CSC2108 - Project Report

Lazy Abstraction on Software Model Checking

Wai Sum Mong

Abstract

This paper is a survey of the BLAST, which is a software model checker for C programs developed at Berkeley. Based on the popular abstract-check-refine paradigm, the concept of lazy abstraction is introduced in the BLAST project. Lazy abstraction is a new idea for the optimization of the abstract-check-refine loop. We present the lazy abstraction concept and the implementation framework of BLAST. In the end, an example of using the BLAST is also given.

1 Introduction

In recent decades, the growing importance of software and the increasing complexity of system design are driving the technologies of software verification. Traditionally, the language syntax and the type system convention are statically checked by the compilers while the program behavior can only be checked at runtime by using assertions and perform testing. Many efforts have recently contributed to enable statically checking the program behavior against the given properties. For example, the MOPS [1] targets to finding security bugs, and the SLAM project [2] aims at checking whether a program obeys the "API usage rules". Despite its difficulty, statically reasoning the semantics of a program does reduce the overhead incurred during checking the behavior at runtime. In this paper, we survey the *BLAST (Berkeley Lazy Abstraction Software Verification Tool)* [3], which is a new effort in this domain.

Developed at Berkeley, the BLAST is a software model checker for C programs. The BLAST shares the same goal of the SLAM project – checking whether a program obeys the given "API usage rules", and both of them are constructed based on the popular *abstract-check-refine* paradigm. However, the BLAST applies the concept of *lazy abstraction* to optimize the abstract-check-refine loop. We will focus on the discussion of the lazy abstraction idea and briefly describe the corresponding implementation in BLAST. Also, we have tried using the recent release of BLAST (version 1.0) on Windows under Cygwin. It is found that the current release of BLAST is still not sophisticated enough to work for arbitrary verification purpose. In the end, we will share our experience of using BLAST.

The remainder of this paper is organized as follows. Section 2 states the problem addressed by the BLAST. Section 3 presents the lazy abstraction methodology. Section 4 is the description of the implementation framework of BLAST. We discuss our experience of using the current release of BLAST (version 1.0) in Section 5, which is followed by the conclusion in Section 6.

2 Problem Statement

We first describe the problem addressed by BLAST in Section 2.1. The abstract-check-refine approach, based on which is the BLAST constructed, is then discussed in Section 2.2.

2.1 The Software Verification Problem of BLAST

The BLAST is to check the safety properties of C programs. This is done by reasoning whether the line(s) labeled by `ERROR` is reachable. Ideally, if the checking algorithm terminates, the checker either tells the user the program is safe (the `ERROR` label is not reachable) or gives the user a feasible execution path to the `ERROR` label (a counterexample).

Figure 1 gives a simple example. Given the program in Figure 1(a), the model checker will determine that the program is *safe*. However, the program 1(b) is *unsafe* because the `ERROR` label can be reached. These answers can be deduced through static analysis despite the values of x and y.

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

(a) Safe program.

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

(b) Unsafe program.

Figure 1: A simple example (modified from [4]).

2.2 The Abstract-Check-Refine Approach

The abstract-check-refine approach has been adopted by many previous works [2, 5, 6]. Figure 2 illustrates the naïve abstract-check-refine loop. At the *"Abstract"* stage, a set of predicates is chosen to abstract the program such that each abstracted state is represented by the truth assignments of the chosen predicates. At the *"Check"* stage, the abstracted model will be used to check the safety property. If the abstracted model is safe, the concrete model is safe also. Otherwise, an abstract counterexample is generated and it is checked whether it corresponds to a concrete counterexample. The model checker outputs the concrete counterexample if it is proved to be a real bug; otherwise, the spurious counterexample may be used to guide the *"Refine"* stage [12]. At the *"Refine"* stage, new predicates are generated and they will be used to build a new abstracted model in the *"Abstract"* stage. This loop iterates until an answer can be generated.

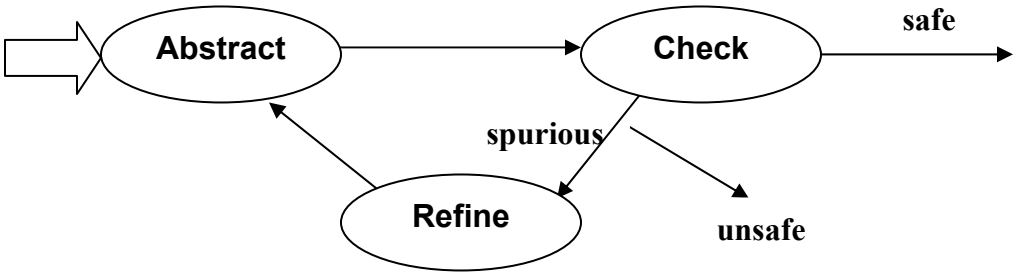
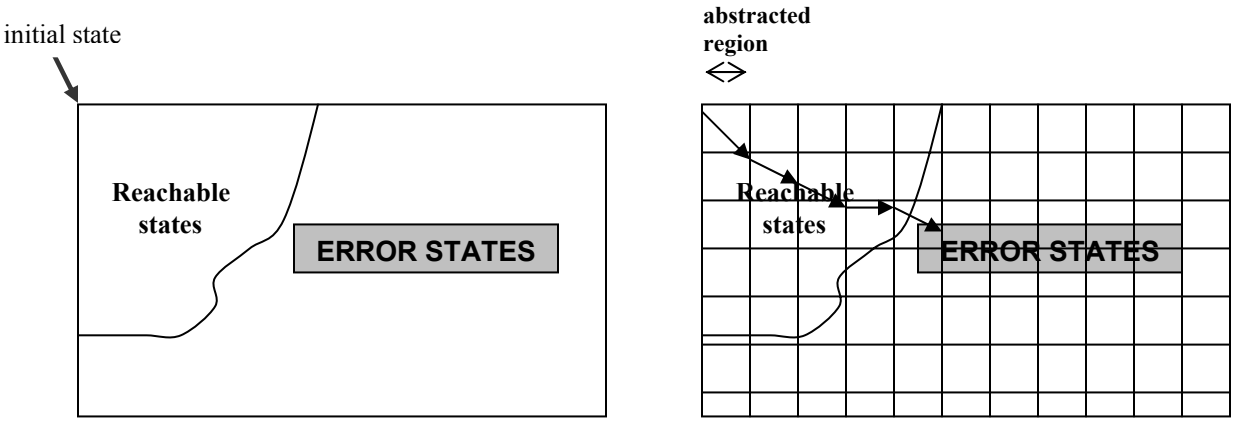


Figure 2: The naïve abstract-check-refine loop.



