A Verified and Compositional Translation of LTL to Deterministic Rabin Automata

Julian Brunner
Technische Universität München, Germany
julian.brunner@tum.de

Benedikt Seidl
Technische Universität München, Germany
benedikt.seidl@tum.de

Salomon Sickert[1]
Technische Universität München, Germany
salomon.sickert@tum.de

Abstract

We present a formalisation of the unified translation approach from linear temporal logic (LTL) to ω-automata from [19]. This approach decomposes LTL formulas into "simple" languages and allows a clear separation of concerns: first, we formalise the purely logical result yielding this decomposition; second, we develop a generic, executable, and expressive automata library providing necessary operations on automata to re-combine the "simple" languages; third, we instantiate this generic theory to obtain a construction for deterministic Rabin automata (DRA). We extract from this particular instantiation an executable tool translating LTL to DRAs. To the best of our knowledge this is the first verified translation of LTL to DRAs that is proven to be double-exponential in the worst case which asymptotically matches the known lower bound.

2012 ACM Subject Classification Theory of computation → Automata over infinite objects; Theory of computation → Modal and temporal logics; Theory of computation → Interactive proof systems

Keywords and phrases Automata Theory, Automata over Infinite Words, Deterministic Automata, Linear Temporal Logic, Model Checking, Verified Algorithms

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.11

Supplement Material The described Isabelle/HOL development is archived in the "Archive of Formal Proofs" and is split into the entries [10] and [39].

Funding This work was partially funded and supported by the German Research Foundation (DFG) project "Verified Model Checkers" (317422601).

Acknowledgements The authors want to thank Manuel Eberl, Javier Esparza, Lars Hupel, Peter Lammich, and Tobias Nipkow for their helpful comments and technical expertise.

1 Introduction

As time has shown again and again, bugs in hardware and software can have dramatic costs, ranging from monetary damages over destroyed property to life-threatening situations. In order to prevent the introduction of unwanted behaviour into software or hardware designs, an immense amount of testing and debugging is applied. However, for critical systems such methods are not enough, since they simply cannot guarantee the absence of bugs in general. Formal methods offer here a way forward by applying mathematical rigour to detect and rule out unwanted behaviour. Model checking [14] is one of the most successful techniques

[1] Corresponding author

in the area of formal methods. A key component for model checking reactive systems, i.e., non-terminating systems interacting with an open environment, against a temporal specification language, in our case linear temporal logic (LTL), is the translation to a suitable automaton model over infinite words.

Throughout the last decades, a wide variety of translation strategies to different types of ω-automata have been proposed and implemented, e.g. [23, 22, 2, 4, 43, 17]. However, as mentioned before, software development seems to be inherently error-prone, not to mention there might be mistakes in the definition of these constructions themselves. So, how can we trust these implementations to produce the correct automata for identifying bugs or proving their absence? Who watches the watchers?

Exactly that train of thought lead to the development of the Cava LTL model checker [20] which is verified in Isabelle and exported as an executable tool. The model checker includes a translation from LTL to nondeterministic Büchi automata due to [23]. However, for model checking other structures, such as probabilistic systems, other types of automata are necessary, such as limit-deterministic [44, 15] or deterministic automata [5]. Consequently, there is the need to formalise new translations from scratch which seems wasteful and cumbersome.

It would be desirable to have a separation of concerns: a theory that captures the common essence of LTL for all desired translations and that leaves a small gap to deal with the specifics of a chosen automaton model. The logical framework of [19] sketches an approach to such a modularisation: a theorem decomposing an LTL formula φ into "simple" languages, named $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$, such that:

$$\mathcal{L}(\varphi) = \bigcup_{\substack{X \subseteq \nu(\varphi) \\ Y \subseteq \mu(\varphi)}} (L^1_{\varphi,X} \cap L^2_{X,Y} \cap L^3_{X,Y})$$

where X and Y are sets of least- and greatest-fixed operators – hence the names ν and μ – that are subformulas of φ. We will later see a formal definition of these sets. This decomposition outlines a simple strategy to obtain a translation from LTL to our chosen automaton model: first, we define constructions for the "simple" languages; second, we implement two Boolean operations, namely union and intersection, in the automaton model; third, we combine the automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$ using these Boolean operations.

Contribution

We provide a formalisation of [19] in Isabelle and contribute the following components: (1) a generic and expressive automata library[2] providing the necessary Boolean operations, (2) a formalisation of the Master Theorem [19] decomposing LTL formulas, (3) a combination of these two components to obtain an executable and verified translation from LTL to deterministic Rabin automata (DRA) of asymptotic optimal size, and (4) an implementation extracted from the Isabelle theory combined with an LTL parser, a verified LTL simplifier, and a serialisation to the HOA format [3], a textual format for ω-automata. Note that the resulting implementation is just one use-case and using the same framework we can also obtain a construction for other types of ω-automata, e.g. nondeterministic Büchi automata (NBA) or deterministic generalised Rabin automata. However, this would exceed the scope and space of this paper.

[2] The scope of the library is actually wider than just the support of ω-automata: automata on finite words and abstract transition systems can also be expressed.

Isabelle/HOL [36] is a proof assistant based on Higher-Order Logic (HOL), which can be thought of as a combination of functional programming and logic. Formalisations done in Isabelle are trustworthy for two reasons: First, Isabelle's LCF architecture guarantees that all proofs are checked using a very small logical core which is rarely modified but tested extensively over time. This reduces the trusted code base to a minimum. Second, bugs in the core rarely lead to accidentally proving false propositions. Bugs that have large effects are easily caught, while the limited applicability of bugs with small effects is unlikely to coincide with a logical mistake in the large-scale structure of the proof. In order to export executable code, we use the Isabelle code generator in conjunction with the monadic refinement framework [26] and automatic refinement [27]. Finally, we use several entries from the "Archive of Formal Proofs" (AFP), a collection of formalisations for Isabelle that are maintained and continuously machine-checked.

Related Work

A substantial amount of work has already been invested into verifying translations from linear temporal logic (LTL) to nondeterministic Büchi automata (NBA[3]): We already mentioned [20] which includes a translation to NBAs following the tableau construction from [23]. Further, the translation proposed by [22], which translates LTL via very-weak alternating automata to NBAs, has been formalised by [25] in HOL4. This work also includes an executable refinement of the abstract algorithm.

Alternating automata have been previously studied in [34] with an application to the translation of LTL to alternating ω-automata. However, the translation from alternating automaton to NBAs is not included. At the other end of the spectrum the publication [17], with the formal proof development archived in [40], presents a direct, verified, and executable translation from LTL to deterministic generalised Rabin automata. However, this construction is only shown to be triple-exponential and thus one exponential larger than the known, optimal lower bound. It is also important to mention that with the help of the Isabelle formalisation errors in the original publication [18] were uncovered and removed for the journal version [17]. This highlights again how important such a rigorous development for verification tools is.

Another interesting point is that the DRA constructions we provide for the "simple" languages can be seen as a version of Brzozowski's derivatives [13] applied to LTL formulas. Derivative-based constructions seem to be more natural in the functional programming paradigm as the work on regular expression equivalence from [37] shows.

Outline

After a brief introduction of the preliminaries in Section 2 we discuss the used automata formalisation in Section 3. We then give an overview of the LTL decomposition results in Section 4 and finally derive an executable LTL to DRA translation in Section 5.

[3] In the context of this paper we do not distinguish minor variations of acceptance conditions and the term NBA includes also nondeterministic generalised Büchi automata as well as transition-based Büchi automata. Similar we use the term DRA also for deterministic generalised Rabin automata.

2 Preliminaries

Locales

Isabelle provides a mechanism for parameterized theory contexts in the form of locales [6]. In a simplified sense, this means that a named context can be defined that is both parameterized by types and terms as well as augmented with assumptions. It is then possible to add various definitions and theorems within this context. Finally, by instantiating the parameters and proving the assumptions, these definitions and theorems also become instantiated and available to the enclosing context.

ω-Words

Let Σ be a finite alphabet. An ω-word w over Σ is an infinite sequence of letters $a_0 a_1 a_2 \ldots$ with $a_i \in \Sigma$ for all $i \geq 0$ and an ω-language is a set of ω-words. We use two different representations for ω-words over a type α: as a function "$\alpha\,\mathsf{word} = \mathsf{nat} \Rightarrow \alpha$" and as a codatatype "$\alpha\,\mathsf{stream} = \alpha \mathbin{\#\#} \alpha\,\mathsf{stream}$". The reason for this division is historic and is due to the fact that the material building on the *LTL* entry [41] predates the development of the codatatype package [7]. Observe that these two types are isomorphic.

