From Spot 2.0 to Spot 2.10: What's New?

Alexandre Duret-Lutz[1], Étienne Renault[1], Maximilien Colange[2], Florian Renkin[1], Alexandre Gbaguidi Aisse[2], Philipp Schlehuber-Cassier[1], Thomas Medioni[2], Antoine Martin[1], Jérôme Dubois[1], Clément Gillard[2], and Henrich Lauko[2]

[1] LRDE, EPITA, Le Kremlin-Bicêtre, France
{adl, renault, frenkin, philipp, amartin, jdubois}@lrde.epita.fr
[2] Previously at LRDE.

Abstract. Spot is a C++17 library for LTL and ω-automata manipulation, with command-line utilities, and Python bindings. This paper summarizes its evolution over the past six years, since the release of Spot 2.0, which was the first version to support ω-automata with arbitrary acceptance conditions, and the last version presented at a conference. Since then, Spot has been extended with several features such as acceptance transformations, alternating automata, games, LTL synthesis, and more. We also shed some lights on the data-structure used to store automata.

1 Availability, Purpose, and Evolution

Spot is a library for LTL and ω-automata manipulation, distributed under a GPLv3 license. Its source code is available from `https://spot.lrde.epita.fr/`. We provide packages for some Linux distributions like Debian and Fedora, but other packages can also be found for Conda-Forge [15] (for Linux & Darwin), Arch Linux, FreeBSD...

Spot can be used via three interfaces: a C++17 library, a set of command-line tools that give easy access to many features of the library, and Python bindings, that makes prototyping and interactive work very attractive. Our web site now contains many examples of how to perform some tasks using these three interfaces, and we have a public mailing list for questions.

In our last tool paper [18], Spot 2.0 had just converted from being a library for working on Transition-based Generalized Büchi Automata and had become a library supporting ω-automata with arbitrary Emerson-Lei [19, 32] acceptance conditions, as enabled by the development of the HOA format [5].

In the HOA format, transitions can carry multiple colors, and acceptance conditions are expressed as a positive Boolean formulas over atoms like $\mathsf{Fin}(i)$ or $\mathsf{Inf}(i)$ that tell if a color should be seen finitely or infinitely often for a run to be accepting. Table 1 gives some examples.

While Spot 2.0 was able to read automata with arbitrary acceptance conditions, not all of its algorithms were able to support such a generality. For instance testing an automaton for emptiness or finding an accepting word, would only work on automata with "Fin-less" acceptance conditions. For other conditions,

Table 1. Acceptance formulas corresponding to classical names.

Büchi	$\mathsf{Inf}(0)$
generalized Büchi	$\bigwedge_i \mathsf{Inf}(i)$
Fin-less [9]	any positive formula of $\mathsf{Inf}(...)$
co-Büchi	$\mathsf{Fin}(0)$
generalized co-Büchi	$\bigvee_i \mathsf{Fin}(i)$
Rabin	$\bigvee_i (\mathsf{Fin}(m_{2i}) \wedge \mathsf{Inf}(m_{2i+1}))$
generalized Rabin [23]	$\bigvee_i (\mathsf{Fin}(m_i) \wedge \bigwedge_{j \in J_i} \mathsf{Inf}(m_j))$
Streett	$\bigwedge_i (\mathsf{Inf}(m_{2i}) \vee \mathsf{Fin}(m_{2i+1}))$
parity min even	$\mathsf{Inf}(0) \vee (\mathsf{Fin}(1) \wedge (\mathsf{Inf}(2) \vee (\mathsf{Fin}(3) \wedge \ldots)))$
parity min odd	$\mathsf{Fin}(0) \wedge (\mathsf{Inf}(1) \vee (\mathsf{Fin}(2) \wedge (\mathsf{Inf}(3) \vee \ldots)))$
parity max even	$\ldots \vee (\mathsf{Fin}(3) \wedge (\mathsf{Inf}(2) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(0))))$
parity max odd	$\ldots \wedge (\mathsf{Inf}(3) \vee (\mathsf{Fin}(2) \wedge (\mathsf{Inf}(1) \vee \mathsf{Fin}(0))))$

Spot 2.0 would rely on a procedure called `remove_fin()` to convert automata with arbitrary acceptance conditions into "Fin-less" acceptance conditions [9]. This was ultimately fixed by developing a generic emptiness check [6]. Additionally the support for arbitrary acceptance conditions has allowed us to implement many useful algorithms; the most recent being the Alternating Cycle Decomposition [13, 14] a powerful data structure with many applications (conversion to parity acceptance, degeneralization, typeness checks...)[3].

