Rabinizer 4: From LTL to Your Favourite Deterministic Automaton

Jan Křetínský, Tobias Meggendorfer, Salomon Sickert, and Christopher Ziegler

Technical University of Munich

Abstract. We present Rabinizer 4, a tool set for translating formulae of linear temporal logic to several types of deterministic ω-automata. The tool implements and optimizes several recent constructions, including the first implementation translating the frequency extension of LTL. We also provide a distribution of PRISM that links Rabinizer and allows for several model checking procedures for probabilistic systems that are not yet in the official PRISM distribution. Further, we describe a library Owl, which forms the backbone of Rabinizer, but can also be re-used for general work with omega-automata and implementing other constructions. Finally, we evaluate the performance and in cases with any previous implementations we show significant enhancement both in terms of the size of the automata and computational time, due to algorithmic as well as implementation improvements.

1 Introduction

Automata-theoretic approach [VW86] is a key technique for verification and synthesis of systems with linear-time specifications, such as formulas of linear temporal logic (LTL) [Pnu77]. Verification of non-deterministic systems proceeds as follows: first, the formula is translated into a corresponding automaton; second, the product of the system and the automaton is further analyzed. For this purpose, mostly non-deterministic Büchi automata (NBA) are used [Cou99,DGV99,EH00,SB00,GO01,GL02,Fri03,BKŘS12,DLLF+16] since they are typically very small and easy to produce. The size of the automaton is important as it directly affects the size of the product and thus typically also the analysis time. In contrast to verification of non-deterministic systems, verification of probabilistic systems, such as Markov decision processes (MDP), or synthesis require either more involved techniques, e.g. [KPV06], restrictions to logical fragments, e.g. [AT04,BJP+12], or other types of automata than NBA as detailed below.

Probabilistic LTL model checking of MDP cannot profit directly from NBA. Even the qualitative questions, whether the formula holds with probability 0 or 1, require automata with at least a restricted form of determinism. The prime example are the limit-deterministic (also called semi-deterministic) Büchi automata (LDBA) [Var85,CY88]. For the general quantitative questions, where the probability of satisfaction is computed, general limit-determinism is not sufficient. Instead, deterministic Rabin automata (DRA) have been mostly

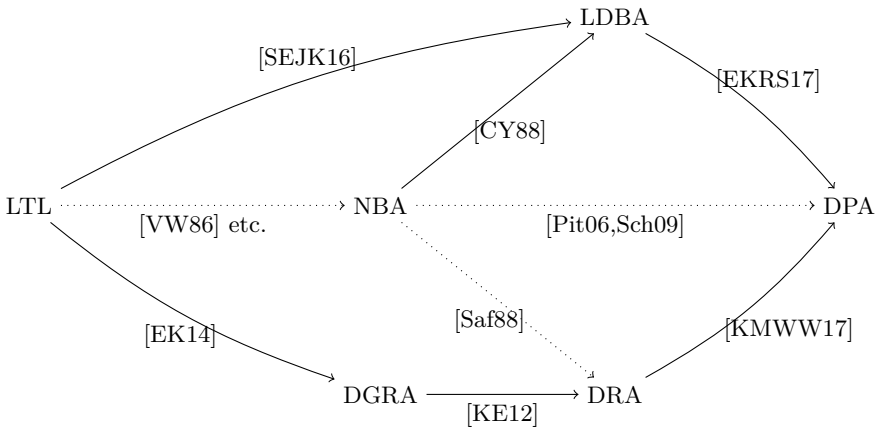

Fig. 1. LTL translations to different types of automata. Translations implemented in Rabinizer 4 are indicated with a solid line. The traditional approaches to obtain DRA and DPA are depicted as dotted arrows. The determinization of NBA is implemented to DRA in ltl2dstar [Kle] and to (mostly) DPA in spot [DLLF+16].

used [BK08,KNP11] and recently also deterministic generalized Rabin automata (DGRA) [CGK13]. In principle, all standard types of deterministic automata are applicable here except for deterministic Büchi automata (DBA), which are not as expressive as LTL. However, other types of automata, such as deterministic Muller and deterministic parity automata (DPA) are typically larger than DGRA. Recently, several approaches with specific LDBA were proved applicable to the quantitative setting [HLS+15,SEJK16] and competitive with DGRA. Besides, model checking MDP against LTL properties involving frequency operators also allows for an automata-theoretic approach, using deterministic generalized Rabin mean-payoff automata DGRMA [FKK15].

LTL synthesis can be similarly solved using the automata-theoretic approach. Since the system is a game (between inputs and outputs), two opposing non-determinisms are present already and the third non-determinism of the automaton would make the product hard to analyze. Therefore, good-for-games automata [HP06] have been considered. Further, fully deterministic DRA and DGRA can also be used, but the algorithms for Rabin games [PP06] are not very efficient in practice. In contrast, DPA may be larger, but in this setting they are the automata of choice due to the good practical performance of parity-game solvers [FL09,ML16].

Types of translations. The translations of LTL to NBA, e.g. [VW86], are typically *"semantic"* in the sense that each state is given by a set of logical formulae and the language of the state can be captured in terms of semantics of these formulae. However, the subsequent determinization of Safra [Saf88] and its improvements [Pit06,Sch09], typically ignore this structure, which also results in producing unnecessarily large automata. In contrast, the recent works [KE12,EK14,KV15,SEJK16,EKRS17,KV17] provide "semantic" constructions,

resulting in smaller automata. Furthermore, the transformations such as degeneralization [KE12], index appearance record [KMWW17] or determinization of limit-deterministic automata [EKRS17] preserve the semantic description, allowing for further optimizations of the resulting automata.

