Owl: A Library for ω-Words, Automata, and LTL

Jan Křetínský, Tobias Meggendorfer, and Salomon Sickert

Technical University of Munich

Abstract. We present the library `Owl` (**O**mega-**W**ords, automata, and **L**TL) for ω-automata and linear temporal logic. It forms a backbone of several translations from LTL to automata and related tools by different authors. We describe the functionality of the library and the recent experience, which has already shown the library is apt for easy prototyping of new tools in this area.

1 An Owl is Born: Introduction

ω-automata are finite automata over infinite words. As opposed to finite automata over finite words, there is not a single acceptance condition, but a wide variety of possibilities, each being more appropriate for certain applications. To give a few examples, non-deterministic Büchi automata are the most used kind, useful in many contexts, including the modelling and analysis of reactive systems, where both the system and the property of interest, say in linear temporal logic (LTL) [31], are transformed into these automata. In contrast, the classical approach for synthesis of reactive systems prefers deterministic parity automata. Further, while the textbook approach to probabilistic LTL model checking suggest to translate LTL formulae to deterministic Rabin automata, recent approaches show that deterministic generalized Rabin automata or limit-deterministic automata are more preferable. Consequently, a zoo of automata arises, both due to theoretical limitations of certain kinds as well as practical efficiency. While the theoretical complexity of the transformations between the automata and of translations from LTL to automata is long settled, the research on practically more efficient approaches is flourishing, both for non-deterministic [6, 7, 14, 36, 15, 16, 3, 8] and more recently deterministic [22, 2, 10, 18, 12, 34, 11, 19, 13] automata. Notably, while these constructions are based on diverse ideas, their implementation requires almost the same infrastructure.

Tools in this area have very different purposes, ranging from tools for one specific task, e.g. translating LTL into a particular type of automaton, e.g. [15, 3, 20, 19, 21], to educational GUI tools demonstrating the constructions, e.g. `JFLAP` [32], to tools implementing a comprehensive collection of algorithms from literature, e.g. `GOAL` [38], to tool sets with a highly configurable CLI procedures focusing on efficiency, e.g. `Spot` [8]. We contribute to this spectrum with the library `Owl`, which enables easy and fast development of transformation/translation tools, yet yielding efficient implementations.

Owl is a full-fledged library for manipulating ω-automata and LTL both on low and high level. One of the main characteristics is that it links the functionality

for automata and logic in a very tight and explicit way, providing additional support for "semantic" translations of LTL to automata. These are translations where states are described using structures over logical formulae, as we know it from the classical, e.g. the tableaux-based, tradition. This tradition was disrupted for deterministic automata due to Safra's construction [33], where the meaning of a state (the language it recognizes) cannot be easily described in terms of the meaning of the corresponding formulae. The "semantic" tradition has been restored recently in the works on deterministic automata cited above.

Apart from this characteristics, our library has several other user-friendly traits and distinguishing features. For instance, it supports parallelism, it is built according to the on-the-fly philosophy, it is written in Java (with no memory management issues left for the user, being more accessible to students), extensive pipe-style CLI support for quick and easy prototyping, and an easy-to-configure testing framework checking correctness of translations written with the library.

In this tool paper, we briefly describe the functionality of the library and then provide a series of actual use cases (not only by the authors), demonstrating the usability and particular advantages of this library.

2 The Anatomy of the Owl: Functionality

`Owl` (**O**mega-**W**ords, automata, and **L**TL) arose from the needs when implementing `Rabinizer` 3.1 [21, 12] and `ltl2ldba` [34]. When developing such translations a lot of infrastructure is necessary, e.g., LTL parsing and representation, while the actual construction is only a small fraction of the written code. Thus, we implemented commonly needed functionality in a reusable Java library for LTL and ω-automata and extended it with numerous features to provide a flexible infrastructure for rapid and seamless development of algorithms in these domains.

2.1 Data Structures and Algorithms

The majority of data structures and algorithms concerns LTL and automata.

LTL. The library provides an LTL parser, a simplifier with state-of-the-art rewrite rules, classification into syntactic fragments and transformation into normal forms. Additionally, a parser for the synthesis specification format TLSF [17] is available and includes a conversion to LTL.

