Applications of predicate abstraction to software analysis

Wilmot Li (wilmotli@cs)

February 25, 2002

1 Introduction

Given the increasingly significant roles of software systems in the world today, the need for reliable, automatic verification techniques continues to grow. Unfortunately, concrete systems (i.e. specifications or actual programs) generally posess characteristics that are problematic for many types of analysis, making program verification a particularly difficult task. For example, many concrete systems cannot be model checked to find or verify invariants because they correspond to infinite state machines. As a consequence, one of the main challenges in this area of research is coming up with expressive models for software that are amenable to analysis.

Program abstraction is one common approach to this problem. By converting a complicated concrete system into a simpler representation, we can often analyze the abstract model to determine properties of the original specification or program. In particular, *predicate abstraction* has recently emerged as a promising new technique; the high-level idea is to map a concrete system to a finite state boolean representation defined by a set of "interesting" predicates over program variables. The hypothesis is that this model is expressive enough to represent a wide range of important program properties and can in general be analyzed more effectively than the original specification or program. Another implicit assumption at the foundation of predicate abstraction is that concrete systems can be easily mapped to boolean representations. Recent results seem to support these claims, suggesting the promise of predicate abstraction for performing software analysis.

Since boolean representations are relatively new, much of the work in this area so far has been directed towards the basic mechanisms of predicate abstraction.

Thus, the bulk of this paper (Section 2) provides an overview of the abstraction process as formalized in Graf/Saïdi's pioneering work [7], as well as refinements and extensions of the original abstraction procedure. We then survey the different ways in which these boolean models have been analyzed to tackle a variety of example problems (Section 3). As a relatively new and very active area of research, there remain several open questions about predicate abstraction that are discussed in Section 4. Finally, we draw some conclusions about this approach and attempt to evaluate its potential in the realm of software analysis.

2 Predicate abstraction

In general, a program abstraction is only useful for proving things about the original specification or program if it preserves the concrete system's properties. Using the definitions provided in [9], an abstraction is *weakly preserving* if every property that is true in the abstract model is also true in the original system, whereas a *strongly preserving* abstraction retains exactly the same set of properties. A related notion is that of *over* and *under approximation*. If the abstraction's behaviours are a superset of the original system's behaviours, then the abstract model is over approximating, whereas when the converse is true, the abstraction is under approximating. Thus, over approximating abstractions are weakly preserving.

The objective of predicate abstraction is to convert a given concrete system into a property preserving boolean representation whose reachable abstract states S_A correspond to an approximation of the original system's reachable concrete states S_C. In some cases, it is possible to achieve a correspondence that results in a strongly preserving abstraction. However, S_A often represents an over approximation of the concrete system's behaviours, making the boolean model weakly preserving. In either case, once the specification or program has been abstracted we are able to analyze the *abstract* system to prove meaningful properties of the *concrete* system. More specifically, if we can show that some invariant holds over S_A, we are guaranteed via property preservation that this invariant also holds over S_C.

2.1 Concrete and boolean systems

Before diving into the details of the predicate abstraction procedure, (i.e. how to compute S_A) we first define some terms and notation (borrowed primarily from [5]) that will be useful later in this discussion. In general, we can think of a concrete system as a set of commands or statements with rules that determine allowable

transitions between them. The system's state at any point during execution can be represented as a record whose fields indicate the current value of each concrete variable. For example, for a program that has only two variables, x and y, we might represent its concrete state as $q_C = \{x = 5, y = 2\}$. Notice that the domain of any individual variable might be infinite (e.g. a variable of integer type). Furthermore, with dynamically allocated memory, the number of program variables may also be infinite. As a consequence, the concrete state space is potentially unbounded, a prohibitive property for many types of analyses, especially those with a model checking flavour that search the state space exhaustively.

Predicate abstraction, as introduced in [7], proposes a transformation of this concrete system using a set of predicates $\Phi = \{\phi_1, \ldots \phi_N\}$, defined over program variables. The choice of these predicates is a significant issue that is discussed later in the paper. For now, however, let us assume that they are manually selected. Given a concrete system and Φ, the idea is to create an abstract system with a set of N boolean variables $B = \{b_1, \ldots, b_N\}$ that correspond to the predicates in Φ. Thus, the abstract system's state at any point during execution is a truth assignment to B that represents the current values of the N predicates. For example, if x and y are concrete variables, with $\Phi = \{(x == y), (x > 0)\}$, then the abstract state might be $q_A = \{b_1 = true, b_2 = false\}$, if x and y are both equal to the same negative number. Note that, unlike the concrete system, the abstract state space is finite (exponential in N).

