers [41,43] chronologuing the research itself. Much of this research has focused on the state complexity of the resulting automata (see Sect. 2.3). However, Büchi complementation also has compelling applications. Model checking usually requires having the property to be checked against as either a formula or a deterministic Büchi automaton, as those are easily negated and complemented, respectively [15]. However, having access to a general complementation procedure, it becomes possible to decide

Research supported by Dfg grant Cava (Computer Aided Verification of Automata, ES 139/5-1, NI 491/12-1, SM 73/2-1) and Cava2 (Verified Model Checkers, KR 4890/1-1, LA 3292/1-1).

B. Dongol and E. Troubitsyna (Eds.): IFM 2020, LNCS 12546, pp. 239–256, 2020.
https://doi.org/10.1007/978-3-030-63461-2_13

language containment between arbitrary nondeterministic Büchi automata. This not only allows for more general model checking, but also enables checking if two automata are equivalent in terms of their language.

Unfortunately, complementation algorithms are complicated and their correctness proofs are involved. This is common in the model checking setting and there are examples of algorithms widely believed to be correct turning out not to be [8,18,40]. The situation is especially troubling as these tools act as trust multipliers. That is, the trust in the correctness of one tool is used to justify confidence in the correctness of the many entities that it checks. Motivated by this situation, our goal is to formally verify one such complementation algorithm.

We use the proof assistant Isabelle/HOL [34] for this. Thanks to its LCF-like architecture, Isabelle/HOL and the formalizations it facilitates grant very strong correctness guarantees. Our contributions are as follows.

1. Formalization of rank-based complementation [20] theory
2. Formally verified complementation implementation
3. Formally verified equivalence checker
4. Extension and continued development of automata library [7,9]

In previous work, Stephan Merz formalized complementation of weak alternating automata [33]. He also started working on a formalization of Büchi complementation. However, this only covers the first part of the complementation procedure and was never finished or published. Thus, our work constitutes what we believe to be the first formally verified implementation of Büchi complementation. The verification gaplessly covers everything from the abstract correctness proof to the executable SML code.

The equivalence checker can be used as a command-line tool to check automata in the Hanoi Omega-Automata format [1]. It takes the role of a trusted reference implementation that can be used to test the correctness of other tools, like those translating from LTL formulae to automata. With an equivalence checker, it is possible to test whether several algorithms produce automata with identical languages, given the same formula. It is also possible to test if an algorithm that simplifies automata preserves their language.

2 Theory

We follow the rank-based complementation construction described in [20]. The central concept here is the *odd ranking*, a function f that certifies the rejection of a word w by the automaton A. The complement automaton $\overline{A}$ is then designed to nondeterministically search for such an odd ranking, accepting if and only if one exists. Thus, the complement automaton accepts exactly those words that the original automaton rejects.

$$w \notin \mathcal{L}\ A \iff \exists f.\ \text{odd\_ranking}\ A\ w\ f \iff w \in \mathcal{L}\ \overline{A} \tag{1}$$

Having access to complement and product operations as well as an emptiness check, we can then decide language containment.

$$\mathcal{L}\ A \subseteq \mathcal{L}\ B \iff \mathcal{L}\ A \cap \overline{\mathcal{L}\ B} = \emptyset \iff \mathcal{L}\ \left(A \times \overline{B}\right) = \emptyset \tag{2}$$

Checking for containment in both directions then leads to a decision procedure for language-wise equivalence of Büchi automata.

2.1 Notation

We introduce some basic notation. Let $w \in \Sigma^\omega$ be an infinite sequence and $w_k \in \Sigma$ be the symbol at index k in w. Let $A = (\Sigma, Q, I, \delta, F)$ be a nondeterministic Büchi automaton with alphabet Σ, states Q, initial states $I :: Q$ set, successor function $\delta :: \Sigma \rightarrow Q \rightarrow Q$ set, and acceptance condition $F :: Q \rightarrow$ bool. Let $\mathcal{L}\ A \subseteq \Sigma^\omega$ denote the language of automaton A.

2.2 Complementation

We want to realize complementation according to Eq. 1. For this, we need to define odd rankings and the complement automaton. We also need to define run Dags as a prerequisite for odd rankings.

A run Dag is a graph whose nodes are pairs of states and natural numbers. Given an automaton A and a word w, we define it inductively as follows.

Definition 1 (Run Dag). $G = (V, E)$ *with* $V \subseteq Q \times \mathbb{N}$ *and* $E \subseteq V \times V$

$$\begin{array}{rcccr} & & p \in I & \Longrightarrow & (p, 0) \in V \\ (p, k) \in V & \Longrightarrow & q \in \delta\ w_k\ p & \Longrightarrow & (q, k+1) \in V \\ (p, k) \in V & \Longrightarrow & q \in \delta\ w_k\ p & \Longrightarrow & ((p, k), (q, k+1)) \in E \end{array}$$

Intuitively, each node $(p, k) \in V$ represents A being in state p after having read k characters from w. With that, the run Dag contains all possible paths that A can take while reading w.

We can now define odd rankings. An odd ranking is a function assigning a rank to each node in the run Dag. Given an automaton A and a word w, we require the following properties to hold for odd rankings.