(a) Concrete Model (b) Abstraction Model

Figure 3: The scenario of abstraction (modified from slides of [7])

Figure 3 shows the scenario of abstraction. Here, we will define some terminologies used in the rest of the paper also. The diagram in Figure 3 (a) represents the concrete program model. Each point in the diagram represents one *state* in the concrete model. Figure 3(a) represents a safe model since the error states are not reachable. Figure 3 (b) is an abstraction of Figure 3(a). In the abstraction model, a set of states is abstracted as a *region*, which is represented as a square in the diagram. A region is an overapproximation (describing a bigger world) of a set of concrete states. From Figure 3(b), it is observed that it is possible to have a path to a region which overlaps with the error region, even though the concrete counterpart doesn't exist. This is the reason that we have to refine the model.

3 Lazy Abstraction

The lazy abstraction concept, which is proposed and implemented in the BLAST project, is aimed at optimizing the naïve abstract-check-refine loop by integrating the three steps. It means that the three steps (abstraction, checking, and refinement) are performed in an interleaving manner. The lazy abstraction is based on the following two principles:

1. ***On-the-fly abstraction*:** The naïve approach generates the entire abstract model at the *"Abstract"* stage. However, some abstracted regions may never be visited (e.g. unreachable regions). The lazy abstraction concept suggests abstracting a region only when it is needed in the next step of checking. In this case, the abstraction task is driven by the checking process. Figure 4 illustrates the scenario of on-the-fly abstraction. By comparing Figure 3(b) and Figure 4, it is observed that a lighter abstraction task is involved in lazy abstraction.

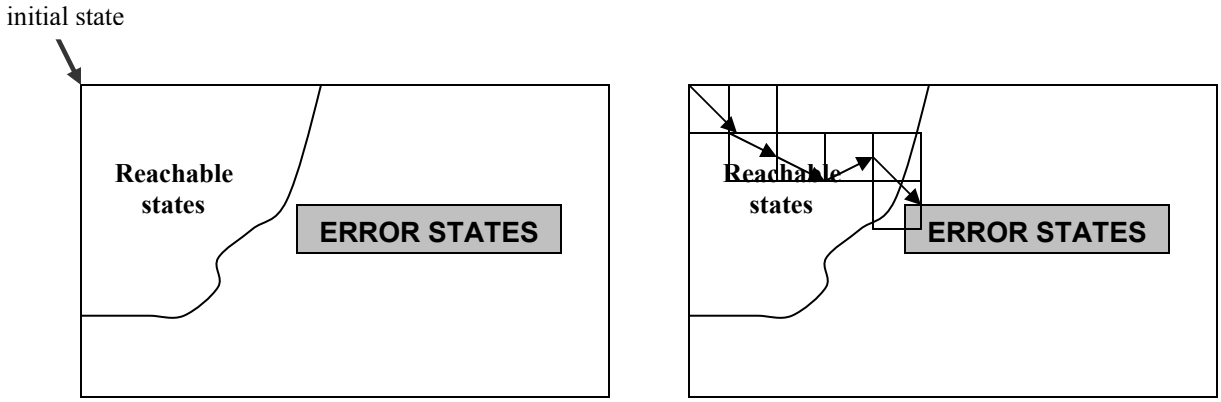


Figure 4: The scenario of on-the-fly abstraction (modified from slides of [7])

2. ***On-demand refinement:*** In the naïve approach, the entire abstract model has to be re-built after refinement. The lazy abstraction concept suggests that we can re-use the partial answer that is obtained in previous iterations. As a result, we can avoid refining those regions that have already been proved to be safe. Refinement is applied starting from the earliest state at which the abstract counterexample fails to have a concrete counterpart. This state is called *pivot state*. The identification of the pivot state becomes the new task in the new abstract-check-refine approach. Figure 5 shows the scenario of on-demand refinement. In Figure 5, two regions are involved at the first iteration and the pivot state is located at the second region being visited and the refinement starts at there also. The first visited region, which is shaded with grey, has been proved to be safe at the first iteration. As a result, we don't have to apply refinement there.

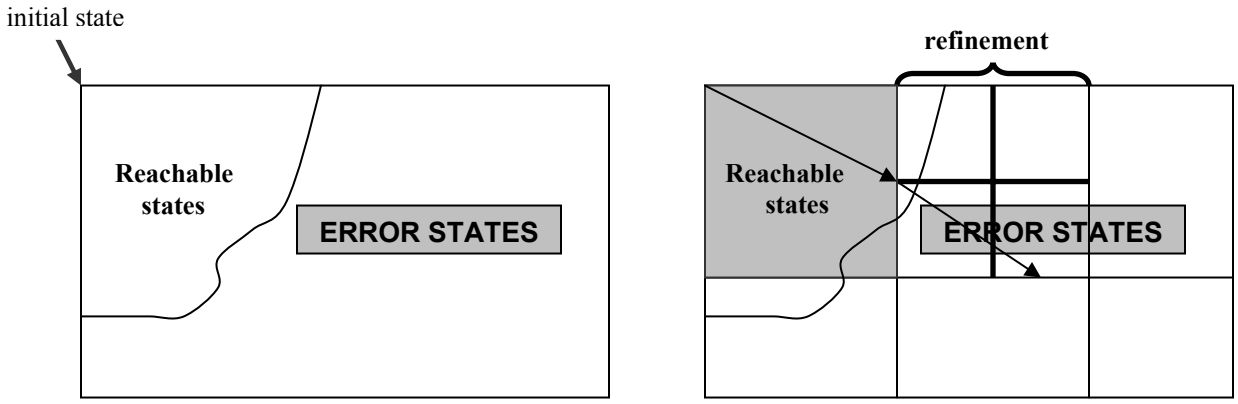


Figure 5: The scenario of on-demand refinement (modified from slides of [7])

The lazy abstraction starts from abstracting the initial region. According to the current region, the corresponding next region is abstracted. For every next region, if it reaches the `ERROR` region, we have to check whether it is a real bug. If refinement is required, the pivot state is identified and starting from which the refinement is applied, and then the abstraction and checking procedure goes on. When we check a region, we can re-use the partial answers in the past iterations, such that the region is safe if it is included inside any region that has been proved to be safe. In Figure 6, the region labeled *'A'* resides in the grey area, that has been proved to be safe. By reusing the previously proved answer (the grey area), we can determine that region A is safe without performing the model checking algorithm.

