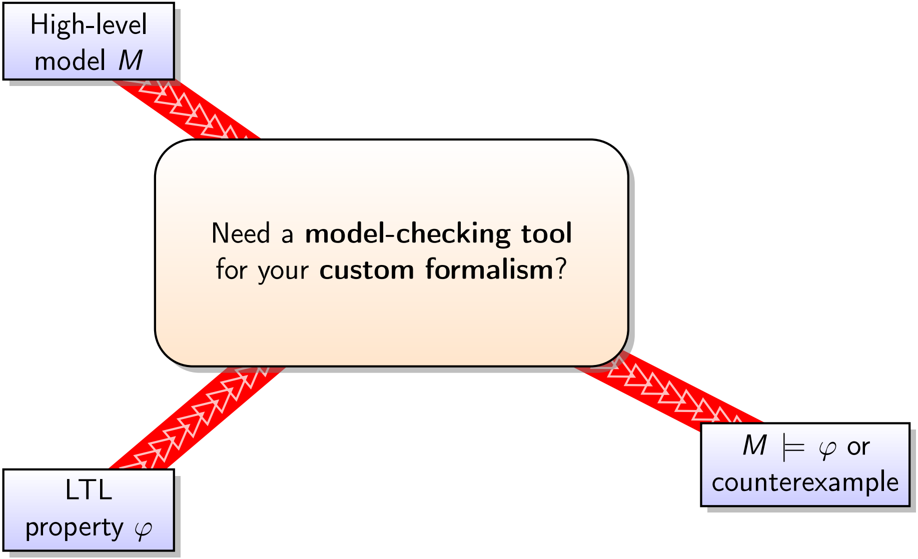
Spot 0.8.3: a C++ model-checking library

(Includes ads for the upcoming Spot 0.9)

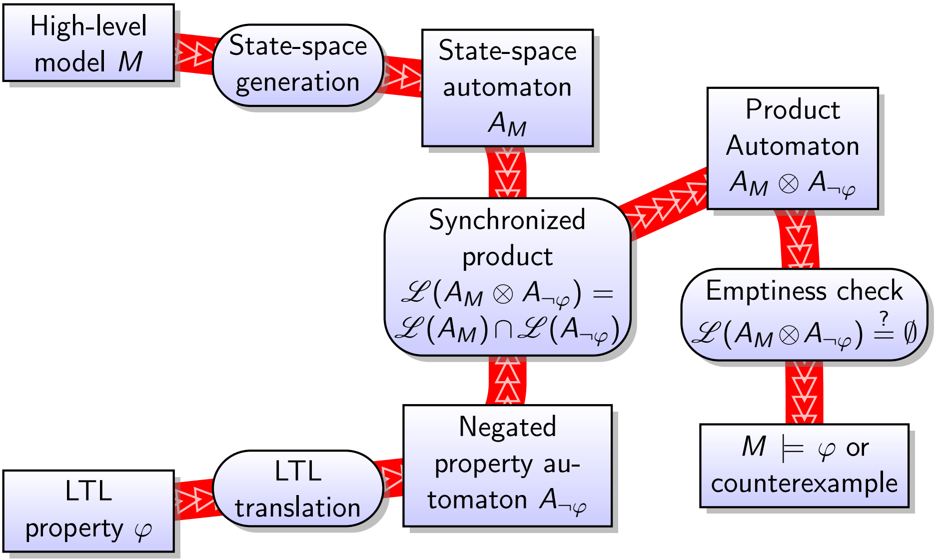
Alexandre Duret-Lutz
<adl@lrde.epita.fr>

MeFoSyLoMa 2012-03-23

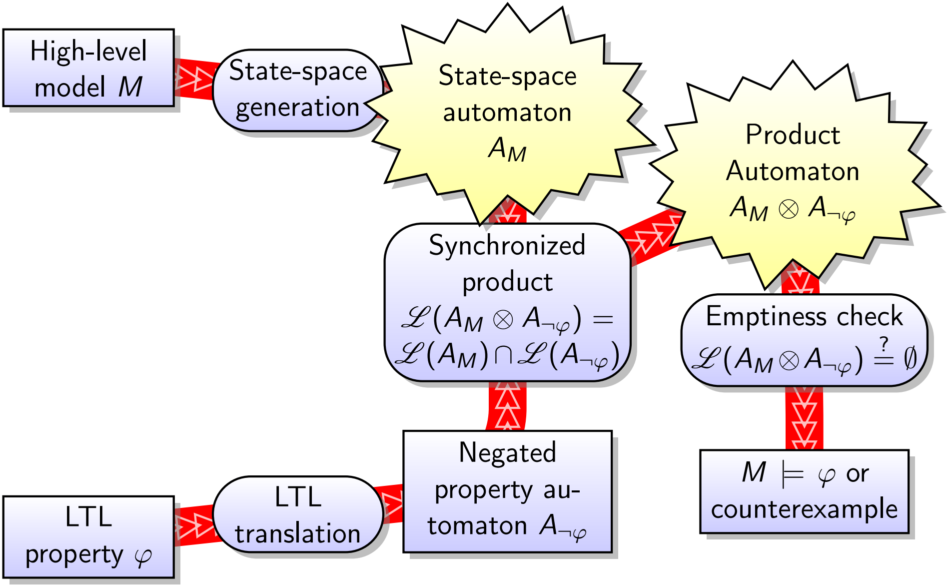
Context


High-level model M
Need a **model-checking tool** for your **custom formalism**?
LTL property φ
$M \models \varphi$ or counterexample

