Acacia+ v2.3 - User manual

Aaron Bohy

May 10, 2019

Contents

1 What is Acacia+?

Acacia+ is a tool for synthesis. It is an open source implementation in Python/C of some of the theoretical results obtained by our research team. It is the successor of Acacia, written in Perl, but Acacia+ is more scalable, flexible and modular. It uses the public library AaPAL [1] for the manipulation of antichains and pseudo-antichains.

Given an LTL formula and a partition of the signals into output signals controlled by the system and input signals controlled by the environment, Acacia+ checks for realizability of the formula. If the formula is realizable, it synthesizes a Moore machine such that no matter what the environment sends, the executions of the machine all satisfy the LTL formula. Acacia+ also supports compositional synthesis from LTL specifications that consist of conjunctions of sub-specifications. For monolithic (i.e. non-compositional) unrealizable specifications, Acacia+ can synthesize a counter strategy for the environment.

Acacia+ implements the antichain-based incremental procedure proposed in [9] for LTL realizability and synthesis, with reduction to two-player safety games. It also implements optimizations detailed in [10].

Acacia+ v.2 proposes an extension to synthesis from LTL specifications with secondary mean-payoff objectives, called $\mathsf{LTL}_{\mathsf{MP}}$ synthesis. Given an LTL formula, a partition of the signals into output and input signals, a weight function that associates a value to signals, and a threshold value, it checks for, and in positive case outputs, a strategy for the system to (1) satisfy the LTL formula and (2) ensure a value greater than or equal to the given threshold, against any strategy of the environment, where the value of an infinite word is the mean-payoff of the weights of its letters.

The procedure for $\mathsf{LTL}_{\mathsf{MP}}$ synthesis is detailed in [4, 3]. As for LTL synthesis, this is an antichain-based incremental procedure with reduction to safety games.

On the top of $\mathsf{LTL}_{\mathsf{MP}}$ synthesis, Acacia+ also supports the synthesis of worst-case winning strategies, i.e. strategies that ensure the $\mathsf{LTL}_{\mathsf{MP}}$ specification against all behaviors of the environment, with good expected performance against a stochastic behavior of the environment. Details of this functionality are given in [5, 6].

To summarize, Acacia+ has the following features:

- Realizability check of the controller specification.
- Compositional realizability from conjunctions of sub-specifications.
- Synthesis of a winning strategy for the controller if the specification is realizable (for LTL and $\mathsf{LTL}_{\mathsf{MP}}$).
- Synthesis of a winning strategy for the environment if the specification is monolithic and unrealizable (only for LTL).
- Synthesis of a worst-case winning strategy with good expected performance against a stochastic environment (only for $\mathsf{LTL}_{\mathsf{MP}}$).

2 Content of the archive

In addition to Acacia+, this archive includes:

- LTL2BA 1.1, an LTL formula to Büchi automata translator, written by Dennis Oddoux and Paul Gastin (http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/).
- a modified version of Lily 1.0.2, a linear logic synthesizer by Barbara Jobstmann and Roderick Bloem, called Lily-AC.
- seven benchmarks of examples:
  - the test suite included in Lily.
  - the generalized buffer controller from the IBM RuleBase tutorial (http://research.ibm.com/haifa/projects/verification/rb_homepage).
  - the load balancing system provided with Unbeast, a symbolic bounded synthesis tool written by Rudiger Ehlers.
  - a benchmark of examples for the translation from LTL to equivalent deterministic Büchi automaton.
  - a benchmark of examples for the translation from LTL to equivalent deterministic parity automaton.
  - the shared resource arbiter (SRA), a benchmark of examples for synthesis from LTL with mean-payoff objectives.
  - the stochastic shared resource arbiter, an extension of SRA with 2 clients for the problem of worst-case synthesis with optimization of the expected case. Those examples are designed for four different probability distributions.

3 Installing Acacia+

Acacia+ can be installed on Linux and Mac OS X, for both 32- and 64-bit variants of these operating systems. Acacia+ requires several programs to get installed and run.

3.1 Installing the dependencies

The following dependencies need to be installed.

