

Tutorial on Blade Model Checker

Contents

- [How to get it work](#)
 - [Supported platforms](#)
 - [Prerequisites](#)
 - [Run it](#)
- [Input](#)
 - [Control flow automata](#)
 - [Expressions](#)
 - [Predicates](#)
 - [Formulae](#)
 - [Assign and assume edges](#)
 - [Error node and entry nodes](#)
 - [Examples](#)
- [Output](#)
 - [Counterexamples](#)
 - [Statistics on runtime](#)
 - [Statistics on storage usage](#)
- [Limitation](#)
 - [Front-end](#)
 - [Pointers](#)
 - [Recursive function call](#)
 - [Non-termination](#)
- [Known problems](#)
 - [Blade runs much slower on Windows than on Linux](#)
 - [Blade may fail to recognize the input file](#)
- [Appendix](#)
 - [Hijack the communication between BLAST and Simplify](#)
 - [Count the maximum complexity of the formulae of BLAST](#)

How to get it work

1. Supported platforms

Currently Blade supports Windows, and x86-Linux. Binary download is available at [here](#). Source code will be available later.

2. Prerequisites

Blade depends on [Simplify](#), the theorem-prover developed at [Compaq](#). Please download and install [Simplify](#) first.

3. Run it

To get started with Blade,

- [Download the binary package](#) and unpack it into a desired directory
- Start a console and change to the directory where the executable locates
- Run Blade with the following command

```
> blade
```

- If you are on a Linux box, maybe the following is a safer choice

```
> ./blade
```

If everything is OK, Blade will print out the following message and terminate.

```
Usage: blade filename simplify
```

As you may have figured out, Blade takes two command line arguments. The first one, `filename`, is the name of the input file which is supposed to contain a [control flow automata](#). The input format is documented in Section [Input](#). The second argument, `simplify`, is the path pointing to the executable of the [Simplify](#) theorem-prover. If you have properly set up the environment variables to make [Simplify](#) executable under any directory, then simply put "Simplify" as the second argument. You should have a sample input file "a.cfa" located at the current directory. So the following command

```
> blade a.cfa Simplify
```

would probably work, which will print out the following output.

```
no counter example found
Statistics:
On Simplify: 1) Complexity vs. No. Call; 2) Formula vs. No. Reuse (Saved Call to Simplify)

On Slice Segments: 1) Slice Segment vs. No. Reuse (Saved Backward Explore)

On Predicates: 1) Predicate vs. No. Reuse (Saved Call to Simplify)

On Proof Slices Tree: 1) Proof Slice Tree; 2) Storage; 3) Reuse
[ Node ID:0 # Pred:0 ]
[ Node ID:1 # Pred:0 ]
# Predicates:0 # Nodes:2 Ratio:0
# At Slices:0 # Nodes:2 Ratio:0
# All Query:0 # Reuse Query:0 Ratio:nan

Number of Query: 0
Max Complexity of Formulae: 0
```

The first line "no counter example found" tells that Blade detects no error. The meaning of the output will be explained in Section [Output](#).

If otherwise your [Simplify](#) executable is located under the directory

```
C:\hai\simplify_theorem_prover\
```

Then you should probably use the following command, which will print out the same output.

```
> blade a.cfa C:\hai\simplify_theorem_prover\Simplify
```

Input

1. Control flow automata

The input file shall contain a control flow automata, which is essentially a control flow graph. Roughly speaking, each [edge](#) of the graph denotes a statement, also known as a transition. There are only two types of transitions, [assign](#) and [assume](#). Transition `assign` assigns a new value to a specific variable. The value could be a constant, another variable, or an [expression](#) on constants and variables. Transition `assume` comes together with a logic

formula. An assume edge will be taken only when the associated formula satisfies. A control flow automata has a unique **entry node**, from where the execution starts. It could also have multiple **error node**. An error occurs when the execution reaches an error node, which corresponds to the semantics of the `assert` statement. A program is considered to be safe if no error node is reachable. If any error node is found reachable, an execution trace will be printed out to indicate how the error node is reached. The trace is known as a **counterexample**.

Each line in the input file shall be either an edge, an error node, or an entry node. Nodes are identified by a integer. Edges are directional.

2. Expressions

An expression is a constant, a variable, or constants and variables connected by arithmetic operators. Currently Blade support four operators,

- "+" (plus),
- "-" (minus),
- "*" (multiply), and
- "/" (divide).

Constants are integers. Variables are represented by variable names, which are strings of letters and numbers with its initial character being a letter. Variable names are case sensitive. Blade does not support pointers yet. Hence

```
link_node->p_next
```

is interpreted as a strange variable name representing a single variable. Hence Blade does not know that the following two statements

```
cur_node = cur_node->p_next;
cur_node = cur_node->p_prev;
```

will move `cur_node` back. It will, however, interprets it as

```
var_1 = var_2;
var_1 = var_3;
```

Expressions shall be in pre-order, i.e., operator comes first, then the left operand, and the right operand. For example,

```
x + y
```

shall be written as

```
(+ x y)
```

Parentheses are of intensive use to separate parts in an expression. So

```
x + y + z + w
```

shall be written as

```
(+ x (+ y (+ z w)))
```

Or equivalently

```
(+ (+ (+ x y) z) w)
```

Spaces shall be used properly to separate different parts in the expressions.

3. Predicates

A predicate is two [expressions](#) connected by relation operators. Blade supports six operators,

- ">" (greater than),
- ">=" (greater than or equal to),
- "<" (less than),
- "<=" (less than or equal to),
- "EQ" (equal to), and
- "NEQ" (not equal to).

