Synthesizing Predicates from Abstract Domain Losses

Bogdan Mihaila and Axel Simon

Technical University of Munich, Garching b. München, Germany
{firstname.lastname}@in.tum.de

Abstract Numeric abstract domains are key to many verification problems. Their ability to scale hinges on using convex approximations of the possible variable valuations. In certain cases, this approximation is too coarse to verify certain verification conditions, namely those that require disjunctive invariants. A common approach to infer disjunctive invariants is to track a set of states. However, this easily leads to scalability problems. In this work, we propose to augment a numeric analysis with an abstract domain of predicates. Predicates are synthesized whenever an abstract domain loses precision due to convexity. The predicate domain is able to recover this loss at a later stage by re-applying the synthesized predicates on the numeric abstract domain. This symbiosis combines the ability of numeric domains to compactly summarize states with the ability of predicate abstraction to express disjunctive invariants and non-convex spaces. We further show how predicates can be used as a tool for communication between several numeric domains.

1 Introduction

Verification by means of a reachability analysis is based on abstract domains that over-approximate the possible concrete states that a program can reach. The forte of abstract domains is their ability to synthesize new invariants that are not present in the program. However, their inherent approximation may mean that the invariant required to verify a program cannot be deduced. On the contrary, the strength of predicate abstraction used in software model checking is that predicates precisely partition the state space of a program. The challenge here is to synthesize new predicates that eventually suffice to verify a program. This work combines the benefits of both approaches: we synthesize new predicates by observing the precision loss in numeric domains and refine the precision of the numeric domains using the predicates. Our technique is particularly useful for expressing non-convex invariants that are commonly lost when using off-the-shelf numeric abstract domains that are based on convex approximations.

The importance of non-convex invariants is illustrated by the C code in Fig. 1. Here, line 1 computes a flag `f` that is true if the divisor `d` of the expression in line 5 is non-zero. Assuming that the initial value of `d` lies in $[-2, 2]$, the possible values when evaluating the conditional are shown in Fig. 1b). Abstracting this set of discrete points using, say, the abstract domain of intervals yields the state

a)
```
1 f = d!=0;
2 ...
3 if (f) {
4   assert(d!=0);
5   y = x / d;
6 }
```

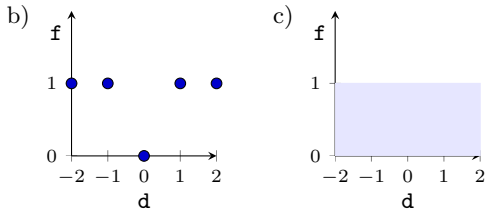

Figure 1. Avoiding a division by zero.

in Fig. 1c). This state space is too imprecise to deduce that `d` is non-zero if `f` is one. As a consequence, testing that f is one in line 3 does not restrict the abstract state sufficiently to show that the assertion holds.

Interestingly, analyzing the same example using predicate abstraction does not suffer from this imprecision as non-convex spaces can naturally be represented using disjuncts. In the example, a predicate $p_f \equiv d \neq 0$, which is equivalent to the disjunction $d \leq -1 \vee d \geq 1$, suffices to verify the assertion since testing `f` in line 3 results in p_f being true.

A common approach to enriching numeric abstract domains to allow expressing non-convex states is to use disjunctive completion [4], that is, a set of states. In particular, several works have proposed to some variant of a binary decision-diagram (BDD) where decision nodes are labeled with predicates and the leaves are abstract domains [10,14]. A similar effect is obtained by duplicating the control flow graph (CFG) for each subset of satisfied predicates [6,15,17]. In both settings, the number of numeric domains that are tracked may be exponential in the number of predicates. Our work improves over this setup by combining classic predicate abstraction [1] with a single numeric domain, thereby avoiding this exponential duplication of the numeric state. In particular, we present a generic *combinator domain* that is parameterized over any numeric abstract domain and allows any predicate expressible by the abstract domain. We thereby also generalize over bespoke domains that explicitly track specific disjunctive information, such as disequalities [16]. Overall, we make the following contributions:

- We propose an abstract domain that tracks implications between two predicates. By combining this domain with a single numeric state, we retain the performance and simplicity of the numeric state transformers.
- We present an effective reduction mechanism that refines a numeric state based on the implications in the predicate domain.
- By observing precision losses in the numeric domain, relevant predicates are synthesized that preempt a loss of precision during the computation of a join. This novel mechanism addresses precision losses due to convexity without a costly replication of the numeric state.

The remainder of this paper is organized as follows: after presenting the setup of our domains and necessary notation, Sect. 3 defines the transfer functions of the predicate domain and the reduction with numeric states. Section 4 details the lattice operations and shows how new implications can be synthesized by the numeric domain. Section 5 presents experiments, related work and conclusions.

$$
\begin{array}{lll}
\mathit{Pred} & ::= \mathit{Test} & \mathit{Lin} \quad ::= c_1 x_1 + \ldots + c_n x_n \\
\mathit{Test} & ::= \mathit{Lin} \bowtie \mathit{Lin} & \mathit{NonLin} ::= \mathit{Lin} \boxtimes \mathit{Lin} \\
\mathit{Assign} & ::= x = \mathit{Expr} & \bowtie \quad ::= \leq | \not\leq | < | \not< | = | \neq \\
\mathit{Expr} & ::= \mathit{Lin} \mid \mathit{NonLin} \mid \mathit{Test} & \boxtimes \quad ::= \times \mid / \mid \% \mid \hat{} \\
\end{array}
$$

Figure 2. The grammar decorating a control flow graph (CFG)

2 Preliminaries

Our analysis operates on the control flow graph (CFG) of a program. The CFG is represented by a set of vertices labeled $v_1, v_2, \ldots$ and a set of directed edges representing the transfer functions. The transfer functions are either assignments $v_i \xrightarrow{\mathit{Assign}} v_j$ or assumptions $v_i \xrightarrow{\mathit{Pred}} v_j$ where *Assign* and *Pred* are given by the grammar in Fig. 2. Additionally, we use assertions in programs, e.g. `assert(x == 0)` that correspond to edges $v_i \xrightarrow{x \neq 0} v_e$ to a designated error node v_e. We associate each vertex v_i with an abstract state $d_i \in \mathcal{D}$ where $\mathcal{D}$ is the universe of a lattice $\langle \mathcal{D}, \sqsubseteq_{\mathcal{D}}, \sqcup_{\mathcal{D}}, \sqcap_{\mathcal{D}}, \top_{\mathcal{D}}, \bot_{\mathcal{D}} \rangle$. Initially the states are $d_0 = \top_{\mathcal{D}}$ and $d_i = \bot_{\mathcal{D}}$ for $i \neq 0$. The solution to the program analysis problem is characterized by a set of constraints $d_j \sqsupseteq_{\mathcal{D}} [\![l_i^j]\!]^{\mathcal{D}}(d_i)$, each constraint corresponding to an edge $v_i \xrightarrow{l_i^j} v_j$. It can be inferred using chaotic iteration which picks indices i, j for which the constraint is not satisfied and, for the edge from v_i to v_j updates d_j to $d_j := d_j \sqcup_{\mathcal{D}} [\![l_i^j]\!]^{\mathcal{D}}(d_i)$. In general, the lattice $\mathcal{D}$ may have infinite ascending chains. We therefore assume that each cycle in the CFG contains at least one application of the widening operator ∇ in order to ensure termination [4].