There have been over 2400 commits and 55 releases of Spot since version 2.0, but only 10 of these are major releases. Releases are numbered $2.x.y$ where y is updated for minor upgrades that mostly fix bugs, and x is updated for major release that add new features. (The leading 2 would be incremented in case of a serious redesign of the API.) Table 2 summarizes the highlights of the various releases in chronological order. Not appearing in this list are many micro-optimizations and usability improvements that Spot has accumulated over the years. The rest of the text mostly focuses on how the storage for automata evolved to support alternation, games, and Mealy machines.

2 Automata Representation

The main automaton class of Spot is called `twa_graph` and inherits from the `twa` class. The letters `twa` stand for *Transition-based ω-Automaton.*

The class `twa` implements an abstract interface that allows on-the-fly exploration of an automaton similar to what had been present in Spot from the start: essentially, one can query the initial state, and query the transitions leaving any known state. In particular, before exploring the state-space of a `twa`, it is unknown how many states are reachable. Various subclasses of `twa` are provided in Spot, for instance to represent the state-space of Promela or Divine models [18]. Users may create subclasses, for instance to create a Kripke structure on-the-fly.[4]

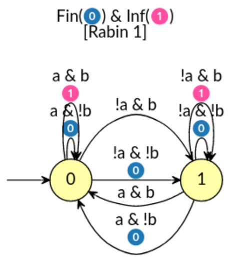
[3] `https://spot.lrde.epita.fr/ipynb/zlktree.html`

[4] As demonstrated by `https://spot.lrde.epita.fr/tut51.html`

Table 2. Milestones in the history of Spot.

2004	0.x	C++03	Prehistory of the project. [17]
2012	0.9		Support for some PSL operators.
2013	1.0		Command-line tools, mostly focused on LTL/PSL input [16]. Includes `ltlcross`, a clone of LBTT [33]. Python bindings.
	1.1		Automatic detection of stutter-invariant formulas. [27]
	1.2		SAT-based minimization [3, 4]. `ltlcross` and the new `dstar2tgba` can read Rabin and Streett automata produced by `ltl2dstar` [22].
2016	2.0	C++11	Rewrite of the LTL formulas representation. Rewrite of the automaton class to allow arbitrary acceptance. Support for the HOA format. More command-line tools, now that automata can be exchanged with other tools. [18] New determinization procedure.
	2.1		Conversion to generalized Streett or Rabin. Small usability improvements all around (like better support for CSV files).
	2.2		LTLf→LTL conversion [21]. Faster simulation-based reduction of deterministic automata.
2017	2.3		Initial support for alternating automata and alternation removal. 400% faster emptiness check. Incremental SAT-based minimization. Classification in the temporal hierarchy of Manna & Pnueli [25].
	2.4	C++14	New command-line tools: `autcross` to check and compare automata transformations, `genaut` to generate families of automata. Dualization of automata. Conversion from Rabin to Büchi [24] updated to support transition-based input. Relabeling of LTL formulas with large Boolean subformulas to speedup their translation.
2018	2.5		New command-line tool `ltlsynt` for synthesis of AIGER circuits from LTL specifications. [26] Conversions to co-Büchi [10]. Utilities for converting between parity acceptance conditions. Detection of stutter-invariant *states*. Determinization optimized.
	2.6		Compile-time option to support more than 32 colors. Specialized translation for formulas of the type $\mathsf{GF}(\varphi)$ if φ is a guarantee. New translation mode to output automata with unconstrained acceptance condition. Semi-deterministic complementation [8]. Faster detection of obligation properties. Online LTL translator replaced by a new web application (see Figure 4).
	2.7		LAR-based paritization in `ltlsynt`. Generic emptiness check [6]. Detection of liveness properties [2].
2019	2.8		Accepting run extraction for arbitrary acceptance. Introduction of an "`output_aborter`" to abort constructions that are too large. Support for SVA's delay syntax, and `first_match` operator [1]. Minimization of parity acceptance [12].
2020	2.9		Better paritization, partial degeneralization, and acceptance simplifications [30]. Weak and strong variants of X. Xor product of automata, used while translating formulas to automata with unconstrained acceptance.
2021	2.10	C++17	`ltlsynt` overhauled [31]. Support for games and Mealy machines. Mealy machines simplifications. Multiple encodings from Mealy machine to AIGER. Experimental `twacube` class for parallel algorithms. Support for transition-based Büchi. Zielonka Trees and Alternating Cycle Decomposition [13, 14]

```
In [2]: aut = spot.translate('GF(a <-> Xa) & FGb', 'det', 'gen')
        aut
Out[2]:
```



```
In [3]: aut.show_storage()
Out[3]:
```