The function $\mathsf{prefix}\ i\ w$ returns the finite prefix of w of length i and the function $\mathsf{suffix}\ i\ w$ gives the infinite suffix of w starting at i. The concatenation operator $w' \frown w$ prepends the finite word w' to w.

We introduce the constants scan and sscan for lists and streams, respectively. They work like the identically named function in Haskell, in that they perform a fold with accumulation. That is, they fold over a list or stream and collect the state of the fold at each step and return this collection as a list or stream, respectively. Thus, unlike fold, it is also possible to define this function on infinite sequences.

We also introduce the constant "$\mathsf{infs} :: (\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha\ \ \mathsf{stream} \Rightarrow \mathsf{bool}$" that indicates if a predicate is fulfilled infinitely often in a stream. We will use infs to define acceptance conditions for ω-automata.

Linear Temporal Logic

We base our contribution on the *LTL* entry found in the AFP [41] and extend it where necessary. The datatype we use for LTL syntactically enforces formulas to be in negation normal form. In order to preserve the expressiveness of LTL with negation, we need to include for the $\mathbf{U}$ (Until) operator its dual $\mathbf{R}$ (Release). For the logical decomposition result it is also essential to include $\mathbf{W}$ (Weak-Until) and $\mathbf{M}$ (Strong-Release). As usual we use $\mathbf{F}\ \varphi$ (Eventually) as an abbreviation for $\mathbf{tt}\ \mathbf{U}\ \varphi$ and $\mathbf{G}\ \psi$ (Always) for $\mathbf{ff}\ \mathbf{R}\ \psi$.

▶ **Definition 1** (Linear Temporal Logic).

$$\begin{aligned}\mathbf{datatype}\ \ \alpha\,\mathsf{ltl} = {} & \mathbf{tt} \mid \mathbf{ff} \mid \alpha \mid \neg\alpha \mid (\alpha\,\mathsf{ltl}) \wedge (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl}) \vee (\alpha\,\mathsf{ltl}) \mid \mathbf{X}\ (\alpha\,\mathsf{ltl}) \\ & \mid (\alpha\,\mathsf{ltl})\ \mathbf{U}\ (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl})\ \mathbf{R}\ (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl})\ \mathbf{W}\ (\alpha\,\mathsf{ltl}) \mid (\alpha\,\mathsf{ltl})\ \mathbf{M}\ (\alpha\,\mathsf{ltl})\end{aligned}$$

The type variable α determines the type of the atomic propositions. We write $\mathsf{atoms}\ \varphi$ to refer to the set of atomic propositions in a formula φ. The function $\mathsf{sf}\ \varphi$ computes all subformulas of φ, i.e., all subtrees of its syntax tree. Additionally, we define $\mathsf{subformulas}_\mu\ \varphi$ as set of subformulas of the form $\psi\ \mathbf{U}\ \chi$ or $\psi\ \mathbf{M}\ \chi$, and $\mathsf{subformulas}_\nu\ \varphi$ as the set of subformulas of the form $\psi\ \mathbf{R}\ \chi$ or $\psi\ \mathbf{W}\ \chi$.

▶ **Definition 2** (Semantics). *The entailment relation* $\models :: \alpha\ \mathsf{set}\ \mathsf{word} \Rightarrow \alpha\ \mathsf{ltl} \Rightarrow \mathsf{bool}$ *is defined recursively as follows:*

$$
\begin{array}{lll}
w \models \mathbf{tt} & & w \models \mathbf{X}\ \varphi \quad = \mathsf{suffix}\ 1\ w \models \varphi \\
w \not\models \mathbf{ff} & & \\
w \models a \quad = a \in w\ 0 & & w \models \varphi\ \mathbf{U}\ \psi = \exists i.\ \mathsf{suffix}\ i\ w \models \psi \wedge (\forall j < i.\ \mathsf{suffix}\ j\ w \models \varphi) \\
w \models \neg a \quad = a \notin w\ 0 & & w \models \varphi\ \mathbf{R}\ \psi = \forall i.\ \mathsf{suffix}\ i\ w \models \psi \vee (\exists j < i.\ \mathsf{suffix}\ j\ w \models \varphi) \\
w \models \varphi \wedge \psi = w \models \varphi \wedge w \models \psi & & w \models \varphi\ \mathbf{W}\ \psi = \forall i.\ \mathsf{suffix}\ i\ w \models \varphi \vee (\exists j \leq i.\ \mathsf{suffix}\ j\ w \models \psi) \\
w \models \varphi \vee \psi = w \models \varphi \vee w \models \psi & & w \models \varphi\ \mathbf{M}\ \psi = \exists i.\ \mathsf{suffix}\ i\ w \models \varphi \wedge (\forall j \leq i.\ \mathsf{suffix}\ j\ w \models \psi)
\end{array}
$$

We define the set of all words over an alphabet Σ *satisfying a formula* φ*:*

$$\mathsf{language}\ \Sigma\ \varphi = \{w.\ w \models \varphi \wedge \mathsf{range}\ w \subseteq \Sigma\}\,.$$

Equivalence Relations over LTL

We define three equivalence relations over LTL formulas: The largest equivalence relation is *language equivalence*. Two formulas are *(language-)equivalent* if they are satisfied by exactly the same words.

A smaller relation is defined by *propositional equivalence*. We interpret an LTL formula φ in propositional logic by treating every subformula that is a literal (a, $\neg a$) or a modal operator ($\mathbf{X}$, $\mathbf{U}$, $\mathbf{M}$, $\mathbf{R}$, $\mathbf{W}$) as a propositional variable. If a set of these subformulas $\mathcal{I}$ is a propositional model for φ, we write $\mathcal{I} \models_p \varphi$. Two formulas are *propositionally equivalent* if are satisfied by the same propositional models.

▶ **Definition 3** (Propositional Semantics). *The propositional entailment relation* $\models_p :: \alpha\ \mathsf{ltl}\ \mathsf{set} \Rightarrow \alpha\ \mathsf{ltl} \Rightarrow \mathsf{bool}$ *is defined recursively as follows:*

$$
\begin{array}{lll}
\mathcal{I} \models_p \mathbf{tt} & & \mathcal{I} \models_p \mathbf{X}\ \varphi \quad = (\mathbf{X}\ \varphi) \in \mathcal{I} \\
\mathcal{I} \not\models_p \mathbf{ff} & & \\
\mathcal{I} \models_p a \quad = a \in \mathcal{I} & & \mathcal{I} \models_p \varphi\ \mathbf{U}\ \psi = (\varphi\ \mathbf{U}\ \psi) \in \mathcal{I} \\
\mathcal{I} \models_p \neg a \quad = (\neg a) \in \mathcal{I} & & \mathcal{I} \models_p \varphi\ \mathbf{R}\ \psi = (\varphi\ \mathbf{R}\ \psi) \in \mathcal{I} \\
\mathcal{I} \models_p \varphi \wedge \psi = \mathcal{I} \models_p \varphi \wedge \mathcal{I} \models_p \psi & & \mathcal{I} \models_p \varphi\ \mathbf{W}\ \psi = (\varphi\ \mathbf{W}\ \psi) \in \mathcal{I} \\
\mathcal{I} \models_p \varphi \vee \psi = \mathcal{I} \models_p \varphi \vee \mathcal{I} \models_p \psi & & \mathcal{I} \models_p \varphi\ \mathbf{M}\ \psi = (\varphi\ \mathbf{M}\ \psi) \in \mathcal{I}
\end{array}
$$

Finally, *constants equivalence* is the smallest of the three equivalence relations. We use the function $\mathsf{eval} :: \alpha\ \mathsf{ltl} \Rightarrow \mathsf{tvl}$ with the three-valued logic "$\mathsf{tvl} = \mathsf{Yes} \mid \mathsf{No} \mid \mathsf{Maybe}$". It returns Yes iff φ is propositionally equivalent to $\mathbf{tt}$, and No iff φ is propositionally equivalent to $\mathbf{ff}$, respectively. Otherwise, Maybe is returned. The actual Isabelle formalisation does not refer to propositional equivalence, but in order to simplify the presentation we use the presented characterisation. Two formulas φ and ψ are *constants-equivalent* iff they are (syntactically) identical or "$\mathsf{eval}\ \varphi = \mathsf{eval}\ \psi \neq \mathsf{Maybe}$".