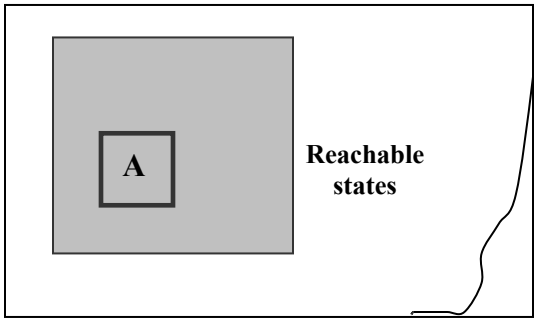


Figure 6: Reusing the partial answer on model checking.

The lazy abstraction concept applies to any abstract-check-refine model checking approach, even though it is implemented for checking C programs in BLAST. The generic lazy abstraction

algorithm is presented in [7] by representing the concrete model as a *labeled transition system* [8]. In addition, an example of applying lazy abstraction on checking a C program is also given in [7]. We don't want to "copy" the description and discussion to this survey paper, so please read [7] for the detailed algorithm. However, we will try to summarize the important concepts with the simplified example from [7] in the follows.

```
1:  do  {                          lock() {
      lock();                        if (LOCK == 0) {
      old = new;                       LOCK = 1;
2:    if (*) {                       } else {
3:      unlock();                      ERROR
        new++;                       }
      }                             }
4:  } while( new != old);
5:  unlock();                      unlock() {
    return;                          if (LOCK == 1) {
                                       LOCK = 0;
                                     } else {
                                       ERROR
                                     }
                                   }
```

Figure 7: Our example program.

Figure 7 shows the C program example used in the following discussion. The global variable `LOCK` is used to keep track of the usage of the functions `lock()` and `unlock()`. The `ERROR` label will be reached if two consecutive `lock()` or two consecutive `unlock()` is invoked at execution. The `(*)` represents a condition that is irrelevant to the checking, the `(*)` can be true or false at runtime.

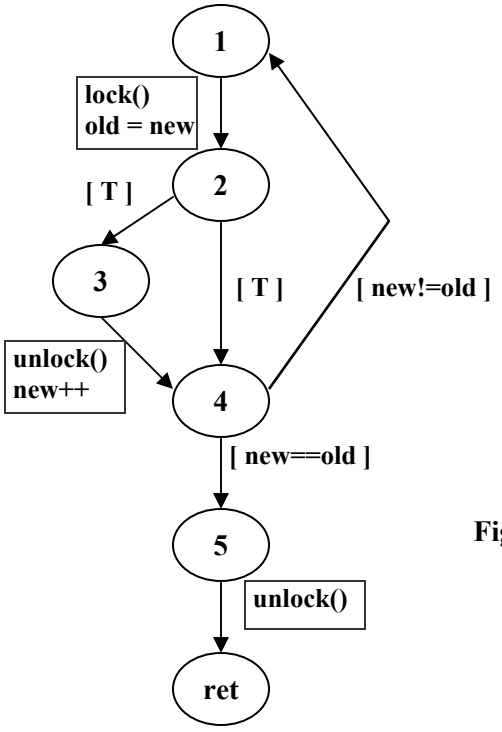


Figure 8: Control flow automaton.

First of all, the input C program is translated into a *control flow automaton (CFG)*, which is similar to a control flow graph. The edges of the graph are labeled with either an *assume predicate* corresponding to the condition that must be satisfied for that branch to be taken, or the basic block of instructions that are executed in the transition. The control flow automaton translated from the program in Figure 7 is given in Figure 8.

Given a CFG, the verification process loops through two phases: *forward search* and *backwards counterexample analysis*. A distinct predicate set for abstraction is used for each iteration. There is no new idea about predicate discovery is proposed from the BLAST project. In the implementation, the BLAST depends on predicate discovery engine from others' project. The forward search is responsible for abstraction and checking, while the backwards counterexample analysis is responsible for checking the feasibility of an abstract counterexample.

With the CFG and the predicate set, the forward search constructs in depth-first order a search tree whose nodes correspond to vertices of the CFG. Each node of the constructing tree is labeled with a formula, called *reachable regions*, which represent what is known about the state of the program with respect to the predicate set. The reachable region is computed from the reachable region of the parent node and the instructions on the corresponding edge in CFG. The forward search keeps on until we hit the ERROR region or the traversal safely finishes.

