Combining Predicate and Numeric Abstraction for Software Model Checking

Arie Gurfinkel and Sagar Chaki
Software Engineering Institute, Carnegie Mellon University
{arie,chaki}@sei.cmu.edu

***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.**

I. INTRODUCTION

Predicate abstraction (PA) [2] and Abstract Interpretation (AI) with numeric abstract domains, called Numeric abstraction (NA) [5], are two mainstream techniques for automatic program verification. 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, e.g., 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 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,c_2 are numeric 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, e.g., 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, e.g., for the code fragment in Fig. 1(a).

```
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)

Fig. 1. Two example programs.

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 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. Achieving an effective combination of PA and NA is the subject of our paper.

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. We propose four data-structures — NEXPOINT, NEX, MTNDD and NDD — that implement NUMPREDDOM. The data structures (summarized in Table I) differ in their expressiveness and in the choice of representation for the numeric part of the domain. Our target is PA-based software analysis. Thus, all of the data-structures allow for efficient (symbolic) propositional reasoning. We present experimental results on non-trivial examples and compare and contrast

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 I
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).

```
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)

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

(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 $q \triangleq (A[x_1 + x_2] = 3)$.

between the four data-structures on the basis of these results. Our experiments show that the proposed combination is more powerful and more efficient than either PA or NA alone.

The rest of the paper is structured as follows. We survey related work in Section II and review background material in Section III. In Section IV, we present the interface of NUMPREDDOM. In Section V, we describe the particularities of each of our NUMPREDDOM implementations. Finally, experimental results and conclusions are presented in Section VI.

II. RELATED WORK

The problem of combining PA and NA involves combining their abstract domains, and is well studied in AI [9]. A typical solution is to combine the domains using a domain combinator such as direct, reduced [8], [9], or logical [12] products. The result can be further extended with disjunctions (or unions) using a disjunctive completion [9]. The domains we develop in this paper are variants of (disjunctive completion of) reduced product between domains of PA and NA.

One approach for combining abstract domains is to combine results of the analyses – e.g., by using light-weight 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. For instance, Jain et al.[14] 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 [5], that use different abstract domains to abstract distinct program variables. Recently, a similar approach has been incorporated into software model-checker BLAST [10], [4], [3] 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. [7] 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 [6] for propositional reasoning and the Omega library 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) [15] 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 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. A preliminary version of this work has appeared in [13].

III. BACKGROUND

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

Expressions and Statements. Let V denote the set of program variables, and E denote the set of expressions over V. A program is built out of statements S of the form: (1) an assignment $l := e$, where l is a variable in V and e is an expression in E, and (2) *assume*(e), where e is in E. Assume operations are used to model conditional branches. We write $||s||$ to denote the collecting semantics, or strongest post-condition transformer, as a function from E to itself. For example, $||x{:=}x{+}1||(x > 3) = (x > 4)$, $||x{:=}5||(x = 3 \wedge y = 6) = (x = 5 \wedge y = 6)$ and $||assume(x > 4)||(y = 6) = (x > 4 \wedge y = 6)$. Atomic statements can be composed in several ways: (a) sequentially, written $s_1; s_2$, meaning s_1 followed by s_2; (b) with alternative choice, written $s_1 \vee s_2$, meaning non-deterministic choice between s_1 and s_2, and (c) in parallel,

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

γ	$: A \to E$	α	$: E \to A$
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	$: A \times A \to A$

$$\alpha\text{Post} : S \to (A \to A)$$

Requires:

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

$\text{true} \Rightarrow e \Rightarrow \gamma(\alpha(e))$ $\quad (\alpha\text{Post}(s)(a) = b) \Rightarrow ||s||(x) \Rightarrow y$

$\text{leq}(a, b) \Rightarrow (a \Rightarrow b)$ $\quad (\text{meet}(a, b) = c) \Rightarrow (x \wedge y \Rightarrow z)$

$\text{isTop}(a) \Rightarrow (\text{true} \Rightarrow a)$ $\quad (\text{join}(a, b) = c) \Rightarrow (x \vee y \Rightarrow z)$

$\text{isBot}(a) \Rightarrow (a \Rightarrow \text{false})$ $\quad (\text{widen}(a, b) = c) \Rightarrow (x \vee y \Rightarrow z)$

