A Binary Decision Diagram Lifted Domain for Analyzing Program Families

Aleksandar S. Dimovski[a]

[a]*Mother Teresa University, Mirche Acev nr. 4, 1000 Skopje, North Macedonia*

Abstract

Many software systems today are highly configurable. They can produce a potentially large variety of related programs (variants) by selecting suitable configuration options (features) at compile time. Recently, specialized *variability-aware* (*lifted*, *family-based*) static analyses based on abstract interpretation have been developed. They allow analyzing all variants of a program family (or, any other configurable software system), simultaneously, in a single run without generating any of the variants explicitly. In effect, they produce precise analysis results for all individual variants. The elements of the underlying lifted analysis domain represent tuples (i.e. disjunction of properties), which maintain one property from an existing single-program analysis domain per variant. Nevertheless, explicit property enumeration in tuples, one by one for all variants, immediately yields to combinatorial explosion given that the number of variants can grow exponentially with the number of features. Therefore, such lifted analyses may be too costly or even infeasible for program families with a large number of variants.

In this work, we propose a more efficient lifted static analysis of program families with Boolean features, where sharing is explicitly possible between analysis elements corresponding to different variants. This is achieved by giving a symbolic representation of the lifted analysis domain, which can efficiently handle disjunctive properties in program families. The elements of the new lifted domain are *binary decision diagrams*, where decision nodes are labeled with Boolean features and leaf nodes belong to an existing single-program analysis domain. The lifted domain is parametric in the choice of the abstract (property) domain for leaf nodes. To illustrate the potential of this representation, we have implemented a lifted static analyzer that uses a combination of forward and backward analyses for inferring numerical invariants and necessary preconditions of C program families. It uses

APRON and BDDAPRON libraries for implementing the new lifted analysis domain. The APRON library, used for the leaves, is a widely accepted API for numerical abstract domains (e.g. polyhedra, octagons, intervals), while the BDDAPRON is an extension of APRON which adds the power domain of Boolean formulae and any APRON domain. An empirical evaluation on C benchmarks taken from SV-COMP and BusyBox indicates that our binary decision diagram-based approach is effective and outperforms the baseline tuple-based approach.

Keywords: Lifted static analysis, Abstract interpretation, Software product lines, Binary decision diagram lifted domain

---

1. Introduction

Highly configurable (variable) software systems are becoming increasingly common today in many application areas. Many software projects adopt the *Software Product Line* (SPL) methodology [1] for building a *family* of similar systems, known as *variants* or *valid products*, from a common code base. Each variant is specified in terms of specially defined Boolean variables called *features* (or, statically configurable options), which are selected (switched on) for that particular variant. The SPL methodology is frequently seen in the development of the embedded software (e.g., cars, phones, avionics), system level software (e.g. Linux kernel), many web solutions (e.g. Drupal, Wordpress), etc. Configurable options (features) are used to either support different application scenarios for embedded components, to provide portability across different hardware platforms and configurations, or to produce variations of products for different market segments or customers. We apply the facilities of the C preprocessor [2] to support such compile-time configurability. They use `#ifdef` preprocessor directives to annotate optional and alternative code fragments, which are included or excluded from a variant in compile-time depending on the selected configuration options.

In many of the application domains, a rigorous verification and formal analysis of program families is of paramount importance. Among the methods included in current practices, static program analysis by abstract interpretation [3, 4] is a powerful technique for automatic verification of software systems. Abstract interpretation [3, 4] is a general theory for approximating the semantics of programs. It provides sound (all confirmative answers are indeed correct) and efficient (with a good trade-off between precision and

cost) static analyses of run-time properties of real programs without actually executing them. Abstract interpretation is based on the idea of approximations between concrete and abstract domains of program properties. It has been used as the foundation for various successful industrial-scale static analyzers, such as ASTREE [5], that infer numerical invariants using numerical abstract domains [6]. They have been applied for ensuring the correctness of various software systems. However, current standard static analyzers based on abstract interpretation do not handle compile-time variability. They can analyze only single programs, but not the entire configuration space (all variants). Unfortunately, the static analysis of program families is harder than the static analysis of single programs, because the number of possible variants can be very large (often huge). The simplest brute-force approach, that uses a preprocessor to generate all variants of a family and then applies an existing off-the-shelf single-program analyzer to each individual variant, one-by-one, is very inefficient. This approach has to compile (preprocess and build control flow graph) as well as execute the fixed point iterative algorithm once for each possible variant. To overcome this problem, *variability-aware* (lifted, family-based) static analyses based on abstract interpretation have been proposed [7, 8]. They work on the family level, analyzing all variants of the family simultaneously, without generating any of them explicitly. The lifted approaches compile and execute the fixed point iterative algorithm only once per family. They take as input the common configurable code base, which encodes all variants of a program family, and produce precise disjunctive analysis results corresponding to all variants. They use a lifted analysis domain that represents the n-fold product of a single-program analysis domain used for expressing program properties. Here, n denotes the number of valid configurations (i.e., variants). That is, the lifted analysis domain maintains one property element per valid variant in tuples. This explicit property enumeration in tuples can be a bottleneck when dealing with families that have high variability. The problem is that this enumeration becomes computationally intractable with larger program families because the number of variants grows exponentially with the number of features. This is known as the configuration space explosion problem.

