On Reasoning About Finite Sets in Software Model Checking

Pavel Shved
Institute for System Programming, RAS
email: shved@ispras.ru

Abstract

A number of static checking techniques is based on constructing and refining an abstract reachability tree (ART) and reasoning about Linear Arithmetics. For example, in BLAST, each program statement is represented as a series of assignments of a linear functions to variables, and the procedure of predicate discovery relies on Craig interpolation of linear arithmetics and equality with uninterpreted function symbols.

In this paper we propose an approach to extend the domain of mathematical operations a checker described can reason about with the certain operations with finite sets: adding and removing elements, testing whether set contains a particular element, or is empty. It being implemented, the ART doesn't split at each operation. The tradeoff of it is more complex formulas for a solver to handle and incapability of using set-related operations in loops.

We implemented the algorithm, proceeding from the resriction of making no modification in LA+EUF interpolation algorithm. We also provide results of the performance evaluation of the algorithm proposed and of the other known way to reason about finite sets.

Index Terms

Software verification and validation

1. Introduction

The approach to static verification through utilizing CEGAR [6] with reasoning about linear arithmetics has been proven succesful by such tools as SLAM [2] and BLAST [5]. The advantage of such approach to analysis is its precision (interprocedural analysis, lower false positive rate) compared to the other static analysis techniques. However, the tradeoff to precision has always been the speed and scalability of the process.

The tools described above work by constructing the locations reachable from the entry point by all syntactically plausible program paths. The locations reached are tagged with a set of linear constraints, which are discovered by an interpolation procedure (in BLAST it's Craig interpolation). These constraints limit the possible state of program variables just roughly, but precise enough to prove unreachability of specially marked "error locations".

The analysis stops when the program proves reachability of the error location or marks it with an empty region (i. e. with an unsatisfiable conjunction of constraints), having examined all possible paths. To avoid exploring an infinite amount of paths (for example, if there's a loop), the checker doesn't explore the locations that are tagged with a contraints implied by the constraints of some other location. The reason is that such exploration would produce a superset of paths and states as the children of the other node would.

All that being said, the analysis algorithm still has to produce different paths if the program branches execution, with an **`if`** statement, for example. Even if the control-flows are then combined together the ART will remain split, and each branch will be tracked separately until one of the branches is proven infeasible. This doubles the time required to verify the program.

It is a natural disadvantage of the approach, but such branching is an inherent property of the program being verified. However, the code verified consists not only of the program code itself, which comes as an input. Another part of code represents the instrumented safety property being verfied, and is a native language implementation of the abstract concepts behind the logic being verified. When an integrated verification system is built, the developers should pay attention that such verification code doesn't add extraneous complexity to the code of the original program.

Some safety properties can be defined in terms of finite sets. For example, a heap model may be represented as a set of currently available memory locations. Each `malloc()` operation adds a unique pointer to that set, and each `free(p)` call deletes a pointer p from that set. At each `free(p)` invocation a check is performed, whether the pointer being freed belongs to that set, and if it doesn't, the control flows to error location. Also at the end of the program a check that that set is empty is performed—i. e. that there's no memory leak.

A classical approach to this problem, universal quantification trick, attributed to [7], is that the set is represented as a single varible. This variable represents one of possible elements currently in the set. This is achieved by the following means. Each opertaion of adding element to set is implemented as an **`if`** statement, one branch of which overwrites the characteristical variable, the other keeping it intact. However, each branch doubles the amount of ART nodes consecutively build, thus expanding it exponentially. Ditto for deletion form the set.

One of the approaches to reducing the numbers of nodes checked is Large-Block Encoding [4]. While this approach is generic and will be of help in the case under consideration,

it suffers from a yet unsolved disadvantage. At the current state of art, it is not known how to produce a useful trace from the entry point to the error location.

The solution we propose is to define set operations that will be treated differently by the checking engine. We present a way to build a path formula that utilizes existing LA+EUF interpolation procedure to devise constraints caused by performing operations on sets. We also describe the model of regions, by which the ART nodes will be tagged.

