

CMSC631 Notes on *Automatic Predicate Abstraction of C Programs*

Vasile Găburici
gaburici@cs.umd.edu

December 3, 2002

Abstract

These notes try to “fill in the blanks” in the paper by providing a minimal but necessary background from the references. They do not attempt to be an extended abstract of the paper itself. I conclude with a few comments.

1 Model Checking of Programs

Predicate abstraction from real programs is an important issue in the more general problem of model checking. Model checking for arbitrary programs is an undecidable problem. As Thomas Ball states it in one of his SLAM presentations, that reason alone won't make the problem go away. For restricted models of computation in the areas of hardware and network protocol validation, model checking has achieved some non-trivial successes. For these types of applications, there number of states of the system is normally finite.

Interesting programs have a potentially infinite number of states. (Well, it may be finite at limit due to finite representation of numbers etc., but still unmanageable). Reducing a system with potentially infinite number of states to a systems with a finite number of states requires an abstraction mechanism. This mechanism can be a form of abstract interpretation [P11]¹.

The SLAM model checking toolkit deals with real C programs by decomposing the model checking process in three stages:

- abstraction - the C2BP tool
- model checking in the abstract domain - the BEBOP tool

¹All references prefixed by the letter P are from the original paper.

- counterexample concretization - the NEWTON tool

Model checking in the abstract domain is conservative: model checking terminates successfully if the abstract model is okayed by BEBOP. The NEWTON tool comes into play only if there is an error in the checking the abstract model. NEWTON performs a concretization of the error, i.e. it generates series of transitions of the original C program that lead to the error.

The focus of the paper is the C2BP tool. Before we dwell into the abstraction mechanism itself, it is important to outline what are the main advantages of model checking in the abstract domain.

2 Boolean Programs

The abstraction that BEBOP uses is a boolean program. A boolean program is a program where all the variables have boolean type, but general (C-like) control flow statements are allowed, including function calls. Due to the finite amount of storage accessible at any program point, a boolean program is computationally equivalent in power to a push-down automaton [P5]. Therefore, the reachability and termination problems (undecidable for Turing machines) are decidable for boolean programs.

BEBOP solves the reachability problem for boolean programs: given a statement s in a boolean program, BEBOP outputs a *shortest* path from some initial state that reaches s .

It turns out that the reachability problem for boolean programs is a *locally separable* (gen-kill) instance of IFDS (interprocedural, finite, distributive, subset) precise dataflow-analysis problem.

Precise interprocedural dataflow analysis is more difficult than intraprocedural dataflow analysis, because “precise” in this context means “meet-over-all-valid-paths” as opposed to “meet-over-all-paths”; an interprocedural path is valid if it respects the fact that when a procedure terminates it returns to the site of the most recent call. IFDS problems may be solved in polynomial time wrt. the size of the flow graph using the algorithm due to Reps, Horwitz, and Sagiv (RHS) [P28]. For locally locally separable problems, this algorithm requires $O(ED)$ time, where E is the number of edges in the flow graph, and D is the size of the dataflow facts set.

Of the key ideas of the RHS algorithm is the computation of *summary edges* which describe the “behavior” (aggregate transfer function) of a procedure call; these are computed only once for each procedure call, thus the algorithm naturally handles recursive calls, and so can BEBOP.

The (interprocedural) control-flow graph for a boolean program B with n statements and p procedures is $(V_B, Succ_B)$, where $|V_B| = n + p + 1$ because p additional nodes are introduced for procedure exits, and one node for all assertion failures. The successors of node are given by $Succ_B : V_B \rightarrow \mathcal{P}(V_B)$, and rules for their construction are derived from the grammar of boolean programs.

The definition of transfer function (and summary edges) for BEBOP is based on *valuations*. Given a set V of variables in B , a valuation Ω is a function that associates a value to every variable in V . Valuations are also defined over expressions; e.g. if $\Omega(x) = 1$ and $\Omega(y) = 0$ then $\Omega(x \& y) = 0$.

A transfer function over a vertex $v \in V_B$ is simply a predicate that is true for any pair of valuations $\langle \Omega_1, \Omega_2 \rangle$, where Ω_1 is a valuation immediately before v , and Ω_2 is a valuation immediately after v . For instance, the vertex corresponding to the assignment $\mathbf{x} := \mathbf{e}$ has the transfer function $\lambda \langle \Omega_1, \Omega_2 \rangle. (\Omega_2 = \Omega_1[x/\Omega_1(e)])$, where $[x/y]$ means substitution of x with y .