Often, it is useful to talk about sets of (rather than individual) concrete or abstract states. A collection of abstract states can be represented conveniently as a boolean formula on B, and a group of concrete states can be written as a first-order formula over predicates in Φ. Notice that a set of concrete states thus defined can be infinite in size, a fact that is consistent with our knowledge that concrete systems are potentially unbounded. This notation will be useful in describing the actual process of predicate abstraction.

2.2 The abstraction procedure

The crux of predicate abstraction is the computation of S_A given S_C and Φ. There are two basic ways to go about doing this. We can either statically abstract all concrete transitions to generate the abstract state system, or we can dynamically construct an abstract state graph by repeatedly applying the concrete transitions to the abstract states. Since the second approach can eliminate unnecessary abstract states after executing each transition, it has the potential to generate a more precise approximation of S_C, at the cost of a much more intensive computation. Since the

quality of the resulting abstraction has an impact not only on the speed of analysis, but also potentially what we can succesfully prove, it seems that most published predicate abstraction results perform the more expensive and more precise dynamic computation of the abstract state graph. Furthermore, the dynamic approach seems more appropriate for abstracting programs written in imperative languages such as C or Java. In this context, executing concrete transitions roughly corresponds to updating the current approximation of S_A after each program statement. [1]

The key operation at the core of both the static and dynamic abstraction algorithms is the abstraction function α that maps a set of concrete states to a set of abstract states. Abstracting a *single* concrete state q_C simply involves evaluating each predicate in Φ given the variable values at q_C to determine a truth assignment for the N boolean variables in B. As explained in Section 2.1, this assignment corresponds to an abstract state q_A. However, abstracting a *set* of concrete states, represented by the first-order formula Q, is slightly more involved. We can define the answer we're looking for in terms of the concretization function γ. If f is a boolean formula over B representing a set of abstract states, then

$$\gamma(f) = f(b_1 \leftarrow \phi_1, \ldots, b_N \leftarrow \phi_N) \tag{1}$$

In other words, γ maps a set of abstract states to the set of concrete states represented by the first-order formula on the righthand side of Equation 1 by replacing each boolean variable in B with its corresponding predicate in Φ. Given this definition, [6] describes $\alpha(Q)$ as the strongest boolean formula on B that satisifies

$$Q \Rightarrow \gamma(\alpha(Q)) \tag{2}$$

2.3 Predicate abstraction results

Although the abstraction process is relatively easy to describe, developing practical predicate abstraction algorithms for real concrete systems involves a number of challenges, including efficient implementation of α and handling source language features such as pointers and procedures in C, and dynamically allocated memory in object-oriented systems. This section surveys recent work that addresses these issues.

[1] Clearly, many details have been omitted here, but a full explanation of these algorithms is beyond the scope of this paper.

2.3.1 Efficient abstraction

As mentioned previously, the key step in the abstraction procedure is the computation of $\alpha(Q)$. Implementing α efficiently is one of the main challenges in designing a practical abstraction algorithm. To understand why this is a difficult problem, consider the most naive approach. To come up with a boolean formula f that satisfies Equation 2, we could enumerate every possible formula over B, and just test each one. As long as B contains a finite number of variables, this algorithm will terminate; unfortunately, it requires approximately 2^N calls to a theorem prover in order to verify every formula.

Efforts to develop more efficient algorithms have concentrated on quickly finding "smaller" boolean formulas (i.e. ones that contain fewer literals) that satisfy Equation 2. Although we do not describe any implementations in detail, it is worth noting that [5], [8], [2] and most recently [6] have demonstrated abstraction algorithms that perform significantly better in practice than the naive exponential time approach outlined above.

2.3.2 From specifications to real programs

One trend in the evolution of predicate abstraction algorithms is the natural shift from concrete systems described in specification-like languages to those written in real programming languages. Early work such as [7] and [5] demonstrated predicate abstraction algorithms on programs written in relatively simple and limited languages, often without features like pointers or procedures. However, recent work by the SLAM group at Microsoft Research has taken significant steps towards implementing predicate abstraction for systems specified in more widely-used languages.