Our contribution. While the previous versions of Rabinizer [KK14] featured only the translation LTL→DGRA→DRA, Rabinizer 4 now implements all the translations depicted by the solid arrows in Fig. 1, covering all the discussed recent semantic translations and more, for instance, the first implementation of the proposed translation of a frequency extension of LTL [FKK15].

Further, in order to utilize the resulting automata for verification, we provide our own distribution of the PRISM model checker, which allows for model checking MDP against LTL using not only DRA and DGRA, but also LDBA and for the first time also against frequency LTL using DGRMA. (Merging these features into the public release of PRISM [KNP11] is subject to current collaboration with the authors of PRISM.) Moreover, the produced DPA can be turned into parity games between the players with input and output variables. Therefore, when linked to parity-game solvers, Rabinizer 4 can be also used for LTL synthesis.

Beside the basic functionality of Rabinizer, we provide a re-usable library Owl (Omega-Words and automata Library), which forms the backbone of Rabinizer. It provides functionality not only for the semantic translations, but generally for ω-automata (featuring basic operations, transformations, state-space reductions etc.), LTL (parsing, transformations, simplifications etc.), and auxiliary services (optimized Java BDD library, command-line interface for easy linking of the mentioned functionalities, setting up server for parallel processing of formulae and automata, I/O etc.).

Due to the highly modular structure of Owl, Rabinizer 4 and any other translations built on top of Owl can readily profit from the improvements of the library. Our experience with Master students has demonstrated that a tool for a complex translations, such as the semi-determinization of generalized NBA to LDBA [BDK$^+$17], can be easily implemented on top of Owl using less than 100 lines of code. One simply defines the mathematical type of the state space, the initial state, the successor function, and the indicator function of accepting states, whereas the rest is taken care of by the infrastructure provided by the library. Furthermore, the continuous development of the library resulted in seamless performance improvements of the translations.

To summarize, our contribution is the following:

- We improve algorithms for several constructions and optimize the respective implementations. We also implement the translation of a frequency extension of LTL [FKK15] for the first time.
- The tool comes with a distribution of PRISM, allowing for model-checking procedures using the provided automata and the first procedure for DGRMA and the frequency extension of LTL.
- The tool is built on top of a re-usable library, useful for processing ω-automata and development of further translations, both on the research and educational level.

2 Preliminaries

Before we describe Rabinizer 4, we briefly recall linear temporal logic (LTL) [Pnu77] and the functionality of Rabinizer 3 [KK14].

We consider the following syntax of LTL

$$\varphi ::= a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \mathbf{X}\varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi \mid \varphi\mathbf{U}\varphi \mid \varphi\mathbf{W}\varphi \mid \varphi\mathbf{R}\varphi \mid \varphi\mathbf{M}\varphi$$

with standard Boolean connectives and the temporal operators next, eventually, globally, until, weak until, release, and strong release, respectively. Although most of the operators are redundant in terms of expressive power, they are important for conciseness and practical efficiency of the translations.

Additionally, we consider a specific fragment $\mathrm{LTL}_{\setminus \mathbf{GU}}$ [KLG13] with the temporal operators $\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$, and $\mathbf{U}$, where $\mathbf{U}$ does not occur in the scope of $\mathbf{G}$.

Finally, we consider the *frequential globally* construct [BDL12,BMM14] $\mathbf{G}^{\sim\rho}\varphi$ with $\sim \in \{\geq, >, \leq, <\}, \rho \in [0, 1]$. Intuitively, it means that the fraction of positions satisfying φ satisfies $\sim\rho$. Formally, the fraction on an infinite run is defined using the long-run average [BMM14].

Rabinizer 3 has the following functionality:

- It translates LTL formulae with operators $\mathbf{X}$, $\mathbf{F}$, $\mathbf{G}$, and $\mathbf{U}$ into equivalent DGRA or DRA; moreover, a choice between state-based and transition-based acceptance is available.
- It is linked to PRISM, allowing for probabilistic verification using DGRA (previously PRISM could only use DRA).

3 Functionality

Rabinizer 4 is accessible at `http://www7.in.tum.de/~kretinsk/rabinizer4.html` together with an online demo, basic usage instructions and examples as well as the full manual.

3.1 Translations

Rabinizer 4 inputs formulae of the extended syntax of LTL defined above. The output format for automata is the standard HOA format [BBD+15], making it accessible to other post-processing tools such as Spot as well as the probabilistic model checker PRISM and to transformers into games for synthesis developed for this standard format.

It features the following command-line tools for the respective translations depicted as the solid arrows in Fig. 1:

ltl2dgra and **ltl2dra** correspond to the original functionality of Rabinizer 3, i.e., they translate LTL (now with the extended syntax) to DGRA and DRA [EK14], respectively.

ltl2ldba translates LTL to LDBA using two modes:
- The default mode is to use the construction of [SEJK16].
- The alternative mode requires a tool for translating LTL to NBA, e.g. spot [DLLF+16], and then transforms the resulting NBA to LDBA using the procedure of [CY88].

ltl2dpa translates LTL to DPA using two modes:
- The default mode uses the translation to LDBA, followed by a LDBA-to-DPA determinization [EKRS17] specially tailored to LDBA with the "semantic" labelling of states, avoiding additional exponential blow-up of the resulting automaton.
- The alternative mode uses the translation to DRA, followed by an improvement of the index appearance record [KMWW17].

fltl2dgrma translates the frequency extension of $\mathrm{LTL}_{\setminus \mathbf{GU}}$, i.e. $\mathrm{LTL}_{\setminus \mathbf{GU}}$ with $\mathbf{G}^{\sim\rho}$, to DGRMA using the construction of [FKK15]. This is the first implementation of this procedure.