In this work, we show how to speed up the lifted analysis by improving the *representation* of the lifted analysis domain. The key for this is the proper handling of disjunctions of properties that arise in the lifted analysis due to the variability-specific constructs of the language (e.g. feature-based runtime tests and `#ifdef` directives). In particular, we propose a novel

lifted analysis domain that enables an explicit interaction (sharing) between analysis elements corresponding to different variants. The lifted analysis domain is given as a *binary decision diagram domain functor*, where Boolean features are organized in decision nodes and leaf nodes contain a particular analysis property. Binary decision diagram (BDD) domains represent an instance of the reduced cardinal power of domains [9, Sect. 10.2], which map the values of Boolean features (represented in decision nodes) to an analysis property (represented in leaf nodes) for the variants specified by the values of features along the path leading to the leaf. These decision diagram domains are particularly well suited for representing disjunctive properties (which is the key aspect of a lifted analysis). The lifted domain is parametric in the choice of the abstract domain for the leaf nodes, and so it can be used for inference of different program properties. The efficiency of BDDs comes from the opportunity to share equal subtrees, in case some properties are independent from the value of some features.

On the practical side, we have developed a prototype lifted static analyzer which uses the BDDAPRON library [10] to implement the binary decision diagram domain. BDDAPRON uses any property domain from the APRON library [11] for the leaf nodes. For example, APRON provides a common high-level API to the most common numerical property domains, such as intervals, octagons, and polyhedra. We have implemented two types of lifted analyses of C program families: a *forward* reachability lifted analysis for the automatic inference of *invariants*, and a *backward* lifted analysis for the automatic inference of *necessary preconditions*. These two analyses can be combined and run one after the other, such that the results of the second one (backward) refine and focus the results obtained from the first one (forward). The tool computes a set of possible numerical invariants (and preconditions) in all program locations, thanks notably to the design of numerical property domains, which allow one to extract the information about the possible values of individual program variables along with the possible relationships between them. In particular, we use: the (non-relational) interval domain [3], the (weakly-relational) octagon domain [12], and the (fully-relational) polyhedra domain [13]. The precision of numerical property domains increases from non-relational (interval) to fully-relational domains (polyhedra), but so does the computational complexity. We can use the implemented lifted static analyzer to prove the absence of runtime errors in `#ifdef`-enriched C programs, which represent majority of industrial embedded code. In particular, we are able to check invariance properties, such as assertions, buffer over-

flows, division by zero, etc [5]. We can also use the combination of forward and backward lifted analyses to automatically generate stronger invariants as well as necessary preconditions that lead to the satisfaction of a given assertion [14, 15].

Let us summarize the contributions of this work:

(C1) We propose a novel lifted domain based on BDDs, which is well suited for handling disjunctive properties that come from variability.

(C2) We develop a lifted static analyzer in which the lifted domains are instantiated to numerical property domains from the APRON library. The analyzer performs either only a forward analysis to find numerical invariants or a backward analysis in combination with a preliminary forward analysis to refine the found invariants.

(C3) Finally, we evaluate our approach for lifted static analysis of `#ifdef`-enriched C programs. We compare performances of our lifted analyzers based on tuples and binary decision diagrams, as well as showing their concrete applications in assertion checking.

This work can be of interest to program analysis and software engineering researchers. The approach of constructing and implementing the BDD lifted domain (**C1** – **C2**) is directed at designers of lifted analyses for configurable software systems. The evaluation lessons (**C3**) are relevant for software engineers working on `#ifdef`-based configurable programs and who would like to speed up existing analyzers.

This work extends and revises a GPCE conference article [16]. Additional material that did not appear in the conference version include: (1) a complete description of the concrete and abstract semantics; (2) a backward analysis that can be used in combination with a forward analysis to derive stronger invariants as well as necessary preconditions that lead to the occurrence of some property; (3) additional illustrations, explanations, and examples; (4) more evaluation results including analysis on more subject systems (BusyBox) and more practical applications. The paper proceeds with a motivating example that illustrates our new approach for lifted static analysis by abstract interpretation. The language for writing program families is introduced in Section 3. A complete description of concrete and abstract (forward and backward) analyses as well as the most common numerical property domains are given in Section 4. The basics of tuple-based lifted

analysis are introduced in Section 5. Section 6 defines our new BDD-based lifted analysis. Section 7 presents the evaluation on benchmarks taken from SV-COMP and BusyBox. Finally, we discuss related work and conclude.

2. Motivating Example

To better illustrate the issues we are addressing in this work, we now present a motivating example using the code base of the program family P:

```
(1)       int y := input([0,9]);
l_input   int x := 10;
(3)       while (x!=0) {
(4)           x := x-1;
(5)           #ifdef (A) y := y+1; #endif
(6)           #ifdef (B) y := y+1; #endif
(7)       }
(8)       assert (y ≤ 15);
```

The set of Boolean features in the program family P is $\mathbb{F} = \{A, B\}$ and the set of valid configurations is $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$. The initial value of the input variable `y` is non-deterministically chosen from the interval $[0, 9]$. The code base of P contains two `#ifdef` directives, which increase the variable `y` by 1, depending on which features from $\mathbb{F}$ are enabled at compile time. For each configuration from $\mathbb{K}$, a different variant (single program) can be generated by appropriately resolving `#ifdef`-s. For example, the variant corresponding to the configuration $A \wedge B$ will have both features A and B enabled (set to true), so that both assignments `y := y+1` in locations ⑤ and ⑥ will be included in this variant. On the other hand, the variant for configuration $\neg A \wedge \neg B$ will have both features A and B disabled (set to false), so the above assignments in locations ⑤ and ⑥ will not be included in it. There are $|\mathbb{K}| = 4$ variants that can be derived from P, shown in Fig. 1.