In particular, `C2BP` is a tool developed by SLAM that (given a set of interesting predicates) automatically computes a predicate abstraction for a C program, converting it into a *boolean program*, which has the same control flow as the original but contains only boolean variables [2]. The basic approach is to process the original code line-by-line, updating the current approximation of the reachable abstract state space and generating appropriate boolean program statements. Since the abstracted program can only contain boolean variables that correspond to the specified predicates, some special language features are added to the abstract representation, including a non-deterministic control expression (*) for modelling conditional blocks and an `unknown()` value that is assigned to a boolean variable when the effect of a C program statement on a predicate cannot be determined.

The primary contribution of this work is a set of general techniques for handling language features such as, pointers and procedures when performing predicate abstraction. Due to aliasing it is often difficult to determine the effect of a C program statement on any predicate that contains pointers or pointer dereferences, causing potentially severe degradation of precision in the boolean program. C2BP tries to mitigate this problem through aliasing information gleaned via a flow-insensitive points-to analysis. In previous predicate abstraction work, procedures were handled via inlining, essentially eliminating any modularity in the original system and precluding the abstraction of recursive programs. C2BP supports modular predicate abstraction (and thus, recursion) by conservatively estimating a called procedure's effect on predicates without resorting to inlining. This is done by providing the caller with *signatures* of any function it calls that specify predicates over the callee procedure's formal parameters and return values. Although C2BP works exclusively on C programs, the authors point out that their techniques for handling pointers and procedures are sufficiently general to be adopted for the abstraction of programs "written in other imperative languages such as Java."

Along these lines, [9] presents techniques for abstracting object-oriented languages such as C++ or Java. In particular, one of its primary contributions is *dynamic predicate abstraction*, a way to handle predicates that relate dynamic data (such as data in the fields of class instances in C++). Their idea is to annotate predicates with dynamic information that becomes available at runtime. Given the success of this work and C2BP, it seems as if the push towards predicate abstraction algorithms for popular languages is gaining momentum.

3 Analyzing boolean programs

Thus far, we have yet to discuss how boolean representations have been analyzed to prove program properties. Since predicate abstractions are finite state systems, model checking and other state space exploration algorithms will terminate when applied to boolean models. As a result, model checking has been a popular approach when it comes to analyzing predicate abstractions [3, 2, 9, 7, 4, 8]. Most of these analyses work by exploring the abstract state space and checking to make sure a particular property holds in all reachable states, although some papers couch this process in different terms. For example, [1] formulates the problem in terms of Context-Free-Language (CFL) reachability. One notable exception is a recent paper by Flanagan/Qadeer [6] that shows how boolean models can be used to automatically determine loop invariants. The basic idea in this work is to first abstract

the original program and then, whenever a loop is encountered, to iteratively update a candidate invariant (based on the specified predicates) until fixpoint is reached.

Although the goal of this section is not to describe all the different example programs on which predicate abstraction has been tested, it is worth mentioning a few notable results to indicate the range of problems this approach can potentially address. In [2], a number of toy programs are used to show how invariants computed via predicate abstraction can be used to check for NULL pointer dererences, array bounds violations, aliasing information and heap properties [2]. Small list partition and selection sort examples were also used in [6] to demonstrate the automatic computation of loop invariants via boolean programs. There have also been a few successful results using larger systems. [2] applied predicate abstraction to check the saftey properties of 5 NT device drivers that ranged in size from 236 to 6500 lines of code. [6] and [9] also report successful tests on larger systems. These results suggest that predicate abstraction can be used on systems with up to 50K lines of code; however, the scalability of this approach beyond that remains an interesting issue that is discussed in more detail in the next section.

One final result that is worth discussing is the use of predicate abstraction demonstrated in [5]. Here, the authors abstract and analyze the FLASH cache protocol in order to strengthen previously identified invariants. This work is significant because it suggests the application of boolean models as a component within a more extensive verification system, rather than using predicate abstraction in isolation to model check a specification or program.

4 Open problems

As mentioned earlier, predicate abstraction is a relatively recent idea, and as a consequence, there are still many open issues regarding this approach. The most general question involves the range of problems that are "suitable" for predicate abstraction. In other words, what should this technique be used for? Section 3 presented results that suggest the promise of predicate abstraction for verifying certain kinds of program properties. However, there does not seem to be any kind of consensus yet on what class of properties are most naturally modelled using predicates.

