Abstraction Learning

Joxan Jaffar
National University of Singapore
joxan@comp.nus.edu.sg

Jorge Navas
National University of Singapore
navas@comp.nus.edu.sg

Andrew E. Santosa
National University of Singapore
andrews@comp.nus.edu.sg

Abstract

A state-of-the-art approach for proving safe programs is *Counter-Example-Guided Abstraction Refinement* (CEGAR) which performs an "abstraction-verification-refinement" cycle by starting with a coarse abstract model that is then refined repeatedly whenever a spurious counterexample is encountered. In this paper, we present a dual approach which starts with a concrete model of the program but progressively abstracts away details but only when these are known to be irrelevant. We call this concept *Abstraction Learning* (AL). In order to deal with loops, our algorithm is encapsulated in an iterative deepening search where, because of a particular depth bound, abstraction is applied to program loops to avoid unrolling beyond the bound by building cycles. This abstraction corresponds to the strongest loop invariant we can discover. As in CEGAR, this abstraction is of a speculative nature: if the proof is unsuccessful, the abstraction is removed and another attempt using a deeper bound is initiated.

A key difference between AL and CEGAR is that AL detects more *infeasible paths* in the state-space traversal phase. We argue that this key feature poses unique benefits, and demonstrate the performance of our prototype implementation against the state-of-the-art BLAST system.

1. Introduction

The increasing complexity of industrial software systems urges the development of verification techniques to identify errors or prove programs safe. In the process of verification, there are two main desired characteristics: accuracy (i.e., no false positives) and scalability. In general, one feature is achieved at the expense of the other. One remedy to this fundamental problem is *abstract interpretation* [15] that attempts scalability by eliminating irrelevant details to the property of interest. However, arbitrary abstraction is ineffective since the lack of control on the loss of information easily results in many false positives.

CounterExample-Guided Abstraction Refinement (CEGAR, or more briefly, AR) [2, 14, 34], has been a very successful technique for proving safe programs. This technique starts with a coarse abstraction that represents an abstract model of the program (*abstraction phase*). The abstract model is checked automatically for the desired property (*verification phase*). If no error is found, then the program is safe. Otherwise, an abstract counterexample is produced which shows how the abstract model violates the property. The counterexample is then analyzed to test if it corresponds to a concrete counterexample in the original program. If this is the case, a real error has been found and the program is reported as unsafe. Otherwise, a *counterexample-driven refinement* is performed to refine the abstract model such that the abstract counterexample is excluded (*refinement phase*) and the process starts again. Several systems have been developed during recent years following this approach such as SLAM [3], Magic [12], BLAST [10], ARMC [32], and Eureka [1].

In this paper, we present a *dual* algorithm to AR, called *Abstraction Learning* (AL). Essentially, this technique starts with the most possible precise abstraction: the concrete model of the program. Then, the concrete model is checked for the desired property (*verification phase*). If a counterexample is found, then it must be a real error and hence, the program is unsafe. Otherwise, the program is safe. In order to achieve scalability during the verification phase, our technique abstracts the model by learning the facts that are irrelevant to refuting error states (*learning phase*), and then it eliminates those facts from the model (*abstraction phase*).

Since a program may contain loops, the above process needs to be run on the top of an *iterative deepening* search described as follows. For a given depth, an abstraction is computed and used to generalize the states at the traversed looping points (program points where the merging of control paths construct some cyclical paths). Although the abstraction is nevertheless an over-approximation, our algorithm attempts to minimize the loss of information during the process. The abstraction constitutes what is known as a *loop invariant*. Typical generation of loop invariants employ advanced mathematical reasoning (e.g., in [8]). In contrast, our computation of the abstraction is *lightweight* as it is computed by simple syntactic manipulation of constraints. These *speculative* abstractions for loops may be still too coarse to establish safety. In this case, the depth-bounded search is run repeatedly increasing the depth limit and executing again the *verification-learning-abstraction* process. More importantly, the facts needed to exclude the counterexample are kept in subsequent iterations. This iterative deepening search will eventually either discover loop abstractions precise enough to verify the program, find a true counterexample, or runs forever.

The rationale behind of the iterative deepening algorithm is to be as tight as possible to the concrete domain while still making the verification phase finite. At this point, it is clear that our treatment of loops resembles AR where an abstraction is obtained via refinement in the hope of being invariant through loops. If not, AR needs to refine on demand parts of the abstract model which have been already constructed. However, a fundamental distinction is that AL attempts always to construct the most precise abstraction for loops by searching for the strongest lightweight loop invariants.

In summary, the main principle of design of the verification-learning-abstraction loop performed by AL is to be as accurate as possible with respect to the concrete model in order to perform an earlier detection of *infeasible paths* during the verification stage. This principle is in opposition to AR in which many actual infea-

sible paths may be considered since they can be feasible under the current coarse abstract model. While there are clearly some benefits of using the concrete model, it is also clear that the investment in added work for accurately reasoning about paths may not always pay off. Our thesis is that this investment often pays off, and even in examples where it does not, it is affordable. We will discuss and experimentally evaluate the performance issues using academic examples against the BLAST and ARMC systems, and we will use real benchmarks (in fact these are BLAST benchmarks) in comparison with the BLAST system.

1.1 Related Work

Our work is related to counterexample-guided abstraction refinement (CEGAR) [1, 2, 14, 20, 21, 34], which perform successive refinements of the abstract domain to discover the right abstraction to establish a safety property of a program. An approach that uses interpolation to improve the refinement mechanism of CEGAR is presented in [20, 27]. Here, interpolation is used to improve the method given in "Lazy Abstraction" [21], by achieving better locality of abstraction refinement. Ours differs from CEGAR formulations in some important ways: first, instead of *refining* abstract states, we *abstract* a set of concrete states. In fact, our algorithm abstracts a state after the computation subtree emanating from the state has been completely traversed so that infeasible paths can be detected, and the abstraction is constructed from a *learning* phase where the interpolations of the constraints along the paths in the subtree are generated. A second difference is: our algorithm interpolates a *tree* as opposed to a *path*. More importantly, our algorithm only traverses *spurious paths* due to abstractions done at looping points, unlike CEGAR where abstraction is done everywhere. We shall exemplify these differences with some academic examples in Sec. 3 and in a comparison with the BLAST tool [10] in Sec. 6.

Recently, there have been some interesting works that use test-generation features to enhance the process of satisfying properties. Synergy [19] carries a forest of tests cases in order to find quickly a counterexample through loops with deterministic choices where BLAST-like tools may fully unroll the loops. DASH [7] uses only test generation operations, and it refines and maintains a sound program abstraction as a consequence of failed test generation operations. By doing so, expensive calls to the theorem prover can be avoided during the refinement phase. The previous idea is improved and extended for intraprocedural programs by SMASH [18]. We consider that these ideas can be also applied to our main algorithm.

An important alternative method for proving safety of programs is translating the verification problem into a Boolean formula that can then be subjected to SAT or SMT solvers [4, 5, 24, 25, 30, 36]. In particular, [11] introduces bounded model checking, which translates *k*-bounded reachability into a SAT problem. While practically efficient in case when the property of interest is violated, this approach is in fact incomplete, in the sense that the state space may never be fully explored. An improvement is presented in [28], which achieves unbounded model checking by using interpolation to successively refine an abstract transition relation that is then subjected to an external bounded model checking procedure. Techniques for generating interpolants, for use in state-of-the-art SMT solvers, are presented in [13]. The use of interpolants can also be seen in the area of theorem-proving [29].

Our work is also related to various no-good learning techniques in CSP [17] and conflict-driven and clause learning techniques in SAT solving [6, 11, 31, 35]. These techniques identify subsets of the minimal *conflict set* or *unsatisfiable core* of the problem at hand w.r.t. a subtree. This is similar to our use of interpolation, where we generalize a precondition "just enough" to continue to maintain the verified property.

```
main() {                main() {                main() {
  x=0;                    x=0;                  lock=0; new=old+1;
0: if (*) x++;          0: if (*) x++;          flag=1;
1: if (*) x+=2;         1: if (y>=1) x+=2;    1: while(new!=old){
2: if (*) x+=4;         2: if (y<1) x+=4;     2: lock=1; old=new;
3: if (x>7)             3: if (x>5)           3: if(*)
4:   error();           4:   error();         4:   lock=0; new++;
}                       }                       }
                                              5: if(!flag)
                                              6:   lock=0;
                                              7: if(lock==0)
                                              8:   error();
                                              }
        (a)                     (b)                     (c)
```

Figure 1. Three Example Programs

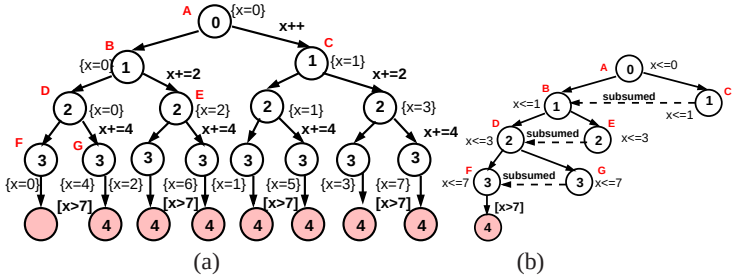


Figure 2. Interpolation and Subsumption of Feasible Paths in Program 1(a)

Finally, our closest related work has been recently presented in [23]. They consider the problem of exploring the search tree of a CLP goal in pursuit of a target property. Although some of the main features of our algorithm such as detection of infeasible paths, interpolation, and subsumption are already presented, the algorithm does not perform automatic abstractions for loops which we consider the core of our algorithm proposed in this paper.

2. The Basic Idea

Our algorithm traverses the execution tree of the programs while attempting to find an execution path that reaches the `error()` function. If such path cannot be found, the it concludes that the program is safe.

