Spot 2.0 — a framework for LTL and ω-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 ω-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 ω-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 $\mathsf{Inf}(n)$ and $\mathsf{Fin}(n)$, 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:	$\mathsf{Inf}(0)$	generalized-Büchi:	$\mathsf{Inf}(0) \wedge \mathsf{Inf}(1) \wedge \mathsf{Inf}(2) \wedge \dots$
co-Büchi:	$\mathsf{Fin}(0)$	generalized-co-Büchi:	$\mathsf{Fin}(0) \vee \mathsf{Fin}(1) \vee \mathsf{Fin}(2) \vee \dots$
Rabin:	$(\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)) \vee (\mathsf{Fin}(2) \wedge \mathsf{Inf}(3)) \vee \dots$		
Streett:	$(\mathsf{Fin}(0) \vee \mathsf{Inf}(1)) \wedge (\mathsf{Fin}(2) \vee \mathsf{Inf}(3)) \wedge \dots$		

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.

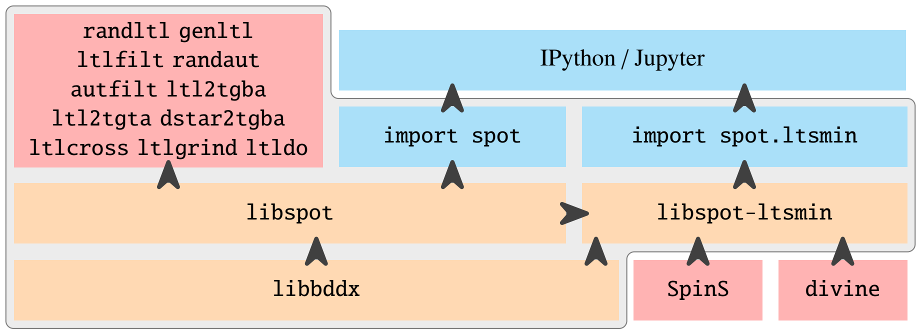


Fig. 1. Architecture of Spot. C++ libraries are in orange boxes, binaries in red, and Python packages in blue. The outlined area is what Spot distributes.

Figure 1 shows that Spot is actually split in three libraries. `libbddx` is a customized version de 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 web site 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
	`ltlfilt`	filter, converts, and transforms LTL/PSL formulas
[13]	`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 degeneralization [4], and it now provides a way to output deterministic automata using transition-based parity acceptance; `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 `ltlcross` can now perform precise equivalence checks of automata in addition to supporting arbitrary acceptance conditions—it has been used by the authors of `ltl3dra` [5], `ltl2dstar` [21, 22], and Rabinizer 3 [23] to test recent releases of their respective tools.

The `dstar2tgba` tool was introduced in Spot 1.2 while working on the minimization of deterministic generalized Büchi automata using a SAT-solver [1]. 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 `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 `autfilt` tool. This command can input a stream of automata in 4 different formats (HOA [6], LBTT's format [33], never claims [19], or `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 $\mathsf{GF}a$ into a Büchi automaton using `spin` [19], the resulting never claim is then fed into `autfilt` for complementation, and the complemented automaton is output into GraphViz's format for graphical rendering with `dot`. The arguments `a`, `b`, and `r` passed to `--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 `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 `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 `ltl2dstar` [21, 22] and a few of our own.

The `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 `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 `-n 10`, the pipeline is eventually killed once the last command has found 10 matches.

