lbtt

LTL-to-Büchi Translator Testbench
30 August 2005, lbtt Versions 1.2.x

Heikki Tauriainen

Table of Contents

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary.

To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice

that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) yyyy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307,
USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may

1 Overview

`lbtt` is a tool for testing programs that translate formulas expressed in propositional linear temporal logic (*LTL*) into Büchi automata. These finite-state automata over infinite words are used e.g. in automata-theoretic model checking [VW86, Var96], where they can help in detecting errors in the specifications of finite-state hardware or software systems. Usually the model checking procedure involves first composing an automaton with a formal model of a given system, and the result of the composition reveals whether any computation path of the system violates some property that the automaton represents. (For an introduction to model checking techniques in general, see, for example, [CGP99].)

The property to be model checked can be specified as an LTL formula, and the Büchi automaton used for model checking is obtained automatically from the formula with a translation algorithm. (For descriptions and optimization techniques for such algorithms, see the references, for example, [VW86, Isl94, GPVW95, Cou99, DGV99, Ete99, SB00, EH00, EWS01, GO01, Gei01, Sch01, Wol01, Ete02, GL02, GSB02, Thi02, Fri03, GO03, Lat03, ST03].) In practice, ensuring the correctness of the implementation of such a translation algorithm is crucial to guarantee the soundness of the implementation of a model checking procedure.

The goal of `lbtt` is to assist implementing LTL-to-Büchi translation algorithms correctly by providing an automated testing environment for LTL-to-Büchi translators. Testing consists of running LTL-to-Büchi translators on randomly generated (or user-specified) LTL formulas as input and then performing simple consistency checks on the resulting automata to test whether the translators seem to function correctly in practice. (See [TH02] for more information on the theory behind the testing methods.) If the test results suggest that there is an error in an implementation, `lbtt` can generate sample data which causes a test failure and which may also be useful for debugging the implementation.

Additionally, the testing environment can be used for very basic profiling of different LTL-to-Büchi translators to evaluate their performance.

Note: although `lbtt` *might be able to detect inconsistent behavior in an LTL-to-Büchi translator, it is only a testing tool and is therefore incapable of formally proving any translation algorithm implementation to be correct. Therefore, the test results should never be used as the sole basis for any formal conclusions about the correctness of an implementation.*

2 Test methods

This chapter describes the algorithms `lbtt` uses for generating input for the tests and introduces some terminology. A short description of each test is also included together with the outline of `lbtt`'s testing procedure. However, the chapter is not intended to be a thorough introduction to the theoretical background of the different tests; see, for example, [TH02] or [Tau00] for more information.

2.1 Random input generation

By default, all tests `lbtt` makes are based on randomly generated input. However, the LTL formulas used as input for the LTL-to-Büchi translators can be optionally given by the user by telling `lbtt` to read LTL formulas from a file or from standard input (see [‘`--formulafile`’ command line option], page 30).

Additionally, some of the tests make use of randomly generated “state spaces”, which are basically directed labeled graphs with labels on nodes with the additional requirement of having at least one arc leaving each node. The label of each node is a subset of a finite collection of atomic propositions (an uninterpreted set of assertions which may or may not hold in a state), which occur also in the LTL formulas used in the tests.

The following sections describe how `lbtt` generates input for the tests and list the parameters which can be used to adjust the behavior of the input generation algorithms.

2.1.1 Random LTL formulas

The LTL formulas used by `lbtt` are built from atomic propositions (with names of the form ‘`p`*n*’ for some nonnegative integer *n*), the Boolean constants True and False, and logical or temporal operators. `lbtt` supports the following logical operators:

- logical disjunction ($\vee$ — ‘`\/`’ as shown in output messages),
- logical conjunction ($\wedge$ — ‘`/\`’),
- logical negation ($\neg$ — ‘`!`’),
- logical implication ($\rightarrow$ — ‘`->`’)
- logical equivalence ($\leftrightarrow$ — ‘`<->`’)
- logical “exclusive or” ($\oplus$ — ‘`xor`’)

and the following temporal operators:

- “Next time” (**X** — ‘`X`’),
- “(Strong) Until” (**U** — ‘`U`’),
- “Weak Until” (also known as “Unless”) (**W** — ‘`W`’),
- “Finally” (“Eventually”) (**F** — ‘`<>`’)

- "Before" (**B** — 'B')
- "(Weak) Release", the dual of "(Strong) Until" (**V** — 'V'),
- "Strong Release", the dual of "Weak Until" (**M** — 'M'),
- "Globally" ("Always", "Henceforth") (**G** — '[]').

See Section A.1 [LTL formulas], page 63, for a reference on the exact semantics of these operators.

The behavior of `lbtt`'s random formula generation algorithm can be adjusted with the following parameters:

- Number of nodes in the parse tree of a formula (i.e., the total number of occurrences of propositions, Boolean constants and operators in the formula).
- Number of different atomic propositions that can be used for generating a formula. (Note that this does not restrict the total number of atomic propositions in the formula, nor the number of occurrences of any individual proposition. However, none of the generated formulas will have more than this number of *different* atomic propositions.)
- Priorities for the Boolean constants and atomic propositions. The priority of a symbol determines the relative likelihood of its occurrence in a generated formula. The higher the priority of a symbol, the more likely it is that the symbol will occur (with respect to the other symbols) in a generated formula; a zero priority will exclude the symbol altogether.
- Priorities for the logical and temporal operators.

Note that the priorities for atomic symbols (Boolean constants and atomic propositions) and the priorities of the logical and temporal operators are independent, i.e., changing the priority of an atomic symbol does not affect the likelihood of the occurrence of any logical or temporal operator and vice versa.

2.1.1.1 The formula generation algorithm

`lbtt` uses an algorithm similar to the one outlined in [DGV99] for generating random LTL formulas. The algorithm can be described in pseudocode as follows:

```
function RandomFormula($n$ : Integer) : LtlFormula;
    if $n$ = 1 then begin
        $p$ := random atomic proposition or TRUE or FALSE;
        return $p$;
    end
    else if $n$ = 2 then begin
        $op$ := random unary operator;
        $f$ := RandomFormula(1);
        return $op$ $f$;
    end
    else begin
        $op$ := random unary or binary operator;
```

```
13          if op is a unary operator then begin
14              f := RandomFormula(n-1);
15              return op f;
16          end
17          else begin
18              x := random integer in the interval [1,n-2];
19              f1 := RandomFormula(x);
20              f2 := RandomFormula(n-x-1);
21              return (f1 op f2);
22          end;
23      end;
24  end;
```

Each invocation of the algorithm returns an LTL formula with n nodes in the formula parse tree. The behavior of the algorithm can be adjusted by giving values for the parameters n (the number of nodes in the formula parse tree), $|AP|$ (the number of different atomic propositions), and $pri(SYMBOL)$ (the priorities for the different symbols).

In `lbtt`'s implementation of the above algorithm, the priority of a symbol determines the probability with which the symbol is chosen into a generated formula each time line 3, 7 or 12 of the algorithm is executed. For Boolean constants True and False (line 3 of the algorithm), the probability is given by the equation

$$pri(CONSTANT)/(pri(AP) + pri(\textsc{True}) + pri(\textsc{False}))$$

where *CONSTANT* is either True or False, and $pri(AP)$ is the total priority of all atomic propositions.

The probability of choosing a particular atomic proposition into a formula (line 3) is

$$pri(AP)/\Big(|AP| \times (pri(AP) + pri(\textsc{True}) + pri(\textsc{False}))\Big)$$

where $|AP|$ and $pri(AP)$ are as defined above.

Line 7 of the algorithm concerns choosing a unary operator ($\neg$, **X**, **F** or **G**) into a formula. Here the probability of choosing the unary operator *op* is given by the equation

$$pri(op)/\sum_{\mathrm{op}' \in \{\neg, \mathbf{X}, \mathbf{F}, \mathbf{G}\}} pri(op')$$

where op' ranges over all unary operators.

Finally, at line 12 of the algorithm, the probability of choosing the (unary or binary) operator *op* into the formula is

$$pri(op)/\sum_{\mathrm{op}' \in \{\neg, \vee, \wedge, \rightarrow, \leftrightarrow, \oplus, \mathbf{X}, \mathbf{U}, \mathbf{W}, \mathbf{F}, \mathbf{B}, \mathbf{V}, \mathbf{M}, \mathbf{G}\}} pri(op')$$

where op' ranges over all unary and binary operators.

An analysis of this algorithm is included in an appendix of [Tau00]. The analysis shows how to use the operator priorities to calculate the expected number of occurrences of an operator in a randomly generated formula. `lbtt` can optionally compute the expected operator distribution for a given combination of operator priorities; see the '`--showoperatordistribution`' command line option (see ['`--showoperatordistribution`' command line option], page 31) for more information.

See also the web page `<http://www.tcs.hut.fi/Software/lbtt/formulaoptions.php>` for an interface to a small database for adjusting the operator priorities towards certain simple distributions.

2.1.2 Random state spaces

State spaces are needed as input for tests only in the model checking result cross comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16) and the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17). The state spaces are directed labeled graphs, each node of which is labeled with a randomly chosen set of atomic propositions (the propositions that hold in the state corresponding to the graph node). In addition, each state of the state space always has at least one successor.

`lbtt` provides three different random state space generation algorithms that differ in the structure of the generated state spaces. The common parameters for each of these algorithms are:

- Number of states in the state space.
- Maximum number of different atomic propositions allowed in the label of any state of the state space.
- The probability with which each atomic proposition should hold in a state of the state space (which is, for simplicity, common to all atomic propositions).

The different types of random state spaces that can be generated are:

1. Random connected graphs. These state spaces are guaranteed to have at least one state such that every other state of the state space is reachable from this state. In addition to the three above parameters, the behavior of the algorithm generating these state spaces can be adjusted by specifying a probability which approximates the *density* of the graph, i.e., the probability that there is a directed edge from a state x to another state y, where x and y are any two states in the state space. For more details, see Section 2.1.2.1 [Algorithm for generating connected graphs], page 14.
2. Random graphs. These state spaces are constructed simply by taking all pairs (x, y) of states in the state space and connecting state x to state y with a user-specified probability that approximates the graph density.

3. Random paths. A random path is simply a non-branching sequence of states, where the last state of the sequence is connected to a randomly chosen state earlier in the sequence.

`lbtt` also includes a state space generation algorithm which systematically enumerates all "paths" (see above) of a given size with a given number of atomic propositions in each state. If $|S|$ is the number of states in the path and $|AP|$ is the number of atomic propositions in each state of the path, it is easy to see that the number of different paths having these parameters is

$$|S| \cdot 2^{|S| \cdot |AP|},$$

a number which grows exponentially in the product of the two parameters. Obviously, this makes the exhaustive enumeration of all paths of a given size practicable only for very small values of $|S|$ and $|AP|$.

In practice, testing should be started using only very small state spaces (say, with only 10–50 states and a small density) regardless of the particular algorithm chosen for generating the state spaces. The size of the state spaces can then be increased if `lbtt`'s memory consumption and the time spent running the tests stay within acceptable limits.

2.1.2.1 Algorithm for generating connected graphs

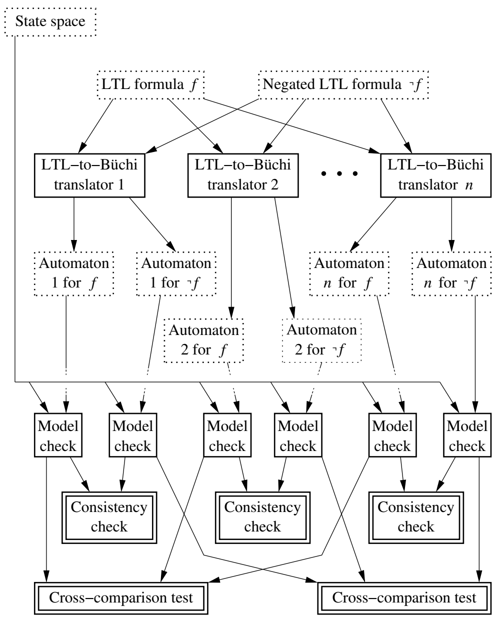
`lbtt` uses the following algorithm for generating random connected graphs:

```
function RandomGraph(n : Integer; d : Real in [0.0,1.0];
                     t : Real in [0.0,1.0]) : Graph;
    S := {s1, s2, ..., sn};
    NodesToProcess := {s1};
    UnreachableNodes := {s2, s3, ..., sn};
    Edges := {};
    while NodesToProcess is not empty do begin
        state := a random node in NodesToProcess;
        remove state from NodesToProcess;
        Label(state) := {};
        for all p in AP do
            if RandomNumber(0.0, 1.0) < t then
                insert p into Label(state);
        if UnreachableNodes is not empty then begin
            state' := a random node in UnreachableNodes;
            remove state' from UnreachableNodes;
            insert state' into NodesToProcess;
            insert (state,state') into Edges;
        end;
        for all state' in S do
            if RandomNumber(0.0, 1.0) < d then begin
                insert (state,state') into Edges;
                if state' is in UnreachableNodes then begin
                    remove state' from UnreachableNodes;
                    insert state' into NodesToProcess;
                end;
            end;
        if there is no edge (state,state') in Edges
                                    for any state' in S then
            insert (state,state) into Edges;
    end;
    return <S, Edges, s1, Label>;
end;
```

The algorithm receives the parameters n (number of states in the state spaces), d (approximate density of the generated graph) and t (the probability with which each of the propositions in AP should hold in a state) and returns the generated state space as a quadruple <*S*, *Edges*, s1, *Label*>. Here S is the set of states, *Edges* is the set of directed edges between the states, $s1$ is a state from which every state of the state space can be reached, and *Label* is a function which maps each state to its label (a subset of AP).