3.1.1 Glib2

Glib2 is a public library for the manipulation of data structures such as hash tables or lists in C. For more information, see http://www.gtk.org/.

3.1.2 Pygraph 1.8.0 or higher

Python-graph is a library for working with graphs in Python. It provides a suitable data structure for representing graphs and a whole set of important algorithms. The python-graph library is available at `http://code.google.com/p/python-graph/`. The dot module is not required to run Acacia+. The python-graph library can be installed using easy_install as follows:

```
$ easy_install python-graph-core
```

3.1.3 PyGraphviz 1.1 or higher

PyGraphviz is a Python interface to the Graphviz graph layout and visualization package. Acacia+ uses PyGraphviz to draw strategies in DOT and PNG. PyGraphviz is available at `http://networkx.lanl.gov/pygraphviz/`. It can be installed using easy_install as follows:

```
$ easy_install pygraphviz
```

3.1.4 Numpy

Numpy is a package for scientific computing (e.g. linear algebra) with Python. Numpy is available at `http://www.numpy.org/` and can be installed using easy_install as follows:

```
$ easy_install numpy
```

3.2 Installing Acacia+

Once all dependencies have been installed, Acacia+ can be installed by running, from the main repository,

```
$ make install
```

This command compiles the C library (see ./lib/), and the tool LTL2BA [11] (see ./tools/ltl2ba-1.1/) used by Acacia+ for Büchi automata construction from LTL formula, both contained in the Acacia+ archive.

3.3 Optional tools

The following tools are only required if you want to use them, instead of LTL2BA, for automata construction.

3.3.1 Wring

Acacia+ may use the Wring [13] module included in the tool Lily [12]. It is provided in the Acacia+ archive (see ./tools/Lily-AC/). Lily requires Perl 5.8.8 or higher. You only need to set the Perl library path and the search path correctly, e.g.

```
$ export LILY=/usr/local/AcaciaPlus-v2.3/Lily-AC/
$ export PERL5LIB=${LILY}:${PERL5LIB}
$ export PATH=${LILY}:${PATH}
```

For permanent use, add the preceding lines in your .bashrc or .profile.

3.3.2 LTL3BA

LTL3BA [2] is a translator of LTL formula to Büchi automata based on LTL2BA. The archive can be downloaded at `http://sourceforge.net/projects/ltl3ba/`. Install it by following instructions contained in the README file of the archive and set the LTL3BA_PATH static variable in file constants.py.
Note: if you are working on Mac OS X, you might need to edit the LTL3BA Makefile and remove the option -static from the compilation line.

3.3.3 Spot

Spot [8] is an object-oriented model checking library written in C++ that provides a translation from LTL formula to Büchi automata. The archive can be downloaded at `http://spot.lip6.fr/wiki/`. Install it by following instructions contained in the INSTALL file of the archive and set the SPOT_PATH static variable in file constants.py.

3.4 Known issue

You might face some problems while compiling the C library (Section 3.2) or while executing Acacia+ (ctypes error when binding Python with the C library). This is an architecture issue that might occur if your version of glib2 is not the default architecture of your machine. In that case, you should edit ./lib/makefile to specify the architecture of your version of glib2 in the ARCHFLAG field (e.g. for Mac OS X: ARCHFLAG=-arch i386). You can then enforce python to execute in the desired architecture (to prevent ctypes error while binding) by running

```
$ arch -my_architecture python acacia_plus.py
```

4 Syntax of formula and signals partition

4.1 LTL formula

An LTL formula may contain atomic signals, boolean operators, temporal operators, and parentheses. Acacia+ accepts both the Wring and LTL2BA/LTL3BA input formats.

4.1.1 Signals and literals

A signal is any lowercase string (optional: directly followed by **=1**). The negation of a signal is any lowercase string directly followed by **=0**. One can also use the negation operator in front of a signal.

4.1.2 Boolean operators

- Negation: !
- Implication: $->$
- Equivalence: $<->$
- And: * or &&
- Or: + or $||$

4.1.3 Temporal operators

- Always: **G**
- Eventually: **F**
- Until: **U**
- Release: **R**
- Next: **X**

4.1.4 Comments

Any text preceded by # is ignored.

