Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Combining Predicate and Numeric Abstraction for Software Model Checking

Arie Gurfinkel and Sagar Chaki

Software Engineering Institute, Carnegie Mellon University

Received: date / Revised version: date

Abstract. Predicate (PA) and Numeric (NA) abstractions are the two principal techniques for software analysis. In this paper, we develop an approach to couple the two techniques tightly into a unified framework via a single abstract domain called NUMPREDDOM. In particular, we develop and evaluate four data structures that implement NUMPREDDOM but differ in their expressivity and internal representation and algorithms. All our data structures combine BDDs (for efficient propositional reasoning) with data structures for representing numerical constraints. Our technique is distinguished by its support for complex transfer functions that allow two way interaction between predicate and numeric information during state transformation. 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.

1 Introduction

Predicate abstraction (PA) [3] and Abstract Interpretation (AI) with numeric abstract domains, called Numeric abstraction (NA) [6], 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 [3]) and those based on NA (such as ASTRÉE [6]) indicates that both techniques can balance efficiency and precision when applied to problems in a particular domain. These two techniques have complementary strengths and weaknesses. A combination of PA and NA is more powerful and efficient than either technique alone. Achieving an effective combination of PA and NA is the subject of our paper.

Send offprint requests to:

Predicate abstraction uses an automated decision procedure (ADP) to reduce program verification to propositional reasoning with a model checker. This makes PA well-suited for verifying programs and properties that are control driven and (mostly) data-independent. For example, PA is well suited for verifying the code fragment in Fig. 1(a). However, in the worst case, the 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 works best for propositional reasoning, and performs poorly for arithmetic.

Numeric abstraction reduces program verification to reasoning about conjunction of linear constraints. For instance, NA with the *Intervals* domain is limited to conjunctions of inequalities of the form $c_1 \leq x \leq c_2$, where x is a variable and c_1, c_2 are numeric constants. Instead of relying on a general-purpose ADP, NA leverages a Numeric Abstract Domain — a collection of special data-structures and algorithms designed to represent and manipulate sets of numeric constraints efficiently, and 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. For example, NA is well suited for verifying the code fragment in Fig. 1(b). On the flip side, NA performs poorly when propositional reasoning (i.e., supporting disjunctions and negations) is required. For example, NA is not well suited for verifying the code fragment in Fig. 1(a).

In practice, precise, efficient and scalable program analysis requires the strengths of both predicate and numeric abstraction. For instance, in order to verify the code fragment in Fig. 2(a), propositional reasoning is needed to distinguish between different program paths, and arithmetic reasoning is needed to efficiently compute an invariant strong enough to discharge the assertion. More importantly, in this example the propositional and numeric reasoning must interact in non-trivial ways.

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

(a)

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

(b)

Fig. 1. Two example programs.

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. First, we present the interface of an abstract domain, called NUMPREDDOM, that combines both PA and NA. The interface is distinguished by very rich syntax for abstract transformers that tightly combines updated to predicate and numeric parts of the abstract state. This allows predicate and numeric state information to influence each other.

Second, 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. All of the data-structures support very efficient (symbolic) propositional reasoning. Thus, they are well suited for our target application – PA-based program analysis.

Third, we present experimental results on non-trivial examples, and compare and contrast between pure predicate abstraction, pure numeric abstraction, and our four data-structures. Our experiments show that the proposed combination is more powerful and more efficient than either PA or NA in isolation and that our four implementations of NUMPREDDOM exhibit meaningful tradeoffs between expressiveness and efficiency of various operations.

The rest of this paper is structured as follows. We survey related work in Section 2 and review background material in Section 3. In Section 4, we present the interface of NUMPREDDOM. In Section 5, we describe the particularities of each of our NUMPREDDOM implementations. Finally, experimental results and conclusions are presented in Section 6.

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

Table 1. Summary of implementations of NUMPREDDOM; V_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).

2 Related Work

There are several approaches to combine propositional and numeric reasoning in a program verifier. These include explicitly combining analysis engines, devising new abstract domains, designing new data structures to allow greater interaction between existing domains, and delegating all reasoning to a decision procedure for a fragment of arithmetic.

Numeric and Predicate Abstraction. The problem of combining PA and NA involves combining their abstract domains, and is well studied in Abstract Interpretation [11]. Typically, abstract domains are combined using a *domain product*, e.g., direct, reduced [10,11], or logical [15]. Furthermore, a *disjunctive completion* [11] is used to extend a domain with disjunctions (or unions). The domains we develop in this paper are variants of a (disjunctive completion of) reduced product between domains of PA and NA. In practice, our combination of PA and NA achieves a form of automated value-based *trace partitioning* [20].

One approach for combining abstract domains is to combine results of the analyses – e.g., by using light-weight dataflow 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. For instance, Jain et al.[18] present a technique to compute numeric invariants using NA which are then used to simplify PA. However, this approach only works when the verification task can be cleanly partitioned into arithmetic and propositional reasoning. For example, it is ineffective for verifying the program in Fig. 2(a), where

purely numeric reasoning is too imprecise to produce any useful invariants.

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. The analyses may influence each other, but only through conditionals of the program. This approach is often taken by large-scale abstract interpreters [6], that use different abstract domains to abstract distinct program variables. Recently, a similar approach has been incorporated into software model checker BLAST [12,4,5] to combine predicate abstraction with various data-flow analyses. In principle, this can be adapted to combining PA and NA. The expressiveness of this combination is comparable to NEXPOINT – our simplest combined domain.

From the approaches that tightly combine predicate and numeric abstractions the work of Bultan et al. [8] is closest to ours. They present a model checking algorithm to reason about systems whose transition relation combines propositional and numeric constraints. Their algorithms are based on a data structure that uses BDDs [7] for propositional reasoning and the Omega library[1] for arithmetic reasoning. While this data structure is similar to NEX, we support more complicated transfer functions and provide an interface to replace the Omega library with an arbitrary numeric abstract domain.

Our domains MTNDD and NDD use BDDs for a purely symbolic representation of abstract values. Thus, they are similar to Difference Decision Diagrams (DDDs) [22] that represent propositional formulas over difference constraints. However, unlike DDD, we do not restrict the domain of numerical constraints. This makes our implementation more general, at the cost of strong canonicity properties of DDDs.

The contribution of our work is in adapting, extending, and evaluating existing work on combining propositional and arithmetic reasoning about programs to the needs of software model checking. To our knowledge, none of the tight combinations of the two abstract domains have been evaluated in the context of PA-based software model checking.

Satisfiability Modulo Theory (SMT). The SMT-problem is the problem of deciding satisfiability of a first order (typically, quantifier free) formula whose atomic terms are interpreted in one or more theories. An SMT-solver is a tool that solves the SMT problem. Current state-of-the-art SMT-solvers can reason about combined theories of propositional logic, uninterpreted functions, and linear arithmetic.

An SMT-solver is often the main theory-aware reasoning engine in a program verifier. For example, it is the main engine for predicate abstraction [2,19,9], or, when combined with interpolation, it can be used to implement predicate transformers (e.g., [21]). In our approach, we use an SMT-solver for computing predicate abstraction part of an abstract transformer and for the refinement step of the Counter-Example Guided Abstraction-Refinment (CEGAR) loop.

[1] `http://www.cs.umd.edu/projects/omega`

There are many similarities and differences between combining theories in an SMT-solver and combining numeric and propositional reasoning in an abstract domain (as we do here). In the rest of the section, we highlight some of the key differences:

- **Data-structures for Boolean formulas over combined theories:** Both abstract domains and SMT-solvers use data-structures to represent Boolean formulas. However, they differ in the requirements they impose on those data-structures. A data-structure for an abstract domain must support efficient application of transfer functions and application of widening. This is not a requirement for a data-structure in an SMT-solver.
  The data-structures we present in this article are based on BDDs. This provides us with a DNF representation of an abstract value that is needed for application of abstract transfer functions. In contrast, SMT-solvers use CNF-based data-structure for Boolean formulas.
- **Precision versus efficiency trade-offs:** In an abstract domain, every abstract operation must *over-approximate* a corresponding concrete one. Thus, the designer of an abstract domain can choose between a more efficient but more approximate implementation and a less efficient but more precise implementation of every operation. This is our main motivation for developing four different combinations of PA and NA – each achieving different trade-off on the precision versus efficiency scale. In contrast, in SMT all operations must be interpreted precisely.
- **Quantifier elimination:** The key steps in an Abstract Interpretation-based program analysis are the computation and application of abstract transformers. In general, these steps are reduced to quantifier elimination (i.e., existentially projecting "previous state" variables in the strongest post-condition computation). Thus, quantifier elimination (or its over-approximation) is an essential operation for any abstract domain. In contrast, quantification is not a standard operation supported by an SMT-solver. Moreover, SMT-solvers are often used to reason about theories that even do not admit quantifier elimination at all (e.g., the theory of uninterpreted functions).
- **Widening:** Another key property of an Abstract Interpretation-based program analysis is that it is always guaranteed to terminate. This is achieved by requiring each abstract domain to have an approximation scheme, called *widening*, that computes a closed form (or a limit) for any increasing chain of abstract values. Widening operation is unique to abstract domains. It is not clear how to define an analogous operation (or why it is even needed) for SMT solvers.

In summary, at a high-level, combining predicate and numeric abstract domains is similar to combining predicate and numeric reasoning in SMT-solving. However, we believe that the requirements and the details of the combination are quite different.

```
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 - 2;
assert(x1==x2);
```

(a)

$$
\begin{array}{l}
\textit{assume}(x_1 = x_2);\\
((\textit{assume}(p);\\
\quad x_1 := y_1 - 2 \wedge q := \textit{choice}(\mathtt{f},\mathtt{f});\\
\quad x_2 := y_2 + 2 \wedge q := \textit{choice}(x_1 + 2 = y_1 \wedge p, \mathtt{f})) \vee\\
(\textit{assume}(\neg p);\\
\quad q := \textit{choice}(\mathtt{f},\mathtt{t})));\\
((\textit{assume}(q);\\
\quad x_1 := x_1 + x_2;\\
\quad x_2 := x_2 + y_1 - 2) \vee \textit{assume}(\neg q));\\
\textit{assert}(x_1 = x_2)
\end{array}
$$

(b)

Fig. 2. A program (a), and its abstraction (b) 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 $q \triangleq (A[x_1 + x_2] = 3)$.

3 Background

In this section, we define our basic notation and our view of abstract domains.

3.1 Expressions and Statements

Let V denote the set of program variables, E denote the set of expressions over V, and $B \subseteq E$ denote Boolean expressions. There are two kinds of *atomic* statements:

1. an assignment, $l := e$, where l is a variable in V and e is an expression in E, and
2. an assumption, *assume*(e), where e is in B.

Assume operations are used to model conditional branches, i.e., if-then-else blocks, as well as assumptions used during verification. We write $||s||$ to denote the collecting semantics, i.e., the strongest post-condition transformer, as a function from B to B.

