Predicate Abstraction of ANSI–C Programs using SAT*

Edmund Clarke† and Daniel Kroening† and Natalia Sharygina‡ and Karen Yorav†

†School of Computer Science
Carnegie Mellon University,
Pittsburgh, PA, USA

‡Software Engineering Institute
Carnegie Mellon University,
Pittsburgh, PA, USA

Abstract

Predicate abstraction is a major method for verification of ANSI-C programs. However, the generation of the abstract Boolean program from the set of predicates and the original program suffers from an exponential number of theorem prover calls as well as from soundness issues. This paper outlines an on-going project that uses an efficient SAT solver for generating the abstract transition relation of C programs. The SAT-based approach computes a more precise and safe abstraction compared to the existing predicate abstraction techniques.

1 Introduction

It is widely believed that effective model checking [5] of software systems could produce major enhancements in software reliability and robustness. However, the effectiveness of model checking of software systems is severely constrained by the state space explosion problem. One principal method in state space reduction of software systems is abstraction. Abstraction techniques reduce the program state space by mapping the set of states of the actual system to an abstract set of states that preserve the actual behaviors of the system. Abstractions are most often performed in an informal, manual manner, and require considerable expertise.

Predicate abstraction [19, 9, 1], is one of the most popular and widely applied methods for systematic abstraction of programs. It abstracts data by only keeping track of certain predicates over the data variables. Each predicate is represented by a Boolean variable in the abstract program, while the original data variables are eliminated. The abstract program is created using *Existential Abstraction* [7], resulting in an over-approximation of the set of behaviors of the original program. When Model Checking of the abstract program fails it may produce a counterexample that does not correspond to a concrete counterexample; this is called a *spurious counterexample*. Consequently, the set of predicates is refined heuristically, and a new abstraction is computed.

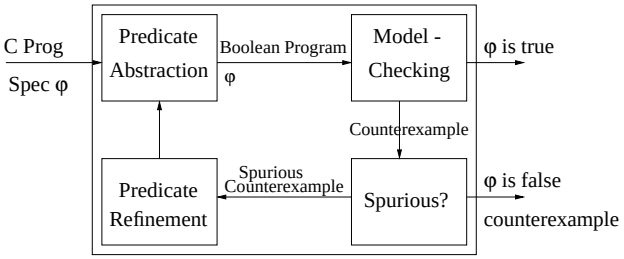

Figure 1. The CEGAR Framework

The abstraction refinement process has been automated by the *Counterexample Guided Abstraction Refinement* paradigm [15, 6, 11], or CEGAR for short. This framework is shown in Figure 1: one starts with a coarse abstraction, and if it is found that an error-trace reported by the model checker is not realistic, the error trace is used to automatically refine the abstract program, and the process proceeds until no spurious error traces can be found.

Related Work. Data abstraction techniques are widely used and they have been explored by a number of researchers [7, 15, 17, 14]. Abstraction techniques are often based on abstract interpretation [10] and require the user to give an abstraction function relating concrete data-types

* This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and the General Motors Collaborative Research Lab at CMU. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, DOD, ARO, or the U.S. government.

to abstract data-types. Earlier applications of the predicate abstraction type of the abstract interpretation approach [19, 3, 9] were dependent on the user identifying the set of predicates that influence the verification property and were utilizing general-purpose theorem proving to compute the abstract program. The user-driven discovery of relevant predicates makes them less effective for large programs.

Recently, various decision procedures have been proposed to compute the set of predicates for the abstraction. The most common approach is to use error traces [6, 1] to guide the discovery of predicates. In [6], the algorithm is based on BDD representations of the program. This is a draw back for large programs, where transition relation BDDs are commonly too large for efficient manipulation. The algorithm from [1] uses an explicit state space representation but it is restricted to safety properties.