4.1.5 Assumptions

The keyword **assume** preceding a formula can be used to specify assumptions. A global implication is a list of assumptions (the left-hand side of the implication) followed by a list of formulas (the right-hand side of the implication), each list being interpreted as a conjunction. For instance,

$$\text{assume } \phi_1\text{; assume } \phi_2\text{; } \phi_3\text{; } \phi_4\text{;}$$

is equivalent to

$$(\phi_1 * \phi_2) -> (\phi_3 * \phi_4)$$

4.2 Compositional specifications

Acacia+ supports compositional synthesis (see [9]) from compositional specifications. A list of sub-specifications is interpreted as the conjunction of them.

Sub-specifications must begin with the keyword **[spec_unit name]** where name is the name of the specification. The list of sub-specifications must be followed by the line **group_order = u;** where u follows the next grammar of well-parenthesized expressions:

$$u ::= (u) \mid uu \mid \text{name}$$

where name is a name of a sub-specification.

The parenthesizing drives Acacia+ on the order in which the sub-specifications must be combined for the compositional synthesis of the whole specification. For instance, assume we have three specifications named u_1, u_2 and u_3, when specifying $u_1\ (u_2\ u_3)$ for parenthesizing, the synthesis engine works as follows: it first solves u_1, u_2 and u_3 locally and gets three sets of solutions S_1, S_2 and S_3. Then it combines the solutions S_2 and S_3 into S_{23}, and then combines S_1 with S_{23}.

The keywords **BINARY** and **FLAT** can be used instead of giving a specific parenthesizing. **BINARY** means that the sub-specifications are grouped two by two (to obtain a binary composition tree) whereas **FLAT** means a parenthesizing $(u_1\ u_2\ \dots\ u_n)$ (to obtain a composition tree of height 1).

4.3 Example of .ltl file

Here is an example the content of a .ltl file. It compositionally defines the LTL formula $\phi = \phi_1 \wedge \phi_2$ with $\phi_1 = \Box\Diamond a \rightarrow \Box\Diamond b$ and $\phi_2 = \Box\Diamond c \rightarrow \Box\Diamond d$. The parenthesizing is **FLAT**, which means that Acacia+ will solve ϕ_1 and ϕ_2 independently, and the compose the computed solutions.

```
# This is spec 1
[spec_unit u1]
assume G(F(a=1));
G(F(b=1));

# This is spec 2
[spec_unit u2]
assume G(F(c=1));
G(F(d=1));

# Parenthesizing
group_order = (u1 u2);
```

4.4 Partition of signals

The set P of signals must be partitioned into the set I of input signals (for the environment) and the O set of output signals (for the system). The .part file contains information about the partition of signals. The input (resp. output) signals are written, separated by a white space, next to keyword **.inputs** (resp. **.outputs**).

4.5 Optional mean-payoff objective

Acacia+ supports synthesis from LTL specifications with secondary (optional) mean-payoff objectives, or $\mathsf{LTL}_{\mathsf{MP}}$ synthesis (see [4, 3]). In the case of $\mathsf{LTL}_{\mathsf{MP}}$ synthesis, the mean-payoff parameters must be specified in the .part file, as follows.

The weight function associates an integers vector, written $(w_1, w_2, \dots, w_k)$, to each literal of P. Keywords **.values_i**, **.values_!i**, **.values_o** and **.values_!o** are respectively for values associated to input signals, the negation of input signals, output signals and the negation of output signals. The ordering of the vectors is the same as for associated signals.

The threshold vector is mandatory if at least one literal has an associated not null weight. It has to be written next to keyword **.nu**.

The minimum (resp. maximum) vector of credits considered is written next to keyword **.c_start** (resp. **.c_bound**). Moreover, the incremental step vector is written next to keyword **.c_step**.

4.6 Optional probability distribution

Acacia+ supports, on the top of $\mathsf{LTL}_{\mathsf{MP}}$ synthesis, the synthesis of a worst-case winning strategy which behaves well against an environment playing according to a probability distribution (see [5, 6]). In this case, the probability distribution has to be specified in the .part file, as follows.