Representing the transfer functions is therefore a difficult problem itself, because these functions may be arbitrary predicates. A useful representation is a binary decision diagram (BDD). A BDD is a directed rooted graph $G(V)$, in which we distinguish terminal and non-terminal vertices. A nonterminal vertex v has an $index(v) \in \{1, \dots, n\}$ and two children $low(v), high(v) \in V$, while a terminal vertex has a value $value(v)$. A BDD having root vertex v encodes the function:

$$f_v(x_1, \dots, x_n) = \begin{cases} value(v) & \text{if } v \text{ is terminal} \\ \neg x_{index(v)} f_{low(v)} + x_{index(v)} f_{high(v)} & \text{if } v \text{ is nonterminal} \end{cases}$$

In [P9] a function graph is defined as a BDD with the additional constraint that when $low(v)$ and $high(v)$ are nonterminals $index(low(v)) < index(v) < index(high(v))$. Algorithms for all binary operations (or, and etc.) on two function graphs G_1, G_2 are polynomial: $O(|G_1||G_2|)$. Furthermore, given any function graph G of a function f , a *reduced* canonical function graph may be obtained in $O(|G| \log |G|)$. This allows the satisfiability problem to be reduced to comparing a reduced function graph with the graph for function $\mathbf{0}$. This does not alleviate the fundamental problem of any boolean function representation using a BDD: many functions require an exponential size graph.

Summarizing all the above, the complexity of the BEBOP algorithm is $O(2^k E)$, where k is the number of variables *in scope* at any point in the program. In practice, the number of variables in scope is modest for reasonable programs, so the authors claim that programs with thousands of lines of code and thousands of variables (in total, not in scope at a given point) can be analyzed in minutes by BEBOP.

3 Abstract State Graphs

Given a C program P and a set $E = \{\varphi_1, \varphi_2, \dots, \varphi_n\}$ of boolean expressions over the variables of P (and C constants), C2BP builds a boolean program $\mathcal{BP}(P, E)$ with variables $V = \{b_1, b_2, \dots, b_n\}$; each variable representing the corresponding predicate. The boolean program \mathcal{BP} is an abstraction in sense of [P19] of the C program P .

[P19] is not the seminal work on abstract state graphs, but is the first abstraction scheme that allowed a network protocol to be checked without user intervention. The abstract state graph is defined in the context of guarded assignment programs used in PVS (Prototype Verification System), thus allowing for concurrency; it is trivial to reduce the definition to a serial program by introducing a program counter which is tested in every guard for the appropriate value, and then assigned so that it enables the unique successor instruction.

A process is defined as a set of typed variables $\{x_1 : T_1, \dots, x_n : T_n\}$, a set of transitions $\{\tau_1, \dots, \tau_p\}$, and a set of initial states: *init*. Each transition is a guarded assignment $g_i(\vec{x}) \rightarrow \vec{x} := \text{ass}_i(\vec{x})$. A state graph for the process P is defined as $S_P = (Q_P, R_P, I_P)$, where

- $Q_P = T_1 \times \dots \times T_n$ is (the complete) set of states
- $R_P = \bigcup_{i=1}^p \tau_i$, where $\tau_i(q) = \begin{cases} \perp & \text{if } g_i(q) \equiv \text{false} \\ \text{ass}_i(q) & \text{otherwise} \end{cases}$
is the (partial) transition function associated with the transition² τ_i .
- $I_P = \{q \mid \text{init}(q) \equiv \text{true}\}$ is the set of initial states.

A set of states $s \in Q_P$ can be represented by a predicate (assertion) φ that is true only for those states. The natural partial order over $\mathcal{P}(Q_P)$ induces a partial order over the assertions: if $s_1 \supseteq s_2$ then $\varphi_1 \Rightarrow \varphi_2$, or φ_1 is a precondition of φ_2 . A domain can also be build for assertions: the least upper bound is the weakest precondition, and so on. Given this equivalence, abusing the notation, we will use $\varphi \in \mathcal{P}(Q_P)$ as state set.

Recall that a given a binary relation R (e.g. successor instruction) on a set Q and $\varphi \in \mathcal{P}(Q_P)$, the strongest postcondition is defined as $\mathbf{post}[R](\varphi) = \exists q' R(q', q) \wedge \varphi(q')$, which means that $\mathbf{post}[R](\varphi)$ is a predicate representing the set of successors of φ “reachable” using the “edges” from R .