2.1 The *Predicate* Abstract Domain

We present our predicate domain as a co-fibered domain [20], that is, as a domain that is parameterized by another domain. Due to an implementation [3] in OCaml, such a domain is also called a functor domain. A co-fibered domain $\mathcal{D}$ is parameterized by a child domain $\mathcal{C}$ that it controls. Their combination is written as $\mathcal{D} \rhd \mathcal{C}$ and a state as a tuple $\langle d, c \rangle \in \mathcal{D} \rhd \mathcal{C}$. A transfer function on $\mathcal{D} \rhd \mathcal{C}$ may apply any number of transfer functions on its child $c \in \mathcal{C}$ before returning a result. Co-fibered domains may be nested. For instance, we combine the predicate domain $\mathcal{P}$ with a co-fibered affine equality domain $\mathcal{A}$ [18] and a plain interval domain $\mathcal{I}$, yielding a stack of domains $\mathcal{P} \rhd \mathcal{A} \rhd \mathcal{I}$ where a state $\langle \bar{\iota}, \langle a, i \rangle \rangle$ contains the individual domain states $\bar{\iota} \in \mathcal{P}$, $a \in \mathcal{A}$ and $i \in \mathcal{I}$. The predicate domain is given by the lattice $\langle \mathcal{P} \rhd \mathcal{C}, \sqsubseteq_{\mathcal{P}}, \sqcup_{\mathcal{P}}, \sqcap_{\mathcal{P}} \rangle$ where the universe $\mathcal{P} : \wp(\mathit{Pred} \times \mathit{Pred})$ is a finite set of implications $p_1 \to p_2$ over predicates $p_i \in \mathcal{L}(\mathit{Pred})$ as defined in Fig. 2. Predicates relate linear expressions over the program variables X using a comparison operator $\bowtie$. Note that the set of operators is closed under negation so that the universe of predicates is closed under negation. The choice of implications between only two predicates allows for a simple yet effective propagation of information, as detailed in the next section.

$[\![x = a \bowtie b]\!]^{\mathcal{P}} \langle \bar{\iota}, c\rangle = \langle \bar{\iota}', [\![x = a \bowtie b]\!]^{\mathcal{C}} c\rangle$
where $\bar{\iota}' = \{p \to q \in \bar{\iota} \mid x \notin vars(p) \cup vars(q)\}$
$\cup \{x = 1 \to a \bowtie b, x = 0 \to a \not\bowtie b, a \bowtie b \to x = 1, a \not\bowtie b \to x = 0\}$
$[\![x = NonLin]\!]^{\mathcal{P}} \langle \bar{\iota}, c\rangle = \langle \bar{\iota}', [\![x = NonLin]\!]^{\mathcal{C}} c\rangle$
where $\bar{\iota}' = \{p \to q \in \bar{\iota} \mid x \notin vars(p) \cup vars(q)\}$
$[\![x = Lin]\!]^{\mathcal{P}} \langle \bar{\iota}, c\rangle = \langle \bar{\iota}', [\![x = Lin]\!]^{\mathcal{C}} c\rangle$
where $\bar{\iota}' = \{p \to q \in \bar{\iota} \mid x \notin vars(p) \cup vars(q)\}$
$\cup \{transform(p \to q) \mid p \to q \in \bar{\iota}\}$ and $\sigma = [x/Lin]$
and $transform(p \to q) = \begin{cases} \sigma^{-1}(p) \to \sigma^{-1}(q) & \text{if } \sigma^{-1}(p) \wedge \sigma^{-1}(q) \text{ exists} \\ true \to true & \text{otherwise} \end{cases}$
$[\![a \bowtie b]\!]^{\mathcal{P}} \langle \bar{\iota}, c\rangle = \langle \bar{\iota}, fixapply(\{a \bowtie b\}, \emptyset, c)\rangle$
where $fixapply(\bar{p}, \bar{u}, c') =$ if $\bar{p} \subseteq \bar{u}$ then c' else
let $t \in \bar{p} \setminus \bar{u}$ and $\bar{n} = \{t\} \cup consequences^{\mathcal{C}}(t, c')$
and $\bar{n}' = \{q \mid p \to q \in \bar{\iota} \wedge n \in \bar{n} \wedge n \vdash p\} \cup \{\neg p \mid p \to q \in \bar{\iota} \wedge n \in \bar{n} \wedge n \vdash \neg q\}$
in $fixapply(\bar{p} \cup \bar{n}', \bar{u} \cup \{t\}, [\![t]\!]^{\mathcal{C}} c')$

Figure 3. Assignments and branch transfer functions for the predicates domain. The comparison operator $\bowtie$ in a predicate is one of $\leq, \not\leq, <, \not<, =, \neq$.

3 Transfer Functions and Reductions

This section details the transfer functions and presents the flow of information between the predicate domain and the numeric child domains.

3.1 Transfer Functions

The transfer functions of the combined domain state $\langle \bar{\iota}, c\rangle \in \mathcal{P} \rhd \mathcal{C}$ are given in Fig. 3. In general, a transfer function $[\![l]\!]^{\mathcal{P}} \langle \bar{\iota}, c\rangle$ applies the corresponding transfer function on the child domain $c \in \mathcal{C}$, yielding $\langle \bar{\iota}', [\![l]\!]^{\mathcal{C}} c\rangle$ where $\bar{\iota}'$ is the new state of the predicate domain. We distinguish three forms of assignments. The first, $[\![x = a \bowtie b]\!]^{\mathcal{P}}$, assigns the result of a comparison to a variable x. Here, the predicate domain removes any predicate that mentions x and adds new predicates based on the comparison. We assume that x is set to one if test $a \bowtie b$ holds and to zero otherwise. Thus, the predicates $x = 0$ and $x = 1$ are used to encode the value of x in the implications. Specifically, the two outcomes $x = 1 \leftrightarrow a \bowtie b$ and $x = 0 \leftrightarrow a \not\bowtie b$ are stored using four implications.