Let us first consider the program in Fig. 1(a). Here the $(*)$ symbol denotes a condition of nondeterministic truth value. The execution of the program starting in states satisfying `x=0` results in the execution tree shown in Fig. 2(a). A node corresponds to a location (i.e., program point) and can be labeled with $\{.\}$ representing the state at that particular location or without $\{.\}$ representing its *interpolant* (see below). We may annotate the node with a capital letter so that we can refer to it without ambiguity. The node is drawn with filled circle if the path reaching it is infeasible. An edge represents the control flow from one location to another and can be labeled with statements (predicates are denoted by [.]).

Suppose that the error condition at $\langle 4 \rangle$ is `x>7`. The algorithm performs a depth-first traversal, first executing the path from A to F symbolically by strongest postcondition computation to discover the constraints that hold at every point along the path. From this, we label F with the state `x=0` which falsifies the error condition. However, a more general formula, say `x<=5` would accomplish the same. The constraint `x<=5` is an interpolant, since it is entailed by `x=0` and it falsifies `x>7`. We could use it to generalize the label of F, however, we rather use as general an interpolant as possible, and clearly in this case, it is `x<=7`. Hence, in Fig. 2(b) we replace the

label of F (x=0) with the interpolant x<=7. In this way, node G with state x=4 (which has not yet been traversed) is now *subsumed* by F with the new interpolant (since x=4 satisfies x<=7). Here we conclude that it is not possible to find error() by expanding G and the algorithm backtracks. Here we can again generate an interpolant for G such that it remains subsumed by F. The most general is again x<=7, which we use to label G in Fig. 2(b).

We next use the interpolants of F and G to produce a generalization of D. We first compute candidate interpolants from the interpolants of F and G, w.r.t. the state transitions from D to F and from D to G. The final interpolant of D is the conjunction of these candidate interpolants. In this process, we first rename the variables of F and G with their primed versions, such that F and G both have the label x′<=7. First consider the transition from D to F, which is in fact equivalent to a skip statement, and hence it can be represented as the constraint x′==x. It can be easily seen that the label x==0 of D ensures the unsatisfiability of x′==x, x′>7. Here again we compute an interpolant, one that is entailed by x==0 but refutes x′==x, x′>7. As interpolant, we again choose x<=7[1].

Similarly, considering the goal reduction D to G as the augmentation of the constraint x′=x+4, we obtain a candidate interpolant x<=3 for D. The final interpolant for D is the conjunction of all candidates, which is x<=7 ∧ x<=3 which in turn is equivalent to x<=3. We label D with this interpolant in Fig. 2(b). In this way, E is now subsumed by D, and its traversal is not necessary.

We then generate an interpolant for E in the same way we did for G. By repeating the process described above for other nodes in the tree, we obtain the smaller tree of Fig. 2(b), which is linear in the size of the program. This tree represents the part of the symbolic computation tree that would *actually be traversed* by the algorithm. Hence, while the tree's size is exponential in the number of **if** statements, our algorithm prunes significant parts of the tree.

2.1 With Infeasible Paths

Now consider the safe program in Fig. 1(b). The error condition is x>5 and the error location is at point ⟨4⟩. We depict in Fig 3(a) the naive tree if interpolation and subsumption are not applied, and in Fig. 3(b) a smaller tree, which still proves the absence of bugs, after using interpolation and subsumption of infeasible paths.

A key principle of our algorithm is that it *preserves the infeasibility* of paths during the traversal. Now let us focus on Fig. 3(b) and consider, for instance, the path ending at F which is detected as infeasible ($x = 0 \wedge y < 1 \wedge y \geq 1$). Applying our infeasibility preservation principle, we keep F labeled with *false*, and the only possible interpolant for F is *false* itself. This would produce the interpolant y<1 at D since this is the most general condition that preserves the infeasibility of F. Note that here, y<1 is entailed by the original state x=0 ∧ y<1 of D and in turn entails y>=1 $\models$ *false*, which is the weakest precondition of *false* w.r.t. the negation of the **if** condition on the transition from D to F.

Now consider G with x=4 ∧ y<1 and note that it satisfies x<=5. G can be interpolated to x<=5. As before, this would produce the precondition x<=1 at D. The final interpolant for D is the *conjunction* of x<=1 (produced from G) and y<1 (produced from F). In this way, E cannot be subsumed by D since its label does not satisfy both the old and the new labels of D.

2.2 Loops

We have previously explained the basis of our algorithm on acyclic execution paths. We now explain how our algorithm handles loops. In essence, our algorithm automatically infers loop invariants based on information learned during traversal. The constructed loop invariant for a given loop is a conjunction of constraints whose truth values remain unchanged after one or more iterations of the loop. Similar to AR, this process may require abstraction refinements in the case the abstraction is too coarse to prove the safety property.

[1] This interpolant corresponds to the *weakest precondition* [16] w.r.t. the statement x=x; and the target property x<=7, however, in general the obtained precondition need not be the weakest, as long as it is an interpolant.

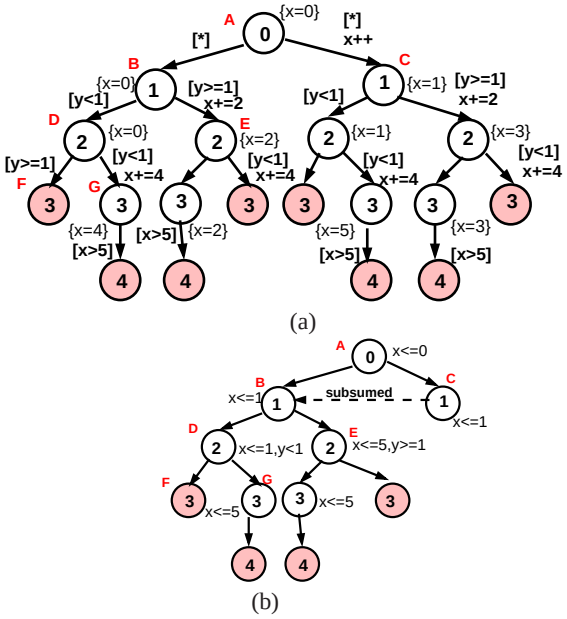


Figure 3. Interpolation and Subsumption of Infeasible Paths in Program 1(b)

We use a slightly modified version of the running example in [21] in order to illustrate the main features of our algorithm. We present the example in Fig. 1(c). Our modification consists of the addition of the variable flag, initialized to 1, and an **if** condition on Lines 5 and 6. Together with the program we depict the relevant paths explored by our algorithm in Fig. 4.

Our algorithm performs a depth-first traversal of the tree from left to right. The first path explored is ⟨0⟩,⟨1⟩,⟨2⟩,⟨3⟩,⟨1'⟩ (Fig. 4(a)) denoting a cyclic path from location ⟨1'⟩ back to ⟨1⟩. Our algorithm examines the constraints at the entry of the loop (i.e., lock=0, new=old+1,flag=1) to discover those whose truth values remain unchanged after the loop (i.e., lock=1,new=old,flag=1). Clearly, the constraints lock=0 and new=old+1 are no longer satisfied while flag=1 still holds.

At this point, our algorithm produces an abstraction at the location ⟨1⟩ by making the truth values of lock=0 and new=old+1 unknown. In this way, the constraints at ⟨1'⟩ now entails the modified constraints of ⟨1⟩ that consists of a single constraint flag=1.

The next path is depicted in Fig. 4(b) and contains the locations ⟨0⟩,⟨1⟩,⟨2⟩,⟨3⟩,⟨4⟩,⟨1"⟩. At ⟨1"⟩, the constraints already entail the generalized constraint of ⟨1⟩, and therefore, for the same reason as ⟨1'⟩ we stop the traversal.

After the loop is traversed, the remaining constraint at ⟨1⟩ is flag==1 and this is in fact a loop invariant discovered by the algorithm. Since we have removed new==old+1 from ⟨1⟩, the exit path of the loop now becomes feasible as the condition new==old becomes satisfiable. For this reason the traversal reaches ⟨5⟩ with the constraint flag==1 propagated from ⟨1⟩ and new==old which is obtained by strongest postcondition propagation through the loop exit transition.

Since we keep flag==1 in the loop invariant at ⟨1⟩, the algorithm manages to reason that the path ⟨1⟩, ⟨5⟩, ⟨6⟩ is infeasible

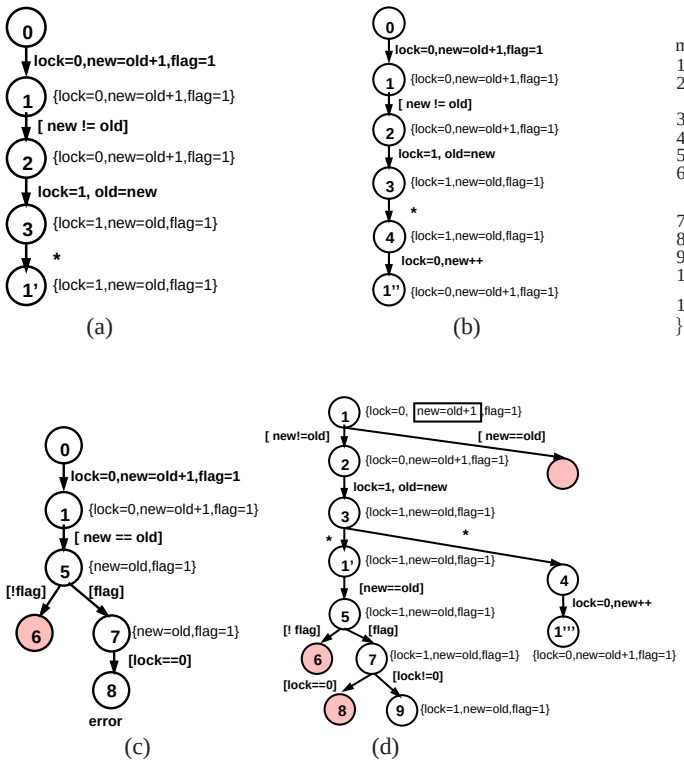


Figure 4. Execution Paths of Program in Fig 1(c)

(Fig. 4(c)). The crucial point here is that the algorithm exits the loop with as much loop invariant information as possible. This, in effect is an attempt to preserve as many infeasible paths as possible. It is worth mentioning that a counterexample-guided refinement tool would not detect that infeasible path and will reach the error location later.