In previous work, including [2, 13], the generation of the abstract Boolean program from the C program and the set of predicates suffers from multiple problems:

- The generation of the Boolean program is done by calling a theorem prover for each potential assignment to the current and next state predicates. For the most precise transition relation, this requires an exponential number of calls of the theorem prover. Several heuristics are used to reduce this number. Existing tools stop the computation after a user-specified number of calls, and add all remaining transitions for which the theorem prover call was skipped. This is a safe over approximation, but will yield a potentially large number of unnecessary spurious counterexamples.

- Existing tools - with the exception of [4] - use general-purpose theorem provers. Program variables are modeled as unbounded integer values, neglecting a possible arithmetic overflow in the ANSI-C program. This can result in false positive answers of the tool.

- Existing tools only support a very limited range of operators, namely Boolean operators, addition/subtraction, equality, and relational operators. Other ANSI-C operators, such as multiplication and division, bitwise operators, type conversion operators, and shift operators are modeled by means of uninterpreted functions. This limits the set of programs and the properties that can be verified.

- Existing tools only provide a limited support for pointer operations. In particular, pointer arithmetic is not handled.

Contribution. This work proposes to use a SAT solver to generate the abstract program. The goal of the SAT solver application is to replace the potentially exponential number of theorem prover calls by a single SAT instance.

For each basic block in the given program, our approach is to first construct a symbolic representation of the concrete (non-abstracted) transition relation by applying symbolic simulation techniques. Next, we add the predicates in current and next state form to the relation between variables, resulting in a Boolean formula. Finally, we enumerate symbolically on the values of the predicates, using a SAT solver. When the abstract program needs to be refined, we use the same formula that we have already created, together with the new set of predicates, to create the new abstraction.

The advantage of this technique is that the exponential number of theorem prover calls is eliminated; instead, the possible assignments to the values of the predicates are searched by the SAT solver. Modern SAT solvers are very efficient, and allow a large number of variables, enabling us to discover many more possible assignments. Thus, one obtains a more precise abstract transition relation, eliminating redundant spurious counter examples.

Another advantage of our approach is that all the ANSI-C constructs can be encoded using CNF, which allows a wider range of programs. Integer operators are encoded using bit vector operators, i.e., they take the potential arithmetic overflow into account. Thus, there are no false positive answers due to the inaccurate assumption that the range of values of the variables is infinite. Moreover, pointer manipulation constructs, including pointer arithmetic, can also be supported. The only limitation is that recursion and dynamic memory allocation are not allowed. This limitation cannot be avoided, since the Boolean program is required to be finite. The symbolic simulation technique we propose to use is taken from [8].

2 A Boolean equation for the Concrete Transition Relation

The concrete transition relation is represented by a Boolean equation that captures the semantics of the program. At this point we only translate basic blocks, which are program segments that contain only sequentially composed assignments. The control flow of the program is left intact, and will be handled later.

Assume a given basic block. At this point we have already performed a pre-processing step that eliminates function calls. The first step is to transform the block into single assignment form, so that a variable is not assigned twice within the same block. Let the program refer to variable v at a given program location. Let α denote the number of assignments made to variable v prior to the location. The variable v is then renamed v_α. Within assignments to v, the expression on the right hand side is considered to be before the assignment. The variable that is assigned to on left the hand side is considered to be after the assignment. Let e denote an expression, then $\rho(e)$ denotes the expression after

renaming. Following is an example of a simple block and its translation:

```
x = z*x;              ρ        x1 = z*x0;
y = x + 1;      ⟶              y1 = x1 + 1;
x = x + y;                     x2 = x1 + y1;
```

For variables that are not assigned into, the final value is the version with index 1, and we will make sure that $v_0 = v_1$. We use v' as a shorthand for the largest index used for v (the final value), while v is a shorthand for the initial value v_0. Thus, the transition relation we are defining is a relation $T(\overline{v},\overline{v}')$, where $\overline{v}$ is the vector of program variables. In the following, we use v for a program variable (such as `x` in the example above) and v_j for one of its renamed versions (`x0, x1, x2` in that example).