Example 1. The following are some examples of collecting semantics of atomic statements:

$$
\begin{array}{c}
||x := x + 1||(x > 3) \equiv (x > 4)\\
||x := 5||(x = 3 \wedge y = 6) \equiv (x = 5 \wedge y = 6)\\
||\textit{assume}(x > 4)||(y = 6) \equiv (x > 4 \wedge y = 6)\,.
\end{array}
$$

□

A program is a control-flow graph annotated by *loop-free* statements S. The set S is constructed by composing atomic statements as follows:

1. sequentially, written $s_1; s_2$, meaning execution of s_1 is followed by the execution of s_2;
2. non-deterministically, written $s_1 \vee s_2$, meaning a non-deterministic choice between execution of s_1 and s_2.

3.2 Abstract Domain

We assume that the reader is familiar with abstract interpretation. For a detailed overview of AI, we refer the reader to the seminal work by Cousot and Cousot [11]. Often, an abstract and concrete domain are viewed as lattices connected by a Galois connection. In this article, we take a more operational view of an abstract domain as an *abstract data type* that satisfies the interface $\textsc{AbsDom}(V)$ shown in Fig. 3. Such a view of an abstract domain is sufficient for our purpose. We assume that the concrete domain is the set of expressions B, and not, for example, program states. We use A to denote the set of all the *abstract elements* of $\textsc{AbsDom}(V)$. The interface $\textsc{AbsDom}(V)$ consists of the following functions:

1. abstraction, α, and concretization, γ, that convert between expressions and abstract elements in A;
2. meet and join that approximate conjunction (intersection) and disjunction (union), respectively;
3. leq that approximates implication (subset);
4. isTop and isBot check for validity (universality), and unsatisfiability (emptiness), respectively;
5. widen is a widening operator [11] that over-approximates a disjunction and guarantees convergence when applied to any (possibly infinite) sequence of abstract elements; and
6. αPost approximates the semantics of a program statement as an *abstract transformer*, i.e., a function from A to A.

The set of requirements at the bottom part of Fig. 3 ensure that the abstract domain is a sound approximation of the concrete one. For example, the first rule ensures that for any expression e, abstraction (α) of e, followed by concretization (γ) of the result is weaker (i.e., bigger, or less precise) than e.

Table 2 shows several commonly used abstract domains. The first three domains, collectively called Numeric, are used

Name	Notation	Abstract Elements
Intervals	$\textsc{Box}(V)$	$\{c_1 \leq v \leq c_2 \mid c_1, c_2 \in \mathcal{N}, v \in V\}$
Octagons	$\textsc{Oct}(V)$	$\{\pm v_1 \pm v_2 \geq c \mid c \in \mathcal{N}, v_1, v_2 \in V\}$
Polyhedra	$\textsc{Pk}(V)$	linear inequalities over V
Predicates	$\textsc{Pred}(V)$	propositional formulas over V

Table 2. Common abstract domains; V is a set of numeric/propositional variables; $\mathcal{N}$ domain of numeric constants.

Interface: $\text{ABSDOM}(V)$

γ	$: A \to B$	α	$: B \to A$
meet	$: A \times A \to A$	join	$: A \times A \to A$
isTop	$: A \to \textbf{bool}$	isBot	$: A \to \textbf{bool}$
leq	$: A \times A \to \textbf{bool}$	widen	$: A \times A \to A$
	$\alpha\mathsf{Post} : S \to (A \to A)$		

Requires:

let $a, b, c \in A, e \in B, x = \gamma(a), y = \gamma(b), z = \gamma(c)$ **in**

$e \Rightarrow \gamma(\alpha(e))$	$(\alpha\mathsf{Post}(s)(a) = b) \Rightarrow (\|\|s\|\|(x) \Rightarrow y)$
$\mathsf{leq}(a, b) \Rightarrow (x \Rightarrow y)$	$(\mathsf{meet}(a, b) = c) \Rightarrow (x \wedge y \Rightarrow z)$
$\mathsf{isTop}(a) \Rightarrow (x)$	$(\mathsf{join}(a, b) = c) \Rightarrow (x \vee y \Rightarrow z)$
$\mathsf{isBot}(a) \Rightarrow (\neg x)$	$(\mathsf{widen}(a, b) = c) \Rightarrow (x \vee y \Rightarrow z)$

Fig. 3. Interface of an abstract domain: B denotes Boolean expressions, S denotes statements, and A denotes abstract values.

to represent and manipulate arithmetic constraints. The last one, $\text{PRED}(V)$, represents propositional formulas over a set of predicates.

3.3 Abstract Transformers

For ease of presentation, we define a syntax for abstract transformers. Let $\text{NDOM}(V_N)$ be a numeric domain over variables V_N. The numeric domain has two abstract transformers: assign and assume. The syntax for the assign transformer of $\text{NDOM}(V_N)$ is

$$x_1 := e_1 \wedge \cdots \wedge x_n := e_n \,,$$

where each x_i is in V_N, and each e_i is a linear arithmetic expression over V_N. The syntax for the assume transformer is

$$\mathit{assume}(e) \,,$$

where e is a linear Boolean expression over V_N. The semantics of the assign and assume transformers are standard – assign models abstract state update via variable assignments, while assume models abstract state update via addition of new constraints.

Example 2. The following are two examples of numeric abstract transformers:

$$x := y + 1 \wedge y := x - 1 \,,$$

$$\mathit{assume}(x + y \leq 5) \,.$$

□

For the predicate domain $\text{PRED}(V_P)$ over a set of predicates V_P, an abstract transformer is represented by a *Boolean* assignment of the form

$$p := \mathit{choice}(t, f) \,,$$

where $p \in V_P$ is a predicate, and t and f are Boolean expressions over V_P. Informally, t represents the condition under which p must be true, and f the condition under which p must be false. Note that t and f do not have to be mutually disjoint. Formally, the semantics of a Boolean assignment is defined as

$$||p := \mathit{choice}(t, f)||(e) = \textbf{let } R = (p' \wedge \neg f) \vee (\neg p' \wedge \neg t) \textbf{ in } (\exists V_P \cdot e \wedge R)[p'/p] \,,$$

where f, t, and e are propositional formulas over predicates in V_P, and the notation $e[p'/p]$ stands for replacing all occurrences of p' in e by p. Semantics of parallel composition of Boolean assignments is obtained by composing the semantic relations of the individual assignments, as usual. To our knowledge, the $\mathit{choice}(t, f)$ function was first introduced (and called **choose**) by Ball et al. [2] in the context of using Boolean and cartesian abstractions for model checking C programs. Ball et al. [2] also described an automated process for constructing abstract transformers involving $\mathit{choice}(t, f)$ from C statements using a theorem prover.

Example 3. The abstract transformer

$$p := \mathit{choice}(p, \neg p)$$

leaves p unchanged (p is true after the transformer *iff* p was true before). The abstract transformer

$$p := \mathit{choice}(\mathsf{false}, \mathsf{false})$$

changes p non-deterministically (nothing prevents p from being either true or false in the next state). The abstract transformer

$$p := \mathit{choice}(p \wedge q, \mathsf{false})$$

makes p true after the transformer if both p and q were true before it and changes p non-deterministically otherwise. □

In the case of a numeric abstract domain, an abstraction of a given a concrete statement s by an abstract transformer is done by the domain itself. In the case of predicate abstraction, an abstraction by Boolean assignments is computed using a theorem prover [13, 3].

3.4 Binary Decision Diagrams

Reduced Ordered Binary Decision Diagrams (ROBDDs, or BDDs for the purpose of this paper) [7] are a canonical representation of propositional formulas. A BDD is a DAG whose inner nodes correspond to propositional variables, two leaf nodes (i.e., nodes with no successors) corresponding to true and false. A path in a BDD corresponds to an assignment of values to variables. The paths leading to the true node correspond to all satisfying assignments of a formula represented by a BDD.

We use $\mathbf{0}$ and $\mathbf{1}$ to denote the constant BDDs for false and true, respectively. For a BDD u, we use $\mathsf{varOf}(u)$ for the variable corresponding to the root of u, $\mathsf{bddT}(u)$ for the then-branch of u, and $\mathsf{bddE}(u)$ for the else-branch of u, respectively. We make use of the following well known BDD operations:

Interface: NUMPREDDOM(V_N, V_P) **extends** ABSDOM

α_P	$: B \to A$	α_N	$: B \to A$
unprime	$: A \to A$	reduce	$: A \to A$
exists	$: 2^{V_P} \times A \to A$	$\alpha\mathsf{Post}_N$	$: S \to (A \to A)$

Fig. 4. The interface of NUMPREDDOM: V_N and V_P are numeric and propositional variables, respectively. E, B, S, and A are as in Fig. 3.

1. conjunction (bddAnd), disjunction (bddOr), and negation (bddNot);
2. if-then-else (bddIte);
3. existential quantification (bddExists); and
4. variable renaming (bddPermute).

Many of the above operations are implemented uniformly using a function bddApply(f, u, v), where u, v are BDDs, and f is a binary operator (i.e., conjunction, disjunction, etc.) that is defined only for the constant BDDs.

4 NUMPREDDOM: Interface

In this section, we describe the interface of NUMPREDDOM and abstract transfer functions supported by it. The interface NUMPREDDOM is shown in Fig. 4. It extends, i.e., has all the functions of, the basic abstract domain ABSDOM shown in Fig. 3. Notably, NUMPREDDOM has two types of variables: numeric, V_N, and propositional (or predicate), V_P. Furthermore, the domain is extended implicitly with "primed" propositional variables

$$V'_P \triangleq \{p' \mid p \in V_P\}\,.$$

The meaning of each predicate p in V_P is given by the concretization function γ. Conceptually, each element of NUMPREDDOM is a quantifier free first-order propositional formula over predicates V_P and numeric constraints over variables V_N.