▶ **Definition 4** (Equivalence Relations). *For* $\varphi :: \alpha\ \mathsf{ltl}$ *and* $\psi :: \alpha\ \mathsf{ltl}$*, we define:*

$$
\begin{array}{lcl}
\varphi \sim_l \psi & = & \forall w.\ w \models \varphi \longleftrightarrow w \models \psi \\
\varphi \sim_p \psi & = & \forall \mathcal{I}.\ \mathcal{I} \models_p \varphi \longleftrightarrow \mathcal{I} \models_p \psi \\
\varphi \sim_c \psi & = & (\varphi = \psi \vee (\mathsf{eval}\ \varphi = \mathsf{eval}\ \psi \wedge \mathsf{eval}\ \psi \neq \mathsf{Maybe}))
\end{array}
$$

▶ **Lemma 5** (Order of Equivalence Relations).

$$\sim_c \leq \sim_p \leq \sim_l$$

Note that this order also corresponds to the computational complexity, with $\sim_c$ being the easiest to compute and $\sim_l$ the hardest.

3 Transition Systems and Automata

Automata are a popular subject in their own right in theoretical computer science and also have many applications, like regular expression matching and model checking. As such, it suggests itself to formalise these concepts separately and generically as a library to be shared. We first establish our goals for such a library. The deceptively simple term automaton covers a diverse range of objects that need to be supported. These differ in various ways, including but not limited to: successors (deterministic, nondeterministic), labelling (state-labeled, transition-labeled), and acceptance condition (finite, Büchi, Rabin, etc.). For each automaton type, we want to formalise fundamental concepts like path, reachability, and language. We would also like to formalise constructions like Boolean operations (union, intersection, complementation), and degeneralisation of Büchi acceptance conditions. As an overall goal, we want to share as much of the formalisation as possible by keeping it abstract. This avoids duplication and often makes definitions and proofs simpler and more elegant. Finally, we want to do all of this while providing good usability and automation, especially concerning the basic concepts that constitute the foundation of the library.

With these goals in mind, we look at the formalisations that are already available in the Isabelle ecosystem. First off, there are many ad-hoc formalisations of transition systems and automata done as part of other formalisations [40, 21, 1, 31]. Furthermore, there have been a few major formalisations as part of the Cava project [21], although not all of them were preserved or published. Stephan Merz and Alexander Schimpf formalised NBAs and NGBAs in preliminary work of the Cava project [38] and then later as part of Cava itself. Peter Lammich is the author of the current Cava automata library [28], which includes state-labeled NBAs and NGBAs. These formalisations cover a very specific set of automata, making them convenient to use, but only if one happens to need exactly that type of automaton. Another unpublished formalisation by Thomas Tuerk is more generic and covers DFAs, NFAs, NBAs, and NGBAs. It achieves this genericity by modelling some of these automata as special cases of others, which allows for sharing of definitions and proofs. For instance, a deterministic transition system would naturally be modelled using the type "$\alpha \Rightarrow \rho \Rightarrow \rho$". Alternatively, it can also be treated as a special case of a nondeterministic transition system with the type "$(\rho \times \alpha \times \rho)\,\mathsf{set}$". However, this causes several issues. Firstly, since the type is too weak, a uniqueness predicate on the term level is needed to only allow those transition relations that act like functions. These predicates then have to be carried around in all proofs explicitly, rather than being encoded in the type. Secondly, due to the type being a poor fit, we can no longer do things like folding over the successor function. Lastly, the user is restricted to a single representation, rather than, for instance, being able to choose between explicit ("$(\rho \times \alpha \times \rho)\,\mathsf{set}$") and implicit ("$\alpha \Rightarrow \rho \Rightarrow \rho\,\mathsf{set}$") representations.

We use these experiences to design a new architecture in order to achieve the goals we set earlier. Since our primary goal is sharing via abstraction, this is what will mainly motivate our decisions. There are two observations to be made. Firstly, acceptance conditions are far too diverse and specific to be treated abstractly. Thus, our abstract representation will cover transition systems instead of automata, with acceptance conditions being added on a more concrete level at a later stage. This idea is not new and was in fact used in most of the earlier formalisations as well. Secondly, as mentioned in the previous paragraph, specialisation as a mechanism of abstraction has various issues. Instead, we choose to use the mechanism of instantiation via locales (Section 2), the advantages of which will become apparent in the following sections. Thus, the library formalises *abstract transition systems* (Section 3.1), which are then instantiated and used as building blocks for *concrete automata* (Section 3.2).

We try to formalise as much as possible in the context of abstract transition systems, since this both often leads to elegance and conciseness and is shared between all concrete automata. Thanks to this, adding a new automaton requires only a minimal amount of setup, allowing users to use the library in conjunction with their own custom automata representations. That being said, the set of automata supplied with the library is also growing and becoming more useful, making this less and less necessary. In the end, we supply both a collection of useful automata as well as the tools to easily add custom ones as needed.

3.1 Abstract Transition Systems

Having decided on our architecture, the central decision lies in the specification of the locale for transition systems. We focus on the defining property of a transition system: its ability to use transitions to move from state to state. This leads us directly to the specification in terms of its types (*transition* and *state*) and its terms (execute and enabled).

▶ **Definition 6.**

locale transition-system =
  fixes execute :: *transition* ⇒ *state* ⇒ *state*
  fixes enabled :: *transition* ⇒ *state* ⇒ bool

Given a transition and a source state, the function execute specifies the target state for that transition. Analogously, the function enabled determines whether the given transition is enabled at the given source state. Together, these functions capture the essence of a transition system in terms of its ability to transition between states. Given the types, it may seem appealing to combine both constants into a single one with result type "*state* option". This sounds great in theory, but unfortunately, is very inconvenient to work with in practice. It mixes the issue of finding the target of a transition with that of whether that transition was valid in the first place. Keeping these two things separate makes definitions simpler and allows for better automation in proofs.

Having defined the transition-system locale, we now develop some abstract theory within this context. So far, we can only execute single transitions, so we look at finite and infinite sequences of transitions. We introduce the following constants based on the execute function.

▶ **Definition 7.**

target = fold execute :: *transition* list ⇒ *state* ⇒ *state*
trace = scan execute :: *transition* list ⇒ *state* ⇒ *state* list
strace = sscan execute :: *transition* stream ⇒ *state* ⇒ *state* stream

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 both the simplicity and elegance of these definitions and how each of them is simply a lifted version of execute.

We can do something similar for the enabled function.

▶ **Definition 8.**

inductive path :: *transition* list ⇒ *state* ⇒ bool **where**
  path [] *p*
  enabled *a p* ⟹ path *r* (execute *a p*) ⟹ path (*a* # *r*) *p*
coinductive spath :: *transition* stream ⇒ *state* ⇒ bool **where**
  enabled *a p* ⟹ spath *r* (execute *a p*) ⟹ spath (*a* ## *r*) *p*

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 in the previous paragraph, these are basically lifted versions of enabled, which is also reflected in their types.

Together, these form the very foundation of the library, since almost every other concept is in some way related to sequences of transitions. The nice thing about these definitions is that they lend themselves very well to automation. In the case of definitions lifted from execute, we can define simplification rules. In the case of definitions lifted from enabled, we can define safe introduction and elimination rules. This works for both the constructors of sequences (# and ##), as well as the operators for concatenation (@, @−). Convenience and automation regarding the basic concepts was a major shortcoming of earlier libraries.

Next, we define the constant reachable for the set of reachable states from a source state. Like path, this is an inductively defined predicate. Alternatively, we could have defined reachable in terms of target and path. Instead, it is defined directly based on execute and enabled and the connection to target and path is shown as a lemma.

There are some interesting things we can formalise even on this very abstract level. We present one such example in the construction of infinite paths.