2. Reasoning about finite sets

To make model checker support reasoning about finite sets, we need to modify certain parts of the algorithm. Second, we need to describe what will be the description of the regions. Third, we should modify predicate discovery algorithm, so that it yields information about finite sets. But at first, we define the concepts the approach works with.

2.1. Concept of a finite set

In this model, each set is recursively defined as either

- **Empty set** that contains no elements, or
- **Union** of an element (or, more precise, of a set that contains one element), which is an arbitrary expression, and of another set; or
- **Subtraction** of an element from a set.

Each set is a finite chain of these operations. That means that at the beginning of each program, all sets are considered empty[1]. This is necessary for all sets to be finite, since an unspecified set (even if it's finite) may have an arbitrarly big lenght.

The relevant operations, as shown in the table 1, operate with sets as with first-class values. These operations are called *set construction operators*, and they shall not branch execution.

Along with some operators to construct sets, a number of *set testing operations* (shown in table 2) are introduced:

- **Check if set contains a particular element**, the element being specified as an expression;
- **Check if set is empty**.

These operations don't branch execution; only examining the result of these checks is what branches execution. However, the common usage pattern is that one of the alternatives of such a check leads to an error state immediately, thus the amount of ART nodes explored doesn't expand exponentially to the number of checks. The problem of the classical quantification trick is in branching that happend at the points of set construction operators.

2.1.1. Definitions used in the article. To shorten the descriptions, we present notations for some concepts used.

Set *constitution* and a set itself are different concepts. Constitution is how the set is constructed, according to set construction operators described in 2.1. A set that has a constitution C is denoted as $[C]$. The set constitution may be denoted like this:

$$C = \emptyset + \{x_1 + z_1\} - \{2 * y_1 + 5\} + \{x_2\} + \{x_3\} - \{y_2\} \quad (1)$$

A number of sets can have a given constitution, depending on other external constraints. For example, given that $x > 5$, any set that contains exactly one number that is greater than ten, has a constitution of $C = \emptyset + \{x + 5\}$.

A set constitution may be presented in a *normalized form*, with consequent add and subtraction operations grouped together. If we unroll the constitution, up to a specific depth, and group together the consequent additions and subtractions, the constitution S may be presented in the following form:

$$S = S_n + \alpha_n - \beta_n + ... + \alpha_1 - \beta_1 \quad (2)$$

where α_i and β_j are sets, and

$$\forall i > 1 \rightarrow (\alpha_i \neq \emptyset) \wedge (\beta_i \neq \emptyset) \quad (3)$$

The normalized form is used to present different set constitutions in a universal way.

As usual, $foo.bar$ means bar "field" in the foo object. The notation is inherited from object-oriented programming and, we suppose, is easier for programmers to follow.

Also the following definitions will be used: given a function f, $D(f)$ is the domain of f, and $E(f)$ is the codomain of f. We will use the notation mostly for maps; thus a codomain of a map is implied to be the smallest set possible.

Table 1. Set construction

`S = SetEmpty();`	place an empty set to S
`S = SetAdd(P,expr);`	place $P \cup \{expr\}$ to S
`S = SetDel(P,expr);`	place $P \setminus \{expr\}$ to S

Table 2. Set examining

`rslt=SetInTest(S,expr);`	check if $expr \in S$
`rslt=SetNotEmptyTest(S);`	check if S is empty

2.2. Predicate discovery algorithm for finite sets

In order to discover predicates specific to the finite sets, we need to build a path formula that describes a concrete state of the program variables in the error location. When we try to add the logic related to finite sets to this formula, it should retain the following properties. First, it should remain **precise**, i.e. only program states that happen during real execution of the program should satisfy the formula. Second, it should be written in **static single assignment form** (or, "SSA form").

Since the formula built without finite sets concept satisfies these properties, we can utilize it for building the formula *with* sets concept. The idea is that set construction operators do not have direct effect to path formula. However, these operators are followed to calculate the exact constitution

1. contrary to integer variables that are considered unspecified untill they're assigned a value

of each set encountered in the trace. The expressions that represent elements of sets are stored in SSA form. When set testing operator is encountered in the trace, the consitution of the set queried is already known; the constraints for the values of set elements are precise, and are described by the rest of the formula.