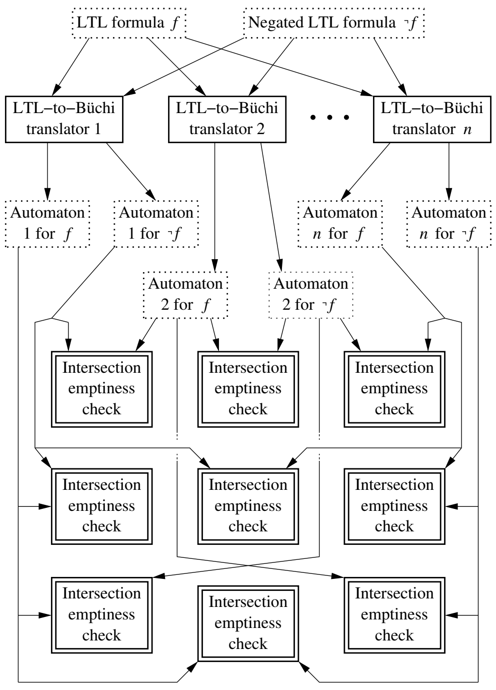
2.2 Testing procedure

The following figure illustrates the first two tests in `lbtt`'s testing procedure:



After obtaining an LTL formula f (either by reading it from a file or by calling the random formula generation algorithm), `lbtt` invokes each LTL-to-Büchi translator participating in the tests in turn to construct a collection of Büchi automata for the formula f *and* the negated formula $\neg f$. Each of these automata is then composed with the randomly generated state space, whereafter `lbtt` performs the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16) and the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17) on the model checking results, and reports all detected failures.

The Büchi automata intersection emptiness check (see Section 2.5 [Automata intersection emptiness check], page 18) operates as follows (note that the LTL-to-Büchi translation phase is repeated in this figure only for completeness; in reality, `lbtt` performs this phase only once):


LTL formula f
Negated LTL formula $\neg f$
LTL–to–Büchi translator 1
LTL–to–Büchi translator 2
LTL–to–Büchi translator n
Automaton 1 for f
Automaton 1 for $\neg f$
Automaton 2 for f
Automaton 2 for $\neg f$
Automaton n for f
Automaton n for $\neg f$
Intersection emptiness check

The test procedure can then be repeated using a different LTL formula and/or a different state space.

2.3 Model checking result cross-comparison test

LTL model checking can be used to test whether any of the infinite paths starting from some state of a state space satisfies a given LTL formula. For a fixed LTL formula, this question may have a different answer in different states of the state space, but the answer should be independent of the details of any (correct) implementation of the LTL model checking procedure.

Therefore, it is possible to test LTL-to-Büchi translators by comparing the results obtained by model checking an LTL formula in a fixed state space several times, using each time a different translator for constructing a Büchi automaton from the LTL formula. Differences in the model checking results then suggest that at least one of the translators failed to translate the LTL formula correctly into an automaton.

To extract as much test data as possible from a state space, `lbtt` will by default make the model checking result comparison “globally” in the state space, which means using each LTL-to-Büchi translator to find *all* states in the state space with an infinite path supposedly satisfying some LTL formula and then comparing the resulting state sets for equality. Alternatively, the test can be performed only “locally” in a single state of each state space (i.e., by choosing some state of the state space and checking that all Büchi automata constructed using the different translators give the same model checking result in that state), which may speed up testing, but will reduce the number of comparison tests. In addition, `lbtt` repeats the result cross-comparison test for the negation of each LTL formula, since model checking also the negated formula permits making an additional consistency check (see below) on the results computed using each implementation.

Note: If the generated state spaces are paths (either random or systematically enumerated, see Section 2.1.2 [Random state spaces], page 13), `lbtt` will then include its internal LTL model checking algorithm (a restricted model checking algorithm used normally in test failure analysis, see Chapter 5 [Analyzing test results], page 44) into the model checking result cross-comparison test. This is especially useful if there is only one translation algorithm implementation available for testing (in which case normal model checking result cross-comparison is obviously redundant) but may be of advantage also in other cases by providing an additional implementation to include in the tests.

2.4 Model checking result consistency check

LTL model checking tells whether any of the infinite paths starting from some state of a state space satisfies a given LTL formula. If there are no such paths beginning from the state, it follows that all infinite paths beginning from the state must then satisfy the *negation* of the same formula. Since all state spaces used by `lbtt` always have at least one path beginning from each state of the state space (guaranteed by the state space generation algorithms), at least one path beginning from any state must satisfy either the formula or its negation, i.e., it cannot be the case that none of the paths is a model of either formula.

However, implementation errors in an LTL-to-Büchi translator used for model checking may actually lead to this inconsistent model checking result if the translation of either of the formulas results in an incorrect automaton, in which case `lbtt` will report an error.

Similarly to the model checking result cross-comparison test, the model checking result consistency check can be performed either in all states of the state space (“globally”) or only in a single state of the state space (“locally"), with the same trade-offs between testing speed and number of tests as described in the previous section.

2.5 Automata intersection emptiness check

The semantics of LTL guarantees that no model of an LTL formula can be the model of the negation of the same formula. In terms of Büchi automata, this implies that the languages accepted by automata constructed from two complementary LTL formulas should be disjoint. This can be confirmed by intersecting the automata (i.e., by composing the automata to construct a third Büchi automaton that accepts precisely those inputs accepted by both of the original automata) and checking the result for emptiness. If the intersection is found to be nonempty, however, at least one of the LTL-to-Büchi translator(s) used for constructing the original automata must have failed to perform the translation of either formula correctly.

3 Invocation

`lbtt` is started with the command `lbtt` with optional command line parameters. Before starting the program, however, you need to create a configuration file which lists the LTL-to-Büchi translators to be tested and defines additional testing parameters. See Section 3.1 [Configuration file], page 19. If no suitable configuration file is found or if the configuration file cannot be processed successfully, `lbtt` exits with an error message.

After reading the configuration file, `lbtt` starts tests on the LTL-to-Büchi translators listed in the configuration file (for details about the testing procedure, see Chapter 2 [Test methods], page 10). The program exits after a predetermined number of test rounds.

If the program is started in any of its interactive modes (see [Interactivity modes], page 21), the program may occasionally pause to wait for user input between test rounds. Type '`quit` ⟨ENTER⟩' at the prompt to exit `lbtt` at this point (or see Chapter 5 [Analyzing test results], page 44, for more information on how to use `lbtt`'s internal commands).

3.1 Configuration file

The configuration file of `lbtt` contains a list of the LTL-to-Büchi translators to be tested along with other options which affect the way the tests are performed. The configuration file is processed before starting the tests. By default, `lbtt` will try to read the configuration from the file '`config`' in the current working directory; a different file name can be specified with the '`--configfile=`*`filename`*' command line option.

The configuration file consists of one or more sections, each of which provides a collection of interrelated configuration options. The general format of the configuration file is

```
section-name
{
  option-name = value
  option-name = value
  ...
}

...
```

Section and option names are case-insensitive. Values can be numbers, strings or truth values ('`yes`' and '`no`', or equivalently, '`true`' and '`false`'). String values are case-sensitive and are subject to common quoting and escaping rules (i.e., string values containing white space should be enclosed in quotes, or the white space characters should be escaped with '\').

Comments can be included by putting a '`#`' symbol before them; the end of any line containing the '`#`' character will be ignored when processing the configuration file.

The configuration file must contain at least one '`Translator`' section specifying an LTL-to-Büchi translator. The other sections are optional and can be used to override the default testing parameters.

3.1.1 The '`Translator`' section

Each LTL-to-Büchi translator to be tested requires a separate '`Translator`' section[1] in the configuration file; there must be at least one such section in the file.

The translators are assumed to be accessible through external executable files. Therefore, this section must at a minimum specify the full file name of the executable to run in order to invoke the translator; see Section 6.1 [Translator interface], page 52, for information about the conventions `lbtt` uses to communicate with the LTL-to-Büchi translators.

Translators specified in the configuration file are given unique integer identifiers in the order they are listed in the file, starting from zero. These numbers can be used when referring to the different translators when using `lbtt`'s internal commands (see Chapter 5 [Analyzing test results], page 44). Alternatively, the translators can be referred to using the names specified in the configuration file.

The following options (in alphabetical order) are available within this section:

'`Enabled =` *`TRUTH-VALUE`*'

> This option determines whether the translator should be initially included in or excluded from the tests. The default value is '`Yes`'. The translator can be enabled or disabled during testing with `lbtt`'s internal commands (see Section 5.3 [Test control commands], page 45).

'`Name =` *`STRING`*'

> This option can be used to specify a unique textual identifier for the LTL-to-Büchi translator. `lbtt` will use this identifier when displaying various messages concerning the implementation; the identifier can also be used to refer to the implementation when working with `lbtt`'s internal commands (see Chapter 5 [Analyzing test results], page 44). (If no name has been explicitly given for the translator, `lbtt` assigns the translator a name of the form '`Algorithm` *`n`*', where *n* is the running integer identifier for the translators.)
>
> The identifier '`lbtt`' is reserved for `lbtt`'s internal model checking algorithm (see Section 2.3 [Model checking result cross-comparison test], page 16).

[1] The '`Algorithm`' and '`Implementation`' keywords are recognized as aliases of the '`Translator`' keyword.

'`Parameters = `*`STRING`*'
: This option can be used to specify any additional parameters that should be passed to the translator executable whenever running it. (The parameter string defaults to the empty string if the option is not used.)

'`Path = `*`STRING`*'
: This option must be given a value for each translator specified in the configuration file. The value should be the complete file name of the program which is used to run the translator.

3.1.2 The '`GlobalOptions`' section

The '`GlobalOptions`' section includes options that affect the general behavior of `lbtt`. Options available within this section are (in alphabetical order):

'`ComparisonCheck = `*`TRUTH-VALUE`*'
'`ComparisonTest = `*`TRUTH-VALUE`*'
: This option can be used to enable or disable the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16). The test is enabled by default.

'`ConsistencyCheck = `*`TRUTH-VALUE`*'
'`ConsistencyTest = `*`TRUTH-VALUE`*'
: This option can be used to enable or disable the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17). The test is enabled by default.

'`Interactive = `*`MODE-LIST`*'
: This option determines when `lbtt` should pause to wait for user input between test rounds. The *MODE-LIST* is a comma-separated list of the following modes (with no spaces in between the modes):

'`Always`'	Pause unconditionally after each test round.
'`OnError`'	Pause testing only after failed test rounds.
'`Never`'	Run all tests without interruption.
'`OnBreak`'	Pause testing when requested by the user (for example, after receiving a break signal from the keyboard). If this mode is not specified, '`lbtt`' will respond to break signals by aborting.

(Since the first three interactivity modes are mutually exclusive, it does not make sense to combine these modes with each other.) The default mode list consists of the value '`Always`', that is, testing is paused after every test round, and signalling a break will abort testing.

'`IntersectionCheck = `*`TRUTH-VALUE`*'
'`IntersectionTest = `*`TRUTH-VALUE`*'
: This option can be used to enable or disable the Büchi automata intersection emptiness check (see Section 2.5 [Automata intersection emptiness check], page 18). The test is enabled by default.

'`ModelCheck = Local | Global`'
: This option determines whether `lbtt` should perform model checking with respect to all states of each state space or only with respect to a single state of each state space. This affects the number of tests that `lbtt` makes during the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16) and the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17). Global model checking (the default) maximizes the number of tests, but may require more time and memory. (Note: This option has no effect if none of the model checking tests is enabled.)

'`Rounds = `*`INTEGER`*'
: The '`Rounds`' option can be used to specify the number of test rounds to run; the default value is 10.

'`TranslatorTimeout = `*`TIME-SPECIFICATION`*'
: This option can be used to specify a time limit (in wall-clock time) after which the execution of a translator is aborted if it fails to produce a result within the given limit. A timeout is considered a test failure. The time specification is of the form '[*`hours`*]`h`[*`minutes`*]`min`[*`seconds`*]`s`' where *hours*, *minutes* and *seconds* specify the time limit in the obvious way (time units having value 0 can be omitted). For example, a limit of '`1h30min`' sets the limit at one hour and thirty minutes.

'`Verbosity = `*`INTEGER`*'
: This option sets the verbosity level for output messages. The value can be an integer between 0 and 5 (inclusive). A value of 0 will suppress all messages (and is therefore useful only when storing test results into a log file; see ['`--logfile`' command line option], page 31); increasing the value results in more output. The default value is 3.