Definition 2 (Odd Ranking). odd_ranking $A\ w\ f$ *with* $f :: V \rightarrow \mathbb{N}$

$$\begin{array}{ll} \forall\, v \in V. & f\ v \leq 2\,|Q| \\ \forall\, (u, v) \in E. & f\ u \geq f\ v \\ \forall\, (p, k) \in V. & F\ p \Longrightarrow \text{even}\ (f\ (p, k)) \\ \forall\, r \in \text{paths}\ G. & \textit{the path } r \textit{ eventually gets stuck in an odd rank} \end{array}$$

Intuitively, the rank of a node indicates the distance to a node from which no more accepting states are visited [15].

The final definition concerns the actual complement automaton. Given an automaton $A = (\Sigma, Q, I, \delta, F)$, we define its complement as follows.

Definition 3 (Complement Automaton). $\overline{A} = (\Sigma, Q_C, I_C, \delta_C, F_C)$

$$\begin{array}{rcl}
\delta_1 & :: & \Sigma \to (Q \rightharpoonup \mathbb{N}) \to (Q \rightharpoonup \mathbb{N}) \text{ set} \\
g \in \delta_1\ a\ f & \Longleftrightarrow & \text{dom } g = \bigcup p \in \text{dom } f.\ \delta\ a\ p \land \\
 & & \forall p \in \text{dom } f.\ \forall q \in \delta\ a\ p.\ f\ p \geq g\ q \land \\
 & & \forall q \in \text{dom } g.\ F\ q \implies \text{even } (g\ q)
\end{array}$$

$$\begin{array}{rcl}
\delta_2 & :: & \Sigma \to (Q \rightharpoonup \mathbb{N}) \to Q \text{ set} \to Q \text{ set} \\
\delta_2\ a\ g\ P & = & \begin{cases} \{q \in \text{dom } g \mid \text{even } (g\ q)\} & \textit{if } P = \{\} \\ \{q \in \bigcup p \in P.\ \delta\ a\ p \mid \text{even } (g\ q)\} & \textit{otherwise} \end{cases}
\end{array}$$

$$\begin{array}{rcl}
Q_C & :: & ((Q \rightharpoonup \mathbb{N}) \times Q \text{ set}) \text{ set} \\
Q_C & = & \delta_C^*\ \Sigma\ I_C
\end{array}$$

$$\begin{array}{rcl}
I_C & :: & Q_C \text{ set} \\
I_C & = & (\lambda p \in I.\ 2\,|Q|\,, \emptyset)
\end{array}$$

$$\begin{array}{rcl}
\delta_C & :: & \Sigma \to Q_C \to Q_C \text{ set} \\
\delta_C\ a\ (f, P) & = & \{(g, \delta_2\ a\ g\ P) \mid g \in \delta_1\ a\ f\}
\end{array}$$

$$\begin{array}{rcl}
F_C & :: & Q_C \to \text{bool} \\
F_C\ (f, P) & = & (P = \emptyset)
\end{array}$$

Since the complement automaton is designed to nondeterministically search for an odd ranking, many of the properties from Definition 2 reappear here. Instead of a ranking on the whole run Dag ($V \to \mathbb{N}$), the complement automaton deals with *level rankings*. These assign ranks to only the reachable nodes in the current level ($Q \rightharpoonup \mathbb{N}$). Furthermore, each state keeps track of which paths have yet to visit an odd rank (Q set). This encodes the property of every path getting stuck in an odd rank, with the acceptance condition requiring this set to become empty infinitely often. Together, these lead to the state type ($Q \rightharpoonup \mathbb{N}) \times Q$ set.

2.3 Complexity and Optimizations

Much of the interest in Büchi complementation focuses on its state complexity [15,38,43]. That is, one considers the number of states in the complement automaton as a function of the number of states in the original automaton. For an automaton with n states, the original construction by Büchi [10] resulted in $2^{2^{\mathcal{O}(n)}}$ states [15]. The complementation procedure derived from Safra's determinization construction [37] reduces this to $2^{\mathcal{O}(n \log n)}$ or n^{2n} states [15]. The algorithm from [20] generates a complement automaton with at most $(6n)^n$ states [15]. In the quest for closing the gap between the known lower and upper bounds, various optimizations to this algorithm have been proposed. The optimization in [15] lowers the bound to $\mathcal{O}((0.96n)^n)$ states. The algorithm is then adjusted further in [38] to lower the bound to $\mathcal{O}((0.76n)^n)$ states.

This being the first attempt at formalizing Büchi complementation, we chose not to implement these more involved optimizations. Instead, we favor the original version of the algorithm as presented in [20]. We do however implement one optimization mentioned in [20]. In Definition 3, for each successor q of p, the function δ_1 considers for q all ranks lower than or equal to the rank of p. We restrict δ_1 so that it only considers for q a rank that is equal to or one less than the rank of p. This does not change the language of the complement automaton and significantly restricts the number of successors generated for each state.

It is worth noting that in practice, factors other than asymptotical state complexity can also play a role. For instance, it turns out that determinization-based complementation often generates fewer states than rank-based complementation [41]. This is despite the fact that rank-based complementation is optimal in terms of asymptotical state complexity.

2.4 Equivalence

We want to realize equivalence according to Eq. 2. For this, we need to define a product operation and an emptiness check on Büchi automata.