If the Craig interpolation procedure succeedes the predicates discovered form constraints for program variables. Under certain conditions, discussed in section 2.2.3, these constraints contain enough information to prove that the error path is infeasible. Note that the constraints should not include any symbols of the variables that represent sets. Only "usual" variables (that contain integers) are the domain of these contraints.

2.2.1. Formula for inclusion check. Each check separates the execution into two branches. Path formula is always built among one branch (i.e. the path from root to current node is exactly known). So, to implement inclusion check, for each of these two branches a separate predicate is needed:

1) SAT if the element belongs to the set, UNSAT otherwise
2) UNSAT if the element belongs to the set, SAT otherwise

Let's build the predicate that satisfies the requirements outlined under the first point. It can be built recursively. Given a constitution of a set, C, and the element (expression) x we want to check for inclusion into $[C]$, the predicate is:

- **Empty set**: $C = \emptyset$, then $x \in [C] \Leftrightarrow false$;
- **Union**: $[C] = [C_1] \cup \{y\}$ then $x \in [C] \Leftrightarrow (x = y) \vee (x \in [C_1])$;
- **Subtraction**: $[C] = [C_1] \setminus \{y\}$ then $x \in [C] \Leftrightarrow (x \neq y) \wedge (x \in [C_1])$;

Since error trace is always finite, the recursive unrolling always terminates, and the formula is always built. Since the path formula is in terms of LA+EUF, Craig interpolant always exists [9], and only (SSA-terms of) variables that appear in source code are used in them.

The predicate built that way satisfies the requirement under the first point. Its negation also satisfies the second point. So, we've built proper predicates for both branches for set inclusion check.

2.2.2. Path formula for set emptiness. As described in section 2.2.1, we need to present algorithm for two predicates:

1) SAT if the set is empty, UNSAT otherwise
2) UNSAT if the set is empty, SAT otherwise

We can notice that **a set is empty iff each element that was added to it was removed from it afterwards**.

Given the set $[S]$ such that, in a normalized form (see section 2.1.1),

$$S = S_n + \alpha_n - \beta_n ... \alpha_1 - \beta_1 \tag{4}$$

let's define S^- function as follows:

$$S^-(S, n) = \bigcup_{i=1..n} \beta_i \tag{5}$$

Figure 1. Demonstration of limitation of predicate-discovery algorithm — source code

```
x = 1; y = 2;
S = SetAdd(SetEmpty(), x);
x = 3; y = 1;
S = SetDel(S, y);
if (SetNotEmptyTest(S)) error();
```

Then the predicate

$$(\beta_1 \neq \emptyset) \wedge \left(\bigwedge_{i=1..depth(S)} \bigwedge_{a \in \alpha_i} \bigvee_{b \in S^-(S,i)} (a = b) \right) \tag{6}$$

is true if and only if a set with constitution S is empty, which is the predicate that satisfies the conditions under the first point.

We should note that the predicate (6) is finite, since all the sets α_i are finite and the result of S^- function is a finite set.

Though finite, (6) is nevertheless large, and it also contains disjunctions. This allows us to suggest that check for emptiness will be more complex and less scalable than check for inclusion. We will see if it's correct in section 4.

We could use negation of (6) as a second predicate, but due to its complexity we should search for another way. We note the following. Given a set with constitution S, the second predicate is equivalent to

$$x \in [S] \tag{7}$$

where x is a variable that doesn't appear anywhere in the trace. Indeed, if set is empty then for every x (7) is false, which means that it's unsatisfiable. If set is not empty, then there exists at least one element that belongs to it, and this makes (7) satisfiable.

The negation of (7) doesn't yield the first predicate, since the set $[S]$ is finite, hence there always exists an element that doesn't belong to it; this makes the negation of (7) satisfiable even if set being checked is not empty.