Assume that we want to perform *lifted analyses* on the family P using the numerical property domains: intervals, octagons, and polyhedra. The standard lifted domain from [7, 8] is defined as cartesian product of $\mathbb{K}$ copies of the basic domain, which corresponds to the client analysis we want to perform. Hence, elements of the lifted domain are tuples containing one component for each valid configuration from $\mathbb{K}$, where each component represents a numerical property over program variables (`x` and `y` in this case).

```
int y := input([0,9]);
int x := 10;
while (x != 0) {
      x := x-1;
      y := y+1;
      y := y+1; }
assert (y ≤ 15);
```
(a) $\mathtt{P}_{A\wedge B}(P)$

```
int y := input([0,9]);
int x := 10;
while (x != 0) {
      x := x-1;
      y := y+1; }
assert (y ≤ 15);
```
(b) $\mathtt{P}_{A\wedge\neg B}(P)$

```
int y := input([0,9]);
int x := 10;
while (x != 0) {
      x := x-1;
      y := y+1; }
assert (y ≤ 15);
```
(c) $\mathtt{P}_{\neg A\wedge B}(P)$

```
int y := input([0,9]);
int x := 10;
while (x != 0) {
      x := x-1; }
assert (y ≤ 15);
```
(d) $\mathtt{P}_{\neg A\wedge\neg B}(P)$

Figure 1: Different variants of the program family P.

$$\big(\overbrace{[\mathtt{y}\geq 0, \mathtt{x}=0]}^{A\wedge B}, \overbrace{[\mathtt{y}\geq 0, \mathtt{x}=0]}^{A\wedge\neg B}, \overbrace{[\mathtt{y}\geq 0, \mathtt{x}=0]}^{\neg A\wedge B}, \overbrace{[0\leq\mathtt{y}\leq 9, \mathtt{x}=0]}^{\neg A\wedge\neg B}\big)$$

(a) Intervals

$$\big(\overbrace{[\mathtt{y}\geq 10 \wedge \mathtt{x}=0]}^{A\wedge B}, \overbrace{[10\leq\mathtt{y}\leq 19 \wedge \mathtt{x}=0]}^{A\wedge\neg B}, \overbrace{[10\leq\mathtt{y}\leq 19 \wedge \mathtt{x}=0]}^{\neg A\wedge B}, \overbrace{[0\leq\mathtt{y}\leq 9 \wedge \mathtt{x}=0]}^{\neg A\wedge\neg B}\big)$$

(b) Octagons

$$\big(\overbrace{[20\leq\mathtt{y}\leq 29 \wedge \mathtt{x}=0]}^{A\wedge B}, \overbrace{[10\leq\mathtt{y}\leq 19 \wedge \mathtt{x}=0]}^{A\wedge\neg B}, \overbrace{[10\leq\mathtt{y}\leq 19 \wedge \mathtt{x}=0]}^{\neg A\wedge B}, \overbrace{[0\leq\mathtt{y}\leq 9 \wedge \mathtt{x}=0]}^{\neg A\wedge\neg B}\big)$$

(c) Polyhedra.

Figure 2: Tuple-based invariants at the location ⑧ of P.

The lifted analyses result in location ⑧ of P obtained using the lifted *interval, octagon, and polyhedra forward* analyses are the 4-sized tuples shown in Fig. 2a, Fig. 2b, and Fig. 2c, respectively. Note that the first component of a tuple in Fig. 2 corresponds to configuration $A \wedge B$, the second to $A \wedge \neg B$, the third to $\neg A \wedge B$, and the fourth to $\neg A \wedge \neg B$. From the analyses results in Fig. 2, we can see that the *interval forward analysis* in Fig. 2a discovers very coarse (approximative) results about the variable `y` for configurations $A \wedge B$, $A \wedge \neg B$ and $\neg A \wedge B$, that is $\mathtt{y} \geq 0$, since it is not able to reason about the relations between the variables `x` and `y`. Using this result in location ⑧, we can only successfully conclude that the assertion $(\mathtt{y} \leq 15)$ is always valid (for all inputs) for configuration $\neg A \wedge \neg B$. Failure to infer the best numerical invariants using intervals motivates the introduction of more expressive domains, such as octagons and polyhedra. The *octagon forward analysis* in Fig. 2b gives more precise (less approximative) results about `y`. That is, $10 \leq \mathtt{y} \leq 19$ for configurations $A \wedge \neg B$ and $\neg A \wedge B$ as well as $0 \leq \mathtt{y} \leq 9$ for $\neg A \wedge \neg B$ are exact results. But, we obtain a coarse result