Further, the LTL support comes with efficient rewriting according to the LTL expansion laws, e.g. [4]. This enables the decomposition of temporal formulas into directly checkable assertions on the current position and on the immediate temporal successor, e.g. $a\mathbf{U}b \equiv b \vee (a \wedge \mathbf{X}(a\mathbf{U}b))$. As such, they are a core component of both classic, e.g. tableaux-based, as well as recent semantic translations.

Automata. The library provides support for deterministic and non-deterministic ω-automata with both classic acceptance conditions, e.g., Büchi, coBüchi, Rabin and parity, as well as, e.g., like generalized Rabin [25] or Emerson-Lei acceptance [9]. Internally, acceptance is represented as transition-based acceptance and a conversion to and from state-based acceptance for interfacing with external tools is present.

Automata can either be stored and modified explicitly, meaning the whole state-space and transitions are kept in memory, or defined implicitly by specifying initial states and a method for successor computation. The latter approach has two main advantages: First, new constructions can be implemented with little effort, transferring the definition of the successor relation into code. For example, see Appendix B for a two-page Java implementation of Safra's determinization procedure. Second, automata can be conveniently traversed on the fly without storing the transition system, allowing operations on huge or potentially even infinite transition structures.

For automata, classic algorithms such as decomposition into strongly connected components (SCC) and lasso-based emptiness checks are included. Furthermore, constructions such as union, intersection and degeneralization are present. In addition, modifications of the transition structure and the acceptance conditions are supported, e.g., removal of non-accepting or unreachable parts of the state space, completing the transition relation, and simplifications of the acceptance condition. Acceptance sets are stored as edge labels for efficient rewriting, supporting arbitrarily sized acceptances, compared to, e.g., Spot [8], which supports at most 32 sets due to its focus on (generalized) Büchi conditions.

2.2 Interfacing

There are two ways to interact with `Owl`: On the one hand, there is a command-line interface with text-based formats, e.g., (Spot-style) LTL, TLSF [17], and the Hanoi ω-automaton format (HOA) [1]. This approach is completely agnostic of the implementation, but the whole output is always constructed, which is prohibitively expensive for huge outputs where only a small fraction might be needed. On the other hand, there is a Java and a (specialized) C++ API offered by `Owl`, which allows fine-grained access and exposes the on-the-fly nature of most of the translations to external code.

Command-line Interface. Major functionality of the library is available via a pipe-style CLI, which makes it easy to specify the sequence of procedures (input parsing, translations, conversions, statistics and serialization) to be performed. For example, `owl ltl --- simplify-ltl --- ltl2dpa --- hoa` reads LTL formulas from `stdin` line-by-line, simplifies them using the default simplifier, translates them to DPAs and writes them to `stdout` in the HOA format. This can be extended to advanced pipelines, e.g., `owl -I "in.ltl" --- ltl --- ltl2dgra --- aut-stat "DGRA:%s" --- dgra2dra --- aut-stat "DRA:%s" --- null`. This pipeline reads LTL formulas from the file `in.ltl`, translates them to DGRAs and DRAs, while outputting the respective sizes of the automata, and finally discards the actual output, saving the time needed for serialization.

Moreover, we support several sources and sinks for data. While one can simply process data from files and the command line, we also added a server mode to reduce the JVM start-up cost, where I/O is bound to a socket. Further details on the CLI together with an in-depth example can be found in Appendix A.

Java and C++ API. Java and Java-like (e.g., Scala) applications can import most of `Owl` and have fine-grained control. For C++ tools, there exists a specialized interface to access core functionality of the library. Among other things, this

enables C++ code to iteratively explore automata state by state instead of forcing a complete construction. This iterative exploration is a core component of the state-of-the-art synthesis tool `Strix` [29] and is crucial for its performance.

2.3 Development Infrastructure and Scalable Architecture

Testing. Small changes to a translation can easily introduce bugs. Thus a test suite is included, which provides several input sets and cross-checks each translation, developed with `Owl`, on hundreds of formulae [26] using `ltlcross` [8]. Apart from detecting bugs, the test suite offers further conveniences, e.g., it automatically generates an image of an erroneous automaton together with an erroneous run. Moreover, various statistics of the generated automata are displayed, usable for performance testing. Lastly, integration of a newly developed translation can be achieved by a few lines of JSON, see Appendix B for an example.