Example 4. Consider NUMPREDDOM(V_N, V_P) where $V_N = \{x, y\}$, and $V_P = \{p\}$. A possible element is

$$(p \wedge (x \geq 0) \wedge (y \geq 0)) \vee (\neg p \wedge x < 0)\,.$$

Note that predicates can be interpreted as constraints over numeric variables. For instance, it is possible that $\gamma(p) = (x \geq 0)$. In this case, the value of x is represented both in predicate and numeric parts of the abstract value. □

The functions provided by NUMPREDDOM in addition to ABSDOM are:

1. abstraction function, α_N, is a restriction of the abstraction function α to conjunctions of linear constraints over V_N; That is, if $\alpha_N(e) = a$, then a is a conjunction of numeric constraints over variables in V_N and $e \Rightarrow \gamma(a)$.
2. abstraction function, α_P, is a restriction of the abstraction function α to minterms over V_P; That is, if $\alpha_P(e) = a$, then a is a propositional formula over predicates in V_P and $e \Rightarrow \gamma(a)$.
3. existential quantification, exists, over-approximates existential quantification of *propositional* variables from an abstract value. It must satisfy the over-approximation condition:
$$(\exists V \cdot \gamma(a)) \Rightarrow \gamma(\mathsf{exists}(V, a))\,;$$
4. variable renaming, unprime, renames all "primed" propositional variables into the corresponding unprimed ones;
5. abstract numeric transformer, $\alpha\mathsf{Post}_N$, lifts an abstract numeric only transformer to the combined domain; Given a numeric transformer τ and a NUMPREDDOM value $a_p \wedge a_n$, where a_p is a conjunction over predicates in V_P and a_n is a conjunction over numeric constraints over V_N,
$$\alpha\mathsf{Post}_N(\tau)(a_p \wedge a_n) \triangleq a_p \wedge \alpha\mathsf{Post}_N(\tau)(a_n)\,.$$
Moreover, it must distribute over disjunction. That is,
$$\alpha\mathsf{Post}_N(\tau)(a_1 \vee a_2) \triangleq \mathsf{join}(\alpha\mathsf{Post}_N(\tau)(a_1), \alpha\mathsf{Post}_N(\tau)(a_2))\,.$$
6. the reduction function, reduce, is a special operation that refines an abstract value by sharing information between propositional and numeric parts of the value. It must satisfy an over-approximation condition:
$$\gamma(a) \Rightarrow \gamma(\mathsf{reduce}(a))\,.$$
During analysis, reduce is applicable before or after any abstract operation to increase the precision of the final result. However, calls to reduce are expensive. By factoring it out in the interface, we allow its judicious use to target a suitable precision versus efficiency tradeoff.

4.1 Projection

To define the abstraction function α of NUMPREDDOM, we first introduce projection functions. These are used to break apart an expression that combines numeric and predicate terms. Let V_P be a set of predicates, V_N a set of numeric variables, and e be a conjunction of numeric terms, predicates, and negations of predicates.

1. The *propositional projection* of e onto V_P, denoted by $\mathsf{proj}_P(V_P, e)$, is a minterm over V_P that is implied by (i.e., over-approximates) e.
2. The *numeric projection* of e onto V_N, denoted by $\mathsf{proj}_N(V_N, e)$, is a conjunction of numeric constraints over V_N that is implied by e.

Example 5. The following are some sample applications of the projection functions:

$$\mathsf{proj}_P(\{p\}, p \wedge (x \geq 0) \wedge (y \geq 0)) \equiv p$$
$$\mathsf{proj}_P(\{x \geq 0\}, p \wedge (x \geq 0) \wedge (y \geq 0)) \equiv (x \geq 0)$$
$$\mathsf{proj}_N(\{y\}, p \wedge (x \geq 0) \wedge (y \geq 0)) \equiv y \geq 0$$

□

```
1: int x, y, *p;
2: ...
3: if (*p > 0) {
4:   *p = x; ...
5: } else {
6:   *p = 3; ...
7: }
8: ...
```

Fig. 5. A fragment of a C program.

Note that we have only partially specified the projection functions. The exact definitions of proj_P and proj_N are left to the implementation. In our implementation, they are done via approximations based on syntactic reasoning. More precise semantic constructions via the use of theorem provers are possible as well. Such implementation choices, as long as they satisfy the over-approximation conditions above, affect the efficiency vs. precision trade off, but not the soundness of the abstract domain.

4.2 Abstraction

Let e be a quantifier free formula in negation normal form. The abstraction function $\alpha(e)$ is defined recursively using α_P and α_N as follows:

- if e is a term, then

  $\alpha(e) \triangleq \mathsf{meet}(\alpha_P(\mathsf{proj}_P(V_P \cup V'_P, e)), \alpha_N(\mathsf{proj}_N(V_N, e)))$

- else if $e = e_1 \wedge e_2$, then

  $$\alpha(e) \triangleq \mathsf{meet}(\alpha(e_1), \alpha(e_2))$$

- else if $e = e_1 \vee e_2$, then

  $$\alpha(e) \triangleq \mathsf{join}(\alpha(e_1), \alpha(e_2))$$

4.3 Abstract Transformers

NUMPREDDOM supports a rich set of abstract transformers (shown in a BNF grammar in Fig. 6). In this section, we describe the syntax and semantics of each type of transformer, illustrate in what situations it is required, and, when applicable, provide a common implementation.

Numeric. The "numeric" abstract transformer's syntax is

$$x_1 := e_1 \wedge \cdots \wedge x_k := e_k \,,$$

where the variables in x_i and e_i are in V_N. Its semantics is defined in terms of the $\alpha\mathsf{Post}_N$ of each implementation of NUMPREDDOM as follows:

$$\lambda X \cdot \alpha\mathsf{Post}_N(x_1 := e_1 \wedge \cdots \wedge x_l := e_k)(X) \,.$$

It is a basic building block for abstracting arithmetic transformations.

Assume. The "assume" abstract transformer's syntax is

$$\mathit{assume}(e) \,,$$

where e is an arbitrary expression. It semantics is

$$\lambda X \cdot \mathsf{meet}(\alpha(e), X) \,.$$

It is used to approximate program conditionals with a combination of predicate and numeric conditions.

Example 6. For example, consider a fragment of a C program shown in Fig. 5. In the program, `x` and `y` are two integer variables, and `p` is a pointer to an integer. Ellipsis indicate that part of the program is not shown. Let the predicates $V_P = \{p = \&x, p = \&y\}$ and numeric variables $V_N = \{x, y\}$. Then, the conditional of the then-branch of the if-statement on line 3,

$$\mathit{assume}(*p > 0) \,,$$

can be approximated by the assume transformer

$$\mathit{assume}((p = \&x \wedge x > 0) \vee (p = \&y \wedge y > 0) \vee (p \neq \&y \wedge p \neq \&x))$$

Informally, the abstract transformer says that either (a) p points to x and x is positive, or (b) p points to y and y is positive, or (c) p does not point to either x or y. □

Conditional. The "conditional" abstract transformer's syntax is $e?\tau$, where e is an arbitrary expression, and τ is a purely numeric transformer. Its semantics is

$$\lambda X \cdot \alpha\mathsf{Post}_N(\tau)(\alpha\mathsf{Post}(\mathit{assume}(e))(X)) \,.$$

The conditional transformer is most useful in a combination with other transformers.

Example 7. Again, consider the program fragment in Fig. 5, and recall that $V_P = \{p = \&x, p = \&y\}$ and $V_N = \{x, y\}$. The assignment $*p := 3$ on line 6 can be abstracted by a conditional transformer:

$$(p = \&x \ ?\ x := e) \vee (p = \&y \ ?\ y := e) \,,$$

Informally, the above transformer means that either p points to x and x gets 3, or p points to y and y gets 3, or p does not point to either x or y, and x and y are unchanged. □

Predicate. The "predicate" abstract transformer's syntax is

$$p_1 := \mathit{choice}(t_1, f_1) \wedge \cdots \wedge p_n := \mathit{choice}(t_n, f_n) \,,$$

where each p_i is in V_P and each t_i and f_i is an expression over V_P and V_N. Its semantics is defined using conjunction and existential quantification as follows:

$$\textbf{let } R = \alpha(\textstyle\bigwedge_i (p'_i \wedge \neg f_i) \vee (\neg p'_i \wedge \neg t_i)) \textbf{ in}$$
$$\lambda X \cdot \mathsf{unprime}(\mathsf{exists}(\{p_1, \ldots, p_n\}, \mathsf{meet}(X, R))) \,.$$

The predicate transformer is the basic building block for predicate abstraction. It depends on both predicate and numeric information.

$$
\begin{array}{rcll}
\tau & ::= & \tau_N \mid \tau_a \mid \tau_c \mid \tau_P \mid \tau_{NP} \mid & \text{(base case)} \\
 & & \tau; \tau \mid & \text{(sequence)} \\
 & & \tau \vee \tau & \text{(non-det.)} \\
\tau_{NP} & ::= & (e?\tau_N) \wedge \tau_P & \text{(numeric + predicate)} \\
\tau_P & ::= & p := \mathit{choice}(e, e) \mid & \text{(predicate)} \\
 & & \tau_P \wedge \tau_P & \\
\tau_c & ::= & e?\tau_N & \text{(conditional)} \\
\tau_a & ::= & \mathit{assume}(e) & \text{(assume)} \\
\tau_N & ::= & x := v \mid & \text{(numeric)} \\
 & & \tau_N \wedge \tau_N &
\end{array}
$$

Fig. 6. BNF grammar for abstract transformers supported by NUMPREDDOM; p is a predicate; x a numeric variable; e an expression over predicates and numeric terms; v a numeric expression.

Example 8. Again, consider the program fragment in Fig. 5. Let the predicates

$$V_P = \{y > 0, p = \&x, p = \&y\}$$

and numeric variables $V_N = \{x\}$. Then, the assignment $*p{:=}\, x$ on line 4 can be abstracted as:

$$
\begin{array}{l}
(y > 0){:=} \\
\quad \mathit{choice}((p = \&x) \wedge (y > 0) \vee (p = \&y) \wedge (x > 0), \\
\qquad\qquad (p = \&x) \wedge (y \leq 0) \vee (p = \&y) \wedge (x \leq 0)) \,.
\end{array}
$$

Intuitively, this abstract transformer means that y becomes positive if p points to x and y was positive, or if p points to y and x was positive. Moreover, y becomes non-positive (i.e., ≤ 0) if either p points to x and y was non-positive, or if p points to y and x was non-positive. □

Numeric and Predicate. The "numeric and predicate" abstract transformer's syntax involves a parallel composition of conditional numeric and predicate transformers as follows:

$$(e?\tau_N) \wedge \tau_P \,,$$

where e is an arbitrary expression, τ_N is a purely numeric transformer, and τ_P is a predicate transformer. Its semantics is defined using the equivalence

$$(e?\tau_N) \wedge \tau_P \equiv \mathit{assume}(e); \tau_P; \tau_N \,.$$

That is, since the purely numeric transformer τ_P does not depend on the predicates, this parallel composition is reduced to a sequential one. This transformer is used to abstract statements that influence both predicates and numeric constraints simultaneously. Even though τ_N does not involve predicates, it is influenced by predicates in the condition e.