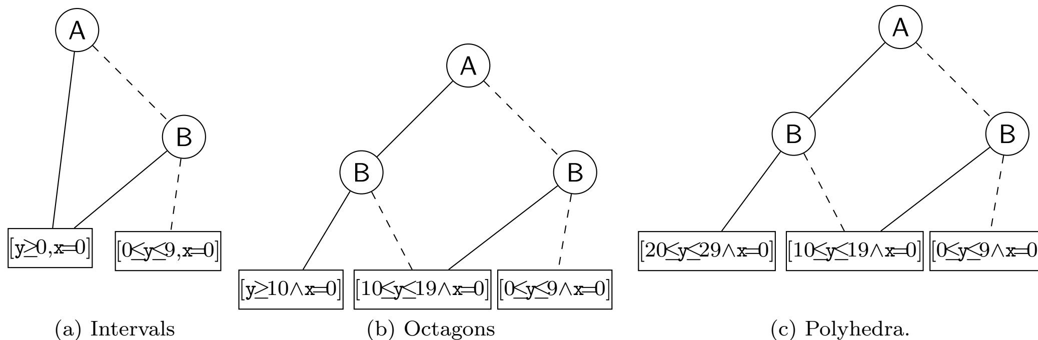

Figure 3: BDD-based invariants at the location ⑧ of P (solid edges = true, dashed edges = false).

$\mathtt{y} \geq 10$ for configuration $A \wedge B$, since the relations that can be tracked using the octagon analysis are limited, i.e. they are of the form $\pm\mathtt{x}\pm\mathtt{y} \geq c$. Finally, the *polyhedra forward analysis* in Fig. 2c reports the most precise results for both x and y in all configurations, since it is a fully relational domain and is able to track all (affine) relations between variables. Using this result in location ⑧, we can conclude that the given assertion always fails (for all inputs) for configuration $A \wedge B$, which was not possible to infer using interval and octagon analyses. However, for configurations $A \wedge \neg B$ and $\neg A \wedge B$ we obtain the invariant $10 \leq \mathtt{y} \leq 19$ at location ⑧, so the assertion can be satisfied (when $10 \leq \mathtt{y} \leq 15$) and can be violated (when $15 < \mathtt{y} \leq 19$). Suppose we are interested in inferring necessary preconditions on the input state at location $l_{\mathtt{input}}$ when the given assertion is satisfied. We back-propagate necessary preconditions of satisfaction of the assertion from location ⑧ to $l_{\mathtt{input}}$. A *backward lifted analysis* will infer the precondition $0 \leq \mathtt{y} \leq 5$ at location $l_{\mathtt{input}}$ for the assertion to be satisfied for configurations $A \wedge \neg B$ and $\neg A \wedge B$.

If we perform lifted forward analyses based on the *binary decision diagram domain* proposed here, then the analyses results (invariants) in the program location ⑧ of P obtained using interval, octagon, and polyhedra domains for leaves are shown in Fig. 3a, Fig. 3b, and Fig. 3c, respectively. Note that the inner nodes of a binary decision diagram (BDD) in Fig. 3 are labeled with Boolean features from $\mathbb{F}$, while the leaves are labeled with the elements from the property domain we use (i.e. affine constraints over program variables x and y). The edges of BDDs are labeled with the truth value of the decision on the parent node: true or false (we use solid edges for true, and

dashed edges for false). It is obvious that BDDs offer more possibilities for sharing and interaction between analysis properties corresponding to different configurations. Thus, they provide symbolic and compact representation of lifted analysis elements. For example, Fig. 3a presents interval properties of program variables `x` and `y`, which are partitioned with respect to Boolean features A and B. When A is true, the property is independent from the value of B, hence the node at level B can be omitted. Moreover, the cases (A is true) and (A is false and B is true) are identical, so they share the same leaf node. As a consequence, this representation uses only two leaf nodes, while the tuple-based representation in Fig. 2a uses four. This ability for sharing is the key motivation behind the BDD-based representation. It makes BDDs more economical than tuples in representing lifted analyses results. Notice that, in the worst case, BDDs still need $|\mathbb{K}|$ different leaf nodes, but experimental evidence shows that sharing often occurs in practice.

3. A Language for Program Families

Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite and totally ordered set of Boolean variables representing the *features* available in a program family. A specific subset of features, $k \subseteq \mathbb{F}$, known as *configuration*, specifies a *variant* of a program family. We assume that only a subset $\mathbb{K} \subseteq 2^{\mathbb{F}}$ of all possible configurations are *valid*. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathbb{K}$ can be represented by a *valuation formula*: $k(A_1) \wedge \ldots \wedge k(A_n)$, where for each feature $A \in \mathbb{F}$, we have $k(A) = A$ if $A \in k$ and $k(A) = \neg A$ if $A \notin k$. The set of valid configurations $\mathbb{K}$ can be also represented as a formula: $\vee_{k \in \mathbb{K}} k$. We will use both representations.

We define *feature expressions*, denoted $\mathit{FeatExp}(\mathbb{F})$, as the set of well-formed propositional logic formulae over $\mathbb{F}$ generated by the grammar:

$$\theta ::= \text{true} \,|\, A \in \mathbb{F} \,|\, \neg\theta \,|\, \theta_1 \wedge \theta_2 | \, \theta_1 \vee \theta_2$$

We will use $\theta \in \mathit{FeatExp}(\mathbb{F})$ to define presence conditions in program families. When a configuration $k \in \mathbb{K}$ satisfies $\theta \in \mathit{FeatExp}(\mathbb{F})$, we write $k \models \theta$, where $\models$ is the standard satisfaction relation of logic. We write $[\![\theta]\!]$ to denote the set of configurations from $\mathbb{K}$ that satisfy θ, that is, $k \in [\![\theta]\!]$ iff $k \models \theta$.