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

Name	Notation	Abstract Elements
Intervals	$\text{BOX}(V)$	$\{c_1 \le v \le c_2 \mid c_1, c_2 \in \mathcal{N}, v \in V\}$
Octagons	$\text{OCT}(V)$	$\{\pm v_1 \pm v_2 \ge c \mid c \in \mathcal{N}, v_1, v_2 \in V\}$
Polyhedra	$\text{PK}(V)$	linear inequalities over V
Predicates	$\text{PRED}(V)$	propositional formulas over V

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

written $s_1 \wedge s_2$, meaning parallel synchronous execution of s_1 and s_2. The usual rules and restrictions of legal compositions apply. For example, we do not allow for a parallel composition $x := 5 \wedge x := 6$ since both statements change x, etc.

Abstract Domain. We assume that the reader is familiar with abstract interpretation and only give the necessary details. For a detailed overview, please consult [9]. In this paper, we view an abstract domain operationally as an abstract data type that satisfies the interface $\text{ABSDOM}(V)$ shown in Fig. 3. For simplicity, we assume that the concrete domain is the set of expressions E, and not, for example, program states. We use A to denote the set of all the elements of $\text{ABSDOM}(V)$. The interface consists of functions: α and γ to convert between expressions and abstract elements in A; meet and join correspond to conjunction (intersection) and disjunction (union), respectively; leq corresponds to implication (subset); isTop and isBot check for validity (universality), and unsatisfiability (emptiness), respectively; widen is a widening operator [9] that over-approximates a disjunction and guarantees convergence when applied to any (possibly infinite) sequence of abstract elements; and, αPost approximates the semantics of a program statement as an abstract transformer, i.e., a function from A to A.

Examples of several abstract domains are shown in Table II. The first three domains, collectively called Numeric, are used to represent and manipulate arithmetic constraints. The last one represents propositional formulas over a set of predicates.

Syntax for Abstract Transformers. For ease of understanding, our syntax for abstract transformers mirrors that of concrete program statements. Let $\text{NDOM}(V)$ be a numeric domain over variables V. The syntax for assign transformers of $\text{NDOM}(V)$ is $x_1 := e_1 \wedge \cdots \wedge x_n := e_n$, where all x_i are in V, and all e_i are linear arithmetic expressions. The syntax for conditional transformers of $\text{NDOM}(V)$ is *assume*(e).

For the predicate domain $\text{PRED}(P)$ over a set of predicates P, an abstract transformer is represented by a *Boolean* assignment of the form $p :=$ *choice*(t, f), where $p \in P$ is a predicate, and t and f are Boolean expressions over P. Informally, t represents the condition forcing p to be true, and f the condition forcing p to be false. For example, $p :=$ *choice*$(p, \neg p)$ leaves p unchanged (p is true iff it was true before), $p :=$ *choice*$(\text{false}, \text{false})$ changes p non-deterministically (nothing forces p to be only true or false), and $p :=$ *choice*$(p \wedge q, \text{false})$ leaves p as true if q is true, otherwise p is changed non-deterministically. Formally, the semantics of a Boolean assignment is a forward image (post) over the relation $(p' \wedge \neg f) \vee (\neg p' \wedge \neg t)$, where p' is the value of p in the next state. Boolean assignments can be composed in parallel using conjunction of their relations, as usual. For numeric abstraction, abstract transformers are computed by the domain itself. For predicate abstraction, the transformer is constructed using a theorem prover [11].

BDDs. Reduced Ordered Binary Decision Diagrams (BDDs) [6] are a canonical representation of propositional formulas. A BDD is a DAG whose nodes correspond to propositional variables, and paths to all satisfying assignments of a formula. We use $\mathbf{0}$ and $\mathbf{1}$ to denote BDDs for true and false, respectively. For a BDD u, we use $\text{varOf}(u)$ for the root variable, $\text{bddT}(u)$ for the then-branch, and $\text{bddE}(u)$ for the else-branch of u, respectively. BDDs have efficient support for conjunction (bddAnd), disjunction (bddOr), negation (bddNot), if-then-else (bddIte), existential quantification (bddExists), and variable renaming (bddPermute). Many of these can be implemented using $\text{bddApply}(f, u, v)$, where u, v are BDDs, and f is a binary operator (i.e., conjunction, disjunction, etc.) that is defined only for constants.