states	0	1
succ	1	5
succ_tail	4	8

edges	1	2	3	4	5	6	7	8
cond	a & !b	a & b	!a & b	!a & !b	a & b	a & !b	!a & !b	!a & b
acc	{0}	{1}	{}	{0}	{}	{0}	{0}	{1}
dst	0	0	1	1	0	0	1	1
next_succ	2	3	4	0	6	7	8	0
src	0	0	0	0	1	1	1	1

init_state:	0
num_sets:	2
acceptance:	Fin(0) & Inf(1)
ap_vars:	b a

prop_state_acc:	maybe
prop_inherently_weak:	maybe
prop_terminal:	no
prop_weak:	maybe
prop_very_weak:	maybe
prop_complete:	maybe
prop_universal:	yes
prop_unambiguous:	yes
prop_semi_deterministic:	yes
prop_stutter_invariant:	maybe

Fig. 1. Internal representation of a `twa_graph` as two vectors.

The class `twa_graph`, introduced in Spot 2.0, implements an explicit, graph-based, representation of an automaton, in which states and edges are designated by integers. This makes for a much simpler interface[5] and usually simplifies the data structures used in algorithms (since states and edges can be used as indices in arrays). The data structure is best illustrated by using the `show_storage()` method of the Python bindings, as shown by Figure 1. Essentially, a `twa_graph` is stored as two C++ vectors: a vector of states, and a vector of edges. For each state, the first vector stores two edge numbers: `succ` is the first outgoing edge, and `succ_tail` is the last one. These number are indices into the edge vector, which stores five pieces of information per edge. Four of them are related to the identity of the edge: `src`, `dst`, `cond`, `acc` are respectively the source, destination, guard, and color sets of the edge. The remaining field, `next_succ` gives the next outgoing edge, effectively creating a linked list of all edges leaving a given state. There is no edge 0, so this value is used as terminator for such lists. Outgoing edges of the same state are not necessarily adjacent in that structure. When a

[5] Contrast on-the-fly and explicit APIs at `https://spot.lrde.epita.fr/tut50.html`.

new edge is added to the automaton, it is simply appended to the edge vector, and the `succ_tail` field of the state is used to locate the previous end of the list to update it.

To iterate over successors of state 1 in C++ or Python, one can ignore the above linked list implementation and write one of the following loops:

```
for (auto& e: aut->out(1))
  // use e.cond, e.acc, e.dst
```

```
for e in aut.out(1):
    # use e.cond, e.acc, e.dst
```

The `twa_graph::out` methods simply returns a lightweight temporary object which can be iterated upon using iterators that will follow the linked list. Then the object `e` is effectively a reference to a column of the edge vector.

As seen on Figure 1, the automaton additionally stores an initial state (Spot only supports a single initial state), a number of colors (`num_sets`), an acceptance condition, a list of atomic propositions (Spot only supports alphabets of the form 2^{AP}), and 10 fields storing structural properties of the automaton.

These property fields have only three possible values: they default to *maybe*, but can be set to *no* or *yes* by algorithms that work on the automaton. They can also be read and written in the HOA format. For instance if `prop_universal` is set to *yes*, it means that automaton does not have any existantial choice (a.k.a. non-determinism). Spot's *is_deterministic()* algorithm can return in constant time if `prop_universal` is known, otherwise it will inspect the automaton and set that property before returning, so that the next call to *is_deterministic()* will be instantaneous. Some algorithms know how to take advantage of any hint they get from those properties: for instance the `product()` of two automata is optimized to use fewer colors when one of the arguments is known to be weak (i.e., in an SCC all transitions have the same colors).