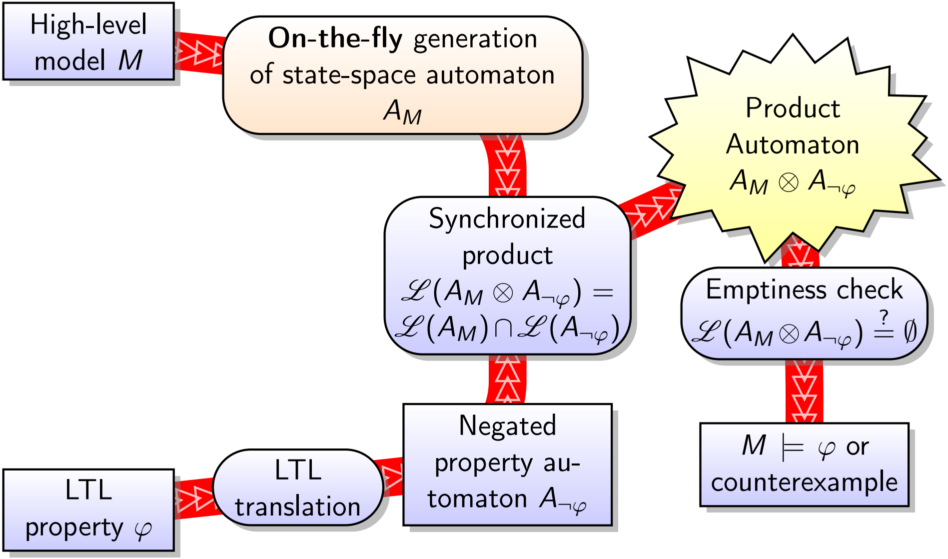
Automata-Theoretic LTL Model Checking



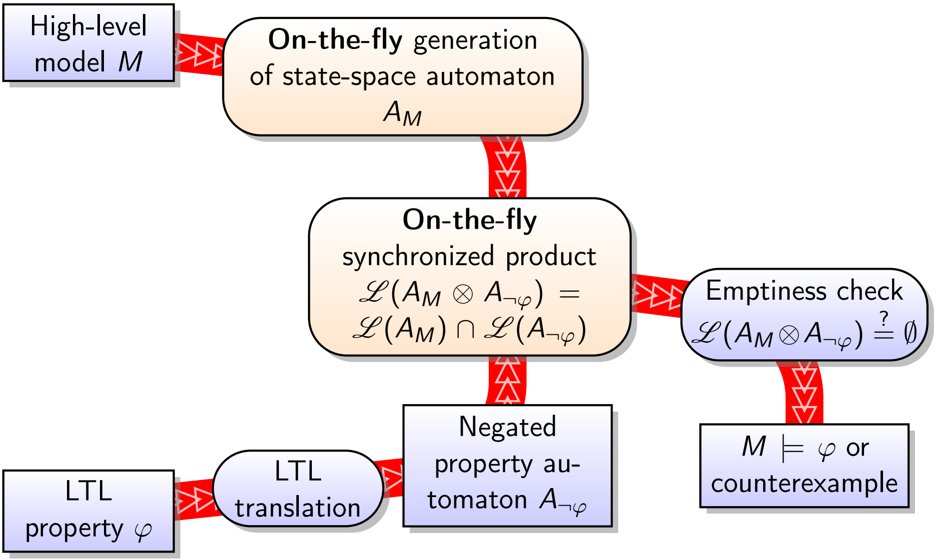
Automata-Theoretic LTL Model Checking



Automata-Theoretic LTL Model Checking



Automata-Theoretic LTL Model Checking



Some (active) Frameworks for LTL Model Checking

JavaPathFinder Java. Model checking of Java bytecode. LTL verification possible using Büchi automata.
http://babelfish.arc.nasa.gov/trac/jpf/

DiVinE C++. Büchi automata. Focus on parallel model checking.
http://divine.fi.muni.cz/