BDDs. Both the LTL part and the automata part of the library use binary decision diagrams (BDD) for some aspects of their functionality, e.g., for a compact representation edge sets and (propositional) equivalence checks of formulas. We implemented our own pure Java BDD library JBDD [28], to (i) achieve portability, not requiring users to compile, e.g., CUDD, and (ii) provide an efficient and tuned implementation for all used BDD operations, e.g. substitution of variables, called `compose`. Particularly, `compose` is fundamental for a symbolic implementation of the semantic constructions and greatly improves their runtime compared to the explicit variants. Since `compose`-related operations often consume well over half of the runtime, `Owl` offers several fine-tuned variants, further improving performance.

3 The Owl in the Wild: Use Cases

`Owl` has been successfully used for several published tools and student projects, demonstrating versatility and usability even for less experienced users. To name a few, the following published tools (in alphabetical order) using `Owl` are available:

`Delag` [30] translates LTL into deterministic Emerson-Lei automata. Reusing other translations based on `Owl`, see `Rabinizer` [23], it adds specialized constructions for fragments of LTL, exploiting a succinct encoding coupled to the Emerson-Lei acceptance condition.

`MoChiBa` [35] is an extension of `PRISM` [27] and uses limit-deterministic automata for quantitative model checking of Markov decision processes [34]. Due to a tight integration with `Owl`, additional information on the automata can be accessed, optimizing the construction.

`Rabinizer` [23] is a collection of tools translating LTL to various types of deterministic automata. It uses a fully BDD-based successor computation of `Owl`, improving performance over the previous versions. The current distribution of `Owl` includes the latest version of `Rabinizer` (4.0).

`Strix` [29] constructs controllers (either Mealy machines or AIGER circuits) from LTL specifications via parity games. Constructing the underlying automata and solving the parity games take an incremental approach and make use of the on-the-fly implementation of the translations.

The list of student projects includes[1]

- a re-implementation of `Seminator` [5],
- a specialized translation of the $(\mathbf{F}, \mathbf{G}, \mathbf{X})$-fragment of LTL to deterministic parity automata, and
- reactive synthesis exploiting the `Owl`-supported semantic labelling of the automata produced by `Rabinizer` through learning approaches.

Furthermore, rLTL (robust LTL) [37] can be easily transformed into LTL using `Owl`[2]. Finally, to illustrate the ease with which new translations can be written, we implemented the notoriously complicated and hard-to-implement [24] Safra's determinization procedure [33], with the complete code listed in Appendix B. A detailed analysis of the lines of code needed to implement the mentioned translations and the percentage of library that is used can be found in Appendix C.

4 This is not the End: Conclusion

We have presented the library `Owl`, which provides infrastructure for easy development of efficient prototypes in the area of LTL and automata. It has already demonstrated its re-usability in several projects, also without the presence of the library authors. For instance, our experience with Master students has demonstrated that a tool for a complex translation, such as [5], can be easily implemented using roughly 400 lines of code, achieving performance comparable to the original dedicated tool. One simply defines the mathematical type of the state space, the initial state, the successor function with the acceptance marking, whereas the rest is taken care of by the library. The library can be found at `https://owl.model.in.tum.de`, including code, documentation, references and an online demo. We greatly appreciate comments and suggestions.

References

1. T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Křetínský, D. Müller, D. Parker, and J. Strejček. The Hanoi omega-automata format. In *CAV, Part I*, 2015.
2. T. Babiak, F. Blahoudek, M. Křetínský, and J. Strejček. Effective translation of LTL to deterministic Rabin automata: Beyond the (F, G)-fragment. In *ATVA*, 2013.
3. T. Babiak, M. Křetínský, V. Řehák, and J. Strejček. LTL to Büchi automata translation: Fast and more deterministic. In *TACAS*, 2012.
4. C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.
5. F. Blahoudek, A. Duret-Lutz, M. Klokočka, M. Křetínský, and J. Strejček. Seminator: A tool for semi-determinization of omega-automata. In *LPAR*, 2017.
6. J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *FM*, 1999.
7. M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear temporal logic. In *CAV*, 1999.
8. A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *ATVA*, 2016.
9. E. A. Emerson and C. Lei. Modalities for model checking: Branching time strikes back. In *POPL*, 1985.