The optional probability distribution on the set of input signals is written next to keyword **.prob_distr**. Probabilities must be values in the interval $]0, 1[$. If the probability distribution is specified, a probability must be associated with each input signal. This corresponds to the probability of this signal to be asserted at each time unit. The ordering of the values is the same as for associated signals. By default, the probability distribution is uniform, i.e. each signal has probability $\frac{1}{2}$ to be asserted at each time unit.
Note: this option is only available when synthesizing from LTL with a one-dimensional mean-payoff objective.

4.7 Examples of .part files

Here is an example the content of a .part file for a two-dimensional mean-payoff objective. It defines a weight function w such that $w(b) = (-1, 0)$, $w(d) = (0, -1)$, and $w(l) = (0, 0)$ for all other literals. The threshold ν is equal to $(-0.2, -0.2)$. The procedure implemented in Acacia+ reduces the $\mathsf{LTL}_{\mathsf{MP}}$ synthesis problem to the problem of solving energy safety games. The algorithms for solving those energy safety games are incremental on the maximum credit considered (see [4] for more details). This .part file indicates that the incremental algorithms must consider maximum credits values $C \in \mathbb{N}^2$ equal to $(0, 0), (5, 5), (10, 10), (15, 15)$ and $(20, 20)$, and stop as soon as a winning strategy is found for some $C \in \mathbb{N}^2$.

```
.inputs a c
.values_i (0,0) (0,0)
.values_!i (0,0) (0,0)
.outputs b d
.values_o (-1,0) (0,-1)
.values_!o (0,0) (0,0)
.nu (-0.2,-0.2)
.c_start (0,0)
.c_bound (20,20)
.c_step (5,5)
```

Here is an example the content of a .part file for a one-dimensional mean-payoff objective with a probability distribution on the set of input signals. This file indicates that the environment has probability $\frac{3}{5}$ (resp. $\frac{1}{5}$) to assert $r1$ (resp. $r2$) at each time unit, and thus probability $\frac{2}{5}$ (resp. $\frac{4}{5}$) not to assert $r1$ (resp. $r2$). The underlying probability distribution $\pi : \Sigma_I \rightarrow]0,1[$ on the actions of the environment is defined such that $\pi(r_1 r_2) = \frac{3}{5} \cdot \frac{1}{5} = \frac{3}{25}$, $\pi(\neg r_1 r_2) = \frac{2}{5} \cdot \frac{1}{5} = \frac{2}{25}$, $\pi(r_1 \neg r_2) = \frac{3}{5} \cdot \frac{4}{5} = \frac{12}{25}$ and $\pi(\neg r_1 \neg r_2) = \frac{2}{5} \cdot \frac{4}{5} = \frac{8}{25}$.

```
.inputs r1 r2
.outputs g1 g2 w1 w2
.values_o 0 0 -1 -2
.nu -1.4
.c_start 0
.c_bound 13
.c_step 1
.prob_distr 0.6 0.2
```

5 Running Acacia+

5.1 Quick start

Acacia+ needs two mandatory arguments to execute: a file containing an LTL specification (.ltl extension) and a file containing a partition of the signals (.part extension). Examples out of seven benchmarks are contained in the Acacia+ archive (see Section 2). Acacia+ offers many execution parameters. To execute Acacia+ with the default configuration on the 13th of the benchmark demo-lily, run

```
& python acacia_plus.py −−ltl examples/demo-lily/demo-v13.ltl −−part
examples/demo-lily/demo-v13.part
```

To display the helper, run

```
& python acacia_plus.py −−help
```

5.2 Execution parameters

In this section, we describe each execution parameter.

5.2.1 -h or --help

Shows the helper and exit Acacia+.

5.2.2 -L or --ltl *filename*

Specifies a .ltl file containing an LTL specification (see Sections 4.1, 4.2 for syntax).

5.2.3 -P or --part *filename*

Specifies a .part file containing the partition of signals and optional mean-payoff and probability distribution (see Sections 4.4, 4.5, 4.6 for syntax).