3.1.3 The '`FormulaOptions`' section

The '`FormulaOptions`' section defines the parameters that affect the algorithm `lbtt` uses for generating random LTL formulas (for more information about the algorithm, see Section 2.1.1 [Random LTL formulas], page 10). This section provides the following options:

‘`AbbreviatedOperators = TRUTH-VALUE`’
: This option determines whether the generated formulas should be allowed to include any of the operators ‘`->`’, ‘`<->`’, ‘`xor`’, ‘`W`’, ‘`<>`’, ‘`B`’, ‘`V`’, ‘`M`’ or ‘`[]`’ (all of which can be given definitions using only the operators ‘`!`’, ‘`\/`’, ‘`/\`’, ‘`U`’ and ‘`V`’). Setting this option to ‘`No`’ assigns each of the abbreviated operators a zero priority, overriding any explicit priorities defined for these operators in the program configuration. The default value for the option is ‘`Yes`’, so abbreviations are allowed by default.

‘`AndPriority = INTEGER`’
: Priority of the logical conjunction operator (‘`/\`’).

‘`BeforePriority = INTEGER`’
: Priority of the temporal operator “before” (‘`B`’). (Note: This option has no effect if ‘`AbbreviatedOperators`’ is set to ‘`No`’.)

‘`ChangeInterval = INTEGER`’
: This option determines how often (in number of test rounds) `lbtt` should generate a new random LTL formula (or read a new formula from a user-specified file). A value of 0 forces `lbtt` to use a fixed LTL formula for all tests. The default value is 1, i.e., a new formula will be generated at the beginning of each test round.

‘`DefaultOperatorPriority = INTEGER`’
: This option sets the priority for all formula operators for which no priority has been given explicitly in the program configuration (i.e., it can be used as a shorthand to initialize the priority of all operators). The default value of this option is 0, so all operators with no explicitly given priorities are disabled by default.

‘`EquivalencePriority = INTEGER`’
: Priority of the logical equivalence operator (‘`<->`’). (Note: This option has no effect if ‘`AbbreviatedOperators`’ is set to ‘`No`’.)

‘`FalsePriority = INTEGER`’
: Priority of the Boolean constant FALSE (with respect to priorities of the constant TRUE and the atomic propositions).

‘`FinallyPriority = INTEGER`’
: Priority of the temporal operator “finally” (‘`<>`’). (Note: This option has no effect if ‘`AbbreviatedOperators`’ is set to ‘`No`’.)

‘`GenerateMode = Normal | NNF`’
: This option determines whether `lbtt` should generate random formulas directly into (a weakened version of) negation normal form in which the negation operator may only precede atomic propositions. Note that the formulas may still contain “abbreviated” operators if they have nonzero priorities—use

'`AbbreviatedOperators=No`' or '`OutputMode=NNF`' if you wish to prevent this. The default value for this option is '`Normal`'. (See the '`OutputMode`' option below for an example about the differences in the effects of the '`GenerateMode`' and '`OutputMode`' options.)

'`GloballyPriority = ` *`INTEGER`*'
: Priority of the temporal operator "globally" ('`[]`'). (Note: This option has no effect if '`AbbreviatedOperators`' is set to '`No`'.)

'`ImplicationPriority = ` *`INTEGER`*'
: Priority of the logical implication operator ('`->`'). (Note: This option has no effect if '`AbbreviatedOperators`' is set to '`No`'.)

'`NextPriority = ` *`INTEGER`*'
: Priority of the temporal operator "next time" ('`X`').

'`NotPriority = ` *`INTEGER`*'
: Priority of the logical negation operator ('`!`').

'`OrPriority = ` *`INTEGER`*'
: Priority of the logical disjunction operator ('`\/`').

'`OutputMode = Normal | NNF`'
: This option determines whether `lbtt` should transform each generated LTL formula into (strict) negation normal form before passing it to LTL-to-Büchi translators. If the value is set to '`NNF`', `lbtt` will rewrite each generated formula into a form consisting of the operators '`!`', '`\/`', '`/\`', '`U`' and '`V`' such that all negations in the formula (if any) precede atomic propositions. The default value is '`Normal`'. (See also the '`GenerateMode`' option that can be used to force formulas to be generated directly into negation normal form.)

The option is probably useful only if you have a translator which does not support the "abbreviated" operators directly, but you still wish to test it with formulas which describe properties expressed using these operators. Note, however, that rewriting may change the size of the formula.

The following table illustrates the effects of the '`GenerateMode`' and the '`OutputMode`' options.

LTL formula `f`	Can `f` be generated if `GenerateMode` `=NNF` ?	`OutputMode`'s effect on the formula passed to the LTL-to-Büchi translators
`(p1 V ! p0)`	Yes	`Normal/NNF: (p1 V ! p0)`

`[] p0 -> <> p1`	Yes*	`Nor: [] p0 -> <> p1`
		`NNF: (true U ! p0) \/ (true U p1)`

`! <> p0`	No	`Nor: ! <> p0`
		`NNF: (false V ! p0)`

* only if `AbbreviatedOperators=Yes`

'`PropositionPriority = INTEGER`'
: Priority for atomic propositions with respect to the priority of Boolean constants. This priority is the common priority of *all* atomic propositions.

'`Propositions = INTEGER`'
: This option sets the maximum number of different atomic propositions in each generated LTL formula. No generated formula will have more than this number of different atomic propositions. A value of 0 will generate random formulas with only Boolean constants (one of which must in this case have a nonzero priority). The default value is 5. The names of the propositions are of the form '`pn`', where *n* is a nonnegative integer less than the maximum number of propositions.

'`RandomSeed = INTEGER`'
: This option specifies a seed value for generating random numbers for the random LTL formula generation algorithm. If this option is not present, the seed defaults to 1. See the next section for information on how to change the default seed for the random state space generation algorithm.

(The reason for having two separate random seeds is to make the sequences of random formulas and state spaces independent of each other. For example, this makes it easy to repeat tests using the same batch of random LTL formulas, but with state spaces of different size.)

'`ReleasePriority = INTEGER`'
: Priority of the temporal "(weak) release" operator ('`V`').

'`Size = INTEGER`'
'`Size = MINIMUM-SIZE-MAXIMUM-SIZE`'
'`Size = MINIMUM-SIZE...MAXIMUM-SIZE`'
: This option defines how many nodes are allowed in the parse tree of each randomly generated LTL formula. If the size is given as an interval (by separating the bounds with '`-`' or '`...`' with no white space in between), `lbtt` chooses the size of each formula randomly in the interval using a uniform random distribution. The default size is 5.

'`StrongReleasePriority = INTEGER`'
: Priority of the temporal "strong release" operator ('`M`'). (Note: This option has no effect if '`AbbreviatedOperators`' is set to '`No`'.)

'`UntilPriority = `*`INTEGER`*'
: Priority of the temporal "(strong) until" operator ('`U`').

'`TruePriority = `*`INTEGER`*'
: Priority of the Boolean constant TRUE (with respect to the priorities of the constant FALSE and the atomic propositions).

'`WeakUntilPriority = `*`INTEGER`*'
: Priority of the temporal "weak until" operator ('`W`'). (Note: This option has no effect if '`AbbreviatedOperators`' is set to '`No`'.)

'`XorPriority = `*`INTEGER`*'
: Priority of the logical "exclusive or" operator ('`xor`'). (Note: This option has no effect if '`AbbreviatedOperators`' is set to '`No`'.)

3.1.4 The '`StateSpaceOptions`' section

The '`StateSpaceOptions`' section defines the parameters that affect the way `lbtt` generates random state spaces for the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16) and the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17). See also Section 2.1.2 [Random state spaces], page 13, for more information about the different types of available state spaces and the algorithms used for constructing them. The options available within this section are:

'`ChangeInterval = `*`INTEGER`*'
: This option determines how often (in number of test rounds) `lbtt` should generate a new random state space. A value of 0 forces `lbtt` to use a fixed state space for all tests. The default behavior is to generate a new state space at the beginning of each test round.

'`EdgeProbability = `*`PROBABILITY`*'
: This option sets the approximate probability (between 0.0 and 1.0) of adding a transition from any state x to some other state y when generating random graphs as state spaces. The default value is 0.2. The probability is approximate because `lbtt` still has to ensure that all states of each generated state spaces have at least one successor, which might require adding extra transitions to the graph. Note: This option has no effect if '`GenerateMode`' is set to '`RandomPath`' or '`EnumeratedPath`'.

'`GenerateMode = RandomConnectedGraph | RandomGraph | RandomPath | EnumeratedPath`'
: This option selects the type of generated state spaces from the four available types. The default value is '`RandomConnectedGraph`'. See Section 2.1.2 [Random state

spaces], page 13, for more information on the different state space types.

Note: Using the '`RandomPath`' or the '`EnumeratedPath`' setting includes `lbtt`'s internal model checking algorithm into the various model checking tests if they are enabled. For more information, see Section 2.3 [Model checking result cross-comparison test], page 16.

'`Propositions = `*`INTEGER`*'

This option sets the number of atomic propositions attached to each state of the generated state spaces. The default value is 5.

Usually this should probably be the same as the maximum number of different atomic propositions in the generated formulas (see Section 3.1.3 [FormulaOptions section], page 22). If the number of propositions attached to each state of the state spaces is less than the maximum number of different propositions that may occur in the generated formulas, all "extra" propositions in the formulas are considered to be false in every state of the state space.

'`RandomSeed = `*`INTEGER`*'

This option specifies a seed value for generating random numbers required by the random state space generation algorithm. If this option is not present, the seed defaults to 1. See the previous section for how to change the random seed used to initialize the random number generator for the random LTL formula generation algorithm.

'`Size = `*`INTEGER`*'
'`Size = `*`MINIMUM-SIZE-MAXIMUM-SIZE`*'
'`Size = `*`MINIMUM-SIZE...MAXIMUM-SIZE`*'

This option sets the number of states in the generated state spaces. If the size is given as an interval, `lbtt` either chooses a random size in the interval (including its endpoints) each time a new state space is generated, or, if '`GenerateMode`' is set to '`EnumeratedPath`', enumerates all state spaces in the specified range systematically, starting from the minimum size. The default size is 20.

'`TruthProbability = `*`PROBABILITY`*'

Probability (between 0.0 and 1.0) with which each individual atomic proposition has the value TRUE in any state of the state space. Note: This option has no effect if '`GenerateMode`' is set to '`EnumeratedPath`'. The default value is 0.5.

3.1.5 Sample configuration file

The following configuration file sets `lbtt` up for testing two imaginary LTL-to-Büchi translators.

```
# Sample configuration file for lbtt

Translator
{
  Name = Translator\ 1
  Path = /home/lbtt-user/bin/t-1    # location of the translator
                                    # executable
  Enabled = Yes
}

Translator
{
  Name = "Translator 2"
  Path = /home/lbtt-user/bin/t-2
  Parameters = "-x -y 3 -v 0"       # parameters to be passed to the
                                    # executable
  Enabled = Yes
}

GlobalOptions
{
  Rounds = 100                      # 100 test rounds

  Interactive = OnError,OnBreak     # pause testing in case of an error
                                    # or when receiving a break signal

  Verbosity = 1                     # show only numeric statistics

  ComparisonTest = Yes              # enable all tests except the
  ConsistencyTest = Yes             # Büchi automata intersection
  IntersectionTest = No             # emptiness test

  ModelCheck = Local                # perform the tests only in a
                                    # single state of each state
                                    # space

  TranslatorTimeout = 30s           # abort the execution of a
                                    # translator if it fails to give
                                    # a result in 30 seconds
}

FormulaOptions
{
  AbbreviatedOperators = Yes        # formula generation mode
  GenerateMode = Normal
  OutputMode = NNF                  # rewrite formulas in negation
                                    # normal form before passing
                                    # them to the translators

  ChangeInterval = 1                # new formula after each round

  RandomSeed = 4632912              # random seed
  Size = 5-15                       # 5 to 15 nodes in the parse
                                    # tree of each formula

  Propositions = 3                  # allow at most three different
                                    # propositions in each LTL formula

  PropositionPriority = 50          # priorities for propositional
  TruePriority = 1                  # symbols
  FalsePriority = 1

  AndPriority = 10                  # priorities for some logical
  OrPriority = 10                   # operators
  NotPriority = 10
  EquivalencePriority = 5

  NextPriority = 5                  # priorities for some temporal
  UntilPriority = 5                 # operators
  ReleasePriority = 5
  FinallyPriority = 2

  DefaultOperatorPriority = 0       # disable all the remaining
                                    # operators
}

StatespaceOptions
{
  GenerateMode = RandomGraph        # generate random (not
                                    # necessarily connected) graphs
                                    # as state spaces

  ChangeInterval = 10               # new state space after every
                                    # 10th test round

  RandomSeed = 37620                # random seed

  Size = 50                         # 50 states in each state space

  Propositions = 3                  # three propositions in each
                                    # state of each state space

  EdgeProbability = 0.1             # approximate probability of
                                    # having a transition between
                                    # any two states

  TruthProbability = 0.5            # probability with which any
                                    # atomic proposition is true in
                                    # a state
}
```

3.2 Command line options

This section lists the command line options that may be used when invoking `lbtt`. The command line options are processed only after reading the configuration file, so they can be used to override the settings given in the file. There are also a few options for which there is no direct equivalent in the configuration file options.