All the resulting automata can be output in the standard HOA format and thus also directly visualized. The visualization can also display the "semantic" description of the states. Moreover, the default transition-based acceptance can be changed into the state-based acceptance.

3.2 Verification and Synthesis

The resulting automata can be used for model checking probabilistic systems and for LTL synthesis. To this end, we provide our distribution of the probabilistic model checker PRISM as well as a procedure transforming automata into games to be solved.

Model Checking: PRISM distribution For model checking Markov chains and Markov decision processes, PRISM [KNP11] in line with tradition [BK08] used DRA, yielded by a re-implementation of ltl2dstar [Kle]. Since DGRA are more compact and pose negligible model-checking overhead compared to DRA of the same size, they have been shown superior for probabilistic model checking and implemented within PRISM [CGK13,KK14]. Subsequently, the official distribution of PRISM allowed for linking external DRA translators, such as Rabinizer, and also enabled the use of DGRA.

On the top, our distribution, which links Rabinizer, features model checking using the LDBA [SEJK16,SK16] that are created by our ltl2ldba. Note that general LDBA, in contrast, cannot be used for quantitative model checking. This approach has been shown mostly superior to DGRA [SK16].

Furthermore, the distribution provides the first implementation of frequency $\mathrm{LTL}_{\setminus \mathbf{GU}}$ model checking, using DGRMA. To the best of our knowledge, there are no other implemented procedures for logics with frequency. To this end, techniques of linear programming for multi-dimensional mean-payoff satisfaction [CKK15] and the model-checking procedure of [FKK15] are implemented and applied.

Synthesis: Games The automata-theoretic approach to LTL synthesis requires to transform the LTL formula into a game of the input and output players. This game can be obtained from the corresponding deterministic automaton and the partitioning of atomic propositions into input and output signals. The resulting game essentially arises from the automaton by adding copies of some states and has a bipartite graph structure. As such it can be represented in the standard automata format, e.g. HOA.

We provide this transformer and thus an end-to-end LTL synthesis solution, provided a respective game solver is linked. Since current solutions to Rabin games are not very efficient, we recommend to use PG Solver [FL09] and we provide a transformation of DPA into parity games and the re-formatting from HOA to the format of PG solver.

3.3 Library

Owl (Omega-Words and automata Library) arose from the infrastructure necessary for implementing Rabinizer and turned into a stand-alone re-usable library for basic manipulation of LTL and ω-automata, their conversions and specific needs of "semantic" translations. In short, it features automata data structures, graph algorithms, a BDD library, representations for LTL, simplifiers, and input and output facilities.

LTL The basic functionality includes parsing, semantic simplifications, rewriting of operators, normal forms conversions, substitutions etc. Besides, we provide a Java re-implementation of the parser for the high-level synthesis language TLSF to LTL [JKS16], which can thus be directly processed.

Further, efficient rewriting according to the expansion laws [BK08] is supported. The expansion laws describe temporal formulae in a recursive fashion, decomposing them into directly checkable assertions on the current position and assertions on the direct successor, e.g. $a\mathbf{U}b \equiv b \vee (a \wedge \mathbf{X}(a\mathbf{U}b))$. As such they are at the heart of both classical, e.g. tableaux-based, as well as recent semantic translations.

Automata operations The library covers both deterministic and non-deterministic automata. It works natively with transition-based acceptance and offers a conversion to and from state-based acceptance, supporting all HOA-definable generic acceptance conditions.

The basic functionality includes efficient data structures for storing large automata, procedures for direct modification of the transition structure and the acceptance condition, basic adjustments such as removal of non-accepting or unreachable parts of the state space, completing the transition function, or decomposition into strongly connected components (SCC).

Besides, we provide simplifiers for various acceptance conditions such as colour reduction for DPA or the following procedure for DGRA: Apart from the standard set-based subsumptions checks, we also apply topological arguments. We simplify acceptance sets for each SCC separately and then merge compatible acceptance sets from different SCCs. This reduces the

overall number of acceptance sets, which is usually the most important aspect of the acceptance condition from the perspective of computational complexity, e.g. for probabilistic model checking.

Furthermore, arbitrary transition systems can be generated by solely specifying the initial state, and the successor function. New constructions can thus be coded in a few lines. Moreover, the automata can be conveniently generated on the fly, efficiently streaming huge output files, arising for instance due to large alphabets.

Automata conversions We provide the automata conversions shown in Fig. 1:

dgra2dra degeneralization of DGRA to DRA [KE12]

dra2dpa conversion of DRA to DPA by a newly improved version of index appearance record of [KMWW17]

nba2ldba semi-determinization of NBA to LDBA [CY88]

nba2dpa determinization of NBA via LDBA (by nba2ldba) to DPA [EKRS17]

Command-line interface The library can be operated by a convenient pipe-style CLI, making easy to specify the type and the source of inputs, sequence of procedures (translations, conversions, pre- and post-processing) to be performed, the desired statistics and the output formats. For example, `owl --- stream --- ltl --- ltl2dgra --- aut-stat "DGRA states: %s" --- dgra2dra --- dra2dpa --- aut-stat "DPA states: %s" --- null` inputs a stream of LTL formulae on the standard input, processes them to DGRA and to DPA, while outputting the respective sizes of the automata and not outputting the automata (e.g. to speed up benchmarking).

The input and output is automatically and transparently taken care of. Moreover, we support several sources and sinks for data. While one can simply read and write from standard I/O or files, we also added a basic server-mode, where each incoming connection is treated as such a source and sink for data. The framework can also be easily extended to, for example, obtain data from JSON messages.