The algorithm next visits the nodes $\langle 7 \rangle$ and $\langle 8 \rangle$ also in Fig. 4(c), which is an error location. The path is actually spurious, and the algorithm discovers that the reason for the reachability of this point is the removal of `new==old+1` at $\langle 1 \rangle$. This is because the edge from $\langle 1 \rangle$ to $\langle 5 \rangle$ cannot be executed when `new==old+1` is considered. After knowing this, the algorithm *locks* `new==old+1` at $\langle 1 \rangle$ and restart the traversal from $\langle 1 \rangle$. The purpose of the locking of a constraint is to declare that the constraint cannot be removed for the purpose of generating loop invariant.

The next traversal after the locking is depicted in Figure 4(d). The locked constraint is enclosed in a box. Similar to the first traversal, the path $\langle 1 \rangle$, $\langle 2 \rangle$, $\langle 3 \rangle$, $\langle 1' \rangle$ is again re-traversed. At $\langle 1' \rangle$, the constraints do not entail the constraints of $\langle 1 \rangle$ anymore. Whereas in the first traversal we can obtain a candidate loop invariant by the removal of `lock==0` and `new==old+1`, here, due to locking of `new==old+1`, we are prevented from generating a loop invariant. As the result, the traversal simply continues, and it manages to complete the traversal without visiting the error location which is infeasible.

3. Comparison with Counterexample-Guided Abstraction Refinement

In this section we present informally several academic examples in order to highlight some essential differences between AR and AL. We will use the BLAST the ARMC systems as examples for AR

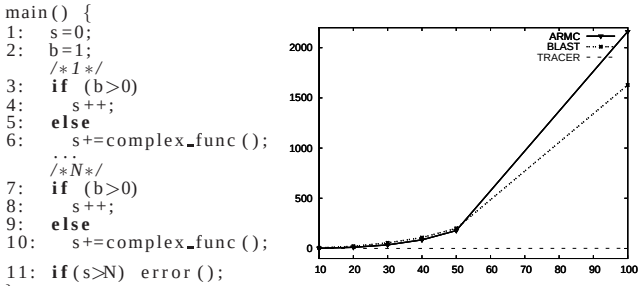


Figure 5. Bad for AR, Good for AL: Exploration of Infeasible Paths

here, and use as our own prototype TRACER to exemplify AL. We consider real benchmarks later in Section 6.

3.1 Exploration of Infeasible Paths

The core idea of abstraction refinement with predicate abstraction is to be as coarse as possible until a counterexample is found. If the error is spurious, it extracts from the path some predicates which demonstrate this. By minimizing the use of these predicates, one can minimize the search space. On the other hand, minimizing the use of predicates gives rise to the possibility of exploring *infeasible paths*. This causes two problems: first, the verification stage of the abstraction-verification-refinement cycle is expensive, even exponential, in terms of the number of predicates. Second, the process of refinement (in order to discover yet more predicates) is also expensive. This is perhaps the major shortcoming for counterexample-guided refinement tools. We quote "*A challenge is how to perform an efficient analysis of an spurious counterexample and learn from it a small set of facts such that the refined abstraction does not contain the spurious error*" [20].

Our first program is described in Fig. 5 (a) and illustrates the expenses of exploring infeasible paths. A counter variable `s` is initialized to 0 together with another variable `b` which is given 1 (Lines 1-2) [2]. Then, the following code is repeated N times: if `b` is positive then `s` is incremented by 1 through the statement `s++` (Line 4). Otherwise, assume that it is also incremented by 1 but through a complex function `complex_func` (Line 6). After all the if's are considered, the value of `s` is checked for being greater than `N` (Line 11). A counterexample-guided refinement tool based on predicate abstraction [10, 32] will discover the $N+1$ predicates for the case `b` is positive: $(s = 0)$, $(s = 1)$, $(s = 2)$, $\ldots$, $(s = N)$, one by one before checking the error is not reachable. Similarly, it will add the predicates corresponding to the case when `b` is non-positive (Lines 6-...-10). However, notice that this code will be never executed since it lies on infeasible paths.

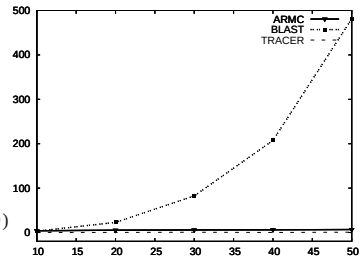
AL will start the traversal considering the concrete state $s = 0 \wedge b = 1$. A crucial distinction with respect to AR is that AL will not consider the code when `b` is non-positive (Lines 6-...-10) since it is able to detect those infeasible paths (i.e., $s = 0 \wedge b = 1 \wedge b \leq 0$ is unsatisfiable). A performance comparison, for this example, is shown in Fig. 5(b). A proof of absence of bugs was achieved with different values of N where N is the number of times the code in Lines 3-6 is repeated. The horizontal axis represents the different values of N and the vertical axis the total time in seconds.

[2] The ARMC [32] tool performs some constant propagation optimization so we hide `b=1` to disable standard compiler optimizations. For simplicity, we do not show the full code.

3.2 Using Newly Discovered Predicates in Future Traversal

Another fundamental question in AR: after the set of predicates required to exclude the spurious counterexample has been discovered, how should those predicates should be used in other paths? We quote: "*Most predicates are only locally useful, i.e., only useful when analyzing certain parts of the program, and irrelevant in others. If locality is not exploited, then the sheer number of facts may render the abstract system too detailed to be amenable to analysis, as the size of the abstract system grows exponentially with the number of predicates.*" [20]

```
main() {
1:   s=0;
2:   if(*) z=0;
3:   else   z=999;
     // 1
4:   if(*) s++
5:   else   s+=2;
     ...
     // N
6:   if(*) s++
7:   else   s+=2;
8:   if(s+z>2*N && z==0)
9:     error();
}
```



(a) (b)

Figure 6. Bad for AR, Good for AL: Locality Not Exploited Properly

Consider our next program in Fig 6(a). The program initializes `s` to 0 (Line 1). Given a non-deterministic choice it assigns the variable `z` to either 0 (Line 2) or 999 (Line 3). After that, the program executes N times the following code: given another non-deterministic choice `s` is incremented by 1 (Lines 4-...-6) or by 2 (Lines 5-...-7). Finally, the program checks the error condition `s+z> 2*N && z==0` at Line 8. A counterexample-guided tool will discover the predicates $(s=0),(s=1),\ldots,(s=2*N)$. Then, it will either add $(z=0)$ or $(z=999)$ depending on the traversal order. W.l.o.g assume that it first discovers the predicate $(z=0)$. The key observation is that all the paths that include $(z=999)$ (Line 3) will be traversed considering all the predicates discovered from paths that include $(z=0)$ (Line 2), and hence, the traversal will be prohibitively expensive.

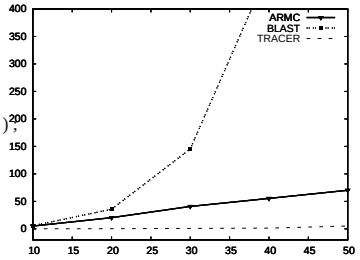
Now, we explain how our AL algorithm verifies this program without facing the same problem than AR. Our algorithm will basically perform the same amount of work for the case $z=0$. However, it traverses the paths that include $z=999$ without consideration of the facts learnt from paths that include $z=0$ since it only keeps track of the concrete state collected so far (i.e., $s=0 \wedge z=999$). Then, after a path is traversed (e.g., 1-3-4-...-6-8) our algorithm can discover in a straightforward manner that $z=999$ suffices to refute the error state and hence, the rest of the paths will be subsumed. It is worth mentioning that AR will also discover the predicate $(z=999)$ after the counterexample is found. The essential difference, for this example, is that the predicates discovered previously ($(s=0),(s=1),\ldots,(s=2*N)$) are used, and hence, the traversal will be significantly affected by them.

Another performance comparison, for this program, is shown in Fig. 6(b). The program was proved safe with different values of N where N is the number of times the code in Lines 4-5 is repeated. The x-axis represents the values of N and in the y-axis the time in seconds to traverse all the paths that contain `z=999` (Line 3).

Note that ARMC does not suffer from the same problem than BLAST in this program. In fact, it is comparable to TRACER in this example. We believe that after ARMC discovers predicates for the current path it is very conservative when it comes to considering them for other paths. Moreover, ARMC performs more propagation than BLAST. More discussion in the next contrived program.