3.2.1 Special options

The following list presents all command line options for which there is no (directly) corresponding option that may be set in the program configuration file.

'`--configfile=`*`FILE-NAME`*'
: This option can be used to instruct `lbtt` to read program configuration from another file instead of the default configuration file '`config`' in the current working directory.

'`--formulafile=`*`FILE-NAME`*'
: This option instructs `lbtt` to read the LTL formulas used in the tests from a file (or standard input) instead of generating them randomly. The special filename '`-`' refers to standard input. Each input formula should be followed by a newline. The formulas can be specified either in `lbtt`'s own prefix notation (see Section 6.2 [Format for LTL formulas], page 53; also the infix notation used in output messages is supported) or in a variety of formats found in some LTL-to-Büchi translator implementations (Spin, LTL2BA, LTL2AUT, Temporal Massage Parlor, Wring, Spot, LBT), however with the restriction that all atomic propositions should have names of the form '`p`*`N`*' for some nonnegative integer *N*.

  (When using one of the alternative formats, it is recommended to use parentheses to avoid possible ambiguities in the precedence and associativity of the various operators; in `lbtt`, the unary operators have the highest precedence, '`/\`' has higher precedence than '`\/`', which in turn has higher precedence than any of '`->`', '`<->`' or '`xor`', and the binary temporal operators have the lowest precedence. All binary logical operators are left-associative; all binary temporal operators are nonassociative.)

  If this option is used, all command line or configuration file parameters affecting the generation of random LTL formulas (excluding their mode of output) are ignored.

'`--h`'
'`--help`'
: These options list all the available command line parameters.

‘`--logfile=`*`FILE-NAME`*’
: This option instructs `lbtt` to create a log of all errors encountered during testing. By default no log will be created.

‘`--profile`’
: This option can be used as a shorthand for disabling all Büchi automata correctness tests. The test report generated at the end of testing then shows only the running times of each tested LTL-to-Büchi translator and the sizes of the generated automata.

‘`--quiet`’
‘`--silent`’
: These options suppress any messages that are normally displayed during testing. Use the ‘`--logfile`’ option (see above) with these options to save a test failure report into a log file.

‘`--showconfig`’
: If this option is present on the command line, `lbtt` will write the current configuration to standard output (see Section 4.1 [Configuration information], page 38) and then exit. This option can be used together with the ‘`--configfile`’ option to test the settings defined in a configuration file without actually performing any tests.

‘`--showoperatordistribution`’
: With this option `lbtt` uses the priorities defined for the LTL formula operators available for random LTL formula generation to compute the expected number of occurrences of each operator in a single randomly generated formula. The distribution is then displayed along with other configuration information when the program starts.

‘`--skip=`*`NUMBER-OF-ROUNDS`*’
: This option can be used to skip the first *NUMBER-OF-ROUNDS* test rounds, i.e., begin testing from round *NUMBER-OF-ROUNDS*+1.

‘`-V`’
‘`--version`’
: This option displays the version of the `lbtt` executable.

3.2.2 Global options

The following list presents the options that can be used to override the values specified in the ‘`GlobalOptions`’ section of the configuration file.

‘`--comparisontest`[`=yes | no`]’
‘`--nocomparisontest`’
: These options enable or disable the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16).

'`--consistencytest`[`=yes | no`]'
'`--noconsistencytest`'
: These options enable or disable the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17).

'`--disable=`*`IMPLEMENTATION-ID`*[`,`*`IMPLEMENTATION-ID`*...]'
: This option can be used to exclude some implementations from the tests by specifying a comma-separated list of implementation names or their numeric identifiers. (The implementations are numbered in the order in which they appear in the configuration file, starting from zero. Use the '`--showconfig`' option, see Section 3.2.1 [Special options], page 30, to obtain a list of the implementations specified in the configuration file, together with their identifiers.)

'`--enable=`*`IMPLEMENTATION-ID`*[`,`*`IMPLEMENTATION-ID`*...]'
: This option can be used to include implementations into the tests (in the case they are initially disabled in the configuration file).

'`--globalmodelcheck`'
: This option instructs `lbtt` to perform model checking globally (with respect to all states of each random state space) in the model checking result cross-comparison test and the model checking result consistency check. Using a global check increases the number of possible tests.

'`--interactive`[`=`*`MODE-LIST`*]'
'`--pause`[`=`*`MODE-LIST`*]'
: These options can be used to override whether `lbtt` should pause between test rounds to wait for user input. The optional *MODE-LIST* is a comma-separated list of interactivity modes ('`Always`', '`OnError`', '`Never`', '`OnBreak`') with no spaces in between (see [Interactivity modes], page 21, for the mode descriptions). If omitted, the mode list defaults to '`Always`'.

'`--intersectiontest`[`=yes | no`]'
'`--nointersectiontest`'
: These options enable or disable the Büchi automata intersection emptiness check (see Section 2.5 [Automata intersection emptiness check], page 18).

'`--localmodelcheck`'
: This option instructs `lbtt` to perform model checking only with respect to a single state of each random state space in the model checking result cross-comparison test and the model checking result consistency check.

‘`--modelcheck=global | local`’
: This option can be used to select the model checking mode.

‘`--pause`[`=`*`MODE-LIST`*]’
: See ‘`--interactive`’.

‘`--rounds=`*`NUMBER-OF-ROUNDS`*’
: This option can be used to override the number of test rounds to run.

‘`--translatortimeout=`*`TIME-SPECIFICATION`*’
: This option can be used to override the running time limit (in wall-clock time) for translators (see [Timeouts], page 22, for more information).

‘`--verbosity=`*`INTEGER`*’
: This option sets the verbosity of output messages. The value must be between 0 and 5 (inclusive).

3.2.3 LTL formula options

The following command line options can be used to control the behavior of `lbtt`'s random LTL formula generation algorithm. They correspond to the options available in the ‘`FormulaOptions`’ section of the configuration file.

‘`--abbreviatedoperators`[`=yes | no`]’
‘`--noabbreviatedoperators`’
: These options can be used to allow or prevent `lbtt` from using any of the “abbreviated” operators (‘`->`’, ‘`<->`’, ‘`xor`’, ‘`W`’, ‘`<>`’, ‘`B`’, ‘`M`’ and ‘`[]`’) when generating random LTL formulas.

‘`--andpriority`’
: This option sets the priority for logical conjunction (the ‘`/\`’ operator).

‘`--beforepriority`’
: This option sets the priority for the temporal “before” operator (‘`B`’).

‘`--defaultoperatorpriority`’
: This option sets the default priority for all logical and temporal operators.

‘`--equivalencepriority`’
: This option sets the priority for logical equivalence (the ‘`<->`’ operator).

‘`--falsepriority`’
: This option sets the priority for the Boolean constant ‘`false`’.

‘`--finallypriority`’
: This option sets the priority for the temporal “finally” operator (‘`<>`’).

‘`--formulachangeinterval=`*`NUMBER-OF-ROUNDS`*’
: This option determines how often (in number of test rounds) `lbtt` should generate a new random LTL formula. A value of 0 forces `lbtt` to use a fixed LTL formula for all tests.

‘`--formulagenerratemode=normal | nnf`’
: This option can be used to choose how `lbtt` should generate random LTL formulas. With the option ‘`--formulagenerratemode=nnf`’, all generated formulas will be in (a weakened) negation normal form in which all negations in the formula (if any) precede atomic propositions. (Note that the formulas may still contain some of the “abbreviated” operators if their priorities are not explicitly set to zero.)

‘`--formulaoutputmode=normal | nnf`’
: This option can be used to force or prevent `lbtt` from converting each LTL formula into (strict) negation normal form (i.e., rewriting it with the operators ‘`!`’, ‘`/\`’, ‘`\/`’, ‘`U`’ and ‘`V`’) before passing it to the LTL-to-Büchi translators.

‘`--formulapropositions`’
: This option sets the maximum number of different atomic propositions that `lbtt` may use for generating random LTL formulas.

‘`--formularandomseed=`*`INTEGER`*’
: This option gives a seed value for generating random numbers used by the random LTL formula generation algorithm.

‘`--formulasize=`*`INTEGER`*’
‘`--formulasize=`*`MINIMUM-SIZE-MAXIMUM-SIZE`*’
‘`--formulasize=`*`MINIMUM-SIZE`*`...`*`MAXIMUM-SIZE`*’
: This option sets the size of the random LTL formulas generated for the tests. The size can be given either as a fixed integer or as an interval, in which case the size of each generated formula will be chosen randomly in the interval using a uniform random distribution.

‘`--generatennf`’
‘`--nogeneratennf`’
: These options can be used instead of the ‘`--formulagenerratemode`’ option to select the random formula generation mode.

‘`--globallypriority`’
: This option sets the priority for the temporal “globally” operator (‘`[]`’).

‘`--implicationpriority`’
: This option sets the priority for logical implication (the ‘`->`’ operator).

‘`--nextpriority`’
: This option sets the priority for the temporal “next time” operator (‘`X`’).

‘`--notpriority`’
: This option sets the priority for logical negation (the ‘`!`’ operator).

‘`--orpriority`’
: This option sets the priority for logical disjunction (the ‘`\/`’ operator).

‘`--outputnnf`’
‘`--nooutputnnf`’
: These options can be used instead of the ‘`--formulaoutputmode`’ option to choose the format in which `lbtt` passes LTL formulas to LTL-to-Büchi translators.

‘`--propositionpriority`’
: This option sets the priority for atomic propositions.

‘`--releasepriority`’
: This option sets the priority for the temporal “(weak) release” operator (‘`V`’).

‘`--strongreleasepriority`’
: This option sets the priority for the temporal “strong release” operator (‘`M`’).

‘`--truepriority`’
: This option sets the priority for the Boolean constant True.

‘`--untilpriority`’
: This option sets the priority for the temporal “(strong) until” operator (‘`U`’).

‘`--weakuntilpriority`’
: This option sets the priority for the temporal “weak until” operator (‘`W`’).

‘`--xorpriority`’
: This option sets the priority for the logical “exclusive or” operator.

Note also the ‘`--formulafile=`*`FILE-NAME`*’ option (see [‘`--formulafile`’ option], page 30), which can be used to instruct `lbtt` to read LTL formulas from a file (or standard input) instead of generating them randomly.

3.2.4 State space options

The following command line options affect the way in which `lbtt` generates state spaces that are then used in the model checking tests. They correspond to options in the '`StateSpaceOptions`' section of the configuration file. See also Section 2.1.2 [Random state spaces], page 13, for more information about the graph generation modes.

'`--edgeprobability=`*PROBABILITY*'
: This option sets the approximate random edge probability for state spaces. (The option has no effect if the generated state spaces are random or enumerated paths.)

'`--enumeratedpath`'
: This option instructs `lbtt` to enumerate all paths of a given size as state spaces instead of generating random state spaces for model checking tests. The option also enables `lbtt`'s internal model checking algorithm.

'`--randomconnectedgraph`'
: This option makes `lbtt` generate random connected graphs as state spaces for model checking tests.

'`--randomgraph`'
: This option makes `lbtt` generate random graphs as state spaces for model checking tests.

'`--randompath`'
: This option forces `lbtt` to generate random paths as state spaces. The option also enables `lbtt`'s internal model checking algorithm in the model checking tests.

'`--statespacechangeinterval=`*NUMBER-OF-ROUNDS*'
: This option sets the frequency (in test rounds) in which new state spaces are generated. A value of 0 forces `lbtt` to use a fixed state space for all tests.

'`--statespacegeneratemode=randomconnectedgraph | randomgraph | randompath | enumeratedpath`'
: This option can be used instead of one of the four options above to select the state space generation mode.

'`--statespacerandomseed=`*INTEGER*'
: This option gives a seed value for generating random numbers required by the random state space generation algorithm.

'`--statespacesize=`*INTEGER*'
'`--statespacesize=`*MINIMUM-SIZE-MAXIMUM-SIZE*'
'`--statespacesize=`*MINIMUM-SIZE*`...`*MAXIMUM-SIZE*'
: This option can be used to change the size of the generated state spaces.

'`--truthprobability=`*PROBABILITY*'
: This option sets the probability that `lbtt` uses for choosing the valuation for each atomic proposition in each state of the ran-

domly generated state spaces. (This option has no effect if using enumerated paths as state spaces.)

4 Interpreting the output

This chapter briefly introduces the most typical messages that `lbtt` outputs during testing. Most of the examples in this section illustrate the output when `lbtt` is running in its default output verbosity mode (3). In lower verbosity modes some (or in verbosity mode 0, all) of these messages will be suppressed; in higher verbosity modes, some additional information about `lbtt`'s internal behavior is shown.