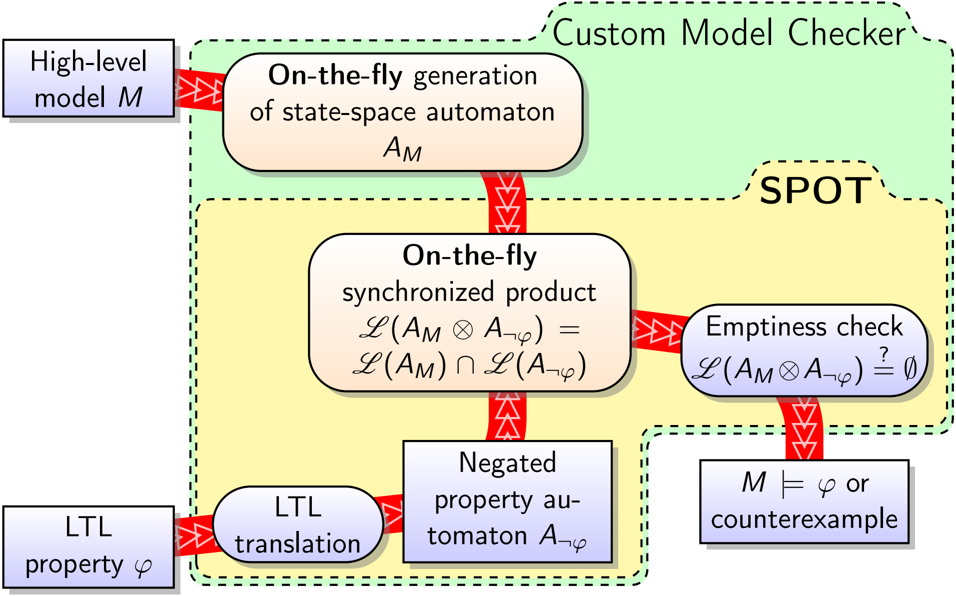
LTSmin C. Büchi automata. Various input formalisms are supported thanks to a simple state-space interface.
http://fmt.cs.utwente.nl/tools/ltsmin/

Spot C++. Transition-based Generalized Automata. An abstract state-space interface, but less input formalisms implemented.
http://spot.lip6.fr/

This talk...

1. What's in Spot?
2. How to use Spot to build a model checker?
3. Some examples of how it has been (ab)used.

Automata-Theoretic LTL Model Checking



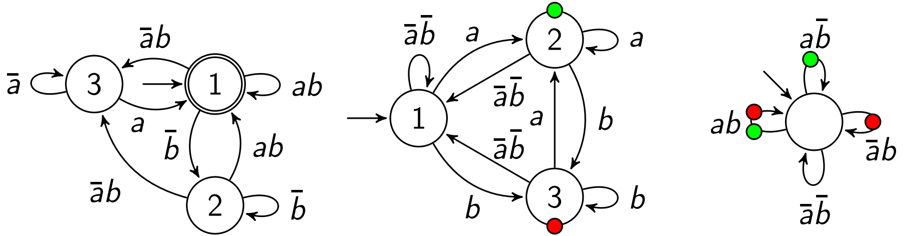
The Spot Library

http://spot.lip6.fr/

- A C++ model checking library started in 2003
  over 10 contributors
- Cornerstone: Transition-based Generalized Büchi Automata
- Features several algorithms to combine and build your own model checker
  - 4 algorithms to translate LTL into Büchi automata
  - 5 emptiness-check algorithms (with many variants)
  - 2 Büchi complementation algorithms
  - simplifications for formulas and automata
- We mostly use it to evaluate different algorithms and to develop new model-checking techniques
- Other people usually use Spot just to translate LTL formulas into Büchi Automata (thanks to Rozier & Vardi)

Different Kinds of Büchi Automata ($\mathbf{G}\,\mathbf{F}\,a \wedge \mathbf{G}\,\mathbf{F}\,b$)

Büchi Automaton	Generalized Büchi Automaton	Transition-based Generalized Büchi Automaton



- Same expressive power.
- Converting BA to GBA, or GBA to TGBA, is trivial.
- The opposite direction requires a degeneralization.
- (T)GBA occur naturally when translating LTL.

From LTL to TGBA

Translating LTL to Automata: Overview

Several steps:

1. Parsing the formula.
2. Simplification of the LTL formula.
3. Conversion from LTL to TGBA.
4. Simplification of the resulting TGBA.
5. Optional degeneralization.

Parsing LTL formulas

- Our LTL parser attempts to support various LTL syntaxes (Spin, SMV, Wring, ...).
- Missing: the prefix syntax used for instance by `scheck` or `lbt`.
- Error reporting can be done by the client, with provided location information.
- atomic propositions are either plain identifiers, or double-quoted strings. This can be used to embed language-specific operators. (E.g. `"CS[2] >= 3"` is considered as an atomic propositions.)
- ASTs are DAGs with sharing of common subexpressions.

Parsing and Atomic Properties

```
formula*
parse(const string& ltl_string,
      parse_error_list& error_list,
      environment& env = default_environment::instance(),
      bool debug = false);
```

- The environment is used by the parser to convert identifiers/strings to atomic propositions.
- The default environment accepts everything.
- Another could be used to **reject** propositions that are invalid in the model.

```
class environment {
public:
  virtual formula* require(const string& atom) = 0;
};
```

Simplification of LTL formulas

- Basic rewritings:
  $\mathbf{F}(f\,\mathbf{U}\,g) \equiv \mathbf{F}\,g$, $\mathbf{F}\,a \vee \mathbf{F}\,b \equiv \mathbf{F}(a \vee b)$, etc.
- Implication-based reductions:
  $f \vee g \equiv g$ if f implies g,
  $f\,\mathbf{U}\,g \equiv g$ if f implies g, etc.
  Implications detected syntactically or via language containment.
- Reductions for eventual and universal formulas:
  $\mathbf{F}\,e \equiv e$ if e is a pure eventuality,
  $\mathbf{G}\,u \equiv u$ if u is a purely universal.