Example 1. *Let us revisit the program family P from Section 2. The set of features is $\mathbb{F} = \{A, B\}$, and there are four possible valid configurations*

$\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$ *(or, equivalently using sets* $\mathbb{K} = \{\{A, B\}, \{A\}, \{B\}, \emptyset\}$*). For the feature expression* $A \vee B$*, we have* $[\![A \vee B]\!] = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$*. Therefore, it holds that* $(A \wedge B) \models (A \vee B)$ *and* $(\neg A \wedge \neg B) \not\models (A \vee B)$*.* □

We consider a simple deterministic programming language, which will be used to exemplify our work. The variables are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. Note that our implementation, described in Section 7, actually supports a subset of the C language enriched with `#ifdef`-s, which is sufficient to handle realistic program families. To encode multiple variants, a new compile-time conditional statement is included. The new statement "`#ifdef` (θ) s `#endif`" contains a feature expression $\theta \in \mathit{FeatExp}(\mathbb{F})$ as a presence condition, such that only if θ is satisfied by a configuration $k \in \mathbb{K}$ the statement s will be included in the variant corresponding to k. The language is also extended with non-deterministic interval assignment in order to model input uncertainties. The control location before each statement and at the end of each block is associated to a unique label $l \in \mathbb{L}$. The syntax of the language is given by:

$$\begin{aligned} s ::= \;& \texttt{skip} \mid \texttt{x := } e \mid \texttt{x := input}([n, n']) \mid s\texttt{ ; }s \mid \texttt{if } e \texttt{ then } s \texttt{ else } s \mid \\ & \texttt{while } e \texttt{ do } s \mid \texttt{assert}(e) \mid \texttt{\#ifdef } (\theta)\; s \texttt{ \#endif} \\ e ::= \;& n \mid \texttt{x} \mid e \oplus e \end{aligned}$$

where n ranges over integers, $[n, n']$ ranges over integer intervals, `x` ranges over variable names *Var*, and $\oplus \in \{+, -, *, \backslash, <, =, \neg, \wedge\}$. [1] Non-deterministic interval assignment `x := input`$([n, n'])$ represents an input statement which assigns to the input variable `x` a random value from the interval $[n, n']$. This interval assignment can occur only in the *input section*, and is used to model input uncertainties that are out of the control of the program. The set of all generated statements s is denoted by *Stm*, while the set of all expressions e is denoted by *Exp*. We assume $l_{\texttt{input}}$ is the location after the input statements, thus it denotes the end of the input section.

Note that the C preprocessor uses the following keywords: `#if`, `#ifdef`, and `#ifndef` to start a conditional construct; `#elif` and `#else` to create additional branches; and `#endif` to end a construct. Any of such preprocessor

[1] Following the convention popularized by C, we model Boolean values as integers, with zero interpreted as false and everything else as true.

conditional constructs can be desugared and represented only by the `#ifdef` construct we use in this work.

A program family is evaluated in two stages. First, a *preprocessor* takes a program family s and a configuration $k \in \mathbb{K}$ as inputs, and produces a variant (that is, a single program without `#ifdef`-s) corresponding to k as the output. Second, the obtained variant is evaluated using the standard single-program semantics [8]. The first stage is specified by the projection function $\mathtt{P}_k$, which is an identity for all basic statements and recursively pre-processes all sub-statements of compound statements. Hence, $\mathtt{P}_k(\mathtt{skip}) = \mathtt{skip}$ and $\mathtt{P}_k(s;s') = \mathtt{P}_k(s);\mathtt{P}_k(s')$. The interesting case is "`#ifdef` (θ) s `#endif`", where the statement s is included in the resulting variant if $k \models \theta$, otherwise, if $k \not\models \theta$ the statement s is removed. [2] That is,

$$\mathtt{P}_k(\mathtt{\#ifdef}\ (\theta)\ s\ \mathtt{\#endif}) = \begin{cases} \mathtt{P}_k(s) & \text{if } k \models \theta \\ \mathtt{skip} & \text{if } k \not\models \theta \end{cases}$$

For example, the variants $\mathtt{P}_{A\wedge B}(P)$, $\mathtt{P}_{A\wedge\neg B}(P)$, $\mathtt{P}_{\neg A\wedge B}(P)$, and $\mathtt{P}_{\neg A\wedge\neg B}(P)$ shown in Fig. 1a, Fig. 1b, Fig. 1c, and Fig. 1d, respectively, are derived from the program family P defined in Section 2.

4. Background: Analyses based on Abstract Interpretation

This section represents a brief summary of the ideas and concepts of abstract interpretation, and in particular how it can be used for deriving forward and backward abstract analyses of single programs. We also briefly recall the well-known numerical property domains of intervals [3], octagons [12], and polyhedra [13], that can be used for automatic discovery of numerical properties of program variables. They are the foundation upon which we implement in practice new lifted domains introduced in Sections 5 and 6. We leave `#ifdef` directives aside in this section and work only with single programs without `#ifdef`-s in the following.