▶ **Lemma 9** (Recurring Condition).

$$\begin{aligned}
&\textbf{fixes } P :: \mathit{state} \Rightarrow \mathsf{bool} \textbf{ and } p :: \mathit{state} \\
&\textbf{assumes } P\ p \textbf{ and } \bigwedge p.\ P\ p \implies \exists r.\ r \neq [] \wedge \mathsf{path}\ r\ p \wedge P\ (\mathsf{target}\ r\ p) \\
&\textbf{obtains } r :: \mathit{transition}\,\mathsf{stream} \\
&\textbf{where } \mathsf{spath}\ r\ p \textbf{ and } \mathsf{infs}\ P\ (p\ \#\#\ \mathsf{strace}\ r\ p)
\end{aligned}$$

Here, the premises only guarantee the repeated existence of a finite extension to an existing finite path, which we want to use to construct an infinite path. Proving a statement like this is cumbersome, as it requires skolemisation of the premise, construction of a stream via iteration combinators and finally proving the properties via coinduction. By providing generic rules like these, all this complexity is hidden and users can restrict themselves to easy-to-work-with constants like spath and infs.

3.2 Concrete Automata

In order for our formalisation of abstract transition systems to be useful, it needs to be able to express a wide range of transition system types and their representations. We now present instantiations for various transition systems with labels of type α and states of type ρ.

Given a successor function "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho\,\mathsf{option}$", we instantiate as follows.

▶ **Example 10** (Incomplete Deterministic Transition System).

$$\begin{aligned}
\mathsf{execute} &= \lambda a\ p.\ \mathsf{the}\ (\mathsf{succ}\ a\ p) & \mathit{transition} &= \alpha \\
\mathsf{enabled} &= \lambda a\ p.\ \mathsf{succ}\ a\ p \neq \mathsf{None} & \mathit{state} &= \rho
\end{aligned}$$

Note how the deterministic successor function fits the interface straightforwardly.

Given a successor function "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho$" we instantiate as follows.

▶ **Example 11** (Complete Deterministic Transition System).

$$\begin{aligned}
\mathsf{execute} &= \mathsf{succ} & \mathit{transition} &= \alpha \\
\mathsf{enabled} &= \top & \mathit{state} &= \rho
\end{aligned}$$

Things get interesting when considering "succ :: $\alpha \Rightarrow \rho \Rightarrow \rho$ set". Textbooks teach us that deterministic transitions systems are a special case of nondeterministic ones. At first glance, it may seem like we are trying to do the impossible opposite here. However, since we get to instantiate the type variables, there is a surprisingly elegant solution.

▶ **Example 12** (Implicit Nondeterministic Transition System).

$$\text{execute} = \lambda(a, q)\ p.\ q \qquad \textit{transition} = \alpha \times \rho$$
$$\text{enabled} = \lambda(a, q)\ p.\ q \in \text{succ}\ a\ p \qquad \textit{state} = \rho$$

Note how unlike in the first two examples, the type variable *transition* gets instantiated in a nontrivial way. While it may seem backwards at first, this actually works out perfectly and gives our constants the strongest possible type for this scenario. For instance, we get "path :: $(\alpha \times \rho)$ list $\Rightarrow \rho \Rightarrow$ bool". That is, the path predicate expects a source state as well as a list of the traversed labels and states. This expression contains exactly the necessary amount of information, nothing more, nothing less. Note that the fact that we are dealing with pairs is not an issue, as Isabelle has good automation for those. We also added some more automation for sequences of pairs as part of this library. In the end, neither the deterministic nor the nondeterministic case necessitates inconvenient wellformedness predicates while sharing the same abstract formalisation.

Finally, we consider an explicit representation in "trans :: $(\rho \times \alpha \times \rho)$ set". Being isomorphic to the previous case, the type variables as well as execute are instantiated the same way.

▶ **Example 13** (Explicit Nondeterministic Transition System).

$$\text{execute} = \lambda(a, q)\ p.\ q \qquad \textit{transition} = \alpha \times \rho$$
$$\text{enabled} = \lambda(a, q)\ p.\ (p, a, q) \in \text{trans} \qquad \textit{state} = \rho$$

Unsurprisingly, an isomorphic change in representation does not make a difference since the instantiation absorbs such details.

Having shown that we can instantiate a variety of transition systems using our abstract theory, we can now use these as building blocks for concrete automata. Since the abstraction is achieved via type instantiation and locales, it only minimally impacts the usability compared to a fully specific formalisation. Moreover, since it does not restrict the type of the automaton at all, the user can use a representation that exactly fits their needs.

There are some definitions that one would expect to be part of a general automata library that unfortunately cannot be formalised on transition systems. One of these are Boolean operations, since they require information about the automaton's successors, labelling, and acceptance condition. With some effort, they could be formalised on intermediate abstraction over a family of similar automata (for instance, DBA, DCA, DRA). However, we could not justify the effort for our purposes, since these formalisations do not contain much substance.

Degeneralisation, which plays an important part in defining aforementioned Boolean operations, can be generalised a little easier. The reason for this is that it is independent from successors and labelling, requiring only the concept of state-based Büchi acceptance. Thanks to this, we were able to abstractly formalise degeneralisation in a transition system locale augmented with an acceptance condition. This intermediate abstraction is then instantiated in order to facilitate the formalisation of Boolean operations on DBAs and DCAs.

3.3 Predefined Automata

While the focus of the automata library is on the abstract part and the provision of tools to build concrete automata, it also comes with a growing collection of the latter. At the time of writing, it contains (non)deterministic finite automata, (non)deterministic Büchi automata, as well as deterministic co-Büchi and Rabin automata. Each of these incurs around 50 lines of proof text in order to set-up the automaton and to define its language. The latter is fairly simple to achieve, as all the constituents (paths and acceptance conditions) are already available and just need to be composed to yield a language definition.

3.4 Executable Implementation

One of our goals is also the ability to implement executable versions of some algorithms. As mentioned earlier, we will use the refinement frameworks and the Isabelle code generator for this. Most of this needs to be done on concrete automata, as it depends on details of the representation. Furthermore, in many cases it is advantageous to be able to choose data structures depending on the representation. Because of these reasons, all the executable implementations are done on the concrete level, with only some proofs being reused.

We build on existing algorithms for graph structures to implement versions that work with automata. For instance, we use the AFP entry about depth-first search [32, 33] to explore all reachable states of an automaton. This is used to generate explicit representations of automata in order to be able to serialise and output them. In the case of NBAs we consider the successor function "$\mathsf{succ} :: \alpha \Rightarrow \rho \Rightarrow \rho\,\mathsf{set}$", which implicitly represents the transitions of the automaton. The algorithm can then turn this into an explicit set of transitions "$\mathsf{trans} :: (\rho \times \alpha \times \rho)\,\mathsf{set}$". We also implement an algorithm for translating an automaton with an arbitrary state type into one whose states are natural numbers. Furthermore, we use the AFP entry about Gabow's algorithm for strongly-connected components [29, 30] to decide language emptiness of NBAs.

3.5 Formalisation

The library is available in the form of the AFP entry *Transition Systems and Automata* [10]. At the time of writing, it comprises about 5800 lines of theory text. Other than in this paper, the library is used in the partial order reduction optimisation [12, 11] of the Cava model checker [21]. It is also used as the foundation of the AFP entry about rank-based complementation of Büchi automata [9].

3.6 Contributions to the Translation Formalisation

For this paper, we contribute deterministic Büchi, co-Büchi, and Rabin automata. For instance, the constructor for deterministic Büchi automata is "$\mathsf{dba} :: \alpha\,\mathsf{set} \Rightarrow \rho \Rightarrow (\alpha \Rightarrow \rho \Rightarrow \rho) \Rightarrow (\rho \Rightarrow \mathsf{bool}) \Rightarrow (\alpha, \rho)\ \mathsf{dba}$". Furthermore, we add corresponding union and intersection operations to the library (Figure 1). In addition to those operations, we also implement a specialised operation dbcrai that provides the intersection of a DBA and a DCA resulting in a DRA. We prove both their correctness in terms of language as well as upper bounds on the number of states of the resulting automata. Since the resulting automata are implicit, we also provide an executable algorithm for exploration and subsequent conversion to an explicit representation together with a numbering of the states.

Automaton	$\cap$ (Pair)	$\bigcap$ (List)	$\cup$ (Pair)	$\bigcup$ (List)
DBA		dbail	dbau	dbaul
DCA	dcai	dcail		dcaul
DRA				draul

Figure 1 Boolean Operations on Deterministic ω-Automata. Shown are the Boolean operations that were implemented for deterministic Büchi, co-Büchi, and Rabin automata.

4 The Master Theorem: Decomposing LTL Formulas

The centrepiece for all translations is the *Master Theorem* [19] that decomposes LTL formulas into a Boolean combination, in our case union and intersection, of "simple" languages. We will recall important definitions from [19] in order to state the theorem itself and to highlight obstacles we encountered in our formalisation. For an in-depth discussion and exposition of the theory and its proof we refer the reader to the primary source [19].