The product construction follows the textbook approach, where the product of two nondeterministic Büchi automata results in one nondeterministic generalized Büchi automaton. For the emptiness check, we use Gabow's algorithm for strongly-connected components [16]. This enables checking emptiness of generalized Büchi automata directly, skipping the degeneralization to regular Büchi automata that is usually necessary for nested-Dfs-based algorithms.

3 Formalization

With the theoretical background established, we now describe the various aspects of our formalization. This section will mostly give a high-level overview, highlighting challenges and points of interest while avoiding technical details. However, specific parts of the formalization will be presented in greater detail.

3.1 Isabelle/Hol

Isabelle/Hol [34] is a proof assistant based on Higher-Order Logic (Hol), which can be thought of as a combination of functional programming and logic. Formalizations done in Isabelle are trustworthy due to its Lcf architecture. It guarantees that all proofs are checked using a small logical core which is rarely modified but tested extensively over time, reducing the trusted code base to a minimum.

Code generation in Isabelle/Hol is based on a shallow embedding of Hol constants in the target language. Equational theorems marked as code equations are translated into rewrite rules in the target language [17]. This correspondence embodies the specification of the target language semantics. As this process does not involve the Lcf kernel, the code generator is part of the trusted code base.

3.2 Basics

The most basic concept needed for our formalization is that of sequences. The Hol standard library already includes extensive support for both finite and infinite sequences. They take the form of the types list and stream.

Definition 4 (Sequences)

$$\textbf{datatype}\ \alpha\ \text{list} = [\,]\ |\ \alpha \mathbin{\#} \alpha\ \text{list}$$
$$\textbf{codatatype}\ \alpha\ \text{stream} = \alpha \mathbin{\#\#} \alpha\ \text{stream}$$

The new datatype package [3,4] allows for codatatypes like stream. The libraries of both list and stream include many common operations and their properties.

We also make use of a shallow embedding of linear temporal logic (Ltl) on streams that is defined using inductive and coinductive predicates. This is used to define a predicate holding infinitely often in an infinite sequence.

Definition 5 (Infinite Occurrence). $\text{infs}\ P\ w \iff \text{alw}\ (\text{ev}\ (\text{holds}\ P))\ w$

3.3 Transition Systems and Automata

In our formalization, we both use and extend the *Transition Systems and Automata* library [7,9]. The development of this library was in fact motivated by the idea of formalizing Büchi complementation and determinization. Since then, it has been used in several other formalizations [6,8,9,35,36,39].

The goal of this library is to support many different types of automata while avoiding both duplication and compromising usability. This is achieved via several layers of abstraction as well as the use of Isabelle's locale mechanism. For an in-depth description, see [9]. Since then, an additional abstraction layer has been introduced to consolidate various operations on automata like intersection, union, and degeneralization. However, describing this in detail is outside the scope of this paper. Thus, we will only introduce the concepts and constants that are used in later sections. We start with the definition of a transition system.

Definition 6 (Transition System)

$$\begin{aligned}&\textbf{locale}\ \text{transition\_system} =\\ &\quad \textbf{fixes}\ \text{execute} :: \mathit{transition} \Rightarrow \mathit{state} \Rightarrow \mathit{state}\\ &\quad \textbf{fixes}\ \text{enabled} :: \mathit{transition} \Rightarrow \mathit{state} \Rightarrow \text{bool}\end{aligned}$$

It fixes type variables for transitions and states as well as constants to determine which transitions are enabled in each state and which target states they lead to. This locale forms the backbone of the library. Note that it may look like it can only be used to model (sub-)deterministic transition systems. However, by instantiating the type variable *transition*, we can actually model many different types of transition systems, including nondeterministic ones [9].

We can then define concepts concerning sequences of transitions.

Definition 7 (Targets and Traces)

$$\begin{aligned}
&\text{target} = \text{fold execute} :: \textit{transition}\ \text{list} \Rightarrow \textit{state} \Rightarrow \textit{state}\\
&\text{trace} = \text{scan execute} :: \textit{transition}\ \text{list} \Rightarrow \textit{state} \Rightarrow \textit{state}\ \text{list}\\
&\text{strace} = \text{sscan execute} :: \textit{transition}\ \text{stream} \Rightarrow \textit{state} \Rightarrow \textit{state}\ \text{stream}
\end{aligned}$$

Given a sequence of transitions and a source state, these functions give the target state and the finite and infinite sequence of traversed states, respectively. Note how each of these is simply a lifted version of execute.

We can also define constants for finite and infinite paths, respectively.

Definition 8 (Paths)

$$\begin{aligned}
&\textbf{inductive}\ \text{path} :: \textit{transition}\ \text{list} \Rightarrow \textit{state} \Rightarrow \text{bool}\ \textbf{where}\\
&\quad \text{path}\ []\ p\\
&\quad \text{enabled}\ a\ p \implies \text{path}\ r\ (\text{execute}\ a\ p) \implies \text{path}\ (a \mathbin{\#} r)\ p\\
&\textbf{coinductive}\ \text{spath} :: \textit{transition}\ \text{stream} \Rightarrow \textit{state} \Rightarrow \text{bool}\ \textbf{where}\\
&\quad \text{enabled}\ a\ p \implies \text{spath}\ r\ (\text{execute}\ a\ p) \implies \text{spath}\ (a \mathbin{\#\#} r)\ p
\end{aligned}$$

These constants are (co)inductively defined predicates that capture the notion of all the transitions in a sequence being enabled at their respective states. Like before, these are lifted versions of enabled, which is also reflected in their types.