4.1 Configuration information

Before starting tests, `lbtt` outputs (in verbosity modes 2 and above) a summary of the current program configuration as obtained by reading the program configuration file and interpreting the command line parameters. The same summary can be obtained without running any tests by using the '`--showconfig`' command line option (see ['`--showconfig`' option], page 31). The information will be written also to the error log file if one was specified in the command line with the '`--logfile`' option (see ['`--logfile`' option], page 31). The summary consists of the following information:

- LTL-to-Büchi translator implementations enabled for testing.
- List of enabled tests.
- Random state space generation parameters.
- Random LTL formula generation parameters (unless reading LTL formulas from an external source; see ['`--formulafile`' command line option], page 30). This includes information about all enabled formula operators and their priorities. When using the command line option '`--showoperatordistribution`' (see ['`--showoperatordistribution`' option], page 31), `lbtt` shows also the expected number of occurrence of each operator in each randomly generated formula.

Example:

```
Program configuration:
----------------------

  1000 test rounds.
  Testing will be interrupted in case of an error.
  Signalling a break will interrupt testing.
  Using global model checking for tests.
  Writing error log to ‘error.log’.

  Implementations:
    0: ‘Implementation 0’
    1: ‘Implementation 1’

  Timeout for translators is set to 30 seconds.

  Enabled tests:
    Model checking result cross-comparison test
    Model checking result consistency check
    Büchi automata intersection emptiness check

  Random state spaces:
    Random graphs (50 states, 5 atomic propositions)
    New state space will be generated after every 5th round.
    Random seed: 98
    Random edge probability: 0.10
    Propositional truth probability: 0.50

  Random LTL formulas:
    5 parse tree nodes, 5 atomic propositions
    New LTL formula will be generated after every round.
    Random seed: 17991
    Atomic symbols in use (priority):
      false (5); propositions (90); true (5)
    Operators used for random LTL formula generation:
      operator  !        /\       U        V        X        \/
      priority  10       10       20       20       10       20
```

4.2 Test round messages

In verbosity modes 1 and 2, `lbtt` reports numeric statistics on the generated automata in tabular form. Each row of this table contains the following information (in this order):

- number of the current test round (verbosity mode 1 only);
- numeric identifier of an implementation;
- formula identifier (‘`+`’ or ‘`-`’);
- time consumed when generating an automaton from the formula using the implementation;
- number of states, transitions and acceptance conditions in the automaton;
- number of states and transitions in the product automaton
- number of accepting cycles in the state space (see below), and
- result of the consistency check (verbosity mode 2 only).

The following example shows a fragment of the output that `lbtt` might produce during a test round when running in the default verbosity mode 3.

```
1. Round 6 of 10

2.     Generating random state space

3.     Random LTL formula:
         formula:                 ((p1 <-> p0) U (p0 \/ ! p3))
         negated formula:         ! ((p1 <-> p0) U (p0 \/ ! p3))

      0: ‘Implementation 0’
```

```
        Positive formula:
4.        Büchi automaton:
            number of states:      6
            number of transitions: 15
            acceptance sets:       1
            computation time:      0.03      seconds (user time)
5.        Product automaton:
            number of states:      582       [97.00% of worst case (600)]
            number of transitions: 7188
6.        Accepting cycles:
            cycle reachable from   0         states
            not reachable from     100       states
7.      Negated formula:
          Büchi automaton:
            number of states:      4
            number of transitions: 6
            acceptance sets:       0
            computation time:      0.04      seconds (user time)
          Product automaton:
            number of states:      363       [90.75% of worst case (400)]
            number of transitions: 2581
          Accepting cycles:
            cycle reachable from   25        states
            not reachable from     75        states
8.      Result consistency check:
          result:                  failed    [75 (75.00%) of 100 test cases]
```

The numbered parts of the output are:

1. Number of the test round.
2. `lbtt` generates a new random state space for model checking tests. (In this case the size of the state spaces was fixed in the configuration; if the state space size is allowed to vary in an interval, `lbtt` would also show here the actual size of the generated state space.)
3. Information about a random LTL formula and its negation. To simplify the notation, it is assumed that all unary formula operators have higher precedence than binary operators.
4. Information about the Büchi automaton that 'Implementation 0' generated from the positive LTL formula (number of states, transitions and acceptance conditions, and the amount of user time elapsed in generating the automaton).
5. Information about the synchronous product of the state space and the Büchi automaton constructed from the positive formula.
6. Model checking result information. In this case, the automaton cannot reach an "accepting cycle" regardless of the state of the state space in which the automaton could begin its execution. In other words, the random state space contains no states with an infinite path beginning from the state such that the Büchi automaton accepts the temporal interpretation of the path (the infinite sequence of state labels on the path).

7. The model checking process is repeated using the negated formula as input for the LTL-to-Büchi translator 'Implementation 0'.
8. `lbtt` performs the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17) using the model checking results computed for the positive and the negative formula. In this example, the result consistency check fails in 75 states of the state space. This implies that 'Implementation 0' failed to translate one (or both) of the formulas into a Büchi automaton correctly.

The output of phases 4—8 will be repeated for each implementation included in the tests. After this `lbtt` proceeds to the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16) and the Büchi automata intersection emptiness test (see Section 2.5 [Automata intersection emptiness check], page 18).

The model checking result cross-comparison test might result in the following output (shown in verbosity modes greater than 1):

```
Model checking result cross-comparison:
  result:
    failed (+)  0: 'Implementation 0', 1: 'Implementation 1'
```

Throughout all test failure reports, `lbtt` refers to the positive and negated formulas with the symbols '`+`' and '`-`', respectively. Therefore, the above message indicates that the model checking results obtained using 'Implementation 0' and 'Implementation 1' for the positive formula do not agree. A similar line will be shown for all pairs of implementations for which the test failed.

`lbtt` also reports if the model checking result cross-comparison could not be performed between a pair of implementations (for example, if one of the implementations failed to generate an automaton); in this case, the result of the test is '`N/A`'.

If using enumerated or randomly generated paths as state spaces, the model checking results are also compared against those given by `lbtt`'s internal model checking algorithm.

A similar convention is used to report failures in the Büchi automata intersection emptiness check. However, because this test is always performed on Büchi automata constructed from two complementary LTL formulas, a test failure report shows LTL formula information beside the name of the implementation used for generating the Büchi automaton from that formula. Note that the Büchi automata intersection emptiness check may fail on the automata constructed by the same implementation; in the following example, the check failed between the automata constructed by 'Implementation 0', and the automata constructed by 'Implementation 0' and 'Implementation 1' from the positive and negative formulas, respectively.

```
Büchi automata intersection emptiness check:
  result:
    failed      0: 'Implementation 0'
    failed (+)  0: 'Implementation 0', (-) 1: 'Implementation 1'
```

If using a log file (see ['--logfile' command line option], page 31), a summary of all testing errors will be written to the file using the output format specified above.

4.3 Test statistics

At the end of testing, `lbtt` outputs some simple statistics computed over all tests in verbosity modes 2 and above. If using an error log file (see ['--logfile' command line option], page 31), the statistics will be stored also in the log file. These statistics can be also accessed during interactive testing by using the internal command '`statistics`' (see ['`statistics`' command], page 48). In brief, the statistics include:

- Number of generated state spaces and the total number of states and transitions in them.
- Number of processed LTL formulas (not counting the negations of each formula). If using random formulas, `lbtt` also shows the overall distribution of each individual proposition, Boolean constant and logical or temporal operator in the sample of randomly generated formulas. Theoretically, in a large sample of random formulas, this distribution should correspond to the one that can be computed before testing by using the '`--showoperatordistribution`' command line option (see ['`--showoperatordistribution`' command line option], page 31).
- Automata statistics for each implementation:
  - number of generated Büchi automata and product automata
  - total and average numbers of states, transitions and acceptance sets in the generated Büchi/product automata, and
  - total and average time consumed in generating the Büchi automata.
- Number of times that each implementation failed to generate an acceptable automaton from an input formula.
- Number of failures in the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17) for each implementation.
- Number of result inconsistencies detected in pairwise comparison of the Büchi automata generated by different implementations. Depending on the model checking mode and which correctness tests are enabled, the output may include none, some or all of the following information:
  - Overall number of failures in the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16) for each pair of implementations.
  - Number of failures in the model checking result cross-comparison test in a single fixed state of each generated state space (called the "initial" state of the state space).

- Number of failures in the Büchi automata intersection emptiness check (see Section 2.5 [Automata intersection emptiness check], page 18) for each pair of implementations.

Note that the pairwise inconsistency results form a symmetric matrix (possibly shown in several parts), which means that the same information is repeated on both sides of the matrix diagonal.

Where applicable, the statistics are shown separately for positive, negative and all LTL formulas used in the tests.

5 Analyzing test results

This chapter documents how to use `lbtt`'s internal commands to analyze test results.

To use the internal commands, `lbtt` must be started in one of its interactive modes (see [Interactivity modes], page 21). Depending on the mode, `lbtt` may occasionally pause (for example, after each test round, or when a test failure is detected) between test rounds to wait for user input by showing a prompt of the form

```
** [Round 22 of 1000] >>
```

5.1 Command conventions

Commands are entered by typing a command name followed by any parameters for the command and then pressing ⟨ENTER⟩. The command names are case-sensitive. Each parameter should be separated from the command name and other parameters with white space.

Command names can be abbreviated to the shortest prefix that identifies the command unambiguously (for example, '`h`' could be used in place of the '`help`' command).

Some of the commands expect lists of implementation or state identifiers as parameters. The lists can be specified as comma-separated numbers (for example, '`8`') or intervals (for example, '`3-11`') with no white space between the commas and the numbers or intervals that belong to the same list. For example, assuming that the state space used in the current test round has at least 23 states, the command '`statespace -5,8,14-18,22-`' would display information about all state space states with an identifier less than or equal to 5, together with information about state 8, states 14 to 18 (inclusive) and all states with an identifier greater than or equal to 22. The '`*`' symbol can be used as a shorthand for all identifiers in the available range.

`lbtt` also recognizes the symbolic names of implementations (defined in the configuration file) in implementation identifier lists. The names can be used in place of the numeric identifiers. Quotes or the escape character ('`\`') should be used to handle white space in identifiers.

Some of the commands require a formula identifier as a parameter for choosing between a positive and a negative LTL formula. The formula identifier ('`+`' for positive formula, '`-`' for negative formula) must follow the command name as the first parameter for the command. If the formula identifier is omitted, the positive formula is assumed.

The output of most commands (excluding the test control commands, see Section 5.3 [Test control commands], page 45) can be redirected or appended to a file by ending the command line with '`>filename`' or '`>>filename`', respectively.

Optionally, the output can be handed over to an external program by ending the command line with '`| `*`command`*', where *command* is the command line used for invoking the external program. For example, the output of the (`lbtt`'s internal) command can be piped to a pager application if the entire output does not fit on the screen by itself. Using the pipe construct without specifying any internal command will simply invoke the external program.

5.2 Getting help

Use the '`help`' command to access on-line help. Typing '`help`' with no parameters shows a list of all available commands, together with general conventions for using the commands. The '`help`' command can be optionally given a command name as a parameter to access command-specific help.

In command-specific help, arguments in angle brackets (`<`, `>`) denote obligatory command parameters, while arguments in square brackets ([,]) are optional. A vertical bar (`|`) denotes selection between several alternatives. Arguments in double quotes should be entered literally (without the quotes themselves).

5.3 Test control commands

The following commands can be used to continue or abort testing, skip a number of test rounds, enable or disable implementations for testing, and change the verbosity of `lbtt`'s output messages.

'`continue` [*`number-of-rounds`*]'
: Continue testing. If no argument is given, testing will be interrupted again when mandated by the current interactivity mode (see [Interactivity modes], page 21). The optional argument *number-of-rounds* can be used to specify a number of rounds to run; testing is then interrupted again after the given number of test rounds (or in case of a new test failure if mandated by the current interactivity mode).

'`disable` [*`implementation-id-list`*]'
: Disable testing of a list of implementations (all implementations if no list of implementations is specified). `lbtt` will not include these implementations in the tests in subsequent test rounds. (See Section 5.1 [Command conventions], page 44, for the syntax used for the list of implementations.)

'`enable` [*`implementation-id-list`*]'
: Enable testing of a list of implementations.

'`quit`'
: Display test statistics (see Section 4.3 [Test statistics], page 42) over the test rounds performed and then abort testing.

'`skip` [*`number-of-rounds`*]'
: Skip a number of test rounds and then return to wait for further user input. If not explicitly specified, the number of rounds to skip defaults to 1. Use the '`--skip`' command line option (see

[The '`--skip`' command line option], page 31) to begin testing from another test round than 1.

'`verbosity` [*`verbosity-level`*]'
: Display or change the verbosity of `lbtt`'s output messages. If no argument is given, show the current verbosity level, otherwise change the verbosity setting to the given value. The argument must be an integer between 0 and 5 (inclusive). (The new value will take effect when testing is resumed.)

5.4 Data display commands

The following commands can be used to access test result information and to inspect the LTL formulas, Büchi automata and the state space used in the current test round.

'`algorithms`'
'`implementations`'
'`translators`'
: Show a list of implementations declared in the program configuration file and tell whether they are currently enabled for testing. The list also shows the numeric identifiers of the implementations.