Finally we may define what an abstraction means. Let $S = (Q, \bigcup \tau_i, I)$ be a state graph, Q^A a lattice and $(\alpha : \mathcal{P}(Q) \rightarrow Q^A, \gamma : Q^A \rightarrow \mathcal{P}(Q))$ a Galois connection³. $S^A = (Q^A, \bigcup \tau_i^A, I^A)$ is an abstraction of S iff:

²The astute reader will notice French overloading of symbols.

³ $\alpha(\gamma(q^A)) = q^A$ and $\varphi \Rightarrow \gamma(\alpha(\varphi))$.

- $I \subseteq \gamma(I^A)$: abstract initial states represents at least all initial states
- $\forall i \forall q^A \in Q^A \gamma(\tau_i^A(q^A)) \Rightarrow \mathbf{post}[\tau_i](\gamma(q^A))$: the set of successors⁴ of an abstract state q^A by some abstract transition τ_i^A is a superset of the set of successors of the concrete states represented by q^A by the corresponding concrete transition τ_i .

This construction ensures that every concrete execution sequence is represented by at least one abstract sequence, which is basically abstract interpretation of state graphs.

For a boolean program $\mathcal{BP}(P, E)$, the abstract state lattice Q^A is induced by the set E of predicates. It is the set of (potentially all⁵) predicates on $|E| = n$ boolean variables. Each variable b_i represents all concrete states satisfying the predicate φ_i .

An arbitrary element of Q^A is a boolean expression on the variables b_1, \dots, b_n : $\exp^A(\vec{b})$. Obtaining the set of concrete states represented by $\exp^A(\vec{b})$ is only a matter of substituting each abstract variable by the corresponding predicate: $\gamma(\exp^A(\vec{b})) = \exp^A(\vec{\varphi}/\vec{b})$.

The abstraction function is however impractical to compute; it is a least upper bound (weakest precondition) for arbitrary predicates:

$$\alpha(\varphi) = \bigwedge \{q^A \in Q^A \mid \varphi \Rightarrow \gamma(q^A)\} = \bigwedge \{\exp^A(\vec{b}) \mid \varphi \Rightarrow \exp^A(\vec{\varphi}/\vec{b})\}$$

4 Widening or Abstraction of Abstraction

The basic idea for a practical abstraction function is again given in [P19]. Instead of considering arbitrary expressions on Q^A , only monomials are considered, i.e. conjunctions of b_i and $\neg b_i$, each appearing at most once, and the predicate *false*. The parallel assignment, coupled with the `choose` construct implements this in C2BP.

The set of monomials forms a complete lattice \mathcal{M} that is used instead of the general one. There are two ways of interpreting this simplification:

- as an upper approximation (widening) of the abstraction function [P19]
- as a composition of two abstraction functions [the paper][P4]: one is the boolean abstraction, and the second is the Cartesian abstraction.

Even with this fundamental simplification, it took a great deal of engineering to produce a working system: basically going from $2^{|E|}$ calls to theorem prover to $O(2^{|E|})$, as detailed in [P4].

⁴Note that by further abuse of symbols τ_i defines the binary relation induced by the transition.

⁵This is highly undesirable. Choosing only some predicates is essential for a practical scheme.

5 Comments

The SLAM toolkit is impressive by the sheer number of things it ties together. It looks very good in terms of features:

- it is sound; it rules out false positives. This is what a model checker is supposed to do.
- it handles C reasonably well (interprocedural, recursion etc.), except for pointer analysis and the simplified memory model.
- it is supposed to avoid spurious concrete traces from the abstract trace (using the NEWTON tool)

SLAM is somewhat disappointing when it comes to actual results. Quite a few papers were published, but the only relevant examples it was run on are a couple of publicly available NT drivers.

The implementation is not publicly available, but is “coming soon”. According to the authors’ web page SLAM is being productized.

Comparing model checking with heuristic bug finding is apples to oranges, given that bug finding need not be sound. It is worth pointing that model checking for bug finding is hard to use: one must define have a model (predicates) first, and who checks the model?

The major drawback of SLAM is perhaps BLAST. The three-stage iterative SLAM process (abstraction, verification, counterexample concretization) is integrated into a tightly coupled loop called lazy abstraction [LA], which reuses most of the expensive computation from the first two stages. BLAST 0.1 is the publicly available implementation. On an ancient Pentium II (that I own), it checked most test programs in a matter of seconds.

References

- [LA] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pp. 58–70, 2002