Spot 2.0 — a framework for LTL and \(\omega\)-automata manipulation

Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Étienne Renault, and Laurent Xu

LRDE, EPITA, Kremlin-Bicêtre, France
spot@lrde.epita.fr

Abstract. We present Spot 2.0, a C++ library with Python bindings and an assortment of command-line tools designed to manipulate LTL and \(\omega\)-automata in batch. New automata-manipulation tools were introduced in Spot 2.0; they support arbitrary acceptance conditions, as expressible in the Hanoi Omega Automaton format. Besides being useful to researchers who have automata to process, its Python bindings can also be used in interactive environments to teach \(\omega\)-automata and model checking.

1 Introduction

Spot is a C++ library of model-checking algorithms that was first presented in 2004 [15]. It contains algorithms to perform the usual tasks in the automata-theoretic approach to LTL model checking [36]. It was purely a library until Spot 1.0, when we started distributing command-line tools for LTL manipulation [13] and translation of LTL to some generalizations of Büchi Automata.

Spot 2.0 is a very large rewrite of the core of the library, in C++11, with a focus on supporting automata with arbitrary acceptance conditions as described in the Hanoi Omega Automata format (HOA) [6]. Those acceptance conditions are expressed as positive Boolean formulas over terms such as \(\text{Inf}(n)\) and \(\text{Fin}(n)\), which indicate respectively that some set \(S_n\) of states or transitions should be visited infinitely or finitely often. Traditional acceptance conditions look as follows in this formalism:

- **Büchi:** \(\text{Inf}(0)\)  
- **generalized-Büchi:** \(\text{Inf}(0) \land \text{Inf}(1) \land \text{Inf}(2) \land \ldots\)
- **co-Büchi:** \(\text{Fin}(0)\)  
- **generalized-co-Büchi:** \(\text{Fin}(0) \lor \text{Fin}(1) \lor \text{Fin}(2) \lor \ldots\)
- **Rabin:** \((\text{Fin}(0) \land \text{Inf}(1)) \lor (\text{Fin}(2) \land \text{Inf}(3)) \lor \ldots\)
- **Streett:** \((\text{Fin}(0) \lor \text{Inf}(1)) \land (\text{Fin}(2) \lor \text{Inf}(3)) \land \ldots\)

Parity acceptance, generalized-Rabin [24, 5], and any Boolean combination of the above can be expressed as well. The use of HOA as default format makes it easy to chain Spot’s command-line tools, and interact with other tools that implement HOA, regardless of the actual acceptance condition used.

Additionally, Spot 2.0 ships with Python bindings usable in interactive environments such as IPython/Jupyter [29], easing development, experimentation, and teaching.

Spot is a free software and can be obtained from https://spot.lrde.epita.fr/. The reader who wants to try Spot without installing it is invited to visit http://spot-sandbox.lrde.epita.fr/ where a live installation of Jupyter and Spot allows all examples (command lines or Python) of this paper to be replayed.
Figure 1 shows that Spot is actually split in three libraries. `libbddx` is a customized version of BuDDy [26] for representing Binary Decision Diagrams [10] which we use to label transitions in automata, and to implement a few algorithms [14, 4]. `libspot` is the main library containing all data structures and algorithms. `libspot-ltsmin` contains code to interface with state-spaces generated as shared libraries by LTSmin [20].

In the rest of this article, we highlight some of the features of Spot by presenting the command-line tools and the Python bindings built on top of these libraries. The reader should keep in mind that everything that we illustrate as shell command or in Python can be performed directly in C++; in fact our website gives several examples of tasks implemented with each of these three interfaces.

2 Command-line tools

Spot 2.0 installs the following eleven command-line tools, that are designed to be combined as traditional Unix tools.