3.4 Run Dags

Having established all the basics and foundations, we can now turn to the actual formalization of Büchi complementation. We start with formalizing Definition 1 concerning run Dags. We do this by instantiating the transition system locale from Definition 6. This yields definitions for all the required graph-related concepts, like finite and infinite paths as well as reachability.

We then establish a tight correspondence between these definitions and the ones concerning automata. This requires mostly elemental induction and coinduction proofs. Only minor technical work was required to translate between paths in the automaton being labeled and paths in its run Dag being indexed.

3.5 Odd Rankings

Having formalized run Dags, we can now formalize Definition 2 concerning odd rankings. The resulting formal definition does not differ significantly from its informal counterpart and will thus not be repeated here.

We prove the left equivalence from Eq. 1, which states that an odd ranking f exists if and only if the automaton A rejects the word w.

$$w \notin \mathcal{L}\ A \iff \exists f.\ \text{odd\_ranking}\ A\ w\ f$$

We follow the proof given in [20].

The direction ⟸ is fairly straightforward. Given an odd ranking, we immediately have that all infinite paths in the run Dag get trapped in an odd rank. Together with the fact that odd ranks are not accepting, we obtain that all infinite paths in the automaton are not accepting. Formally proving this is mainly technical work consisting of establishing the correspondence between the run Dag and the automaton. However, there is one exception. In [20], the fact that all infinite paths get trapped in some rank is merely stated as part of the definition of rankings. While this is intuitively obvious from the fact that ranks are natural numbers and always decreasing along a path, it still requires rigorous proof in a formal setting. Thus, we need to define the notion of decreasing infinite sequences and prove this property via well-founded induction on the ranks.

The direction ⟹ is a lot more involved. It requires defining an infinite sequence of subgraphs of the run Dag in order to construct an odd ranking. Again, we follow the proof given in [20]. As before, we were able to follow the high-level ideas of this proof in the formalized version, with some parts requiring more fine-grained reasoning or additional technical work. However, we want to highlight one particular technique that is used several times in the proof and that required special attention in the formalized version. While most of our descriptions focus on high-level ideas, we also want to take this opportunity to present one part of the formalization in greater detail.

The idea in question concerns itself with the construction of infinite paths in graphs and transition systems. We already encountered this type of reasoning in [8]. Assume that there is a state with property P, and that for every state with property P we can find a path to another state with property P. Then, there exists an infinite path that contains infinitely many states with property P. Intuitively, this seems obvious, which is why in informal proofs, statements like these require no further elaboration. However, in a formal setting, this requires rigorous reasoning, resulting in the following proof.

Lemma 1 (Recurring Condition)

```
lemma recurring_condition:
  assumes "P p" "∀p. P p ⟹ ∃r. r ≠ [] ∧ path r p ∧ P (target r p)"
  obtains r where "spath r p" "infs P (p ## strace r p)"
proof -
  obtain f where "f p ≠ []" "path (f p) p" "P (target (f p) p)"
    if "P p" for p by ...
  let ?g = "λp. target (f p) p"
  let ?r = "λp. flat (smap f (siterate ?g p))"
  have "?r p = f p @- ?r (?g p)" if "P p" for p by ...
  show ?thesis
  proof
    show "spath (?r p) p" by ...
    show "infs P (p ## strace (?r p) p)" by ...
  qed
qed
```

This theorem is stated for general transition systems and corresponds closely to the informal one presented earlier. That is, we assume that P holds at some state p. We also assume that for every state p where P holds, we can find a nonempty path r leading to a state target r p where P holds again. We prove that from these assumptions, one can obtain an infinite path r such that for the states it traverses, P holds infinitely often. The proof consists of three major steps.

1. Skolemization of the assumption. We obtain a function f that for each state in which P holds, gives a nonempty path that leads to another state in which P holds. This can be done by either explicitly invoking the choice theorems derived from Hilbert's epsilon operator, or by using `metis`.
2. Definition of the state iteration function $?g$ and the infinite path $?r$. We define a function $?g$ that for each state p gives the target state of the path given by f. Iterating $?g$ yields all those states along the infinite path where P holds. We can then define $?r$, which is the infinite path obtained by concatenating all the finite paths given by f from each state in the iteration of $?g$.
3. Proving the required properties of $?r$. We now prove both that $?r$ is an infinite path and that for the states it traverses, P holds infinitely often. Both of these proofs require specific coinduction rules for the constants spath and infs. This is because the coinduction cannot consume the infinite sequence one item at a time, instead having to operate on finite nonempty prefixes. However, these coinduction rules are generally useful and once proven, can be reused and improve compositionality.

In the end, a surprising amount of work is necessary to prove a seemingly obvious statement. While it is easy to dismiss this as a shortcoming of either formal logic in general or of a particular proof assistant, we do not think that this is the case. Instead, we believe that situations like these point out areas where informal proofs rely on intuition, thereby hiding the actual complexity of the proof. Since this can lead to subtle mistakes, the ability of formal proofs to make it visible is valuable and one of the reasons for our confidence in them. It is also worth noting that these situations do not persistently hinder the construction of formal proofs. By proving the statement in its most general form, this needs to be done only once for this type of reasoning to become available everywhere.

