Software Verification with Predicate Abstraction

Marius Minea

Department of Computing, “Politehnica” University of Timişoara
Institute e-Austria Timişoara
marius@cs.utt.ro

Abstract

Predicate abstraction with iterative refinement has recently emerged as a powerful and practical software verification technique. We compare current approaches and outline proposed work.

Abstraction is the centerpoint of scaling verification to large systems, which would otherwise far exceed the capabilities of existing tools. The technique constructs an abstract, i.e., simplified model of a concrete system, while ensuring that certain types of properties are preserved. Verifying the simpler abstract system guarantees correctness without having to analyze the complex original one.

Predicate abstraction is an instance of this technique that is recently developing into a promising approach to practical software verification, having already proved itself on programs of realistic size. The term was first coined by Graf and Saidi [GS97] and now forms the core of verification tools like SLAM [BR00] or BLAST [HJMS02], where its application can be summarized as follows.

The program data is abstracted to a set of boolean variables denoting *predicates* which represent relations between key variables affecting the specification. For example, $size \leq lim$ could be a predicate relevant to checking buffer overflow. The resulting *boolean program* has the same control flow graph as the original one, but with transformed statements. Conditionals are expressed depending only on the chosen predicates, and assignments update the predicate values, as deduced from the changes in the original program variables. This may introduce nondeterminism, since branching and new predicate values cannot generally be determined based on the current values of the chosen predicates alone.

Consequently, verification is done by iterative refinement. Consider checking for reachability of an error state. The abstracted boolean program is model checked; if it is correct, so is the original program, since predicate abstraction is sound (conservative). If an error trace is found, it is executed symbolically in the original program, checking the feasibility at each step with a theorem prover. A feasible trace denotes a real error; otherwise, the unsatisfiability proof indicates additional predicates needed for reasoning. The process is repeated with the expanded predicate set. Although program verification is undecidable, the presented semi-algorithm often terminates in a few iterations.

The key step in this computation is expressing the effects of program statements in terms of the predicates defining the abstraction. A state in the program is defined by program counter and variable values. Since a predicate represents a *set of states* satisfying a relation between variables, the analysis is performed symbolically on such sets (called *regions*). Let $post(r, t) = \{s' \mid \exists s \in r\,.\, s \xrightarrow{t} s'\}$ be the set of states obtained by executing statement t in a state of region r. If α and γ are functions converting from sets of abstract states to sets of concrete states and conversely, the abstract successor $post^a$ can be obtained from the concrete $post^c$ as: $post^a(r, t) = \alpha(post^c(\gamma(r), t))$. Individual approaches differ in the approximations chosen for this computation which is generally not feasible precisely.

The original framework of Graf and Saidi [GS97] considers regions represented in disjunctive normal form over the predicates of the abstraction. For each individual conjunct (monomial) ψ, the abstract successor $post^a(\psi, t)$ is also approximated by a monomial. A theorem prover is used to determine for each predicate p_i whether either p_i or $\neg p_i$ is a postcondition of ψ for statement t, by verifying whether $\psi \Rightarrow wp(p_i, t)$ or $\psi \Rightarrow wp(\neg p_i, t)$, where wp denotes Dijkstra's weakest precondition.

In followup work, Das, Dill and Park [DDP99] avoid approximation with monomials by using binary decision diagrams to efficiently handle the representation of abstract regions and the successor computation. For each predicate p_i, the examined region ϕ is split recursively into fragments from which either p_i or $\neg p_i$ can hold as postconditions, by conjuncting ϕ with the concrete predecessors of p_i and $\neg p_i$: $post_k(\phi, t) = p_k \wedge post_{k+1}(\phi \wedge pre^c(\gamma(p_k), t), t) \vee \neg p_k \wedge post_{k+1}(\phi \wedge \neg pre^c(\gamma(p_k), t), t)$ for $1 \leq k \leq n$, and $post^a(\phi, t) = post_1(\phi, t)$. Here, $pre^c(r, t) = \{s \mid \exists s' \in r \,.\, s \xrightarrow{t} s'\}$. In the last step, $post_{n+1}(\phi, t)$ is true iff ϕ (by now conjuncted with all p_i or $\neg p_i$) is satisfiable, and false otherwise.