3.3 Running an Abstract State Hampers Subsumption

```
main(){
1:   if(*){
2:     x=0;
3:     y=0;
     }
4:   else{
5:     x=complex_function();
6:     y=0;
     }
7:   s=x;
8:   t=y;
     /*1*/
9:   if(*){s++;t++;}
     ...
     /*N*/
10:  if(*){s++;t++;}
11:  if(t>N && s>N) error();
}
```



(a) (b)

Figure 7. Bad for AR, Good for AL: Coarseness means Less Subsumption

The next example illustrates another potential weakness of AR that is not present in AL. Even if locality is well exploited, the likelihood of subsuming the *currently traversed* state may be diminished because the state, being abstract, is too coarse. Consider now the program in Fig. 7(a). Given a non-deterministic choice, `x` and `y` are assigned to 0 (Lines 2-3). For one of the choices `x` is assigned to 0 through a complex function (Line 5) and for the other it is assigned directly (Line 2). The use of a complex function in this example is to avoid propagation of information performed in practice by CEGAR tools, even though this feature is not part of the basic principles of AR. In the next statements, the variable `s` is assigned to `x` (Line 7) and `t` to `y` (Line 8). Then, the following code is executed N times: given another non-deterministic choice `s` and `t` are incremented by 1 (Lines 9-...-10). Finally, the condition `t>N && s>N` is checked at Line 11.

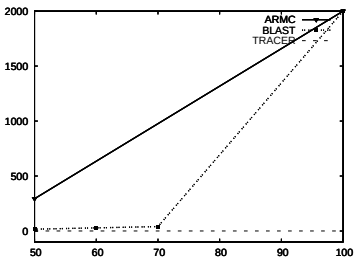
In principle, a counterexample-guided tool will behave very similarly as in the program in Fig. 6(a). Assume again w.l.o.g. that the if-branch is first taken (Lines 1-3). After that, it will discover the predicates $(x=0),(s=0),(s=1),\ldots,(s=N),(y=0),(t=0),(t=1),\ldots,(t=N)$. Again, those predicates will be likely used during the exploration of the else-branch (Lines 4-6). However, an essential difference with respect to program in Fig. 6(a) is that although the discovered predicate (x=0) is taken into consideration, the abstract state cannot be covered since it is too coarse and does not entail the predicate due to the loss of information caused by the complex function. Therefore, the rest of paths will be explored with the previously discovered predicates.

In contrast, AL does perform a systematic propagation of the program state along the whole program. Therefore, even if `x` is assigned 0 through a complex function it will be able to know the resulting value. The main consequence is that the state now will be subsumed.

In Fig. 7(b) we depict the result of the verification of the program in Fig. 7(a) with different values of N, on the horizontal axis, where N is the number of times the code in Lines 9-10 is duplicated. The vertical axis is the time (sec) of exploring the paths that contain the statement `x=complex_func()` (Line 5).

Here again, the performance of ARMC is comparable to TRACER although with an important overhead. The reason is simple: ARMC can propagate the value of `x` through the `complex_function` and

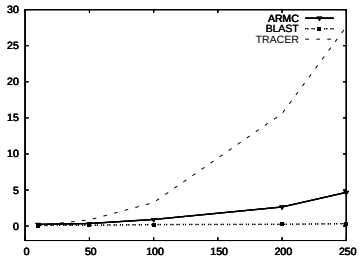
```
main(int N) {
1:    assume(y >= 0);
2:    n = 0;
3:    while (n < N) {
4:      y++;
5:      n++;
      }
6:    if (y+n < N)
7:      error();
}
```



(a) (b)

Figure 8. Bad for AR, Good for AL: No Loop Invariant Discovery

```
main(){
1:  if(rand()>0.5) s+=2;
2:  if(rand()>0.5) s+=3;
    ...
3:  c=1;
4:  if(c==0) error();
}
```



(a) (b)

Figure 9. Bad for AL, Good for AR: Unnecessary Infeasible Paths If Trivial Safety

hence, it is not as abstract as BLAST and all paths that contain the else-branch (Lines 4-6) will be subsumed.

3.4 Discovering Loop Invariants

Another major characteristic of AR lies in the simple treatment of loops.

We now address the treatment of loops. Any symbolic traversal method will have to eventually discover, implicitly or explicitly, loop invariants that are strong enough for the proof process to conclude successfully. In the case of AR, the abstract model is refined from spurious counterexamples by discovering which predicates can refute the error path, and in this process, it is *hoped* that these predicates will in fact will be invariant through loops. (If not, then loop unwinding will be performed. This process may not terminate.) A crucial observation is that the inference of invariant predicates can speedup significantly the convergence of loops. However, AR does not have a systematic methodology for searching for those invariant predicates although each AR tool implements multiple heuristics that work in practice very often. This topic is widely treated in [8] but without experimental evaluation. Moreover, the solution relies basically on an external loop-invariant generator to guide the refinement phase. We consider the use of these tools an orthogonal issue since both AL and AR would benefit from it.

This challenge motivates another principal design consideration of AL. We take the philosophy that we should discover the *strongest* loop invariant, in accordance with the philosophy that AL uses the concrete model during normal straight-line traversal of paths. In AR terminology, this may be described as choosing *as many* predicates as possible. In contrast, AR, in accordance with its basic philosophy of choosing the fewest predicates in straight-line paths, essentially adopts a *weakest invariant* philosophy. Note that the AR approach to loops is an implicit one because it focusses on conterexamples *alone*, whereas in AL, there is also focus on predicates that are loop invariant. Our next program exhibits a bad behavior of AR when invariant predicates are not considered during the refinement phase. Conversely, AL searches for invariant constraints to strengthen the abstract model as much as possible. The program in Fig. 8 (a) illustrates the benefits. Given an input finite bound `N` and assuming that a variable `y` is always non-negative, a while loop is executed N times. The body of the loop increments by one the variables `y` and `n`. After the loop, the condition $y+n<N$ is checked.

AR tools will discover the predicates $(n=0),(n=1),\ldots,(n=N-1)$ and also $(y=0),\ldots,(y=N)$. The key point is that they perform a full unwinding of the loop. To understand why AL avoids the full unwinding is essential the concept of inference of invariant constraints. Consider the path inside the loop until a back edge is found (i.e., $\langle 1\rangle,\langle 2\rangle,\langle 3\rangle,\langle 4\rangle,\langle 5\rangle,\langle 3\rangle$). The state at the endpoint can be specified by the constraint sequence $y\geq 0\wedge n=0\wedge n<N\wedge y'=y+1\wedge n'=n+1$ on the variables x' and y'. The constraint sequence is obtained from the statements along the path. In the next step, AL will attempt to infer which constraints are individually invariant in order to get child-parent subsumption (i.e., "close" the loop). It is straightforward to see that $y\geq 0$ is invariant through the loop because when the program point 3 is first visited, $y\geq 0$ is satisfied and the aforementioed constraint sequence obtained after one iteration of the loop constrains y' to $y'\geq 0$.

The second essential step is when the exit condition is taken (i.e., $\langle 1\rangle,\langle 3\rangle,\langle 6\rangle$). AL will attach $n\geq N$ to all individually invariant constraints (in this case $y\geq 0$). More importantly, those two constraints $(y\geq 0\wedge n\geq N)$ suffice to prove that the error condition $y+n<N$ is false. Therefore, AL can prove the safety of the program with only one iteration through the loop.

A performance comparison for this program is shown in Fig. 8(b). The program was proved safe with different values of N. The x-axis represents the values of N, and the y-axis the time in seconds. Both AR tools exhibit a prohibitive cost due to the full loop unwinding.

3.5 Unnecessary Detection of Infeasible Paths

So far we have illustrated the cases where AL performs better than AR. The advantage of AR being exploited in the preceding examples is the preservation infeasible paths while abstracting, modulo the abstraction by lightweight loop invariants.

Unfortunately, this characteristic might be an important downside for AL if the program can be proved safe even traversing infeasible paths since all the work of generating abstraction while maintaining infeasible paths would be wasteful.

Consider the program in Fig. 9(a). Assume an arbitrary number of branches which can be false with some probability. After that, there is an assignment `c=1` that trivially makes false the error condition $(c=0)$. In the case of AL, all infeasible paths will be detected and, more importantly, the interpolants that ensure them will be computed. However, in counterexample-guided refinement tools the proof of the absence of bugs is found after the discovery of a single predicate $(c=1)$. The performance of AL and AR proving safe the program is shown in Fig. 9(b) where the number of branches is on the horizontal axis and the time in seconds on the vertical axis.

We claim that eager detection of infeasible paths even if they are not relevant to the safety property at hand is not limiting in practice. Intuitively, many of the infeasible paths must be considered in real programs to block the error paths. Therefore, we believe that the amount of extra work is often insignificant. The evaluation of our approach with real programs in Sec. 6 supports this view.

To ellaborate let us consider a real program `statemate` [26] (1276 LOC) used commonly for testing WCET tools. The program is automatically generated code by the STAtechart Real-time-Code generator STARC. Its main feature is the significant amount of

infeasible paths produced by the generator. We did the following experiments by running it on three different properties, all of them safe:

- We instrument the program as follows: we add in the last statement of the program `x=0` where `x` is a fresh variable, and then add the condition `if (x>0) error()`. That is, the instrumented program poses a similar behavior to the one in Fig 9(a). For a BLAST-like tool, it suffices to add the predicate $(x = 0)$ to prove that the program is bug free. On the contrary, AL needs to traverse the whole program with the corresponding cost due to the encountered infeasible paths. The analysis time for BLAST was negligible (less than 1 second) while our tool took 88 seconds.[3]
- We now modify slightly the program: we now add the statement `x=0` at the beginning of the `statemate` program and add as before the condition `if (x>0) error()` after the last statement. Here, BLAST performance deteriorates since it discovers now 21 predicates. The reason of this degradation is that if the abstract counterexample has more than one causes for infeasibility to select the best cause to be considered in refinements is not straightforward. In fact, this challenge was already observed in [19]: " *If an abstract error trace has more than one infeasibility, then existing refinement techniques used by Slam-like tools have difficulties in choosing the "right" infeasibility to refine.* "

  Surprisingly, BLAST takes 74 seconds and AL again 88 seconds.
- Finally, we instrument the same program to prove that the number of statements does not exceed a certain bound. With this more realistic safety property, AL overperforms BLAST significantly since it can prove safety in less than 15 minutes while BLAST's run exceeds 1 hour.