Example 9. Let $V_P = \{y = 1\}$ and $V_N = \{x, v, w\}$. Then, the parallel statement

$$y := x \wedge x := (y = 1)?v : w$$

is abstracted as

$$
\begin{array}{l}
((y = 1)?x = v : x = w) \wedge \\
\qquad\qquad (y = 1) := \mathit{choice}(x = 1, x \neq 1) \,.
\end{array}
$$

The above abstract transformer means that y becomes 1 iff x was 1, and x gets v or w depending on whether y was equal to 1 before. Note that the predicate $y = 1$ is influenced by numeric constraints on x, and influences the next value of x. □

Sequential and Non-Deterministic. The syntax of "sequential" and "non-deterministic" abstract transformers is given by $\tau_1; \tau_2$ and $\tau_1 \vee \tau_2$, respectively. Their semantics is defined using function composition and join operator, respectively:

$$
\begin{array}{l}
\alpha\mathsf{Post}(\tau_1; \tau_2) = \lambda X \cdot \alpha\mathsf{Post}(\tau_2)(\alpha\mathsf{Post}(\tau_1)(X)) \\
\alpha\mathsf{Post}(\tau_1 \vee \tau_2) = \lambda X \cdot \mathsf{join}(\alpha\mathsf{Post}(\tau_1)(X), \alpha\mathsf{Post}(\tau_2)(X))
\end{array}
$$

Example 10. A complete example of the combined predicate and numeric abstraction is shown in Fig. 2. Part (a) of the figure shows a fragment of a program. Part (b) of the figure shows the abstraction of Part (a) with predicates $V_P = \{p, q\}$, where $p \triangleq ((A[y_1 + y_2] = 3)$, and $q \triangleq (A[x_1 + x_2] = 3)$, and numeric variables $V_N = \{x_1, x_2, y_1, y_2\}$. Note that the two parts of Fig. 2 are formatted to align their corresponding statements. Moreover, *assume* and disjuction are used in Fig. 2(b) to model `if-then-else` statements in Fig. 2(a).

The abstraction is precise enough to establish that the program is safe (i.e., the assertions are not violated). The predicates p and q are necessary to separate different paths through the control flow. A transfer function for predicate q must depend on a combination of numeric constraints and the value of the predicate p. In this example, the tight combination of predicate and numeric abstraction is crucial: an abstraction of Part (a) using only numeric domain over V_N is not precise enough to establish safety; discovering the predicates for a precise predicate abstraction is non-trivial. □

From Concrete to Abstract Programs. The transformers presented in this section are abstract in the sense that their semantics is defined using the transformers of the underlying predicate and numeric abstract domains and the basic operations (i.e, meet, join, etc.) of NUMPREDDOM.

Using NUMPREDDOM to abstract and reason about a concrete program requires an additional abstraction function, α^τ, that maps concrete statements to abstract transformers. An implementation of α^τ must be sound: for any concrete program statement s, the semantics of $\alpha^\tau(s)$ must "over-approximate" the semantics of s. However, an implementation is free to make its own trade-off between precision and efficiency. We describe our implementation of α^τ in Section 6.

In summary, the critical operations in the NUMPREDDOM interface are exists, unprime, proj_N, proj_P, α_N, α_P, γ, leq, meet, join, widen, $\alpha\mathsf{Post}_N$ and reduce. In the rest of the article, we present four different implementations of these operations and evaluate them empirically.

5 NUMPREDDOM: Implementations

In this section, we describe four different implementations of NUMPREDDOM. We use N to denote the set of abstract values of the underlying numeric domain over V_N, and P to denote the set of propositional formulas over V_P. In other words, $P = 2^{2^{V_P}}$. We write $N.\mathsf{op}$ and $P.\mathsf{op}$ to mean the abstract operation op over numerics and predicates respectively. We write $\sqsubseteq$, $\sqcap$, $\sqcup$ and ∇ to mean leq, meet, join and widen when the abstract domain is clear from context. We write $X.\mathsf{top}$, $X.\mathsf{bot}$ to mean $X.\alpha(\mathsf{true})$ and $X.\alpha(\mathsf{false})$, respectively, representing the top and the bottom elements of the domain X. All four implementations of NUMPREDDOM share the definitions of proj_N and proj_P, which are based on syntactic simplification of expressions to a normal form.

5.1 NEXPOINT: Numeric Explicit Points

NEXPOINT domain is the simplest of our combinations. The set of abstract values of NEXPOINT is $P \times N$. A NEXPOINT value is represented by a pair (p, n) where p is a BDD and n is a numeric abstract value. Intuitively, a pair (p, n) represents the expression $P.\gamma(p) \wedge N.\gamma(n)$. The top and bottom elements of NEXPOINT are defined as follows:

$$\text{NEXPOINT.top} \triangleq (P.\mathsf{top}, N.\mathsf{top})$$
$$\text{NEXPOINT.bot} \triangleq (P.\mathsf{bot}, N.\mathsf{bot})$$

The exists and unprime operations are performed on the BDD part of the tuple:

$$\mathsf{exists}(S, (p, n)) \triangleq (\mathsf{bddExists}(S, p), n)$$
$$\mathsf{unprime}(S, (p, n)) \triangleq (\mathsf{bddPermute}(S', S, p), n)\,,$$

where $S \subseteq V_P$ is a set of propositional variables, and $S' = \{s' \mid s \in S\}$. Most of the remaining operations are performed pointwise. Specifically,

$$\alpha_N(e) \triangleq (P.\mathsf{top}, N.\alpha(e))$$
$$\alpha_P(e) \triangleq (P.\alpha(e), N.\mathsf{top})$$
$$\gamma(p, n) \triangleq P.\gamma(p) \wedge N.\gamma(n)$$
$$\mathsf{op}((p, n), (p', n')) \triangleq (P.\mathsf{op}(p, p'), N.\mathsf{op}(n, n'))$$
$$\mathsf{leq}((p, n), (p', n')) \triangleq p \sqsubseteq p' \wedge n \sqsubseteq n'$$
$$\alpha\mathsf{Post}_N(s) \triangleq \lambda(p, n) \,.\, (p, N.\alpha\mathsf{Post}(s)(n))\,,$$

where $\mathsf{op} \in \{\mathsf{meet}, \mathsf{join}, \mathsf{widen}\}$. Note that our definition of leq above is sound, i.e., satisfies the requirements of leq shown in Fig. 3. However, it is not the strongest (most precise) possible one. In particular, it does not ensure the precision condition:

$$\mathsf{leq}((p, n), (p', n')) \Leftrightarrow (\gamma(p, n) \Rightarrow \gamma(p', n'))\,. \qquad (\star)$$

The advantage of our definition is that it admits an efficient implementation on top of the leq operators of the underlying numeric and predicate domains. A more precise leq, ensuring the condition $(\star)$, would be much more expensive to implement. The reduce operation is defined as follows:

$$\mathsf{reduce}(v) \triangleq \alpha(\gamma(v))\,.$$

Example 11. Consider NEXPOINT domain with $V_P = \{q, r\}$, $V_N = \{x, y\}$, and predicates q and r interpreted as $q \triangleq (x = 0)$ and $r \triangleq (y = 0)$. Then,

$$\mathsf{reduce}(q \vee r, x = 3 \wedge y \geq 0) = (\neg q \wedge r, x = 3 \wedge y = 0)\,.$$

Note that the output of reduce is equivalent to its input in the concrete world, since

$$\gamma(q \vee r, x = 3 \wedge y \geq 0) \equiv \gamma(\neg q \wedge r, x = 3 \wedge y = 0)\,.$$

However, the output is strictly more precise than the input in the abstract world since

$$\mathsf{leq}((\neg q \wedge r, x = 3 \wedge y = 0), (q \vee r, x = 3 \wedge y \geq 0)) \;\bigwedge$$
$$\neg\mathsf{leq}((q \vee r, x = 3 \wedge y \geq 0), (\neg q \wedge r, x = 3 \wedge y = 0))\,.$$

This means that applying reduce during abstract analysis has the potential of yielding more precise results. Similarly,

$$\mathsf{reduce}(q \vee r, x = 3 \wedge y < 0) = \text{NEXPOINT.bot}\,.$$

Once again, the output of reduce is equivalent to its input in the concrete world, since

$$\gamma(q \vee r, x = 3 \wedge y < 0) \equiv \mathsf{false}\,.$$

However, the output is strictly more precise than the input in the abstract world, since

$$\mathsf{leq}(\text{NEXPOINT.bot}, (q \vee r, x = 3 \wedge y < 0)) \;\bigwedge$$
$$\neg\mathsf{leq}((q \vee r, x = 3 \wedge y < 0), \text{NEXPOINT.bot})\,.$$

□

The abstract domains NEX and MTNDD, presented in the next two sections, share the above definition of reduce with NEXPOINT. Therefore, we only define reduce specifically for our fourth abstract domain NDD. The following theorem summarizes the correctness of NEXPOINT.

Theorem 1. NEXPOINT *implements* NUMPREDDOM.

Proof. We know that NEXPOINT exports all operations required by NUMPREDDOM. We prove that the operations satisfy the required properties.

For the meet operation, let (p, n) and (p', n') be two NEXPOINT abstract values. We know that:

$$P.\gamma(p) \wedge P.\gamma(p') \Rightarrow P.\gamma(p \sqcap p') \;\bigwedge$$
$$N.\gamma(n) \wedge N.\gamma(n') \Rightarrow N.\gamma(n \sqcap n')$$

which implies

$$(P.\gamma(p) \wedge N.\gamma(n)) \bigwedge (P.\gamma(p') \wedge N.\gamma(n')) \Rightarrow$$
$$P.\gamma(p \sqcap p') \wedge N.\gamma(n \sqcap n')$$

which implies

$$\gamma(p,n) \wedge \gamma(p',n') \Rightarrow \gamma(p \sqcap p', n \sqcap n') \ .$$

For the join operation, let (p,n) and (p',n') be two NEXPOINT abstract values. We know that:

$$P.\gamma(p) \vee P.\gamma(p') \Rightarrow P.\gamma(p \sqcup p') \ \bigwedge \\ N.\gamma(n) \vee N.\gamma(n') \Rightarrow N.\gamma(n \sqcup n')$$

which implies

$$(P.\gamma(p) \wedge N.\gamma(n)) \bigvee (P.\gamma(p') \wedge N.\gamma(n')) \Rightarrow \\ P.\gamma(p \sqcup p') \wedge N.\gamma(n \sqcup n')$$

which implies

$$\gamma(p,n) \vee \gamma(p',n') \Rightarrow \gamma(p \sqcup p', n \sqcup n') \ .$$

The proof for widen is similar to that of join. For isTop, let (p,n) be an abstract NEXPOINT value such that isTop(p,n). Therefore $p = P.\mathsf{top}$ and $n = N.\mathsf{top}$, and hence, $P.\gamma(p) = \mathsf{true}$ and $N.\gamma(n) = \mathsf{true}$. That is, $\gamma(p,n) = \mathsf{true}$, which is what we want.

For isBot, let (p,n) be an abstract NEXPOINT value such that isBot(p,n). Therefore $p = P.\mathsf{bot}$ and $n = N.\mathsf{bot}$, and hence, $P.\gamma(p) = \mathsf{false}$ and $N.\gamma(n) = \mathsf{false}$. That is, $\gamma(p,n) = \mathsf{false}$, which is what we want.