Similar to [expressions](#), predicates shall also be in pre-order. For example

```
x == y + z
```

shall be written as

```
(EQ x (+ y z))
```

Spaces shall be used properly to separate different parts in the predicates.

4. Formulae

A formula is one or two [predicate](#) connected by logic operators. Blade supports three operators,

- "AND" (and)
- "OR" (or), and
- "NOT" (not).

Similar to [expressions](#) and [predicates](#), formulae shall also be in pre-order. For example

```
x == y && y == z
```

shall be written as

```
(AND (EQ x y) (EQ y z))
```

Spaces shall be used properly to separate different parts in the formulae.

5. Assign and assume edges

An assign edge is in the format

```
ASSIGN (from_node, to_node) var_name := expression
```

where `ASSIGN` is the keyword, `from_node` and `to_node` are two integers indicating the entry and exit node of the edge, `var_name` is the name of the variable to which a value is assigned, and `expression` is the [expression](#) producing the value.

An `assume` edge is in the format

```
ASSUME (from_node, to_node) formula
```

where `ASSUME` is the keyword, `from_node` and `to_node` are two integers indicating the entry and exit node of the edge, `formula` is the [formula](#) indicating when to take the edge.

For example, the following two edge firstly assign `y` to `x`, then branch to node 3, if `x` is not equal to `y`.

```
ASSIGN (1,2) x := y
ASSUME (2,3) (NOT (EQ x y))
```

Spaces shall be used properly to separate different parts in the edges.

6. Error node and entry nodes

A line in the format

```
ERROR n
```

indicates that node `n` is an error node.

A line in the format

```
ENTRY n
```

indicates that node `n` is an entry node. A control flow automata shall have a unique entry node, and 0 or more error nodes.

Spaces shall be used properly to separate different parts in the nodes.

7. Examples

The following example first assign `y` to `x`, then branch to an error node, if `x` is not equal to `y`.

```
ASSIGN (0,1) x := y
ASSUME (1,2) (NOT (EQ x y))

ERROR 2

ENTRY 0
```

More examples can be found in the [Blade package](#), and the [experimental result page](#).

Output

1. Counterexamples

A counterexample is an execution trace ended at an [error node](#) indicating how the [error node](#) is reached. If Blade finds any counterexample, the first line of the output will be the counterexample in reverse order. Otherwise, the first line will be "no counter example found".

2. Statistics on runtime

Runtime is measured by the number of calls to [Simplify](#), because theorem-prover dominates the runtime of abstraction-based model checker, such as Blade and [BLAST](#). The complexity (the number of predicates in a formula) of the formulae in each call to the theorem-prover affects its runtime severely, because theorem proving subsumes the [SAT \(Satisfiability\) problem](#). Blade will print out the number of calls to the theorem-prover and the maximum complexity of the formulae as the last two lines of the output upon termination. For example,

```
Number of Query: 113
Max Complexity of Formulae: 56
```

3. Statistics on storage usage

Storage usage is measured by the number of [predicates](#) stored per node in the reachability tree. A reachability tree is generated during the model checking process, to record whether the statements are checked or not. Statements inside a loop or below a branch may occur multiple times in the reachability tree to account for their different status. Blade will print out the total number of predicates, the number of nodes of the reachability tree, and the number of predicates per node as the last sixth line of the output upon termination. Blade will also print out the reachability tree, also known as the slice tree. An example output is as follows.

```
# Predicates:56 # Nodes:142 Ratio:0.394366
```

Limitation

1. Front-end

We are short of a front-end to translate C code into [control flow automata](#). Other front-ends, such as for [OWL-S](#), are also desirable.

2. Pointers

Blade does not support pointers yet.

3. Recursive function call

Currently all function calls are in-lined to form a single [control flow automata](#). Hence it is impossible for Blade to handle recursive function call.

4. Non-termination

It could happen that Blade does not terminate when checking some programs. Blade is an abstraction-based model checker. The theoretical explanation is that the problem to determine a proper set of predicates that witness the abstraction is [undecidable](#).

Know problems

1. Blade runs much slower on Windows than on Linux

This is because the communication between Blade and [Simplify](#) is implemented as file exchange on Windows, and by pipe on Linux. Pipe is much faster than file exchange.

2. Blade may fail to recognize the input file

This happens if space is not properly used to separate the parts in [expressions](#), [predicates](#), [formulae](#), [edges](#), or [nodes](#).

Appendix

1. Hijack the communication between [BLAST](#) and [Simplify](#)

The [Blade package](#) includes a small program `hijacker.c` to hijack the communication from [BLAST](#) to [Simplify](#) and output to a log file. The log file is used to compare the performance of Blade and [BLAST](#) in the [experiments](#). Slight modification is necessary before one can use the hijacker. To get started with the hijacker,

- Copy the [Simplify](#) executable to a different name or location.
- Change the two global string variables accordingly to indicate the name or location of the real [Simplify](#) executable, and the name and location of the log file.
- Compile `hijacker.c`, rename the executable to be "Simplify".
- Copy the renamed hijacker executable to "\$BLASTHOME/simplify/bin/".

The hijacker always append to the log file if it exists. Manual removal of the log file is required if one does not want to keep old log files.

The hijacker only works on Linux box.

2. Count the maximum complexity of the formulae of BLAST

The [Blade package](#) include a binary executable "counter" to count the [maximum complexity of the formulae](#) in the log file generated by [hijacker](#). It takes one command line argument as the name of the log file, and prints out the maximum complexity. [BLAST](#) may push on to stack several predicates as premises before it incurs a call to the theorem-prover, the complexity of which is counted by adding them to that of forth coming formulae. An execution example is as follows.

```
> ./counter simplify.log  
Max Complexity: 10
```

The counter only works on Linux box.