Combining Predicate and Numeric Abstraction for Software Model Checking (Extended Abstract)[⋆]

Arie Gurfinkel and Sagar Chaki

Software Engineering Institute, Carnegie Mellon University

Abstract

Predicate abstraction [6] (PA) and Abstract Interpretation [4] (AI) with numeric abstract domains, called Numeric abstraction (NA), are two mainstream techniques for automatic program verification. Although it is sometimes assumed that the difference between the two is that of precision versus efficiency, experience of projects based on PA (such as SLAM [1]) and those based on NA (such as ASTRÉE [3]) indicate that both techniques can balance efficiency and precision when applied to problems in a particular domain. However, the two techniques have complimentary strengths and weaknesses.

Predicate abstraction reduces program verification to propositional reasoning via an automated decision procedure, and then uses a model checker for analysis. This makes PA well-suited for verifying programs and properties that are control driven and (mostly) data-independent. An example of such a program is the code fragment in Fig. 1(a). However, in the worst case, reduction to propositional reasoning is exponential in the number of predicates. Hence, PA is not as effective for data-driven and (mostly) control-independent programs and properties, such as the code fragment shown in Fig. 1(b) In summary, PA is works best for propositional reasoning, and performs poorly for arithmetic.

On the other hand, Numeric abstraction restricts all reasoning to conjunction of linear constraints. For instance, NA with Intervals is limited to conjunctions of inequalities of the form $c_1 \leq x \leq c_2$, where x is a variable and c_1, and c_2 are constants. Instead of relying on a general-purpose decision procedure, NA leverages a special data structure – Numeric Abstract Domain. The data structure is designed to represent and manipulate sets of numeric constraints efficiently; and provides algorithms to encode statements as transformers of numeric constraints. Thus, in contrast to PA, NA is appropriate for verifying properties that are (mostly) control-independent, but require arithmetic reasoning. One example of such a program is the code fragment in Fig. 1(b). On the flip side, NA performs poorly when propositional reasoning (i.e., precisely representing disjunctions and negations) is required. For example, the code fragment in Fig. 1(a) is hard for NA.

In practice, precise, efficient, and scalable program analysis requires the strengths of both predicate and numeric abstraction. Consider the problem of verifying the code fragment in Fig. 1(c). In this case, propositional reasoning is needed to distinguish

[⋆] An article reporting on this research is currently under submission to a conference.

```
assume(i==1 || i==2);
switch(i)
 case 1: a1=3; break;
 case 2: a2=-4; break;
switch (i)
 case 1: assert(a1>0);
 case 2: assert(a2<0);
 default: assert(0);
```

(a)

```
if(3 <= y1 <= 4)
  x1 = y1 - 2;
  x2 = y2 + 2;
else if(3 <= y2 <= 4)
  x1 = y2 - 2;
  x2 = y2 + 2;
assert(5 <= (x1+x2) <= 10);
```

(b)

```
assume(x1==x2);
if (A[y1 + y2] == 3)
  x1 = y1 - 2;
  x2 = y2 + 2;
else
  A[x1 + x2] = 5;
if (A [x1 + x2] == 3)
  x1 = x1 + x2;
  x2 = x2 + y1;
assert(x1==x2);
```

(c)

```
$\mathit{assume}(x_1 = x_2)$;
$((\mathit{assume}(p)$;
  $x_1 := y_1 - 2 \wedge q := \mathit{choice}(\mathsf{f}, \mathsf{f})$;
  $x_2 := y_2 + 2 \wedge q := \mathit{choice}(x_1 + 2 = y_1 \wedge p, \mathsf{f})) \vee$
$(\mathit{assume}(\neg p)$;
  $q := \mathit{choice}(\mathsf{f}, \mathsf{t})))$;
$((\mathit{assume}(q)$;
  $x_1 := x_1 + x_2$;
  $x_2 := x_2 + y_1) \vee \mathit{assume}(\neg q))$;
$\mathit{assert}(x_1 = x_2)$
```

(d)

Fig. 1. Example programs (a), (b), (c). Part (d) is an abstraction of (c) with $V_P = \{p, q\}$, $V_N = \{x_1, x_2, y_1, y_2\}$, where $p \triangleq ((A[y_1 + y_2] = 3)$, and $q \triangleq (A[x_1 + x_2] = 3)$.

between different program paths, and arithmetic reasoning is needed to efficiently compute strong enough invariant to discharge the assertion. More importantly, the propositional and numeric reasoning must interact in non-trivial ways. Therefore, a combination of PA and NA is more powerful and efficient than either technique alone.