For leq, let (p,n) and (p',n') be two NEXPOINT abstract values such that $(p,n) \sqsubseteq (p',n')$. Therefore,

$$p \sqsubseteq p' \wedge n \sqsubseteq n'$$

which implies

$$P.\gamma(p) \Rightarrow P.\gamma(p') \wedge N.\gamma(n) \Rightarrow N.\gamma(n')$$

which implies

$$P.\gamma(p) \wedge N.\gamma(n) \Rightarrow P.\gamma(p') \wedge N.\gamma(n')$$

which is what we want. To prove that $e \Rightarrow \gamma(\alpha(e))$ for any expression e, we induct on the structure of e and consider three cases:

– *Case 1.* e is a term. In this case,

$$\alpha(e) \triangleq \mathsf{meet}(\alpha_P(\mathsf{proj}_P(V_P \cup V_P', e)), \alpha_N(\mathsf{proj}_N(V_N, e)))$$

Let us write e_P to mean $\mathsf{proj}_P(V_P \cup V_P', e)$ and e_N to mean $\mathsf{proj}_N(V_N, e)$. Therefore,

$$\alpha(e) = (P.\alpha(e_P), N.\mathsf{top}) \sqcap (P.\mathsf{top}, N.\alpha(e_N)) = \\ (P.\alpha(e_P) \sqcap P.\mathsf{top}, N.\mathsf{top} \sqcap N.\alpha(e_N))$$

Hence, from the definitions of meet and γ, and the fact that $P.\gamma(P.\mathsf{top}) = \mathsf{true}$ and $N.\gamma(N.\mathsf{top}) = \mathsf{true}$, we know that

$$P.\gamma(P.\alpha(e_P)) \wedge N.\gamma(N.\alpha(e_N)) \Rightarrow \gamma(\alpha(e))$$

Now, we know that

$$e_P \Rightarrow P.\gamma(P.\alpha(e_P)) \bigwedge e_N \Rightarrow N.\gamma(N.\alpha(e_N))$$

Therefore, $e_P \wedge e_N \Rightarrow \gamma(\alpha(e))$. Finally, from the definitions of proj_P and proj_N, we know that $e \Rightarrow e_P$ and $e \Rightarrow e_N$. Therefore, $e \Rightarrow e_P \wedge e_N \Rightarrow \gamma(\alpha(e))$, which is what we want.

– *Case 2.* $e = e_1 \wedge e_2$. In this case,

$$\alpha(e) \triangleq \mathsf{meet}(\alpha(e_1), \alpha(e_2))$$

By inductive application, we know that

$$e_1 \Rightarrow \gamma(\alpha(e_1)) \bigwedge e_2 \Rightarrow \gamma(\alpha(e_2))$$

Therefore,

$$e \equiv e_1 \wedge e_2 \Rightarrow \gamma(\alpha(e_1)) \wedge \gamma(\alpha(e_2))$$

Also, by the definition of meet, we know that:

$$\gamma(\alpha(e_1)) \wedge \gamma(\alpha(e_2)) \Rightarrow \gamma(\mathsf{meet}(\alpha(e_1), \alpha(e_2))) \equiv \gamma(\alpha(e))$$

Hence, $e \Rightarrow \gamma(\alpha(e))$, which is what we want.

– *Case 3.* $e = e_1 \vee e_2$. In this case,

$$\alpha(e) \triangleq \mathsf{join}(\alpha(e_1), \alpha(e_2))$$

By inductive application, we know that

$$e_1 \Rightarrow \gamma(\alpha(e_1)) \bigwedge e_2 \Rightarrow \gamma(\alpha(e_2))$$

Therefore,

$$e \equiv e_1 \vee e_2 \Rightarrow \gamma(\alpha(e_1)) \vee \gamma(\alpha(e_2))$$

Also, by the definition of join, we know that

$$\gamma(\alpha(e_1)) \vee \gamma(\alpha(e_2)) \Rightarrow \gamma(\mathsf{join}(\alpha(e_1), \alpha(e_2))) \equiv \gamma(\alpha(e))$$

Hence, $e \Rightarrow \gamma(\alpha(e))$, which is what we want.

Finally, the requirement on αPost is satisfied by combining the semantics of NUMPREDDOM abstract transformers with the requirement on α^τ. This completes the proof. □

5.2 NEX*: Numeric Explicit Sets*

The NEX domain extends the expressive power of NEXPOINT by allowing different predicate valuations to map to different numeric constraints. Each abstract value of the NEX domain is a function $2^{V_P} \mapsto N$ and is represented as a set of pairs

$$\{(p_1, n_1), \ldots, (p_k, n_k)\} \subseteq P \times N \ ,$$

where each p_i is a BDD, each n_i is a numeric abstract value. To ensure that each NEX value is indeed a function, the set must satisfy the following well-formedness conditions:

$$\forall 1 \leq i \leq k \,.\, p_i \neq P.\mathsf{bot} \wedge n_i \neq N.\mathsf{bot} \qquad \textbf{(C1)}$$

$$\forall 1 \leq i < j \leq k \,.\, n_i \neq n_j \qquad \textbf{(C2)}$$

$$\forall 1 \leq i < j \leq k \,.\, p_i \sqcap p_j = P.\mathsf{bot} \qquad \textbf{(C3)}$$

Conditions **C1**–**C3** above ensure that the data structures are as "tight" as possible: **C1** guarantees that the representation of any abstract value does not include any "empty" components, **C2** ensures that any two elements (p_1, n_1) and (p_2, n_2) are distinguished by their numeric components, and **C3** ensures that the elements of a NEX value are "mutually disjoint". Intuitively, a NEX value is a "union" of NEXPOINT values that are distinguished by their numeric components. Thus, NEX improves upon the precision of NEXPOINT by replacing imprecise numeric join with union.

1: NEX norm ($2^{P\times N}$ u)
2: while ($\exists(p,n) \in u$. $p = P.\mathsf{bot} \vee n = N.\mathsf{bot}$) do
3: $u := u \setminus \{(p,n)\}$
4: while ($\exists(p,n) \in u \wedge \exists(p',n) \in u$) do
5: $u := u \setminus \{(p,n),(p',n)\} \cup \{(p \sqcup p', n)\}$
6: return u

Fig. 7. Implementation of norm.

Example 12. Suppose that $V_N = \{x, y\}$ and $V_P = \{p, q\}$. Then the expression

$$e \triangleq (p \wedge \neg q \wedge x < 0) \vee (\neg p \wedge q \wedge y < 0)$$

is represented precisely by the NEX abstract value

$$\{(p \wedge \neg q, x < 0), (\neg p \wedge q, y < 0)\}\,.$$

Note that e has no precise representation in terms of NEXPOINT abstract values. □

The top and bottom elements of NEX are defined as

$$\text{NEX.}\mathsf{top} \triangleq \{(P.\mathsf{top}, N.\mathsf{top})\}$$
$$\text{NEX.}\mathsf{bot} \triangleq \emptyset\,.$$

To explain the other NEX operations, we introduce a normalizing procedure called norm. The implementation of norm is given in Fig. 7. Given a set $u \subseteq P \times N$ satisfying **C3**, $\mathsf{norm}(u)$ returns a NEX value v, i.e., $v \subseteq P \times N$ satisfies **C1**–**C3**. The following theorem summarizes the key properties of norm.

Theorem 2. *Let u be an element of $2^{P\times N}$ satisfying condition **C3**. Then, (a) $\mathsf{norm}(u)$ is a* NEX *value that is semantically equivalent to u; (b) the complexity of* norm *is in $O(|u|^2)$.*

Proof. Proof of Part(a) follows from the fact that every step of norm maintains the semantic value of its input u, and that $\mathsf{norm}(u)$ is a legal NEX abstract value. Proof of Part(b) follows from the fact that norm looks at all pairs of elements in u. Note that it is also possible to implement norm in linear time by using a hashtable. □

The operations exists and unprime are performed on the BDDs, and are then joined together. Specifically,

$$\mathsf{exists}(S, \{(p_i,n_i)\}_{i=1,..k}) \triangleq \bigsqcup_{i=1}^{k} \{(\mathsf{bddExists}(S,p_i), n_i)\}\,,$$

and

$$\mathsf{unprime}(\{(p_i,n_i)\}_{i=1,..k}) \triangleq \bigsqcup_{i=1}^{k} \{(\mathsf{bddPermute}(V_P, V'_P, p_i), n_i)\}\,,$$

where $\bigsqcup$ is the join operator of NEX that is defined later in this section.

1: $2^{P\times N}$ NEXJoin (NEX u, NEX v)
2: if ($u = \emptyset$) return v
3: if ($v = \emptyset$) return u
4: let u be $\{(p,n)\} \cup X$ and v be $\{(p',n')\} \cup X'$
5: $x := \{(p \sqcap p', n \sqcup n')\}$
6: $y := \mathsf{NEXJoin}(\{(p \sqcap \neg p', n)\}, X')$
7: $z := \mathsf{NEXJoin}(\{(p' \sqcap \neg p, n')\}, X)$
8: return $x \cup y \cup z \cup \mathsf{NEXJoin}(X, X')$

Fig. 8. Implementation of NEXJoin.

The abstraction and concretization operations for NEX are defined as follows:

$$\alpha_N(e) \triangleq \{(P.\mathsf{top}, N.\alpha(e))\}$$
$$\alpha_P(e) \triangleq \{(P.\alpha(e), N.\mathsf{top})\}$$
$$\gamma(\{(p_i,n_i)\}_{i=1,..k}) \triangleq \bigvee_{1\le i\le k} P.\gamma(p_i) \wedge N.\gamma(n_i)$$

We define the leq operation for NEX in two stages. First we define leq between a NEXPOINT and a NEX value. Let $v = (p,n)$ be a NEXPOINT value and

$$v' = \{(p'_i, n'_i)\}_{i=1,..k}$$

be a NEX value. Then, we say that $v \sqsubseteq v'$ iff

$$p \sqsubseteq \bigsqcup_{\{i \mid n \sqsubseteq n'_i\}} p'_i\,.$$

Finally, for any two NEX values

$$v = \{(p_i,n_i)\}_{i=1,..k}$$

and v', we say that $\mathsf{leq}(v,v')$ iff

$$\forall 1 \le i \le k \,.\, (p_i,n_i) \sqsubseteq v'\,.$$

We now define the operations meet, join, and widen.

$$\mathsf{meet}(u,v) \triangleq \mathsf{norm}(\{(p \sqcap p', n \sqcap n') \mid (p,n) \in u \wedge (p',n') \in v\})$$
$$\mathsf{join}(u,v) \triangleq \mathsf{norm}(\mathsf{NEXJoin}(u,v))$$
$$\mathsf{widen}(u,v) \triangleq \mathsf{norm}(\mathsf{NEXWiden}(u,v))$$

The function NEXJoin used to define join above is described in Fig. 8. The key idea behind NEXJoin is to ensure that its output satisfies **C3** by splitting $p \sqcup p'$ into three mutually disjoint fragments: $p \sqcap p'$, $p \sqcap \neg p'$ and $p' \sqcap \neg p$. The algorithm NEXWiden is identical to NEXJoin except that it uses ∇ instead of $\sqcup$ at Line 5. Note that meet is defined differently because, unlike join and widen, it distributes over union. The complexity of $\mathsf{meet}(u,v)$, $\mathsf{join}(u,v)$ and $\mathsf{widen}(u,v)$ operations is in $O(|u| \cdot |v|)$. Finally, the operation $\alpha\mathsf{Post}_N$ is defined as follows:

$$\alpha\mathsf{Post}_N(s) \triangleq \lambda v.\mathsf{norm}(\{(p, N.\alpha\mathsf{Post}(s)(n)) \mid (p,n) \in v\})\,.$$

Theorem 3. NEX *implements* NUMPREDDOM.