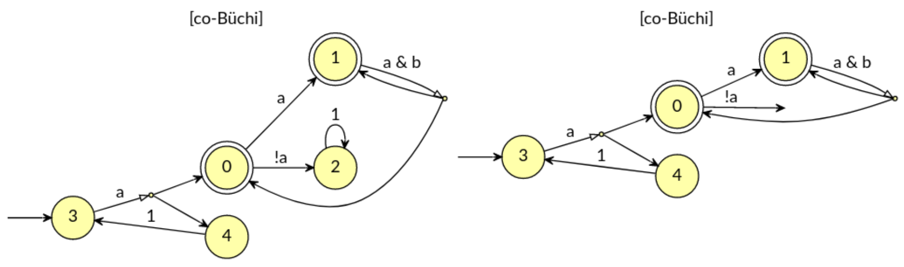
A drawback of these properties, is that algorithms that modify an automaton in place always have to remember that they may need to update the properties. This has caused a couple of bugs over the years.

3 Introduction of Alternating Automata

Support for alternating ω-automata, as defined in the HOA format, was added to Spot in version 2.3 without introducing a new class. Rather, the `twa_graph` class was extended to support alternation in such a way that existing algorithms would not require any modifification to continue working on automata without universal branching. This was done by reserving the sign bit of a state number to denote a *universal destination group*. Testing whether the destination of an edge has its sign bit set can be done efficiently by the processor, and when that is detected, the complement of that number can be used as an index in a third vector that stores the sequences of states that form those *universal destination groups*.

Figure 2 shows an example of Alternating automaton (top-left) with co-Büchi acceptance. In many works on alternating automata, it is conventional to not

```
display_inline(aut2, aut2.show('.u'), per_row=2)
```



```
aut2.show_storage()
```

states	0	1	2	3	4
succ	1	3	4	5	6
succ_tail	2	3	4	5	6

edges	1	2	3	4	5	6
cond	a	!a	a & b	1	a	1
acc	{0}	{0}	{0}	{ }	{ }	{ }
dst	1	2	~0	2	~3	3
next_succ	2	0	0	0	0	0
src	0	0	1	2	3	4

dests	~0			~3		
#cnt/dst	#2	0	1	#2	0	4

init_state:	3
num_sets:	1
acceptance:	Fin(0)
ap_vars:	b a

prop_state_acc:	yes
prop_inherently_weak:	maybe
prop_terminal:	maybe
prop_weak:	maybe
prop_very_weak:	maybe
prop_complete:	no
prop_universal:	yes
prop_unambiguous:	yes
prop_semi_deterministic:	yes
prop_stutter_invariant:	maybe

Fig. 2. Internal representation of alternating automata.

represent accepting sinks, and instead have transition without destination. The top-right picture shows that Spot has a rendering option to hide accepting sinks.

The bottom of the figure shows that the automaton has `prop_state_acc` set, which means that the automaton is meant to be interpreted as using state-based acceptance. Colors are still stored on edges internally, but all edges leaving a state have the same colors. Seeing that the condition is co-Büchi ($\mathsf{Fin}(0)$), the display code automatically switched to the convention of using double-circles for rejecting states.

Universal destination groups appear as pink in the figure. There are two groups here: `~0` and `~3`. The complement of these numbers can be used as indices in the `dests` vector. At the given index, one can read the size n of the destination group, followed by the state number of the n destinations.

Algorithms that work on alternating automata need to be able to iterate over all destinations of an edge. The process of checking the sign bit of the destination to decide if its a group, and to iterate on that group is hidden by the `univ_dests()` method:

```
for(auto& e: aut->out(1)) {
 // use e.cond, e.acc, e.src
 for(unsigned d:aut->univ_dests(e))
  // use d
}
```

```
for e in aut.out(1):
  # use e.cond, e.acc, e.src
  for d in aut.univ_dests(e):
     # use d
```

Note that this code works on non-universal branches as well: if `e.dst` is unsigned, `univ_dests(e)` will simply iterate on that unique value.

Spot has two alternation removal procedures. One is an on-the-fly implementation of the Breakpoint construction [28] which transforms an n-state alternating Büchi automaton into a non-alternating Büchi automaton with at most 3^n states. For very weak alternating automata, it is know that a powerset-based procedure can produce a transition-based generalized Büchi automaton with 2^n states [20]; in fact that algorithm even works on *ordered* automata [11], i.e., alternating automata where the only rejecting cycles are self-loops. The second alternation removal procedure of Spot is a mix between these two procedures but does not work on the fly: it takes a *weak* automaton as input, and uses the break-point construction on rejectings SCCs that have more than one state, and uses the powerset construction for other SCCs.