We now generate an equation $eq(v_j)$ for each (renamed) variable with $j > 0$, describing what happens to v_j after executing this code. The equations are created using the following rules:

Case 1 If v is not assigned to in the block we define

$$eq(v_1) ::= (v_1 = v_0)$$

Case 2 For a simple variable v, i.e., not of an array or structure type, and an assignment `v`$_j$`= exp`, we have

$$eq(v_j) ::= (v_j = exp)$$

Note that `v`$_j$`= exp` is the assignment after the translation into single assignment form, thus all the variables in exp are already renamed.

Case 3 If v is an array, let a be the array index, i.e., the assignment is $v_j[a] = exp$. In this case we create an equation that states that the value of v_j, at index i is equal to exp if $i = a$ and equal to $v_{j-1}[i]$ otherwise:

$$eq(v_j) ::= \bigwedge_i (v_j[i] = ((i = a)?exp : v_{j-1}[i]))$$

The "?" operator is a Boolean choice operator. The expression $x?a : b$ evaluates to a if $x = 1$ and to b if $x = 0$. This form is more efficient, in terms of the number of clauses generated, than the equivalent formula using only the basic Boolean operators. Assignments to variables that have a `struct` type are handled in a similar manner.

Case 4 When an array is used on the right hand side of the assignment we use a case split on the array index to reference it. For example, the assignment $v_j = v_i[a]$ generates the formula:

$$eq(v_j) ::= \bigwedge_k ((k = a) \rightarrow (v_j = v_i[k]))$$

Case 5 When an address is assigned to a pointer variable we treat it as any simple assignment. The address of a variable is a value that we can compute at compile time. For the assignment $p_j = \&x$ we have:

$$eq(p_j) ::= (p_j = \&x)$$

Case 6 When a pointer is dereferenced we use a case split on all the variables that match the type of the pointer. Let p be a pointer variable, and V_p be the set of variables to which p can point. The assignment $v_j = *p$ results in the following:

$$eq(v_j) ::= \bigwedge_{x \in V_p} (p = \&x) \rightarrow v_j = x$$

This equation can be simplified if the predicates we are about to use for the abstraction include information about the possible variables in V_p. However, at this point we give the general solution that does not rely on the set of predicates.

Case 7 The case where a pointer dereference appears on the left hand side of an assignment is handled by a transformation of the program rather than an equation. The assignment `*p = exp` is capable of affecting any of the variables in V_p. We therefore replace this assignment with a series of assignments – for each variable $v \in V_p$ we add the assignment

```
v = (p=&v) ?  exp :  v
```

This transformation of the program is done *before* the renaming step. The renamed program does not have pointer dereferences on the left hand side of assignments, and can be translated into an equation system using the previous rules. For example, assuming that the variable p can point to both x and y, the following transformation occurs:

```
x = 5;              x1 = 5;
*p = y + x;  →      x2 = (p==&x) ? (y0+x1) : x1;
x = x + 1;          y1 = (p==&y) ? (y0+x2) : y0;
                    x3 = x2 + 1;
```

Putting together the equations for all variables we get a bit-vector equation such that each solution represents a possible computation of the basic block. We note again that our tool has support for *all* bit-vector operations, and the implementation is sound because it takes into account overflow situations. We show an example of the process described above in Figure 2. The example gives a basic block, the renamed version, and the resulting equation system.