We will now introduce the functions used in the scope of the Master Theorem: the "after"-function $\mathsf{af}\ \varphi\ w$, read "φ after w", and the two "advice" functions $\varphi[X]_\nu$ and $\psi[Y]_\mu$ which are pronounced as "φ with **GF**-advice X" and "ψ with **FG**-advice Y", respectively.

4.1 The "after"-Function

Let us begin with the definition of the "after"-function [18, 17, 19]. The function application $\mathsf{af}\ \varphi\ w$ computes a new formula such that for every infinite word w' we have:

▶ **Lemma 14** ([19]).

$$w \frown w' \models \varphi \iff w' \models \mathsf{af}\ \varphi\ w.$$

We can intuitively see af as a function that returns a formula representing the language that we obtain *after* reading the prefix w. We achieve this by using well-known LTL expansion rules combined with partial evaluation.

▶ **Definition 15** ("after"-Function [19]). *The function* $\mathsf{af} :: \alpha\ \mathsf{ltl} \Rightarrow \alpha\ \mathsf{set} \Rightarrow \alpha\ \mathsf{ltl}$ *is defined for a single letter recursively as follows:*

$$\begin{aligned}
\mathsf{af}\ \mathbf{tt}\ \sigma &= \mathbf{tt} & \mathsf{af}\ (\mathbf{X}\ \varphi)\ \sigma &= \varphi \\
\mathsf{af}\ \mathbf{ff}\ \sigma &= \mathbf{ff} & & \\
\mathsf{af}\ a\ \sigma &= \mathbf{if}\ a \in \sigma\ \mathbf{then}\ \mathbf{tt}\ \mathbf{else}\ \mathbf{ff} & \mathsf{af}\ (\varphi\ \mathbf{U}\ \psi)\ \sigma &= (\mathsf{af}\ \psi\ \sigma) \vee ((\mathsf{af}\ \varphi\ \sigma) \wedge (\varphi\ \mathbf{U}\ \psi)) \\
\mathsf{af}\ (\neg a)\ \sigma &= \mathbf{if}\ a \notin \sigma\ \mathbf{then}\ \mathbf{tt}\ \mathbf{else}\ \mathbf{ff} & \mathsf{af}\ (\varphi\ \mathbf{R}\ \psi)\ \sigma &= (\mathsf{af}\ \psi\ \sigma) \wedge ((\mathsf{af}\ \varphi\ \sigma) \vee (\varphi\ \mathbf{R}\ \psi)) \\
\mathsf{af}\ (\varphi \wedge \psi)\ \sigma &= (\mathsf{af}\ \varphi\ \sigma) \wedge (\mathsf{af}\ \psi\ \sigma) & \mathsf{af}\ (\varphi\ \mathbf{W}\ \psi)\ \sigma &= (\mathsf{af}\ \psi\ \sigma) \vee ((\mathsf{af}\ \varphi\ \sigma) \wedge (\varphi\ \mathbf{W}\ \psi)) \\
\mathsf{af}\ (\varphi \vee \psi)\ \sigma &= (\mathsf{af}\ \varphi\ \sigma) \vee (\mathsf{af}\ \psi\ \sigma) & \mathsf{af}\ (\varphi\ \mathbf{M}\ \psi)\ \sigma &= (\mathsf{af}\ \psi\ \sigma) \wedge ((\mathsf{af}\ \varphi\ \sigma) \vee (\varphi\ \mathbf{M}\ \psi))
\end{aligned}$$

We generalise this definition to finite words by overloading $\mathsf{af} :: \alpha\ \mathsf{ltl} \Rightarrow \alpha\ \mathsf{set\ list} \Rightarrow \alpha\ \mathsf{ltl}$*:*

$$\mathsf{af}\ \varphi\ w = \mathsf{foldl}\ \mathsf{af}\ \varphi\ w.$$

▶ Remark 16. The reader might have noticed that the definition of af resembles the idea of Brzozowski's derivatives for regular expressions [13]. In fact, as we will see later, the DRA construction relies on af and the previously introduced LTL equivalence relations again mirroring the idea of Brzozowski. However, this approach alone can only be applied to fragments of LTL.

4.2 Syntactic Fragments of LTL

We already teased the idea of the "simple" languages, but what is special about these? What is the mechanism to achieve this? These languages are made simple by the fact that they can be expressed by fragments of LTL. To be more precise, let μLTL be the fragment that only contains modal operators that can be expressed as least-fixed points, i.e., we disallow the operators $\mathbf{R}$ and $\mathbf{W}$. Dually, νLTL contains only modal operators that can be expressed as greatest-fixed points, i.e., we disallow the operators $\mathbf{U}$ and $\mathbf{M}$. The fragments $\mathbf{GF}(\mu\text{LTL})$ and $\mathbf{FG}(\nu\text{LTL})$ contain all formulas $\mathbf{GF}\varphi$ and $\mathbf{FG}\psi$ where $\varphi \in \mu\text{LTL}$ and $\psi \in \nu\text{LTL}$, respectively. For these fragments one can easily define translations to NBAs or DRAs, e.g. [19].

Let us now think about how to make use of this: Assume one gets a promise set $X = \{a\,\mathbf{U}\,b\}$ guaranteeing that $a\,\mathbf{U}\,b$ holds infinitely often, i.e., $w \models \mathbf{GF}(a\,\mathbf{U}\,b)$, and assume we have access to a translation for νLTL. Can we simplify $\varphi = \mathbf{G}(a\,\mathbf{U}\,b) \vee \mathbf{G}c$ with this information? Since $w \models \mathbf{GF}(a\,\mathbf{U}\,b)$ implies that b is infinitely often true, we can replace the $\mathbf{U}$ by an $\mathbf{W}$. Under the assumption that X is a correct promise, we simplify φ to an equivalent formula $\mathbf{G}(a\,\mathbf{W}\,b) \vee \mathbf{G}c$ which is a formula of νLTL. Then we can apply our translation for the νLTL fragment.

Formally, we define the functions $\varphi[X]_\nu$ and $\varphi[Y]_\mu$ such that $\varphi[X]_\nu$ takes a promise set X and produces a formula of νLTL, and such that $\varphi[Y]_\mu$ takes a promise set Y and produces a formula of μLTL:

▶ **Definition 17** ("Advice"-Functions [19]). *The function* $\cdot\,[\,\cdot\,]_\nu :: \alpha\,\mathsf{ltl} \Rightarrow \alpha\,\mathsf{ltl\,set} \Rightarrow \alpha\,\mathsf{ltl}$ *is defined for the cases* $\mathbf{U}$ *and* $\mathbf{M}$ *as follows:*

$$(\varphi\,\mathbf{U}\,\psi)[X]_\nu = \textbf{if } (\varphi\,\mathbf{U}\,\psi) \in X \textbf{ then } (\varphi[X]_\nu)\,\mathbf{W}\,(\psi[X]_\nu) \textbf{ else ff}$$
$$(\varphi\,\mathbf{M}\,\psi)[X]_\nu = \textbf{if } (\varphi\,\mathbf{M}\,\psi) \in X \textbf{ then } (\varphi[X]_\nu)\,\mathbf{R}\,(\psi[X]_\nu) \textbf{ else ff}$$

The function $\cdot\,[\,\cdot\,]_\mu :: \alpha\,\mathsf{ltl} \Rightarrow \alpha\,\mathsf{ltl\,set} \Rightarrow \alpha\,\mathsf{ltl}$ *is defined for the cases* $\mathbf{R}$ *and* $\mathbf{W}$ *as follows:*

$$(\varphi\,\mathbf{R}\,\psi)[Y]_\mu = \textbf{if } (\varphi\,\mathbf{R}\,\psi) \in Y \textbf{ then tt else } (\varphi[Y]_\mu)\,\mathbf{M}\,(\psi[Y]_\mu)$$
$$(\varphi\,\mathbf{W}\,\psi)[Y]_\mu = \textbf{if } (\varphi\,\mathbf{W}\,\psi) \in Y \textbf{ then tt else } (\varphi[Y]_\mu)\,\mathbf{U}\,(\psi[Y]_\mu)$$

For all other cases, both functions are defined as a recursive descent over the syntax tree.