IV. NUMPREDDOM: INTERFACE

In this section, we describe the interface of NUMPREDDOM and its supported transfer functions. NUMPREDDOM deals with propositional formulas over predicates and numeric constraints. A numeric constraint can be treated as both a numeric and a predicate term. For example, in the formula $p \wedge (x \ge 0) \wedge (y \ge 0)$, p is definitely a predicate, but so can be $(x \ge 0)$ and $(y \ge 0)$. Let V_P be a set of predicates, V_N a set of numeric variables, and e be a conjunctive expression. The *propositional projection* of e onto V_P, denoted by $\text{proj}_P(V_P, e)$, is a conjunction of predicates from V_P that is implied by (i.e., over-approximates) e. Similarly, the *numeric projection* of e onto V_N, denoted by $\text{proj}_N(V_N, e)$, is a conjunction of numeric constraints over V_N that is implied by e. Some examples of the projections are:

$$\text{proj}_P(\{p\}, p \wedge (x \ge 0) \wedge (y \ge 0)) = p$$
$$\text{proj}_P(\{x \ge 0\}, p \wedge (x \ge 0) \wedge (y \ge 0)) = (x \ge 0)$$
$$\text{proj}_N(\{y\}, p \wedge (x \ge 0) \wedge (y \ge 0)) = y \ge 0$$

Note that the exact definitions of proj_P and proj_N are implementation dependent. We implement them via approximations based on syntactic reasoning. However, more precise semantic constructions via the use of theorem provers is also possible. Such implementation choices affect the efficiency vs. precision trade off, but not correctness.

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

α_P	$: E \to A$	α_N	$: E \to A$
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, S, and A are as in Fig. 3.

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. The interface NUMPREDDOM has two types of variables: numeric, V_N, and propositional, V_P. Moreover, the domain is extended with "primed" variables $V'_P \triangleq \{p' \mid p \in V_P\}$. Additional functions provided by the interface are: α_N, α_P are restrictions of the abstraction function α to conjunction of numeric and propositional expressions, respectively; exists existentially quantifies propositional variables from an abstract value and must satisfy the over-approximation condition: $(\exists V \cdot \gamma(a)) \Rightarrow \gamma(\mathsf{exists}(V, a))$; unprime renames all "primed" variables into the corresponding unprimed ones; $\alpha\mathsf{Post}_N$ lifts an abstract numeric only transformer to the combined domain. Finally, the interface has a special operation, called reduce, that refines an abstract value by sharing information between propositional and numeric parts of the value. Note that it is possible to apply reduce at any time during analysis to increase precision of the result. However, since excessive calls to reduce are expensive, we have factored it out in the interface.

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

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

NUMPREDDOM is distinguished by its support for a rich set of abstract transformers. The grammar for the supported transformers is shown in Fig. 5. We now describe each type of transformer, illustrate in what situations it is required, and provide a common implementation when applicable.

Numeric. Written as $x_1 := e_1 \wedge \cdots \wedge x_k := e_k$, where the variables in x_i and e_i are in V_N. It is handled by $\alpha\mathsf{Post}_N$ of each implementation of NUMPREDDOM. It is a basic building block for abstracting arithmetic transformations.

Assume. Written as *assume*(e), where e is an arbitrary expression, and interpreted as $\lambda X \cdot \mathsf{meet}(\alpha(e), X)$. It is used to approximate program conditionals with a combination of predicate and numeric conditions. For example, in the presence of aliasing, the C program statement *assume*$(*p > 0)$ can be approximated by:

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

with predicates $V_P = \{p = \&x, p = \&y\}$ and numeric variables $V_N = \{x, y\}$.

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

Fig. 5. 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.

Conditional. Written as $e?\tau$, where e is an arbitrary expression, and τ is a purely numeric transformer. It is interpreted as: $\lambda X \cdot \alpha\mathsf{Post}_N(\tau)(\alpha\mathsf{Post}(\textit{assume}(e))(X))$. It is most useful in a combination with other transformers. For example, it is used to abstract an assignment $*p := e$ through a pointer as:

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

with $V_P = \{p = \&x, p = \&y\}$ and $V_N = \{x, y\}$ and variables in e.