Proof. Follows from the above definitions. □

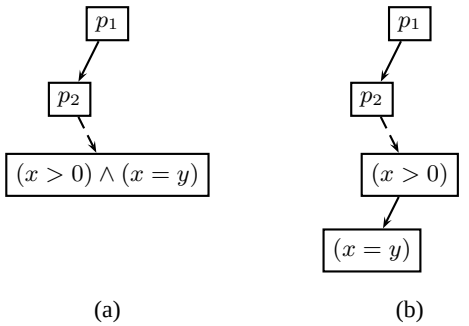


Fig. 9. A value from MTNDD domain: shown as an MTDD (a), and a BDD (b). 1-edges are solid, 0-edges are dashed. Edges to $\mathbf{0}$ are omitted for brevity.

5.3 MTNDD*: Multi-Terminal Numeric Decision Diagrams*

MTNDD is a symbolic alternative to NEX. The values of MTNDD are also functions of type $2^{V_P} \mapsto N$. Unlike in NEX, in MTNDD a value is represented as a BDD over predicate and numeric terms. This symbolic representation automatically maintains the partitioning conditions **C1**–**C3** of NEX.

Conceptually, an MTNDD value is a Multi-Terminal BDD [1], whose terminals are numeric abstract values from N. In our implementation, we simulate MTBDDs with BDDs by (a) associating a BDD variable with each predicate and numeric term, and (b) restricting variable ordering to ensure that predicate variables always precede numeric ones. For any term t that is both predicate and numeric (i.e., $\mathsf{proj}_P(t) = t = \mathsf{proj}_N(t)$), we allocate two distinct BDD variables: one representing the predicate, and one representing the numeric term. Note that although there are infinitely many numeric terms, only finitely many are used in any analysis. Thus, we allocate variables for numeric terms dynamically.

Example 13. Let $V_P = \{p_1, p_2\}$, $p_1 \equiv (x > 0)$, $p_2 \equiv (z < y)$, and $V_N = \{x, y\}$. Consider an expression

$$(x > 0) \wedge (z \geq y) \wedge (x = y)\,.$$

Its representation by an MTBDD and by a BDD are shown in Fig. 9(a) and Fig. 9(b), respectively. □

We assume existence of the functions toBdd and toExpr to convert between BDDs and expressions in the usual way. That is, if e is an expression, then $\mathsf{toBdd}(e)$ is the BDD corresponding to e, and if u is a BDD then $\mathsf{toExpr}(u)$ is an expression represented by u. Note that similar conversions for NEXPOINT and NEX were done via $P.\alpha$ and $P.\gamma$ of the predicate abstraction domain. Furthermore, we assume existence of the function $\mathsf{isNum}(v)$ that for a BDD v determines whether the root variable of v is a numeric term.

The top and bottom values of MTNDD, MTNDD.top and MTNDD.bot, are represented by BDDs $\mathbf{1}$ and $\mathbf{0}$, respectively. Abstraction and concretization functions simply convert between expressions and BDDs. Formally,

$$\begin{aligned}
\text{MTNDD.top} &\triangleq \mathbf{1}\\
\text{MTNDD.bot} &\triangleq \mathbf{0}\\
\alpha_P(t) &\triangleq \mathsf{toBdd}(P.\gamma(P.\alpha(t)))\\
\alpha_N(t) &\triangleq \mathsf{toBdd}(N.\gamma(N.\alpha(t)))\\
\gamma(v) &\triangleq \mathsf{toExpr}(v)
\end{aligned}$$

Note that in the definition of α_P and α_N, the corresponding predicate and numeric domains are used to normalize the expression t before it is stored as a BDD. The unprime operation is done using bddPermute. The MTNDD.exists operation is implemented identically to bddExists, with the exception that MTNDD.join is used instead of bddOr. We omit the explicit definition of MTNDD.exists for brevity.

The operations meet, join, widen, leq, and $\alpha\mathsf{Post}_N$ are implemented using bddApply. They work by (a) using bddApply to recursively traverse the input BDD(s) until the inputs are reduced to BDDs over only numeric terms; (b) converting numeric BDDs to abstract values and applying the corresponding numeric operation; and (c) encoding the result back as a BDD. To illustrate, MTNDD.join is defined as:

$$\text{MTNDD.join}(u, v) \triangleq \mathsf{bddApply}(\mathsf{MJoinOp}, u, v)\,,$$

where the code for MJoinOp is shown in Fig. 10. Note that we require that all BDD variables corresponding to predicates precede all BDD variables that correspond to numeric terms. Thus, whenever a root of a BDD v is numeric, the rest of v is numeric as well. The implementations of operations meet, widen, leq, and $\alpha\mathsf{Post}_N$ follow the same pattern. We omit their explicit definitions for brevity.

Theorem 4. MTNDD *implements* NUMPREDDOM.

```
1: BDD MJoinOp (BDD u, BDD v)
2:   if (u = 1 ∨ v = 1) return 1
3:   if (u = 0) return v
4:   if (v = 0) return u
5:   if (isNum(u) ∧ isNum(v))
6:     nu := N.α(toExpr(u))
7:     nv := N.α(toExpr(v))
8:     return toBdd(N.γ(nu ⊔ nv))
9:   return null
```

Fig. 10. Implementation of MJoinOp.

```
1: BDD ctxApply (BDD u, Op g, N c, Set V)
2:   r := g(u, c)
3:   if (r ≠ null) return r
4:   b := varOf(u);  e := term(u)
5:   tt = ctxApply(bddT(u), g, e ⊓ c, V)
6:   ff = ctxApply(bddE(u), g, ¬e ⊓ c, V)
7:   if (b ∈ V)
8:     return bddOr(tt, ff)
9:   else
10:    return bddIte(b, tt, ff)
```

Fig. 11. Implementation of ctxApply.

Proof. Follows from the above definitions. □

MTNDD operations are implemented using generic bddApply. Thus, their complexity is linear in the size of their inputs.

5.4 NDD*: Numeric Decision Diagrams*

NDD is our most expressive domain. Its elements are in $2^{P \times N}$. An NDD value is represented by a BDD encoding a propositional formula over predicate and numeric terms.

Each term t is assigned a unique BDD variable. This assignment takes negation into account: any two complementary terms t_1 and t_2, i.e., $t_1 = \neg t_2$, are associated with the opposite phases of the same BDD variable. For example, if $x > 0$ is mapped to a BDD variable v, then $x \leq 0$ is mapped to $\neg v$. We write $\mathsf{term}(v)$ to denote the term corresponding to v. We extend the notation to BDDs and write $\mathsf{term}(u)$ to mean the term of the root variable of BDD u.

The BDD variable allocated to a term t is independent of whether t is a predicate, a numeric term, or both: each term gets just one variable. Thus, an expression e that is propositionally inconsistent is always represented by the special BDD $\mathbf{0}$. Note that this is not true of the other three implementations. For example, let $V_P = \{p\}$, $p \equiv (x > 0)$, and $V_N = \{x\}$. Then, $p \wedge (x \leq 0)$ is reduced to $\mathbf{0}$.

Almost all of NDD operations are done using corresponding BDD operations. The NDD.top and NDD.bot are represented by BDDs $\mathbf{1}$ and $\mathbf{0}$, respectively. Abstraction and concretization functions α_P, α_N, and γ are exactly the same as in MTNDD — they simply convert between expressions and BDDs. Functions unprime, exists, meet, and join are implemented as bddPermute, bddExists, bddAnd, and bddOr, respectively. The widen operation is implemented by conversion to MTNDD.

All of these operations work on propositional structure of the abstract value. They treat numeric constraints as uninterpreted propositional symbols. Their complexity is linear in the size of the input.

The operations reduce, leq, and αPost are different since they must take into account the semantics of the numeric terms. To implement them, we introduce a function ctxApply, whose implementation is shown in Fig. 11. The function $\mathsf{ctxApply}(u, g, c, V)$ recursively traverses a BDD u, collecting the context of the current path in c, applying operation g at the subtrees, and and existentially eliminating variables in V. The complexity of ctxApply is linear in the number of paths in u.

The reduce operation is implemented by removing all unsatisfiable paths from a BDD. It is implemented using ctxApply as follows:

$$\mathsf{ctxApply}(u, \mathsf{reduceOp}, N.\mathsf{top}, \emptyset)\,,$$

where the code for reduceOp is shown in Fig. 12. The operator reduceOp checks for satisfiability of the current context, and replaces unsatisfiable context with $\mathbf{0}$. The rules for BDD simplification ensure that a path with unsatisfiable context is removed. An important observation is that if a BDD v is semantically unsatisfiable, then $\mathsf{reduce}(v)$ reduces v to $\mathbf{0}$.

```
1: BDD reduceOp(BDD u, N c)
2:   if N.isBot(c) return 0
3:   if (u = 0) return 0
4:   if (u == 1) return 1
5:   return null
```

Fig. 12. Implementation of reduceOp.

```
1: BDD NDDPost(s)(BDD u, N c)
2:   if N.isBot(c) return 0
3:   if (u = 0) return 0
4:   if (u == 1) return α(N.αPost(s)(c))
5:   return null
```

Fig. 13. Implementation of $\mathsf{NDDPost}(s)$; s is a numeric statement.

To implement leq, we use the fact that for any two formulas u, and v, u implies v (i.e., u is less than v) iff $u \wedge \neg v$ is unsatisfiable. We use bddNot for the negation, and reduce to check unsatisfiability. Formally,

$$\mathsf{leq}(u, v) \triangleq \mathsf{reduce}(\mathsf{meet}(u, \mathsf{bddNot}(v))) = \mathbf{0}\,.$$

The implementation of $\alpha\mathsf{Post}_N(s)$ is similar to reduce. It uses ctxApply to apply the numeric transformer of s to every path of a BDD. For a purely numeric statement s, we define a function $\mathsf{NDDPost}(s)(u, c)$ as shown in Fig. 13. Assuming that $NumV$ is the set of all numeric BDD variables, $\alpha\mathsf{Post}_N$ is defined as follows:

$$\alpha\mathsf{Post}_N(s)(u) \triangleq \mathsf{ctxApply}(u, \mathsf{NDDPost}(s), N.\mathsf{top}, NumV)\,.$$

Note that in this case, ctxApply existentially quantifies all numeric terms, and NDDPost adds the result of transforming them.

