BLAST

September 16, 2008

1 Introduction

Blast (**B**erkeley **L**azy **A**bstraction **S**oftware verification **T**ool) is an automatic verification tool for checking temporal safety properties of C programs. Given a C program and a temporal safety property, Blast either statically proves that the program satisfies the safety property, or provides an execution path that exhibits a violation of the property (or, since the problem is undecidable, does not terminate). Blast constructs, explores, and refines abstractions of the program state space based on lazy predicate abstraction and interpolation-based predicate discovery. The concepts of Blast are described in D. Beyer, T.A. Henzinger, Ranjit Jhala, and Rupak Majumdar, "The Software Model Checker Blast: Applications to Software Engineering", Int. Journal on Software Tools for Technology Transfer, 2007. Online: DOI 10.1007/s10009-007-0044-z.

Blast is relatively independent of the underlying machine and operating system. However we have only tested it on Intel Pentium processors under Linux and Microsft Windows with Cygwin. Other operating systems for the processors above have not been tested, but the code may work under other operating systems with little work.

Blast is free software, released under the Modified BSD license.

A PDF version of this document is also available.

2 Download and Installation

Blast can be downloaded from Blast's download web page.

2.1 Downloading Binaries

You can download executable versions of Blast for Linux and Windows with Cygwin. You should independently download and install the Simplify Theorem Prover (see item 5 below). Blast requires that the Simplify theorem prover is in the path.

2.2 Downloading the Source Distribution

You will need OCaml to build Blast. Blast has been tested on Linux and on Windows with Cygwin. If you want to use Blast on Windows then you must get a complete installation of `Cygwin` and the source-code OCaml distribution and compile it yourself using the Cygwin tools (as opposed to getting the Win32 native-code version of OCaml). If you have not done this before then take a look here.

1. Download the Blast source distribution.
2. Unzip and untar the source distribution. This will create a directory called `blast-2.4` whose structure is explained below.
   `tar xvfz blast-2.4.tar.gz`

3. Enter the `blast-2.4/blast` directory and run GNU Make to build the distribution.
   ```
   cd blast-2.4/blast
   make distclean
   make
   ```

4. You should now find the executables `pblast.opt` and `spec.opt` in the directory `bin`. These are symbolic links to files of the same name in the directory `psrc` and `spec`, respectively. The executable `pblast.opt` is the Blast model checker, the executable `spec.opt` is the specification instrumenter.

5. You should also download and install the Simplify Theorem Prover. This involves putting the executables `Simplify` (Linux) or `Simplify.exe` (Windows) in the `bin` directory. Additionally, Blast has interfaces to the Cvc Theorem Prover, should you wish to install and use this tool for theorem prover calls. Again, this involves putting the executable for Cvc in the `bin` directory. Note that in order for Blast to use Simplify or Cvc, the executable for Simplify and Cvc must be in your current path. It is a good idea to add the Blast `bin` directory to your path. Blast can also use any other SMT-LIB compatible theorem prover for its satisfiability queries (cf. release notes).

6. Blast also comes with an independent GUI. In order to install the GUI, you must download and install the LablGTK package in addition to Ocaml. After you have installed LablGTK, you can build the GUI by going to the `blast-2.4` directory and typing:
   ```
   make gui
   ```
   This will create the GUI executable `blastgui.opt` in the directory `bin`.

7. Blast (actually the GUI) requires the use of the environment variable `BLASTHOME`. Therefore you should set the environment variable `BLASTHOME` to point to the directory `blast-2.4/blast` where you have downloaded Blast.

8. Congratulations! You can now start using Blast.

3 A Tutorial Introduction to Blast

In this section we show how Blast can be used to check safety properties of C code. We shall discuss how to specify safety properties in the Blast specification language, and the various options that Blast takes to check C code against the specification. Blast can be run both from the command line and from a GUI.

3.1 Reachability Checking

The simplest way to write a program that Blast should check is to specify a *C label* at a program location that must not be reached. The model checker Blast checks whether such a label in the C source code is reachable. The basic command for running Blast is "`pblast.opt prog.c -main start -L errlabel`". This runs the model checker on the program `prog.c` and checks for reachability of the error label `errlabel`, starting from the initial location of function `start`. The run ends with either an error trace or with a message that the error was unreachable (or, since the reachability problem is undecidable, the program may not terminate). The defaults for `start` and `errlabel` are `main` and `ERROR`, respectively. Therefore, invoking Blast with `pblast.opt prog.c` checks whether the program label `ERROR` is reachable, when execution begins in the function `main`.

Consider the following C program:

```
int main() {
  int x,y;
```

```
  if (x > y) {
    x = x - y;
    if (x <= 0) {
      ERROR: goto ERROR;
    }
  }
}
```

Any other safety property can be transformed to reachability checking by program instrumentation. For example, more sophisticated specifications are instrumented into the code by preprocessing step such that failure of the specification amounts to reaching a particular error label.

This also means that you can add your own annotations directly to the code, and have a special error label to signify violation of the property you are checking.

3.2 Assertion Checking

The most convenient form of safety properties are *assertions* that specify invariants of the program. The programmer writes

```
assert (e)
```

in the code, where `e` is a C expression. If the expression `e` evaluates to 0 at run time, the program aborts. The intent is that the programmer has reasoned that `e` is an invariant of the program at the place the assert has been introduced. Consider for example the following piece of code.

```
#include <assert.h>

int foo(int x, int y) {
if (x > y) {
  x = x - y;
  L: assert(x > 0);
        }
}
```

By the constraints introduced by the checks, the programmer knows that at the label `L`, the value of `x` is greater than 0. Assertions are typically checked at run time during the testing phase of a program. However, using Blast, one can check assertions statically, during compile time. Moreover, there is no explicit test case to be written.