'`buchi` ["+" `|` "-"] *`<implementation-id>`* [*`state-id-list`* `|` "dot"]'
: Display information about the structure of the Büchi automaton generated by the implementation *implementation-id* from the positive ('`+`') or negative ('`-`') LTL formula used in the current test round. The implementation identifier may be optionally followed by a list of state identifiers to display specific states of the automaton (see Section 5.1 [Command conventions], page 44, for details on how the list should be formatted), or the keyword '`dot`' to display the automaton in a format that can be given as input for the '`dot`' tool of the GraphViz graph visualization package [GViz] to obtain a graphical representation of the automaton.

'`evaluate` ["+" `|` "-"] [*`implementation-id-list`*] [*`state-id-list`*]'
: Display the model checking results for the positive ('`+`') or the negative ('`-`') formula computed using a given set of implementations for constructing a Büchi automaton from the formula. If no implementation list is specified, show the results for all implementations. The implementation identifier list may optionally be followed by a list of (state space) state identifiers to restrict the output to only a subset of all states. (See Section 5.1 [Command conventions], page 44, for more information about the format used for the lists.)

  This command can be used to look for states in which the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16) failed for a pair of implementations. These state identifiers can then be used as input for the '`resultanalysis`' command (see Section 5.5 [Failure analysis commands], page 48).

Note 1: Observe that the model checking results shown do not follow the "universal" semantics of LTL (common in model checking), by which a formula is usually considered to hold in a set of infinite paths beginning from a state only if *all* paths in the set are accepted by the Büchi automaton constructed from the formula to be model checked. Instead, `lbtt` will mark the result true if *any* of these paths is accepted by the automaton.

Note 2: If using random or enumerated paths as state spaces, `lbtt` accepts also the identifier '`lbtt`' in the implementation identifier list. This identifier can be used for accessing the model checking results computed using `lbtt`'s internal model checking algorithm for paths.

'`formula` ["`+`" `|` "`-`"] ["`normal`" `|` "`nnf`"]'

Display the positive ('`+`') or the negative ('`-`') LTL formula used for tests in the current test round either in the form in which it was generated ('`normal`' – the default) or in negation normal form ('`nnf`').

'`inconsistencies` [*`implementation-id-list`*]'

List the state space states in which the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17) failed for each implementation in the list (or all implementations if the list is omitted). See Section 5.1 [Command conventions], page 44, for information on formatting the list. The state identifiers can then be used as input for the '`consistencyanalysis`' command (see Section 5.5 [Failure analysis commands], page 48).

'`results` [*`implementation-id-list`*]'

Display test results (in the current test round) for each implementation in the list (or all implementations if the list is omitted). For more information about the output, see Section 4.2 [Test round messages], page 39; see Section 5.1 [Command conventions], page 44, for information on how to specify the implementations.

'`statespace` [*`state-id-list`* `|` "`dot`"]'

Display information about the structure of the state space used for model checking tests in the current test round. The optional *state-id-list* can be used to display only a part of the whole state space (see Section 5.1 [Command conventions], page 44, for information on formatting the state list). Alternatively, the '`dot`' keyword can be used to output the state space description in a format recognized by the '`dot`' tool of the GraphViz graph visualization package [GViz] that can be used to obtain a graphical representation of the state space.

'`statistics`'
: Display statistics computed over all test rounds performed since the program was started. This is the same information that `lbtt` normally outputs at the end of testing; see Section 4.3 [Test statistics], page 42, for more information about the output that is displayed.

5.5 Failure analysis commands

The first part of this section introduces the commands available for identifying an LTL-to-Büchi translator that caused a failure in one of the automata correctness tests. The second part describes the conventions that `lbtt` uses for justifying the result of the analysis.

5.5.1 Alphabetical list of failure analysis commands

'`buchianalysis` *`<implementation-id>`* *`<implementation-id>`*'
: Analyze a failure in the Büchi automata intersection emptiness check (see Section 2.5 [Automata intersection emptiness check], page 18). The two implementation identifiers select the Büchi automata for which to perform the analysis. The Büchi automata intersection emptiness check always involves automata constructed from the positive and the negative formulas used in the current test round. The first implementation identifier chooses an implementation that constructed an automaton from the positive formula, and the second identifier selects an implementation used for translating the negative formula into an automaton. (The identifiers can also be equal if one of the tested implementations failed the check against itself.)

  A failure in the Büchi automata intersection emptiness check implies that there exists an input sequence over subsets of atomic propositions that is accepted by both automata included in the analysis. `lbtt` examines the intersection of the automata to find a witness of such an input, checks whether this witness is a model of the positive formula, and tells which one of the automata is likely to be incorrect according to the following rules:

  - If the positive formula is found to hold in the witness, the automaton constructed from the negative formula is likely to contain an error.
  - If the witness is not a model for the positive formula, then the automaton constructed from the positive formula probably accepts the witness incorrectly.

'`consistencyanalysis <`*`implementation-id`*`>` [*`state-id`*]'

Analyze a failure in the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17). The *implementation-id* parameter chooses the implementation to analyze. In addition, the optional *state-id* parameter can be used to specify a state (in the state space) in which to perform the analysis (use the '`inconsistencies`' command, Section 5.4 [Data display commands], page 46, to see a list of all states in which the check failed). If the state identifier is omitted, `lbtt` will try to find a state where the check failed.

A failure in the model checking result consistency check implies the existence of a witness (i.e., a path in the state space used for the tests in the current test round) whose temporal interpretation is not accepted by either of two automata constructed from two complementary LTL formulas. In the analysis, `lbtt` finds such a witness, checks separately whether it is a model of the positive formula, and then tells which one of the automata seems to reject the witness incorrectly.

'`resultanalysis` ["`+`" `|` "`-`"] `<`*`implementation-id`*`>` `<`*`implementation-id`*`>` [*`state-id`*]'

Analyze a failure in the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16) between two implementations on either the positive ('`+`') or the negative ('`-`') LTL formula used in the current test round. The implementation identifiers can be optionally followed by an identifier of a state in the state space to specify a state in which the analysis should be performed. (Suitable state identifiers can be found by looking for inconsistencies in the model checking results accessible with the '`evaluate`' command, Section 5.4 [Data display commands], page 46; by omitting the state identifier, `lbtt` will try to find a state in which the model checking result comparison failed between the implementations.)

If using randomly generated or enumerated paths as state spaces, `lbtt` also accepts the identifier '`lbtt`' in place of either of the implementation identifiers. This instructs `lbtt` to perform the analysis against `lbtt`'s internal model checking algorithm.

A failure in the model checking result cross-comparison test suggests that the state space used in the current test round contains a path which is accepted by one, but rejected by another automaton constructed from the same LTL formula. To determine which one of these automata accepts or rejects the input incorrectly, `lbtt` finds a witness path giving contradictory model checking results, model checks the formula separately in the witness, and tells which one of the automata seems to accept or reject the witness incorrectly.

5.5.2 Witnesses, proofs and refutations

All of the above analysis commands use `lbtt`'s internal model checking algorithm to determine which one of the two automata involved in each test is incorrect by checking whether an LTL formula holds in a witness path extracted from the state space used in the current test round or from the intersection of two Büchi automata. The witness path is a sequence of consecutive states that ends in a loop, and is represented in two parts as an initial "prefix" (which may be empty) and a "cycle" that is considered to repeat itself indefinitely. The witness might, for example, look as follows:

```
Execution M:
  prefix:
    s3 {p0,p2,p4} --> s4
  cycle:
    s4 {p1,p3} --> s5
    s5 {p3} --> s6
    s6 {p1,p2,p3} --> s7
    s7 {p3,p4} --> s8
    s8 {p1} --> s9
    s9 {} --> s2
    s2 {} --> s3
    s3 {p0,p2,p4} --> s4
```

In this case, the witness (or "execution" as displayed in the output) M consists of a single-state prefix followed by a cycle of eight states. The atomic propositions that hold in each state are also shown in the output.

(The witness can be considered a small state space $M = \langle S, \rho, \mathcal{L} \rangle$ following the definition in Section A.3 [State spaces], page 65; in the example above, $S = \{s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\}$, $\rho = \{(s_2, s_3), (s_3, s_4), (s_4, s_5), (s_5, s_6), (s_6, s_7), (s_7, s_8), (s_8, s_9), (s_9, s_2)\}$, $\mathcal{L}(s_2) = \mathcal{L}(s_9) = \emptyset$, $\mathcal{L}(s_3) = \{p_0, p_2, p_4\}$, $\mathcal{L}(s_4) = \{p_1, p_3\}$, $\mathcal{L}(s_5) = \{p_3\}$, $\mathcal{L}(s_6) = \{p_1, p_2, p_3\}$, $\mathcal{L}(s_7) = \{p_3, p_4\}$, and $\mathcal{L}(s_8) = \{p_1\}$.)

In the model checking result cross-comparison test and the model checking result consistency check, the witness is an actual path extracted from the state space used for the tests in the current test round. In this case, the state identifiers correspond to the states of the state space, and can be accessed with the '`statespace` [*`state-id`*]' command (see Section 5.4 [Data display commands], page 46).

To justify the result of the analysis, `lbtt` also displays a proof or a refutation for the LTL formula in the witness. The proof or refutation is constructed by a recursive examination of the subformulas of the (positive or negative) formula used in the current test round according to the semantics of LTL and might look as follows:

```
Analysis of the formula in the execution:
  M,<s3, ...> |/= ((X p0 U ! p4) <-> p0) :
  +-> M,<s3, ...> |/= (X p0 U ! p4) :
  |   +-> M,<s3, ...> |/= X p0 :
  |   |   +-> s3 --> s4
  |   |   +-> M,<s4, ...> |/= p0
  |   +-> M,<s3, ...> |/= ! p4 :
  |       +-> M,<s3, ...> |== p4
  +-> M,<s3, ...> |== p0
```

The proof (or refutation) can be considered a tree of statements of the form '`M,<s, ...> |==` *`subformula`*' or '`M,<s, ...> |/=` *`subformula`*'. Here, the symbol '`|==`' is used to denote that the formula *subformula* holds in the (infinite) subsequence beginning at state '`s`' of the witness, and the relational symbol '`|/=`' denotes the opposite. The children of each proof tree node give justification for the claim in their parent node; the children might be further expanded if the claims in them do not directly follow from the definition of $\mathcal{L}$. In the presence of temporal operators, the proofs may need to be based also on the structural properties of M. These are shown as statements of the form '`sn --> sm`' to indicate that M contains a transition from the state '`sn`' to the state '`sm`' (and, since the states in M are connected into a non-branching sequence, that this is the *only* transition originating from '`sn`').

In the above example, `lbtt` claims that the formula '`((X p0 U ! p4) <-> p0)`' does not hold in the witness presented earlier in this section, and that this follows (by the semantics of logical equivalence) from the claims that the subformula '`(X p0 U ! p4)`' does not hold, but the subformula '`p0`' holds in this witness. '`(X p0 U ! p4)`' does not hold in the witness, because neither '`X p0`' nor '`! p4`' holds in the first state of the witness ($p_4 \in \mathcal{L}(s_3)$, and $p_0 \notin \mathcal{L}(s_4)$, where s_4 is the only successor of s_3). On the other hand, '`p0`' holds in the witness because of the fact that $p_0 \in \mathcal{L}(s_3)$.

6 Interfacing with `lbtt`

The output generated by `lbtt` consists of textual messages and an optional error log file (see ['`--logfile`' command line option], page 31). The format of the output messages is determined by the verbosity mode; for more information, see Chapter 4 [Interpreting the output], page 38. In addition, `lbtt` returns one of the following three values as its exit status upon normal termination:

- 0: `lbtt` exited successfully; no errors were detected during testing.
- 1: `lbtt` exited successfully; errors were detected during testing.
- 2: An error was found when reading the program configuration or when processing the command line options.
- 3: `lbtt` exited due to an unrecoverable internal error.

The rest of this chapter gives the details on how to use `lbtt` for testing LTL-to-Büchi translation algorithm implementations that are not supported by the basic distribution. (See Section 6.4 [The lbtt-translate utility], page 57 for information on how to connect several publicly available LTL-to-Büchi translator implementations to `lbtt`.)

6.1 Requirements for translator executables

`lbtt` assumes each tested LTL-to-Büchi translator to be accessible by running an executable file which should read in an LTL formula from a file, convert it into a Büchi automaton and then write the automaton into another file. For this purpose, the executable should support the following command line interface:

```
path-to-program parameters input-file output-file
```

where *path-to-program* is the full name (and location) of the executable, *parameters* are any optional parameters that might be needed for running the executable, and *input-file* and *output-file* are two file names. The translator executable should read its input (an LTL formula) from *input-file* and write its output (a Büchi automaton) into *output-file* (without removing the input file); see the following two sections for a description on how these files should be formatted.

The translator executable should always create an output file and then return with a zero exit status in case no errors occur during the translation. `lbtt` interprets a missing output file or a nonzero exit status as an error and will not in this case try to run any tests, even if an automaton were successfully saved in an output file.