Any meaningful combination of PA and NA must have at least two features: (a) propositional predicates are interpreted as numeric constraints where appropriate, and (b) abstract transfer functions respect the numeric nature of predicates. The first requirement means that, unlike most AI-based combinations, the combined abstract domain cannot treat predicates as uninterpreted Boolean variables. The second requirement implies that the combination must support abstract transformers that allow the numeric information to affect the update of the predicate information, and vice versa.

Against this background we make the following contributions. We present the interface of an abstract domain, called NUMPREDDOM, that combines both PA and NA, and supports a rich set of abstract transfer functions that enables the updates of numeric and predicate state information to be influenced by each other. For example, an NUMPREDDOM-based abstraction of the code fragment in Fig. 1(c) is shown in Fig. 1(d). Here, two predicates are used to relate reasoning about conditions in control-flow statements with reasoning about numeric variables. Note that the value of the predicates depends on values of numeric constraints.

We propose four data-structures — NEXPoint, NEX, MTNDD and NDD — that implement NUMPREDDOM. The data structures (summarized in Table 1) differ in their expressiveness and in the choice of representation for the numeric part of the domain.

Name	**Value**	**Example**	**Num.**
NEXPoint	$2^{2^P} \times N$	$(p \vee q) \wedge (0 \leq x \leq 5)$	EXP
NEX	$2^P \mapsto N$	$(p \wedge 0 \leq x \leq 3) \vee (q \wedge 1 \leq x \leq 5)$	EXP
MTNDD	$2^P \mapsto N$	$(p \wedge 0 \leq x \leq 3) \vee (q \wedge 1 \leq x \leq 5)$	SYM
NDD	$2^P \mapsto 2^N$	$(p \wedge (x = 0 \vee x = 3) \vee (q \wedge (x = 1 \vee x = 5)))$	SYM

Table 1. Summary of implementations of NumPredDom; P = predicates; N = numerical abstract values; **Value** = type of an abstract element; **Example** = example of allowed abstract value; **Num** = numeric part representation (explicit or symbolic).

Our target is PA-based software analysis. Thus, all of the data-structures use BDDs for efficient (symbolic) propositional reasoning.

Related work. A typical way to combine PA and NA in AI is to use a direct, or reduced [4] product, possibly extending it with disjunctions (or unions) using a disjunctive completion [4]. The domains we develop in this paper are variants of (disjunctive completion of) reduced product between PA and NA. One practical approach for combining these domains is to combine results of the analyses [7], e.g., by using lightweight data-flow analyses, such as alias analysis and constant propagation, to simplify a program prior to applying predicate abstraction. Thus, the invariants discovered by one analysis are assumed by the other. Another approach is to run the analyses over different abstract domains in parallel within a single analysis framework, using the abstract transfer functions of each domain as is [5, 2]. The analyses may influence each other, but only through conditionals of the program.

The contribution of our work is in adapting, extending, and evaluating existing work on combining propositional and arithmetic reasoning to the needs of software model-checking. We have implemented a general framework for reachability analysis of C programs on top of our four data structures. Our experiments on non-trivial examples show that our proposed combination of PA and NA is more powerful and more efficient than either technique alone. Finally, by coupling PA and NA tightly, our approach opens up new research directions toward automated abstraction refinement techniques that are more efficient that existing solutions.

References

1. T. Ball and S.K. Rajamani. “Automatically Validating Temporal Safety Properties of Interfaces”. In *Proc. of SPIN*, 2001.
2. Dirk Beyer, Thomas A. Henzienger, and Gregory Theoduloz. ‘Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis”‘. In *CAV*, 2007.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. “A Static Analyzer for Large Safety-Critical Software”. In *PLDI*, 2003.
4. P. Cousot and R. Cousot. “Abstract Interpretation Frameworks”. *JLC*, (4), 1992.
5. Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. “Joining dataflow with predicates”. In *FSE*, 2005.
6. S. Graf and H. Saïdi. “Construction of Abstract State Graphs with PVS”. In *CAV*, 1997.
7. Himanshu Jain, Franjo Ivancic, Aarti Gupta, Ilya Shlyakhter, and Chao Wang. “Using Statically Computed Invariants Inside the Predicate Abstraction and Refinement Loop”. In *CAV*, 2006.