To run the above example through Blast, you must do the following. First, instead of the system header file `assert.h`, you must use the `assert.h` file provided with Blast (in the directory `test/headers` of the distribution. Then, you must produce a preprocessed file containing the source code. The above file is in `test/tutorial/tut1.c`. Create a preprocessed file by using the commands:

```
gcc -E -I $BLASTHOME/test/headers/ tut1.c > tut1.i
```

This creates a preprocessed file called `tut1.i`. Now we can run Blast on this file.

Running Blast **from the Command Line.** The basic command to run Blast is `pblast.opt filename -main mainfun` where `filename` is the file you are checking, `mainfunction` is the name of the starting function, and `ErrorLabel` is an error label. The default for mainfuntion is `main`, and ErrorLabel is `ERROR`, so e.g., if you are checking a program starting from `main`, and checking reachability of the label `ERROR`, you can just write `pblast.opt filename`. In our case, the filename is `tut1.i`, and the start function is `foo`; the assertion check automatically introduces an error label `ERROR`. So we invoke Blast with

```
pblast.opt tut1.i -main foo
```

(make sure Simplify is in your path). Blast comes back and reports that the system is safe, i.e., the assertion is not violated.

Running BLAST **from the GUI.** First, at the command prompt, type `blastgui.opt` after making sure that that file is in your path. First we must load the file `foo.i`: this is done by clicking on `File` and then selecting `Load Source` and then selecting the file `tut1.i`. Next, to run BLAST on this file, in the text pane labelled "Command" type `-main foo`, and then click the button labelled `Run`.

The text written into the command pane is the options that BLAST is run with `-main foo` tells BLAST to start the analysis from the function called `foo`. Quickly, the tool pops up a window that states that the system is safe, meaning the asserts never fail. In the pane labelled `Log`, BLAST prints out the predicates that it learns and uses to prove the property. When the model checking is finished, it comes with other statistics related to the analysis, the number of iterations, and the number of refinements required to prove the property.

Error Traces. Now consider an erroneous version of the same program – where the programmer swaps the variables `x` and `y` in the subtraction.

```
#include <assert.h>

int foo(int x, int y) {
        if (x > y) {
          x = y - x;
          assert(x > 0);
        }
}
```

Make the above change in the file `tut.c`, and then do
`gcc -E -I $BLASTHOME/test/headers/ tut1.c > tut1.i $.`

This time, BLAST comes back and says that an error is found, and moves to a pane where the error trace is shown in the middle.

```
Size:5

4 :: 4:          Pred(x@foo>y@foo) :: 5
5 :: 5:          Block(x@foo = y@foo  -  x@foo;) :: 6
6 :: 6:          Pred(Not (x@foo>0)) :: 6
6 :: 6:          FunctionCall(__assert("foo.c", 6, "x > 0")) :: -1
```

The size parameter gives the length of the error trace. The numbers on the left give the line numbers in the source code for the corresponding program actions. Notice that the statement `assert(x>0)` of the source code is expanded to the two actions

```
6 :: 6:          Pred(Not (x@foo>0)) :: 6
6 :: 6:          FunctionCall(__assert("foo.c", 6, "x > 0")) :: -1
```

that checks the asserted invariant, and calls `__assert` if the invariant fails.

Exercise 1 Assertions are also used to specify unreachable code, for example, at the end of an infinite server loop. Consider the simple example.

```
main () {
  int x,y;
  x = 0;
  y = 0;
  while (x==y) {
    x++;
    y++;
  }
  assert(0);
}
```

Check that BLAST finds that the assert is indeed unreachable (note that `assert(0)` always fails).

3.3 Temporal Safety Specifications

Assertions are a particularly simple and local way to specify program correctness. More generally, we are interested in temporal safety properties, where we wish to check that our program satisfies some finite state property. For example, we might wish to check that a program manipulating locks acquires and releases locks in strict alternation (that is, two calls to lock without a call to unlock in the middle is an error, and vice versa). More generally, we wish to check safety properties concerning the proper sequencing of program "events". Let us be concrete. Consider the following program that manipulates locks (in the file `tut2.c`).

```
int STATUS_SUCCESS = 0;
int STATUS_UNSUCCESSFUL = -1;
struct Irp {
  int Status;
  int Information;
};

struct Requests {
  int Status;
  struct Irp *irp;
  struct Requests *Next;
};

struct Device {
  struct Requests *WriteListHeadVa;
  int writeListLock;
};

void FSMInit() {
// code to initialize the global lock to unlocked state
}

void FSMLock() {
// code for acquiring the lock

}
void FSMUnLock() {
// code for releasing the lock
}

void SmartDevFreeBlock(struct Requests *r) {
// code omitted for simplicity
}

void IoCompleteRequest(struct Irp *irp, int status) {
// code omitted for simplicity
}

struct Device devE;
```

```
void main () {
 int IO_NO_INCREMENT = 3;
 int nPacketsOld, nPackets;
 struct Requests *request;
 struct Irp *irp;
 struct Device *devExt;

 FSMInit();
 devExt = &devE;


 /* driver code */
 do {
   FSMLock();
   nPacketsOld = nPackets;

   request = devExt->WriteListHeadVa;

   if(request!=0 && request->Status!=0){
     devExt->WriteListHeadVa = request->Next;

     FSMUnLock();
     irp = request->irp;

     if((*request).Status >0) {
       (*irp).Status = STATUS_SUCCESS;
       (*irp).Information = (*request).Status;
     } else {
       (*irp).Status = STATUS_UNSUCCESSFUL;
       (*irp).Information = (*request).Status;
     }
     SmartDevFreeBlock(request);
     IO_NO_INCREMENT = 3;
     IoCompleteRequest(irp, IO_NO_INCREMENT);
     nPackets = nPackets + 1;
   }
 } while (nPackets != nPacketsOld);
 FSMUnLock();
}
```

For simplicity, we assume that there is just one global lock that is acquired by a call to `FSMLock` and released by a call to `FSMUnLock`. We wish to check that the function `main` calls `FSMLock` and `FSMUnLock` in alternation: if there are two successive calls to `FSMLock` without a call to `FSMUnLock` in the middle, it is an error; similarly, if there are two successive calls to `FSMUnLock` without a call to `FSMLock` in the middle, it is an error.