Recall that in NDD predicate and numeric terms share BDD variables. This complicates the implementation of the "numeric and predicate" abstract transformer. Specifically, it is not possible to reduce $(e?\tau_N) \wedge \tau_P$ to a sequential composition (as in Section 4). Part of the BDD that is affected by τ_P may be needed for application of τ_N. We solve this problem by adding special "shadow" BDD variables to represent predicate terms during the computation of the transformer. The transformer is implemented in three steps: first, τ_P is applied with its result stored in "shadow" variables, second τ_N is applied eliminating variables changed by τ_P, third the state is restored from the shadow variables. Let τ_P be a predicate transformer of the form $\bigwedge_i p_i := \mathit{choice}(t_i, f_i)$. Let R be the relational semantics of τ_P (as defined in Section 4). Let $V = NumV \cup \{p_i\}_i$ be the set of all numeric variables and all variables changed by τ_P. Then, $\alpha\mathsf{Post}(\tau_N \wedge \tau_P)(u)$ is defined as:

$$\mathsf{unprime}(\mathsf{ctxApply}(u \sqcap R, \mathsf{NDDPost}(\tau_N), N.\mathsf{top}, V))$$

We further elaborate on the definition: the $u \sqcap R$ part corresponds to partial application of τ_P, ctxApply applies τ_N and eliminates all current-state variables in V, and unprime copies shadow variables into current state.

Example 14. For example, let V_P be $\{(x = 3), (x = 4)\}$, V_N be $\{x\}$, τ_N be $x{:=}x{+}1$, and τ_P be $(x = 4){:=}\mathit{choice}(x = 3, \mathsf{f})$. Assume that u is $(x = 3) \wedge (x \geq 3)$. Then, applying τ_P partially results in $(x = 3) \wedge (x \geq 3) \wedge (x = 4)'$; applying τ_N and eliminating $(x = 3)$ produces $(x \geq 4) \wedge (x = 4)'$, and renaming yields $(x \geq 4) \wedge (x = 4)$. □

Theorem 5. NDD *implements* NUMPREDDOM.

Proof. Follows from the above definitions. □

5.5 Summary

In summary, we (informally) compare our four implementations with respect to six criteria: precision, i.e., ability to represent different abstract values; succinctness, i.e., conciseness of representation; performance of the data structure when used solely for predicate (PA) or numeric abstraction (NA); and efficiency of propositional (i.e., meet, join), and numeric operations. The results are shown in Table 3.

NDD is the most precise domain. Furthermore, since it uses BDDs to encode the propositional structure of the value, it is more succinct than NEX and MTNDD that do not share storage between predicate and numeric parts of the abstract value. Succinctness of NEXPOINT is a side-effect of its imprecision.

All of the data-structures reduce to BDDs when there are no numeric terms present. Thus, they are all equally well suited for predicate abstraction. NEXPOINT and NEX represent numeric abstract value explicitly and, therefore, are efficient for numeric abstraction. Both MTNDD and NDD encode numeric values symbolically and introduce additional overhead.

NDD is the best data structure for propositional operations since those are implemented directly using BDDs. At the same time, it is the worst for numerical operations — those use ctxApply, whose complexity is linear in the number of paths in a diagram. Again, the efficiency of NEXPOINT is a by-product of its imprecision.

As shown by our informal comparison, there is no clear winner between the four choices. In the next section, we evaluate the data structures empirically in the context of software model checking.

6 Empirical Evaluation

To evaluate our data-structures, we have build a general reachability analysis engine for C programs. The engine is implemented in JAVA. In addition to the four NUMPREDDOM implementations described in Section 5, we have also implemented a traditional abstract interpreter, referred to as "Numeric", and traditional predicate abstraction, referred to as "Predicate". Note that both "Numeric" and "Predicate" domains are implemented as instances of NUMPREDDOM. Moreover, our NEXPOINT domain corresponds to the typical combination of PA and NA as suggested in [12,4,5]. Thus, our experiments compare our new technique against the standard abstraction interpretation-based approach, the standard predicate abstraction approach, and standard combination of predicate and numeric domains.

All experiments were done on a 2.4GHz machine with 4GB of RAM. In the rest of this section, we describe our implementation and experimental results.

6.1 Implementation

For our experiments, we implemented a tool that checks for the reachability of a control flow location $ERROR$ in a program $Prog$ by using the following general strategy.

1. Initially, one of our six implementations of NUMPREDDOM is selected with $V_P = V_N = \emptyset$. Let us denote this implementation by NPD.
2. Each statement s in $Prog$ is converted to the abstract transformer $\alpha^\tau(s)$. This yields an abstract program $\widehat{Prog}$. For an expression e, let $Approx(e)$ denote the weakest formula over V_P whose interpretation implies e. We implemented α^τ as follows, where $Approx(e)$ is computed using a theorem prover, using the same algorithm as in the SLAM tool [3]:
   - $\alpha^\tau(\mathit{assume}(e)) \triangleq \mathit{assume}(e \wedge \neg Approx(\neg e))$. Note that: (i) we overapproximate e in terms of V_P by first underapproximating $\neg e$, and then negating the result, and (ii) the abstract transformer obtained by applying α^τ to $\mathit{assume}(e)$ is of the form $\mathit{assume}(e')$ where e' is a Boolean expression over $V_N \cup V_P$.
   - $\alpha^\tau(v := e) \triangleq \tau_N \wedge \tau_P$ where:
     (a) $\tau_N \triangleq \bigwedge_{v_i \in V_N} v_i = e_i$ where $e_i = e$ if $v_i = v$ and $e_i = v_i$ otherwise, and
     (b) $\tau_P \triangleq \bigwedge_{p_i \in P_N} p_i := \mathit{choice}(t_i, f_i)$ such that:

$$t_i \triangleq Approx(WP_i) \wedge WP_i \text{ , and}$$
$$f_i \triangleq Approx(\neg WP_i) \wedge \neg WP_i$$

     where WP_i is the weakest precondition [3] of $\gamma(p_i)$ with respect to $v := e$. Note that t_i and f_i are Boolean expressions over $V_N \cup V_P$.
   - Let $s \triangleq s_1; s_2$. Then $\alpha^\tau(s) \triangleq \alpha^\tau(s_1); \alpha^\tau(s_2)$.
   - Let $s \triangleq s_1 \vee s_2$. Then $\alpha^\tau(s) \triangleq \alpha^\tau(s_1) \vee \alpha^\tau(s_2)$.
3. An inductive invariant is computed for $\widehat{Prog}$ using standard abstract interpretation with NPD, and iterative fixed point computation. If the invariant at $ERROR$ is found to be $NPD.\mathsf{bot}$, then $ERROR$ is declared to be unreachable and the procedure terminates.
4. A trace CE exhibiting the reachability of $ERROR$ in $\widehat{Prog}$ is constructed by replaying the abstract interpreter backwards, using a method analogous to that of Gulavani et al. [14]. Next, the satisfiability of the weakest-precondition of CE is checked. If the weakest precondition is found to be satisfiable, then CE corresponds to a concrete execution of $Prog$. In this case, $ERROR$ is declared to be reachable, and the procedure terminates.

	Precision	Succinct	PA	NA	Prop Op	Num Op
NEXPOINT	-	+ +	+	+	++	+ +
NEX	+	-	+	+	-	+ +
MTNDD	+	-	+	-	+	-
NDD	+ +	+	+	-	++	- -

Table 3. Summary of the implementations; **Precision** = precision of abstract values; **Succinct** = succinctness of the representation; **PA** = applicability to predicate abstraction; **NA** = applicability to numeric abstraction; **Prop Op** = complexity of propositional operations (meet, join, etc.); **Num Op** = complexity of numeric operations.

(a)
```
int x = 0;
while (x < C) ++x;
assert(x == C);
```

(b)
```
n = 1;
if(x0 < 0) n = 0; ...
else if(xC < 0) n = 0;
if(x0 < 0) assert(n == 0); ...
else if(xC < 0) assert(n = 0);
```

Fig. 14. Two templates for synthetic examples.

5. Otherwise, NPD is "refined" by adding new numeric variables or predicates via the following simple scheme.
   - Construct an UNSAT-core of the weakest precondition of CE.
   - If a numeric variable in the UNSAT core is not present in V_N, add it to V_N and repeat from Step 1.
   - Else, if a boolean expression in the UNSAT core is absent in V_P, add it to V_P and repeat from Step 1.
   - Else, the overall procedure terminates with failure.

We used the APRON package for numeric reasoning (in our experiments we used the Polyhedra domain), a JAVA implementation of BDDs, and CVCLITE for building the PA part of the abstraction and for analyzing counterexamples.

6.2 Synthetic Examples

NEX and MTNDD join numeric constraints, but NDD maintains an exact union. Thus, we conjecture that NDD performs poorly when numeric joins are exact. To validate this hypothesis we experimented with the template shown in Fig. 14(a). Our experiments support this hypothesis. NEX and MTNDD scale beyond `C = 10000` (NEX performs better than MTNDD since it does not have the extra overhead of manipulating BDDs). NDD blows up even for `C = 400`.

Our second conjecture was that when a problem requires a propositionally complex invariant, the sharing capability of NDD will place it at an advantage to NEX and MTNDD. To test this conjecture we experimented with the template in Fig. 14(b). Our experiments support this hypothesis as well. NDD requires seconds for `C = 10` while NEX and MTNDD both require several minutes with NEX being the slowest.