[1] Authored by Florian Barta, Matthias Franze, and Sebastian Fiss, respectively.
[2] Originally implemented by Daniel Neider.

10. J. Esparza and J. Křetínský. From LTL to deterministic automata: A Safraless compositional approach. In *CAV*, 2014.
11. J. Esparza, J. Křetínský, J.-F. Raskin, and S. Sickert. From LTL and limit-deterministic Büchi automata to deterministic parity automata. In *TACAS*, 2017.
12. J. Esparza, J. Křetínský, and S. Sickert. From LTL to deterministic automata - A safraless compositional approach. *Formal Methods in System Design*, 2016.
13. J. Esparza, J. Kretínský, and S. Sickert. One theorem to rule them all: A unified translation of LTL into ω-automata. 2018. Preprint at `arxiv.org/abs/1805.00748`.
14. K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In *CONCUR*, 2000.
15. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV*, 2001. Tool accessible at `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/`.
16. D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *FORTE*, 2002.
17. S. Jacobs, F. Klein, and S. Schirmer. A high-level LTL synthesis format: TLSF v1.1. In *Fifth Workshop on Synthesis (SYNT@CAV)*, 2016.
18. D. Kini and M. Viswanathan. Limit deterministic and probabilistic automata for LTL \ GU. In *TACAS*, 2015.
19. D. Kini and M. Viswanathan. Optimal translation of LTL to limit deterministic automata. In *TACAS*, 2017.
20. J. Klein. ltl2dstar - LTL to deterministic Streett and Rabin automata. `http://www.ltl2dstar.de/`.
21. Z. Komárková and J. Křetínský. Rabinizer 3: Safraless translation of LTL to small deterministic automata. In *ATVA*, vol. 8837 of *LNCS*, 2014.
22. J. Křetínský and J. Esparza. Deterministic automata for the (F,G)-fragment of LTL. In *CAV*, vol. 7358 of *LNCS*, 2012.
23. J. Křetínský, T. Meggendorfer, S. Sickert, and C. Ziegler. Rabinizer 4: From ltl to your favourite deterministic automaton. In *CAV*, 2018. To appear.
24. O. Kupferman. Recent challenges and ideas in temporal synthesis. In *SOFSEM*, vol. 7147 of *LNCS*. Springer, 2012.
25. J. Křetínský and J. Esparza. Deterministic automata for the (F,G)-fragment of LTL. In *CAV*, 2012.
26. J. Křetínský, T. Meggendorfer, and S. Sickert. LTL Store: Repository of LTL formulae from literature and case studies. *CoRR*, abs/1805.xxxx, 2018.
27. M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, 2011.
28. T. Meggendorfer. JBDD: A java BDD library. `github.com/incaseoftrouble/jbdd`.
29. P. Meyer, S. Sickert, and M. Luttenberger. Strix: Explicit reactive synthesis strikes back! In *CAV*, 2018. To appear.
30. D. Müller and S. Sickert. LTL to deterministic Emerson-Lei automata. In *GandALF*, 2017.
31. A. Pnueli. The temporal logic of programs. In *FOCS*, 1977.
32. S. H. Rodger, H. Qin, and J. Su. Changes to JFLAP to increase its use in courses. In *SIGCSE*, 2011.
33. S. Safra. On the complexity of omega-automata. In *FOCS*, 1988.
34. S. Sickert, J. Esparza, S. Jaax, and J. Křetínský. Limit-deterministic büchi automata for linear temporal logic. In *CAV*, 2016.
35. S. Sickert and J. Křetínský. Mochiba: Probabilistic LTL model checking using limit-deterministic Büchi automata. In *ATVA*, 2016.
36. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV*, 2000.
37. P. Tabuada and D. Neider. Robust linear temporal logic. In *CSL*, 2016.
38. M.-H. Tsai, Y.-K. Tsay, and Y.-S. Hwang. GOAL for games, omega-automata, and logics. In *CAV*, 2013.

A 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 three different flavours, namely input parsers, transformers, and output writers, all of which are pluggable and extensible.

These 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".

Once configured, a pipeline is passed to an executor, which sets up the input/output behaviour and actually executing the pipeline. Usually, users will be content with reading from standard input or a file, which is handled the default executor. Other possibilities, like a network server, will be mentioned later.