One way to check this property is to instrument the program manually by adding some observer variables, and instrumenting the program to update the observer variables at each interesting program event. Then the check for the safety property can be reduced to checking an assertion on the observer variables. For example, in the above code, we can add the observer variable `int lockStatus` that tracks the current state of the global lock (locked or unlocked), and update `lockStatus` to 1 (for locked) whenever `FSMLock` is called, and update `lockStatus` to 0 (for unlocked) whenever `FSMUnLock` is called. We also initialize `lockStatus` to 0 in the call to `FSMInit`. Finally, we add the assertion `assert(lockStatus == 0);` in the beginning of `FSMLock` (to say that the lock is not held), and the assertion `assert(lockStatus == 1);` in the beginning of `FSMUnLock` (to say that the lock is held). Then the instrumented program satisfies our property iff the assertions are never violated.

The problem with this instrumentation is that it is specific for this code, and tedious to do manually. Therefore BLAST comes with a specification language where safety properties can be specified at a higher level. The specification is automatically compiled into an instrumented program with assertions as above.

For our safe locking property, we write the property as follows (in a file called `lock.spc`).

```
global int lockStatus = 0;

event {
  pattern { FSMInit(); }
  action { lockStatus = 0; }
}

event {
  pattern { FSMLock(); }
  guard { lockStatus == 0 }
  action { lockStatus = 1; }
}

event {
  pattern { FSMUnLock(); }
  guard { lockStatus == 1 }
  action { lockStatus = 0; }
}
```

We give an intuitive description of the specification. The next section gives a detailed account of the specification language. The declaration `global int lockStatus` defines an observer variable `lockStatus` and initializes it to 0. The specification is given in terms of program events. There are three interesting events in this specification: calls to the functions `FSMInit`, `FSMLock`, and `FSMUnLock`. Each event is associated with a syntactic pattern in the code. For example, the pattern `FSMLock();` is matched every time there is a call to `FSMLock` in the code. Each pattern has a guard that must be met when that pattern matches in the code: for example, when the pattern `FSMLock` is matched, the variable `lockStatus` must be 0. If not, there is a violation of the property. If the guard is true, it can be omitted. Moreover, there is an action associated with the event that specifies how the observer variables must be updated.

BLAST comes with an instrumentation tool that takes a specification and a program and builds an instrumented program from them. To use this, we say

```
spec.opt lock.spc tut2.c
```

This builds an instrumented program called `instrumented.c` as well as a predicate file `instrumented.pred`.

We can now run BLAST on this file: first, clear the contents of the "Command" pane, then in the `File` menu load `instrumented.c` as the source file and `instrumented.pred` as the predicate file. Notice that in the Source pane, the source of this program is given and in the `Predicates` pane are listed the predicates created from the specification, namely a predicate tracking the values of `lockstatus`, `lockstatus == 0, lockstatus == 1` . Now press the `Run` button to run BLAST, which reports that the specification is indeed met by the program.

Exercise 2

1. What predicates are used by BLAST to prove this property?

Let us now repeat the above with a buggy version of the `tut2.c`. Comment out the line `nPackets = nPackets` 1; + and repeat the above – *i.e.* , run `spec.opt` and then run BLAST on `instrumented.c`. This time the tool reports an error trace in the counterexample trace pane.

3.3.1 The Counterexample Trace Pane

The counterexample trace pane is broken into 3 subpanes – the leftmost is the program source, the middle pane is the sequence of operations that is the counterexample and the rightmost pane contains the state of the system given by values for the various predicates in the top half and the function call stack in the lower pane at the corresponding points in the counterexample. One can see the state of the system at different points of the trace by clicking on the corresponding operation in the middle pane. When one chooses an operation in the middle pane, the corresponding program text is highlighted in the left pane and the predicate values and control stack are shown in the right pane. Alternatively, one can go back and forth along the trace using the arrows along the bottom.

Adjacent The first pair move to the appropriate adjacent operation,

Undo/Redo The second pair are like web-browser buttons in that they move to the previous or next operation among the various operations that have been hitherto selected,

FF/REW The third pair move to the next or previous point in the trace where the values of the predicates are different from the present values – thus they skip over irrelevant operations and proceed directly to the nearest operation that changes the values of the predicates.

Pred Change The fourth pair are used after first selecting some predicate in the rightmost pane – after selecting the predicate if one clicks the right (left) button one is taken to the first operation in the future (past) where the value of that predicate is about to (has just) change(d). This pair is used to see which operations cause a predicate to get its value at some point in the trace.

In the given counterexample, we see that the first operation is a call to `__initialize__` which is a spec function that sets the initial value of the spec variable `lockStatus`. The very next statement sets `lockStatus` to `0`, and on selecting that operation, notice that in the rightmost pane, the state is such that `lockStatus == 0` and `lockStatus == 1` is false (*i.e.* its negation is true). In this pane, select the predicate `lockStatus == 0` and then click the Right Pred. Change button (the right arrow with the orange "sun" behind it). The gui leaps forward to the operation where `lockStatus` is going to be set to `1`. On pressing the Undo button, the gui jumps back to the previous operation that was being viewed. You can play around with the trace and view the various operations and convince yourself that this is indeed a violation of the specification.