The transfer function $[\![x = NonLin]\!]^{\mathcal{P}}$ for non-linear assignment removes all implications in the predicate domain containing x. An assignment $[\![x = Lin]\!]^{\mathcal{P}}$ of a linear expression to x tries to transform implications containing x if Lin contains x, e.g. `x=x+1`. For example, consider the predicates state $\bar{\iota} = \{f = 0 \to x \leq 5, x \not\leq 10 \to y = 10\}$ and the assignment `x=x+1` mentioned above. Given the substitution $\sigma = [x/x + 1]$ that describes the change of the state space, we compute $\sigma^{-1} = [x/x-1]$ that describes how predicates can be transformed so that they are valid in the new state. In the example, applying σ^{-1} to the implications yields $\bar{\iota}' = \{f = 0 \to x \leq 6, x \not\leq 11 \to y = 10\}$. In all three assignments, more predicates can be retained by testing if they are still valid after the assignment.

We now consider the transfer function for an assumption $[\![a \bowtie b]\!]^{\mathcal{P}}$. The information from the test $a \bowtie b$ is used by the predicate domain to gather further facts about the state. The process of applying these facts to the child domain is called reduction [4]. The reduction is performed as a fixpoint computation and can be seen as an instance of Granger's framework for reduction by local iteration [8]. Specifically, the function *fixapply* gathers a set of deduced predicates $\bar{p}$ and a set of predicates $\bar{u}$ that have already been used. In each iteration a predicate $t \in \bar{p} \setminus \bar{u}$ is applied to the child state c', yielding $[\![t]\!]^{\mathcal{C}} c'$. Furthermore, a set of new predicates that are implied by t are computed in two steps. First, t is combined with a set $\bar{n}$ of semantic consequences which is computed by $consequences^{\mathcal{C}}$ as detailed below. Second, a set of syntactically implied predicates $\bar{n}'$ is computed from $\bar{n}$ by inspecting the implications in the predicate domain. We use modus ponens resolution to deduce q from an implication $p \to q \in \bar{\iota}$ where $t \vdash p$ and deduce $\neg p$ if $t \vdash \neg q$. Here, the syntactic entailment $\vdash$ is defined as follows:

Definition 1 (Syntactic Predicate Entailment $\vdash$). *A predicate q is entailed by another predicate p, written as $p \vdash q$, if $p \equiv q$ or if p describes a weaker condition that subsumes the condition of q. We use the following syntactic entailment rules (analogous definitions for the negations of the comparison operators $\not\leq, \not<$):*

$$p \vdash x \neq c \quad \textit{if} \quad p \in \{x = c' \mid c' \neq c\} \cup \{x \leq c' \mid c' < c\} \cup \{x < c' \mid c' \leq c\}$$
$$p \vdash x \leq c \quad \textit{if} \quad p \in \{x = c' \mid c' \leq c\} \cup \{x \leq c' \mid c' \leq c\} \cup \{x < c' \mid c' - 1 \leq c\}$$
$$p \vdash x < c \quad \textit{if} \quad p \in \{x = c' \mid c' < c\} \cup \{x \leq c' \mid c' < c\} \cup \{x < c' \mid c' \leq c\}$$

The set of syntactically implied predicates $\bar{n}'$ is added to $\bar{p}$ and, hence, eventually applied to the child state. Since at most two predicates for each implication in $\bar{\iota}$ can be added to $\bar{p}$, this iterative reduction terminates.

Although not strictly necessary, the $consequences^{\mathcal{C}}$ function allows information to flow from the child domain to the predicate domain. The function synthesizes new predicates that become valid after applying the test t. It is different for each child domain. An implementation for the interval domain $\mathcal{I}$ is as follows:

$$consequences^{\mathcal{I}}(t, c) = \text{ let } c' = [\![t]\!]^{\mathcal{I}} c \text{ in } \{x = l \mid c(x) \neq c'(x) \wedge c'(x) \in [l, l]\}$$

Here, $c(x)$ is the interval of the variable x in the state c. The insight in this definition is that the only additional information inferable by the interval domain is that a variable x may have become constant due to a test such as $x \leq c$. Returning these equality predicates may allow additional deductions in the predicate domain. Note that other child domains may deduce different facts.

3.2 Example for the Reduction after Executing Assumptions

We illustrate the reduction when applying an assumption $[\![a \bowtie b]\!]^{\mathcal{P}}$ using an example. Consider applying the test $f < 1$ to the state $s = \langle \bar{\iota}, c \rangle$ that consists of the predicates $\bar{\iota} \in \mathcal{P}$ and the intervals $c \in \mathcal{I}$ as child domain. Let $\bar{\iota} = \{f = 0 \to x \leq 0\}$ and $c = \{f \in [0, 1], x \in [-1, 1]\}$. The first step in the transfer function is to infer the consequences of the test: $\bar{n} = consequences^{\mathcal{I}}(f < 1, c)$. As the child state becomes $c' = \{f \in [0, 0], x \in [-1, 1]\}$, the consequences are

$\bar{n} = \{f = 0\}$. The synthesized predicate in $\bar{n}$ syntactically entails the left-hand side of the implication $f = 0 \to x \leq 0$ that is tracked in the predicate domain. Thus, *fixapply* calls itself recursively with the new predicate $x \leq 0$ which results in a call to $\mathit{consequences}^{\mathcal{I}}(x \leq 0, c') = \emptyset$. Now, the set of implied predicates $\bar{n}'$ is empty and a fixpoint is reached since $\bar{p} = \bar{u} = \{f < 1, x \leq 0\}$. Thus, the result of the transfer function is $[\![f < 1]\!]^{\mathcal{P}} s = \langle \{f = 0 \to x \leq 0\}, \{f \in [0, 0], x \in [-1, 0]\} \rangle$. This recursive reduction mechanism implements all required reductions between the predicate and the child domain. The next section illustrates how this reduction mechanism is used to preempt the loss of precision due to convexity.