Concluding remarks on Academic Examples

We have highlighted several technical aspects that are central in AR: in particular, (a) traversing thru infeasible paths, (b) using predicates in future traversal, (c) traversal using abstract states, (d) discovering loop invariants, and finally, (e) reasoning about infeasible paths when this is unnecessary. In our examples above to highlight the shortcomings of AR, we note that ARMC did not share the same behavior as BLAST in (b) and (c). The reason for this is that, in general, the issue about what predicates to discover and then, how to use these discovered predicates, is not standardized. We call this the *propagation* method. Indeed, BLAST has several "options" implementing different strategies for propagation. In the ARMC statistics for (b) and (c), it happens that the examples used were effective against BLAST (using some particular of its options) and not against ARMC.

In order not to suggest a relative performance between ARMC and BLAST, consider a further example.

```
main{
1:   x=0;s=0;
2:   if (*) s++; else s=s+2;
     ...
3:   if (*) s++; else s=s+2;
4:   if (x!=0) error()
}
```

This program is clearly safe and can be proved by BLAST by adding a single predicate $(x = 0)$. Our algorithm will also prove the absence of bugs after a path is traversed since the code in Lines 2-...-4 will be further subsumed. Surprisingly, ARMC will add predicates for the irrelevant code that updates `s`. Therefore, the analysis will be prohibitively expensive. The reason we suspect is that ARMC decides to propagate the state of the variable `s`, and hence it will add the predicates $(s = 0)$, $(s = 1)$, $(s = 2)$, ... More importantly, after a spurious counterexample is found it does not have the "systematic mechanism to release" those predicates.

[3] on Intel 2.33Ghz 3.2 GB.

```
main() {
   x=0;
0: if (*) x++;
1: if (y>=1) x+=2;
2: if (y<1)  x+=4;
3: if (x>5)
4:    error();
}
```

(a)

$$
\begin{array}{l}
(0,x,y) \text{ :- } (1,x,y).\\
(0,x,y) \text{ :- } (1,x',y), x' = x+1.\\
(1,x,y) \text{ :- } (2,x,y), y < 1.\\
(1,x,y) \text{ :- } (2,x',y), x' = x+2, y \geq 1.\\
(2,x,y) \text{ :- } (3,x,y), y \geq 1.\\
(2,x,y) \text{ :- } (3,x',y), x' = x+4, y < 1.\\
(3,x,y) \text{ :- } (4,x,y), x > 5.
\end{array}
$$

(b)

Figure 10. A Program and Its CLP Model

In summary, while AR systems have the flexibility to adjust their propagation strategy, they can *in principle* mimic our dual strategy AL which uses *exact* propagation, up to loops. However, the critical difference is that AL has the ability on the one hand to use predicates in traversal, and on the other hand, *disregard* predicates when they are not needed. This on-demand use of predicates is the reason why we have called our method "learning".

4. Formalities

Here we briefly formalize a program as a transition system and the proof process as one of producing a closed tree of the transition steps. It is convenient to use the formal framework of Constraint Logic Programming (CLP) [22], which we outline as follows.

The *universe of discourse* is a set of terms, integers, and arrays of integers. A *constraint* is written using a language of functions and relations. In this paper, we will not define the constraint language explicitly, but invent them on demand in accordance with our examples. Thus the terms of our CLP programs include the function symbols of the constraint language.

An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and the $\tilde{t}$ a tuple of terms. A *rule* is of the form A :- $\tilde{B}, \phi$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint ϕ constitute the *body* of the rule. A *goal* G has exactly the same format as the body of a rule. A *ground instance* of a constraint, atom and rule is defined in the obvious way.

A *substitution* simultaneously replaces each variable in a term or constraint into some expression. We specify a substitution by the notation $[\tilde{E}/\tilde{X}]$, where $\tilde{X}$ is a sequence $X_1, \ldots, X_n$ of variables and $\tilde{E}$ a list $E_1, \ldots, E_n$ of expressions, such that X_i is replaced by E_i for all $1 \leq i \leq n$. Given a substitution θ, we write as $E\theta$ the application of the substitution to an expression E. A *renaming* is a substitution which maps variables into variables. A *grounding* is a substitution which maps each variable into a value in its domain.

Given a goal $\mathcal{G} \equiv p_k(\tilde{X}), \Psi(\tilde{X})$, $[\![\mathcal{G}]\!]$ is the set of the groundings θ of the primary variables $\tilde{X}$ such that $\exists \Psi(\tilde{X})\theta$ holds. We say that a goal $\overline{\mathcal{G}} \equiv p_k(\tilde{X}), \overline{\Psi}(\tilde{X})$ *subsumes* another goal $\mathcal{G} \equiv p_{k'}(\tilde{X}'), \Psi(\tilde{X}')$ if $k = k'$ and $[\![\overline{\mathcal{G}}]\!] \supseteq [\![\mathcal{G}]\!]$. Equivalently, we say that $\overline{\mathcal{G}}$ is a *generalization* of $\mathcal{G}$. We write $\mathcal{G}_1 \equiv \mathcal{G}_2$ if $\mathcal{G}_1$ and $\mathcal{G}_2$ are generalizations of each other. We say that a sequence is *subsumed* if its last goal is subsumed by another goal in the sequence.

Let $\mathcal{G} \equiv (B_1, \cdots, B_n, \phi)$ and P denote a goal and program respectively. Let $R \equiv A$:- $C_1, \cdots, C_m, \phi_1$ denote a rule in P, written so that none of its variables appear in $\mathcal{G}$. Let $A = B$, where A and B are atoms, be shorthand for equations between their corresponding arguments. A *reduct* of $\mathcal{G}$ using R is of the form

$(B_1, \cdots, B_{i-1}, C_1, \cdots, C_m, B_{i+1}, \cdots, B_n, B_i = A \wedge \phi \wedge \phi_1)$

provided $B_i = A \wedge \phi \wedge \phi_1$ is satisfiable.

A *derivation sequence* is a possibly infinite sequence of goals $\mathcal{G}_0, \mathcal{G}_1, \cdots$ where $\mathcal{G}_i, i > 0$ is a reduct of $\mathcal{G}_{i-1}$. Given a sequence

τ defined to be $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n$, then *cons*($\tau$) is all the constraints of the goal $\mathcal{G}_n$. We say that a sequence is *feasible* if *cons*(τ) is satisfiable, and *infeasible* otherwise.

A *derivation tree* for a goal has as branches all derivation sequences emanating from this goal. In this tree, the *ancestor-descendant* relation between nodes is defined in the usual way. A derivation tree is *closed* if all its leaf goals are either successful, infeasible, or subsumed by some other goal in the tree.

In this paper, a program is compiled into a CLP program using just one predicate symbol, . We thus omit writing a predicate symbol in CLP rules and goals. Symbolic states are goals of the form $(k, \tilde{x}), \tilde{c}$ where k is a program point, $\tilde{x}$ is a list of variables representing the variables of the underlying program, and $\tilde{c}$ is a sequence of constraints. Henceforth goals and states are synonymous. We say that $\tilde{x}$ are the *primary* variables of the goal. Where $\mathcal{G}$ is a goal, we write $\mathcal{G}(\tilde{x})$ to indicate that $\tilde{x}$ are the primary variables of $\mathcal{G}$. The details of how to generally perform a compilation of an underlying program into a CLP program is straightforward, and so omitted. Instead, we refer to an example in Figure 10.

We can now informally present, in Figure 11, a naive proof method for programs which have been encoded as a CLP program. The purpose of the description here is to set the stage for the description of our advanced algorithm in the next section. The algorithm starts with an initial goal representing the initial symbolic state and produces a derivation tree. The problem at hand is to prove that the constraints of certain goals meet a given safety obligation. As the examples suggest, we assume this is implemented by an *error* condition and so the problem reduces to proving that the error state is not reachable. If the algorithm terminates normally (without aborting), then the underlying program is safe. If not, the current path in the tree provides a *counterexample* to safety.

We assume, without losing generality, that each goal or state has zero or exactly two descendants.

```
Naive(G)
  switch(G)
    case G is an error state: ABORT
    case G is terminal: return
    case G is infeasible: return
    case G subsumed by a state already traversed
    default :
        // G has descendants G1 and G2; w.l.o.g. assume G1 feasible
        Naive(G1)
        Naive(G2)
        return
```

Figure 11. A Naive Algorithm

As an example, given the program in Fig. 1 (b), the derivation tree is shown in Fig. 12.

The naive algorithm is *sound* in the sense that if it terminates successfully, then the safety properties are assured. However, there are two main shortcomings: it does not terminate in general, even if it did, it does not scale. We address these shortcomings in the next section.

5. Algorithm: Minimax

The algorithm below maintains knowledge about a state $\mathcal{G} \equiv ((k, \tilde{x}), c_1, \ldots, c_n)$ by means of a vector $\langle \alpha_1, ..., \alpha_n \rangle$ where each α_i is an *annotation* of one of the following kinds:

- a *max* annotation, indicating that the constraint c_i *must be kept*
- a *min* annotation, indicating that the constraint c_i *must be deleted*, or
- a *neutral* annotation.