4 Extending Automata via Named Properties

Spot's automata have a mechanism to attach arbitrary data to automata, called *named properties*. (This is similar to the notion of attributes in the R language.) An object can be attached to the automaton with:

```
  aut->set_named_prop("property-name", new mytype(...));
```

and later retrieved with:

```
  mytype* data = aut->get_named_prop<mytype>("property-name");
```

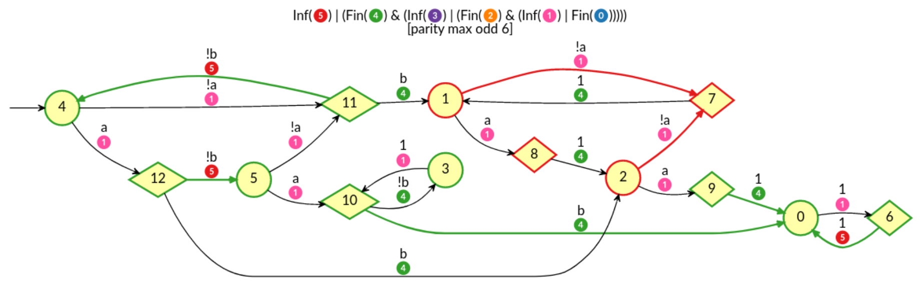
Ensuring that `mytype` is the correct type for the retrieved property is the programmer's responsability.

Spot has grown a list of many such properties over time.[6] For instance `automaton-name` stores a string that would be displayed as the name of the automaton. The `highlight-edges` and `highlight-states` properties can be used to color edges and states. The `state-names` is a vector of strings that gives a name to each state, etc. While those examples are mostly related to the graphical rendering of the automata, some algorithms store useful byproducts as properties. For instance the `product()` algorithm will define a `product-states` named property that store a vector of pairs of the original states.

These named properties are sometimes used to provide additonnal semantics to the automaton, for instance to obtain a game or a Mealy machine.

[6] `https://spot.lrde.epita.fr/concepts.html#named-properties`

```
game = spot.automaton("ltlsynt --outs=b -f 'F(a&Xa) <-> Fb' --print-game-hoa |")
spot.solve_game(game)
spot.highlight_strategy(game)
game
```



```
mealy = spot.solved_game_to_mealy(game)
aig = spot.mealy_machine_to_aig(mealy, "isop")
display_inline(mealy, aig.show('h'))
```

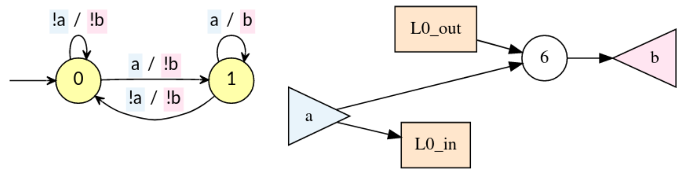


Fig. 3. (top) Solving a game to display the strategy. States with green borders are winning for player 1, who wants to satisfy the acceptance condition, by following the green arrows. States with red color are winning for player 0, who wants to fail the acceptance condition, by following the red arrow. (bottom) Conversion of the winning strategy to a Mealy machine and then an AIGER circuit.

5 Games, Mealy Machines, and LTL Synthesis

The application of Spot to LTL synthesis was introduced in Spot 2.5 in the form of the `ltlsynt` tool [26], but the inner workings of this tool were progressively redesigned and publicly exposed until version 2.10.

An automaton can now be turned into a game by attaching the `state-player` property to it.[7] Only two-player games are supported, so `state-player` should be a `std::vector<bool>`. Currently Spot has solvers for safety games or for games with parity max odd acceptance, but we plan to at least generalize the later to any kind of parity condition. Once a game has been solved, it contains two new named properties: `state-winner` (another `std::vector<bool>` indexed by state numbers), and `strategy` (a `std::vector<unsigned>` that gives for each state the edge that its owner should follow).

Figure 3 shows an example of game generated by `ltlsynt`, and how we can display the winning strategy once the game is solved. The winning strategy can be extracted and converted into a Mealy machine, which is just an automaton that uses the `synthesis-output` property to specify which atomic propositions

[7] `https://spot.lrde.epita.fr/tut40.html` illustrates how a game you be used to decide if a state simulates another one.

belong to the output. Such a Mealy machine can then be encoded into an AND-inverter graph, and saved into the AIGER format [7]. Here L0 represents a latch, basically one bit of memory, that stores the previous value of a so that the circuit can output b if and only if a is true in the present and in the previous step.