4.3 The Master Theorem

We are now equipped with the necessary definitions to state the Master Theorem. Note that the formulation we use is taken nearly verbatim from the Isabelle theory, apart from the annotations $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$ that we added to relate to the introduction.

▶ **Theorem 18** (Master Theorem [19]).

$$\begin{aligned} w \models \varphi \iff (\exists X \subseteq \mathsf{subformulas}_\mu\ \varphi.\ \exists Y \subseteq \mathsf{subformulas}_\nu\ \varphi.&\\ (\exists i.\ \mathsf{suffix}\ i\ w \models \mathsf{af}\ \varphi\ (\mathsf{prefix}\ i\ w)[X]_\nu) &\quad -- L^1_{\varphi,X}\\ \wedge\ (\forall \psi \in X.\ w \models \mathbf{G}\ (\mathbf{F}\ \psi[Y]_\mu)) &\quad -- L^2_{X,Y}\\ \wedge\ (\forall \psi \in Y.\ w \models \mathbf{F}\ (\mathbf{G}\ \psi[X]_\nu)) &\quad -- L^3_{X,Y} \end{aligned}$$

The proof of this theorem intrinsically depends on the fact that we can check promise sets bottom-up, as formalised by the following lemma. We highlight this intermediate lemma, because we needed to introduce a custom induction mechanism over finite sets to our theory. The remaining material needed to show Theorem 18 is obtained in straight-forward manner and closely resembles the proofs of [19].

▶ **Lemma 19** ([19]).

$$\begin{aligned}
&\textbf{fixes } w :: \alpha\,\mathsf{set\,word} \textbf{ and } \varphi :: \alpha\,\mathsf{ltl}\\
&\textbf{assumes } X \subseteq \mathsf{subformulas}_\mu\ \varphi \textbf{ and } Y \subseteq \mathsf{subformulas}_\nu\ \varphi\\
&\textbf{and } \forall \psi \in X.\ w \models \mathbf{G}\ (\mathbf{F}\ \psi[Y]_\mu) \textbf{ and } \forall \psi \in Y.\ w \models \mathbf{F}\ (\mathbf{G}\ \psi[X]_\nu)\\
&\textbf{shows } \forall \psi \in X.\ w \models \mathbf{G}\ (\mathbf{F}\ \psi) \textbf{ and } \forall \psi \in Y.\ w \models \mathbf{F}\ (\mathbf{G}\ \psi)
\end{aligned}$$

The corresponding proof from [19] proceeds by constructing a sequence of pairs (X_i, Y_i) where we have $(X_0, Y_0) = (\emptyset, \emptyset)$ and $(X_n, Y_n) = (X, Y)$. Moreover, in each step a single formula $\psi_i \in X \uplus Y$ is added to either X_i or Y_i, depending on whether $\psi_i \in X$ or $\psi_i \in Y$. However, ψ_i cannot be chosen arbitrarily and ψ_i must respect the subformula order, i.e., if $\psi_i \in \mathsf{sf}\ \psi_j$, then $i \leq j$. Then the proof proceeds by an induction over this sequence.

Since to the best of our knowledge there has been at the time of writing no matching induction rule in Isabelle or its libraries, we derived a suitable induction rule for our purposes. First, note that instead of sorting the formulas by the subformula order, it is sufficient to order them by their size, because all subformulas of a formula φ are smaller than φ. Second, an induction over pairs of sets seemed inconvenient to us in the context of our theorem prover. Hence we combined the two disjoint sets into a single one and used a suitable case distinction. Finally, we arrived at the following, general induction rule[4] for finite sets with an additional order constraint:

▶ **Lemma 20** (Finite Ordered Induction).

$$\begin{aligned}
&\textbf{fixes } S :: \alpha\,\mathsf{set} \textbf{ and } P :: \alpha\,\mathsf{set} \Rightarrow \mathsf{bool} \textbf{ and } f :: \alpha \Rightarrow (\beta :: \mathsf{linorder})\\
&\textbf{assumes } \mathsf{finite}\ S \textbf{ and } P\ \emptyset\\
&\textbf{and } \bigwedge x\ S.\ \mathsf{finite}\ S \wedge (\forall y.\ y \in S \longrightarrow f\ y \leq f\ x) \wedge P\ S \implies P\ (\mathsf{insert}\ x\ S)\\
&\textbf{shows } P\ S
\end{aligned}$$

5 Deriving the DRA Construction

With the necessary decomposition theorem in place, we now can follow our automata construction blue-print to obtain a translation from LTL to DRAs. We will first build automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$, named $\mathfrak{A}_1$, $\mathfrak{A}_2$, and $\mathfrak{A}_3$, respectively. In the subsequent section, we will assemble these pieces to the final automaton and end the section with a description of the extracted, verified tool.

5.1 Constructing Automata for $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$

We parametrise our automata constructions for the "simple" components by an equivalence relation $\sim$. The most important requirement for $\sim$ is that $\sim_c \leq \sim \leq \sim_l$ holds, i.e., that $\sim$ does not consider two formulas with different languages equivalent and $\sim$ eventually detects equivalence to $\mathbf{tt}$ and $\mathbf{ff}$ for certain fragments. This abstraction has two advantages over fixing a concrete equivalence: first, our proofs stay as abstract as possible and the proof automation does not rely accidentally on irrelevant properties of the chosen equivalence relation; second, we can instantiate the final automaton with any suitable equivalence relation. In Section 5.3 we exemplarily use propositional equivalence but one can easily replace it by a different equivalence without any additional effort to speak of.

[4] This induction rule has now been included in Isabelle/HOL, is located in `HOL/Lattices_Big.thy`, and is named `finite_ranking_induct`.

In this paper, we will only discuss the construction of $\mathfrak{A}_2$ for $L^2_{X,Y}$. The constructions for $L^1_{\varphi,X}$ and $L^3_{X,Y}$ as defined by [19] are formalised analogously. Remember that $L^2_{X,Y}$ is defined as "$\bigcap \psi \in X$. language UNIV ($\mathbf{GF}(\psi[Y]_\mu)$)" for the finite sets X and Y. Hence it suffices to define a translation for formulas of the fragment $\mathbf{GF}(\mu\text{LTL})$ and then apply the intersection construction from the automaton library.

For the translation of formulas from the fragment $\mathbf{GF}(\mu\text{LTL})$ we make use of the following lemma. It states the we can monitor a formula from μLTL using af and the constrained equivalence relation $\sim$, and if a word satisfies the formula, then we will notice this after a finite amount of steps. Furthermore, the lemma states that we can deal with $\mathbf{GF}(\mu\text{LTL})$ by repeatedly doing this:

▶ **Lemma 21** (Logical Characterisation of μLTL and $\mathbf{GF}(\mu\text{LTL})$ [19, 42][5]).

$$\begin{aligned}
&\textbf{assumes } \varphi \in \mu\text{LTL} \textbf{ and } \sim_c \leq \sim \leq \sim_l \\
&\quad \textbf{shows } w \models \varphi \iff \exists i.\ \mathsf{af}\ \varphi\ (\mathsf{prefix}\ i\ w) \sim \mathsf{tt} \\
&\quad\quad \textbf{and } w \models \mathbf{G}\ (\mathbf{F}\ \varphi) \iff \forall i.\ \exists j.\ \mathsf{af}\ (\mathbf{F}\ \varphi)\ (\mathsf{prefix}\ j\ (\mathsf{suffix}\ i\ w)) \sim \mathsf{tt}
\end{aligned}$$

Since $\sim$ is such a fundamental ingredient throughout the formalisation of the automata constructions, we use locales in Isabelle to fix $\sim$ and assumptions about it. In particular, we use the equivalence classes of $\sim$ as states in our constructed automata. To define the quotient type for a given equivalence relation, we use the Isabelle's *Quotient* package introduced in [8] and revised in [24]. However, it is not possible to define such a quotient type within a locale. Thus we present a primitive, ad-hoc mechanism to simulate the quotient type in our locale. We fix a type parameter γ and the functions Rep and Abs that compute the representative of an equivalence class and the equivalence class of a formula, respectively. In other words we use Rep and Abs to map between equivalence classes and representatives. Further, we assume the quotient type invariant "Abs (Rep x) = x" and require that equality on γ is equivalent to $\sim$ on formulas. Thus we can pretend γ to be a quotient type over $\sim$ which resembles "duck typing" found in programming languages such as Python.