To start testing the translator, add a new '`Translator`' section for it into `lbtt`'s configuration file (see Section 3.1 [Configuration file], page 19), for example

```
Translator
{
  Name = "LTL-to-Büchi translator"
  Path = /home/lbtt-user/bin/ltl-to-buchi-translator
  Parameters = "-x -y -z"
  Enabled = Yes
}
```

6.2 Input file format for LTL formulas

`lbtt` passes each LTL formula to each LTL-to-Büchi translator in a file containing an LTL formula in a prefix notation followed by a single newline. The precise grammar for the LTL formulas (in a BNF-style notation) is as follows:

```
formula  ::=  ‘t’
                                                // “true”
           |  ‘f’
                                                // “false”
           |  ‘p’[0—9]+
                                                // atomic proposition with
                                                // a nonnegative integer
                                                // identifier
           |  ‘!’ sp formula
                                                // negation
           |  ‘X’ sp formula
                                                // “next time”
           |  ‘F’ sp formula
                                                // “finally”
           |  ‘G’ sp formula
                                                // “globally”
           |  ‘&’ sp formula sp formula
                                                // conjunction
           |  ‘|’ sp formula sp formula
                                                // disjunction
           |  ‘i’ sp formula sp formula
                                                // implication
           |  ‘e’ sp formula sp formula
                                                // equivalence
           |  ‘^’ sp formula sp formula
                                                // exclusive or
           |  ‘U’ sp formula sp formula
                                                // “(strong) until”
           |  ‘V’ sp formula sp formula
                                                // “(weak) release”
           |  ‘W’ sp formula sp formula
                                                // “weak until”
           |  ‘M’ sp formula sp formula
                                                // “strong release”
           |  ‘B’ sp formula sp formula
                                                // “before”
```

(The quoted characters denote the characters themselves; *sp* denotes any nonempty string of white space. Lines containing a // are comments and are not part of the grammar. All atomic propositions in the formula have a nonnegative numeric identifier.)

For example, the LTL formula $(p_0 \mathbf{U}\ p_1) \rightarrow (\mathbf{F}\ \mathbf{G}(\neg p_2 \leftrightarrow p_3))$ would be expressed in the form

```
i U p0 p1 F G e ! p2 p3
```

in an output file.

If your translator does not support all of the above operators, edit the configuration file (see Section 3.1 [Configuration file], page 19) or use the command line options (see Section 3.2 [Command line options], page 29) to prevent `lbtt` from generating random LTL formulas with these operators.

6.3 Output file format for automata

`lbtt` expects the Büchi automata generated by each LTL-to-Büchi translator implementation to be in the format specified below. The format encodes a generalized Büchi automaton (a Büchi automaton with zero or more acceptance conditions) with a single initial state and labels (guards) on transitions. For the full formal definition and examples on how to reduce other definitions into the one used by `lbtt`, see Appendix A [Definitions], page 63.

The output file generated by the translator should contain an *automaton* described using the following grammar (as before, quoted characters denote the characters themselves, *sp* denotes any nonempty string of white space, lines containing a // are comments that are not part of the grammar, and '`\n`' corresponds to the newline character).

```
automaton  ::=  num-states sp cond-specifier state-list

num-states  ::=  [0—9]+

cond-specifier  ::=  [0—9]+[st]*

state-list  ::=  state-list sp state
             |   // empty
```

The automaton description begins with a nonnegative number that gives the number of states in the automaton. If the number of states is 0, the automaton will not accept any input. If the number is positive, it should be followed by a *cond-specifier* that determines the number and placement of acceptance conditions in the automaton. If the number of acceptance conditions is 0, the automaton accepts an input word if and only if it has a run on that word according to the definition given in the Appendix (see Appendix A [Definitions], page 63).

The placement of acceptance conditions is specified by concatenating a string formed from the symbols '`s`' and '`t`' to the number of acceptance conditions (with no white space in between). The interpretation of this string is as follows:

- If the string is empty or does not include the symbol '`t`', the acceptance conditions of the automaton are placed exclusively on its states. (This alternative corresponds to the definition supported by `lbtt` 1.0.x.)
- If the string is nonempty but does not include the symbol '`s`', the automaton has acceptance conditions exclusively on its transitions.
- Otherwise, the automaton has acceptance conditions on both states and transitions.

The *cond-specifier* is followed by a list of the descriptions of states in the automaton. The format of this list is affected by the choice of the placement of the acceptance conditions. More precisely, the choice affects the interpretation of the *cond-list* nonterminal symbol in the following fragment of the grammar: we indicate this by prefixing the nonterminal with either "<s>" or "<t>" to denote that the list (together with its terminating '`-1`') should be omitted in automata that do not associate acceptance conditions with states or transitions, respectively.

```
state  ::=  state-id sp initial? <s>cond-list transition-list

state-id  ::=  [0—9]+

initial?  ::=  '0'  |  '1'

cond-list  ::=  sp acceptance-condition-id cond-list
             |  sp '-1'

acceptance-condition-id  ::=  [0—9]+

transition-list  ::=  sp transition transition-list
                   |  sp '-1'

transition  ::=  state-id <t>cond-list sp guard-formula '\n'

guard-formula  ::=  't'
                                           // "true"
                 |  'f'
                                           // "false"
                 |  'p'[0—9]+
                                           // atomic proposition
                 |  '!' sp guard-formula
                                           // negation
                 |  '&' sp guard-formula sp guard-formula
                                           // conjunction
                 |  '|' sp guard-formula sp guard-formula
                                           // disjunction
                 |  'i' sp guard-formula sp guard-formula
                                           // implication
                 |  'e' sp guard-formula sp guard-formula
                                           // equivalence
                 |  '^' sp guard-formula sp guard-formula
                                           // exclusive or
```

The description of each state begins with a numeric state identifier, which can be any nonnegative integer. The state identifier should be followed by a number telling whether the state is initial (‘`1`’ if yes). The automaton should have exactly one initial state. If the automaton has acceptance conditions associated with its states, this number should then be followed by a list of acceptance condition identifiers separated by white space. This list should be terminated with ‘`-1`’.

The state description should be followed by the list of transitions starting from the state (terminated again by ‘`-1`’). Each transition consists of a state identifier (the target state of the transition), a list of acceptance condition identifiers (if the automaton has acceptance conditions on transitions), and a propositional formula[1] that encodes the symbols of the alphabet 2^{AP} (where AP is a finite set of atomic propositions) on which the automaton is allowed to take the transition. The propositional formula should be terminated with a newline.

The state and acceptance condition identifiers need not be successive, and the states or acceptance conditions can be listed in any order. The only restrictions are that the identifiers of different states and acceptance conditions should be unique and that the total number of different identifiers should equal *num-states* or *num-conds*, respectively. (The same identifiers can be shared between states and acceptance conditions, however.)

Note that the output file should always contain a valid automaton description if the LTL-to-Büchi translation was successful, even in the case that the resulting automaton is empty (`lbtt` interprets a missing automaton description file as an error).

The following examples illustrate the file format. The first example gives the description of an automaton with acceptance conditions on states. Note that in this case the ‘`s`’ is optional for describing the placement of acceptance conditions; therefore, the automaton files used with `lbtt` 1.0.x are upwards compatible with newer versions of the tool (provided that each guard of a transition is terminated by a newline).

```
6 2s          // an automaton with six states and two acc. conditions on states
0 1 -1        // state 0: initial state, no acceptance conditions
2 p1          //     transition to state 2, guard 'p1'
5 p2          //     transition to state 5, guard 'p2'
15 p3         //     transition to state 15, guard 'p3'
-1            // end of state 0
2 0 1 -1      // state 2: non-initial state, acceptance condition 1
2 p1          //     transition to state 2, guard 'p1'
5 p2          //     transition to state 5, guard 'p2'
15 p3         //     transition to state 15, guard 'p3'
-1            // end of state 2
```

[1] Although not described formally in the grammar, the guard formulas can be specified in any of the formats `lbtt` supports in its formula input files (see [‘`--formulafile`’ command line option], page 30). Note that the formula always needs to be terminated with a newline, though.

```
5 0 0 -1       // state 5: non-initial state, acceptance condition 0
5 p2           //     transition to state 5, guard 'p2'
8 & p1 p2      //     transition to state 8, guard 'p1 /\ p2'
12 & p1 p3     //     transition to state 12, guard 'p1 /\ p3'
15 p3          //     transition to state 15, guard 'p3'
-1             // end of state 5
8 0 0 -1       // state 8: non-initial state, acceptance condition 0
5 p2           //     transition to state 5, guard 'p2'
8 & p1 p2      //     transition to state 8, guard 'p1 /\ p2'
12 & p1 p3     //     transition to state 12, guard 'p1 /\ p3'
15 p3          //     transition to state 15, guard 'p3'
-1             // end of state 8
15 0 1 0 -1    // state 15: non-initial state, acceptance conditions 1 and 0
2 p1           //     transition to state 2, guard 'p1'
5 p2           //     transition to state 5, guard 'p2'
15 p3          //     transition to state 15, guard 'p3'
-1             // end of state 15
12 0 1 0 -1    // state 12: non-initial state, acceptance conditions 1 and 0
2 p1           //     transition to state 2, guard 'p1'
5 p2           //     transition to state 5, guard 'p2'
15 p3          //     transition to state 15, guard 'p3'
-1             // end of state 12
```

The following example illustrates an automaton in which acceptance conditions are placed on transitions.

```
4 3t           // four states, three acceptance conditions on transitions
5 0            // state 5: non-initial state
84 0 -1 p1     //     transition to state 84, condition 0, guard 'p1'
27 0 -1 & p1 ! p2 // tr. to state 27, condition 0, guard 'p1 /\ ! p2'
5 -1 t         //     transition to state 5, no conditions, guard 'true'
-1             // end of state 5
84 1           // state 84: initial state
5 1 -1 t       //     transition to state 5, condition 1, guard 'true'
27 0 -1 p1     //     transition to state 27, condition 0, guard 'p1'
-1             // end of state 84
49 0           // state 49: non-initial state
5 -1 t         //     transition to state 5, no conditions, guard 'true'
49 1 4 -1 & p1 ! p2 // tr. to state 49, conds. 1 and 4, guard 'p1 /\ ! p2'
84 -1 p1       //     transition to state 84, no conditions, guard 'p1'
-1             // end of state 49
27 0           // state 27: non-initial state
49 -1 & p1 p3 //     transition to state 49, no conds., guard 'p1 /\ p3'
-1             // end of state 27
```

Automata with acceptance conditions on both states and transitions can be specified using a combination of the above two formats, that is, by using 'st' as the acceptance condition placement specifier and including a list of acceptance conditions both after the value determining the initialness of a state, and after the identifier of the target state of each transition.

6.4 The `lbtt-translate` utility

The `lbtt` source distribution includes a small utility which can be used as a common interface for the following publicly available LTL-to-Büchi translator algorithm implementations:

- `lbt` — an LTL-to-Büchi translation algorithm implementation based on the algorithm described in [GPVW95]. See `<http://www.tcs.hut.fi/Software/maria/tools/lbt/>` for more information, including the source code of the implementation.
- Spin [Hol97] — a model checking tool that includes a module for translating LTL formulas into Büchi automata originally based on the algorithm presented in [GPVW95]. See `<http://spinroot.com/spin/whatispin.html>` for more information.
- Spot [DP04] — a model checking library that includes a module for translating LTL formulas into Büchi automata incorporating optimization techniques from several different sources. See `<http://spot.lip6.fr/>` for more information.

To use `lbtt` for testing the LTL-to-Büchi translators included in these tools, you should first install the tool normally by following its installation instructions. Then add the following '`Translator`' section in `lbtt`'s configuration file:

```
Translator
{
  Name = "[name for the implementation]"
  Path = "[path to lbtt-translate]"
  Parameters = "[implementation selector] [path to executable]"
  Enabled = Yes
}
```

where [*path to* `lbtt-translate`] contains the complete path and file name of the `lbtt-translate` tool executable, [*implementation selector*] is either of the options '`--lbt`' or '`--spin`', and [*path to executable*] is the full path of the tool executable. The names of these executables are usually (assuming a normal installation) `lbt` and `spin`, respectively.

Note: These implementations may not have built-in support for all of the LTL formula operators available for generating random LTL formulas with `lbtt`. See the documentation of each translator for information about which operators are supported, and then change the parameters in `lbtt`'s configuration file accordingly to disable the unsupported operators (or instruct `lbtt` to read the formulas from an external source by invoking `lbtt` with the ['`--formulafile`' command line option], page 30).

The `lbtt-translate` utility can also be invoked directly from the shell to translate an LTL formula into a Büchi automaton using either of the above translators. Use the command `lbtt-translate --help` to see a short summary of available options.

References

[CGP99] E. Clarke Jr., O. Grumberg and D. Peled. Model checking. The MIT Press, 1999.

[Cou99] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM'99), volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 253—271. Springer-Verlag, 1999.

[DGV99] M. Daniele, F. Giunchiglia and M. Y. Vardi. Improved automata generation for linear temporal logic. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 249—260. Springer-Verlag, 1999.