2.2.3. Correctness of predicate discovery. The predicate discovery procedure described above sometimes yields predicates that are false for each program location. Let's consider a sample program shown on figure 1.

If we convert this program to SSA form (and expand set checking predicate), we will get the program shown at figure 2. Interpolation procedure that may yield the predicate $x0 == y1$, it proves that `error()` is inreachable.

However, when we convert it back from SSA form, the predicate would look like $x = y$. If we check figure 1 again, we note that **in no program location this predicate was true**! That means that this predicate discovery procedure based on LA+EUF interpolation can't yield a predicate good enough if the value added to (or removed from) the set was later changed.

Figure 2. Demonstration of limitation of predicate-discovery algorithm — SSA form

```
x0 = 1;
y0 = 2;
S0 = SetAdd(SetEmpty(), x0);
x1 = 3;
y1 = 1;
S1 = SetDel(S0, y1);
if (x0 <> y1) error();
```

Figure 3. Instrumented program for which predicate discovery is correct

```
x = 1; y = 2;
Add1 = x;
S = SetAdd(SetEmpty(), Add1);
x = 3; y = 1;
Rm1 = y;
S = SetDel(S, Rm1);
if (SetNotEmptyTest(S)) error();
```

We can mitigate this obstacle by instrumenting before each set construction function call an assignment to a special variable that doesn't change later (see figure 3). It is possible unless a set construction operator is in a loop.

So, we later assume that **no set construction operation should be in a loop**. We beleive that it's an inherent limitation of our approach. However, even with this restriction the approach amy be useful if it performs well in experiments.

2.3. Regions for finite sets

Some variables encountered during the ART construction are considered to contain sets. The basis, based on which they're considered as such, is the application of relevant operations to them, and storing the result of those into them. The list of supported operations is in table 1. If a result is stored into a variable, or a variable is a set operand of any of those operations, the constitution of such a set is tracked in this ART node and its subsequent children.

The regions are constructed from the very beginning of the processing; this behaviour resembles lattice-shape-analysis [3]. During that initial step a set constitution (in the shape depicted at section 2.1) is tracked. When a value of an expression is added to or removed from a set, the expression is converted by assigning new names to its underlying variables. The converted expression then becomes a part of this set constitution. The rationale is similar to what was discussed in seciton 2.2.3: a value may change after it was added to a set, and set checking should take the old value into account. Instead of checking this requirement, we design region processing in such a way that it tries to work for programs that don't satisfy this condition, but does not guarantee results in this case.

Sometimes constitution tracking alone may be of help. For example, if it's known that a value was recently added to a particular set, the test whether it's empty should return negative response (note the first minterm in (6)). However, in other cases it's insufficient, and the relationship between expressions that represent set elements should be taken into account.

This is where the predicates discovered by the modified trace analysis procedure come into play. Each time a variable is added into a set, all predicates at the current region (we assuime Cartesian abstraction [1]) are also added to the set region, the variables in these predicates being converted in the same way as set elements have been. When determining the post-region of a set testing operation, both these accumulated predicates and set constitutions are taken into account. If they are sufficient to prove the program state infeasible the post operation adds to the cartesian predicate region the predicate that keeps the information about the variable the result is stored in.

Formally speaking, each region $reg = post(reg', e)$ consists of the following components:

- $reg.binding : variable \rightarrow variable$ (such that $E(reg.binding) \cap D(reg.binding) = \emptyset$) is the current mapping between actual variables, in terms of which the CFA blocks are expressed, and those used within sets. This binding is used to convert elements and predicates at current point. It is updated at each location, when e is a basic block of assignments.
- $reg.sets : variable \rightarrow constitution$ is a mapping from program variables to set constitutions (described in section 2.1). Elements of the sets are expressions over the $E(reg.binding)$ set of variables. This mapping is updated at each construction operation (see fig. 1); the mapping between how constitution is updated and what operation is along the edge is obviously inferred from the description in seciton 2.1;
- $reg.predicate_i$ is the predicate over the $E(reg.binding)$ set of variables. In a post-region of reg' after an edge e, a predicate belongs to this set if and only if:
  - it belongs to $reg'.predicate$ set; or
  - it is equal to one of the predicates in cartesian region in $post(region', e)$, the variables being updated according to $reg.binding$.