3.4 Additional Predicates

Now consider the following example.

```
void main () {
  int x, y;
  int __BLAST_NONDET;
  x = 0;
  y = 0;
  while (__BLAST_NONDET) {
    x ++ ;
    y ++ ;
  }
  while (x > 0) {
    x -- ;
    y -- ;
  }
  assert (y == 0);
}
```

The variable `__BLAST_NONDET` has special internal meaning: it is treated as a nondeterministic choice. This is often useful to model nondeterministic behavior for stub functions. If you run BLAST on this example (after processing with `gcc` and `mfilter`) , it fails with the error message: `No new predicates found!`

This is because the predicate discovery engine of BLAST is not powerful to infer the predicates `x==y, x >= 0` from this example. In such a case, the user can give additional useful predicates to BLAST. Additional predicates can be given using a *predicate file*. A predicate file contains a list of useful predicates separated by semicolons that BLAST reads in at the start of the model checking. Each predicate is an atomic predicate written in C syntax, but where the local variable `x` in function `f` is written as `x@f` (global variables `y` are still written `y`). In this example, we can create a file `tut3.pred` with the single entry `x@main == y@main; x@main >= 0;` (the semicolons are necessary). We then run BLAST after selecting the file `tut3.pred` as a predicate file. and this time it reports that the program is safe. Methods to automatically infer suitable predicates is still an open area of research – the problem is of course undecidable in general.

Of course, additional predicates can be given for any program, not necessarily those on which BLAST fails. Our experience is that it is often useful to start with the predicates generated from the guards of the specification. Therefore, the instrumentation of the specification also generates a predicate file called `instrumented.pred`, which we used earlier for the previous example.

3.5 Smarter Predicate Discovery

The default predicate discovery engine in BLAST may not always succeed in finding "good" predicates. Therefore, BLAST implements some additional analysis for finding suitable predicates. These can be accessed with the command line options -craig 1, -craig 1 -scope, and -craig 2. These implement predicate discovery based on interpolants. The option -craig 1 -scope is an efficiency heuristic on top of -craig 1, it removes predicates not in scope. The option -craig 2 does a precise analysis. On the first run, -craig 2 often takes more time than -craig 1, but it produces a much finer abstraction. Even when the default predicate discovery succeeds, the -craig options often succeed with fewer predicates. Additional heuristics are implemented, and can be accessed with the -predH flag. We recommend running BLAST with -predH 7 (level 7 is the highest level).

Consider for example the program `foo.c`:

```
#include <assert.h>

int main() {
  int i, x, y, ctr;

  x = ctr;
  y = x;
  y = x + 1;
  if (ctr == i) {
    assert (y == i + 1);
  }
}
```

Let `foo.i` be the preprocessed program as above. Running `pblast.opt foo.i` does not succeed, and says "No new predicates found." However, running

```
pblast.opt foo.i -craig 1 -predH 7
```

verifies that the program is safe.

3.6 Saved Abstractions

When BLAST finishes running on the program filename.c, it creates a file called filename.abs containing the abstraction that was used. This abstraction can be used in subsequent runs to save time in predicate discovery. Subsequent runs may be required in a regression testing scenario where some parts of the program has changed, and BLAST is run to verify the property on this new program. The old abstraction file can be read in with the -loadabs flag. For example, after we have run

```
pblast.opt foo.i -craig 1 -predH 7
```

a file called `foo.abs` is created. Subsequently, running

```
pblast.opt foo.i -craig 1 -predH 7 -loadabs
```

will reuse the previous abstraction and do the verification much faster.

4 The Specification Language

4.1 Tool usage

The specification language is processed by a command-line tool that takes as input a specification and a list of C source files. A single instrumented C source file is created that combines the input sources and ensures that they satisfy the properties described in the specification. An `instrumented.pred` file containing hint predicates for BLAST is also generated. For example, running

```
spec.opt myspec.spc myfile.c
```

will produce `instrumented.c` and `instrumented.pred` in the current directory. You can then feed this file into BLAST for verification. There is no need to tell BLAST the error label, since the generated file uses the default label. For example, you could check the output by running

```
pblast.opt -bddcov -nofp -predH 6 -block -pred instrumented.pred instrumented.c
```

There are other ways to invoke `spec.opt`. Running

```
spec.opt myspec.spc myfile1.c myfile2.c myfile3.c
```

merges all of the specified C sources into a checkable `instrumented.c`.

```
spec.opt myspec.spc
```

merges all C sources in the current directory (except `instrumented.c`, if it exists) into a checkable `instrumented.c`.

4.2 Example specification files

4.2.1 A global lock

```
#include <locking_functions.h>

global int locked = 0;

event {
  pattern { $? = init(); }
  action { locked = 0; }
}

event {
  pattern { $? = lock(); }
  guard { locked == 0 }
  action { locked = 1; }
}

event {
  pattern { $? = unlock(); }
  guard { locked == 1 }
  action { locked = 0; }
}
```

This specification models correct usage of abstract global locking functions. A global variable is created to track the status of the lock. Simple events match calls to the relevant functions. The event for `init` initializes the global variable. The other two events ensure that the lock is in the right state before making a function call. When these checks succeed, the global variable is updated and execution proceeds. When they fail, an error is signalled.

Pattern matching is performed in an intermediate language where code is broken down into sequences of function calls and assignments. The `$?`'s above match either a variable to which the result of a function call is assigned or the absence of such an assignment, thus making the patterns cover all possible calls to the functions.