Additionally, we also provide transparent support for parallel processing of inputs. With a single switch, one thread is started for each input, speeding up batch-processing of large input sets significantly. The framework also takes care of, e.g., aggregating and outputting the results in the correct order.

For more detail, see Appendix A.

4 Optimizations and Implementation

Compared to the theoretical constructions and previous implementations, there are numerous improvements, heuristics and engineering enhancements.

4.1 Algorithmic Improvements and Heuristics

Now we describe algorithmic improvements for each of the constructions.

ltl2dgra and **ltl2dra** The principle of these translations is to create a master automaton monitoring the satisfaction of the given formula and a dedicated slave automaton for each subformula of the form $\mathbf{G}\psi$, which monitors the satisfaction of ψ in each step and determines whether *all but finitely many* positions satisfy ψ. This information is passed to slaves for larger formulae and to the master. The final automaton is then a product of the master and all the slaves. There are two issues regarding attention:

- Firstly, it is not the case that at all times all slaves have to be actively monitoring. For instance, for a formula $\mathbf{F}a \wedge \mathbf{FG}b$ there is no need to monitor whether b's keep on appearing unless we have already satisfied a. (Intuitively, if a never holds the whole formula cannot be satisfied; otherwise, not noticing several b's at the beginning does not make any difference.) Such slaves can be temporarily "suspended" (in the spirit of [BBDL$^+$13]) and not appear in the final product at these places, thereby decreasing the number of states. Rabinizer 3 has a heuristic for deciding suspendability based on the logical form of the formulae labelling each state.

  Rabinizer 4 goes further and adds another criterion, based on the topology of the complete automaton. In essence, only subformulae relevant for all states in a strongly connected component need to be monitored. In particular, in transient states all slaves are suspended. We illustrate the effect in Table **??**.
- Secondly, since slaves monitor satisfaction of a formula at all positions, they are infinite automata, which further need to be turned into finite ones, actually DRA. In technical terms, this is done using "ranks" [EK14] deciding which representative positions to monitor. We show this general approach can be simplified for some classes of slaves, e.g. for subformulae $\mathbf{G}\psi$ or $\mathbf{GF}\psi$ where the only temporal operators in ψ are $\mathbf{X}$. In particular for slaves with acyclic graphs, no ranks are needed and all positions can be tracked without this information (in the spirit of [KLG13]), typically decreasing the resulting size. For further experiments, see Section 5.

ltl2ldba The principle of this translation differs from the previous ones in that (i) the monitoring of $\mathbf{G}\psi$ starts at a non-deterministic moment, in terms of [SEJK16] a non-deterministic transition called "jump" can be taken, and (ii) from that moment on the slave monitors whether *all* positions satisfy ψ, which is achieved by means of a break-point construction.

Again, we provide a heuristic to determine whether for a given ψ we can simplify the construction. Concretely, we determine whether ψ is syntactically a safety, i.e. ψ contains only $\mathbf{G}$, $\mathbf{R}$, $\mathbf{X}$, or a liveness property, i.e. containing only $\mathbf{F}$, $\mathbf{U}$, $\mathbf{X}$, starting with $\mathbf{F}$, in which case we can avoid break-points. For safety properties there is a notable state space reduction, see the upper part of Table 1. Interestingly, for liveness properties the optimisation itself has no effect when the generalised LBDA is produced. However, if the degeneralization to standard LDBA is switched on, the number of states drops, see the third line of the lower part of Table 1.

Table 1. Effect of break-point elimination for ltl2ldba on safety formulae $s(n,m) = \bigwedge_{i=1}^{n} \mathbf{G}(a_i \vee \mathbf{X}^m b_i)$ and liveness formulae $l(n,m) = \bigwedge_{i=1}^{n} \mathbf{GF}(a_i \wedge \mathbf{X}^m b_i)$, displaying #states (#Büchi conditions)

	$s(1,1)$	$s(2,1)$	$s(3,1)$	$s(4,1)$	$s(1,2)$	$s(2,2)$	$s(3,2)$	$s(4,2)$
original	3 (1)	9 (2)	27 (3)	81 (4)	8 (1)	64 (2)	512 (3)	4096 (4)
improved	2 (1)	4 (1)	8 (1)	16 (1)	4 (1)	16 (1)	64 (1)	256 (1)
	$s(1,3)$	$s(2,3)$	$s(3,3)$	$s(4,3)$	$s(1,4)$	$s(2,4)$	$s(3,4)$	$s(4,4)$
original	20 (1)	400 (2)	$8 \cdot 10^3$(3)	$16 \cdot 10^4$(4)	48 (1)	2304 (2)	-	-
improved	8 (1)	64 (1)	512 (1)	4096 (1)	16 (1)	256 (1)	4096 (1)	65536 (1)

	$l(1,1)$	$l(2,1)$	$l(3,1)$	$l(4,1)$	$l(1,2)$	$l(2,2)$	$l(3,2)$	$l(4,2)$
original	2 (1)	4 (2)	8 (3)	16 (4)	4 (1)	16 (2)	64 (3)	256 (4)
improved	2 (1)	4 (2)	8 (3)	16 (4)	4 (1)	16 (2)	64 (3)	256 (4)
imp.+degen.	2 (1)	4 (1)	6 (1)	8 (1)	4 (1)	8 (1)	12 (1)	16 (1)
	$l(1,3)$	$l(2,3)$	$l(3,3)$	$l(4,3)$	$l(1,4)$	$l(2,4)$	$l(3,4)$	$l(4,4)$
original	8 (1)	64 (2)	512 (3)	4096 (4)	16 (1)	256 (2)	4096 (3)	65536 (4)
improved	8 (1)	64 (2)	512 (3)	4096 (4)	16 (1)	256 (2)	4096 (3)	65536 (4)
imp.+degen.	8 (1)	16 (1)	24 (1)	32 (1)	16 (1)	32 (1)	48 (1)	64 (1)