5.2.4 -n or --nbw *construction_method*

Specifies the method used to construct universal co-Büchi automata from LTL specifications. Two methods are available: monolithic (**MONO**, by default) and compositional (**COMP**). In the first case, a single automaton $\mathcal{A}_\phi$ is constructed from the given LTL specification ϕ. For the latter case, the specification must be given as a conjunction $\phi_1 \wedge \cdots \wedge \phi_n$ of sub-specifications (see Section 4.2 for syntax). In this case, an automaton $\mathcal{A}_{\phi_i}$ is constructed for every ϕ_i.

5.2.5 -s or --syn *synthesis_method*

Specifies the method used for synthesis. Two methods are available: monolithic (**MONO**, by default) and compositional (**COMP**, only if the specification ϕ is a conjunction of ϕ_i's). When the automaton construction method is compositional and the synthesis method is monolithic, the latter starts with the union of all automata $\mathcal{A}_{\phi_i}$. If both the construction and synthesis methods are compositional, each automaton $\mathcal{A}_{\phi_i}$ is solved independently, and the computed results are composed according to the given parenthesizing (see Section 4.2 for syntax). The compositional approach allows to solve much larger specifications. The parenthesizing may also influence the performance.

5.2.6 -a or --algo *sg_algorithm*

Specifies the algorithm used for solving the underlying safety game. Two algorithms are available: the forward (**FORWARD**, by default) and the backward (**BACKWARD**) algorithms. The forward algorithm is proposed in [9]. This algorithm is a variant of the OTFUR algorithm of [7]. The backward algorithm is a classical fixpoint algorithm.

Compared to the backward algorithm, the forward algorithm has the advantage of computing the winning states in the safety game that are reachable from the initial state. Nevertheless, it computes a single winning strategy if it exists, whereas the backward algorithm computes a fixpoint from which we can easily enumerate the set of all winning strategies in the safety game. In general, the forward algorithm is faster than the backward one.

In the case of a compositional synthesis with forward option enabled, each intermediate safety game is solved backward and the game is solved forward.

5.2.7 -k or --kstart *starting_k_value*

Specifies the starting value of K for the incremental algorithm. Recall that K is the maximum number of final states that are seen by words accepted in a universal K-co-Büchi automaton.

5.2.8 -K or --kbound *largest_k_value*

Specifies the maximal value of K for the incremental algorithm.

5.2.9 -y or --kstep *k_step*

Specifies the incremental step on K values.

5.2.10 -t or --tool *toolname*

Specifies the tool used by Acacia+ to convert the LTL specification into an equivalent universal co-Büchi automaton. Four tools are available: LTL2BA (by default), LTL3BA, Wring and Spot (see Section 3.3 for installation instructions).

5.2.11 -p or --player *starting_player*

Specifies the starting player in the realizability game: the system (**2**, by default) or the environment (**1**).

5.2.12 -C or --check *tocheck*

Specifies if Acacia+ has to check realizability of the LTL specification (**REAL**, by default), unrealizability (**UNREAL**) or both in parallel (**BOTH**).
Note: Unrealizability checking is only available for monolithic formulas, without mean-payoff objective.

5.2.13 -v or --verb *verbosity*

Specifies the verbosity level: no text (**0**), execution recap only (**1**), digest display (**2**, by default) or detailed display (**3**).

5.2.14 -O or −−output *output*

Specifies what Acacia+ needs to synthesize from the antichain of winning states in the safety game. The user can ask for the synthesis of one arbitrary winning strategy (**ARBITRARY**, by default); the set of maximal winning strategies, i.e. by only considering the antichain of winning states, not its closure (**MAX**); the set of all winning strategies (**ALL**), i.e. by considering the whole closure of the antichain; the optimal strategy against an environment playing according to a given probability distribution (**OPTIMAL**, see Section 4.6).

5.2.15 -o or −−opt *opt*

Concerns two optimizations of the implemented procedure:

- Detect bounded/unbounded states (on the universal co-Büchi automaton):
  Computes an under-approximation of the set of states which cannot carry a counter value at least k, i.e. which cannot be reached by a path containing more than k final states. Those states are called bounded states. The optimization reduces the state space of the underlying safety game.