In some sense, it can be argued that practically *any* program property can be represented (although not necessarily very well) using predicates. After all, most invariants can somehow be expressed as a predicate over program variables. However, to illustrate how predicate abstraction may not be the right approach to take in some

situations, consider the following example. Suppose we need to perform constant propagation, and instead of doing a data flow analysis, we want to use predicate abstraction. For every constant that some program variable x could possible take on, we *could* generate a predicate and then check to see which predicates hold at various program points in order to propagate constants. Clearly, this is not a good application for predicate analysis. However, beyond this very obvious case, is there something general we can say about when predicate abstraction is (and is not) appropriate?

One part of the answer may involve the issue of predicate selection raised earlier. In the constant propagation example, one particularly objectionable component of the predicate abstraction approach is figuring out all the possible values for x so that we can specify the right predicates. Although there has been some work in automating the selection of interesting predicates (i.e. generating Φ) for loop invariant computation via heuristics [6], it is unclear exactly how far such efforts will advance, and since the definition of Φ is so crucial to the boolean model's expressiveness, this seems like an extremely important issue. As a result, one factor in determining the class of suitable problems for predicate abstraction may simply be how much user input and effort is necessary to generate Φ for various programs and properties.

Another limitation may be the expressiveness of boolean representations. Although recent work has extended the expressive power of predicate abstractions to handle pointers, multiple procedures [2], dynamic data [9], and unbounded data structures [6], these approaches are not entirely general. Since predicates are not particularly good at modelling memory, it may turn out that some heap properties are very difficult to verify using predicate abstraction. In addition, [2] points out some difficulties in extending boolean representations to for modelling multi-threaded programs.

Finally, there is the question of scale. Although boolean models definitely correspond to finite state systems, state space explosion can still occur as the number of predicates in Φ increases. Up until this point, nobody has attempted to run predicate abstraction on really massive pieces of software, and it seems likely that a significant amount of precision would have to be compromised in order to represent a very large concrete system with a reasonably-sized boolean model (i.e. one that can still be model checked in a reasonable amount of time). Furthermore, as the complexity of the call graph increases, it would be interesting to see how dramatically the precision of the abstract system degrades. Although [2] does make an effort to address the abstraction of multi-procedure programs, there is still some loss of information at call sites.

5 Conclusions

Predicate abstraction is a new program abstraction technique that represents concrete systems as finite state boolean models that are generally more amenable to analysis than the original programs. At the beginning of this paper, we identified two main hypotheses that motivate this line of research. The successful application of predicate abstraction to a wide range of source programs and target properties suggests that boolean representations are at least expressive enough to model a useful (if not complete) set of language features and invariants. Furthermore, the application of various analysis techniques to abstract boolean systems supports the claim that predicate abstractions are convenient (or at least manageable) for the purpose of analysis.

However, whether or not concrete systems can be "easily" mapped to boolean models is a bit less clear. As mentioned previously, the primary difficulty involves the selection of appropriate predicates. Since Φ is so integral to the resulting abstraction's usefulness, it seems important that efforts such as the one presented in [6] and SLAM's current `NEWTON` project succeed in automating (or at least, significantly aiding) the process. As long as predicate selection does not become a prohibitively tedious manual task, predicate abstraction seems to have potential as a technique that can be incoroporated into practical verification and analysis tools.

Overall, predicate abstraction definitely seems worth pursuing. Hopefully, the flurry of recent work in the area is an indicator that the software engineering communicty shares this feeling.

Acknowledgements

Many thanks to the PL posse (Todd Millstein, Sorin Lerner and Mark Seigle) for numerous enlightening discussions about predicate abstraction and model checking, as well as comments and suggestions on earlier drafts of this paper.

References

[1] T. Bal and S. Rajamani. Boolean programs : A model and process for software analysis. Technical report, Microsoft Research, 2000.

[2] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.

[3] T. Ball, A. Podelski, and S. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *TACAS*, pages 268–283, 2001.

[4] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.

[5] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171, 1999.

[6] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, 2002.

[7] 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.

[8] S. Saidi and N. Shankar. Abstract and model check while you prove. In *Computer Aided Verification*, pages 443–454, 1999.

[9] W. Visser, S. Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *FMSP*, pages 3–12, 2000.