Predicate. Written as: $p_1 := \textit{choice}(t_1, f_1) \wedge \cdots \wedge p_n := \textit{choice}(t_n, f_n)$, where p_i are in V_P and t_i and f_i are expressions over V_P and V_N. It is interpreted using conjunction and existential quantification:

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

This transformer is the basic building block for predicate abstraction. It depends on both predicate and numeric information. For example, suppose that $V_P = \{y > 0, p = \&x, p = \&y\}$ and $V_N = \{x\}$. Then the assignment $*p{:=}x$ is abstracted as:

$$(y > 0) := \textit{choice}((p = \&x) \wedge (y > 0), (p = \&y) \wedge (x > 0))\,.$$

Numeric and Predicate. Written as a parallel composition of conditional numeric and predicate transformers: $(e?\tau_N) \wedge \tau_P$, where e is an arbitrary expression, τ_N is a purely numeric transformer, and τ_P is a predicate transformer. It is interpreted with the help of the following equivalence: $(e?\tau_N) \wedge \tau_P \equiv \textit{assume}(e); \tau_P; \tau_N$. That is, since the purely numeric transformer 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. For example, 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:

$$(y = 1) := \textit{choice}(x = 1, x \neq 1) \wedge (y = 1)?x = v : x = w\,.$$

Note that the predicate $y = 1$ is both influenced by numeric constraints on x and influences the next value of x.

Sequential and Non-Deterministic. Written as $\tau_1; \tau_2$ and $\tau_1 \vee \tau_2$, respectively. Interpreted using function sequencing and join operator, respectively:

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

As a complete example, a combined predicate and numeric abstraction of the program in Fig. 2(a) is shown in Fig. 2(b). Both predicates p and q are necessary to separate different paths through the control flow, and predicate q gets its value from a combination of constraints on numeric variables and predicate p.

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.

V. NUMPREDDOM: IMPLEMENTATIONS

In this section, we describe four implementations of NUMPREDDOM. We use N to denote the set of abstract values of the underlying numeric domain over V_N. 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; and, write $X.\mathsf{top}$, $X.\mathsf{bot}$ to mean $X.\alpha(\mathsf{true})$ and $X.\alpha(\mathsf{false})$, respectively. Our domains, NEXPOINT, NEX and MTNDD, share the following definition of reduce: $\mathsf{reduce}(v) \triangleq (\alpha(\gamma(v)))$. Therefore, we only define reduce specifically for (NDD) implementation. In addition, all four implementations share the same definition of proj_N and proj_P based on syntactic simplification of expressions to a normal form.

A. NEXPOINT: *Numeric Explicit Points*

The set of abstract values of NEXPOINT is $2^{2^{V_P}} \times N$. A NEXPOINT value is a pair (p, n) where p is a BDD and n is a numeric abstract value. In particular, $\text{NEXPOINT}.\mathsf{top} = (P.\mathsf{top}, N.\mathsf{top})$ and $\text{NEXPOINT}.\mathsf{bot} = (P.\mathsf{bot}, N.\mathsf{bot})$. The exists and unprime operations are performed on the BDDs. The other operations are performed pointwise. Thus, we have the following definitions:

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

where $\mathsf{op} \in \{\mathsf{meet}, \mathsf{join}, \mathsf{widen}\}$. Recall that reduce is defined as $\alpha(\gamma(v))$. Suppose we have two predicates $q \triangleq (x = 0)$ and $r \triangleq (y = 0)$, where x is also a numeric variable. Then, $\mathsf{reduce}(q \vee r, x = 3 \wedge y \geq 0) = (\neg q \wedge r, x = 3 \wedge y = 0)$. Similarly, $\mathsf{reduce}(q \vee r, x = 3 \wedge y < 0) = \text{NEXPOINT}.\mathsf{bot}$.

B. NEX: *Numeric Explicit Sets*

Each abstract value of the NEX domain is a function $2^{V_P} \mapsto N$. We represent an abstract value as a set of pairs $\{(p_1, n_1), \ldots, (p_k, n_k)\} \subseteq 2^{2^{V_P}} \times N$, where each p_i is a BDD, each n_i is a numeric abstract value, and the following conditions hold:

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

$$\forall 1 \leq i < j \leq k \,.\, n_i \neq n_j \quad \textbf{(C2)} \quad \wedge \quad p_i \sqcap p_j = P.\mathsf{bot} \quad \textbf{(C3)}$$

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. In particular, $\text{NEX}.\mathsf{top} = \{(P.\mathsf{top}, N.\mathsf{top})\}$ and $\text{NEX}.\mathsf{bot} = \emptyset$. Conditions **C1**–**C3** 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** — that the elements of a NEX value are "mutually disjoint".