- `randltl` generates random LTL/PSL formulas
- `genltl` generates LTL formulas from scalable patterns
- `ltlfillt` filters, converts, and transforms LTL/PSL formulas
- `ltl2tgba` translates LTL/PSL formulas into generalized Büchi automata [14], or deterministic parity automata (new in 2.0)
- `ltl2tgta` translates LTL/PSL formulas into Testing automata [8]
- `ltlcross` cross-compares LTL/PSL-to-automata translators to find bugs (works with arbitrary acceptance conditions since Spot 2.0)
- `ltlgrind` mutates LTL/PSL formulas to help reproduce bugs on smaller ones
- `dstar2tgba` converts ltl2dstar automata into Generalized Büchi automata [1]
- `randaut` generates random ω-automata
- `autfilt` filters, converts, and transforms ω-automata
- `ltldo` runs LTL/PSL formulas through other translators, providing uniform input and output interfaces

The first six tools were introduced in Spot 1.0 [13], and have since received several updates. For instance `ltl2tgba` now uses better simulation reductions and de-
generalization [4], and it now provides a way to output deterministic automata using transition-based parity acceptance; \texttt{ltlfilt} has learned to decide stutter-invariance of any LTL/PSL formula using an automaton-based check that is independent on the actual logic used [27]; and \texttt{ltlcross} can now perform precise equivalence checks of automata in addition to supporting arbitrary acceptance conditions—it has been used by the authors of \texttt{ltl3dra} [5], \texttt{ltl2dstar} [21][22], and Rabinizer 3 [23] to test recent releases of their respective tools.

The \texttt{dstar2tgba} tool was introduced in Spot 1.2 while working on the minimization of deterministic generalized Büchi automata using a SAT-solver [11]. It implements algorithms that translate deterministic Rabin automata into Büchi automata, preserving determinism if possible [25], as well as conversion from Streett to generalized Büchi. These two different kinds of input correspond to the possible outputs of \texttt{ltl2dstar}. In Spot 2.0, these specialized acceptance conversions have been preserved, but they are supplemented with more general transformations that input automata with arbitrary acceptance conditions, and transform them into automata with “Fin-less” acceptance, or with (Generalized) Büchi acceptance. These acceptance transformations are essential to a few core algorithms that cannot cope with arbitrary acceptance: for instance currently Spot can only check the emptiness of automata with Fin-less acceptance (all SCC-based emptiness-checks [11][12][31] are compatible with that), so more complex acceptances are transformed when needed.

All these acceptance transformations, as well as other automata transformations are available through the \texttt{autfilt} tool. This command can input a stream of automata in 4 different formats (HOA [6], LBT'T’s format [33], never claims [19], or \texttt{ltl2dstar}’s format [22]), and can output automata, maybe after filtering or transformation, in some other format (including GraphViz’s dot format [17] for display).

As an example of transformation and format conversion, consider:

```
% spin -f '[]<>a' | autfilt --complement --dot=abr | dot -Tpng >aut.png
```

This command translates the LTL formula $\text{GF}_a$ into a Büchi automaton using \texttt{spin} [19], the resulting never claim is then fed into \texttt{autfilt} for complementation, and the complemented automaton is output into GraphViz’s format for graphical rendering with \texttt{dot}. The arguments \texttt{a}, \texttt{b}, and \texttt{r} passed to \texttt{--dot} cause the acceptance condition to be displayed, and the acceptance marks to be shown as colored bullets.

In the above example the input to \texttt{autfilt} happens to be a deterministic Büchi automaton, so the complementation is as simple as changing the acceptance condition into co-Büchi. If a Büchi output is desired instead, the above command should be changed to \texttt{autfilt --complement --ba} and will output a non-deterministic Büchi automaton. This of course works with arbitrary acceptance conditions as input.

Complementation of non-deterministic automata is done via determinization. Our determinization algorithm inputs transition-based Büchi automata (so we may have some preprocessing to do if the input has a different acceptance), and outputs automata with transition-based Parity acceptance. It mixes the construction of Redziejowski [30] with some optimizations of \texttt{ltl2dstar} [21][22] and a few of our own.