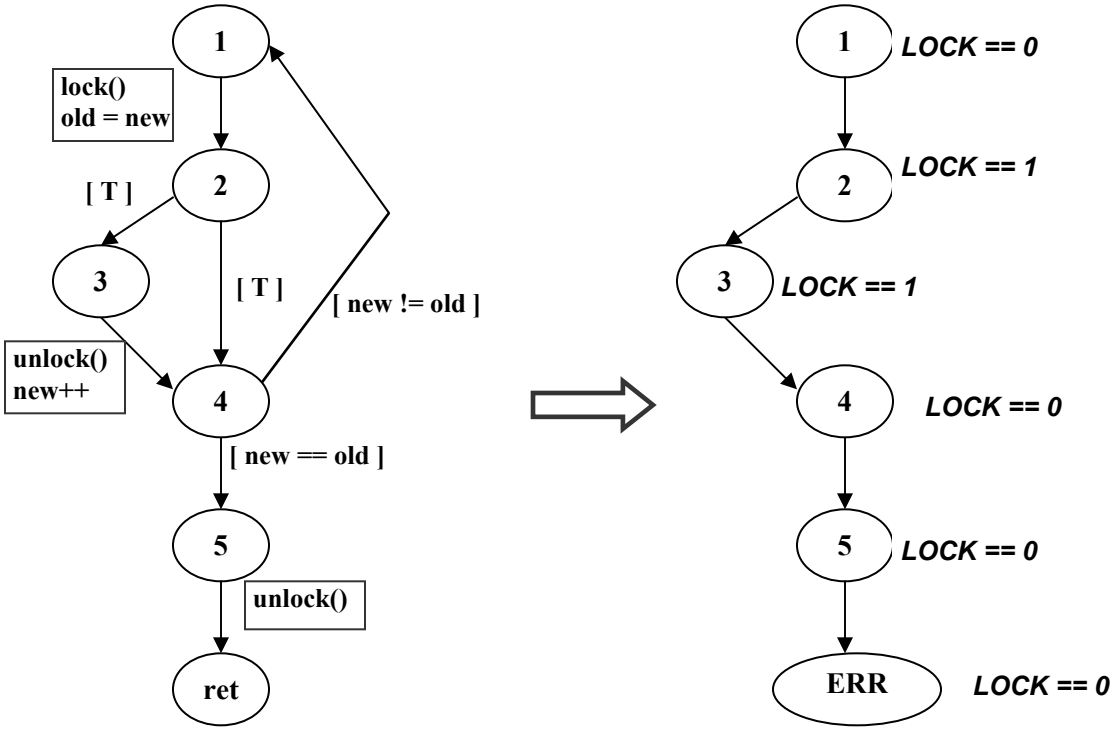


Figure 9: Forward search with predicates ([LOCK==1] and [LOCK==0]).

In our example, we start the forward search with two predicates – `(LOCK == 1)` and `(LOCK == 0)`. Figure 9 is the forward search result of our example. The reachable region, by showing the predicates whose truth value is true, is attached beside each node. The reachable region is calculated from the reachable region of the parent node and taken the effect of the edge. Since the information carried in the traversal consists of the chosen predicates only, so we traverse from node 4 to node 5 in Figure 9 as no information about `new` and `old` is available. With (`LOCK == 0`), the `ERROR` label (error node) is reached when `unlock()` is invoked after node 5.

The backwards counterexample analysis is activated when we hit an error node in the forward search process. As implied from the name, the backwards counterexample analysis is tracing from the error node back to its predecessors. At each node of the trace, the weakest precondition (in the concrete model) that would lead to the error node is calculated; it is called the *bad region*. We try to identify the first node in the search tree where the intersection of the bad region with the reachable region is empty, and this is called the *pivot node*. The pivot node is the pivot state that we mentioned previously, and it is the impossible to reach the error node from the pivot node via the given trace. According to the pivot node, new predicates are added for abstraction. From the pivot node, the verification would resume with the forward search. However, if we cannot find any pivot node before the root node is reached in the backward tracing, it means that the model is proved to be unsafe and the trace path is a concrete counterexample.

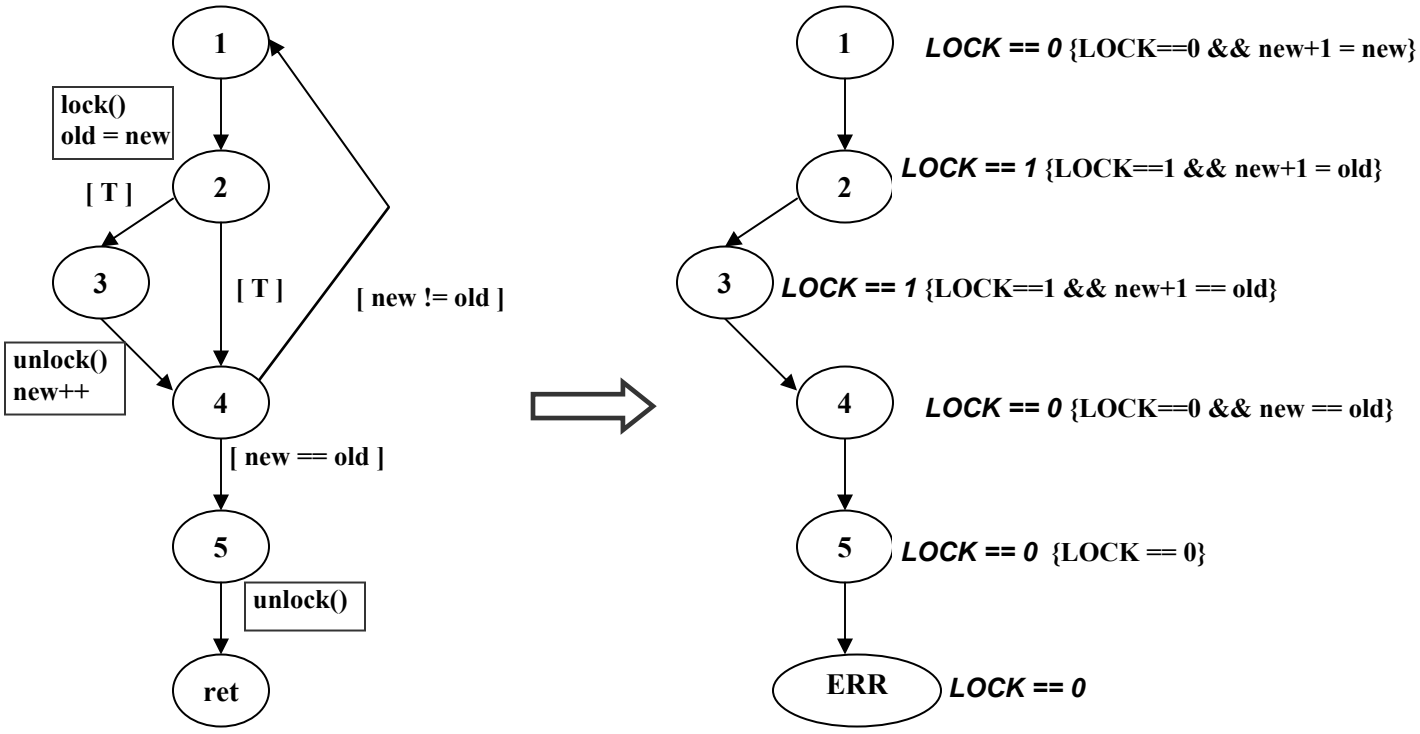


Figure 10: Backwards counterexample analysis after forward search shown in Figure 9.