3 Computing the Abstraction

Let P be the set of predicates, where each predicate is an expression over the (concrete) program variables. Each predicate $p_i \in P$ is associated with a Boolean variable b_i that represents its truth value. These Boolean variables are the variables of the Boolean program we are constructing. Let $\overline{p}$ denote the vector of predicates, and $\overline{b}$ denote the vector of the Boolean variables. The predicates map a concrete

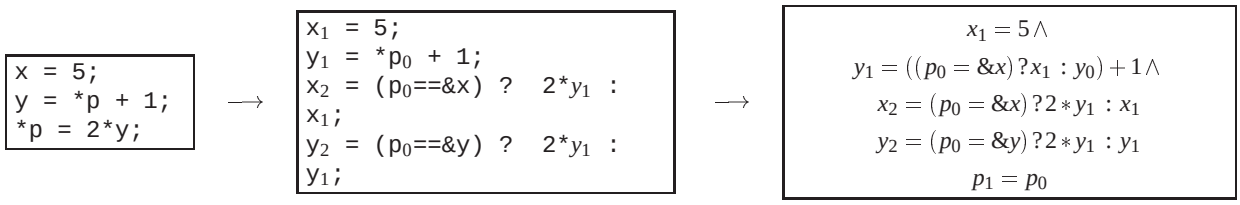

Figure 2. Example: Generation of the concrete transition relation

state $\overline{v}$ into an abstract state $\overline{b}$, and thus $\overline{p}(\overline{v})$ is also called the abstraction function. Given $T(\overline{v},\overline{v}')$ and P, we create an abstract transition relation $B(\overline{b},\overline{b}')$ that is an existential abstraction of the C program.

Our goal is to replace a basic block with an expression that describes what happens to the variables $\overline{b}$ when this basic block is executed. We present a translation that is accurate, i.e., it gives the transition relation as defined by existential abstraction, and not an over-approximation of it as other tools use.

Let $T(\overline{v},\overline{v}')$ denote the concrete transition relation, as defined in the previous section. The abstract transition relation $B(\overline{b},\overline{b}')$ relates a current state $\overline{b}$ (before the execution of the basic block) to a next state $\overline{b}'$ (after the execution of the basic block). It is defined using $\overline{p}$ as follows:

$$\Gamma(\overline{b},\overline{b}',\overline{v},\overline{v}') \triangleq (\overline{p}(\overline{v}) = \overline{b}) \wedge T(\overline{v},\overline{v}') \wedge (\overline{p}(\overline{v}') = \overline{b}') \quad (1)$$

$$B(\overline{b},\overline{b}') \iff \exists \overline{v},\overline{v}' : \Gamma(\overline{b},\overline{b}',\overline{v},\overline{v}') \quad (2)$$

The concrete transition relation $T(\overline{v},\overline{v}')$ is given as a bit vector equation, as described in the previous section. In order to obtain $B(\overline{b},\overline{b}')$, we translate $\Gamma(\overline{b},\overline{b}',\overline{v},\overline{v}')$ into CNF. Every satisfying assignment to $\Gamma(\overline{b},\overline{b}',\overline{v},\overline{v}')$ represents a concrete transition and its corresponding abstract transition. We aim at obtaining all possible satisfying assignments to the abstract variables $\overline{b}$ and $\overline{b}'$, i.e., the set

$$\{ (\overline{b},\overline{b}') \mid B(\overline{b},\overline{b}') \} \quad (3)$$

This set is obtained by modifying the SAT solver Chaff as follows: Every time a satisfying assignment is found, the tool records the values of the literals corresponding to the abstract variables $\overline{b}$ and $\overline{b}'$, and then adds a *blocking clause* in terms of these literals that eliminates all satisfying assignments where these variables have the newly found values. The literals in the blocking clauses all have a decision level, since the assignment is complete. The solver then backtracks to the highest of these decision levels and continues its search for further, different satisfying assignments. Thus, the SAT solver is used to enumerate the set (3). This technique is commonly used in other areas, for example in [18, 12]. In [16], the same algorithm we are using is used to enumerate symbolic solutions to predicate abstraction formulas. On the SLAM benchmarks, it outperforms BDDs in most cases. However, the work does not use bit-vector logic.