Note: all the above rewritings have been seriously overhauled for the upcoming Spot 0.9.

Translation from LTL to TGBA

Four algorithms are implemented:

1. `ltl_to_tgba_fm()`: A tableau construction (FM'99). The most efficient of the lot to build explicit automata.
2. `ltl_to_tgba_lacim()`: A symbolic construction (LaCIM'00).
3. `eltl_to_tgba_lacim()`: A variant where all operators are specified as finite automata.
4. `ltl_to_tgba_taa()`: A construction via transition-based alternating automata (Tauriainen 2006).

J.-M. Couvreur. On-the-fly verification of temporal logic. In Proc. of FM'99, vol. 1708 of LNCS, pp. 253–271. Springer

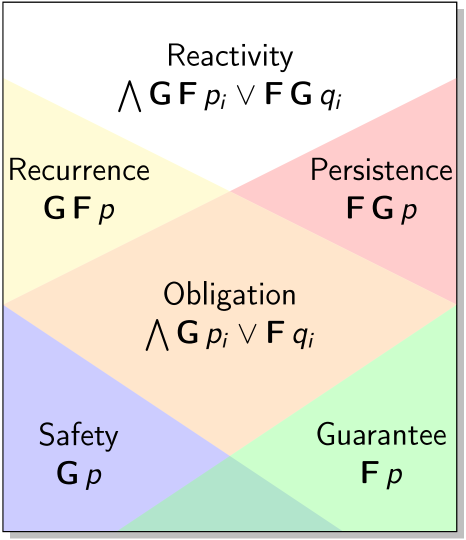
J.-M. Couvreur. Un point de vue symbolique sur la logique temporelle linéaire. In Actes du Colloque LaCIM 2000, vol. 27 of Publications du LaCIM, pp. 131–140. Université du Québec à Montréal, Aug. 2000

H. Tauriainen. Automata and Linear Temporal Logic: Translation with Transition-based Acceptance. PhD thesis, Helsinki University of Technology, Espoo, Finland, Sept. 2006

Simplifications of TGBA

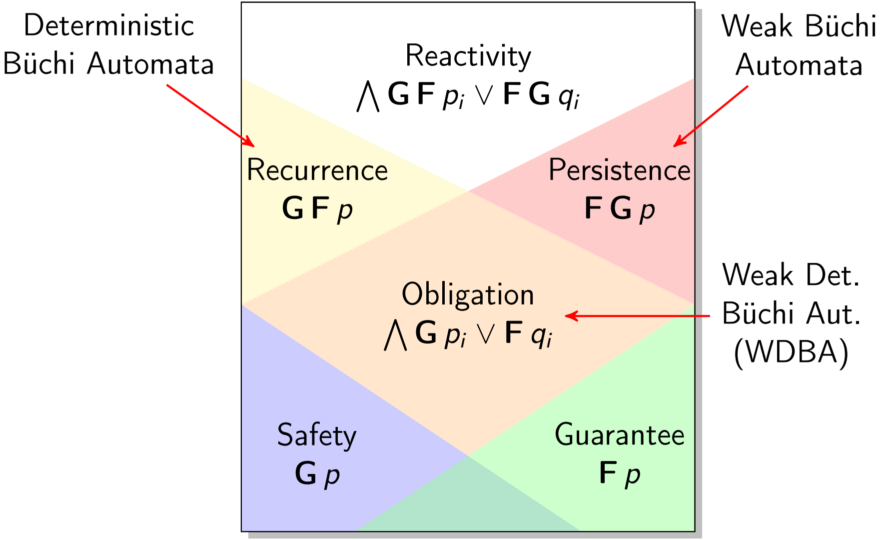
- `scc_filter()`: remove useless SCCs and superfluous acceptance conditions.
- `minimize_obligation()`: create a minimal Weak Deterministic Büchi Automaton for any obligation property.

Temporal Hierarchy



Z. Manna and A. Pnueli. A hierarchy of temporal properties. In Proc. of PODC'90, pp. 377–410. ACM

Temporal Hierarchy



I. Černá and R. Pelánek. Relating hierarchy of temporal properties to model checking. In Proc. of MFCS'03, vol. 2747 of LNCS, pp. 318–327. Springer

Simplifications of TGBA

- scc_filter(): remove useless SCCs and superfluous acceptance conditions.
- minimize_obligation(): create a minimal Weak Deterministic Büchi Automaton for any obligation property.

Simplifications of TGBA

- `scc_filter()`: remove useless SCCs and superfluous acceptance conditions.
- `minimize_obligation()`: create a minimal Weak Deterministic Büchi Automaton for any obligation property.