6.3 Realistic Examples

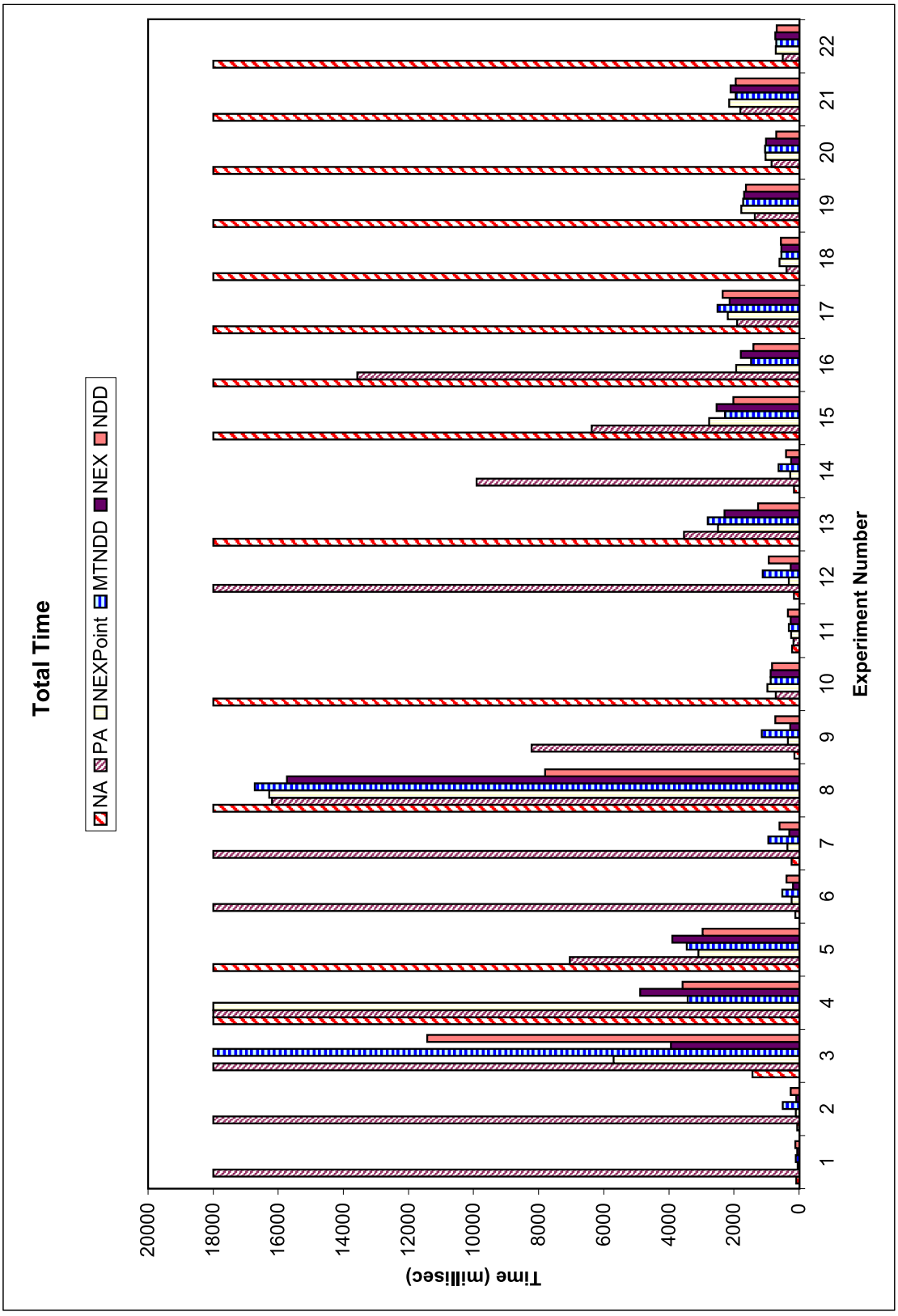
For a more realistic evaluation, we used a set of 22 benchmarks (3 from a suite by Zitser et al. [23], 2 from OpenSSL version 0.9.6c, 9 based on a controller for a metal casting plant, 2 based on the Micro-C OS version 2.72, and 6 based on Windows device drivers). We analysed them using our four implementations of NUMPREDDOM and also with PA and NA separately.

Fig. 15 shows the total time taken by each individual experiment. Since the goal of the experiments is to explore the difference between our data structures, we only report the time taken by the last iteration of abstraction-refinement and do not include the time needed to find a suitable abstraction. Each run was limited to 60 seconds. In the figure, a time of 18 seconds indicates failure, either due to memory exhaustion, or because our simple abstraction-refinement scheme failed to add new variables or predicates. Fig. 15 shows exactly which examples could be analyzed by each domain. In particular, only 9 could be analyzed numerically, and 17 using predicates. In the case of PA, the maximum number of predicates was 10; in the case of NA, the maximum number of numeric variables was 17; in the combined domains, these were at 8 (with 6 for NDD) and 17, respectively. Thus, combining PA and NA requires fewer predicates, with fewest predicates required for the most expressive combination.

In Table 4, we show the number of examples analyzed and the time used by basic abstract operations. The total time includes *all* of the analysis, including predicate abstraction with CVCLITE. Note that the last 4 columns of the table correspond to operations inside the reachability computation (they do not add up to total time). The experiments indicate that a combination of PA and NA is more expressive, and more importantly, more efficient, than either one in isolation. In particular, all of the combined domains could not only solve more problems than PA, but were 6-7 times faster. For this evaluation, NDD performs the best (NEXPOINT solves only 21/22 problems), which is probably explained by lack of deep loops in the benchmarks. The two extremes are NEX and NDD: NEX transformers are efficient to apply, but its join is rather slow, while the opposite is true for NDD.

7 Conclusion

In this article, we have presented an approach to couple PA and NA tightly into a unified analysis framework via a sin-



[t]

Fig. 15. Bar-chart showing total time taken by each experiment.

Domain	Num	Total	γ	join	αPost	Apply
Numeric	9	2.52	0.43	0.41	0.44	0.38
Predicate	17	333.38	0.05	0.03	0.20	0.06
NEXPOINT	21	42.30	0.38	1.13	4.04	8.50
NEX	22	45.17	0.59	2.22	3.99	7.20
MTNDD	22	94.05	0.02	3.71	2.11	56.10
NDD	22	42.15	0.03	0.02	1.96	17.81

Table 4. Time requirements for various operations on realistic examples. Numeric = purely numeric analysis; Predicate = purely predicate analysis; **Num** = no. of examples analysed; Apply = applying abstract transformers. All times are in seconds.

gle abstract domain called NUMPREDDOM. We develop and evaluate four data structures that implement NUMPREDDOM but differ in their expressivity and internal representation and algorithms. 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. Employing these data structures in an industrial setting requires extending automated abstraction-refinement to them. We used a simple refinement strategy for our preliminary experiments. In the future, we plan to further explore the spectrum of possibilities in this area.

Acknowledgements. Preliminary versions of many of the ideas discussed in this work have appeared in [17] and [16]. We are grateful to anonymous referees of LFM'08, FMCAD'08, and STTT for helping improve the presentation and technical clarity of this paper.

References

1. R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. "Algebraic Decision Diagrams and Their Applications". *Formal Methods in System Design (FMSD)*, 10(2/3):171–206, 1997.
2. T. Ball, A. Podelski, and S. K. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '01)*, vol. 2031 of *Lecture Notes in Computer Science*, pp. 268–283, Genova, Italy, April 2001. Springer-Verlag.
3. T. Ball and S. K. Rajamani. "Automatically Validating Temporal Safety Properties of Interfaces". In M. B. Dwyer, editor, *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, vol. 2057 of *Lecture Notes in Computer Science*, pp. 103–122, Toronto, Canada, May 19–20, 2001. New York, NY, May 2001. Springer-Verlag.
4. D. Beyer, T. A. Henzinger, and G. Théoduloz. "Lazy Shape Analysis". In T. Ball and R. B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV '06)*, vol. 4144 of *Lecture Notes in Computer Science*, pp. 532–546, Seattle, WA, August 17–20, 2006. New York, NY, August 2006. Springer-Verlag.
5. D. Beyer, T. A. Henzinger, and G. Theoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, vol. 4590 of *Lecture Notes in Computer Science*, pp. 504–518, Berlin, Germany, July 2007. Springer-Verlag.
6. 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 *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*, pp. 196–207, San Diego, CA, June 9–11, 2003. New York, NY, June 2003. Association for Computing Machinery.
7. R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers (TC)*, 35(8):677–691, August 1986.
8. T. Bultan, R. Gerber, and C. League. "Composite model-checking: verification with type-specific symbolic representations". *ACM Transactions on Software Engineering Methodology (TOSEM)*, 9(1):3–50, January 2000.
9. R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. "Computing Predicate Abstractions by Integrating BDDs and SMT Solvers". In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD '07)*, pp. 69–76, 2007.
10. P. Cousot and R. Cousot. "Systematic Design of Program Analysis Frameworks". In *Proceedings of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '79)*, pp. 269–282, San Antonio, TX, January 1979. Association for Computing Machinery.
11. P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". *Journal of Logic and Computation (JLC)*, 2(4):511–547, August 1992.
12. J. Fischer, R. Jhala, and R. Majumdar. "Joining dataflow with predicates". In *Proceedings of the 13th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '05)*, pp. 227–236, Lisbon, Portugal, September 5–9, 2005. New York, NY, September 2005. Association for Computing Machinery.
13. S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, vol. 1254 of *Lecture Notes in Computer Science*, pp. 72–83, Haifa, Israel, June 22–25, 1997. New York, NY, June 1997. Springer-Verlag.
14. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. "Automatically Refining Abstract Interpretations". In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, vol. 4963 of *Lecture Notes in Computer Science*, pp. 443–458, Budapest, Hungary, March–April 2008. Springer-Verlag.
15. S. Gulwani and A. Tiwari. "Combining Abstract Interpreters". In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI '06)*, pp. 376–386, Ottawa, Ontario, Canada, June 11–14, 2006. New York, NY, June 2006. Association for Computing Machinery.
16. A. Gurfinkel and S. Chaki. "Combining Predicate and Numeric Abstraction for Software Model Checking". In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD '08)*, pp. 127–135, Portland, OR, November 2008. IEEE Computer Society.

17. A. Gurfinkel and S. Chaki. “Combining Predicate and Numeric Abstraction for Software Model Checking (EXTENDED ABSTRACT)”. In K. Y. Rozier, editor, *Proceedings of the 6th NASA Langley Formal Methods Workshop (LFM ’08)*, pp. 47–49, Langley, MD, May 2008.
18. H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. “Using Statically Computed Invariants Inside the Predicate Abstraction and Refinement Loop”. In T. Ball and R. B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV ’06)*, vol. 4144 of *Lecture Notes in Computer Science*, pp. 137–151, Seattle, WA, August 17–20, 2006. New York, NY, August 2006. Springer-Verlag.
19. S. Lahiri, R. Nieuwenhuis, and A. Oliveras. “SMT Techniques for Fast Predicate Abstraction”. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV ’06)*, vol. 4144 of *Lecture Notes in Computer Science*, pp. 424–437, Seattle, August 2006. Springer-Verlag.
20. L. Mauborgne and X. Rival. “Trace Partitioning in Abstract Interpretation Based Static Analyzers”. In S. Sagiv, editor, *Proceedings of the 14th European Symposium On Programming (ESOP ’05)*, vol. 3444 of *Lecture Notes in Computer Science*, pp. 5–20, Edinburgh, UK, April 2005. Springer-Verlag.
21. K. McMillan. “Lazy Abstraction with Interpolants”. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV ’06)*, vol. 4144 of *Lecture Notes in Computer Science*, pp. 123–136, Seattle, August 2006. Springer-Verlag.
22. J. B. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. “Difference Decision Diagrams”. In J. Flum and M. Rodríguez-Artalejo, editors, *Proceedings of Computer Science Logic 1999*, vol. 1683 of *Lecture Notes in Computer Science*, pp. 111–125, Madrid, Spain, September 1999. Springer-Verlag.
23. M. Zitser, R. Lippmann, and T. Leek. “Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code”. In *Proceedings of the 12th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE ’04)*, pp. 97–106, Newport Beach, CA, October 31–November 5, 2004. New York, NY, October–November 2004. Association for Computing Machinery.