As an example, consider the following basic block:

```
d=e;
e++;
```

Suppose the predicates $p_1 = d\&1$ and $p_2 = e\&1$ are given. The operator & is the bitwise conjunction operator, i.e., p_1 holds if and only if `d` is odd, and p_2 holds if and only if `e` is odd. The basic block is translated into the following transition relation:

$$(d_1 = e_0) \wedge (e_1 = e_0 + 1) \quad (4)$$

By adding the constraints for the predicates we get:

$$\begin{array}{c}(b_1 = d_0\&1) \wedge (b_2 = e_0\&1) \wedge \\ (d_1 = e_0) \wedge (e_1 = e_0 + 1) \wedge \\ (b_1' = a_1\&1) \wedge (b_2' = e_1\&1)\end{array} \quad (5)$$

The satisfying assignments for this equation over the variables b_1 and b_2 are:

b_1	b_2	b_1'	b_2'
1	0	0	1
0	1	1	0

In particular, the abstract Boolean program will never make a transition that is contradictory in the sense that both `e` and `e+1` are odd. This is unavoidable if a next state function is computed separately for each Boolean variable in $\overline{b}$, as done by many existing tools.

Consider the basic block above with the predicates $p_1 = e \geq 0$ and $p_2 = e \leq 100$, and suppose that `e` has 32 bits. The equation for the abstract transition relation B is:

$$\begin{array}{c}b_1 = e_0 \geq 0 \wedge b_2 = e_0 \leq 100 \wedge \\ d_1 = e_0 \wedge e_1 = e_0 + 1 \\ b_1' = e_1 \geq 0 \wedge b_2' = e_1 \leq 100\end{array} \quad (6)$$

The satisfying assignments for this equation over the variables b_1 and b_2 are:

b_1	b_2	b_1'	b_2'
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	1	0
1	1	1	1

Note that incrementing a positive number is not guaranteed to yield another positive number because of the finite range (there is a transition from a state with $b_1 = 1$ to a state with $b_1' = 0$).

Besides basic blocks, the concrete program also contains control flow statements such as `if` and `while`. These statements take a condition as an argument, and change the value of the program counter accordingly. To compute the corresponding abstract transitions we can use the SAT solver in a similar way to what was done for basic blocks. In practice, however, this is rarely necessary, since the conditions of the `if` and `while` loops are often chosen as one of the Boolean predicates. In this situation no translation is needed.

4 Conclusion

The paper describes a new method to compute the Boolean abstraction of an ANSI–C program using SAT. It overcomes the limitations of existing tools, which overapproximate and provide support for only few operators. A full implementation of this technique is currently underway.

References

[1] T. Ball, R. Majumdar, T.D. Millstein, and S.K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *The 8th International SPIN Workshop on Model Checking of Software, LNCS vol. 2057*, pages 103–122. Springer, May 2001.

[3] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427, pages 319–331, Vancouver, Canada, 1998. Springer-Verlag.

[4] S. Chaki, E.M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 2003. To appear.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.

[6] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.

[7] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Principle of Programming Languages*, January 1992.

[8] E.M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *40th Desgin Automation Conference*, 2003. To appear.

[9] M. Colon and T.E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification*, pages 293–304, 1998.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages, POPL '77*, pages 238–252, 1977.

[11] S. Das and D.L. Dill. Successive approximation of abstract transition relations. In *LICS*, 2001.

[12] A. Gupta, Z. Yang, P. Ashar, and A. Gupta. SAT-based image computation with application in reachability analysis. In *FMCAD*, 2000.

[13] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[14] Y. Kesten and A. Pnueli. Control and data abstraction: cornerstones of the practical formal verification. *Software Tools and Technology Transfer*, 2(4):328–342, 2000.

[15] R.P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[16] S.K. Lahiri, R.E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *15th Computer-Aided Verification conference*, Boulder, Colorado, 2003.

[17] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:11–45, 1995.

[18] K.L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *14th Conference on Computer Aided Verification*, pages 250–264, 2002.

[19] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th INternational Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.