Table 2. Effect of non-determinism of the initial component for ltl2ldba on formulae $f(i) = \mathbf{F}(a \wedge \mathbf{X}^i \mathbf{G} b)$, displaying #states (#Büchi conditions)

	$f(1)$	$f(2)$	$f(3)$	$f(4)$	$f(5)$	$f(6)$
original	4 (1)	6 (1)	10 (1)	18 (1)	34 (1)	66 (1)
imp.+nondet	2 (1)	3 (1)	4 (1)	5 (1)	6 (1)	7 (1)

Further, we add an option to generate a non-deterministic initial component for the LDBA instead of a deterministic one. Although the LDBA is then no more suitable for quantitative probabilistic model checking, it still is for qualitative checking. At the same time, it can be much smaller, see Table 2.

Finally, Table 3 illustrates the effect of the native support for additional temporal operators, compared to the original re-writing to $\mathbf{F}, \mathbf{G}, \mathbf{X}, \mathbf{U}$ of [SEJK16,SK16].

ltl2dpa Both modes inherit the improvements of the respective ltl2ldba and ltl2dgra translations. Further, since complementing parity automata is trivial, we run in parallel both the translation of the input formula and of its negation, returning the smaller of the two resulting automata. Finally, we suppress various jumps, for instance when satisfaction is due to the safety part of the formula.

dra2dpa The index appearance record [KMWW17] keeps track of a permutation (ordering) of Rabin pairs. To do so, all ties between pairs have to be resolved. In our implementation, we keep a pre-order instead, where irrelevant ties are not resolved. Consequently, it cannot happen that an irrelevant tie is resolved in two different ways like in [KMWW17], thus effectively merging such states.

Table 3. Effect of native support for $\mathbf{R}$ in ltl2ldba on formulae $r(i) = a_0\mathbf{R}(a_1\mathbf{R}\dots(a_{i-1}\mathbf{R}a_i))$, displaying #states (#Büchi conditions)

	$r(1)$	$r(2)$	$r(3)$	$r(4)$	$r(5)$	$r(6)$
original	4 (1)	10 (2)	26 (3)	68 (4)	189 (5)	534 (6)
improved	3 (1)	6 (1)	9 (1)	13 (1)	18 (1)	24 (1)

4.2 Implementation

There were two main bottlenecks in the older implementations: (i) explicit data structures for the transition system (or only rudimentary support for BDD in the unofficial release Rabinizer 3.1) are not efficient for larger alphabets, and (ii) computation of the acceptance condition did not scale for formulae with many $\mathbf{G}$ operators. We focus on our treatment of these two main problems. Besides, there are further engineering improvements on issues such as caching, data-structure overheads, SCC-based divide-and-conquer constructions, or the introduction of parallelization for batch inputs.

Symbolic data structures Compared to Rabinizer 3, we provide symbolic representation of states and edges.

- States are no more represented by an abstract syntax tree of an LTL formula, but as a BDD for the respective propositional formula, where temporal formulae are considered to be atoms, e.g. $a \wedge \mathbf{FG}(a \vee \mathbf{X}a)$ is considered as a conjunction of two atoms. This required a non-trivial definition of the expansion laws in terms of BDD operations rather than by LTL rewriting rules. This itself causes a speed up of around 10–30%.
- Further, labels of transitions are no more explicit sets of letters (altogether thus sets of sets of atomic propositions), but BDDs over the atomic propositions. This improves efficiency particularly for larger alphabets.

To this end, we provide a Java BDD library featuring the standard functionality together with specific support, for instance substitution, which is used for the expansion laws. We did not use JavaBDD due to its issues with memory management and also overheads due to multi-layered wrapping into objects. Further, its back end, the JDD implementation, is missing some of its declared functionality, most importantly `compose` (substitution) and is not maintained. Therefore, we updated, improved, and completed the JDD implementation into our publicly available JBDD [Meg17] library and linked it to Owl. To our knowledge, it is the only available Java BDD implementation with native support for `compose`.

Acceptance condition Firstly, we design a more efficient representation of the acceptance condition, not as a separate object, but as a labelling of edges. Moreover, this labelling is encoded as a `long` or a `BitSet` attribute (the more appropriate one is chosen) and stored with each edge locally.

Table 4. Effect of computing acceptance sets per SCC on formulae $\psi_1 = x_1 \wedge \phi_1$, $\psi_2 = (x_1 \wedge \phi_1) \vee (\neg x_1 \wedge \phi_2)$, $\psi_3 = (x_1 \wedge x_2 \wedge \phi_1) \vee (\neg x_1 \wedge x_2 \wedge \phi_2) \vee (x_1 \wedge \neg x_2 \wedge \phi_3)$, ..., where $\phi_i = \mathbf{XG}((a_i \mathbf{U} b_i) \vee (c_i \mathbf{U} d_i))$. We display execution time in seconds / #acceptance sets.

Tool	ψ_1	ψ_2	ψ_3	ψ_4	ψ_5	...	ψ_8
Rabinizer 3.1	<2 / 2	<2 / 7	<2 / 19	45 / 47	–		–
ltl2dgra	<2 / 1	<2 / 1	<2 / 1	<2 / 1	<2 / 1		<2 / 1

Table 5. State space differences on the formulae $\phi_1 = a_1$, $\phi(i) = (a_i \mathbf{U}(\mathbf{X}\phi_{i-1}))$, $\psi_i = \mathbf{G}\phi_i$ and $\phi'_1 = a_1$, $\phi'_1 = (\phi'_{i-1} \mathbf{U}(\mathbf{X}^i a_i)$, $\psi'_i = \mathbf{G}\phi'_i$. We display execution time in seconds / #states.