4.2.2 Simplified `seteuid` and `system`

```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <stdlib.h>

global int __E__ = 0;

event {
  pattern { $? = seteuid($1); }
  action { __E__ = $1; }
}

event {
  pattern { $? = system($?); }
  guard { __E__ != 0 }
}
```

This specification models the requirement that a setuid program should not call the `system` function until it has changed the effective uid to a nonzero value. The `$1` in the `seteuid` patterns will match any parameter, including the result of a complicated series of function calls. Here `$?` is used as a function parameter to match all remaining actual parameters.

4.2.3 X11 parameter consistency checking

For the sake of this example, we consider types and functions similar to those found in an X11 windowing system API:

```
typedef struct context *Context;
typedef struct image *Image;
typedef struct display *Display;

Display newDisplay(void);
Context genContext(Display);
```

```
Image genImage(Display, int);
void putText(Display, Context, Image);
```

We now define a specification file to verify the property that the `Context` and `Image` passed to `putText` both belong to the `Display` that is passed.

```
#include "x11.h"

shadow Image {
Display display = 0;
}

shadow Context {
Display display = 0;
}

event {
after
pattern { $1 = genContext($2); }
action { $1->display = $2; }
}

event {
after
pattern { $1 = genImage($2, $3); }
action { $1->display = $2; }
}

event {
pattern { $? = putText($1, $2, $3); }
guard { $2->display == $3->display && $2->display == $1 }
}
```

4.3 Informal description of syntax

A specification (`.spc`) file consists of a sequence of the following kinds of directives.

4.3.1 Includes

These are verbatim C-style `#include` directives. You should include the necessary header files to support all of the code contained in the specification. For example, functions used should be prototyped in some header file that is included.

4.3.2 Global variables

These are C-style definitions of single variables with initializers, prefaced by the keyword `global`. For example, `global int flag = 10;`. Each directive creates a global variable to which the other parts of the specification may refer.

4.3.3 Shadowed types

It is possible to replace "abstract types" with structures storing information pertinent to properties to be checked. Here an abstract type is a type used in the code to be checked in such a way that it could be replaced by any other type without creating type errors. For example, a type that has values used as parameters to arithmetic operators or that have struct members projected from them is not abstract. Abstract types will generally arise when dealing with libraries whose source is not available or that you choose to treat as "black boxes."

A type is shadowed by a directive consisting of the keyword `shadow` followed by the name of the type to be shadowed and then a C-style struct definition consisting of a set of field definitions inside braces. The difference from C field definitions is that each field must have a starting value defined in the same manner in which you would define an initial value for a global variable. *Note: The initializers are not used in the current implementation.*

4.3.4 Events

Events are used to change global state and verify properties based on the execution of a C program. An event directive consists of the keyword `event` followed by a sequence of sub-directives within braces.

`pattern` Patterns specify which possible program statements activate an event. Following the `pattern` keyword is a sequence of C statements enclosed in braces. These statements may have *pattern variables* in some positions where expressions belong. A pattern variable is the `$` character followed by a positive integer. An event will be activated for any sequence of statements that matches the pattern sequence for that event, with pattern variables matching any expressions in the actual code. Currently, the same pattern variable may only appear multiple times in a single pattern to match the same C variable used in multiple places.

Patterns may also contain an additional special sequence, `$?`. In most positions, this sequence acts just like a pattern variable, except that matching expressions are not bound in guards, actions, or repairs. It has two additional special functions: A pattern like `$? = function_call(some, args);` matches a function call matching the given function call pattern, regardless of whether or not the result is saved in a variable, discarding the destination variable if it is present. `$?` may be given as the last actual parameter in a function call to match all remaining parameters, zero or more.

Patterns are only matched against straight-line code within basic blocks. Both patterns and C source files are compiled to the Cil intermediate language before matching. In this form, the only valid statements are (1) assignments of side effect-free expressions to variables and (2) function calls, optionally saving the return value to variable.

`guard`, `action`, and `repair` The `guard` directive is followed by a C expression (possibly with pattern variables) inside braces. `action` and `repair` are followed by sequences of C statements (possibly with pattern variables) inside braces.

These directives specify the checks to be made and actions to be taken at certain points during execution, relative to a match of a given pattern. If the guard expression is true with the matching expressions substituted for corresponding pattern variables, then the specified `action` code is run with the same pattern variable substitutions. If the guard expression is false and a `repair` has been specified, then those instructions are run with substitutions. If the guard is false and no repair is specified, then an error is signalled by calling the `__error__` function. Actions and repairs may also call the `__error__` function manually.

These directives are all optional. The default guard is an always-true expression. The default action is empty, and omitting repair causes an error to be signalled when the guard is false. When an event is meant to update global state without verifying a program invariant, it is helpful to specify an empty repair to avoid signalling an error based on conditions used to determine how to change the state.

`before` and `after` These directives take no additional parameters and specify whether to check the guard and perform the appropriate action, repair, or `__error__` call before or after the execution of a matching sequence of statements, respectively. If neither directive is given, then `before` is taken to be present implicitly.

5 Using Blast: User Options

The following command line options are useful for running Blast (see `pblast.opt -help` for a complete list).

Model Checking Options. The following options are available to customize the model checking run.