3.3 Application to Non-Convex Spaces

Reconsider the example in Fig. 1 where a division by zero is prevented by a guard. The problem here is that the state space for d is non-convex and cannot be expressed with the intervals domain $\mathcal{I}$. However, using the predicate domain $\mathcal{P}$ we are able to prove the invariant at program point 4 even though the interval value for d at that point is $d \in [-2, 2]$. We illustrate an analysis of the program for an initial state where the interval domain tracks d with the value $d \in [-2, 2]$. By executing line 1, the four implications for the assignment of a comparison are added to the predicate domain, yielding the state $\bar{\iota} = \{f = 1 \to d \neq 0, f = 0 \to d = 0, d \neq 0 \to f = 1, d = 0 \to f = 0\}$. On entering the `then`-branch, the test $f = 1$ in line 3 restricts the variable f in the interval domain to $f \in [1, 1]$. The predicate domain uses the first implication to deduce $d \neq 0$, which is also applied to the child domain. However, the child domain $\mathcal{I}$ is not able to express the disjunction $d \in [-2, -1] \vee [1, 2]$ thus the state after applying $d \neq 0$ remains $d \in [-2, 2]$. The assertion in line 4 translates to an edge to the dedicated error node that is labelled with the test $d = 0$. Hence, the assertion fails if $d = 0$ is satisfiable. The predicate domain observes that the right-hand side $d \neq 0$ of the implication $f = 1 \to d \neq 0$ is *false* and thus adds the negated left-hand side $f \neq 1$ to $\bar{n}'$. Once the predicate domain applies $f \neq 1$ to the child state $c = \{f \in [1, 1], d \in [-2, 2]\}$, the result is $\bot$, the unreachable state. Thus, the error node is not reachable in the program and the assertion is verified even though the convex numeric domain is not precise enough to express $d \neq 0$. The reduction mechanism is able to exploit the information in the implications for verifying assertions without requiring more complex (i.e. non-convex) numeric domains.

In general, observing predicates from assignments is only a syntactic technique that may fail for more complex disjunctive invariants. The next section therefore illustrates how the reduction mechanism implemented by *fixapply* naturally combines with a more sophisticated way of inferring new implications.

4 Lattice Operations and Predicate Synthesis

We present entailment test, join and widening operations of the predicate domain. Moreover, we introduce a novel *synth* function that synthesizes new implications between predicates that counteract the loss of precision in numeric domains.

$\langle \bar{\iota}_1, c_1 \rangle \sqsubseteq_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle = c_1 \sqsubseteq_{\mathcal{C}} c_2 \wedge \mathit{entailed}(\bar{\iota}_2, \bar{\iota}_1, c_1) = \bar{\iota}_2$
where $\mathit{entailed}(\bar{\iota}', \bar{\iota}, c) = \{p' \to q' \in \bar{\iota}' \mid (\exists p \to q \in \bar{\iota}. p' \vdash p \wedge q \vdash q') \vee ([\![p']\!]^{\mathcal{C}} c \vDash q')\}$
$\langle \bar{\iota}_1, c_1 \rangle \sqcup_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle = \langle \mathit{join}(\bar{\iota}_1, \bar{\iota}_2) \cup \mathit{synth}^{\mathcal{C}}(c_1, c_2), c_1 \sqcup_{\mathcal{C}} c_2 \rangle$
where $\mathit{join}(\bar{\iota}_1, \bar{\iota}_2) = \mathit{entailed}(\bar{\iota}_1, \bar{\iota}_2, c_2) \cup \mathit{entailed}(\bar{\iota}_2, \bar{\iota}_1, c_1)$

Figure 4. Lattice operations for the predicate domain.

4.1 Lattice Operations

We commence by detailing the entailment test $\langle \bar{\iota}_1, c_1 \rangle \sqsubseteq_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle$ in Fig. 4. It performs the entailment test $c_1 \sqsubseteq_{\mathcal{C}} c_2$ on the child domain and tests if all the implications in the right argument $\bar{\iota}_2$ are entailed by the left argument by calling the function $\mathit{entailed}(\bar{\iota}', \bar{\iota}, c)$. The latter function returns an implication $p' \to q' \in \bar{\iota}'$ if it is either syntactically entailed in $\bar{\iota}$ or semantically entailed in the state c. Semantic entailment $\vDash$ is defined as follows:

Definition 2 (Semantic Predicate Entailment $\vDash$). *A predicate q is entailed in a state c, written $c \vDash q$, if testing $\neg q$ in c yields an empty state, i.e., $[\![\neg q]\!]^{\mathcal{C}} c = \bot$.*

By this definition, the test $[\![p']\!]^{\mathcal{C}} c \vDash q'$ in *entailed* reduces to checking whether $[\![\neg q']\!]^{\mathcal{C}}([\![p']\!]^{\mathcal{C}} c) = \bot$. Thus, if the predicate p' on the left-hand side of the implication $p' \to q'$ is false in c then $[\![\neg q']\!]^{\mathcal{C}} \bot = \bot$ follows and the implication is entailed in c. The two tests $[\![\cdot]\!]^{\mathcal{C}}$ on the child domain c can be avoided if the implication is syntactically entailed by an implication in $\bar{\iota}$. Here, the implication $p \to q \in \bar{\iota}$ entails $p' \to q'$ if the premise p' is stronger and the conclusion q' is weaker which is expressed by $p' \vdash p \wedge q \vdash q'$. Note that neither the syntactic nor the semantic entailment test subsumes the other as both approximate the test differently.

The join $\langle \bar{\iota}_1, c_1 \rangle \sqcup_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle$ independently computes a join on the predicate domain and on the child domain. In oder to join the implication sets $\bar{\iota}_1$ and $\bar{\iota}_2$, we define a function *join* that keeps all implications that hold in the respective other state using the *entailed* function described above. Note that the semantic entailment test in *entailed* is particularly important for the join as one of the predicate domain states may be empty so that the syntactic entailment would discard all implications. The semantic join is able to retain newly inferred predicates in, for example, loop bodies as illustrated later.