Tool	ψ_2	ψ_3	ψ_4	ψ_5	ψ_6
Rabinizer 3.1	<2 / 4	<2 / 16	<2 / 73	11 / 332	510 / 1463
ltl2dgra	<2 / 4	<2 / 8	<2 / 36	<2 / 200	20 / 1156
	ψ'_2	ψ'_3	ψ'_4	ψ'_5	ψ'_6
Rabinizer 3.1	<2 / 4	<2 / 16	<2 / 104	535 / 670	–
ltl2dgra	<2 / 3	<2 / 10	<2 / 38	<2 / 251	412 / 1420

5 Performance Evaluation

We test our implementation of the original Rabinizer translation, **ltl2dgra**, on various datasets and compare it to the previous version Rabinizer 3.1, which is a partially symbolic and significantly faster [EKS16] re-implementation of the official release Rabinizer 3 [KK14]. All of the benchmarks have been run on a Linux 4.13.2-gentoo x64 virtual machine with 3.0 GHz per core, equipped with the 64 bit Oracle JDK 1.8.0_144 JVM. Due to the use of JVM, all times below 2 seconds are denoted by < 2 and not specified more precisely. Unless otherwise noted, all experiments were given a time-out of 900 seconds, denoted by $-$.

As the basis for our comparison, we used both hand-crafted sets of formulae and established datasets from literature, e.g. the sets in [BKS13,EKS16], altogether two hundred formulae. In summary, our new implementation has slightly less states on average (5.8 compared to 6.6) and significantly less acceptance sets (3.4 compared to 5.5). Below We detail on some of the most striking differences we found during our evaluation.

Table 4 illustrates the effect of computing the acceptance condition separately for each SCC and then combining them. Observe that, e.g., the automaton for $i = 8$ has 31 atomic propositions, but the number of atomic propositions relevant in each SCC of the master remains constant, which allows **ltl2dgra** to perform much better on this family. Further, Table 5 illustrates savings on the size of the state space. Finally, Table 6 shows the reduction of the computation time for the formula used to show 2-EXPTIME hardness of the translation of LTL to deterministic automata.

Table 6. Execution time comparison for "2-EXPTIME-hardness" formulae $\psi_i = \mathbf{G}(\bigvee_{k=1}^{i}(a_k \wedge \mathbf{F}b_k))$.

Tool	ψ_2	ψ_3
Rabinizer 3.1	<2	>3600
ltl2dgra	<2	90

6 Conclusion

We have presented Rabinizer 4, a set of tools to translate LTL to various deterministic automata and to use them in probabilistic model checking and in synthesis. The tool set dramatically extends the previous functionality of Rabinizer and for several algorithms gives the very first implementations. It is built on a common infrastructure Owl, which is steadily improving and committed to further active development. We greatly appreciate comments and suggestions. All future improvements to the infrastructure such as more involved parallelization, tailored processing for various types of input, or further engineering optimizations will automatically be reflected by all constructions. The library has already demonstrated its re-usability in an independent student project [Bar] of re-implementing Seminator [BDK+17] and has shown high efficiency for prototyping, requiring less than 100 LOC. Finally, the tool set is also more user friendly due to richer input syntax, its connection to PRISM and PG Solver, and the on-line version with direct visualization which can be found at `http://www7.in.tum.de/~kretinsk/rabinizer4.html`.

The future development will focus on merging our distribution of PRISM into the official release of PRISM and application of the infrastructure to translation problems arising for various extensions of LTL considering measures of robustness or degree of satisfaction, e.g. [TN16,ABK16].

A Appendix: Command-Line Interface

Owl comes with a flexible command line interface intended to aid rapid development and prototyping of various constructions, which we explain in this section. To give full control over the translation process to the user, it offers a verbose, modular way of specifying a particular tool-chain. This is achieved by means of multiple building blocks, which are connected together to create the desired translation. These "building blocks" come in four different flavours, namely coordinators, input parsers, transformers, and output writers, all of which are completely pluggable and extensible.

Firstly, coordinators are responsible for setting up input and output behaviour. Usually, users will be content with reading from standard input or a file, which is handled by our "stream" coordinator. Other possible coordinators, like a network server, will be mentioned later. The other three blocks are, as their names suggest, responsible for parsing input, applying operations to objects, and serializing the

results to the desired format, respectively. We refer to a sequence of a parser, multiple transformers and an output writer as "pipeline".

A.1 Basic usage

We explain these concepts through simple, incremental examples. To begin with, we chain an LTL parser to the ltl2dpa construction and output the resulting automaton in the HOA format by writing

```
% owl --- stream --- ltl --- ltl2dpa --- hoa
```

Reading fixed input or from a file can be done by specifying `stream -i "<input>"` or `stream -I "<input.file>"`.

To additionally pre-process the input formula and minimize the result automaton, we simply add more transformers to the pipeline

```
% owl --- stream --- ltl --- rewrite --mode modal-iter ---
  ltl2dpa --- minimize-aut --- hoa
```

For research purposes, we are interested in what exactly happens during the intermediate steps, for example how the rewritten formula looks like, or how large the automaton is prior to the minimization. These values could be obtained by running different configurations, but this is cumbersome and time-consuming for large data-sets. Instead, we offer the possibility of seamlessly collecting meta-data during the execution process. For example, to obtain the above numbers in one execution, we write

```
% owl --- stream --- ltl --- rewrite --mode modal-iter ---
  string --- ltl2dpa --- aut-stat --format "%S/%C/%A" ---
  minimize-aut --- hoa
```

Owl will now output the rewritten formula plus the amount of states, number of SCCs and number of acceptance sets together with the corresponding input to stderr (by default).