A.1 Basic usage

We explain this approach through a simple, incremental example. To begin with, we chain an LTL parser to the `ltl2dpa` construction and output the resulting automaton in the HOA format by

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

Fixed input can be specified with `-i "<input>"`, while `-I "<input.file>"` reads the given file. Similarly, output is written to a file with `-O "<output.file>"`

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

```
% owl  ltl --- simplify-ltl --- ltl2dpa --- minimize-aut --- hoa
```

For research purposes, we may be 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. We could obtain this data by executing several different configurations, which 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  ltl --- simplify-ltl --- 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 for each 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 Rabinizer 3.1, we simply write

```
% owl  ltl --- simplify-ltl --- ltl2aut-ext
    --tool "run-rabinizer.sh %f" --- minimize-aut --- hoa
```

The real strength of this framework 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 our new `ltl2nba` command has some `--fast` flag, the whole description necessary is as follows:

```
public static final TransformerParser CLI_SETTINGS =
  ImmutableTransformerParser.builder()
    .key("ltl2nba")
    .description("Translates LTL to NBA really fast")
    .optionsDirect(new Options()
      .addOption("f", "fast", false, "Turn on fast mode"))
    .parser(settings -> {
      boolean fast = settings.hasOption("fast");
      return env -> (input, context) ->
        LTL2NBA.apply((LabelledFormula) input, fast, env))
    .build();
```

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 and serializers 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 can easily be created by delegating to the generic framework. For example, `ltl2ldba` is created by

```
public static void main(String... args) {
  PartialConfigurationParser.run(args,
    PartialModuleConfiguration.builder("ltl2ldba")
      .reader(InputReaders.LTL)
      .addTransformer(Transformers.LTL_SIMPLIFIER)
      .addTransformer(LTL2LDBACliParser.INSTANCE)
      .writer(OutputWriters.HOA)
      .build());
}
```

This automatically sets up command line argument processing, input / output parsing, help printing, etc.

Server mode listens on a given address and port for incoming TCP connections. Each of these connections then is handled as a separate pair of input source / output sink, i.e. the specified input parser reads from each connection and the resulting outputs are written back to the client, all completely transparent to the translation modules. For example, a `ltl2dpa` server is created by writing

```
% owl-server  ltl --- simplify-ltl --- ltl2dpa --- hoa
```

Sending input is as easy as `nc localhost 5050` and starting to type. We also provide a small C utility `owl-client` dedicated to this purpose for users without access to `netcat`. This allows easy usage as a fast back-end server, since the JVM does not have to start for each input.

B Implementing Safra's construction

To demonstrate the versatility of our library, we implemented Safra's determinization procedure from NBA to DPA. Although this procedure often is described as being tedious to implement, it required only roughly 60 lines of code in `Owl` (plus a few lines for simple data structures). In the following, we present the full implementation, pruned of assertions and logging statements for brevity.