In addition to the predicates returned by the *join* function, new implications are synthesized from the child domain states using the $\mathit{synth}^{\mathcal{C}}$ function. The idea is to synthesize implications that characterize the approximation that occurred as part of the $\sqcup_{\mathcal{C}}$ operation. Which synthesized implications are generated depends on the numeric domain. If the predicate language is sufficiently expressive, a domain could potentially characterize all precision losses that occur during a join. The following $\mathit{synth}^{\mathcal{I}}$ function for the interval domain is an example that generates implications for all changing bounds. Moreover, by relating changes of interval bounds between different variables, it generates relational information that cannot be expressed within the interval domain itself. It is defined as follows:

$$
\begin{aligned}
&\mathit{synth}^{\mathcal{I}}(c_1, c_2) = \text{let } c = c_1 \sqcup_{\mathcal{I}} c_2 \\
&\quad \text{and } \bar{m} = \{x \in \mathit{vars}(c_1) \cap \mathit{vars}(c_2) \mid c_1(x) \neq c_2(x)\} \text{ and } i \in \{1,2\} \\
&\quad \text{and } \bar{u_i} = \{u_{xi} \mid x \in \bar{m} \wedge c_i(x) \in [l_{xi}, u_{xi}] \wedge c(x) \in [l_x, u_x] \wedge u_{xi} < u_x\} \\
&\quad \text{and } \bar{l_i} = \{l_{xi} \mid x \in \bar{m} \wedge c_i(x) \in [l_{xi}, u_{xi}] \wedge c(x) \in [l_x, u_x] \wedge l_x < l_{xi}\} \\
&\quad \text{in } \{u_{x1} < x \rightarrow l_{y2} \leq y, u_{y1} < y \rightarrow l_{x2} \leq x \mid x, y \in \bar{m} \wedge u_{xi}, u_{yi} \in \bar{u_i} \wedge l_{xi}, l_{yi} \in \bar{l_i}\}
\end{aligned}
$$

Let $\mathit{vars}(c)$ return all the variables $\bar{x} \subseteq X$ tracked in the state c and let $c(x)$ denote the interval of the variable x. The set of variables $\bar{m}$ that are not equal in both states are those whose joined value is an approximation of the input intervals. For these variables we compute a set of changing lower and upper bounds $\bar{l_i}$ and $\bar{u_i}$ whose indices indicate the variable and origin of the bound. For example, when joining $c_1(x) \in [0, 5]$ with $c_2(x) \in [10, 15]$, resulting in $c(x) \in [0, 15]$, the upper bound $u_{x1} = 5$ of $c_1(x)$ and the lower bound $l_{x2} = 10$ of $c_2(x)$ are lost whereas the other bounds are retained in $c(x)$. These changing bounds are used for generating implications. Specifically, each implication correlates a lost upper bound u_{xi} from c_i with a lost lower bound $l_{y(2-i)}$ from c_{2-i} where $i = 1, 2$. For the example above $x = y$, thus the only generated implication is $u_{x1} < x \rightarrow l_{x2} \leq x$, that is, $5 < x \rightarrow 10 \leq x$. The implication allows that a test such as $7 < x$ is refined to $10 \leq x$, thereby recovering the precision loss in the join that is due to the convexity of the interval domain. In general, the bounds of several variables can be related, thereby even generating relational information.

One drawback of the definition above is that implications are added for each pair of variables from $\bar{m}$, thus, the returned set of implications is quadratic in $|\bar{m}|$. This quadratic growth can be avoided by not generating a redundant implication $a \rightarrow c$ if both $a \rightarrow b$ and $b \rightarrow c$ are already present. Specifically, by sorting $\bar{m}$ using some total ordering, we only emit implications over variables that are adjacent in this ordering, as well as an implication relating the largest variable with the smallest. As the predicate domain performs a transitive closure on application of a test predicate (through *fixapply*), adding only implications between adjacent variables is sufficient to recover all information expressed in a chain of implications. Using this optimization, we are able to reduce the number of synthesized implications to be linear in the number of changed variables $|\bar{m}|$.

Before we consider further examples, we consider the widening operation, defined by, say, $\langle \bar{\iota}_1, c_1 \rangle \nabla_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle = \langle \mathit{join}(\bar{\iota}_1, \bar{\iota}_2) \cup \mathit{synth}^{\mathcal{C}}(c_1, c_2), c_1 \nabla_{\mathcal{C}} c_2 \rangle$. This definition is analogous to the join operation but applies widening on the child states c_1, c_2. One caveat of this definition is that termination is not guaranteed. Consider an implication $p' \rightarrow q'$ at a loop head and assume that a conditional in the loop refines the child state by using the $[\![a \bowtie b]\!]^{\mathcal{P}}$ transformer in Fig. 3 which, in turn, may use the information in $p' \rightarrow q'$. Suppose that joining the two branches of the conditional creates a new implication $p \rightarrow q$ by means of the $\mathit{synth}^{\mathcal{C}}$ function that is syntactically weaker than $p' \rightarrow q'$. If $[\![p']\!]^{\mathcal{C}} c_1 \not\models q'$ (the previous implication cannot be shown to hold in the new state) then the loop is not stable. If furthermore $[\![p]\!]^{\mathcal{C}} c_2 \models q$ (the new implication holds in the previous state), the loop is analyzed with the new implication. Thus, one implication may be replaced by another one, possibly indefinitely so. In order to ensure termination, standard widening techniques can be used, such as eventually

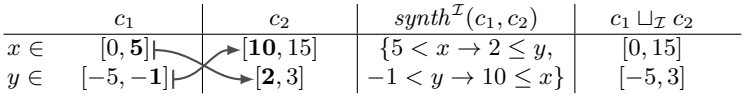

Figure 5. The join of two states in the intervals domain $\mathcal{I}$ and the synthesized implications correlating the bounds lost due to the convex approximation.

disallowing new implications [15]. This can be implemented by using the definition $\langle \bar{\iota}_1, c_1 \rangle \nabla_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle = \langle \mathit{entailed}(\bar{\iota}_1, \bar{\iota}_2, c_2), c_1 \nabla_{\mathcal{C}} c_2 \rangle$ after k iterations. So far, we were unable to find examples that exhibit this non-terminating behavior.

4.2 Recovering Precision using Relational Information

One strength of our $\mathit{synth}^{\mathcal{I}}$ function is that it creates relational information, that is, it generates implications between different variables. This relational information enables *fixapply* to deduce, from a test of one variable, more precise ranges for other variables. In particular, a test t that separates two states, i.e. $[\![t]\!]^{\mathcal{I}} c_1 = c_1$ and $[\![t]\!]^{\mathcal{I}} c_2 = \bot$ is enriched by the relational implications so that all losses due to convexity are recovered, that is, $[\![t]\!]^{\mathcal{P}}(\langle \bar{\iota}_1, c_1 \rangle \sqcup_{\mathcal{P}} \langle \bar{\iota}_2, c_2 \rangle) = \langle \bar{\iota}'_1, c_1 \rangle$.