To understand the NEX operations, we first introduce a normalizing procedure called norm. Given any set $v \subseteq 2^{2^{V_P}} \times N$ satisfying **C3**, norm returns a set $u \subseteq 2^{2^{V_P}} \times N$ that satisfies **C1**–**C3** by performing the following: (i) replacing any $(p_1, n) \in v$ and $(p_2, n) \in v$ with $(p_1 \sqcup p_2, n)$, and (ii) removing every $(p, n) \in v$ such that $p = P.\mathsf{bot} \vee n = N.\mathsf{bot}$. Thus, $\mathsf{norm}(v)$ is a NEX value that is semantically equivalent to v. norm has linear complexity since it makes single pass over its input. The exists and unprime are performed on the BDDs, followed by normalization. The other operations are defined as follows:

$$\alpha_N(e) \triangleq \langle (P.\mathsf{top}, N.\alpha(e)) \rangle \qquad \alpha_P(e) \triangleq \langle (P.\alpha(e), N.\mathsf{top}) \rangle$$

$$\gamma(\langle (p_1, n_1), \ldots, (p_k, n_k) \rangle) \triangleq \bigvee_{1 \leq i \leq k} P.\gamma(p_i) \wedge N.\gamma(n_i)$$

Let $v = (p, n)$ be a NEXPOINT value and $v' = \{(p'_1, n'_1), \ldots, (p'_k, n'_k)\}$ be a NEX value. We say that $v \sqsubseteq v'$ iff $p \sqsubseteq \bigsqcup_{\{i \mid n \sqsubseteq n'_i\}} p'_i$. For any two NEX values $v = \{(p_1, n_1), \ldots, (p_k, n_k)\}$ and v', $\mathsf{leq}(v, v')$ iff $\forall 1 \leq i \leq k \,.\, (p_i, n_i) \sqsubseteq v'$.

$$\mathsf{meet}(v, v') \triangleq \mathsf{norm}(\{(p \sqcap p', n \sqcap n') \mid (p, n) \in v \wedge (p', n') \in v'\})$$

$$\mathsf{join}(v, v') \triangleq \mathsf{norm}(\mathsf{NEXJoin}(v, v'))$$

$$\mathsf{widen}(v, v') \triangleq \mathsf{norm}(\mathsf{NEXWiden}(v, v'))$$

The algorithm NEXJoin is defined recursively as follows: (i) $\mathsf{NEXJoin}(\emptyset, v) = v$, (ii) $\mathsf{NEXJoin}(v, \emptyset) = v$, and (iii) $\mathsf{NEXJoin}(\{(p, n)\} \cup X, \{(p', n')\} \cup X') = \{(p \sqcap p', n \sqcup n')\} \cup \mathsf{NEXJoin}(\{(p \sqcap \neg p', n)\}, X') \cup \mathsf{NEXJoin}(\{(p' \sqcap \neg p, n')\}, X) \cup \mathsf{NEXJoin}(X, X')$. 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 widen instead of join. The meet, join and widen operations have quadratic complexity. Finally, the operation $\alpha\mathsf{Post}_N$ is defined as follows:

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

C. MTNDD: *Multi-Terminal Numeric Decision Diagrams*

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

Conceptually, an MTNDD value is a Multi-Terminal BDD [1] whose terminals are numeric abstract values from N. In practice, we simulate MTBDDs with BDDs. We associate

```
1: BDD MJoinOp (BDD u, BDD v)
2:   if (u = 1 ∨ v = 1) return 1
3:   if (u = 0) return u
4:   if (v = 0) return v
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. 6. Implementation of MJoinOp.

a BDD variable with each predicate and numeric term, and restrict variable ordering to ensure that predicate variables always appear before 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 variables: one predicate, one numeric. Although there are infinitely many numeric terms, only finitely many are used in any analysis. Thus, we allocate variables for numeric terms dynamically.

We use algorithms toBdd and toExpr to convert between BDDs and expressions in the usual way. Note that for NEXPOINT and NEX, this was achieved via $P.\alpha$ and $P.\gamma$. For a BDD v, we use $\mathsf{isNum}(v)$ to determine whether the root variable of v is a numeric term. 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. Thus:

$$\alpha_P(t) \triangleq \mathsf{toBdd}(t) \quad \alpha_N(t) \triangleq \mathsf{toBdd}(N.\gamma(N.\alpha(t)))$$
$$\gamma(v) \triangleq \mathsf{toExpr}(v)$$

The unprime operation is the same as its BDD version. The MTNDD.exists operation is similar to bddExists except that MTNDD.join is used instead of bddOr.

The operations meet, join, widen, leq, and $\alpha\mathsf{Post}_N$ are implemented as operators to bddApply. They work by (a) recursively traversing input BDD(s) until they are reduced to BDDs over numeric terms; (b) converting numeric BDDs to abstract values and applying the corresponding numeric operation; and (c) encoding the result back as a BDD. For example MTNDD.join is implemented using $\mathsf{bddApply}(\mathsf{MJoinOp}, u, v)$, where MJoinOp is shown in Fig. 6. Note that the constraint on the variable ordering ensures that whenever a root of a BDD v is numeric, the rest of v is numeric as well.

Since MTNDD operations are implemented using bddApply, their complexity is linear in the size of their input BDDs. Due to sharing between various BDDs, the memory (and hence time) requirement of MTNDD is expected to be better than NEX.

D. NDD*: Numeric Decision Diagrams*

NDD is our most expressive domain, with elements in $2^{2^{V_P} \times N}$. An NDD value is a BDD representing a propositional formula over predicate and numeric terms. With each term t, we associate a BDD variable. The association takes negation into account. Any two terms t_1 and t_2 that complement each other, i.e., $t_1 = \neg t_2$, are associated with the opposite phases of the same BDD variable. For example, whenever $x > 0$ is mapped to a BDD variable v, $x \leq 0$ is mapped to $\neg v$. We use $\mathsf{term}(v)$ 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. Each term t is allocated a single BDD variable, independently of whether t is a predicate, a numeric term, or both. Thus, propositionally inconsistent expressions are always reduced to $\mathbf{0}$, unlike in the previous three implementations. For example, if $p \triangleq (x > 0)$ is a predicate, then $p \wedge (x \leq 0)$ is reduced to $\mathbf{0}$.

```
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. 7. Implementation of ctxApply.

For the most part, 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. Additionally, bddNot is used to over-approximate negation. That is, whenever v over-approximates an expression e, $\mathsf{bddNot}(v)$ over-approximates $\neg e$.

All of the above operations work on propositional structure of the abstract value. Effectively, they treat numeric constraints as uninterpreted propositional symbols. Their complexity is linear in the size of the input. The operations reduce, leq, and αPost treat numeric terms differently. For these operations, we introduce a function ctxApply, whose implementation is shown in Fig. 7. The function ctxApply recursively traverses a BDD, collecting the context of the current path in c, and existentially eliminating variables in V. The complexity of this operation is linear in the number of paths in a BDD.

The reduce operation is implemented as $\mathsf{ctxApply}(u, \mathsf{reduceOp}, N.\mathsf{top}, \emptyset)$, where $\mathsf{reduceOp}(u, c)$ returns $\mathbf{0}$ when $N.\mathsf{isBot}(c)$ or $u = \mathbf{0}$, returns $\mathbf{1}$ when $u = \mathbf{1}$, and returns **null** otherwise. Essentially, it replaces every unsatisfiable cube in a BDD with $\mathbf{0}$. In particular, for any unsatisfiable BDD v, $\mathsf{reduce}(v)$ is $\mathbf{0}$. For 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. Since both satisfiability and negation are available, we implement leq as $(\mathsf{reduce}(\mathsf{meet}(u, \mathsf{bddNot}(v))) = \mathbf{0})$.

The implementation of $\alpha\mathsf{Post}_N(s)$ is similar to reduce. It uses ctxApply to apply a transformer of s to every path

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

TABLE III
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.

of an input BDD. For a purely numeric statement s, we first define a function $\mathsf{NDDPost}(s)(u, c)$ such that it returns $\mathbf{0}$ if $N.\mathsf{isBot}(c)$ or $u = \mathbf{0}$, returns $N.\alpha\mathsf{Post}(s)(c)$ if $u = \mathbf{1}$, and returns **null** otherwise. Second, let $NumV$ be the set of all numeric BDD variables. Then, $\alpha\mathsf{Post}_N(s)(u) \triangleq \mathsf{ctxApply}(u, \mathsf{NDDPost}(s), N.\mathsf{top}, NumV)$.