A.2 Extending the framework

Often, a researcher might not only be interested in how the existing operations performs, but rather how a new implementation behaves. By simply delegating to an external translator, existing implementations can easily be integrated in such a pipeline. For example, to delegate to the old Rabinizer 3.1, we simply write

```
% owl --- stream --- ltl --- rewrite --mode modal-iter ---
  unabbreviate -r -w -m --- ltl2aut-ext --tool
  "java -jar rabinizer3.1.jar -format=hoa -silent -out=std %f" ---
  minimize-aut --level=light --- hoa
```

The real strength of the implementation comes from its flexibility. The command-line parser is completely pluggable and written without explicitly referencing any of our implementations. For example, in order to add a new algorithm one simply has to provide a name (as, e.g., "ltl2nba"), an optional set of command line options and a way of obtaining the configured translator from the parsed options. For example, supposing that this new "ltl2nba" command has some "`--fast`" flag, the whole description necessary is as follows:

```
public class LTL2NBASettings implements TransformerSettings {
  public Factory createTransformerFactory(CommandLine settings) {
    boolean fast = settings.hasOption("fast");
    return environment -> (input, context) ->
      LTL2NBA.apply((LabelledFormula) input, fast, environment);
  }
  public String getKey() { return "ltl2nba"; }
  public Options getOptions() {
    return new Options()
      .addOption("f", "fast", false, "Turn of fast mode");
  }
}
```

After registering these settings with a one-line call, the tool can now be used exactly as **ltl2dpa** before. Additionally, the tool is automatically integrated into the "help" output of owl, without requiring further interaction from the developer. Parsers, serializers or even coordinators can be registered with the same kind of specification.

A.3 Advanced usage

We also support some advanced features, some of which we highlight briefly.

Dedicated tools like our presented **ltl2dgra** or **nba2dpa** can easily be created by delegating to the generic framework. For example, **ltl2ldba** is created by

```
public static void main(String... args) {
  SingleModuleParser.run(args,
    ImmutableSingleModuleConfiguration.builder()
      .inputParser(new LtlInput())
      .addPreProcessors(new RewriterTransformer(
         RewriterEnum.MODAL_ITERATIVE))
      .transformer(new LTL2LDBAModule())
      .outputWriter(new HoaWriter())
      .build());
}
```

Server mode listens on a given address and port for incoming TCP connection. Each of these connections then is handled as a input / output pair, i.e.

the specified input parser reads from each connection and the resulting outputs are written back to the connection, all completely transparent to the translation modules. A "ltl2dpa" server is created by writing

```
% owl --- server --- ltl --- rewrite --mode modal-iter ---
  ltl2dpa --- minimize-aut --- hoa
```

Sending input is as easy as `nc localhost 5050` and starting to type. This allows easy usage as a fast back-end server, since the JVM does not have to start for each input.

Extended coordinators can also be written by users. It is entirely possible to, e.g., implement a "json" coordinator, which reads input from a file and outputs the results together with all meta-information in form of a JSON-file.

References

[ABK16] Shaull Almagor, Udi Boker, and Orna Kupferman. Formally reasoning about quality. *J. ACM*, 63(3):24:1–24:56, 2016.

[AT04] Rajeev Alur and Salvatore La Torre. Deterministic generators and games for LTL fragments. *ACM Trans. Comput. Log.*, 5(1):1–25, 2004.

[Bar] Florian Barta. Translation of nondeterministic büchi automata to deterministic parity automata via limit-deterministic büchi automata: An implementation and evaluation. Master's thesis, Technical University of Munich. To be submitted.

[BBD+15] Tomáš Babiak, František Blahoudek, Alexandre Duret-Lutz, Joachim Klein, Jan Křetínský, David Müller, David Parker, and Jan Strejček. The hanoi omega-automata format. In *CAV, Part I*, pages 479–486, 2015.

[BBDL+13] Tomáš Babiak, Thomas Badie, Alexandre Duret-Lutz, Mojmír Křetínský, and Jan Strejček. Compositional approach to suspension and other improvements to LTL translation. In *SPIN*, pages 81–98, 2013.

[BDK+17] Frantisek Blahoudek, Alexandre Duret-Lutz, Mikulás Klokocka, Mojmír Kretínský, and Jan Strejcek. Seminator: A tool for semi-determinization of omega-automata. In *LPAR*, pages 356–367, 2017.

[BDL12] Benedikt Bollig, Normann Decker, and Martin Leucker. Frequency linear-time temporal logic. In *TASE*, pages 85–92, 2012.

[BJP+12] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[BKŘS12] Tomáš Babiak, Mojmír Křetínský, Vojtěch Řehák, and Jan Strejček. LTL to Büchi automata translation: Fast and more deterministic. In *TACAS*, pages 95–109, 2012.

[BKS13] František Blahoudek, Mojmír Křetínský, and Jan Strejček. Comparison of LTL to deterministic Rabin automata translators. In *LPAR*, pages 164–172, 2013.

[BMM14] Patricia Bouyer, Nicolas Markey, and Raj Mohan Matteplackel. Averaging in LTL. In *CONCUR*, pages 266–280, 2014.

[CGK13] Krishnendu Chatterjee, Andreas Gaiser, and Jan Křetínský. Automata with generalized Rabin pairs for probabilistic model checking and LTL synthesis. In *CAV*, pages 559–575, 2013.

[CKK15] Krishnendu Chatterjee, Zuzana Komárková, and Jan Křetínský. Unifying two views on multiple mean-payoff objectives in markov decision processes. In *LICS*, pages 244–256, 2015.