B.1 Construction Code

We first show the complete code block used to implement Safra's construction. In the following section, we present the used utility classes `Label` and `Tree`.

```
public static <S> Automaton<Tree<Label<S>>, RabinAcceptance>
  build(Automaton<S, BuchiAcceptance> nba) {
  int nbaSize = nba.size();
  int pairCount = nbaSize * 2;
  RabinAcceptance acceptance = RabinAcceptance.of(pairCount);
  Tree<Label<S>> initialState = Tree.of(Label.of(Set.copyOf(nba.initialStates()), 0));

  BiFunction<Tree<Label<S>>, BitSet, Edge<Tree<Label<S>>>> successor = (tree, valuation) -> {
    BitSet usedIndices = new BitSet(nbaSize);
    tree.forEach(node -> usedIndices.set(node.index()));
    BitSet edgeAcceptance = new BitSet(nbaSize);

    Tree<Label<S>> successorTree = tree.map((father, children) -> { // Successor
      Set<Edge<S>> fatherEdges = father.states().stream().flatMap(state ->
        nba.edges(state, valuation).stream()).collect(Collectors.toSet());

      if (fatherEdges.isEmpty()) return Tree.of(father.with(Set.of()));

      Label<S> newFather = father.with(Edges.successors(fatherEdges));
      Set<S> newChildStates = fatherEdges.stream().filter(edge -> edge.inSet(0))
        .map(Edge::successor).collect(Collectors.toUnmodifiableSet());

      int index = usedIndices.nextClearBit(0);
      usedIndices.set(index);
      return Tree.of(newFather, Collections3.concat(children,
        List.of(Tree.of(Label.of(newChildStates, index)))));
    }).map((father, children) -> { // Horizontal merge
      Set<S> olderStates = new HashSet<>();
      List<Tree<Label<S>>> prunedChildren = new ArrayList<>();

      for (Tree<Label<S>> child : children) {
        Label<S> prunedLabel = child.label().without(olderStates);

        if (prunedLabel.states().isEmpty()) {
          edgeAcceptance.set(acceptance.pairs().get(prunedLabel.index()).finSet());
          usedIndices.clear(prunedLabel.index());
        } else {
          // Recursive pruning of the child
          prunedChildren.add(child.map((subNode, subChildren) ->
            Tree.of(subNode.without(olderStates), subChildren)));
          olderStates.addAll(prunedLabel.states());
        }
      }
      return Tree.of(father, prunedChildren);
    }).map((father, children) -> {
      List<Tree<Label<S>>> nonEmptyChildren = children.stream().filter(
        child -> !child.label().states().isEmpty()).collect(Collectors.toList());
      if (nonEmptyChildren.isEmpty()) return Tree.of(father);

      Set<S> childStates = nonEmptyChildren.stream().map(Tree::label).map(Label::states)
        .flatMap(Set::stream).collect(Collectors.toUnmodifiableSet());
```

```
      // Vertical merge
      if (childStates.equals(father.states())) {
        edgeAcceptance.set(acceptance.pairs().get(father.index()).infSet());
        children.forEach(child -> child.forEach(node -> usedIndices.clear(node.index())));
        return Tree.of(father);
      }
      return Tree.of(father, nonEmptyChildren);
    });

    usedIndices.flip(0, nbaSize);
    BitSets.forEach(usedIndices, index ->
      edgeAcceptance.set(acceptance.pairs().get(index).finSet()));
    return Edge.of(successorTree, edgeAcceptance);
  };

  // Create on-the-fly automaton from initial state and successor function
  return AutomatonFactory.create(initialState, nba.factory(), successor, acceptance);
}
```

B.2 Data Structure Classes

In the above code, we used the following two data structure classes. The implementations are generated by the Immutables framework.

```
@Tuple @Value.Immutable
public abstract static class Label<S> {
  abstract Set<S> states();
  abstract int index();

  static <S> Label<S> of(Collection<S> states, int index) {
    return LabelTuple.create(states, index);
  }

  Label<S> without(Set<S> states) { return of(Sets.difference(states(), states), index()); }
  Label<S> with(Collection<S> states) { return of(states, index()); }
}
```

```
@HashedTuple @Value.Immutable
public abstract class Tree<L> {
  abstract L label();
  abstract List<Tree<L>> children();

  static <L> Tree<L> of(L label) { return TreeTuple.create(label, List.of()); }
  static <L> Tree<L> of(L label, List<Tree<L>> children) {
    return TreeTuple.create(label, children);
  }

  public Tree<L> with(L label) { return of(label, children()); }

  public Tree<L> map(BiFunction<L, List<Tree<L>>, Tree<L>> function) {
    return function.apply(label(), Lists.transform(children(), child -> child.map(function)));
  }
  public Tree<L> map(Function<L, L> function) {
    return of(function.apply(label()), Lists.transform(children(),
      child -> child.map(function)));
  }

  public void forEach(Consumer<L> action) {
    children().forEach(child -> child.forEach(action));
    action.accept(this.label());
  }
  public void forEach(BiConsumer<L, List<L>> action) {
    action.accept(label(), Lists.transform(children(), Tree::label));
    children().forEach(child -> child.forEach(action));
  }
}
```

B.3 Integration

To integrate the new construction with our pipeline, we only needed to register the following field.

```
public static final TransformerParser CLI = ImmutableTransformerParser.builder()
  .key("safra")
  .description("Translates NBA to DRA using Safra's construction")
  .parser(settings -> environment -> (input, context) ->
    SafraBuilder.build(AutomatonUtil.cast(input, BuchiAcceptance.class)))
  .build();
```