3.6 Complement Automaton

Next, we formalize Definition 3 concerning the complement automaton. As in the previous section, the resulting formal definition differs only slightly from the informal one and will thus not be repeated here.

We prove the right equivalence from Eq. 1, which states that the complement automaton $\overline{A}$ accepts a word w if and only if an odd ranking f exists.

$$\exists f.\ \text{odd\_ranking}\ A\ w\ f \iff w \in \mathcal{L}\ \overline{A} \tag{3}$$

We follow the proof given in [20].

There are two main challenges to formalizing this proof. The first one is converting between different representations of rankings. On the side of the odd ranking, a ranking is a function assigning ranks to the nodes in the run Dag. On the side of the complement automaton, a ranking is an infinite sequence of level rankings in the states of the accepting path. While this seems simple enough conceptually, it requires attention to detail and much technical work in the formalization. The second challenge consists of proving that two ways of stating the same property are equivalent. The last condition in the definition of the odd ranking states that all paths eventually get stuck in an odd rank. On the side of the complement automaton, this property takes the form of a set that keeps track of which paths have yet to visit an odd rank. The acceptance condition of the complement automaton then requires this set to infinitely often become empty, ensuring that no path visits even ranks indefinitely. This again requires coinduction and the construction of infinite paths.

Together with the theorem from the previous section, we obtain the correctness theorem of complementation.

Theorem 1 (Complement Language)

theorem complement_language:
  assumes "finite (nodes A)"
  shows "$\mathcal{L}\ \overline{A} = \Sigma^{\omega} \setminus \mathcal{L}\ A$"

3.7 Refinement Framework

We want our complementation algorithm and equivalence checker to be executable. When developing formally verified algorithms, there is a trade-off between efficiency of the algorithm and simplicity of the proof. For complex algorithms, a direct proof of an efficient implementation tends to get unmanageable, as implementation details obfuscate the main ideas of the proof.

A standard approach to this problem is stepwise refinement [2], which modularizes the correctness proof. One starts with an abstract version of the algorithm and then refines it in correctness-preserving steps to the concrete, efficient version. A refinement step may reduce the nondeterminism of a program, replace abstract mathematical specifications by concrete algorithms, and replace abstract datatypes by their implementations. For example, selection of an arbitrary element from a set may be refined to getting the head of a list. This approach separates the correctness proof of the algorithm from the correctness proof of the implementation. The former can focus on algorithmic ideas without implementation details getting in the way. The latter consists of a series of refinement steps, each focusing on a specific implementation detail, without having to worry about overall correctness.

In Isabelle/Hol, stepwise refinement is supported by the Refinement Framework [24–26,32] and the Isabelle Collections Framework [23,29]. The former implements a refinement calculus [2] based on a nondeterminism monad [44], while the latter provides a library of verified efficient data structures. Both frameworks come with tool support to simplify their usage for algorithm development and to automate canonical tasks such as verification condition generation.

3.8 Implementation

Now that the abstract correctness of our complementation procedure is proven, we want to derive an executable algorithm from our definitions. We use the aforementioned refinement framework to refine our definitions that involve partial functions and sets to executable code working on association lists. For instance, the abstract correctness proof is most naturally stated on the complement state type $(Q \rightharpoonup \mathbb{N}) \times Q$ set. However, the isomorphic type $Q \rightharpoonup (\mathbb{N} \times \text{bool})$ is more suitable for the implementation. Thus, this and several other preliminary steps are taken to bring the definition into the correct shape. We also introduce the language-preserving optimization mentioned in Sect. 2.3 at this stage. The correctness proof of this optimization involves establishing a simulation relation between the original automaton and its optimized version.

Once these manual refinement steps are completed, we then use the automatic refinement tool [21,22]. It allows us to automatically refine an abstract definition to an executable implementation. It does this by instantiating abstract data structures like sets and partial functions with concrete ones like lists, hash sets, and association lists. Since refinement is compositional and the structure of the algorithm is not affected by these substitutions, refinement proofs only have to be done once for each concrete data structure. As many of these data structures have already been formalized in the library, very little has to be proven manually by the user. For instance, choosing to implement a set with a hash set instead of a list can be as simple as adding a type annotation. In particular, none of the refinement proofs have to be adjusted or redone.

At this stage, we have an executable definition that takes a successor function and gives the successor function of the complement automaton. However, we also want to be able to generate the complement automaton as a whole in an explicit representation. To do this, we make use of the Dfs Framework [30,31]. It comes with a sample instantiation that collects all unique nodes in a graph. We define the graph induced by a given automaton and generate an executable definition of its successor function. We can then run the previously verified Dfs algorithm on this graph to explore the complement automaton. The correctness proof of this algorithm then states that these are indeed all of the reachable states.

The complement automaton now has the state type $(Q \times (\mathbb{N} \times \text{bool}))$ list. This is the association list implementation of the type $Q \rightharpoonup \mathbb{N} \times \text{bool}$ mentioned earlier. Since this type is rather unwieldy, we use the result of the exploration phase to rename all the complement states using natural numbers. We then use the states explored by the Dfs algorithm to collect all of the transitions in the automaton. The end result is an explicit representation of the complement automaton with label type α and state type $\mathbb{N}$. We have $\overline{A} = (\Sigma, I, \delta, F)$ with $\Sigma :: \alpha$ list, $I :: \mathbb{N}$ list, $\delta :: (\mathbb{N} \times \alpha \times \mathbb{N})$ list, and $F :: \mathbb{N}$ list.