- -main xxx. Specify the entry point of the program. The default is `main`.
- -L xxx. Specify an error label. The default is `ERROR`. Note that if there are several labels in the program with the same name, the effect of Blast is nondeterministic.
- -msvc. Parse file in MSVC mode. This is required by the CIL front end. The default mode is GNU CC. Use this to read and analyze programs that use Microsoft Visual C features.
- -bfs and -dfs. Specify the search strategy, breadth first or depth first. The default is -bfs.
- -pred xxx. Specify a set of seed predicates as the initial abstraction. When this is not specified, Blast starts running with the most abstract program: where all the data has been abstracted. The seed predicate file `xxx` contains a list of seed predicates, each predicate is an atomic predicate using the C expression syntax. See Section 7.3 for the syntax of predicates.
- -cf. Use context free reachability. This feature has not been tested for this release.
- -init xxx. Specify the initial state in the file xxx. The initial state is a (side effect free) C boolean expression in the program variables in scope in the start function.
- -inv xxx. Specify additional invariants in a file. These are conjoined with the set of reachable states. The invariant file contains a C boolean expression.
- -s xxx. Specify the satisfiability solver to be used by the decision procedures in Blast. The current options are Vampyre, Simplify, and Cvc, the default being Simplify. Note however that the option -craig uses Vampyre internally. See the programmer's documentation to see how to add your own decision procedure. We recommend the use of Simplify as default as it is considerably faster than Vampyre.
- -nocache. Do not cache theorem prover calls. This makes the run require less memory.
- -cov. With the option -cov, check for coverability is done only for control flow nodes that have back pointers.
- -reclaim. A space saving heuristic: does not keep whole tree around during the model checking, reclaims tree nodes on backtrack.
- -ax xxx. Specify a set of additional axioms for Simplify. The axioms file is passed to Simplify. Simplify requires that the file ends in ".ax". So, to pass the axiom file file.ax, say -ax file. It is assumed that the axiom file is in the same directory from which Blast is invoked.
- -alias xxx. Alias analysis options. If the option xxx is bdd, invokes a bdd based alias analysis, otherwise reads the alias relationships from the file xxx. Additionally, the option -pta [flow—noflow] specifies a mode for points to analysis: noflow runs a flow insensitive version, flow a flow sensitive version. This version of Blast only supports noflow. If this option is omitted, then no alias analysis is performed, and Blast makes the assumption that lvalues are not aliased. The option -scheck then performs some checks to ensure that this assumption is met. However, it can happen that the analysis is unsound, even though -scheck does not fail.

- -incref (default). Incomplete counterexample analysis. Blast with full alias analysis is expensive, especially when the alias analysis is imprecise. Often aliasing is not important, and this option gets a middle ground: the counterexample analysis is done without the aliasing information, but the model checking uses the alias information. The analysis is sound, but Blast can fail if the analysis requires aliasing relationships to be tracked.
- -cref. Complete counterexample analysis. Blast considers the alias-analysis information not only for the model checking, but also for the counterexample analysis. This analysis is more expensive, but helps extracting predicates that track aliasing relationships.
- -nofp. Ignore function pointer calls. This is put in as a convenience: the programmer can ignore function pointer calls in the analysis. Use with caution: the results of Blast are meaningful only under the assumption that the function pointer calls did not change the predicate state of the program.
- -bddcov and -nobddcov. The option -bddcov uses only bdds in cover check, -nobddcov does a full cover check. The default is -bddcov. It is unlikely you will need to change the default.
- -wpsat. Keep only satisfiable disjuncts in the weakest precondition.
- -restart. Restart model checking after every counterexample analysis (a.k.a. SLAM mode).
- -block. Analyze trace as blocks. This is the default, overriding the earlier counterexample analysis. You can still use the old counterexample analysis by running -noblock. With the old analysis, you can also specify the direction of the analysis with the flag -refine [fwd—bwd] that runs the analysis forwards or backwards, according to the option provided.
- -predH k. Perform predicate discovery heuristics. There are several levels, we recommend using level $k = 7$. The heuristics include checking all causes of unsatisfiability, and adding additional predicates based on the syntax.
- -tsd [zig]. Set trace search direction. [zig] goes zigzagging [] restarts from the end.
- -craig [1—2]. Use craig interpolants (FOCI) to get predicates ([1] use with scope, [2] maintains local tables of predicates). See the paper "Abstractions from Proofs" (by T.A. Henzinger, R. Jhala, R. Majumdar, and K.L. McMillan). -craig internally uses the block based counterexample analysis (i.e., automatically sets -block). See also -scope.
- -scope. Remove predicates not in scope. This should be run with -craig 1, otherwise this fails.

Program Optimization Options. Blast implements a set of program analysis routines that can make the analysis run significantly faster. These can be turned on or off with the following options.

- -pe. Implements an aggressive interprocedural constant propagation algorithm.
- -O x. Turn program optimizations on or off. The levels are 0-1. The default is 0 (off). In level 1, a cone of influence optimization is implemented. It assumes all variables occurring in conditionals are important, and propagates this to find all useful assignments. Further, the constant propagation algorithm is implemented (see also -pe).
- -depth k. Unroll CFA to depth k. This is experimental and not included in the release.

Parallel Model Checking and Races. Blast implements a *Thread modular* algorithm for checking races in multithreaded C programs. These options relate to the algorithm for checking races.

- -checkRace. Invoke the TAR algorithm to check for races on shared variable accesses.
- -par. Use data structures for parallel Blast. Not supported.

Saved Abstractions and Summarization. These options are used to save and load abstractions from a BLAST run.

- -loadabs. When BLAST is run with the file `fname.c`, it continuously outputs the abstraction used in the model checking into the file `fname.abs`. The -loadabs option can be used to read back the abstraction file created in a previous run. This makes subsequent runs faster and is useful for regression testing. This option also allows us to run BLAST with the abstraction generated from an interrupted run.
- -interface f. Specify name of a function to take as describing a component interface. Not supported in this release.
- -component f. Specify name of a function to check for satisfying the interface. Not supported in this release.

Proof generation options. BLAST implements a set of options to generate PCC style proofs. The proofs are output in textual form in LF syntax. These can be read and encoded by a standard PCC proof encoder.