[DP04] A. Duret-Lutz and D. Poitrenaud. SPOT: An Extensible Model Checking Library Using Transition-Based Generalized Büchi Automata. In *Proceedings of the 12th IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004)*, pages 76–83. IEEE Computer Society Press, 2004.

[EH00] K. Etessami and G. Holzmann. Optimizing Büchi automata. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR 2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153—167. Springer-Verlag, 2000.

[Ete99] K. Etessami. Stutter-invariant languages, omega-automata, and temporal logic. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 236—248. Springer-Verlag, 1999.

[Ete02] K. Etessami. A hierarchy of polynomial-time computable simulations for automata. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR 2002)*, volume 2421 of *Lecture Notes in Computer Science*, pages 131—144. Springer-Verlag, 2002.

[EWS01] K. Etessami, Th. Wilke and R. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming (ICALP 2001)*, volume 2076 of *Lecture Notes in Computer Science*, pages 694—707. Springer-Verlag, 2001.

[Fri03] C. Fritz. Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata.

In *Proceedings of the 8th International Conference on Implementation and Application of Automata (CIAA 2003)*, volume 2759 of *Lecture Notes in Computer Science*, pages 35—48. Springer-Verlag, 2003.

[GO01] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53—65. Springer-Verlag, 2001.

[GO03] P. Gastin and D. Oddoux. LTL with past and two-way weak alternating automata. In *Proceedings of the 28th International Symposium on Mathematical Foundations of Computer Science (MFCS 2003)*, volume 2747 of *Lecture Notes in Computer Science*, pages 439—448. Springer-Verlag, 2003.

[Gei01] M. C. W. Geilen. On the construction of monitors for temporal logic properties. *Electronic Notes for Theoretical Computer Science*, 55(2), 2001.

[GPVW95]
R. Gerth, D. Peled, M. Y. Vardi and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing, and Verification (PSTV'95)*, pages 3—18. Chapman & Hall, 1995.

[GL02] D. Giannakopoulou and F. Lerda. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *Proceedings of the 22nd IFIP WG6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, volume 2529 of *Lecture Notes in Computer Science*, pages 308—326. Springer-Verlag, 2002.

[GSB02] S. Gurumurthy, F. Somenzi and R. Bloem. Fair simulation minimization. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, volume 2404 of *Lecture Notes in Computer Science*, pages 610—624. Springer-Verlag, 2002.

[GViz] GraphViz - open source graph drawing software. See `<http://www.research.att.com/sw/tools/graphviz/>`.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279—295, 1997.

[Isl94] A. Isli. Mapping an LPTL formula into a Büchi alternating automaton accepting its models. In *Temporal Logic: Proceedings of the ICTL Workshop*, pages 85—90. Research Report MPI-I-94-230, Max-Planck-Institut für Informatik, 1994.

[Lat03] T. Latvala. Efficient model checking of safety properties. In *Proceedings of the 10th Spin Workshop on Model Checking of Software (SPIN 2003)*, volume 2648 of *Lecture Notes in Computer Science*, pages 74—88. Springer-Verlag, 2003.

[Sch01] K. Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2001)*, volume 2250 of *Lecture Notes in Computer Science*, pages 39—54. Springer-Verlag, 2001.

[ST03] R. Sebastiani and S. Tonetta. "More deterministic" vs. "smaller" Büchi automata for efficient LTL model checking. In *Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003)*, volume 2860 of *Lecture Notes in Computer Science*, pages 126—140. Springer-Verlag, 2003.

[SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 247—263. Springer-Verlag, 2000.

[Tau00] H. Tauriainen. Automated testing of Büchi automata translators for linear temporal logic. Research report A66, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland, 2000. Available on the WWW at `<http://www.tcs.hut.fi/Publications/info/bibdb.HUT-TCS-A66.shtml>`.

[TH02] H. Tauriainen and K. Heljanko. Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer (STTT)* 4(1):57—70, 2002.

[Thi02] X. Thirioux. Simple and efficient translation from LTL formulas to B"uchi automata. *Electronic Notes in Theoretical Computer Science*, 66(2), 2002.

[Var96] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238—265. Springer-Verlag, 1996.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First IEEE Symposium on Logic in Computer Science (LICS'86)*, pages 332—344. IEEE Computer Society Press, 1986.

[Wol01] P. Wolper. Constructing automata from temporal logic formulas: A tutorial. In *Lectures on Formal Methods and Performance Analysis: First EEF/Euro Summer School on Trends in Computer Science, Revised Lectures*, volume 2090 of *Lecture Notes in Computer Science*, pages 261—277. Springer-Verlag, 2001.

Appendix A Definitions

This appendix reviews the formal definitions of the objects that `lbtt` manipulates.

A.1 LTL formulas

`lbtt` uses the traditional definition for propositional linear temporal logic. Let AP be a finite set of atomic propositions. The set of propositional linear temporal logic formulas is defined inductively as follows:

- All atomic propositions in AP and the Boolean constant $\textsc{True}$ are LTL formulas.
- If φ and ψ are LTL formulas, then $\neg\varphi, \mathbf{X}\varphi, (\varphi \lor \psi)$ and $(\varphi\ \mathbf{U}\ \psi)$ are LTL formulas.

The semantics of linear temporal logic (i.e., a satisfiability relation, denoted by $\models$) is defined over infinite sequences $\xi = \langle y_0, y_1, y_2, \ldots\rangle \in (2^{AP})^\omega$ over subsets of AP as follows:

- $\xi \models \textsc{True}$ for all sequences ξ.
- $\xi \models p \in AP$ if and only if $p \in y_0$, the first element of the sequence ξ.
- $\xi \models \neg\varphi$ if and only if it is not the case that $\xi \models \varphi$.
- $\xi \models \varphi \lor \psi$ if and only if $\xi \models \varphi$ or $\xi \models \psi$.
- $\xi \models \mathbf{X}\varphi$ if and only if $\langle y_1, y_2, y_3, \ldots\rangle \models \varphi$.
- $\xi \models \varphi\ \mathbf{U}\ \psi$ if and only if there exists an $i \geq 0$ such that $\langle y_i, y_{i+1}, y_{i+2}, \ldots\rangle \models \psi$ and for all $0 \leq j < i, \langle y_j, y_{j+1}, y_{j+2}, \ldots\rangle \models \varphi$.

`lbtt` also supports the following operators and Boolean constants, the definitions of which can be given in terms of the previously defined operators:

- "false": $\textsc{False} \equiv_{\text{def}} \neg\,\textsc{True}$
- logical conjunction: $(\varphi \land \psi) \equiv_{\text{def}} \neg(\neg\varphi \lor \neg\psi)$
- logical implication: $(\varphi \rightarrow \psi) \equiv_{\text{def}} (\neg\varphi \lor \psi)$
- logical equivalence: $(\varphi \leftrightarrow \psi) \equiv_{\text{def}} ((\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi))$
- logical "exclusive or": $(\varphi \oplus \psi) \equiv_{\text{def}} \neg(\varphi \leftrightarrow \psi)$
- temporal "finally": $\mathbf{F}\varphi \equiv_{\text{def}} (\textsc{True}\ \mathbf{U}\ \varphi)$
- temporal "globally": $\mathbf{G}\varphi \equiv_{\text{def}} \neg\mathbf{F}\neg\varphi$
- temporal "(weak) release": $(\varphi\ \mathbf{V}\ \psi) \equiv_{\text{def}} \neg(\neg\varphi\ \mathbf{U}\ \neg\psi)$
- temporal "weak until": $(\varphi\ \mathbf{W}\ \psi) \equiv_{\text{def}} ((\varphi\ \mathbf{U}\ \psi) \lor \mathbf{G}\varphi)$
- temporal "strong release": $(\varphi\ \mathbf{M}\ \psi) \equiv_{\text{def}} ((\varphi\ \mathbf{V}\ \psi) \land \mathbf{F}\varphi)$
- temporal "before": $(\varphi\ \mathbf{B}\ \psi) \equiv_{\text{def}} \neg(\neg\varphi\ \mathbf{U}\ \psi)$

A.2 Generalized automata

`lbtt` uses internally finite-state automata on infinite words (Büchi automata) over the alphabet 2^{AP} (where AP is a finite set of atomic propositions) with one initial state, labels on transitions and zero or more acceptance conditions.

A.2.1 Formal definition of generalized automata

Formally, a generalized Büchi automaton can be represented as a tuple[1] $\langle \Sigma, Q, \Delta, q_I, \mathcal{F}, \lambda \rangle$, where

- Σ is the finite *alphabet* ($\Sigma = 2^{AP}$ in this case),
- Q is the finite set of *states*,
- $\Delta \subseteq Q \times 2^{\Sigma} \times 2^{\mathcal{F}} \times Q$ is the set of *transitions* (each of which consists of four components called the *start state*, the *guard*, the *acceptance component*, and the *target state*, respectively),
- q_I is the *initial state*,
- $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ (for some finite n) is the set of *acceptance conditions* (a "nongeneralized" Büchi automaton has exactly one acceptance condition), and
- $\lambda : Q \to 2^{\mathcal{F}}$ is a *labeling function* that associates each state of the automaton with a set of acceptance conditions.

A *run* of a Büchi automaton on an infinite sequence $\langle x_0, x_1, x_2, \ldots \rangle \in (2^{AP})^{\omega}$ over the alphabet 2^{AP} is an infinite sequence of pairs of states and transitions $\langle (q_0, t_0), (q_1, t_1), (q_2, t_2), \ldots \rangle \in (Q \times \Delta)^{\omega}$ such that $q_0 = q_I$ and for all $i \geq 0$, $t_i = \langle q_i, X_i, Y_i, q_{i+1} \rangle \in \Delta$ such that $x_i \in X_i$. (Because the relation Δ is not necessarily a function from $Q \times 2^{\Sigma} \times 2^{\mathcal{F}}$ to Q, the automaton may have many runs on the same input.)

A run $\langle (q_0, t_0), (q_1, t_1), (q_2, t_2), \ldots \rangle$ (where $t_i = \langle q_i, X_i, Y_i, q_{i+1} \rangle \in \Delta$ for all i) is *accepting* if and only if additionally, for each acceptance condition $f \in \mathcal{F}$, $f \in \lambda(q_i)$ or $f \in Y_i$ for infinitely many i.

The automaton *accepts* an infinite sequence $\langle x_0, x_1, x_2, \ldots \rangle \in 2^{AP}$ if and only if the automaton has at least one accepting run on this sequence.

A.2.2 Transition label encoding

When working with automata on words over the alphabet 2^{AP}, the guards of transitions can be expressed as propositional formulas by identifying a set of symbols from this alphabet with the set of models of a propositional formula. A transition can then be seen as a rule "if in state q_i and the

[1] This definition differs from those commonly found in the literature by specifying the acceptance conditions in terms of a separate set that is independent of the other components of the automaton, together with an explicit labeling function for the states. This is to allow the definition to correspond more accurately to the automata that can be described in input files.

next input symbol x_i is a model of the propositional formula guarding the transition, the automaton can move to state q_{i+1}". In the context of Büchi automata constructed from LTL formulas, this often allows for a compact representation for the transitions.

A.2.3 Converting between equivalent definitions

Many LTL-to-Büchi translation algorithms presented in the literature (for example, [GPVW95]) are based on a slightly different definition for generalized Büchi automata, where the automata can have several initial states, acceptance is determined using a family of sets of states, and the guards of transitions are replaced with an additional state labeling that associates a set of LTL formulas with each state. These automata can easily be described using the above definition through the following steps:

1. Add a new state (associated with an empty set of LTL formulas) into the automaton and add transitions from it to each initial state of the original automaton. Make the new state the (only) initial state of the automaton.
2. For each state of the (modified) automaton, construct a conjunction of all propositional constraints (all formulas with no temporal operators) associated with the state and make the conjunction the guard of each transition coming into the state (the acceptance component of each transition remains empty). Then remove the association between states and sets of formulas.
3. If $Q_1, Q_2, \ldots, Q_k \in 2^Q$ are the sets of states determining acceptance in the original automaton, let $f_i = Q_i$ for all $1 \leq i \leq k$, and let $\lambda(q) = \{f_i \mid q \in f_i\}$ for all states q.

A.3 State spaces

`lbtt` uses randomly generated state spaces in the model checking result cross-comparison test (see Section 2.3 [Model checking result cross-comparison test], page 16) and the model checking result consistency check (see Section 2.4 [Model checking result consistency check], page 17). Formally, the state spaces are (finite) Kripke structures with a total transition relation, i.e., directed graphs with a set of atomic propositions attached to each state, with each state having at least one immediate successor (which may be the state itself). The precise definition is as follows (as before, let AP be a finite set of atomic propositions).

A state space can be represented as a tuple $\langle S, \rho, \mathcal{L} \rangle$, where

- S is the finite set of *states*,
- $\rho \subseteq S \times S$ is the *transition relation*, and
- $\mathcal{L} : S \rightarrow 2^{AP}$ is the *labeling function* which maps each state to a set of atomic propositions that hold in the state.

Configuration file option index

Command line option index

User command index

Concept index