We illustrate this ability using two states $s_1 = \langle \emptyset, \{x \in [0, 5], y \in [-5, -1]\} \rangle$ and $s_2 = \langle \emptyset, \{x \in [10, 15], y \in [2, 3]\} \rangle$. The joined state $s = s_1 \sqcup_{\mathcal{P}} s_2$ is given by $s = \langle \{5 < x \to 2 \leq y, -1 < y \to 10 \leq x\}, \{x \in [0, 15], y \in [-5, 3]\} \rangle$. This operation is illustrated in Fig. 5 where the bounds in bold are those that are lost and the arrows indicate which bounds are related by the generated implications. We now show how applying the test $0 < y$ on s recovers the numeric state in s_2 and, analogously, that applying $y \leq 0$ recovers the numeric state of s_1. Specifically, when applying the test $0 < y$ on state s, the left-hand side of the implication $-1 < y \to 10 \leq x$ is syntactically entailed, so that $10 \leq x$ is also applied to the child state, yielding the precise value $[10, 15]$ for x. The predicate $10 \leq x$ syntactically entails the other implication $5 < x \to 2 \leq y$. Thus, the predicate $2 \leq y$ is applied to the child state, yielding the precise value $[2, 3]$ for y. After that no new predicates are entailed and the recursive predicate application in the function *fixapply* stops with the state $s'_2 = \langle \{5 < x \to 2 \leq y, -1 < y \to 10 \leq x\}, \{x \in [10, 15], y \in [2, 3]\} \rangle$. Observe that the interval domain is identical to that of s_2. Analogously, we get a state s'_1 in which the interval for x is $[0, 5]$ and for y is $[-5, -1]$ after applying the opposing condition $y \leq 0$.

In summary, the predicate domain improves the precision of a child domain tracking precision losses that are reported by the child. In particular, the domain-specific $\mathit{synth}^{\mathcal{C}}$ function can generate predicates that cannot be expressed in the domain itself. This allows the predicate domain to maintain enough disjunctive information to recover the state before the join whenever a test is able to separate the two states. Note though that there exist cases when this is not completely possible, namely when the value of x in one state overlaps the value in the other state. Consider $c_1(x) \in [0, 4]$ and $c_2(x) \in [2, 8]$. A test $x < 3$ does not separate the two states. However, any test outside the overlapping range $[2, 4]$ is able to separate the two states which, in turn, leads to the refinement of other bounds.

4.3 Application to Path-Sensitive Invariants

This section illustrates how our domain can verify an example taken from [5]. The challenge of analyzing the code in Fig. 6a) is that the join of different paths loses precision and the invariant that a file is only accessed if it was opened before cannot be proved. For the sake of presentation, we use *open* to denote the value of `out->is_open`. Note that the assertion in line 3 can be proved by using the interval domain alone, as *open* is $[1, 1]$ due to line 2. However, the assertion in line 10 cannot be proved by using intervals alone: observe that *open* is set to $[0, 0]$ in line 4 and that the join of this value with the value $[1, 1]$ from line 7 yields the convex approximation of $[0, 1]$ in line 10 of the assertion. As a consequence, the assertion cannot be proved since the edge to the error state with assumption $open = 0$ is satisfiable. Now consider analyzing the example using the predicate domain with the interval domain as child. Then the join of the `then` branch in line 7 and the state before line 6 creates an implication $0 < flag \rightarrow 1 \leq open$. When applying the branch condition $flag = 1$ of line 9, the implied predicate $1 \leq open$ is used to reduce the state, yielding $open \in [1, 1]$ in the interval domain. Thus, the assertion can be proved since the edge to the error state with assumption $open = 0$ is unreachable. The example illustrates how numeric domains may lose precision when joining paths and, thus, may fail to express a path-sensitive invariant which is crucial to prove assertions in the branch of a conditional.

Fischer et al. [5] prove the assertion in line 10 by not joining the states after the conditional in line 6, thus keeping the states where $open = 0$ and $open = 1$ separate. They associate a predicate with a numeric state and join numeric states only if they are associated with the same predicate. Thus, their abstract state before the conditional in line 9 is $\{\langle flag = 0, open \in [0, 0]\rangle, \langle flag = 1, open \in [1, 1]\rangle\}$ which reduces to $\{\langle flag = 1, open \in [1, 1]\rangle\}$ inside the conditional. Although their approach is able to prove the assertion, it is more costly as it tracks several numeric states. Although using sharing can reduce the resource overhead of tracking multiple states [11] the cost of tracking several states is generally higher [14]. Our approach retains the conciseness of a single convex numeric state and merely adds the implications necessary to express certain disjunctive information. In particular, we only infer disjunctive information for variables that actually differ in the join of two numeric states rather than duplicating the information on all variables.

4.4 Application to Separation of Loop Iterations

A particularly challenging example from the literature [12] requires that variable values of certain loop iterations are distinguished. The example in Fig. 6b) is prototypical for a loop that frees a memory region in its last iteration. The assertion in line 4 expresses that the memory region pointed-to by `p` has not yet been deallocated. In order to prove this assertion, an analysis needs to separate the value of the pointer `p` in the last loop iteration from its value in all previous iterations. In particular, the example is difficult to prove using convex numeric

a)

```
1  FILE *out;
2  out->is_open = 1;
3  assert(out->is_open == 1);
4  out->is_open = 0;
5  ...
6  if (flag)
7    out->is_open = 1;
8  ...
9  if (flag)
10   assert(out->is_open == 1);
```

b)

```
1  p = &some_var;
2  n = 5;
3  while (n >= 0) {
4    assert(p != 0);
5    // dereference p
6    ...
7    if (n == 0)
8      p = 0;
9    n--;
10 }
```

Figure 6. Two challenging examples from the literature: a) accessing a file only if it was already opened and b) freeing a pointer in the last loop iteration.

domains due to a precision loss that occurs when joining the point $\langle p, n\rangle = \langle 0, -1\rangle$ at line 10 of the last loop iteration with the earlier states where $p \neq 0$ and $n \geq 0$.