[Cou99] Jean-Michel Couvreur. On-the-fly verification of linear temporal logic. In *FM*, pages 253–271, 1999.

[CY88] Costas Courcoubetis and Mihalis Yannakakis. Verifying temporal properties of finite-state probabilistic programs. In *FOCS*, pages 338–345, 1988.

[DGV99] Marco Daniele, Fausto Giunchiglia, and Moshe Y. Vardi. Improved automata generation for linear temporal logic. In *CAV*, pages 249–260, 1999.

[DLLF+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 *ATVA*, pages 122–129, October 2016.

[EH00] Kousha Etessami and Gerard J. Holzmann. Optimizing Büchi automata. In *CONCUR*, pages 153–167, 2000.

[EK14] Javier Esparza and Jan Křetínský. From LTL to deterministic automata: A Safraless compositional approach. In *CAV*, pages 192–208, 2014.

[EKRS17] Javier Esparza, Jan Křetínský, Jean-Francois Raskin, and Salomon Sickert. From ltl and limit-deterministic Büchi automata to deterministic parity automata. In *TACAS*, 2017. to appear.

[EKS16] Javier Esparza, Jan Kretínský, and Salomon Sickert. From LTL to deterministic automata - A safraless compositional approach. *Formal Methods in System Design*, 49(3):219–271, 2016.

[FKK15] Vojtěch Forejt, Jan Krčál, and Jan Křetínský. Controller synthesis for MDPs and frequency LTL\GU. In *LPAR*, pages 162–177, 2015.

[FL09] Oliver Friedmann and Martin Lange. Solving parity games in practice. In *ATVA*, pages 182–196, 2009.

[Fri03] Carsten Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In *CIAA*, pages 35–48, 2003.

[GL02] Dimitra Giannakopoulou and Flavio Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *FORTE*, pages 308–326, 2002.

[GO01] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In *CAV*, pages 53–65, 2001. Tool accessible at `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/`.

[HLS+15] Ernst Moritz Hahn, Guangyuan Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. Lazy probabilistic model checking without determinisation. In *CONCUR*, volume 42 of *LIPIcs*, pages 354–367, 2015.

[HP06] Thomas A. Henzinger and Nir Piterman. Solving games without determinization. In *CSL*, volume 4207 of *LNCS*, pages 395–410. Springer, 2006.

[JKS16] Swen Jacobs, Felix Klein, and Sebastian Schirmer. A high-level LTL synthesis format: TLSF v1.1. In *Fifth Workshop on Synthesis (SYNT@CAV)*, pages 112–132, 2016.

[KE12] Jan Křetínský and Javier Esparza. Deterministic automata for the (F,G)-fragment of LTL. In *CAV*, volume 7358 of *LNCS*, pages 7–22, 2012.

[KK14] Zuzana Komárková and Jan Křetínský. Rabinizer 3: Safraless translation of LTL to small deterministic automata. In *ATVA*, volume 8837 of *LNCS*, pages 235–241, 2014.

[Kle] Joachim Klein. ltl2dstar - LTL to deterministic Streett and Rabin automata. `http://www.ltl2dstar.de/`.

[KLG13] Jan Křetínský and Ruslán Ledesma-Garza. Rabinizer 2: Small deterministic automata for LTL\GU. In *ATVA*, pages 446–450, 2013.

[KMWW17] Jan Křetínský, Tobias Meggendorfer, Clara Waldmann, and Maximilian Weininger. Index appearance record for transforming rabin automata into parity automata. In *TACAS*, 2017. to appear.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, pages 585–591, 2011.

[KPV06] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. Safraless compositional synthesis. In *CAV*, volume 4144 of *LNCS*, pages 31–44. Springer, 2006.

[KV15] Dileep Kini and Mahesh Viswanathan. Limit deterministic and probabilistic automata for LTL \ GU. In *TACAS*, pages 628–642, 2015.

[KV17] Dileep Kini and Mahesh Viswanathan. Optimal translation of ltl to limit deterministic automata. In *TACAS 2017*, 2017. To appear.

[Meg17] Tobias Meggendorfer. JBDD: A java BDD library. `https://github.com/incaseoftrouble/jbdd`, 2017.

[ML16] Philipp J. Meyer and Michael Luttenberger. Solving mean-payoff games on the GPU. In *ATVA*, pages 262–267, 2016.

[Pit06] Nir Piterman. From nondeterministic Büchi and Streett automata to deterministic parity automata. In *LICS*, pages 255–264, 2006.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[PP06] Nir Piterman and Amir Pnueli. Faster solutions of Rabin and Streett games. In *LICS*, pages 275–284, 2006.

[Saf88] Shmuel Safra. On the complexity of omega-automata. In *FOCS*, pages 319–327, 1988.

[SB00] Fabio Somenzi and Roderick Bloem. Efficient Büchi automata from LTL formulae. In *CAV*, pages 248–263, 2000.

[Sch09] Sven Schewe. Tighter bounds for the determinisation of Büchi automata. In *FoSSaCS*, volume 5504 of *LNCS*, pages 167–181, 2009.

[SEJK16] Salomon Sickert, Javier Esparza, Stefan Jaax, and Jan Kretínský. Limit-deterministic büchi automata for linear temporal logic. In *CAV*, pages 312–332, 2016.

[SK16] Salomon Sickert and Jan Kretínský. Mochiba: Probabilistic LTL model checking using limit-deterministic büchi automata. In *ATVA*, pages 130–137, 2016.

[TN16] Paulo Tabuada and Daniel Neider. Robust linear temporal logic. In *CSL*, pages 10:1–10:21, 2016.

[Var85] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *FOCS*, pages 327–338, 1985.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.