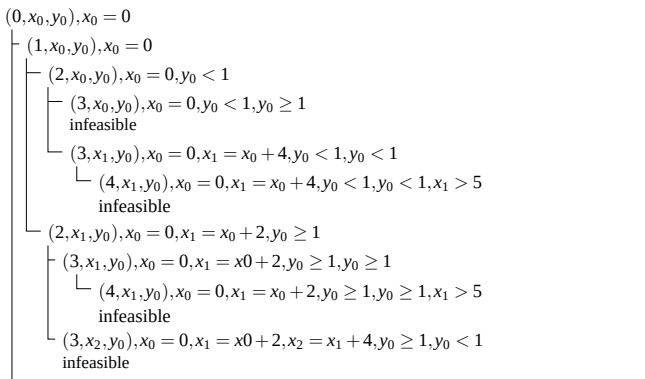


Figure 12. An Example Derivation Tree

A *max* annotation essentially provides a restriction that any attempt at abstracting this state must not "exceed" the constraint in question. That is, any abstraction employed must result in this constraint remaining entailed. Dually, a *min* annotation defines an abstraction on the state in question. Why and when states need to be abstracted is explained below.

A vector v associated with a state $\mathcal{G} \equiv (k, \tilde{x}, \tilde{c})$ is jointly called an *annotated state*. The meaning of an annotated state $(\mathcal{G}, v)$ is obtained in one of two ways:

- a *max* interpretation $max(v, \mathcal{G})$ is the state obtained by deleting all but the *max* annotated constraints therein.
- dually, a *min* interpretation $min(v, \mathcal{G})$ is the state obtained by including all but the *min* annotated constraints therein.

and these are used in two main ways. The former is used in the process of memoization where it is desired to memo the most general state. The latter is used in symbolic traversal. We elaborate below. Meanwhile, suppose we have the annotated state $\mathcal{G}$:

$$(k, x_1, x_2, x_3), x_1 = 1, x_2 = 2, x_3 = 3, \langle min, neutral, max \rangle$$

Then $min(\mathcal{G})$ and $max(\mathcal{G})$ are, respectively, the following two states:

$$(k, x_1, x_2, x_3), x_2 = 2, x_3 = 3$$
$$(k, x_1, x_2, x_3), x_3 = 3$$

The algorithm operates on annotated states, and is presented in Figure 13. We assume, without losing generality, that each state has up to two descendants. We also assume, for notational convenience, that the descendant of a state has exactly one more constraint than the state itself. Finally, where v is a vector for a state, we write *v.neutral* to denote the elongation of v by one neutral annotation.

The algorithm begins with an initial state whose vector has no *min* or *max* annotations, only neutral annotations. As it progresses, *max* annotations are created to indicate constraints that are needed in order to preserve *false* paths (see the case for infeasible state).

In the more general case where state $\mathcal{G}$ has descendants, we consider the first subcase where $\mathcal{G}$ has no looping parent. Here we process its first descendant $\mathcal{G}_1$ which cannot be an infeasible state. (W.l.o.g, we assume every nonterminal feasible state has at least one feasible descendant.) Suppose v was the input vector for $\mathcal{G}$, and that we just obtained a vector v_1 from $\mathcal{G}_1$. Now we process the second descendant but now, instead of using v, we use v_1. This means that we are actually running what is an abstraction of the state $\mathcal{G}_2$. If the processing of $(v_1, \mathcal{G}_2)$ then concludes successfully or with a real counterexample, we are done. Otherwise, it must be the case

that a "conflict" has occurred, that is, some *min* annotations in v_1 conflicts with some *max* annotations required by the processing of $\mathcal{G}_2$. It is here when the system is restarted *at the looping parent which gave rise to the offending min annotation(s)*.

The crucial step to note that the restart will now take place not with the original parent, say $(v_p, \mathcal{G}_p)$, but with a new vector $(v, \mathcal{G}_p)$ where v is such that the offending *min* annotation(s) in v_p are now replaced by *max* annotations (via the maximize() function in Figure 13). The effect of this is that certain constraints are now *locked* and prevented from being *min* annotated again. This is the mechanism for unrolling a loop when a previous attempt at invariant discovery fails.

We now describe the alternative subcase where $\mathcal{G}$ has a looping parent $\mathcal{G}_p$. As mentioned above, straightforward symbolic execution does not terminate and so some speculation in the form of loop invariant *discovering* is required. The *min* annotations represent our mechanism for this. More precisely, at a looping state $\mathcal{G}$, one which has parent state $\mathcal{G}_p$ with the same program point, we employ an algorithm to determine which of the constraints in $\mathcal{G}_p$ are *individually invariant* through this path from $\mathcal{G}_p$ to $\mathcal{G}$. The idea is that the *other* constraints are now *min* annotated. More details are provided in the description of the function invariant() below. Although it seems that a loop invariant can always be found in this manner, eg. we could annotate *all* the constraints in $\mathcal{G}_p$ obtaining a correct (though generally useless) loop invariant, in general some constraints in $\mathcal{G}_p$ may be already *max*-annotated. It is at this point that loop invariant discovery *fails*, and so the attempt to "close" the loop here is abandoned.

Finally we explain the restart strategy of our algorithm. Note that each *min* annotation is associated uniquely with a state $\mathcal{G}_p$ representing a particular (parent) looping point. That is, the min constraint arose because it was not independently invariant. The general idea is simply that when a *min* annotation is designated "conflict", we perform a restart at the associated state $\mathcal{G}_p$. All system data structures are essentially reset to the point where $\mathcal{G}_p$ was previously executed, with one crucial difference. This time, $\mathcal{G}_p$ will be executed with more *max* annotations because some conflicting *min* annotations are rewritten to *max*. In short, we restart but this time some constraints are "locked".

We now fill in some details for the algorithm in Figure 13.

- interpolate($\tilde{c}, v$)
  This returns a "minimal" vector v' instantiating v[4] such that $max(v', \tilde{c})$ is infeasible. Essentially this is computed by adding the fewest *max* annotations to v, thus representing a computation of an "interpolant". In our experiments described below, we employ a greedy algorithm which deletes constraints from $\tilde{c}$, proceeding from the first constraint in $\tilde{c}$, as long as the remaining constraints remain infeasible.

  For example, consider the following annotated state $\mathcal{G}$:

  $$(k, x_1, x_2), x_1 > 3, x_1 = y_1 + 1, y_1 = 2, x_2 = 0,$$
  $$\langle max, max, neutral, neutral \rangle$$

  Then interpolation of $\mathcal{G}$ wrt the condition $x_1 < 0$ would produce the annotation vector:

  $\langle max, max, max, neutral \rangle$

  That is, the third constraint $y_1 = 2$ is now annotated *max* from neutral, while the other annotations are unchanged. Note that the annotation of the last constraint remains neutral because this constraint is not needed to demonstrate infeasibility.

- subsumed($\mathcal{G}, \mathcal{G}_1$)
  Simply, this is *true* if the state $\mathcal{G}$ is subsumed by $\mathcal{G}_1$.

- invariant($\tilde{c}(\tilde{x}), v, \tilde{c_p}(\tilde{x_p})$))
  Note that $\tilde{c}$ is a prefix of $\tilde{c_p}$, and that we have indicated the primary variables of $\tilde{c}$ and $\tilde{c}_p$ here. This function returns a vector v_1 which is identical to v except that some neutral annotations have been changed to *min* annotations such that $min(\tilde{c}(\tilde{x}), v)$ entails $min(\tilde{c_p}(\tilde{x_p}), v)$. More precisely, a constraint in $\tilde{c}$ is said to be *individually invariant* if it holds through the path in question without consideration of any other conditions. All other constraints in $\tilde{c}$ will then be annotated *min*. Essentially, this is the loop invariant discovery phase where the new state is abstracted in such way that it is now subsumed by its parent state.

  For example, consider the program snippet:

  ```
  0: x = 0;
  1: if (y > 0) 2: while (*) 3:  x++; y++;
  ```

  and so the state $\mathcal{G}_p \equiv (2, x, y), x = 0, y > 0$ represents an entry to the loop. We then obtain from this state a derivation sequence leading to $\mathcal{G} \equiv (2, x_1, y_1), x = 0, y > 0, x_1 = x + 1, y_1 = y + 1$ corresponding to the end of the loop body. The idea now is that the constraint x = 0 must be deleted in order for the latter state to be subsumed by the former, that is:

  $(2, x_1, y_1), y > 0, y_1 = y + 1$ is subsumed by $(2, x, y), y > 0$.

  Thus, for example, invariant($x = 0, y > 0, \langle neutral, neutral \rangle, x = 0, y > 0, x_1 = x + 1, y_1 = y + 1$) is $\langle min, neutral \rangle$ where the *min* annotation has now replaced the first neutral annotation.

  It is important to note that such an invariant may not always be found (because of the requirement that only neutral annotated constraints can be abstracted). In such a case, the function returns a null value as failure.

- State associated with a conflict v:
  Recall that a conflict vector is such that at least of one of its *max* annotations had previously been a *min* annotation. Further recall that each *min* annotation had been brought into existence because of one of two reasons:
  (a) the invariant() function, which serves to delete constraints in order to obtain a loop invariant. Here the "associated state" refers to the looping parent state in question.
  (b) in the case for subsumption, it is indicated which constraints must be deleted in order for subsumption to hold. Here "associated state" refers to the closest state which is a common parent of both the subsumed and subsuming state.

- push_system($\mathcal{G}$), pop_system($\mathcal{G}$)
  The push saves the state of the system and timestamps it with the identifier $\mathcal{G}$. The pop restores to system state to the point where the corresponding push took place.

- merge(v_1, v_2)
  This function simply merges the *min* and *max* annotations of the input vectors v_1 and v_2 into a single vector, possibly replacing neutral annotations in one vector if the corresponding annotation in the other vector is is a *min* and *max* annotations. This function is well-defined because v_1 and v_2 do not conflict, that is, it is not the case that one vector has a *min* annotation while the corresponding annotation in the other is a *max* annotation.