3.9 Equivalence

We now want to use our complementation algorithm to build an equivalence checker as outlined in Sect. 2.4. In order to decide language containment and thus equivalence, we still need a product operation and an emptiness check. To this end, we add more operations to the automata library [7]. We already added several operations for deterministic Büchi automata, deterministic co-Büchi automata, and deterministic Rabin automata as part of [9,39]. We now also add intersection, union, and degeneralization constructions for nondeterministic Büchi automata. Thanks to the new intermediate abstraction layer mentioned in Sect. 3.3, these operations generalize to all other nondeterministic automata in the library. The main challenge here was finding an abstraction for degeneralization that enables sharing this part of the formalization between both deterministic and nondeterministic automata. In the end, this was achieved by stating the main idea of degeneralization on streams rather than automata.

As mentioned in Sect. 2.4, we use an emptiness check based on Gabow's algorithm for strongly-connected components. For this, we reuse a formalization originally developed as part of the Cava model checker [13,14]. This formalization [27,28] includes both the abstract correctness proof of the algorithm, as well as executable code. Furthermore, it supports checking emptiness of generalized Büchi automata directly, enabling us to skip the degeneralization step that would usually be necessary after the product. This turns out to be significantly faster.

We now assemble these parts into an equivalence checker and then refine it to be executable. In contrast to complementation, this algorithm is much more compositional, simplifying both the abstract correctness proof and the refinement steps. We ran into one issue with the correctness theorem in the formalization of Gabow's algorithm [27,28] not being strong enough due to some technicalities. We would like to thank the author Peter Lammich for quickly generalizing the theorem after this issue was discovered.

3.10 Integration

With all the pieces in place, it is now time to integrate everything into a command-line tool. Having refined all of our definitions to be executable, we can already export Sml code from Isabelle. In order for these algorithms to function as part of a stand-alone tool, we need the ability to input and output automata. For this, we have decided to use the Hanoi Omega-Automata format [1], also called Hoa. It is used by other automata tools such as Spot [11], Owl [19], and Goal [42]. The handling of command-line parameters as well as Hoa parsing and printing are implemented manually in Sml. This piece of code wraps the verified algorithm in a command-line tool and is the only unverified part of the final executable.

The result is a command-line tool with two modes of operation: complementation and equivalence checking. Complementation takes an input automaton in the Hoa format and outputs the complement automaton either as a transition list or in the HOA format. Equivalence checking takes two input automata in the HOA format and outputs a truth value indicating their equivalence.

Our formalization is available as part of the Archive of Formal Proofs [5].

4 Evaluation

We evaluate the performance of both our complementation implementation and our equivalence checker. As a benchmark for raw complementation performance, we run our implementation on randomly-generated automata. The results are shown in Fig. 1.

States	**Samples**	**Completion Rate**	**Average Time**
5	393 438	100.00 %	0.006 s
10	41 496	99.98 %	0.110 s
15	15 616	98.30 %	3.112 s
20	16 950	36.58 %	22.695 s

Fig. 1. Complementation Performance. We use Spot's `randaut` tool to generate random automata with a given number of states. We then run our complementation implementation on them. The time limit was set to 60 s.

Furthermore, we compare the performance of our complementation implementation to Spot [11] and GOAL [42]. The results are shown in Fig. 2.

Tool	**Completion Rate**	**Average Time**	**States**
Spot (`--complement --ba`)	100.00 %	0.006 s	12.76
GOAL (`rank -tr`)	84.13 %	0.837 s	91.3
GOAL (`rank -rd`)	69.01 %	5.661 s	6 010
Our Tool	79.36 %	0.683 s	6 010

Fig. 2. Complementation Performance Comparison. We use Spot's `randltl` and `ltl2tgba` tools to generate automata from random LTL formulae. Automata with a state count other than 10 are discarded and the rest is complemented with various tools. Out of 5741 samples, 3962 could be complemented by all tools within the time limit of 60 s. To ensure comparability, we use the latter set of automata for the average time and complement states statistics.

Our tool implements the same algorithm as GOAL with the rank decrement option (`rank -rd`), which is also reflected in the identical number of states of the complement automata. However, our implementation has significantly shorter execution times thanks to extensive profiling efforts and use of efficient data structures from the Isabelle Collections Framework [23,29]. In fact, this effect is so large that it somewhat makes up for the worse asymptotical state complexity when compared to Goal with the tight rank option (`rank -tr`). The latter has significant startup overhead, but performs better on automata that are difficult to complement. While the performance of Spot is superior to either of the other tools, we want to emphasize that absolute competitiveness with unverified tools is not the goal of our work. As long as our tool is fast enough to process practical examples, it can serve its purpose as a verified reference implementation.

We also evaluate the performance of the equivalence checker. To do so, we generate random Ltl formulae and translate them to Büchi automata via both Spot [11] and Owl [19]. We then use our equivalence checker on these automata. The results are shown in Fig. 3.