Figure 10 shows the result of applying backwards counterexample analysis in our example. At each node, the bad region is beside the reachable region in a curly bracket ({ }). The backward trace stops at node 1 as it is the intersection of its reachable region and its bad region is empty. Node 1 is then identified to be the pivot node.

We have mentioned that the lazy abstraction encourages using partial answers from previous iterations. This is done in the forward search process. The reachable region of a node is *covered*, if it resides in a safe reachable region of the same node. In this case, any error found from this point on would have been found by the previous exploration. So, we can stop the forward search if the node is covered.

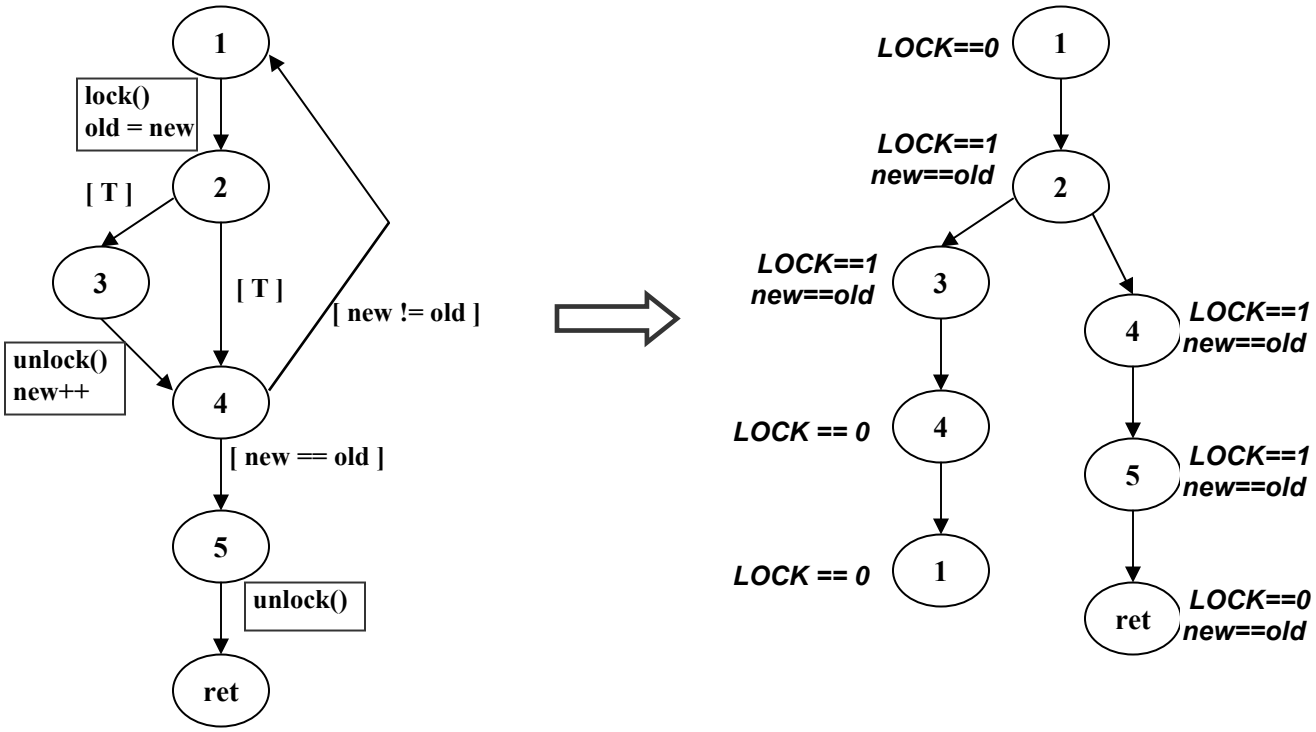


Figure 11: Forward search with new predicates ([LOCK==1], [LOCK==0] and [new==old]).

Adding a new predicate, `(new == old)`, to our example and resume the forward search. The result is illustrated in Figure 11. The forward search finishes without error this time. Because the relationship between `new` and `old` is carried in the traversal this time, the path [node 1 – node 2 - node 3 – node 4 – node 5] is not possible anymore. In the path [node 1 – node 2 – node 3 – node 4 - node 1], we can stop the traverse at node 1 because it is covered as node 1 with reachable region `(LOCK == 0)` has been proved to be safe.

It is observed that, two operations are very important to the lazy abstraction algorithm; they are *abstract post* and *concrete pre*. The abstract post operation is used in the forward search process for calculating the abstract successor of the current node. It is mentioned in [7] that the abstract post operation can be efficiently implemented by the *Cartesian abstract post* [9] operation. On the other hand, the concrete pre operation is used in the backward analysis process for calculating the concrete predecessor of a node. It can be achieved by computing the weakest pre-condition.

4 The BLAST

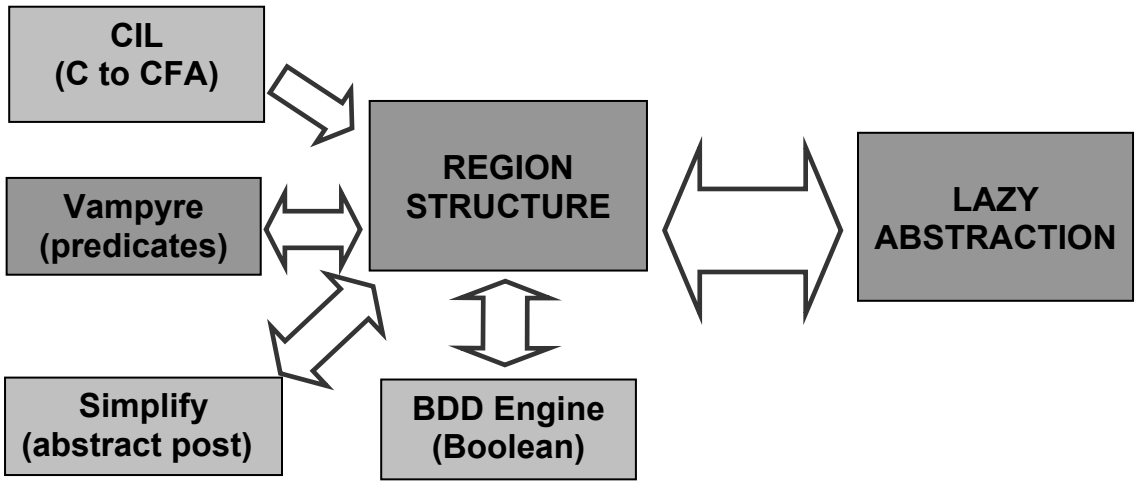


Figure 12: The BLAST architecture (modified from slides of [7])

Figure 12 gives an overview of the architecture of the BLAST system. The tool is written in Objective Caml. Some functions of BLAST are dependent to other tools. For example, the Simplify [10] is responsible for the abstract post computation and the Vampyre [11] provides predicate discovery support.