- memo($\mathcal{G}$, v)
  This simply records in persistent memory the fact that the annotated state $(\mathcal{G}, v)$ has already been processed. It survives forever unless retracted by a pop_system($\mathcal{G}_p$) operation of some ancestor $\mathcal{G}_p$.

We now demonstrate a run of the algorithm on the example of Fig. 1 (c), whose tree traversals are depicted in Fig. 4. Initially, the memo table is empty, and we start the algorithm by calling Minimax with the annotated state $(0, lock_0, new_0, old_0, flag_0), \langle\rangle$ denoting the initial state of the program where all variables are unconstrained.

[4] v' has all the *min*/*max* annotations of v.

```
Minimax(G ≡ ((k, x̃), c̃), v) returns OK(v) or CONFLICT(v)
  switch(G)
    case G is an error state:
        let c̃ denote the constraints in min(G, v)
        if (c̃ is feasible) ABORT
        return CONFLICT(interpolate(c̃1, v))
    case G is terminal: return OK(v)
    case subsumed(min(G, v), max(G1, v1)) for some memo'd (G1, v1):
        let c̃2 denote the common prefix of constraints in G and G1
        v2 = v except some neutral annotations are replaced by min
        In particular, v2 has the fewest such replacements so that
            min(G, v) is subsumed by min(G1, v2).
        return OK(v2)
    case min(G, v) is infeasible:
        return OK(interpolate(c̃, v))
    case G has a looping parent Gp:
        for (each looping parent) Gp of G
            let c̃p denote the constraints of Gp
            if (v = invariant(c̃(x̃), v, c̃p(x̃p)) return OK(v)
            continue // (to the case below)
    default :
        // G has at most two descendants G1 and G2; G1 is feasible
        if (G is a looping point) push_system(G)
        STATUS(v1) = Minimax(G1, v.neutral)
        if (STATUS == CONFLICT) {
            if (G is associated with conflict v1) {
                pop_system(G);
                return Minimax(G, v1)
            } else return CONFLICT(v1)
        }
        if (G doesn't have a second descendant) return OK(v1)
        let G2 denote the second descendant
        // speculative abstraction
        STATUS(v2) = Minimax(G2, v1.neutral)
        if (STATUS == CONFLICT) {
            if (G is associated with conflict v2) {
                pop_system(G);
                return Minimax(G, v2)
            } else return CONFLICT(v2)
        }
        v' = merge(v1, v2)
        memo(G, v');
        return OK(v');
```

Figure 13. The Minimax Algorithm

With these arguments, only the default case can be taken, where the algorithm generates a reduct $\mathcal{G}_1$ of the current state $\mathcal{G}$ and calls Minimax recursively[5]. The reduct $\mathcal{G}_1$ is

$$(1, lock_1, new_1, old_0, flag_1), \quad lock_1 = 0, new_1 = old_0 + 1, flag_1 = 1. \tag{1}$$

The annotation is the vector $\langle neutral, neutral, neutral\rangle$[6]. Each element of the vector corresponds to the constraint added by the reduct. In this recursive call also, only the default case can be selected, where we execute push_system($\mathcal{G}$) (for $\mathcal{G}$ (1)) since the state corresponds to looping point and we recursively call Minimax with the annotated state $(1, lock_1, new_1, old_0, flag_1)$, $lock_1 = 0$, $new_1 = old_0 + 1$, $flag_1 = 1$, $new_1 \neq old_0$, $\langle neutral, neutral, neutral, neutral\rangle$. Notice that here we added a constraint $new_1 \neq old_0$ to the constraint sequence, and correspondingly lengthen the vector with a *neutral*.

We perform generation of reducts recursively, such that the derivation sequence corresponds to the path in Fig. 4 (a). In the recursive call that corresponds to $\langle 1'\rangle$, the state is:

$$(1, lock_2, new_1, old_1, flag_1), lock_1 = 0, new_1 = old_0 + 1, \quad flag_1 = 1, new_1 \neq old_0, lock_2 = 1, old_1 = new_1 \tag{2}$$

and the annotation is a vector of neutral values corresponding to the sequence of constraints. (2) is subsumed by its ancestor (1) if we remove the constraints $lock_1 = 0$ and $new_1 = old_0 + 1$ leaving only $flag_1 = 1$. Projecting this constraint on the primary variables of the ancestor and writing it using the proper variable names we get the constraint $flag = 1$. Removing the same constraints from (2) results in the projection $flag = 1$, $lock = 1$, and $old = new$. Obviously, this conjunction entails $flag = 1$. The algorithm marks $lock_1 = 0$ and $new_1 = old_0 + 1$ for deletion (*min*) in the vector. The vector at $\langle 1'\rangle$ becomes $\langle min, min, neutral, neutral, neutral, neutral\rangle$. The first two *min*s correspond to the first two constraints in the constraint sequence that must be deleted.

Subsequent traversal propagates the deletion (min) information. In the path shown in Fig. 4 (b), for instance, the state at $\langle 1''\rangle$ is

$$(1, lock_3, new_2, old_1, flag_1), lock_1 = 0, new_1 = old_0 + 1, \quad flag_1 = 1, new_1 \neq old_0, lock_2 = 1, \quad old_1 = new_1, lock_3 = 0, new_2 = new_1 + 1 \tag{3}$$

and the vector is $\langle min, min, neutral, neutral, neutral, neutral, neutral, neutral\rangle$. The first two markings of the vector denotes the deletion of $lock_1 = 0$ and $new_1 = old_0 + 1$, inherited from the traversal of Fig. 4 (a). Subsumption of (3) by the ancestor (1), which is handled by the second case of Minimax, holds without further need for deleting constraints. In this case, the same vector is returned without modification.

Notice that $\langle 3\rangle$ is visited both in Fig. 4 (a) and Fig. 4 (b) such that Minimax, combines the vectors returned by the recursive calls using the merge procedure. In this case, both path $\langle 3\rangle, \langle 1\rangle$ and $\langle 3\rangle, \langle 4\rangle, \langle 1''\rangle$ return the same vector $\langle min, min, neutral, neutral, neutral, neutral\rangle$, which is then included in the return value of Minimax.

Again, by the previous deletion of constraints, the program state at $\langle 1\rangle$ is described by $flag = 1$. This allows the algorithm to make the reduction step from $\langle 1\rangle$ to $\langle 5\rangle$ in Fig. 4 (c). The reduction from $\langle 5\rangle$ to $\langle 6\rangle$ is infeasible, where in its third case Minimax handles the state

$$(6, lock_1, new_1, old_0, flag_1), lock_1 = 0, new_1 = old_0 + 1, \quad flag_1 = 1, new_1 = old_0, flag_1 = 0. \tag{4}$$

with annotation $\langle min, min, neutral, neutral, neutral\rangle$. The first two *min* markings are due to subsumption of (2) by (1) explained previously. Here the algorithm computes an interpolant by calling the interpolate function with the constraint sequence as the first argument and the vector as the second to compute a minimal set of constraints that must be kept in order to preserve the unsatisfiability. The result of this computation is the updating of the vector with *max* annotations. Since both $flag_1 = 1$ and $flag_1 = 0$ must be kept to maintain the unsatisfiability of the constraints, we mark the corresponding positions in the vector with *max*, resulting in the vector $\langle min, min, max, neutral, max\rangle$.

The algorithm visits the error point $\langle 8\rangle$ in Fig. 4 with the constraint sequence $lock_1 = 0$, $new_1 = old_0 + 1$, $flag_1 = 1$, $new_1 = old_0$, $flag_1 \neq 0$, $lock_1 = 0$ of the state and vector $\langle min, min, neutral, neutral, neutral, neutral\rangle$. Here the algorithm discovers the unsatisfiability of the sequence, and therefore, it has found a conflict due to over approximation. It also discovers that the point of conflict is at $\langle 1\rangle$, where the removal of the constraints took place. Here the al-

[5] As in this case, there is only one outgoing transition from the initial program point $\langle 0\rangle$. The transition is feasible, and we assume that the generation of second reduct $\mathcal{G}_2$ and its recursive call in Fig. 13 is not executed.

[6] In Fig. 13 we only add one *neutral*, however, in general, this depends on the number of constraints added in the reduction.

Program	LOC	BLAST (AR)		TRACER (AL)	
		P	T	S	T
qpmouse	400	4	0.42	974	0.42
tlan	8069	14	17.10	4382	5.78
cdaudio	8921	*	*	6258	10.53
diskperf	6984	92	82.3	3326	8.21
floppy	8570	*	*	3124	6.47
kbfiltr-safe	5931	45	44.03	1392	2
kbfiltr-unsafe-1		62	108.59	463	0.56
kbfiltr-unsafe-2		53	71.92	283	0.32
serial	10380	*	*	51935	328.6
tcas-1a-safe	394	23	3.6	6029	6.97
tcas-1b-safe		56	78.35	6050	6.77
tcas-2a-safe		22	3.25	6029	6.74
tcas-3b-safe		39	15.68	6017	6.63
tcas-5a-safe		31	10.29	6029	6.36
tcas-2b-unsafe		40	17.46	91	0.01
tcas-3a-unsafe		25	18.96	243	0.16
tcas-4a-unsafe		45	14.44	243	0.15
tcas-4b-unsafe		36	6.44	91	0.01
tcas-5b-unsafe		54	40.31	91	0.02

Table 1. BLAST Benchmarks

gorithm mark the constraint such that it can no longer be removed, that is, "locking" it, and restarts the traversal from the conflict point, that is, $\langle 1 \rangle$ after calling pop_system($\mathcal{G}$) (with $\mathcal{G}$ the state of (1)). The second traversal is shown in Fig. 4 (d). It uses the procedures that have been exemplified here and therefore we do not elaborate further.