The \texttt{ltldo} command wraps third-party LTL translators and provides them with inputs and outputs that are compatible with the Spot tool-suite. In particular it allows using “single-shot” translators in a pipeline. For instance \texttt{spin} can only translate one
formula at a time to produce a never claim. The command `ltldo spin` will process multiple formulas (in any syntax supported by Spot [13]), translate them all using `spin`, and output all results in any supported automaton format (HOA by default). For instance the following command uses Spin to translate 10 random LTL formulas into Büchi automata in the HOA format:

```
% randltl -n 10 a b | ltldo spin --name=%f
```

Option `--name=%f` requests input formulas to be used as the “name:” field in the HOA format. This field could then be used to retrieve the original formula after further processing: `autfilt --stats=%M` can be used to print the name of each input automaton.

As a more complex example, the following pipeline finds 10 formulas for which `ltl3ba` [3] produces a deterministic Büchi automaton, but `ltl2ba` [18] does not.

```
% randltl -n -1 a b |
  ltldo ltl3ba --name=%f | autfilt --is-deterministic --stats=%M |
  ltldo ltl2ba --name=%f | autfilt -v --is-deterministic --stats=%M -n 10
```

This creates an infinite (`-n -1`) stream of LTL formulas over atomic propositions `a` and `b`, translates them using `ltl3ba`, retains those that were translated to deterministic automata, translate them with `ltl2ba` and retains the non-deterministic ones (`-v` inverts matches, as with `grep`). With the final `--name=10`, the pipeline is eventually killed once the last command has found 10 matches.

The `autfilt` tool provides access to other ω-automata algorithms such as product, emptiness checks, language inclusion or equivalence, language-preserving simplifications of automata, refinement of labels [9], strength-based decompositions [32], SAT-based minimization of deterministic automata with arbitrary input and output acceptance [2], or conversion from transition-based acceptance to state-based acceptance. Most algorithms work with arbitrary acceptance conditions, except a few (emptiness checks, determinization) that currently have to reduce the acceptance conditions upfront.

## 3 The Python Interface

Similar tasks can be performed in a more “algorithm-friendly” environment using the Python interface. Combined with the IPython/Jupyter notebook [29] (a web application for interactive programming), this provides a nice environment for experiments, where automata and formulas are automatically displayed. Figure 2 shows two examples that we used in a practical lecture on model checking with students from EPITA.

The first example illustrates how LTL formulas can be parsed (`spot.formula()`), and then translated (using `translate()`) into automata with transition-based generalised Büchi acceptance. Using product, negation, and emptiness check, a student can define a procedure to test the equivalence of two LTL formulas and then use it to explore her understanding of LTL.

The second example illustrates the classical automata-theoretic approach to explicit LTL model checking [36]. Spot can read the shared-libraries used to represent state spaces in the LTSmin project [20]. Those can be compiled from Promela models using SpinS [35], or from DiVinE models using LTSmin’s modified version of DiVinE 2 [7]. In this example the `%%dve` keyword is used to specify a short DiVinE model called adding (this model comes from the BEEM database [28]) which is immediately compiled and
Fig. 2. Two examples of using the Python bindings of Spot in the Jupyter notebook.

loaded as a shared library. Printing the adding Python variable reveals that it is an object using the LTSmin interface, and lists the variables that can be used to build atomic proposition on this model. A Kripke structure can be instantiated from the model by providing a list of atomic propositions that should be valuated on each state. Displaying large Kripke structures is of course not very practical: by default Spot displays only the 50 first states (this can be changed using for instance the max states argument in the first cell). With this interface, we can now easily write a model check() procedure that inputs a model and a formula, instantiates a Kripke structure from the model using all the atomic propositions that appear in the formula, translates the negation of the formula into an automaton, and tests the emptiness of the product between the Kripke structure and this automaton. Note that otf product() performs an on-the-fly product: the state-space and the product are constructed as needed by the emptiness check algorithm.

4 Model checkers built using Spot

At the C++ level, the interface with LTSmin demonstrated above wraps the LTSmin state-space as a subclass of Spot’s Kripke structure class. This class basically just specifies the initial state and how to find the successors of a state, therefore allowing on-the-fly exploration. Model checkers like ITS-Tools [34] or Neco [16] have been implemented in the same way (both have been recently updated to Spot 2.0).
References