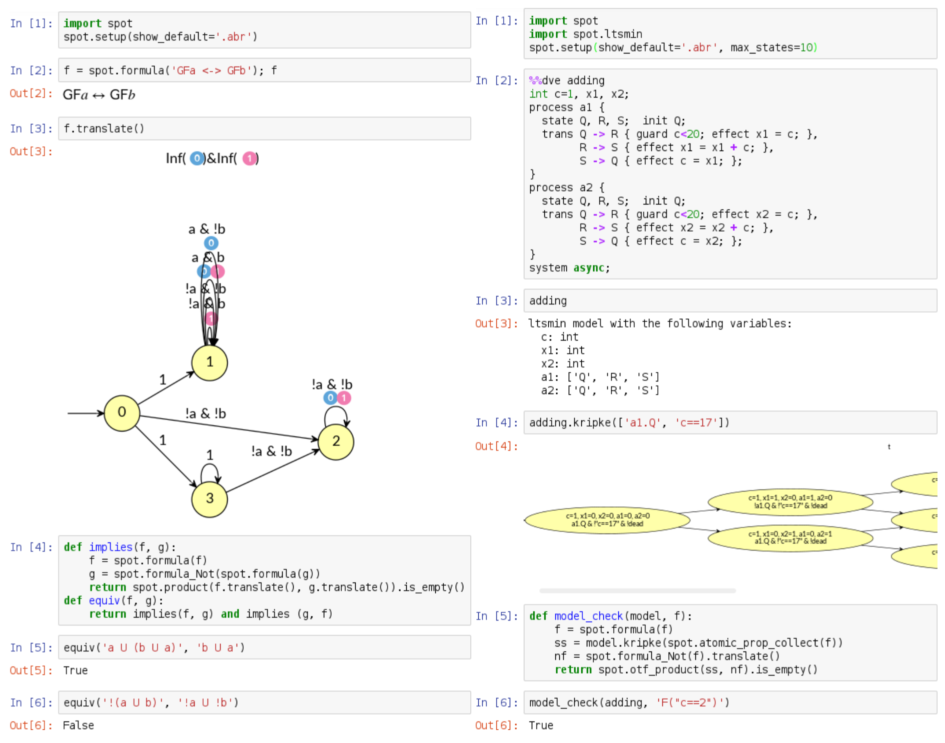
The `autfilt` tool provides access to other ω-automata algorithms such as product, emptiness checks, language inclusion or equivalence, language-preserving simplifications of automata, refinement of labels [9], strength-based decompositions [32], SAT-based minimization of deterministic automata with 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 generalized 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



```
In [1]: import spot
        spot.setup(show_default='.abr')

In [2]: f = spot.formula('GFa <-> GFb'); f
Out[2]: GFa ↔ GFb

In [3]: f.translate()
Out[3]:
```

```
In [4]: def implies(f, g):
            f = spot.formula(f)
            g = spot.formula_Not(spot.formula(g))
            return spot.product(f.translate(), g.translate()).is_empty()
        def equiv(f, g):
            return implies(f, g) and implies (g, f)

In [5]: equiv('a U (b U a)', 'b U a')
Out[5]: True

In [6]: equiv('!(a U b)', '!a U !b')
Out[6]: False
```

```
In [1]: import spot
        import spot.ltsmin
        spot.setup(show_default='.abr', max_states=10)

In [2]: %%dve adding
        int c=1, x1, x2;
        process a1 {
          state Q, R, S;  init Q;
          trans Q -> R { guard c<20; effect x1 = c; },
                R -> S { effect x1 = x1 + c; },
                S -> Q { effect c = x1; };
        }
        process a2 {
          state Q, R, S;  init Q;
          trans Q -> R { guard c<20; effect x2 = c; },
                R -> S { effect x2 = x2 + c; },
                S -> Q { effect c = x2; };
        }
        system async;

In [3]: adding
Out[3]: ltsmin model with the following variables:
          c: int
          x1: int
          x2: int
          a1: ['Q', 'R', 'S']
          a2: ['Q', 'R', 'S']

In [4]: adding.kripke(['a1.Q', 'c==17'])
Out[4]:
```

c=1, x1=0, x2=0, a1=0, a2=0
a1.Q & !"c==17" & !dead
c=1, x1=1, x2=0, a1=1, a2=0
!a1.Q & !"c==17" & !dead
c=1, x1=0, x2=1, a1=0, a2=1
a1.Q & !"c==17" & !dead

```
In [5]: def model_check(model, f):
            f = spot.formula(f)
            ss = model.kripke(spot.atomic_prop_collect(f))
            nf = spot.formula_Not(f).translate()
            return spot.otf_product(ss, nf).is_empty()

In [6]: model_check(adding, 'F("c==2")')
Out[6]: True
```

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, instanciates 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