Present, but usually not used, `reduce_tgba_sim()`:

- direct simulation reduction (for BA only)
- delayed simulation reduction (broken)

(A new implementation of direct simulation, for TGBA, is cooking.)

Translation of Litterature Formulas

Cumulated sizes of automata for 188 formulas from the litterature

Products with a random state-space of 200 states

		$\Sigma\lvert A_{\neg\varphi}\rvert$		$\Sigma\lvert A_M \otimes A_{\neg\varphi}\rvert$	
		st.	tr.	st.	tr.
BA	Spin 6.1.0 (☠×6)	1 676	8 184	292 674	20 821 075
	LTL2BA 1.1	1 080	3 646	196 986	10 869 027
	Modella 1.5.9	1 392	4 570	256 912	10 535 585
	Spot 0.8.3	834	2 419	155 386	7 921 988
	Spot 0.8.3 det.	834	2 419	151 569	5 883 107
	Spot 0.8.3 WDBA	770	2 159	139 776	5 291 801
TGBA	Spot 0.8.3	757	2 085	144 388	7 219 086
	Spot 0.8.3 det.	757	2 085	140 972	5 503 874
	Spot 0.8.3 WDBA	704	1 879	130 977	4 964 962

☠ = 15min timeout

Produce more **det**erministic aut.

WDBA minimization when applicable

Rozier & Vardi's Scalability Experiment (1/2)

- An LTL formula C_n that can be encoded by a $n2^n$-state automaton.

K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In Proc. of SPIN'07, vol. 4595 of LNCS, pp. 149–167. Springer

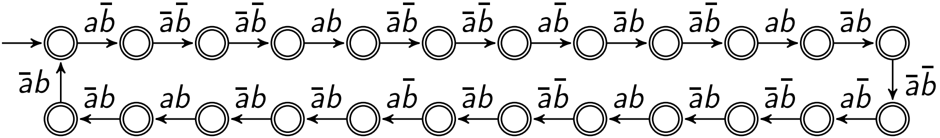
Rozier & Vardi's Scalability Experiment (1/2)

- An LTL formula C_n that can be encoded by a $n2^n$-state automaton.
- E.g. $C_3 = ((a \land (\mathbf{G}(a \rightarrow (\mathbf{X}(\neg a \land \mathbf{X}(\neg a \land \mathbf{X}\, a)))))) \land ((\neg b) \land \mathbf{X}(\neg b \land \mathbf{X}\, \neg b)) \land (\mathbf{G}((a \land \neg b) \rightarrow (\mathbf{X}((\mathbf{X}\,\mathbf{X}\, b) \land (((\neg a) \land (b \rightarrow \mathbf{X}\,\mathbf{X}\,\mathbf{X}\, b) \land ((\neg b) \rightarrow (\mathbf{X}\,\mathbf{X}\,\mathbf{X}\, \neg b)))\, \mathbf{U}\, a))))) \land (\mathbf{G}((a \land b) \rightarrow (\mathbf{X}((\mathbf{X}\,\mathbf{X}\, \neg b) \land ((b \land (\neg a) \land \mathbf{X}\,\mathbf{X}\,\mathbf{X}\, \neg b)\, \mathbf{U} (a \lor ((\neg a) \land (\neg b) \land (\mathbf{X}((\mathbf{X}\,\mathbf{X}\, b) \land (((\neg a) \land (b \rightarrow \mathbf{X}\,\mathbf{X}\,\mathbf{X}\, b) \land ((\neg b) \rightarrow \mathbf{X}\,\mathbf{X}\,\mathbf{X}\, \neg b))\, \mathbf{U}\, a)))))))))))$

K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In Proc. of SPIN'07, vol. 4595 of LNCS, pp. 149–167. Springer

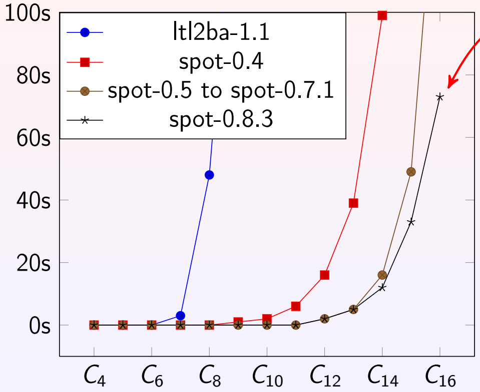
Rozier & Vardi's Scalability Experiment (1/2)