The experimental results presented in [7] are all verification of the device drivers, the complexity of which ranges from 40 lines of code to 6473 lines of code. The predicate language used in BLAST contains the quantifier-free formulas of the theory of equality with uninterpreted functions and of the theory of integers with addition.

The current release of BLAST is 1.0. We found that support of multithreading programs is not provided in the current release. Support of multithreading programs in BLAST is discussed in [13].

5 Using the BLAST

The discussion in this section is based on our experience of using the BLAST (version 1.0) on Cygwin [14]. We will discuss the use of BLAST in Section 5.1. In Section 5.2, we discuss our attempt of performing model checking with BLAST on a simple linklist package.

5.1 Model Checking with BLAST

The problem addressed by BLAST is described in Section 2.1. The basic command to run BLAST is `pblast.opt filename –main mainfunction –L ErrorLabel`, where *`filename`* is the file we want to check, *`mainfunction`* is the name of the starting function (the default is *`main`*), and *`ErrorLabel`* is an error label (the default is *`ERROR`*). The BLAST tool provides two features to facilitate the users – *static assertion checking* and *specification language*.

Static Assertion Checking

Using the `assert()` function is a common approach to check for safety at runtime in C. By providing a modified assert.h file, the BLAST allows the users to use the `assert()` function, in the way as we usually do, to check the safety properties of a program at runtime. Using the `assert()` function defined in the provided assert.h file causes branching to an `ERROR` label when the truth value of the given predicate is false.

To use the static assertion feature, the user has to produce a preprocessed file with the given assert.h file and use it for model checking. An example of using the static assertion in BLAST is given in Figure 13.

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

```
> gcc –E foo.c > foo.i
> pblast.opt foo.i –main foo
```

Figure 13: The static assertion checking.

```
......
......


No error found.  The system is safe :-)

......
......


List of predicates:

//Index = 2 Predicate:
y@foo<x@foo;

//Index = 1 Predicate:
x@foo==x@foo-y@foo;

//Index = 0 Predicate:
0<x@foo;


Number of predicates =3

Maximum number of predicates active together (discounting scope) = 3

......
......


modelCheck                      0.040 s

......
......
```

Figure 14: BLAST Output of the program in Figure 13.

The output of BLAST consists of information more than telling the users whether the program is safe. We discuss some interesting information in the follows. Showing in Figure 14 is the partial output of model checking the example in Figure 13 with BLAST. First of all, the BLAST reports that the "`The system is safe`" with one statement. Predicates that have been used in abstraction are listed at the output also. According to the output in Figure 14, three predicates are used. "`x@foo`" refers to the variable `x` in the function `foo`. The "`maximum number of predicates active together`" refers to the size of the largest predicate set used in the forward search verification process. At the end, the time consumed for model checking is reported also (time taken by each checking stage is reported in detail).

Another program modified from that in Figure 13 is checked by BLAST and the output is shown in Figure 15. This is an unsafe program, so the trace of a counterexample is reported at output. The size parameter gives the length of the counterexample trace. The numbers on the left give the line numbers in the source code for the corresponding action statement.

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

```
......
......

counterex. size:6

......
......


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_fail("x > 0", "foo.c", 6, "foo")) :: -1
77 :: 77:      FunctionCall(__blast_assert()) :: -1

......
......


Error found! The system is unsafe :-(
```

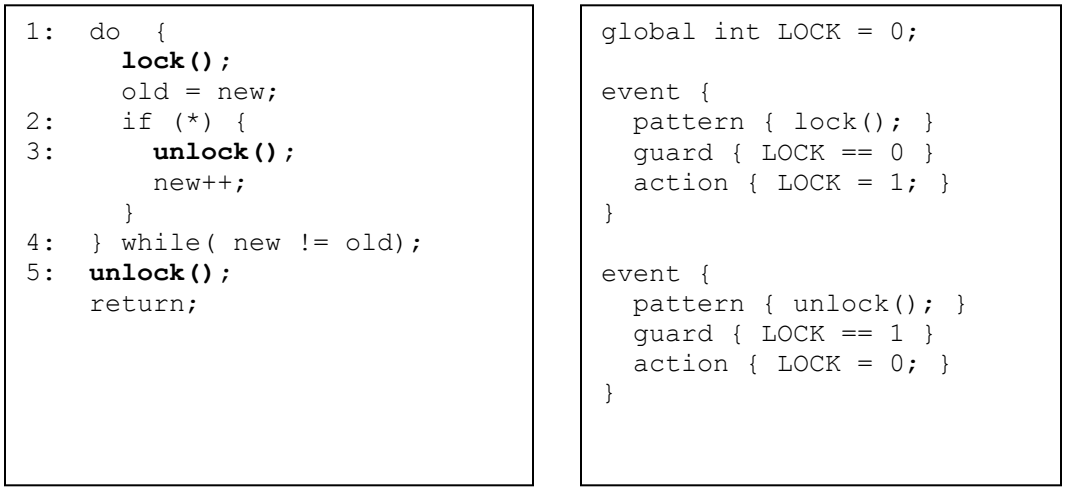
Figure 15: Counterexample output from BLAST.

Specification Language

In order to check the temporal safety properties of a program, we always need to add some observer variables and adding statements to take the corresponding values. The program in Figure 7 is an example – the observer variable `LOCK` and corresponding code modification is added to help checking that the program manipulates locks in a safe manner. To facilitate this task and especially avoid code modification, a specification language is available to allow the users specify the code modification in another file.