When at the location a set testing operator is encountered, the region machinery should decide whether the post-region is feasible. To notify the model checker that an infeasible location is encountered, the predicate over the variable the result is stored into is added to the Cartesian abstraction region.

In a current region reg such a predicate should be added when testing operation in e should fail on each set constitution that belongs to reg' region. Having rewritten the element expression involved in set testing operation with use of $reg'.binding$, and having applied the relevant formula of those in section 2.2, we get a formula $F_{[S]}(E(reg.binding))$. That is the exact formula

Figure 4. Header file with set-related functions

```
typedef int Set;
Set SetEmpty();
Set SetAdd(Set, ...);
Set SetDel(Set, ...);
int SetInTest(Set, ...);
int SetNotEmptyTest(Set);
```

that would have appeared in this location if we started error trace analysis now.

However, such analysis, if it had already been completed for a path this location belongs to, had yielded several interpolants. We know that the error trace—even before it's converted to SSA—already has some variables not changing their values (these are the instrumented variables introduced in section 2.2.3), so these values will be reasoned about in interpolants. These interpolants provide the necessary predicates to prove set-related path infeasible if it really is (since the formulas are equivalent). It means that the formula (with unbound variables $E(reg.binding)$)

$$F_{[S]} \wedge \bigwedge_{p \in reg.predicates} p \tag{8}$$

is UNSAT if the path to current location is on a subpath of an infeasible error path checked for an error before. However, if Cartesian abstraction succeeds in propagating set-related predicated to other paths, this formula may work too. That is one of the main benefits of lazy analysis [8].

2.3.1. Region coverage. Region machinery should address another problem: region coverage. For finite sets it is possible, but unnecessary to devise the relevant formula.

Since finite sets can not be operated with in cycles, there will never be a situation when one region would cover another one due to reasons devised when analyzing set-related operations. So the usual coverage checking procedure would suffice.

3. How it is embedded into BLAST

A special header with C funciton headers is created (see figure 4). It utilizes variadic arguments, since it's not known, what the types of the expressions added to sets will be.

When BLAST interprets the source code of a program that uses these functions, it replaces each function call with a separate edge in the CFA. Set functions are undefined, but a separate edge will be created anyway.

Set construction functions will be inserted as is. Set testing funcitons are intended to use in bodies of **if** operators. However, they will anyways be represented with a separate edge, the program being transformed to something like this:

```
tmp = SetInTest(S, expression);
if (tmp != 0) { ... } else {...}
```

Figure 5. Memory operations implemented in terms of finite sets

```
int counter = 1;
/* Set of allocated regions */
Set memory;
ptr malloc()
{ counter += 1;
  memory = SetAdd(memory, counter);
  return counter; }
void free(void* p)
{ if (!SetInTest(memory,p)) error();
  memory = SetDel(memory,p); }
void check_leaks()
{ if (SetNotEmptyTest(memory)) error(); }
```

Figure 6. Memory operations implemented with path splitting

```
int maybe(); //returns an arbitrary bool
int counter = 1;
/* One of the pointers in set */
void* M = 0;
/* One of the pointers deleted from set */
void* F = 0;
ptr malloc()
{ counter += 1;
  if (maybe()) M = counter;
  return counter; }
void free(void* p)
{ if (M == p) M = 0;
  if (F == p) error();
  if (maybe()) F = p; }
void check_leaks()
{ if (M != 0) error(); }
```

To avoid branching at the point of function call assignment, the predicates for such a check devised in section 2.2 $pred$ are utilized in the following formula:

$$(tmp = 1 \wedge pred_1) \vee (tmp = 0 \wedge pred_2) \tag{9}$$

where $pred_1$ and $pred_2$ are the predicates devised in each of secitons 2.2.2 and 2.2.1: one is satisfiable when condition being checked is true, and the other is satisfiable when it's false.