- An LTL formula C_n that can be encoded by a $n2^n$-state automaton.
- E.g. $C_3 = ((a \land (\mathbf{G}(a \rightarrow (\mathbf{X}(\neg a \land \mathbf{X}(\neg a \land \mathbf{X}\, a)))))) \land ((\neg b) \land \mathbf{X}(\neg b \land \mathbf{X}\,\neg b)) \land (\mathbf{G}((a \land \neg b) \rightarrow (\mathbf{X}((\mathbf{X}\,\mathbf{X}\, b) \land (((\neg a) \land (b \rightarrow \mathbf{X}\,\mathbf{X}\,\mathbf{X}\, b) \land ((\neg b) \rightarrow (\mathbf{X}\,\mathbf{X}\,\mathbf{X}\,\neg b)))\,\mathbf{U}\, a))))) \land (\mathbf{G}((a \land b) \rightarrow (\mathbf{X}((\mathbf{X}\,\mathbf{X}\,\neg b) \land ((b \land (\neg a) \land \mathbf{X}\,\mathbf{X}\,\mathbf{X}\,\neg b)\,\mathbf{U}(a \lor ((\neg a) \land (\neg b) \land (\mathbf{X}((\mathbf{X}\,\mathbf{X}\, b) \land (((\neg a) \land (b \rightarrow \mathbf{X}\,\mathbf{X}\,\mathbf{X}\, b) \land ((\neg b) \rightarrow \mathbf{X}\,\mathbf{X}\,\mathbf{X}\,\neg b))\,\mathbf{U}\, a)))))))))))$



K. Y. Rozier and M. Y. Vardi. LTL satisfiability checking. In Proc. of SPIN'07, vol. 4595 of LNCS, pp. 149–167. Springer

Rozier & Vardi's Scalability Experiment (2/2)



Time to translate C_n into BA

Other explicit translators are off the chart:

- Modella 1.5.9 took nearly 6 minutes to compute C_4 and ran out of memory on C_5.

- Spin 6.1.0 took more than 11 hours to translate C_1 into a 33-state automaton with 447 transitions (instead of 2 states and 2 transitions). Many transitions having unsatisfiable guards such as "`((!b) && (a) && (b))`".

All experiments done on an Intel Core2 Q9550 @2.83GHz with 8GB of RAM.

Using TGBA as an Interface

Design choice 1: using TGBA as an interface

- Any automaton in Spot is a TGBA.
  - This also includes classical Büchi Automata: a BA is a TGBA with a single acceptance condition, and in which outgoing transitions are either all accepting or not.
- TGBA is an abstract class (i.e., an interface).
  - It can be implemented in different ways. (Explicit graphs, symbolic representations...)
  - It can also be a front-end for algorithms that compute automata on-the-fly. (E.g., a product.)

Design choice 2: using BDD to label TGBA

Each transition in a TGBA has two labels:

1. a guard, which is a Boolean function represented as a BDD
2. a set of acceptance conditions, which is also represented as a BDD

The BDD library is BuDDy with a couple of extensions.

(Note: I regret all the above today.)

TGBA Interface

```
class tgba {
public:
 // -- main interface
 virtual state* get_init_state() const = 0;
 virtual tgba_succ_iterator*
          succ_iter(const state* s) const = 0;
 virtual bdd all_acceptance_conditions() const = 0;
 // -- miscellaneous methods
 virtual bdd_dict* get_dict() const = 0;
 virtual string format_state(const state* s) const = 0;
 virtual state* project_state(const state* s,
                              const tgba* t) const;
 virtual bdd neg_acceptance_conditions() const = 0;
 // -- optional method
 virtual string
   transition_annotation(const tgba_succ_iterator* t) const;
};
```

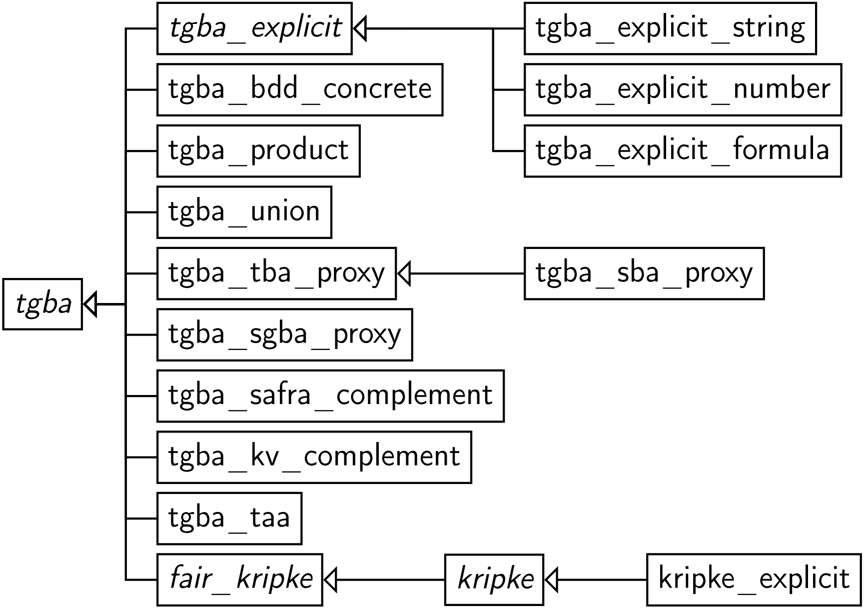
STATE Interface

```
class state {
public:
  virtual int compare(const state* other) const = 0;
  virtual size_t hash() const = 0;
  virtual state* clone() const = 0;
  virtual void destroy() const { delete this; }
protected:
  virtual ~state();
};
```