▶ **Definition 22** (Locale for LTL to DRA translation[6]).

$$\begin{aligned}
&\textbf{locale } \mathsf{ltl\text{-}to\text{-}dra} = \\
&\qquad \textbf{fixes } \sim :: \alpha\ \mathsf{ltl} \Rightarrow \alpha\ \mathsf{ltl} \Rightarrow \mathsf{bool} \\
&\qquad\ \textbf{and } \mathsf{Rep} :: \gamma \Rightarrow \alpha\ \mathsf{ltl}\ \ \textbf{and } \mathsf{Abs} :: \alpha\ \mathsf{ltl} \Rightarrow \gamma \\
&\quad \textbf{assumes } \mathsf{equivp} \sim \textbf{ and } \sim_c \leq \sim \leq \sim_l \\
&\qquad\ \textbf{and } \mathsf{Abs}\ (\mathsf{Rep}\ x) = x\ \textbf{ and } \mathsf{Abs}\ \varphi = \mathsf{Abs}\ \psi \longleftrightarrow \varphi \sim \psi \\
&\qquad\ \textbf{and } \varphi \sim \psi \implies (\mathsf{af}\ \varphi\ \sigma \sim \mathsf{af}\ \psi\ \sigma) \wedge (\varphi[X]_\nu \sim \psi[X]_\nu)
\end{aligned}$$

In this definition two new assumptions can be found that we have not talked about yet: We also demand that af and $\cdot\,[\,\cdot\,]_\nu$ are congruent with respect to $\sim$. This is due to the fact that our the automata use equivalence classes as states and for computing the successor with af the choice of the representative must be irrelevant.

[5] This lemma is a generalised version of [19] which only considers the special case for $\sim_p$.

[6] We only present the final combination of several locales defined in our Isabelle formalisation to give an overview of all assumptions required by our proofs.

Within this locale we now define the deterministic Büchi automaton $\mathfrak{A}_\mu^{\mathbf{GF}}$ for a single formula of the fragment $\mathbf{GF}(\mu\text{LTL})$. The DBA $\mathfrak{A}_2$ for $L^2_{X,Y}$ is then computed by a Büchi intersection (dbail). Note that this intersection construction requires the operands to be ordered. Hence we represent the advice sets X and Y as the lists xs and ys and propagate this order to dbail.

▶ **Definition 23.**

$$\mathfrak{A}_\mu^{\mathbf{GF}}\ \varphi = \mathsf{dba}\ \mathsf{UNIV}\ (\mathsf{Abs}\ (\mathbf{F}\ \varphi))\ (\mathsf{af}_F\ \varphi)\ (\lambda\psi.\ \psi = \mathsf{Abs}\ \mathbf{tt})$$
$$\mathsf{af}_F\ \varphi\ \sigma\ \psi = \mathbf{if}\ \ \psi = \mathsf{Abs}\ \mathbf{tt}\ \ \mathbf{then}\ \ \mathsf{Abs}\ (\mathbf{F}\ \varphi)\ \ \mathbf{else}\ \ \mathsf{Abs}\ (\mathsf{af}\ (\mathsf{Rep}\ \psi)\ \sigma)$$
$$\mathfrak{A}_2\ xs\ ys = \mathsf{dbail}\ (\mathsf{map}\ (\lambda\psi.\ \mathfrak{A}_\mu^{\mathbf{GF}}\ (\psi[\mathsf{set}\ ys]_\mu))\ xs)$$

Using Lemma 21 we show correctness for a single component and using the lemmas from the automata library we also prove the intersection correct. The constructions for $L^1_{\varphi,X}$ and $L^3_{X,Y}$ are analogous and thus skipped from the presentation in this paper.

5.2 Assembling the Pieces

It now remains to intersect the (co-)Büchi automata "$\mathfrak{A}_1\ \varphi\ xs$", "$\mathfrak{A}_2\ xs\ ys$", and "$\mathfrak{A}_3\ xs\ ys$", representing $L^1_{\varphi,X}$, $L^2_{X,Y}$, and $L^3_{X,Y}$, respectively. Again we need to use a list representation for X and Y to fix an iteration order and thus we use xs and ys. We call the resulting Rabin automaton "$\mathfrak{A}\ \varphi\ xs\ ys$". To finish the construction, we then iterate over all possible choices for $X \subseteq \mathsf{subformulas}_\mu\ \varphi$ and $Y \subseteq \mathsf{subformulas}_\nu\ \varphi$ and take the union of all languages accepted by "$\mathfrak{A}\ \varphi\ xs\ ys$" with draul (DRA union):

▶ **Definition 24.**

$$\mathsf{ltl\text{-}to\text{-}dra}\ \varphi = \mathsf{draul}\ (\mathsf{map}\ (\lambda(xs, ys).\ \mathfrak{A}\ \varphi\ xs\ ys)\ (\mathsf{advice\text{-}sets}\ \varphi)).$$

Using the Master Theorem (Theorem 18) and the correctness lemmas for the intermediate constructions, we obtain the correctness of the translation:

▶ **Theorem 25.**

$$\mathsf{language}\ (\mathsf{ltl\text{-}to\text{-}dra}\ \varphi) = \mathsf{language}\ \mathsf{UNIV}\ \varphi.$$

5.3 A Verified LTL Translator

We extract the executable translation of LTL formulas into ω-automata by instantiating the locale with a suitable equivalence relation. As mentioned above we use $\sim_p$ and we show for this equivalence relation that the constructed automaton indeed has at most a double-exponential number of states in the size of the formula. Hence an exploration by depth-first search terminates, and more importantly, this makes the construction the first LTL to DRA translation with a formally verified double exponential size bound.

▶ **Lemma 26.**

$$\mathsf{card}\ (\mathsf{nodes}\ (\mathsf{ltl\text{-}to\text{-}dra}\ \varphi)) \leq 2\ \hat{}\ 2\ \hat{}\ (2 * \mathsf{size}\ \varphi + \mathsf{floorlog}\ 2\ (\mathsf{size}\ \varphi) + 4).$$

Exporting code for the LTL part needs only minor adjustments through code lemmas, e.g. we instantiate $\sim_p$ with code provided by [35]. For the parts related to automata we rely on the code export feature of the automata library, see Section 3.4. Notice that Theorem 25

refers to the potentially infinite alphabet UNIV. Choosing UNIV as the alphabet simplified the proofs leading up to the result, but potentially infinite alphabets make an exploration using depth-first search using a naive enumeration of letters impossible. Consequently, we restrict the alphabet to a finite set for the code export by only considering atomic propositions occurring in φ. The resulting constant ltl-to-draei has the signature α ltl $\Rightarrow$ (α set, nat) draei which is then exported to Standard ML. The overall correctness theorem is as follows:

▶ **Theorem 27.**

language (draei-dra (ltl-to-draei φ)) = language (Pow (atoms φ)) φ.

Note that the constant language is only defined for DRAs with a transition function (dra) while we obtain from the translation a DRA with a list of transitions (draei). The constant draei-dra converts an automaton of type draei back to one of type dra.

In the final tool, we combine the function ltl-to-draei with an unverified LTL parser and an unverified serialisation to the Hanoi Omega Automata format [3], a text-based format for representing ω-automata. It is then compiled with `mlton` or `polyc` using the build scripts included in the formalisation [39].

▶ **Example 28.** The following command translates the formula $\mathbf{FG}a$ to a DRA in HOA format and then, using `autfilt` from Spot [16], prints it in the `dot`-format. The result gets rendered by `dot` and is written to a PDF file.

```
./ltl_to_dra "F G a" | autfilt --dot --merge-transitions | dot -Tpdf -O
```

6 Concluding Remarks

The formalisation of the "Master Theorem" itself did not pose major obstacles and did not require special care except for the mentioned techniques. However, the LTL entry [41] and dependencies are host to several LTL datatypes and matching lemmas and notation. This excessive amount of copy-pasting is due the inability to define fragments of datatypes, i.e., restrictions on the constructors used. While one could use **typedef** to carve out restricted types using a predicate, this new type misses the structure of the type we started with. Thus we choose in some cases to have separate datatypes connected by translations, while in other cases we used simple predicates to capture fragments. We think the addition of a mechanism addressing this issue – the definition of datatype fragments and the addition of necessary constants and proof automation – would be worthwhile, since we conjecture it would significantly reduce the size and complexity of LTL related theories.