Adding this construction to the testing framework then only required the following two changes: First, we declare our new construction in `tools.json` by

```
"ltl-safra": {
  "type": "owl",
  "input": "ltl",
  "output": "hoa",
  "name": "safra",
  "pre": [
    "simplify-ltl",
    [ "ltl2aut-ext", "-t", "ltl2tgba -B" ]
  ],
  "post": [ "minimize-aut" ]
}
```

Now, `ltl-safra` refers to a pipeline which reads and simplifies LTL formulae, passes them to Spot's `ltl2tgba`, producing a Büchi automaton, and then applies the above construction to it. Then, we declare a test case in `tests.json` by

```
"safra": {
  "tools": "ltl-safra",
  "data": "small"
}
```

This test now takes our above declared pipeline `ltl-safra` and runs it on the `small` data set. It can be manually executed by running `python scripts/util.py test safra`, or integrated in the CI pipeline by adding

```
Safra:
  stage: test
  variables:
    TEST_NAME: "safra"
  <<: *ltlcross_template
```

to `.gitlab-ci.yml`, i.e. GitLab's CI specifications.

C Detailed Code Metrics

In this section, we present two code metrics, showing the reusability and versatility of `Owl`. In particular, these metrics indicate the total lines of code that could be saved by re-using `Owl` for a new algorithm.

C.1 Raw Size

Table 1. Total lines of code (LoC) (including imports, comments, etc.) in `Owl` (left) and some translations together with their relative size compared to `Owl` (right).

Package	LoC
`automaton`	8321
`ltl`	6922
`factories`	1425
`run`	2570
JBDD	4913
Other utility	2246
Total	27251

Tool	LoC	Percentage
`Delag`	1357	5%
`Rabinizer`	7965	29%
`dra2dpa`	672	2%
`ldba2dpa`	434	2%
`ltl2ldba`	3646	17%
`ltl2dpa`	406	1%
`ltl2dgra`	2595	10%
Seminator reimpl.	390	1%

Table 1 presents the total lines of code (including import statements, comments, etc.) currently in `Owl` together with the respective share of each major package, namely `ltl` (LTL syntax, simplifications, etc.), `automaton` (Automaton representation, acceptance, etc.), `factories` (symbolic data structure abstraction), and `run` (CLI and I/O infrastructure). Further, the table also includes the total size of some translations mentioned in the main body together with their relative size compared to `Owl`. We did not include `MoChiBa` (model checking based on `ltl2ldba`, integrated in PRISM) and `Strix` (LTL synthesis tool, implemented in C++), since their architecture complicates a comprehensive comparison.

Many constructions only need a few hundred lines of code for their implementation, demonstrating the significant aid provided by `Owl`. See the following section for another measurement of code re-use by the translations.

C.2 Coverage Data

One might argue that most of the implemented functionality may be superfluous and actually is not needed for the constructions. To this end, Table 2 shows the code coverage of `Owl` achieved by a typical run of each construction in its default configuration. Coverage refers to the relative amount of actually executable lines being "covered", i.e. executed, by a particular invocation. This excludes comments, documentation, imports, etc. from consideration. We used the IntelliJ coverage tool to determine these values.

We highlight that coverage is an under-approximation of actually re-used code, since some code paths may only occur in a particular scenario or with different options. Nevertheless, even in their default configuration most tools reuse

Table 2. Coverage data of several constructions on a typical run, measured with IntelliJ's coverage tool. Each column denotes the percentages of `Owl`'s classes, methods, and lines of code covered by the execution.

Tool	Class	Method	Line
`Delag`	37%	26%	23%
`Rabinizer`			
`dra2dpa`	26%	14%	14%
`ltl2dgra`	49%	34%	33%
`ltl2dpa`	51%	34%	31%
`ltl2ldba`	48%	31%	29%
`ltl2dra`	53%	37%	36%
Seminator reimpl.	27%	14%	15%

a significant percentage of `Owl` while only using a few hundred lines themselves. Note that the automaton-to-automaton translations `dra2dpa` and the Seminator reimplementation don't make use of the `ltl` package, thus having significantly less re-use. Unfortunately, due to restrictions of the coverage tool, JBDD had to be excluded.