States	**Samples**	**Completion Rate**	**Average Time**
$(0, 5]$	73 001	100.00 %	0.004 s
$(5, 10]$	16 024	98.49 %	0.632 s
$(10, 15]$	4 128	88.32 %	3.607 s
$(15, 20]$	1 347	64.88 %	5.203 s
$(20, \infty)$	1 370	39.12 %	8.543 s
total	95 870	97.88 %	0.347 s

Fig. 3. Equivalence Checker Performance. We use Spot's `randltl` tool to generate random Ltl formulae. We then use Spot's `ltl2tgba` tool as well as Owl's `ltl2dra` translation in conjunction with Spot's `autfilt` tool to obtain two translations of the same formula. Finally, we use our equivalence checker to check if both automata do indeed have the same language. The time limit was set to 60 s. The state count shown is that of the larger of the two automata.

When running the equivalence checker on automata that are not equivalent, the performance is often better. This is due to the fact that the algorithm searches for an accepting cycle in either $A \times \overline{B}$ or $\overline{A} \times B$. As soon as such a cycle is found, it can abort and return a negative answer. Since both complement and product are represented implicitly, this avoids constructing the full state space.

Finally, we use the same testing procedure on translations of the well-known "Dwyer"-patterns [12]. We were able to successfully check 52 out of the 55 formulae with the following exceptions. One formula resulted in automata of sizes 13 and 8, respectively, whose equivalence could not be verified within the time limit of 600 s. Two more formulae were successfully translated by Owl's `ltl2dra` translation procedure, but Spot's `autfilt` tool could not translate them to a nondeterministic Büchi automaton within the time limit of 600 s. Note that Spot's `autfilt` tool was also set up to simplify the resulting automata, as otherwise, they would quickly grow to be too large. Out of the 52 checked formulae, 49 could be processed in a matter of milliseconds, with two taking about a second and one taking 129 s.

From these tests we conclude that the performance of our tool is good enough to serve as a verified reference tool for examples of practical relevance. Note that tools like Spot include many more optimizations and heuristics that enable them to complement into much smaller automata as well as check the equivalence of much larger automata. However, it is not our goal to compete with Spot, but rather to provide a verified reference tool that is fast enough to be useful for testing other tools.

It turns out that we do not have to look far to find an illustration for this point. While gathering data for this section, our equivalence checker discovered a language mismatch between Spot's and Owl's translation of the same LTL formula. The developers of Owl confirmed that this was indeed a bug in the implementation of its `ltl2dra` translation procedure and promptly fixed it. Manifestation of this issue was very rare, first occurring after about 50 000 randomly-generated formulae. This demonstrates the need for verified reference implementations, as even extensively tested software can still contain undetected issues.

5 Conclusion

We developed a formally verified and executable complementation procedure and equivalence checker. The formal theory acts as a very detailed and machine-checkable description of rank-based complementation. Additionally, our formalization includes executable reference tools. These come with a strong correctness guarantee as everything from the abstract correctness down to the executable SML code is covered by the verification. This high confidence in their correctness justifies their use to test other, unverified tools.

We also contributed additional functionality as well as an improved architecture to the automata library. This emphasizes the software engineering aspect of formal theory development where theories can be reused and become more and more useful as they mature.

For future work, it would be desirable to formalize an algorithm that generates a complement automaton with fewer states. As mentioned in Sect. 2.3, this concerns both asymptotical state complexity as well as performance in practice. It would also be of interest to verify the bounds on asymptotical state complexity.

References

1. Babiak, T., et al.: The Hanoi omega-automata format. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 479–486. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_31
2. Back, R.-J., von Wright, J.: Refinement Calculus - A Systematic Introduction. Texts in Computer Science. Springer, New York (1998). https://doi.org/10.1007/978-1-4612-1674-2
3. Biendarra, J., et al.: Foundational (Co)datatypes and (Co)recursion for higher-order logic. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 3–21. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66167-4_1
4. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (Co)datatypes for isabelle/HOL. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 93–110. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_7