Consider the program in Figure 7, we use the BLAST specification language to specify the modification we wish to make on the program and the specification file (lock.spc) is given in Figure 16. The statement "`global int LOCK = 0`" defines the observer variable and initializes it to 0. There are two interesting events in this specification. The first event is raised when the pattern "`lock();`" is matched. The `guard` statement in the event specifies the condition that must be met when the event happens. The `guard` statement of the first event specifies that `LOCK` must be equal to 0. The program is regarded as *unsafe* if the guarded condition is not satisfied. The `action` statement describes the action statement associated with

the event. As specified in the first event of our example, the `LOCK` variable is set to be 1 when the event is raised. Please consult the BLAST manual [4] for additional features of the specification language.



```
1:  do  {
      lock();
      old = new;
2:    if (*) {
3:      unlock();
        new++;
      }
4:  } while( new != old);
5:  unlock();
    return;
```

lock.c

```
global int LOCK = 0;

event {
  pattern { lock(); }
  guard { LOCK == 0 }
  action { LOCK = 1; }
}

event {
  pattern { unlock(); }
  guard { LOCK == 1 }
  action { LOCK = 0; }
}
```

lock.spc

Figure 16: Using the specification language.

The BLAST comes with an instrumentation tool that takes a specification and a C program and builds an instrumented program (instrumented.c), which is the result of applying the specified code modification to the given C program. A predicate file (instrumented.pred), which contains the predicate set which guards the model checking process, is generated along with the instrumented file also. The commands to build the instrumented file and invoke the model checker for the example in Figure 16 is as follows:

```
> spec.opt lock.spc lock.c
> pblast.opt -pred instrumented.pred instrumented.c
```

5.2 Model Checking a Simple Linklist Package

We tried to do model checking with BLAST on a simple linklist package. A lot of limitations or errors hinder us trying a reasonable model.

Consider the three files in Appendix: linklist_1.h, linklist_1.spc and linklist_1.c. The specification given in linklist_1.spc is used to help checking the temporal safety properties of the program. It describes two events and one variable (`list_size`) is used in checking. The first

event describes that just before the method addNode() is invoked, we will increase list_size by 1. The second event describes that just before the method removeNode() is invoked, we have to verify (`list_size > 0`), and then decrease `list_size` by 1. The program in linklist_1.c invokes addNode() three times and followed by removeNode() once. We run it by the following command:

```
> spec.opt linklist_1.spc linklist_1.c
spec.work:1:9: warning: no newline at end of file
Parsing sources....
Spec read
Process patterns
Add definitions
Parsing
Done
> pblast.opt -pred instrumented.pred instrumented.c
```

There are one predicate generated in the instrumented.pred file – `(list_size > 0)`, and the result shows that the *"system is safe"*. This is our expected result.

In the second test, we modified linklist_1.c and made the program invoke removeNode() one more time at the end of the program. We run it with the same procedure. Two same predicates are generated in the instrumented.pred file this time – `(list_size > 0)` and `(list_size > 0)`. However, exception occurs this time with the following error trace.

```
8 :: 8:        FunctionCall(__initialize__()) :: -1
7 :: 7:        Block(list_size = 0;) :: 17
17 :: 17:      Block(Return(0);) :: -1
-1 ::          Skip :: 12
12 :: 12:      Block(list_size = list_size  +  1;) :: 7
7 :: 7:        FunctionCall(addNode(l@main, n@main)) :: -1
-1 ::          Skip :: 12
12 :: 12:      Block(list_size = list_size  +  1;) :: 8
8 :: 8:        FunctionCall(addNode(l@main, n@main)) :: -1
-1 ::          Skip :: 12
12 :: 12:      Block(list_size = list_size  +  1;) :: 9
9 :: 9:        FunctionCall(addNode(l@main, n@main)) :: -1
-1 ::          Skip :: 19
19 :: 19:      Pred(list_size>0) :: 20
20 :: 20:      Block(list_size = list_size  -  1;) :: 11
11 :: 11:      FunctionCall(removeNode(l@main)) :: -1
-1 ::          Skip :: 19
19 :: 19:      Pred(Not (list_size>0)) :: 21
21 :: 21:      FunctionCall(__error__()) :: -1
Ack! The gremlins againAck! The gremlins again!
Fatal error: exception Failure("No new preds found !-- and not running
allPreds ...")
```

From the above error trace, we can observe that list_size is incremented three times. However, the trace branches to "`Pred(Not (list_size>0))`" at the second time the removeNode() method is invoked. We tried many different similar examples and got the same problem.

Another modeling attempt is shown in the other three files of Appendix: linklist_2.h, linklist_2.spc and linklist_2.c. First, it seems that pointer operations inside methods are invisible to the BLAST, so we are forced put the statements outside the corresponding method. For example, in linklist_2.c, we define the relevant statements of the `initList()` method at `CALL_INIT_LIST(l)`. The C source code of the invoked methods must be provided at the input; otherwise, the BLAST assumes the method has no effect to the variables. It is however difficult, especially due to the fact that many C library functions are coded in assembly. In the linklist_2.c, we tried to model the effect of some library functions (e.g. `malloc()` and `strdup()`). Unfortunately, we found that it is not good enough for checking our linklist package as the value returned from the `malloc()` function does matter to the state of the link list. Generally speaking, we found that the static analysis implemented in BLAST is not good enough to keep track of the state of the link list. Without correctly keeping track the state of the link list (at least the relationship and the position of the carried nodes), it is difficult to check whether the functions (`makeNode()`, `addNode()` and `removeNode()`) in the package are being used in a safe manner.

6 Conclusion

In this survey, we discuss the lazy abstraction idea proposed and implemented in the BLAST. The concept seems theoretically applicable to the abstract-check-refine model checking approach and optimization seems promising. However, due to the instability and the incomplete of the current release, it is difficult to evaluate the ability of BLAST so far. We think the BLAST is very similar to the SLAM project by Microsoft. However, the SLAM is not free software. It hinders from evaluating the efficiency achieved by lazy abstraction through comparing the BLAST and the SLAM model checker.

References

[1] H. Chen and D. Wagner, *MOPS: an Infrastructure for Examining Security Properties of Software*. ACM Conference on Computer and Communications Security, Washington, DC, USA, Nov 2002.

[2] T. Ball, S. K. Rajamani. *The SLAM Project: Debugging System Software via Static Analysis*. Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, USA, Jan 2002.