TGBA_SUCC_ITERATOR Interface

```
class tgba_succ_iterator {
public:
  // iteration
  virtual void first() = 0;
  virtual void next() = 0;
  virtual bool done() const = 0;
  // inspection
  virtual state* current_state() const = 0;
  virtual bdd current_condition() const = 0;
  virtual bdd current_acceptance_conditions() const = 0;
};
```

TGBA Hierarchy



KRIPKE Interface

```
class kripke_succ_iterator: public tgba_succ_iterator {
public:
  virtual void first() = 0;
  virtual void next() = 0;
  virtual bool done() const = 0;
  virtual state* current_state() const = 0;
  // other methods are predefined
};
class kripke {
public:
  virtual state* get_init_state() const = 0;
  virtual tgba_succ_iterator*
           succ_iter(const state* s) const = 0;
  virtual bdd state_condition(const state* s) const = 0;
  virtual string format_state(const state* s) const = 0;
  // and optionally, override transition_annotation
};
```

Emptiness Checks

Emptiness-Check classes

```
class emptiness_check_result { public:
  virtual tgba_run* accepting_run();
  virtual const unsigned_statistics* statistics() const;
};
class emptiness_check {
public:
  emptiness_check(const tgba* a, option_map o = option_map());
  virtual bool safe() const;
  virtual emptiness_check_result* check() = 0;
  virtual const unsigned_statistics* statistics() const;
};
class emptiness_check_instantiator {
public:
  static emptiness_check_instantiator*
    construct(const char* name, const char** err);
  emptiness_check* instantiate(const tgba* a) const;
};
```

Emptiness-Check variants

Algorithms for (T)BA:

- **CVWY90** With option for bit-state hashing.
- **GV04** No options.
- **SE05** With option for bit-state hashing.

Algorithms for TGBA:

- **Cou99** With options for various heuristics.
- **Tau03** No options.
- **Tau03_opt** With options for various heuristics. (Some implemented by Tauriainen himself.)

We do not have parallel emptiness checks (because our TGBA interface uses BDD everywhere, and Buddy is not thread-safe).

J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In Proc. of SPIN'05, vol. 3639 of LNCS, pp. 143–158. Springer

Plugging it together

A Minimal Model Checker (1/2)

```
using namespace spot;
std::string filename = "peterson3.kripke";
std::string notphi = "!G(P_0.wait -> F P_0.CS)"; // ¬φ
bdd_dict* dict = new bdd_dict();

// Parse a Kripke structure A_M
kripke_parse_error_list kpel;
tgba* model = kripke_parse(filename, kpel, dict);
if (format_kripke_parse_errors(std::cerr, filename, kpel))
  exit(1);

// Parse and simplify the LTL formula ¬φ
ltl::parse_error_list pel;
ltl::formula* tmpf = ltl::parse(notphi, pel);
if (ltl::format_parse_errors(std::cerr, notphi, pel))
  exit(1);
```

A Minimal Model Checker (2/2)

```
// Build A_¬φ
ltl::formula* f = ltl::reduce(tmpf); tmpf->destroy();
tgba* tmpaut = spot::ltl_to_tgba_fm(f, dict);
tgba* prop = spot::scc_filter(tmpaut); delete tmpaut;

// Build the product A_M ⊗ A_¬φ.
tgba* product = new tgba_product(model, prop);

// Search for a counterexample.
emptiness_check* ec = couvreur99(product);
emptiness_check_result* res = ec->check();

if (res) std::cout << "counterexample found" << std::endl;
else     std::cout << "property holds" << std::endl;
```

Actual Model-Checker Exemples

Replace the Kripke structure input by a `tgba` that computes the state-space on-the-fly.

Examples:

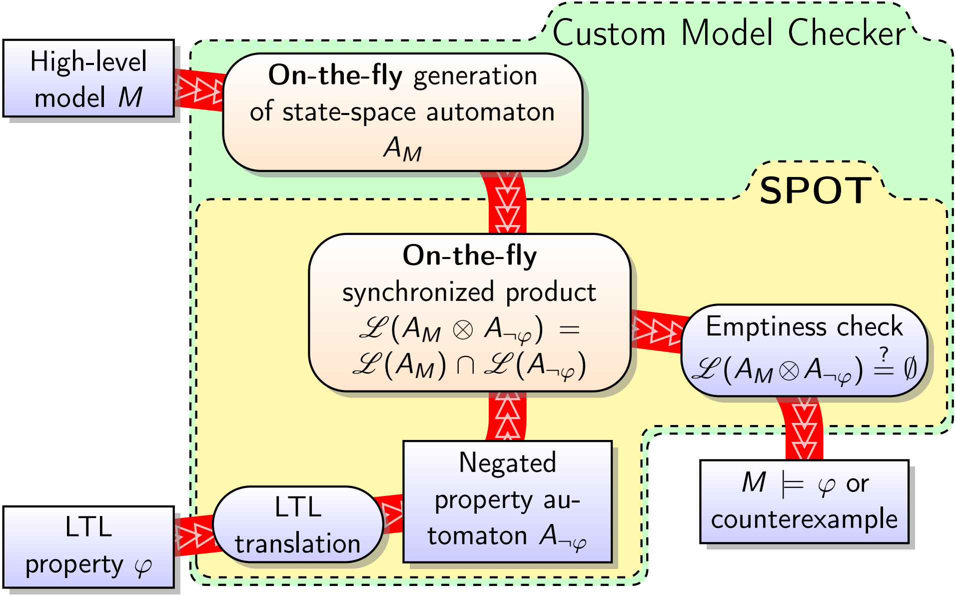
1. `ltlgspn`: for Colored Petri Nets (for GreatSPN)
   `iface/gspn/` in the Spot distribution.
2. `checkpn`: for Petri Nets.
   `git clone git://git.lrde.epita.fr/checkpn`
3. `MC-SOG`: for Petri Nets, using Symbolic Observation Graphs.
4. `dve2check`: for DiVinE models with an LTSmin interface.
   `iface/dve2/` in the Spot distribution.
5. `its-ltl`: for ITS models.
   `http://move.lip6.fr/software/DDD/download.html`
6. `Neco` (IBISC): for Petri Nets
   `http://code.google.com/p/neco-net-compiler/`