6 Online Application for LTL Formulas

The Python ecosystem makes it easy to develop web interfaces for convenient access to a subset of features of Spot. For instance Figure 4 shows screenshots (taken on a mobile phone) of a web application built using a React frontend, and running Spot on the server. It can transform LTL formulas into automata using various acceptance conditions, can display many properties of a formula (membership to the Manna & Pnueli hierarchy [25], Safety/Liveness classification [2], Rabin and Streett indices [12], stutter-invariance [27]), or simply compare two formulas. In the latter case the two given LTL formula are related using a Venn diagram, and an example word belonging to each zone is given.

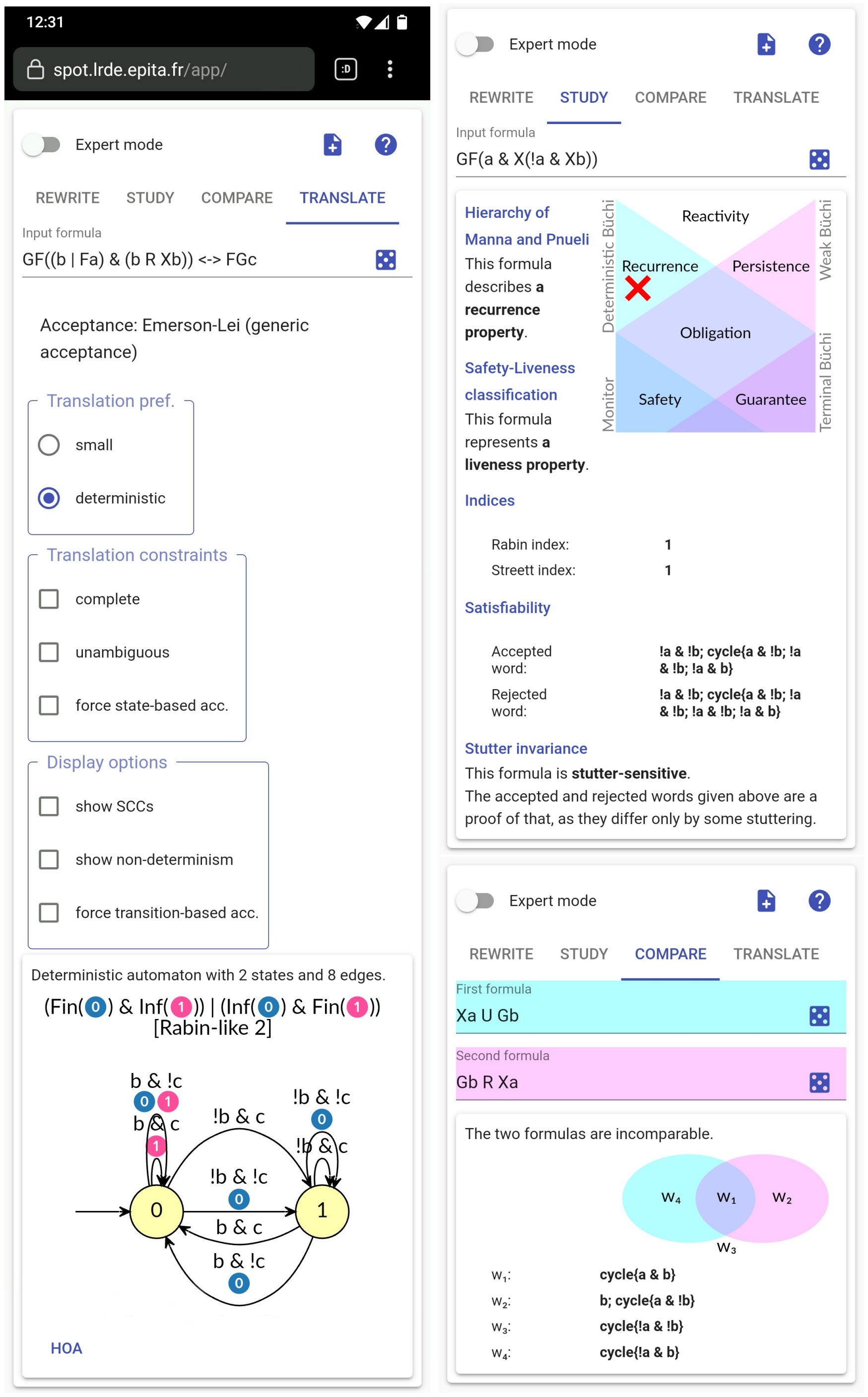
7 Shortcomings and one Future Direction

While Spot has been used for many applications[8], there are two recurrent issues: they are related to the types used for some fields of the edge vector (see Figures 1–2). By default, the set of colors that labels an edge (the `acc` field) is stored as a 32-bit bit-vector, the transition label (`cond`, a formula over 2^{AP}), is stored as a BDD identified by a unique 32-bit integer, and the other three fields (`src`, `dst`, `next_succ`) are all 32-bit integers. One edge therefore takes 20 bytes.

While limiting the number of states to 32-bit integers has never been a problem so far, the limit of 32 colors can be hit easily. Spot 2.6 added a compile-time option to enlarge the number of supported colors to any multiple of 32; this evidently has a memory cost (and therefore also a runtime cost) as the `acc` field will be larger for each edge. However this constraint generally means that all the algorithms we implement try to be "color-efficient", i.e., to not introduce useless colors. For instance while the product of an automaton with x colors and an automaton with y colors is usually an automaton with $x + y$ colors, the `product()` implementation will output fewer colors in presence of a *weak* automaton.