In this domain, predicate and numeric terms share BDD variables. Thus, parallel composition $\tau_N \wedge \tau_P$ of a numeric (τ_N) and a predicate (τ_P) transformers cannot be reduced to a sequential composition (as in Section IV). Part of the BDD that is affected by τ_P may be needed for application of τ_N. To solve this, we implement the transformer by first applying τ_P partially by storing its result in "shadow" variables, then applying τ_N while eliminating variables changed by τ_P, and finally restoring the state from the shadow variables. Let τ_P be 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 IV), and $V = NumV \cup \{p_i\}_i$ 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:

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

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. 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 $(x \geq 4) \wedge (x = 4)$.

E. Summary

To summarize our four implementations, we compare them with respect to six different 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 III.

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.

VI. EMPIRICAL EVALUATION AND CONCLUSION

We implemented a general reachability analysis engine for C programs in Java on top of the four implementations of NUMPREDDOM. 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. We experimented with two types of examples: (a) synthetic examples designed to compare and contrast our four implementations of NUMPREDDOM with each other, and (b) examples derived from more realistic benchmarks. For the synthetic examples, we only compare NEX, MTNDD and NDD since NEXPOINT is less expressive. All our experiments were carried out on a 2.4 GHz machine with 4 GB of RAM.

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. 8(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. 8(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.