Other usages

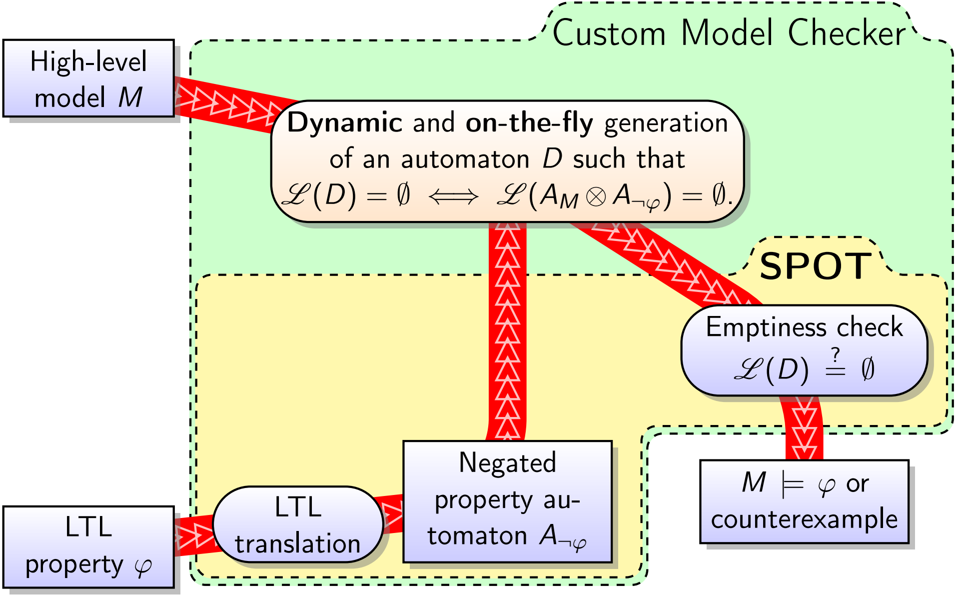
Uses of Spot

- As an LTL translator, for benchmarking.
  Sebastiani et al. (CAV'05), Rozier & Vardi (Spin'07),
  Cichoń et al. (DEPCOS'09), Babiak at el. (TACAS'12), ...
- As an LTL translator, for building upon.
  Staat & Heimdahl (ICFEM'08), Tabakov & Vardi (RV'10),
  Ehlers (SAT'10), Cabalar & Diéguez (LPNMR'11)
- To evaluate new emptiness checks or heuristics.
  Couvreur et al. (SPIN'07), Taurainen (PhD 2006),
  Li et al. (IMSCCS'06), Rebiha & Ciampaglia (ISDA'07)
- To evaluate new model-checking approaches.
  Baarir & Duret-Lutz (ACSD'07), Duret-Lutz et al. (ATVA'11),
  Ben Salem et al. (SUMo'11)
- To help check automata equivalence (using Safra) in a tool for proving inductive theorems.
  Brotherston et al. (submitted to IJCAR'12)

Automata-Theoretic LTL Model Checking



Automata-Theoretic LTL Model Checking



What's coming

What's coming?

Work started:

1. PSL support. (Already working, needs some polishing.)
2. Simulation-based reductions for TGBA.
3. Speeds improvements in degeneralization and some automata simplifications.
4. Testing Automata and variants. (Part of Ala Eddine Ben Salem's PhD work.)
5. Emptiness-check improvements based on the temporal hierarchy. (Part of Etienne Renault's PhD work.)

Needed but not started:

1. Partial order reductions
2. Fixing the interface to make it thread-safe
   Maybe replacing BuDDy by JINC
   (`http://www.jossowski.de/projects/jinc/jinc.html`)

Contact

Project's Web Page

`http://spot.lip6.fr/`

On-line Translator

`http://spot.lip6.fr/ltl2tgba.html`

Mailing List

`spot@lrde.epita.fr`

Git Repository

`git://git.lrde.epita.fr/spot`