The use of BDDs as edge labels causes another type of issues. Spot uses a customized version of the BuDDy library, with additional functions, and several optimizations (more compact BDD nodes for better cache friendliness, most operations have been rewritten to be recursion-free). However BuDDy is inherently not thread safe, because of its global unicity table and caches. This prevents us from doing any kind of parallel processing on automata. A long term plan is to introduce a new class `twacube` that represent an automaton in which edges are cubes (i.e., conjunctions of literals) represented using two bit-vectors. Such a class was experimentally introduced in Spot 2.10 and is currently used in some parallel emptiness check procedures [29].

[8] The previous tool paper [18] has over 250 citations according to Google scholar



12:31

spot.lrde.epita.fr/app/

Expert mode

REWRITE STUDY COMPARE TRANSLATE

Input formula

GF((b | Fa) & (b R Xb)) <-> FGc

Acceptance: Emerson-Lei (generic acceptance)

Translation pref.

- small
- [✓ deterministic]

Translation constraints

- ☐ complete
- ☐ unambiguous
- ☐ force state-based acc.

Display options

- ☐ show SCCs
- ☐ show non-determinism
- ☐ force transition-based acc.

Deterministic automaton with 2 states and 8 edges.

(Fin(0) & Inf(1)) | (Inf(0) & Fin(1))
[Rabin-like 2]

HOA

Expert mode

REWRITE STUDY COMPARE TRANSLATE

Input formula

GF(a & X(!a & Xb))

Hierarchy of Manna and Pnueli

This formula describes a **recurrence property**.

Safety-Liveness classification

This formula represents a **liveness property**.

Indices

Rabin index:	1
Streett index:	1

Satisfiability

Accepted word:	!a & !b; cycle{a & !b; !a & !b; !a & b}
Rejected word:	!a & !b; cycle{a & !b; !a & !b; !a & !b; !a & b}

Stutter invariance

This formula is **stutter-sensitive**.
The accepted and rejected words given above are a proof of that, as they differ only by some stuttering.

Expert mode

REWRITE STUDY COMPARE TRANSLATE

First formula

Xa U Gb

Second formula

Gb R Xa

The two formulas are incomparable.

w_1:	cycle{a & b}
w_2:	b; cycle{a & !b}
w_3:	cycle{!a & !b}
w_4:	cycle{!a & b}

Fig. 4. A web application, built on top of Spot. `https://spot.lrde.epita.fr/app/`