4.1. Concrete Semantics and Analysis

We first introduce a *concrete semantics* of our language, which is the starting point in abstract interpretation. Then, we introduce *invariant inference* (forward) and *necessary precondition* (backward) *analyses* defined on

[2] Since any $k \in \mathbb{K}$ is a valuation formula, we have that either $k \models \theta$ holds or $k \not\models \theta$ (which is equivalent to $k \models \neg\theta$) holds, for any $\theta \in \mathit{FeatExp}(\mathbb{F})$.

the concrete semantics. They can be seen as concrete analyses that do not introduce any imprecision. Such analyses are obviously *uncomputable*, i.e. they cannot be computed statically, since our language allows unbounded program executions and unbounded numerical values for variables (i.e. the language is Turing complete). In the next subsection, we introduce the notion of a *Galois connection*, which represents a pair of functions capturing information loss between two domains. Then, we demonstrate how to use concrete analyses and Galois connections to derive approximate, albeit computable analyses, which can statically determine dynamic properties of programs.

A *program state* is given by a control location in $\mathbb{L}$ and an environment in $\mathcal{E} : \mathit{Var} \to \mathbb{Z}$ mapping each variable to its value (integer number). We write $\Sigma = \mathbb{L} \times \mathcal{E}$ to denote the set of all possible program states. Programs are modelled as transition systems $(\Sigma, \longrightarrow)$, where Σ is a set of states and $\longrightarrow \subseteq \Sigma \times \Sigma$ is a transition relation modelling atomic execution steps. The relation $\longrightarrow$ is defined by local rules, such as the following:

do-nothing $l_0 : \texttt{skip}; l_1$ **::** $(l_0, \rho) \longrightarrow (l_1, \rho)$.

assignment $l_0 : \texttt{x := } e; l_1$ **::** $(l_0, \rho) \longrightarrow (l_1, \rho[\texttt{x} \mapsto [\![e]\!](\rho)])$, where $[\![e]\!](\rho) \in \mathbb{Z}$ is the result of the evaluation of e in the environment ρ, and $\rho[\texttt{x} \mapsto n]$ denotes the environment that updates ρ at variable `x` to equal value n.

input $l_0 : \texttt{x := input}([n, n']); l_1$ **::** $(l_0, \rho) \longrightarrow (l_1, \rho[\texttt{x} \mapsto n''])$, where $n'' \in [n, n']$.

conditional $l_0 : \texttt{if } (e) \texttt{ then } \{l_0^t : s;\ l_1^t\} \texttt{ else } \{l_0^f : s';\ l_1^f\}; l_1$ **::** $(l_0, \rho) \longrightarrow (l_0^t, \rho)$ if $[\![e]\!](\rho) \neq 0$, $(l_0, \rho) \longrightarrow (l_0^f, \rho)$ if $[\![e]\!](\rho) = 0$, $(l_1^t, \rho) \longrightarrow (l_1, \rho)$, and $(l_1^f, \rho) \longrightarrow (l_1, \rho)$. Note that the earlier two rules show how control moves from the initial label l_0 of the conditional to initial labels l_0^t and l_0^f of `then` and `else` branches, whereas the latter two rules from final labels l_1^t and l_1^f of `then` and `else` branches to the final label l_1 of the whole conditional.

loop $l_0 : \texttt{while } (e) \texttt{ do } \{l_0^t : s;\ l_1^t\}; l_1$ **::** $(l_0, \rho) \longrightarrow (l_0^t, \rho)$ if $[\![e]\!](\rho) \neq 0$, $(l_0, \rho) \longrightarrow (l_1, \rho)$ if $[\![e]\!](\rho) = 0$, and $(l_1^t, \rho) \longrightarrow (l_0, \rho)$. Note that control moves from the final label l_1^t of `while`-body s to the initial label l_0 of `while`.

assertion $l_0 : \texttt{assert}(e); l_1$ **::** $(l_0, \rho) \longrightarrow (l_1, \rho)$ if $[\![e]\!](\rho) \neq 0$.

Let $\mathbb{E} \subseteq \mathcal{E}$ be the set of input environments obtained after executing the input statements. The set of input states is $\mathcal{I} = \{(l_{\texttt{input}}, \rho) \mid \rho \in \mathbb{E}\}$. An *invariant inference* (forward) analysis consists of finding out the reachable environments (values of all variables) in all control locations. The concrete semantic domain is the complete lattice of the powerset of states $\langle \mathcal{P}(\Sigma), \subseteq, \cup, \cap, \emptyset, \Sigma \rangle$, and the forward analysis in the form of invariant states encountered branching from the set of input states $\mathcal{I}$, denoted $\texttt{inv}(\mathcal{I})$, is:

$$\texttt{inv}(\mathcal{I}) = \texttt{lfp}_{\mathcal{I}} \lambda X.X \cup \texttt{post}(X)$$

where $\texttt{post}(X) = \{\sigma \in \Sigma \mid \exists \sigma' \in X.\sigma' \longrightarrow \sigma\}$ and $\texttt{lfp}_{\mathcal{I}} f$ is the least fixed point of the function f greater than or equal to $\mathcal{I}$.