- -pf. Generate proof tree in the file `foo.tree`.
- -pfgen. Spit out vampyre proofs. Always use this: the other option is buggy!
- -pffile xxx. File to write vampyre proofs. If no file is specified, proofs are written to `/tmp/lf.pfs`.

Old Heuristics that are no longer used/supported. You can omit reading about the options in this section. These pertain to several heuristics in the older version. The default is set to the heuristic that we found to work best. Many of the following heuristics are no longer supported.

- -comp xxx. This implements the `keep_subtree` heuristic from the Lazy Abstraction paper. The options are cut to remove the subtree, and path to keep the subtree. We found that path was nominally faster, but went into loops very often. So the default is set to cut.
- -post xxx. The algorithms to compute post. The options are slam for an approximate *Cartesian post* as implemented in SLAM. and H for the most precise predicate abstraction. We saw that slam post is vastly more efficient, yet is precise enough to prove properties of interest. The default is slam.
- -forget. This option naively forgets predicates found when it backtracks out of a subtree. This repeats work: the next time the same part of the program is visited, the same predicates are found again. The default is not to forget.
- -dc. Some don't care heuristic for predicates not in scope. Deprecated. Do not use, see -craig instead.

General Options. The following options let the user select different configurations, mostly for debugging.

- -debug. Prints out copious debugging information.
- -cfa-dot xxx. Output the CFA of every function in ATT dot format in the file xxx.
- -stop. Stop when the model checker hits the first (possibly invalid) counterexample. Useful for debugging.
- -traces. Every time a false counterexample is encountered, the trace itself is dumped. Used for diagnostic purposes. With the option -tracefile xxx, you can additionally specify the name of the file containing trace information. This is used by the trace viewer.

- -demo. Run in demo mode for the GUI.
- -xml. Generate error traces as a bunch of xml files that can be read in and displayed by SLAM's GUI.
- -help or –help. Display the list of options.

6 Graphical User Interface

BLASTcomes with a rudimentary whose chief purpose is to make it easier to view counterexample traces. In this section we discuss the GUI.

The GUI is started by the command `blastgui.opt`.

Source and predicate files are loaded in using `File` in the main toolbar, or by entering the filenames in the appropriate text boxes and clicking the load button. There are four sub-panes showing respectively a log of events, the source file, the predicate file and counterexample traces.

To run BLAST, the user must first select the source file and then optionally a predicate file and then type in the options in the text pane labelled options, and click the `Run` button. If the system is free of errors, BLASTwill (hopefully) pop up a window saying so, if not, it will (hopefully) switch to the counterexample trace pane showing a counterexample that violates the specification. We say hopefully as it is possible as we saw before that BLASTwill be be stuck at some point unable to find the right predicates to continue. In this case also, the GUI moves to the counterexample trace pane which now shows a trace on which BLASTis stuck – the user can then stare at the trace and guess some predicates which can then be fed to BLAST.

The Counterexample Trace Pane

The counterexample trace pane is broken into 3 subpanes – the leftmost is the program source, the middle pane is the sequence of operations that is the counterexample and the rightmost pane contains the state of the system given by values for the various predicates in the top half and the function call stack in the lower pane at the corresponding points in the counterexample. One can see the state of the system at different points of the trace by clicking on the corresponding operation in the middle pane. When one chooses an operation in the middle pane, the corresponding program text is highlighted in the left pane and the predicate values and control stack are shown in the right pane. Alternatively, one can go back and forth along the trace using the arrows along the bottom.

FW/BK The first pair move to the next and previous operation,

Undo/Redo The second pair are like web-browser buttons in that they move to the previous or next operation among the various operations that have been hitherto selected,

FF/REW The third pair move to the next or previous point in the trace where the values of the predicates are different from the present values – thus they skip over irrelevant operations and proceed directly to the nearest operation that changes the values of the predicates.

Pred Change The fourth pair are used after first selecting some predicate in the rightmost pane – after selecting the predicate if one clicks the right (left) button one is taken to the first operation in the future (past) where the value of that predicate is about to (has just) change(d). This pair is used to see which operations cause a predicate to get its value at some point in the trace.

7 Modeling Heuristics

7.1 Nondeterministic Choice

BLAST uses the special variable `__BLAST_NONDET` to implement nondeterministic choice. Thus,

```
if (__BLAST_NONDET) {
// then branch
} else {
// else branch
}
```

is treated as a nondeterministic `if` statement whose either branch may be taken. This is sometimes useful in modeling nondeterministic choice in specification functions or in models of library functions.

7.2 Stubs and Drivers

Blast is essentially a whole program analysis. If there are calls in your code to library functions, it expects to see the body of the function. If the body of a function is not present, Blast optimistically assumes that the function has no effect on the variables of the program other than the one in which the return value is copied.

Sometimes we are interested in the effect of library functions, but not in their detailed implementation. For example, we may be interested in knowing that `malloc` returns either a null pointer or a non-null pointer, without knowing exactly how memory allocation works. This is useful for scalability: we are abstracting unnecessary details of the library. Sometimes this is necessary as well: certain system services are written in assembly and not amenable to our analysis.

Blast expects in these cases that the user provides stubs for useful library functions. Each stub function is basically a piece of C code, possibly with the use of `__BLAST_NONDET` to allow nondeterministic choice.

7.3 Syntax of Seed Predicates

You can input initial predicates on the command line using the option -pred. This section gives the syntax for input predicates. The format of the predicate file is a list of predicates, separated by semicolons. Each predicate is a valid boolean expression in C syntax. However, we change variable names to also reflect the scope of the variable. So the variable `x` in function `foo` is written `x@foo`. The detailed syntax can be seen in the file `inputparse.mly` in the directory `psrc`.

Notice that if the same syntactic name is used for multiple variables in different scopes then Cil renames the local variables. In this case, one has to look at the names produced by Cil to use the appropriate variable in the predicates.