5. Brunner, J.: Büchi complementation. In: Archive of Formal Proofs (2017). https://www.isa-afp.org/entries/Buchi_Complementation.html
6. Brunner, J.: Partial order reduction. In: Archive of Formal Proofs (2018). https://www.isa-afp.org/entries/Partial_Order_Reduction.html
7. Brunner, J.: Transition systems and automata. In: Archive of Formal Proofs (2017). https://www.isa-afp.org/entries/Transition_Systems_and_Automata.html
8. Brunner, J., Lammich, P.: Formal verification of an executable LTL model checker with partial order reduction. J. Autom. Reason. **60**, 3–21 (2018). https://doi.org/10.1007/s10817-017-9418-4
9. Brunner, J., Seidl, B., Sickert, S.: A verified and compositional translation of LTL to deterministic rabin automata. In: ITP 2019 (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.11
10. Richard Büchi, J.: On a decision method in restricted second order arithmetic. In: Proceedings of the International Congress on Logic, Methodology, and Philosophy of Science, p. 1962, Berkeley, California, USA (1960)
11. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, É., Xu, L.: Spot 2.0 — a framework for LTL and ω-automata manipulation. In: Artho, C., Legay, A., Peled, D. (eds.) ATVA 2016. LNCS, vol. 9938, pp. 122–129. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46520-3_8
12. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property specification patterns for finite-state verification. In: Proceedings of the Second Workshop on Formal Methods in Software Practice (1998). https://doi.org/10.1145/298595.298598
13. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.-G.: A fully verified executable LTL model checker. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 463–478. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_31
14. Esparza, J., Lammich, P., Neumann, R., Nipkow, T., Schimpf, A., Smaus, J.: A fully verified executable LTL model checker. In: Archive of Formal Proofs 2014 (2014). https://www.isa-afp.org/entries/CAVA_LTL_Modelchecker.shtml
15. Friedgut, E., Kupferman, O., Vardi, M.Y.: Büchi complementation made tighter. Int. J. Found. Comput. Sci. **17**(4), 851–868 (2006). https://doi.org/10.1142/S0129054106004145
16. Gabow, H.N.: Path-based depth-first search for strong and biconnected components. Inf. Process. Lett. **74**(3–4), 107–114 (2000). https://doi.org/10.1016/S0020-0190(00)00051-X
17. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol 6009, pp. 103–117. Springer, Berlin, Heidelberg (2010).https://doi.org/10.1007/978-3-642-12251-4_9
18. Holzmann, G.J., Peled, D.A., Yannakakis, M.: On nested depth first search. In: The Spin Verification System, Proceedings of a DIMACS Workshop (1996). https://doi.org/10.1090/dimacs/032/03
19. Křetínský, J., Meggendorfer, T., Sickert, S.: Owl: a library for ω-Words, automata, and LTL. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 543–550. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_34
20. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. ACM Trans. Comput. Log. **2**(3), 408–429 (2001). https://doi.org/10.1145/377978.377993
21. Lammich, P.: Automatic data refinement. In: Archive of Formal Proofs (2013).https://www.isa-afp.org/entries/Automatic_Refinement.shtml

22. Lammich, P.: Automatic data refinement. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 84–99. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_9
23. Lammich, P.: Collections framework. In: Archive of Formal Proofs (2009). https://www.isa-afp.org/entries/Collections.shtml
24. Lammich, P.: Refinement for monadic programs. In: Archive of Formal Proofs (2012). https://www.isa-afp.org/entries/Refine_Monadic.shtml
25. Lammich, P.: Refinement to imperative/HOL. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 253–269. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_17
26. Lammich, P.: The imperative refinement framework. In: Archive of Formal Proofs (2016). https://www.isa-afp.org/entries/Refine_Imperative_HOL.shtml
27. Lammich, P.: Verified efficient implementation of Gabow's strongly connected component algorithm. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 325–340. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_21
28. Lammich, P.: Verified efficient implementation of Gabow's strongly connected components algorithm. In: Archive of Formal Proofs (2014). https://www.isa-afp.org/entries/Gabow_SCC.shtml
29. Lammich, P., Lochbihler, A.: The Isabelle collections framework. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 339–354. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14052-5_24
30. Lammich, P., Neumann, R.: A framework for verifying depth- first search algorithms. In: CPP 2015 (2015). https://doi.org/10.1145/2676724.2693165
31. Lammich, P., Neumann, R.: A framework for verifying depth- first search algorithms. In: Archive of Formal Proofs (2016). https://www.isa-afp.org/entries/DFS_Framework.shtml
32. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32347-8_12
33. Merz, S.: Weak alternating automata in Isabelle/HOL. In: Aagaard, M., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 424–441. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44659-1_26
34. Nipkow, T., Wenzel, M., Paulson, L.C. (eds.): Isabelle/HOL. A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
35. Sachtleben, R.: Formalisation of an adaptive state counting algorithm. In: Archive of Formal Proofs (2019). https://www.isaafp.org/entries/Adaptive_State_Counting.html
36. Sachtleben, R., et al.: A mechanised proof of an adaptive state counting algorithm. In: ICTSS 2019. https://doi.org/10.1007/978-3-030-31280-0_11
37. Safra, S.: On the complexity of omega-automata. In: 29th Annual Symposium on Foundations of Computer Science (1988). https://doi.org/10.1109/SFCS.1988.21948
38. Schewe, S.: Büchi complementation made tight. In: STACS 2009 (2009). https://doi.org/10.4230/LIPIcs.STACS.2009.1854
39. Seidl, B., Sickert, S.: A compositional and unified translation of LTL into !-Automata. In: Archive of Formal Proofs (2019). https://www.isa-afp.org/entries/LTL_Master_Theorem.html

40. Siegel, S.F.: What's wrong with on-the-fly partial order reduction. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11562, pp. 478–495. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25543-5_27
41. Tsai, M.-H., et al.: State of büchi complementation. Log. Method Comput. Sci. **10**4 (2014). https://doi.org/10.2168/LMCS-10(4:13)2014
42. Tsay, Y.-K., Chen, Y.-F., Tsai, M.-H., Wu, K.-N., Chan, W.-C.: GOAL: a graphical tool for manipulating Büchi automata and temporal formulae. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 466–471. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_35
43. Vardi, M.Y.: The Büchi complementation saga. In: STACS 2007 (2007). https://doi.org/10.1007/978-3-540-70918-3_2
44. Wadler, P.: Comprehending monads. Math. Struct. Comput. Sci. **4**, 461–493 (1992). https://doi.org/10.1017/S0960129500001560