In this work, we also consider a backward analysis for inferring *necessary preconditions*. Assume that we want an expression $e_{\texttt{f}}$ to hold at a program location $l_{\texttt{final}}$. Let $\mathcal{F} = \{(l, \rho) \in \texttt{inv}(\mathcal{I}) \mid l = l_{\texttt{final}} \implies [\![e_{\texttt{f}}]\!](\rho) \neq 0\}$ be the invariant set that enforces the expression $e_{\texttt{f}}$ at location $l_{\texttt{final}}$ to be satisfied (i.e., $e_{\texttt{f}}$ is not equal to zero), and $\mathcal{F}$ coincides with $\texttt{inv}(\mathcal{I})$ everywhere else. Given an invariant set $\mathcal{F}$ to obey, we want to infer the set of necessary preconditions $\texttt{cond}(\mathcal{F})$ that guarantee that all program executions branching from $(l, \rho) \in \texttt{cond}(\mathcal{F})$ stay in $\mathcal{F}$:

$$\texttt{cond}(\mathcal{F}) = \texttt{gfp}_{\mathcal{F}} \lambda X.X \cap \texttt{pre}(X)$$

where $\texttt{pre}(X) = \{\sigma \in \Sigma \mid \exists \sigma' \in X.\sigma \longrightarrow \sigma'\}$ is the set of predecessors of X, and $\texttt{gfp}_{\mathcal{F}} f$ is the greatest fixed point of the function f smaller than or equal to $\mathcal{F}$. The above two fixed points ($\texttt{lfp}$ and $\texttt{gfp}$) exist according to Tarski [17], as the corresponding functions are monotone and continuous in the complete lattice of state sets $\langle \mathcal{P}(\Sigma), \subseteq \rangle$.

Given a set of input environments $\mathbb{E} \subseteq \mathcal{E}$, we can compute the subset $\mathbb{E}_{sat}$ of input environments that lead to satisfaction of the expression $e_{\texttt{f}}$ as:

$$\mathbb{E}_{sat} = \mathbb{E} \cap \{\rho \mid (l_{\texttt{input}}, \rho) \in \texttt{cond}(\mathcal{F})\}$$

4.2. Abstract Semantics and Analysis

Transition systems can become large or infinite for real programs, so that neither $\texttt{inv}(\mathcal{I})$ nor $\texttt{cond}(\mathcal{F})$ can be computed at all. Therefore, we seek for sound approximations. The actual computable abstract analyses can be defined as over-approximations of the concrete analyses and semantics. A static analyzer will infer over-approximated invariants and necessary preconditions

in all program locations. For example, the computed necessary precondition on the input state in $l_{\texttt{input}}$ will represent an over-approximation of $\mathbb{E}_{sat}$.

A *Galois connection* is a pair of functions, $\alpha : \mathbb{C} \to \mathbb{A}$ and $\gamma : \mathbb{A} \to \mathbb{C}$ (respectively known as the *abstraction* and *concretization* functions), connecting two partially ordered sets, $\langle \mathbb{C}, \leqslant \rangle$ and $\langle \mathbb{A}, \sqsubseteq \rangle$ (often called the *concrete* and *abstract* domain, respectively), such that:

$$\forall c \in \mathbb{C}, a \in \mathbb{A} : \quad \alpha(c) \sqsubseteq a \;\Leftrightarrow\; c \leqslant \gamma(a) \tag{1}$$

which is often typeset as: $\langle \mathbb{C}, \leqslant \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{A}, \sqsubseteq \rangle$. For a concrete domain $\mathbb{C}$, we define *abstraction* and *concretization* functions to and from a more abstract domain $\mathbb{A}$, where information has been abstracted away.

We now show how to derive abstract analyses via a Galois connection. Suppose that we have an abstract domain $\langle \mathbb{A}, \sqsubseteq_{\mathbb{A}} \rangle$, such that there exist a Galois connection $\langle \mathcal{P}(\mathcal{E}), \subseteq \rangle \underset{\alpha_{\mathbb{A}}}{\overset{\gamma_{\mathbb{A}}}{\leftrightarrows}} \langle \mathbb{A}, \sqsubseteq_{\mathbb{A}} \rangle$. We assume that the abstract domain $\mathbb{A}$ is equipped with sound operators for ordering $\sqsubseteq_{\mathbb{A}}$, least upper bound (join) $\sqcup_{\mathbb{A}}$, greatest lower bound (meet) $\sqcap_{\mathbb{A}}$, bottom $\perp_{\mathbb{A}}$, top $\top_{\mathbb{A}}$, widening $\triangledown_{\mathbb{A}}$, and narrowing $\triangle_{\mathbb{A}}$, as well as sound transfer functions for forward assignments $\text{ASSIGN}_{\mathbb{A}} : \mathit{Stm} \times \mathbb{A} \to \mathbb{A}$, tests $\text{FILTER}_{\mathbb{A}} : \mathit{Exp} \times \mathbb{A} \to \mathbb{A}$, and backward assignments $\text{B-ASSIGN}_{\mathbb{A}} : \mathit{Stm} \times \mathbb{A} \times \mathbb{A} \to \mathbb{A}$. We let $\texttt{lfp}^{\#}$ (resp., $\texttt{gfp}^{\#}$) denote an abstract post-fixpoint (resp., pre-fixpoint) operator, derived using widening $\triangledown_{\mathbb{A}}$ and narrowing $\triangle_{\mathbb{A}}$, that over-approximates the concrete $\texttt{lfp}$ (resp., $\texttt{gfp}$) [18]. Finally, the concrete domain on which concrete semantics is defined $\langle \mathcal{P}(\Sigma), \subseteq \rangle$ is abstracted using a Galois connection $\langle \mathcal{P}(\Sigma), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{L} \to \mathbb{A}, \dot{\sqsubseteq} \rangle$ where $\alpha(R) = \lambda l \in \mathbb{L}. \sqcup_{\mathbb{A}} \{a \in \mathbb{A} \mid (l, \rho) \in R, \alpha_{\mathbb{A}}(\rho) = a\}$. Hence, each control location $l \in \mathbb{L}$ is associated with an element $a \in \mathbb{A}$ in the abstract semantics. Note that abstract elements $a \in \mathbb{A}$ must be computer representable, and all operators and transfer functions of $\mathbb{A}$ must be given as effective algorithms.