8 Aliasing

Pointer aliasing is a major source of complexity in the implementation of Blast. Blast comes with a flow insensitive and field insensitive Andersen's analysis for answering pointer aliasing questions internally. The implementation of the pointer analysis uses BDDs. Additionally, Blast allows the user to input alias information (generated from some other alias analysis) from a file. The syntax of an alias file is a list of C equalities between C memory expressions (variables, dereferences, field accesses) separated by commas.

Considering possibly aliased lvalues is essential for soundness of the analysis. Consider for example the following code:

```
int main() {
  int *a, *b;
  int i;
  i = 0;
  a = &i;
  b = &i;
  *b = 1 ;
  assert(*a == 1);
}
```

If the analysis proceeds without considering the alias relationship between `*a` and `*b`, the assertion passes. However, updating `*b` also updates `*a`. The analysis is expensive if the alias information is not precise, since all (exponentially many) alias scenarios between the variables must be considered. In order to improve precision, Blast makes the (possibly unsound) assumption that the program is type-safe, so that only variables of the same type may be aliased. Moreover, the current implementation does not handle function pointers. Code with function pointer calls therefore cause Blast to fail with an exception, unless the flag -nofp is used, in which case all function pointer calls are ignored.

The option -alias is used to provide aliasing information to Blast. The option takes a string argument. If the argument is bdd then the BDD based Andersen's analysis is run. If it is some other string, then Blast assumes that the string indicates a filename where aliasing relationships are given. If the alias option is omitted, Blast makes the unsound assumption that there are no aliases in the program.

Exercise 3 Consider the program

```
#include <assert.h>

int __BLAST_NONDET;


void swap1(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}

void* malloc(int k);

void main () {

  int *i, *j;

  int v1, v2;

  i = malloc(4);
  j = malloc(4);

  *i = v1;
  *j = v2;

  swap1 (i, j);
  swap1 (i, j);
  assert (  *i == v1 &&   *j == v2 );
}
```

1. Run Blast with

   ```
   pblast.opt foo.i -craig 1 -predH 7
   ```

   There is an error trace because Blast does not consider the aliasing among the variables. Now run Blastwith

```
pblast.opt -alias bdd -cref foo.i -craig 1 -predH 7
```

   BLAST says that the system is safe.

2. Now comment out the second call to `swap1` in `main`. Check that BLAST produces an error trace.

3. Now add a second swap routine

   ```
   void swap2(int *a, int *b) {
     *a = *a + *b;
     *b = *a - *b;
     *a = *a - *b;
   }
   ```

   Replace one of the calls to `swap1` with `swap2`. Verify that BLAST still proves the program correct.

4. Consider the following variant of `main`.

   ```
   void main () {

     int *i, *j;

     int v1, v2;

     i = malloc(4);
     j = malloc(4);

     *i = v1;

     swap1 (i, i);
     assert (  *i == v1 );
   }
   ```

   Does the assertion hold? What happens if you replace `swap1` with `swap2`? Run BLAST and verify in each case.

9 Programmer's Manual

9.1 Architecture of BLAST

BLAST uses the CIL infrastructure as the front end to read in C programs. The programs are internally represented as *control-flow automata* (implemented in module CFA). Sets of states are represented by the Region data structure. The Region module represents sets of states as boolean formulas over a set of base predicates and allows boolean operations on regions, and checks for emptiness and inclusion. The Abstraction functor takes the Region module and the CFA module, providing in addition (concrete and abstract) *pre* and *post* operations, and methods to analyze counterexamples. Using the Abstraction module, the LazyAbstraction functor implements the model checking algorithm at a high level of abstraction.

BLAST uses the Simplify Theorem Prover and the Vampyre Proof-Generating Theorem Prover as underlying decision procedures. Boolean formula manipulations are done using the Colorado University Decision Diagram package.

9.2 API Documentation

The architecture of Blast is described in the file `src/blastArch.ml`. We also have an online documentation extracted from the code. We index below the main types that are used to represent C programs in CIL:

- An index of all types
- An index of all values

10 Known Limitations

1. The current release does not support function pointers. With the flag -nofp set, you can disregard all function pointer calls. The correctness of the analysis is then modulo the assumption that function pointer calls are irrelevant to the property being checked.
2. **Recursive functions.** Currently Blast inlines function calls. This means that it loops on recursive calls. This is a big limitation on the files that can be analyzed. The option -cf implements context-free reachability. However it has not been tested.
3. There are several bugs in the options -craig [1—2]. If -craig should fail, we suggest you run Blast without this option and check.

11 Authors

Blast was developed by Dirk Beyer, Adam Chlipala, Tom Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre, with contributions from (among others) Yinghua Li, Ken McMillan, Shaz Qadeer, and Westley Weimer.

12 Troubleshooting

1. Blast fails with

   ```
   Failure(‘‘Simplify raised exception End_of_file’’)
   ```

   Is Simplify in your path?
2. Blast fails with

   ```
   Failure(‘‘convertExp: This expression Eq should not be handled here’’)
   ```

   Blast does not like expressions like

   ```
   return (x==y);
   ```

   Change this to

   ```
   if (x==y) return 1; else return 0;
   ```

   Similarly, change

   ```
   a = (x==y) ;
   ```

   to

   ```
   if (x==y) a = 1; else a = 0;
   ```

   and similarly for the other relational operators <=, >=, >, <, ! =.

Don’t see your problem? Send mail to blast@ic.eecs.berkeley.edu.

13 Bug reports

We are certain that there are still bugs in Blast. If you find one please send email to Blast at blast@ic.eecs.berkeley.edu or to Rupak Majumdar or Ranjit Jhala.

14 Changes