A soundness proof for the algorithm is nontrivial only in the following aspect: when a state $\mathcal{G}$ is declared subsumed by an entry $\mathcal{G}_1$ in the memo table, it is possible that the memo entry is for a state within a loop that is *not yet closed*. Closure here means that the state at hand has been fully traversed and analyzed. That is, even though $\mathcal{G}_1$ appears in the memo table, its parent $\mathcal{G}_p$ is a looping state which is not yet memoed. This means that $\mathcal{G}_1$ itself is not yet closed. More specifically, the *max* annotations for $\mathcal{G}_p$ are not yet finalized, and because $\mathcal{G}_1$ is in the loop body of $\mathcal{G}_p$, its own max annotations are not finalized. A problem can thus arise when we use the current memo information about $\mathcal{G}_1$, which depends only on its *max* annotations, in order to subsume other states (such as $\mathcal{G}$). It is for this reason that in the process of subsumption, the subsumed state generates a *min* annotation on constraints that it *depends on to be deleted* in order to be subsumed. Overall, the formal proof of soundness, while nontrivial, is not deep and hence omitted.

We conclude this section by mentioning that the central step of deleting constraints, the effect of a *min* annotation, can in fact be relaxed to some other mechanism that abstracts the state at hand. Instead of deleting a constraint, one could *transform* a constraint. For example, one could apply a process of "slackening" to equations $x = y$ to obtain an inequality, either $x \leq y$ or $x \geq y$. This kind of abstraction is in fact employed in the BLAST system which we benchmark against, but at this time, we do not use for our own experimental results. Even more generally, we could replace not one but a collection of constraints by another collection which is entailed by the original collection.

6. Experimental Evaluation

We ran our prototype, TRACER, on several programs and compare with BLAST [10][7], a state-of-the-art verification tool based on abstraction refinement. Since we wanted a faithful comparison with BLAST we downloaded all programs from [9] already instrumented with safety conditions, and together with a script which runs those programs with the most favorable system options.

The first two programs are Linux device drivers: qpmouse and tlan. The device drivers can acquire or release a spinlock to write or read data. The safety condition checks that the drivers do not perform two consecutive calls in order to either acquire the spinlock or release it.

The next five programs are Microsoft Windows device drivers: cdaudio, diskperf, floppy, kbfiltr, and serial. These programs are run on an IO request packet (IRP) completion specification. The specification provides correct ways for the device drivers to handle IRPs by specifying a sequence of functions to be called in a certain order, and specific return codes.

Finally, tcas is an implementation of a traffic collision avoidance system, a real-life safety-critical embedded system. The program is instrumented with five safe conditions and five unsafe. Those properties are about anti-collision conditions: safe advisory selection, best advisory selection, avoid unnecessary crossing, no crossing advisory selection, and optimal advisory selection.

The results, obtained on an Intel 2.33Ghz 3.2 GB, are summarized in Table 1. We present two set of numbers: for BLAST the number of discovered predicates (P) the total time in seconds (T), and for TRACER, our prototype tool, the number of nodes of the exploration tree (S) and also the total time in seconds (T). Although the number of discovered predicates and nodes of the exploration tree are not comparable they are shown to provide an idea about the hardness of the proof.

In summary, TRACER is competitive with BLAST in most of the benchmark examples, sometimes much faster. However, there are two programs where BLAST is faster (tcas-1a-safe, and tcas-2a-safe). We believe the main reason is that TRACER does perform some extra work due to unnecessary infeasible paths. Nevertheless, the numbers show that the differences are not significant.

Note that programs such as cdaudio, floppy, and serial are annotated with the symbol '*' in the BLAST column which means that BLAST raised an exception and aborted. Therefore, we were not able to verify those programs using BLAST. However, we are aware that cdaudio and floppy have been proved safe in [20] after 21m59s and 11m17s discovering 196 and 156 predicates, respectively on an IBM ThinkPad T30 laptop with 2.4Ghz Pentium processor and 512MB RAM. We were surprised by the numbers obtained for the cases where the programs were proved unsafe. In these cases, TRACER found a real counterexample much faster than BLAST. We believe that the reason can lie on the difficult in choosing the "right" infeasibility to refine since the tcas program poses an important amount of infeasible paths.

7. Concluding Remarks

We presented Abstraction Learning (AL), a dual approach to symbolic traversal to CEGAR (AR). The main algorithm is a process of classifying constraints into *min* and *max* constraints. The *min* constraints are those which must be abstracted in order to achieve subsumption and loop invariance, while the *max* constraints are those those which must not be abstracted so as to detect infeasible paths and also to preserve safety. The idea is to have as few of these two kinds of constraints as possible. We then discussed the relative merits of AL and AR using academic examples. While AL escapes several shortcomings of AR, it does have the shortcoming of detecting infeasible paths when this may not be necessary.

We then evaluated our prototype implementation TRACER against BLAST, using a significant benchmark collection obtained from the BLAST literature. The results showed competitive performance, with some examples showing great improvement. In all

[7] We also tried with ARMC available at [33] but we were only successful to run on tcas and statemate but timeout expired in both cases after 30m and 1h, respectively.

cases, the results show that the potential shortcoming of systematically detecting infeasible paths was in fact affordable.

In order to strengthen this point, first note that verification was not the only motivation for this work on AL; also, it was *analysis*. That is, we wished to *discover* properties, and not just to verify them[8]. As a quick example of analysis, consider running AL on an underlying program but *without a safety property*[9] The minimax algorithm would still produce a useful closed proof tree. For example, this tree could serve as a (now path-sensitive) control flow graph for use by standard analysis algorithms. such as those for determining an upper bound for a variable, or for determining certain relationsips between variables. Such a graph could also be used by path-sensitive algorithms such as BLAST and TRACER, but this time the motivation is *speed*. For example, constructing such a graph for the `tcas` program, and running the graph (as though it were just another program) in TRACER using the provided safety properties, we saw a three-fold increase in speed. Perhaps most importantly, this particular proof tree can be constructed *offline*.

We have in fact run many other programs with no safety property, and have found the size of the tree (which will always be not greater in size than when a nontrivial safety property is used) to be very manageable. This then, is added evidence that the apparent shortcoming of systematically detecting infeasible paths is, in fact, affordable in practice.

References

[1] A. Armando, M. Benerecetti, and J. Mantovani. Abstraction refinement of linear programs with arrays. In *TACAS'07*, pages 373–388.

[2] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *15th PLDI*, pages 203–213. ACM Press, May 2001. SIGPLAN Notices 36(5).

[3] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *IFM'2004*, 2004.

[4] C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *16th CAV*.

[5] C. Barrett, D. L. Dill, and J. R. Levitt. Validity checking for combinations of theories with equality. In *FMCAD'96*, pages 187–201.

[6] R. J. Bayardo, Jr. and R. Schrag. Using csp look-back techniques to solve real-world sat instances. In *14th AAAI/9th IAAI*, pages 203–208. AAAI Press, 1997.

[7] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA '08*, pages 3–14.

[8] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *PLDI'07*, pages 300–309.

[9] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. BLAST. URL http://mtc.epfl.ch/software-tools/blast/index-epfl.php.

[10] D. Beyer, T.A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST. *Int. J. STTT*, 9:505–525, 2007.

[11] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In R. Cleaveland, editor, *5th TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[12] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, June 2004.

[13] A. Cimatti, A. Griggio, and R. Sebastiani. Efficient interpolant generation in satisfiability modulo theo ries. In *TACAS'08*, pages 397–412.

[14] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. CounterrExample-Guided Abstraction Refinement. 2000.

[15] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis. In *4th POPL*, pages 238–252. ACM Press, 1977.

[16] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.

[17] D. Frost and R. Dechter. Dead-end driven learning. In *12th AAAI*, pages 294–300. AAAI Press, 1994.

[18] P. Godefroid, A. V. Nori, S. K. Rajamani, and Sai Deep Tetali. Compositional may-must program analysis: unleashing the power of alternation. *POPL'10*, pages 43–56.

[19] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT '06/FSE-14*, pages 117–127, 2006.

[20] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, pages 232–244. ACM Press, 2004.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *29th POPL*, pages 58–70. ACM Press, 2002. SIGPLAN Notices 37(1).

[22] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.

[23] J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *15th CP*, volume 5732 of *LNCS*. Springer, 2009.

[24] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *ICCAD 1995*, pages 2–6.

[25] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. SMT techniques for fast predicate abstraction. In *CAV'06*, pages 424–437.

[26] Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2006.

[27] K. L. McMillan. Lazy abstraction with interpolants. In *CAV'06*, pages 123–136.

[28] K. L. McMillan. Interpolation and SAT-based model checking. In *15th CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[29] K. L. McMillan. An interpolating theorem prover. *TCS*, 345(1):101–121, 2005.

[30] Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. pages 250–??, 2002.

[31] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th DAC*, pages 530–535. ACM Press, 2001.

[32] Andreas Podelski and Andrey Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *PADL*, 2007.

[33] A. Rybalchenko. ARMC: Abstraction Refinement Model Checker. URL http://www.mpi-sws.org/~rybal/armc/.

[34] Hassen Saïdi. Model checking guided abstraction and analysis. In *SAS '00*, 2000.

[35] J. P. Marques Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *ICCAD 1996*, pages 220–227.

[36] A. Stump, C. Barrett, and D. L. Dill. CVC: A cooperating validity checker. In E. Brinksma and K. G. Larsen, editors, *14th CAV*, volume 2404 of *LNCS*, pages 500–504. Springer, 2002.

[8] AR fundamentally depends on the notion of a counterexample, but not AL.

[9] Technically, this is achieved by simply using *true* as the safety property at the terminal points.