This doesn't affect correctness of any claims made above, although requires some extra prover work as formulas produced this way contain more disjunctions.

4. Performance evaluation

We evaluated our algorithms for simple programs. Each test program consists of consequent allocations of several

Table 3. Evaluation results (seconds elapsed; "X" means CSIsat failure)

# of regions allocated	1	2	3	4	5	6	7	8	9	10	15
"Trick" with checking leaks	1	5	52	540	1553	> 2000	> 2000	> 2000	> 2000	> 2000	> 2000
Sets with checking leaks	1	4	10	X	80	X	X	X	X	X	X
"Trick" without checking leaks	1	4	41	443	1289	> 2000	> 2000	> 2000	> 2000	> 2000	> 2000
Sets without checking leaks	1	3	6	17	36	70	200	333	X	X	X

memory regions and consequent deallocations of them, followed by an optional check for unfreed memory. We also introduced double-free errors and memory leaks to verify correctness of our approach. All such tests didn't demonstrate any errors in the algorithms used.

The checking for memory operation safety was similar to that presented in the introduction. For finite set it's presented on figure 5, and for universal quantification trick—on figure 6[2].

The complexity of the test programs range from one to fifteen allocated regions. We also thought that it would be fruitful to check how algorithms behave in absense of leak checking, because formula to check emptines. (6), is more complex than other set-related formulas. The results are in table 3.

We found out that after amount of allocations exceeds a certain limit, algorithms start failing due to the failures of the underlying Craig interpolant generator, CSIsat. We believe that the reason is that formulas fed to it appear to be more complex than it is capable to handle. We also see that leak checking (i. e. emptiness checks) is way more complex than reasoning if an element belongs to a set.

5. Conclusion

In this paper we showed that certain common properties checked by static analysis frameworks can be represented in terms of finite sets. Sections 2.1 and 2.2 contain sound algorithms to discover predicates and refine abstraction with utilization of set-related functions in C code.

The algorithms proposed have a serious limitation: each variable in expressions being added to/removed from sets may not be used later in the code. Further research could discard this requirement if it extended the logic used beyond LA+EUF, by adding some set-related concepts into interpolation procedure. In this article we failed to demonstrate that it is unnecessary.

The concepts described here were prototyped as a patch for BLAST of version 2.5, the prototype being of low quality and was just to check if the concepts presented here are correct and viable.

Several simple artificial programs were generated for the experiments. The tests held compare the approach proposed with already known solutions. It's clear that our novel algorithm performs better than the known one. However, the complexity of formulas generated limits the scalability of the approach.

2. note that in the original paper [7] the elaborated algorithm was not presented. The algorithm on figure 6 was devised by us, but we beleive that Bandera tool has something similar.

Thus, the approach proposed in the paper doesn't perform well in the experiments with the currently used interpolating tools. Given also the severe limitations on its applicability, we think that further improvement of the prototype developed to make it useful in industrial application is ineffectual.

References

[1] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking c programs. *Proc. TACAS*, page 268–283, 2001.

[2] T. Ball and S.K. Rajamani. The slam project: Debugging system software via static analysis. *Proc. POPL*, page 1–3, 2002.

[3] D. Beyer, T.A. Henzinger, and G. Théoduloz. Lazy shape analysis. *Proc. CAV, LNCS*, 4144:532–546, 20006.

[4] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2009, Austin (TX), November 15-18)*, pages 25–32. IEEE Computer Society Press, Los Alamitos (CA), 2009.

[5] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker blast. *Int J Softw Tools Technol Transfer*, 9:505–525, 2007.

[6] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Proc. CAV, LNCS*, 1855:154–169, 2000.

[7] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, Psreanu, Robby C.S., and H Zheng. Zheng, h.: Bandera: extracting finite-state models from java source code. *ICSE'00: Proc. 22nd Intl. Conf. on Software Engineering*, page 439–448, 2000.

[8] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *Proc. POPL*, page 58–70, 2002.

[9] K.L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, pages 101–121, 2005.