In the SLAM project at Microsoft Research, Ball and Rajamani [BR00] develop an approach that is applicable for C programs in an interprocedural setting. Since contexts of procedure calls cannot be predicted, this requires expressing the abstract $post^a$ operator in closed form. To avoid an exponential number of theorem prover calls that would result following [GS97] (where it suffices to compute $post^a$ on demand for each region), a *cartesian* abstraction is performed on top of the boolean one. Each assignment is abstracted to simultaneous updates for each boolean variable separately. After statement t, the boolean variable p_i will be *true* if starting from $wp(\gamma(p_i), t)$, *false* if starting from $wp(\gamma(\neg p_i), t)$, and unknown otherwise. An underapproximation for the abstraction of these two regions is computed as disjunction of implicants which are conjunctions of predicates or their negations. The number of candidate implicants (and thus of theorem prover calls) is reduced by considering only syntactically relevant predicates, and limiting conjunctions to at most 3 predicates, which is sufficient in practice.

The BLAST toolkit from Berkeley [HJMS02] proposes *lazy abstraction*, an approach that performs refinement only when and where necessary. The forward exploration phase can use any of the abstraction computations outlined so far. The counterexample refinement calculates weakest preconditions backwards from the error state and stops on an empty intersection with the region from the forward phase (denoting an infeasible trace), or at the initial state (for a real error). In the former case, the abstraction is only refined from the point that generated the unsatisfiability proof. This reuses part of the already explored state space and allows abstractions to vary over different parts of the program.

Making predicate abstraction usable for programs realistic both in language features and size has been aided by dataflow analysis techniques adapted for verification. For SLAM, [BMMR01] discusses applying the interprocedural dataflow analysis algorithm of Reps, Horwitz and Sagiv to compute procedure summaries for abstraction; the flow-insensitive alias analysis of Das has been generalized to compute value flow, guiding the selection of initial predicates, and reducing refinement iterations.

Practical results indicate so far that the boolean and cartesian abstraction of SLAM gains through simplicity and efficiency without usually losing in accuracy. Still, a technique that would detect which abstraction to use for various parts of the program would be interesting to investigate. Another issue concerns the contrast between fast predicate discovery in SLAM and lazy abstraction in BLAST. A first step for a better evaluation would to implement an interprocedural analysis for lazy abstraction. Next, a hybrid approach could be imagined that detects different predicate sets for various parts of the program, and a mechanism to reuse abstraction results through incremental program changes.

The commonality of predicate abstraction approaches makes a uniform implementation framework desirable. We propose to design a tool with a core implementation of fundamental notions like regions, *pre/post* computation, abstraction, etc. and with a language for expressing various program analyses. This would create the base for prototyping and evaluating extensions like the ones proposed above, as well as new analysis algorithms.

References

[BMMR01] T. Ball, R. Majumdar, T. Milstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM Conf. on Prog. Lang. Design and Implem., PLDI 2001*, ACM Press.

[BR00] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proc. 7th Int. SPIN Workshop on Software Verification*, LNCS vol. 1885, pp. 113–130. Springer Verlag, 2000.

[DDP99] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *Proc. 11^{th} Int. Conf. on Computer Aided Verification, CAV'99*, LNCS vol. 1633, pp. 160–171. Springer Verlag, 1999.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Proc. 9^{th} Int. Conf. on Computer Aided Verification, CAV'97*, LNCS vol. 1254, pp. 72–83. Springer Verlag, 1997.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. 29^{th} ACM Symposium on Principles of Programming Languages, POPL 2002*, pp. 58–70. ACM Press.