Realistic Examples. For a more realistic evaluation, we used a set of 22 benchmarks (3 from a suite by Zitser et al. [16], 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. The results are summarized in Table IV. The total time taken by each individual experiment is shown

```
(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. 8. Two templates for synthetic examples.

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 IV
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.

in Fig. 9. For the experiments, we have implemented a simple abstraction-refinement scheme based on the analysis of an UNSAT-core of the WP of an infeasible counterexample. First, the scheme adds all of the numeric variables in the UNSAT core; second, the predicates in the core are added when the first step fails to eliminate the spurious counterexample. Since the goal of the experiments is to explore the difference between our data structures, we only report on the time taken by the last iteration of abstraction-refinement, and do not include the time needed to find a suitable abstraction. Not all examples could be analyzed by every 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 less predicates, with fewer predicates required for the most expressive combination.

In Table IV, we show the number of examples analyzed, as well as 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, then either one in isolation. In particular, all of the combined domains could not only solve more problems that 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.

In summary, we have presented an approach to couple PA and NA tightly into a unified analysis 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. 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.

REFERENCES

[1] R. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. "Algebraic Decision Diagrams and Their Applications". *FMSD*, 10(2/3), 1997.

[2] T. Ball and S. K. Rajamani. "Automatically Validating Temporal Safety Properties of Interfaces". In *Proc. of SPIN*, 2001.

[3] D. Beyer, T. A. Henzienger, and G. Theoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In *CAV*, 2007.

[4] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Lazy Shape Analysis". In *CAV*, 2006.

[5] 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.

[6] R. Bryant. "Graph-based Algorithms for Boolean Functions Manipulation". *IEEE Transactions on Computers*, 35(8), 1986.

[7] T. Bultan, R. Gerber, and W. Pugh. "Composite Model Checking: Verification with Type-Specific Symbolic Representations". *TOSEM*, 9(1), 2000.

[8] P. Cousot and R. Cousot. "Systematic Design of Program Analysis Frameworks". In *POPL'79*, 1979.

[9] P. Cousot and R. Cousot. "Abstract Interpretation Frameworks". *JLC*, 2(4), 1992.

[10] J. Fischer, R. Jhala, and R. Majumdar. "Joining dataflow with predicates". In *FSE*, 2005.

[11] S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In *CAV*, 1997.

[12] S. Gulwani and A. Tiwari. "Combining Abstract Interpreters". In *PLDI*, 2006.

[13] A. Gurfinkel and S. Chaki. "Combining Predicate and Numeric Abstraction for Software Model Checking (EXTENDED ABSTRACT)". In *LFM*, 2008.

[14] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. "Using Statically Computed Invariants Inside the Predicate Abstraction and Refinement Loop". In *CAV*, 2006.

[15] J. B. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. "Difference Decision Diagrams". In *CSL*, 1999.

[16] M. Zitser, R. Lippmann, and T. Leek. "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code". In *FSE*, 2004.

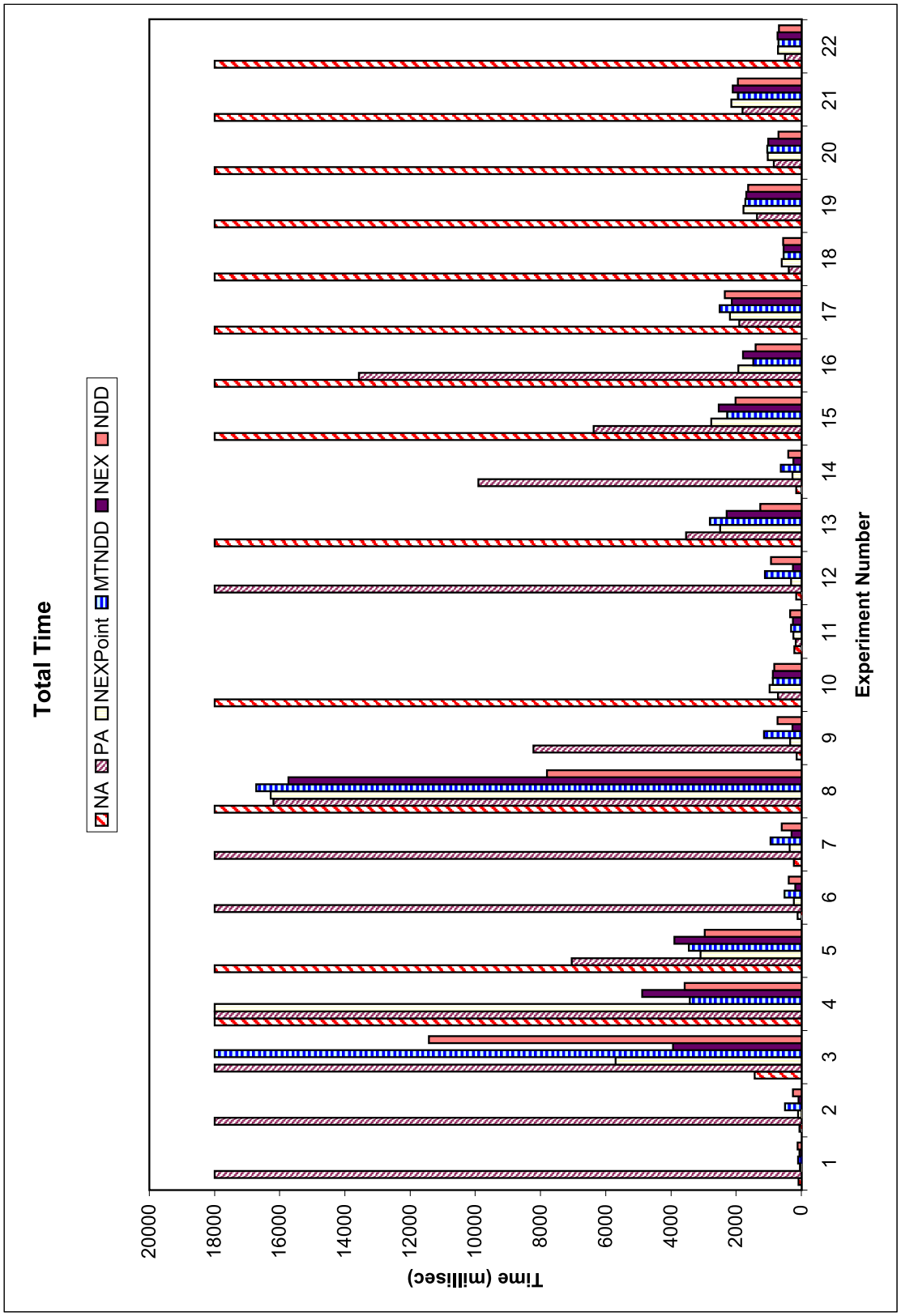

Fig. 9. Bar-chart showing total time taken by each experiment.