However, using the simple interval numeric domain and our predicate domain, the example is proved using the fixpoint computation detailed in Fig. 7. In step 1 of the table, `p` is initialized to a non-zero address of a variable, which we illustrate by using the value 99. After initializing the loop counter `n` in step 2, the loop is entered as the loop condition `n >= 0` is satisfied. In step 5, it is determined that the `then`-branch in line 8 is not reachable. After decrementing `n`, the state is propagated to the loop head via the back-edge in step 8. At this point, widening is applied. Additional heuristics [15] ensure that the interval $[-1, 5]$ is tried for n, rather than widening n immediately to $[-\infty, 5]$. By applying the loop condition `n >= 0`, a new state for the loop body is obtained in step 9. In step 12, it is observed that the `then`-branch in line 8 is reachable. In the next step in line 9 the states of both branches are joined and the interval domain approximates `p` with $[0, 99]$. In the same step, the implications $0 < n \rightarrow 99 \leq p, 0 < p \rightarrow 0 \leq n$ are synthesized. In step 14 these predicates are transformed using $\sigma^{-1} = [n/n + 1]$. This state is joined with the previous state at the loop header at step 15. Our widening heuristic suppresses widening since a new branch in the program has become live [15]. Since the resulting numeric state has changed due to the new value of `p`, the fixpoint computation continues. Note that during the join in step 15, both implications $-1 < n \rightarrow 99 \leq p, 0 < p \rightarrow -1 \leq n$ are semantically entailed in the current state at the loop head (as computed in step 8') and therefore kept in the joined state. Evaluating the loop condition in step 16 enforces that $n \geq 0$, that is, $0 \leq n$. The latter predicate syntactically entails the predicate $-1 < n$. Thus, the *fixapply* function deduces that $99 \leq p$ holds, yielding $p \in [99, 99]$. The assertion holds since intersecting the state at step 16 with $p = 0$ yields $\bot$. Thus, at line 4, `p` cannot be 0 and the assertion holds. Continuing the analysis of the loop observes a fixpoint in step 22. Note that the assertion can also be shown when using standard widening that sets n to $[-\infty, 0]$ in step 8'. However, for the sake of presentation, we illustrated the example with the more precise states.

step	line		intervals		implications
			p	n	
1	2		$[99, 99]$		
2	3		$[99, 99]$	$[5, 5]$	
3	4		$[99, 99]$	$[5, 5]$	
...			...		...
5	7		$[99, 99]$	$[5, 5]$	
6	9		$[99, 99]$	$[5, 5]$	
7	10		$[99, 99]$	$[4, 4]$	
8	3	$\sqcup$	$[99, 99]$	$[4, 5]$	
8'	3'	∇	$[99, 99]$	$[-1, 5]$	
9	4		$[99, 99]$	$[\mathbf{0}, \mathbf{5}]$	
...			...		...
12	**8**		$[99, 99]$	$[0, 0]$	
13	9	$\sqcup$	$[\mathbf{0}, \mathbf{99}]$	$[0, 5]$	$\{\mathbf{0} < \mathbf{n} \rightarrow \mathbf{99} \leq \mathbf{p}, \mathbf{0} < \mathbf{p} \rightarrow \mathbf{0} \leq \mathbf{n}\}$
14	10		$[0, 99]$	$[-1, 4]$	$\{-1 < n \rightarrow 99 \leq p, 0 < p \rightarrow -1 \leq n\}$
15	3	$\sqcup$	$[0, 99]$	$[-1, 5]$	$\{-1 < n \rightarrow 99 \leq p, 0 < p \rightarrow -1 \leq n\}$
16	4		$[\mathbf{99}, \mathbf{99}]$	$[0, 5]$	$\{\mathbf{-1} < \mathbf{n} \rightarrow \mathbf{99} \leq \mathbf{p}, 0 < p \rightarrow -1 \leq n\}$
...			...		...
22	3	$\sqsubseteq$	$[0, 99]$	$[-1, 5]$	$\{-1 < n \rightarrow 99 \leq p, 0 < p \rightarrow -1 \leq n\}$

Figure 7. States during the analysis of the loop example in Fig. 6b).

benchmark suite	programs	lines	lines avg.	time avg.	time avg. (P)	time avg. (D)
literature	9	9–17	14 ms	38 ms	99 ms	381 ms
test	8	66–274	115 ms	393 ms	1658 ms	-

Figure 8. Evaluation of our implementation. Due to technical reasons, the "test" benchmark suite could not be analyzed using the disjunctive domain (D).

5 Related Work and Evaluation

The Predicate abstract domain was inspired by a weaker domain that tracked bi-implications of the form $f \leftrightarrow x \leq c$ [18]. This domain is useful in the analysis of machine code where conditional branches are encoded using two separate instructions. The first instruction is a comparison that stores the result of $x \leq c$ in a processor flag f. The second instruction is a branch instruction that determined the jump target based on f. By tracking an association between the comparison result f and the predicate $x \leq 0$, the edge of the jump with the assumption $f = 1$ can be made more precise by also assuming $x \leq 0$ and analogously for $f = 0$. However, the use of simple bi-implications only states additional invariants rather than predicates that hold conditionally. Hence, disjunctive information cannot be described by using only bi-implications.

We evaluated our combined predicate/numeric domain on several examples in the literature, including the ones presented in this paper, shown as "literature" in Fig. 8. We also evaluated larger examples shown as "test". All examples from the literature required the predicate domain to verify except for the example in Fig. 1 that our weaker predicate domain [18] already handles. The times are shown when running without and with the predicate domain "(P)". The last column shows the running time with a disjunctive domain "(D)" that tracks different numeric states depending on the index ranges of a loop [6,15]. Due to this, only one example in the "literature" benchmark suite could possibly profit. A precision comparison of our disjunctive and the predicate domain can therefore not be conclusive for the various disjunctive domains in the literature [11,14].

5.1 Related Work

The idea of abstracting a system relative to a set of predicates was first applied by Graf and Saïdi to state graphs created during model checking [7]. This approach has later been generalized to software model checking by Ball et al. [1]. Here, an abstraction tool C2BP translates a C program to a program over Boolean variables. The value of a Boolean variable is true if the corresponding predicate holds in the input C program. The universe of possible predicates is very large as the semantics of each assignment and test is expressed by predicates. For scalability, C2BP abstracts the input C program only with respect to a few predicates. The idea of counter-example driven refinement is to increase this set of predicates by deducing which additional predicates are needed to discharge a verification condition. This deduction is performed on a path through the program on which the current Boolean abstraction is insufficient to prove a verification condition. There are two ways in which this refinement may fail: Firstly, the translation of C statements and tests into predicates may be inaccurate or the logic of the predicates may be insufficient to represent the C semantics precisely. Secondly, a set of predicates that suffices to discharge the verification condition on the chosen path may be insufficient when considering the whole program.