References

1. *1800-2017 - IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. IEEE, 2018.
2. B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
3. S. Baarir and A. Duret-Lutz. Mechanizing the minimization of deterministic generalized Büchi automata. In *FORTE'14*, *LNCS* 8461, pp. 266–283. Springer, 2014.
4. S. Baarir and A. Duret-Lutz. SAT-based minimization of deterministic ω-automata. In *Proc. of the 20th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-20)*, *LNCS* 9450, pp. 79–87. Springer, 2015.
5. 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*, *LNCS* 8172, pp. 442–445. Springer, 2015. See also `http://adl.github.io/hoaf/`.
6. C. Baier, F. Blahoudek, A. Duret-Lutz, J. Klein, D. Müller, and J. Strejček. Generic emptiness check for fun and profit. In *ATVA'19*, *LNCS* 11781, pp. 445–461. Springer, 2019.
7. A. Biere, K. Heljanko, and S. Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.
8. F. Blahoudek, M. Heizmann, S. Schewe, J. Strejček, and M.-H. Tsai. Complementing semi-deterministic Büchi automata. In *Proc. of the 22th International Conférence on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 770–787. Springer, 2016.
9. V. Bloemen, A. Duret-Lutz, and J. van de Pol. Model checking with generalized Rabin and Fin-less automata. *International Journal on Software Tools for Technology Transfer*, 2019.
10. U. Boker and O. Kupferman. Co-büching them all. pp. 184–198, 2011. URL `http://www.cs.huji.ac.il/~ornak/publications/fossacs11b.pdf`.
11. U. Boker, O. Kupferman, and A. Rosenberg. Alternation removal in Büchi automata. In *ICALP'10*, *LNCS* 6199, pp. 76–87. Springer, 2010.
12. O. Carton and R. Maceiras. Computing the Rabin index of a parity automaton. *Informatique théorique et applications*, 33(6):495–505, 1999. URL `http://www.numdam.org/item/ITA_1999__33_6_495_0/`.
13. A. Casares, T. Colcombet, and N. Fijalkow. Optimal transformations of games and automata using Muller conditions. In *ICALP'21*, vol. 198, pp. 123:1–123:14, 2021.
14. A. Casares, A. Duret-Lutz, K. J. Meyer, F. Renkin, and S. Sickert. Practical applications of the alternating cycle decomposition. In *Proc. of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS, 2022. To appear.
15. C.-F. Community. The conda-forge project: Community-based software distribution built on the conda package format and ecosystem, July 2015. URL `https://doi.org/10.5281/zenodo.4774216`.
16. A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In *ATVA'13*, *LNCS* 8172, pp. 442–445. Springer, 2013.
17. 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.

18. 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'16*, *LNCS* 9938, pp. 122–129. Springer, 2016.
19. E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.
20. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *CAV'01*, *LNCS* 2102, pp. 53–65. Springer, 2001.
21. G. D. Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *IJCAI'13*, pp. 854–860, 2013.
22. J. Klein and C. Baier. On-the-fly stuttering in the construction of deterministic ω-automata. In *CIAA'07*, *LNCS* 4783, pp. 51–61. Springer, 2007.
23. J. Křetínský and J. Esparza. Deterministic automata for the (F,G)-fragment of LTL. In *CAV'12*, *LNCS* 7358, pp. 7–22. Springer, 2012.
24. S. C. Krishnan, A. Puri, and R. K. Brayton. Deterministic ω-automata vis-a-vis deterministic Büchi automata. In *ISAAC'94*, *LNCS* 834, pp. 378–386. Springer, 1994.
25. Z. Manna and A. Pnueli. A hierarchy of temporal properties. In *PODC'90*, pp. 377–410. ACM, 1990.
26. T. Michaud and M. Colange. Reactive synthesis from LTL specification with Spot. In *SYNT'18*, 2018. URL `http://www.lrde.epita.fr/dload/papers/michaud.18.synt.pdf`.
27. T. Michaud and A. Duret-Lutz. Practical stutter-invariance checks for ω-regular languages. In *SPIN'15*, *LNCS* 9232, pp. 84–101. Springer, 2015.
28. S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.
29. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Variations on parallel explicit model checking for generalized Büchi automata. *International Journal on Software Tools for Technology Transfer (STTT)*, 19(6):653–673, 2017.
30. F. Renkin, A. Duret-Lutz, and A. Pommellet. Practical "paritizing" of Emerson-Lei automata. In *ATVA'20*, *LNCS* 12302, pp. 127–143. Springer, 2020.
31. F. Renkin, P. Schlehuber, A. Duret-Lutz, and A. Pommellet. Improvements to `ltlsynt`. Presented at the SYNT'21 workshop, without proceedings, July 2021. URL `https://www.lrde.epita.fr/~adl/dl/adl/renkin.21.synt.pdf`.
32. S. Safra and M. Y. Vardi. On ω-automata and temporal logic. In *STOC'89*, pp. 127–137. ACM, 1989.
33. H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulæ into Büchi automata. In *CS&P'99*, pp. 251–262, 1999.