- Detect k-surely losing states (on the turn-based universal co-Büchi automaton):
  Tries to reduce the size of the turn-based automaton by computing k-surely losing states and replacing them by a single trap state. A state is k-surely losing if the environment has a strategy from this state to prevent the system to win.

Both optimizations are enabled by default. Set to **NONE** to disable both optimizations, to **1** to enable optimization 1 only, and to **2** to enable optimization 2 only.

5.2.16 -c or −−crit *crit_opt*

Concerns the critical signals optimization. This optimization makes the backward fixpoint algorithm more efficient by limiting the input signals to critical ones, at each step of the fixpoint computation (see [10] for details). The optimization is enabled by default. Set to **OFF** to disable.

5.3 Execution results

The output of the execution indicates if the input specification ϕ is realizable, and in this case proposes at least one winning strategy for the system. At least one winning strategy for the environment can be returned when ϕ is unrealizable (only in case of a monolithic automaton construction, without mean-payoff objective).

The synthesized strategies are represented by transition systems where transitions are labeled with symbols $(o \cup i)$ or $(i \cup o)$ such that o is an action of the system and i is an action of the environment. The action of the starting player is always leading on transitions labels. Let M be a transition system representing strategies for the system (resp. the environment). If the transition system only represents one strategy, the outgoing transitions of each state share the same o (resp. i). The transition system has to be interpreted as follows: in some state q, the system (resp. the environment) asserts one of the actions available on the outgoing transitions of q and the next state is determined by the actions of the environment (resp. the system).

When the output transition system is small enough (< 20 states), it is also drawn in PNG and DOT using PyGraphviz.

6 Web interface

For convenience, Acacia$+$ can be used directly online via a user-friendly web interface, where a number of examples and benchmarks have already been pre-loaded with adequate parameters configuration. The URL of the web interface is `http://lit2.ulb.ac.be/acaciaplus/onlinetest/`.

References

[1] AaPAL website. `http://lit2.ulb.ac.be/aapal/`.

[2] T. Babiak, F. Blahoudek, M. Kretínský, and J. Strejcek. Effective translation of LTL to deterministic Rabin automata: Beyond the (F, G)-fragment. In D. V. Hung and M. Ogawa, editors, *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2013.

[3] A. Bohy, V. Bruyère, E. Filiot, and J.-F. Raskin. Synthesis from LTL specifications with mean-payoff objectives. *CoRR*, abs/1210.3539, 2012.

[4] A. Bohy, V. Bruyère, E. Filiot, and J.-F. Raskin. Synthesis from LTL specifications with mean-payoff objectives. In N. Piterman and S. A. Smolka, editors, *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 169–184. Springer, 2013.

[5] A. Bohy, V. Bruyère, and J.-F. Raskin. Symblicit algorithms for optimal strategy synthesis in monotonic Markov decision processes. In *SYNT*, *EPTCS*, 2014. 17 pages.

[6] A. Bohy, V. Bruyère, and J.-F. Raskin. Symblicit algorithms for optimal strategy synthesis in monotonic Markov decision processes. *CoRR*, abs/1402.1076, 2014.

[7] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In M. Abadi and L. de Alfaro, editors, *CONCUR*, volume 3653 of *Lecture Notes in Computer Science*, pages 66–80. Springer, 2005.

[8] A. Duret-Lutz and D. Poitrenaud. SPOT: An extensible model checking library using transition-based generalized Büchi automata. In D. DeGroot, P. G. Harrison, H. A. G. Wijshoff, and Z. Segall, editors, *MASCOTS*, pages 76–83. IEEE Computer Society, 2004.

[9] E. Filiot, N. Jin, and J.-F. Raskin. Antichains and compositional algorithms for LTL synthesis. *Formal Methods in System Design*, 39(3):261–296, 2011.

[10] E. Filiot, N. Jin, and J.-F. Raskin. Exploiting structure in ltl synthesis. *STTT*, 15(5-6):541–561, 2013.

[11] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65. Springer, 2001.

[12] B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD*, pages 117–124. IEEE Computer Society, 2006.

[13] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In E. A. Emerson and A. P. Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.