[3] *BLAST : Berkeley Lazy Abstraction Software Verification Tool*, http://www-cad.eecs.berkeley.edu/~blast.

[4] University of California, Berkeley. *Blast's User Manual*, Oct 2003.

[5] *MAGIC : Modular Analysis of Programs in C*, http://www-2.cs.cmu.edu/~chaki/magic.

[6] *The BOOP Toolkit*, http://boop.sourceforge.net/index.html.

[7] T. A. Henzinger, R. Jhala, R. Majumdar and G. Sutre. *Lazy Abstraction*. In ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages, Portland, USA, Jan 2002.

[8] A. Finkel, S. P. Iyer, and G. Sutre. *Well-abstracted Transition Systems*. In CONCUR 00: Concurrency Theory, LNCS 1877, pp. 566-580. Springer, 2000.

[9] T. Ball, A. Podelski, and S.K. Rajamani. *Boolean and Cartesian abstractions for model checking C programs*. In TACAS 01: Tools and Algorithms for Construction and Analysis of Systems, LNCS 2031, pp. 268-283. Springer, 2001.

[10] D. Detlefs, G. Nelson, and J. Saxe. *Simplify theorem prover*. http://research.compaq.com/SRC/esc/Simplify.html.

[11] D. Blei et al. Vampyre: *A proof generating theorem prover*.

[12] E. Clarke, O. Grumberg, S. Jha, Y.Lu, and H. Veith. *Couterexample-guided abstraction refinement*. In CAV 00: Computer Aided Verification, LNCS 1855, pp. 154-169. Springer, 2000.

[13] T. A. Henzinger, R. Jhala, R. Majumdar and S. Oadeer. *Thread-modular Abstraction Refinement*. Proceedings of the 15th International Conference on Computer-Aided Verification (CAV), Lecture Notes in Computer Science 2725, Springer-Verlag, pages 262-274, 2003.

[14] *Cygwin Information and Installation*, http://www.cygwin.com.

Appendix

```
typedef struct node *Node;
typedef struct list *List;
typedef char *String;

struct list {
       Node   head;
       Node   tail;
       };

struct node {
       String str;
       Node   next;
       Node   prev;
       };

#define NULL        0
#define TRUE        1
#define FALSE       0

#define LIST_SIZE   5
#define NODE_SIZE   5
#define STR_SIZE    10

#define SOME_STR    1

void *malloc(int s);
void addNode(List l, Node n);
void removeNode(List l);
Node makeNode(String name);
String strdup(String s);
List initList(void);
```

File: linklist_1.h

```
#include "linklist_1.h"

global int list_size = 0;

event {
  pattern { addNode($1, $2); }
  action { list_size++; }
}

event {
  pattern { removeNode($1); }
  guard { list_size > 0 }
  action { list_size--; }
}
```

File: linklist_1.spc

```
#include "linklist_1.h"

int main() {
       List          l;
       Node          n;

       addNode(l,n);
       addNode(l,n);
       addNode(l,n);

       removeNode(l);
//     removeNode(l);      // add it at 2nd test

       return 0;
       }
```

File: linklist_1.c

```
typedef struct node *Node;
typedef struct list *List;
typedef char *String;

struct list {
       Node   head;
       Node   tail;
       };

struct node {
       String str;
       Node   next;
       Node   prev;
       };

#define NULL        0
#define TRUE        1
#define FALSE       0

#define LIST_SIZE   5
#define NODE_SIZE   5
#define STR_SIZE    10

#define SOME_STR    1

void *malloc(int s);
void addNode(List l, Node n);
void removeNode(List l);
Node makeNode(String name);
String strdup(String s);
List initList(void);
```

File: linklist_2.h

```
#include "linklist_2.h"

global int list_size = 0;

event {
  after
  pattern { $1 = initList(); }
  guard { $1 != NULL }
  action { list_size = 0; }
}

event {
  pattern { $? = strdup($1); }
  guard { $1 != NULL }
}

event {
  pattern { $1 = makeNode($2); }
  guard { $2 != NULL }
}

event {
  pattern { addNode($1, $2); }
  guard { $1 != NULL && $2 != NULL }
  action { list_size++; }
}

event {
  pattern { removeNode($1); }
  guard { $1 != NULL && list_size > 0 }
  action { list_size--; }
}
```

File: linklist_2.spc

```
#include "linklist_2.h"

void *malloc(int s) {
       return (void *)1;
       }

char *strdup(char *s) {
       char *clone;

       clone = malloc(STR_SIZE);

       return clone;
       }

List initList(void) {
       List          l;

       l = malloc(LIST_SIZE);

       return l;
       }

Node makeNode(String name) {
       Node          n;

       n = malloc(NODE_SIZE);

       return n;
       }

#define CALL_INIT_LIST(l)  \
    (l) = initList();      \
    (l)->head = NULL;      \
    (l)->tail = NULL;

#define CALL_MAKE_NODE(s,n)       \
    (n) = makeNode((s));   \
    (n) = node_addr++;            \
    (n)->str = s;   \
    (n)->next = NULL;      \
    (n)->prev = NULL;

#define CALL_ADD_NODE(l,n) \
    if((l)->head == NULL) {       \
        (l)->head = (n);          \
        (l)->tail = (n); }        \
    else {                                \
        (l)->tail->next = (n);    \
        (n)->prev = (l)->tail;    \
        (l)->tail = (n); } \
    addNode(l,n);
```

File: linklist_2.c

```
#define CALL_REMOVE_NODE(l)       \
    if((l)->tail != NULL) {       \
        if((l)->head == (l)->tail) {    \
           (l)->head = NULL;      \
           (l)->tail = NULL;      }      \
        else {      \
           (l)->tail->prev->next = NULL;       \
           (l)->tail = (l)->tail->prev; }      \
        removeNode(l);     }      \
    else     \
        chkRemoveNode(l);



int main() {
       List          l;
       Node          n;
       String        name;
       struct node _node_place[10];
       int node_addr = 0;

       CALL_INIT_LIST(l);

       name = strdup(SOME_STR);
       CALL_MAKE_NODE(name,n);
       CALL_ADD_NODE(l,n);

       return 0;
       }
```

File: linklist_2.c (cont'd)