1. S. Baarir and A. Duret-Lutz. Mechanizing the minimization of deterministic generalized Büchi automata. In *FORTE'14*, vol. 8461 of *LNCS*, pp. 266–283. Springer, 2014.
2. S. Baarir and A. Duret-Lutz. SAT-based minimization of deterministic ω-automata. In *LPAR'15*, vol. 9450 of *LNCS*, pp. 79–87. Springer, 2015.
3. T. Babiak, M. Křetínský, V. Řehák, and J. Strejček. LTL to Büchi automata translation: Fast and more deterministic. In *TACAS'12*, vol. 7214 of *LNCS*, pp. 95–109. Springer, 2012.
4. T. Babiak, T. Badie, A. Duret-Lutz, M. Křetínský, and J. Strejček. Compositional approach to suspension and other improvements to LTL translation. In *SPIN'13*, vol. 7976 of *LNCS*, pp. 81–98. Springer, 2013.
5. T. Babiak, F. Blahoudek, M. Křetínský, and J. Strejček. Effective translation of LTL to deterministic Rabin automata: Beyond the (F, G)-fragment. In *ATVA'13*, vol. 8172 of *LNCS*, pp. 24–39. Springer, 2013.
6. 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'15*, vol. 9206 of *LNCS*. Springer, 2015. See also `http://adl.github.io/hoaf/`.
7. J. Barnat, L. Brim, and P. Rockai. DiVinE 2.0: High-performance model checking. In *HiBi'09*, pp. 31–32. IEEE Computer Society Press, 2009.
8. A. E. Ben Salem, A. Duret-Lutz, and F. Kordon. Model checking using generalized testing automata. *Transactions on Petri Nets and Other Models of Concurrency (ToPNoC VI)*, 7400: 94–112, 2012.
9. F. Blahoudek, A. Duret-Lutz, V. Rujbr, and J. Strejček. On refinement of Büchi automata for explicit model checking. In *SPIN'15*, vol. 9232 of *LNCS*, pp. 66–83. Springer, 2015.
10. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
11. J.-M. Couvreur. On-the-fly verification of temporal logic. In *FM'99*, vol. 1708 of *LNCS*, pp. 253–271. Springer, 1999.
12. J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *SPIN'05*, vol. 3639 of *LNCS*, pp. 143–158. Springer, 2005.
13. A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In *ATVA'13*, vol. 8172 of *LNCS*, pp. 442–445. Springer, 2013.
14. A. Duret-Lutz. LTL translation improvements in Spot 1.0. *Int. J. on Critical Computer-Based Systems*, 5(1/2):31–54, 2014.
15. A. Duret-Lutz and D. Poitrenaud. SPOT: an Extensible Model Checking Library using Transition-based Generalized Büchi Automata. In *MASCOTS'04*, pp. 76–83. IEEE Computer Society Press, 2004.
16. Ł. Fronc and A. Duret-Lutz. LTL model checking with Neco. In *ATVA'13*, vol. 8172 of *LNCS*, pp. 451–454. Springer, 2013. Code moved to `https://github.com/Lvyn/neco-net-compiler`.
17. E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000.
18. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV'01*, vol. 2102 of *LNCS*, pp. 53–65. Springer, 2001.
19. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
20. G. Kant, A. Laarman, J. Meijer, J. Pol, S. Blom, and T. Dijk. LTSmin: High-performance language-independent model checking. In *TACAS'15*, vol. 9035 of *LNCS*, pp. 692–707. Springer, 2015.

21. J. Klein and C. Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theoretical Computer Science*, 363(2):182–195, 2006.
22. J. Klein and C. Baier. On-the-fly stuttering in the construction of deterministic ω-automata. In *CIAA'07*, vol. 4783 of *LNCS*, pp. 51–61. Springer, 2007.
23. Z. Komárková and J. Křetínský. Rabinizer 3: Safraless translation of LTL to small deterministic automata. In *ATVA'14*, vol. 8837 of *LNCS*, pp. 235–241. Springer, 2014.
24. J. Křetínský and J. Esparza. Deterministic automata for the (F,G)-fragment of LTL. In *CAV'12*, vol. 7358 of *LNCS*, pp. 7–22. Springer, 2012.
25. S. C. Krishnan, A. Puri, and R. K. Brayton. Deterministic ω-automata vis-a-vis deterministic Büchi automata. In *ISAAC'94*, vol. 834 of *LNCS*, pp. 378–386. Springer, 1994.
26. J. Lind-Nielsen and H. Cohen. BuDDy: Binary Decision Diagram package. Release 2.4, 2014. `https://sourceforge.net/projects/buddy/`.
27. T. Michaud and A. Duret-Lutz. Practical stutter-invariance checks for ω-regular languages. In *SPIN'15*, vol. 9232 of *LNCS*, pp. 84–101. Springer, 2015.
28. R. Pelánek. BEEM: benchmarks for explicit model checkers. In *Proc. of the 14th international SPIN conference on Model checking software*, LNCS, pp. 263–267. Springer, 2007.
29. F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, 2007. See also `http://ipython.org`.
30. R. Redziejowski. An improved construction of deterministic omega-automaton using derivatives. *Fundamenta Informaticae*, 119(3-4):393–496, 2012.
31. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In *LPAR'13*, vol. 8312 of *LNCS*, pp. 668–682. Springer, 2013.
32. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Strength-based decomposition of the property Büchi automaton for faster model checking. In *TACAS'13*, vol. 7795 of *LNCS*, pp. 580–593. Springer, 2013.
33. H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *STTT*, 4(1):57–70, 2002.
34. Y. Thierry-Mieg. Symbolic model-checking using its-tools. In *TACAS'15*, pp. 231–237. Springer, 2015.
35. F. I. van der Berg and A. W. Laarman. SpinS: Extending LTSmin with Promela through SpinJa. In *PDMC'12*, vol. 296 of *ENTCS*, pp. 95–105. Elsevier, 2012.
36. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Banff'94*, vol. 1043 of *LNCS*, pp. 238–266. Springer, 1996.