An abstract interpretation over domains that lose precision due to convexity is naturally improved by avoiding the computation of joins. This approach is commonly known as disjunctive completion [4]. In practice, the disjunctions are qualified by a set of predicates and are stored in a binary decision-diagram (BDD) where decision nodes are labeled with predicates and the leaves are convex numeric abstract domains [11,14]. The challenge in implementing these domains is that evaluation of transfer functions in one leaf may lead to a result that has to be propagated to many other leaves. A particular challenge is the widening operator and the reduction between predicates and states [10,15]. One drawback of using a BDD as state is that computing a fixpoint of a loop will perform all operations on each leaf of the BDD, even those that are stable within, say, the current loop. This can be avoided by lifting the fixpoint computation from tracking a map $P \to S$ to $P \times C \to S$ where P are program points, S are states and C is a context. By using the predicates on a path in the decision diagram as context, the whole decision tree can be encoded by using one context per path. The advantage of this encoding is that stable leaves in the original decision tree are no longer propagated since they are each checked for stability by the fixpoint engine [17]. Using predicates as context can be seen as an elegant way of duplicating the CFG which is a technique often used to improve widening [6].

Beyer et al. combine abstract domains with predicates [2]. Their framework associates a precision level Π with each domain that can be adjusted based on observed values in the program. A value-set analysis, for instance, may specify that only variables with less than five values are tracked while a predicate domain will store the set of possible predicates in Π. They propose to change this precision level during the analysis, so that a precision loss in one domain can be met with a precision increase in another. They instantiate their framework by an analysis that switches from tracking value sets to tracking predicates once the former

becomes too expensive. Their states are tuples of the precision levels and the domain states so that a different domain state is tracked for each precision level. Their approach thereby resembles the disjunctive completion approaches discussed earlier. Interestingly, they propose the use of a function *abstract* to synthesize predicates from an abstract state. However, in their implementation it only returns predicates occurring in the current program.

Further afield are techniques to refine abstract interpretations based on counter examples [13,9]. The idea here is to re-run the abstract interpretation once a verification condition cannot be discharged. An improved precision of the abstract interpreter is obtained by improving the widening or the abstract state based on the path of the counter example. Our work can be seen as dual to counterexample-driven refinement as we employ predicates to avoid a precision loss rather than to refine a state that is too coarse. An approach that uses counterexample-driven refinement and which is seemingly close to ours is that of Fischer et al. [5] who propose a domain containing a map from a predicate to a numeric abstract domain. Like our setup, their construction is a reduced cardinal power domain [4] or, more generally, a co-fibered domain [20]. However, since they track one numeric abstract domain for each predicate, there is no bound on the number of states that they infer. Future work should address if their techniques can be incorporated into our abstract domain, that is, if new predicates can be synthesized without duplicating the numeric state.

Interestingly, when state spaces are bounded, disjunctive invariants can be encoded using integral polyhedra [19]. However, since even rational polyhedra are expensive, storing disjunctive information explicitly seems to be preferable.

5.2 Conclusion

We presented a co-fibered domain that tracks implications between predicates. This domain takes a single numeric abstract domain as child and thereby avoids tracking several child domains which is the most prominent way to encode disjunctive information. We illustrated that our domain solves challenging verification examples form the literature while using a simple deduction and reduction mechanism in form of the two novel functions *synth* and *fixapply*, respectively.

References

1. T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic Predicate Abstraction of C Programs. In *Programming Languages, Design and Implementation*, pages 203–213. ACM, 2001.
2. D. Beyer, T. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Automated Software Engineering*, 2008.
3. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Programming Language Design and Implementation*, San Diego, USA, June 2003. ACM.
4. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Principles of Programming Languages*, pages 269–282, San Antonio, Texas, USA, January 1979. ACM.

5. J. Fischer, R. Jhala, and R. Majumdar. Joining Dataflow with Predicates. In M. Wermelinger and H. Gall, editors, *European Software Engineering Conference*, volume 30, pages 227–236. ACM, September 2005.
6. D. Gopan and T. W. Reps. Guided Static Analysis. In H. R. Nielson and G. Filé, editors, *Static Analysis Symposium*, volume 4634 of *LNCS*, pages 349–365, Kogens Lyngby, Denmark, August 2007. Springer.
7. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.
8. P. Granger. Improving the Results of Static Analyses of Programs by Local Decreasing Iterations. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Comp. Sci.*, volume 652 of *LNCS*, pages 68–79. Springer, 1992.
9. B. S. Gulavani and S. K. Rajamani. Counterexample Driven Refinement for Abstract Interpretation. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*, pages 474–488, Vienna, Austria, March 2006. Springer.
10. A. Gurfinkel and S. Chaki. Boxes: A Symbolic Abstract Domain of Boxes. In R. Cousot and M. Martel, editors, *Static Analysis Symposium*, volume 6337 of *LNCS*, pages 287–303. Springer, 2010.
11. A. Gurfinkel and S. Chaki. Combining Predicate and Numeric Abstraction for Software Model Checking. *Software Tools for Techn. Transfer*, 12(6):409–427, 2010.
12. M. Heizmann, J. Hoenicke, and A. Podelski. Software Model Checking for People Who Love Automata. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, volume 8044 of *LNCS*, pages 36–52. Springer, July 2013.
13. R. Leino and F. Logozzo. Loop Invariants on Demand. In K. Yi, editor, *Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 119–134, Tsukuba, Japan, 2005. Springer.
14. L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In M. Sagiv, editor, *European Symposium on Programming*, volume 3444 of *LNCS*, pages 5–20, Edinburgh, UK, April 2005. Springer.
15. B. Mihaila, A. Sepp, and A. Simon. Widening as Abstract Domain. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 170–186, Moffett Field, California, USA, May 2013. Springer.
16. M. Péron and N. Halbwachs. An Abstract Domain Extending Difference-Bound Matrices with Disequality Constraints. In B. Cook and A. Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *LNCS*, pages 268–282, Nice, France, January 2007. Springer.
17. S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta. Static Analysis in Disjunctive Numerical Domains. In Kwangkeun Yi, editor, *Static Analysis Symposium*, volume 4134 of *LNCS*, pages 3–17, Seoul, Korea, August 2006. Springer.
18. A. Sepp, B. Mihaila, and A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In M.Pinzger and D. Poshyvanyk, editors, *Working Conference on Reverse Engineering*, Limerick, Ireland, October 2011. IEEE.
19. A. Simon. Splitting the Control Flow with Boolean Flags. In M. Alpuente and G. Vidal, editors, *Static Analysis Symposium*, volume 5079 of *LNCS*, pages 315–331, Valencia, Spain, July 2008. Springer.
20. A. Venet. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *Static Analysis Symposium*, LNCS, pages 366–382, London, UK, 1996. Springer.