There are several topics we want to investigate going forward: First, we also want to derive constructions for NBAs and LDBAs. Second, we plan to reduce the size of the generated automata by restricting the possible choices for the advice sets X and Y. Third, we want to provide implementations using better instantiations for the equivalence relation to further reduce the size of the computed automata. Fourth, provide constructions for DRA variants, e.g., transition-based or generalised acceptance. Fifth, while adding some of the Boolean operations, we realised that constructions for ω-automata could potentially be shared and consolidated in an intermediate abstraction.

References

1 Romain Aïssat, Frédéric Voisin, and Burkhart Wolff. Infeasible Paths Elimination by Symbolic Execution Techniques: Proof of Correctness and Preservation of Paths. *Archive of Formal Proofs*, 2016, 2016. URL: `https://www.isa-afp.org/entries/InfPathElimination.shtml`.

2 Tomás Babiak, Thomas Badie, Alexandre Duret-Lutz, Mojmír Křetínský, and Jan Strejcek. Compositional Approach to Suspension and Other Improvements to LTL Translation. In Ezio Bartocci and C. R. Ramakrishnan, editors, *Model Checking Software - 20th International Symposium, SPIN 2013, Stony Brook, NY, USA, July 8-9, 2013. Proceedings*, volume 7976 of *Lecture Notes in Computer Science*, pages 81–98. Springer, 2013. doi:10.1007/978-3-642-39176-7_6.

3 Tomás Babiak, Frantisek Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejcek. The Hanoi Omega-Automata Format. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 479–486. Springer, 2015. doi:10.1007/978-3-319-21690-4_31.

4 Tomás Babiak, Frantisek Blahoudek, Mojmír Křetínský, and Jan Strejcek. Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F, G)-Fragment. In Dang Van Hung and Mizuhito Ogawa, editors, *Automated Technology for Verification and Analysis - 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings*, volume 8172 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2013. doi:10.1007/978-3-319-02444-8_4.

5 Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

6 Clemens Ballarin. Locales: A Module System for Mathematical Theories. *J. Autom. Reasoning*, 52(2):123–153, 2014. doi:10.1007/s10817-013-9284-7.

7 Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly Modular (Co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 93–110. Springer, 2014. doi:10.1007/978-3-319-08970-6_7.

8 Maksym Bortin and Christoph Lüth. Structured Formal Development with Quotient Types in Isabelle/HOL. In Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors, *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*, volume 6167 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2010. doi:10.1007/978-3-642-14128-7_5.

9 Julian Brunner. Büchi complementation. *Archive of Formal Proofs*, 2017, 2017. URL: https://www.isa-afp.org/entries/Buchi_Complementation.html.

10 Julian Brunner. Transition Systems and Automata. *Archive of Formal Proofs*, 2017, 2017. URL: https://www.isa-afp.org/entries/Transition_Systems_and_Automata.html.

11 Julian Brunner. Partial Order Reduction. *Archive of Formal Proofs*, 2018, 2018. URL: https://www.isa-afp.org/entries/Partial_Order_Reduction.html.

12 Julian Brunner and Peter Lammich. Formal Verification of an Executable LTL Model Checker with Partial Order Reduction. *J. Autom. Reasoning*, 60(1):3–21, 2018. doi:10.1007/s10817-017-9418-4.

13 Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.

14 Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018. doi:10.1007/978-3-319-10575-8.

15 Costas Courcoubetis and Mihalis Yannakakis. The Complexity of Probabilistic Verification. *J. ACM*, 42(4):857–907, 1995. doi:10.1145/210332.210339.

16 Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 - A Framework for LTL and ω-Automata Manipulation. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129, 2016. doi:10.1007/978-3-319-46520-3_8.

17 Javier Esparza, Jan Křeínský, and Salomon Sickert. From LTL to deterministic automata - A safraless compositional approach. *Formal Methods in System Design*, 49(3):219–271, 2016. doi:10.1007/s10703-016-0259-2.

18 Javier Esparza and Jan Křetínský. From LTL to Deterministic Automata: A Safraless Compositional Approach. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 192–208. Springer, 2014. doi:10.1007/978-3-319-08867-9_13.

19 Javier Esparza, Jan Křetínský, and Salomon Sickert. One Theorem to Rule Them All: A Unified Translation of LTL into ω-Automata. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 384–393. ACM, 2018. doi:10.1145/3209108.3209161.

20 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 2013. doi:10.1007/978-3-642-39799-8_31.

21 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/CAVA_LTL_Modelchecker.shtml.

22 Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001. doi:10.1007/3-540-44585-4_6.

23 Rob Gerth, Doron A. Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In Piotr Dembinski and Marek Sredniawa, editors, *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995*, volume 38 of *IFIP Conference Proceedings*, pages 3–18. Chapman & Hall, 1995.

24 Brian Huffman and Ondrej Kuncar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *Lecture Notes in Computer Science*, pages 131–146. Springer, 2013. doi:10.1007/978-3-319-03545-1_9.

25 Simon Jantsch and Michael Norrish. Verifying the LTL to Büchi Automata Translation via Very Weak Alternating Automata. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2018. doi:10.1007/978-3-319-94821-8_18.

26 Peter Lammich. Refinement for Monadic Programs. *Archive of Formal Proofs*, 2012, 2012. URL: https://www.isa-afp.org/entries/Refine_Monadic.shtml.

27 Peter Lammich. Automatic Data Refinement. *Archive of Formal Proofs*, 2013, 2013. URL: https://www.isa-afp.org/entries/Automatic_Refinement.shtml.

28 Peter Lammich. The CAVA Automata Library. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/CAVA_Automata.shtml.

29 Peter Lammich. Verified Efficient Implementation of Gabow's Strongly Connected Component Algorithm. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2014. doi:10.1007/978-3-319-08970-6_21.

30 Peter Lammich. Verified Efficient Implementation of Gabow's Strongly Connected Components Algorithm. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/Gabow_SCC.shtml.

31 Peter Lammich and Markus Müller-Olm. Formalization of Conflict Analysis of Programs with Procedures, Thread Creation, and Monitors. *Archive of Formal Proofs*, 2007, 2007. URL: https://www.isa-afp.org/entries/Program-Conflict-Analysis.shtml.

32 Peter Lammich and René Neumann. A Framework for Verifying Depth-First Search Algorithms. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 137–146. ACM, 2015. doi:10.1145/2676724.2693165.

33 Peter Lammich and René Neumann. A Framework for Verifying Depth-First Search Algorithms. *Archive of Formal Proofs*, 2016, 2016. URL: https://www.isa-afp.org/entries/DFS_Framework.shtml.

34 Stephan Merz. Weak Alternating Automata in Isabelle/HOL. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, volume 1869 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2000. doi:10.1007/3-540-44659-1_26.

35 Tobias Nipkow. Boolean Expression Checkers. *Archive of Formal Proofs*, 2014, 2014. URL: https://www.isa-afp.org/entries/Boolean_Expression_Checkers.shtml.

36 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

37 Tobias Nipkow and Dmitriy Traytel. Unified Decision Procedures for Regular Expression Equivalence. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2014. doi:10.1007/978-3-319-08970-6_29.

38 Alexander Schimpf, Stephan Merz, and Jan-Georg Smaus. Construction of Büchi Automata for LTL Model Checking Verified in Isabelle/HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 424–439. Springer, 2009. doi:10.1007/978-3-642-03359-9_29.

39 Benedikt Seidl and Salomon Sickert. A Compositional and Unified Translation of LTL into ω-Automata. *Archive of Formal Proofs*, 2019, 2019. URL: https://isa-afp.org/entries/LTL_Master_Theorem.html.

40 Salomon Sickert. Converting Linear Temporal Logic to Deterministic (Generalised) Rabin Automata. *Archive of Formal Proofs*, 2015, 2015. URL: https://www.isa-afp.org/entries/LTL_to_DRA.shtml.

41 Salomon Sickert. Linear Temporal Logic. *Archive of Formal Proofs*, 2016, 2016. URL: https://www.isa-afp.org/entries/LTL.shtml.

42 Salomon Sickert. *A Unified Translation of Linear Temporal Logic to ω-Automata*. PhD thesis, Technical University Munich, Germany, 2019.

43 Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Křetínský. Limit-Deterministic Büchi Automata for Linear Temporal Logic. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 312–332. Springer, 2016. doi:10.1007/978-3-319-41540-6_17.

44 Moshe Y. Vardi. Automatic Verification of Probabilistic Concurrent Finite-State Programs. In *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*, pages 327–338. IEEE Computer Society, 1985. doi:10.1109/SFCS.1985.12.