Remark 1. *In this work, backward assignments* $\text{B-ASSIGN}_{\mathbb{A}} : \mathit{Stm} \times \mathbb{A} \times \mathbb{A} \to \mathbb{A}$, *and in general backward transfer functions* $\overleftarrow{\delta}_{l,l'} : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$, *are used to refine the results of a previous forward analysis meaning that an over-approximation of the set of environments reachable before any statement is available. Therefore,* $\text{B-ASSIGN}_{\mathbb{A}}$ *(resp.,* $\overleftarrow{\delta}_{l,l'}$*) takes two elements of* $\mathbb{A}$ *as inputs: the first one is an invariant in the initial label of the assignment (resp., statement) found by the forward analysis that needs to be refined, and the second one is an invariant in the final label of the assignment (resp., statement) that needs to be propagated backwards.*

We define a family of *forward abstract transfer functions* $\overrightarrow{\delta}_{l,l'} : \mathbb{A} \to \mathbb{A}$ that compute the effect of any concrete transition at the abstract level. They use only abstract versions of operators and transfer functions we assume given with the domain $\mathbb{A}$. The definition of $\overrightarrow{\delta}_{l,l'}$ is:

do-nothing $l_0 : \texttt{skip}; l_1$ **::** $\overrightarrow{\delta}_{l_0,l_1}(a) = a$.

assignment $l_0 : \texttt{x := } e; l_1$ **::** $\overrightarrow{\delta}_{l_0,l_1}(a) = \text{ASSIGN}_{\mathbb{A}}(\texttt{x:=}e, a)$.

conditional $l_0 : \texttt{if } (e) \texttt{ then } \{l_0^t : s;\ l_1^t\} \texttt{ else } \{l_0^f : s';\ l_1^f\}; l_1$ **::** $\overrightarrow{\delta}_{l_0,l_0^t}(a) = \text{FILTER}_{\mathbb{A}}(e, a)$, $\overrightarrow{\delta}_{l_0,l_0^f}(a) = \text{FILTER}_{\mathbb{A}}(\neg e, a)$, $\overrightarrow{\delta}_{l_1^t,l_1}(d) = d$, $\overrightarrow{\delta}_{l_1^f,l_1}(d) = d$.

loop $l_0 : \texttt{while } (e) \texttt{ do } \{l_0^t : s;\ l_1^t\}; l_1$ **::** $\overrightarrow{\delta}_{l_0,l_0^t}(a) = \text{FILTER}_{\mathbb{A}}(e, a)$, $\overrightarrow{\delta}_{l_0,l_1}(a) = \text{FILTER}_{\mathbb{A}}(\neg e, a)$, and $\overrightarrow{\delta}_{l_1^t,l_0}(a) = a$.

assertion $l_0 : \texttt{assert}(e); l_1$ **::** $\overrightarrow{\delta}_{l_0,l_1}(a) = \text{FILTER}_{\mathbb{A}}(e, a)$.

The requirement for soundness of $\{\overrightarrow{\delta}_{l,l'} \mid l, l' \in \mathbb{L}\}$ is written as: $\forall a \in \mathbb{A}, \forall \rho \in \gamma_{\mathbb{A}}(a), (l, \rho) \longrightarrow (l', \rho') \implies \rho' \in \gamma_{\mathbb{A}}(\overrightarrow{\delta}_{l,l'}(a))$.

Suppose that the abstract element $\alpha_{\mathbb{A}}(\mathbb{E}) = a_{\texttt{input}} \in \mathbb{A}$ is at the input control location $l_{\texttt{input}}$. We can collect the abstractions of possible environments at each control location using the following forward abstract interpreter:

$$\overrightarrow{F}^{\#} = \lambda I.\lambda(l \in \mathbb{L}). \sqcup_{\mathbb{A}} \{\overrightarrow{\delta}_{l',l}(I(l')) \mid l' \in \mathbb{L}\}$$

such that the result of the forward analyzer is $\overrightarrow{\mathcal{I}}^{\#} = \texttt{lfp}^{\#}_{I_0} \overrightarrow{F}^{\#}$, where $I_0(l_{\texttt{input}}) = a_{\texttt{input}}$.

Assume $\overrightarrow{\mathcal{I}}^{\#}(l_{\texttt{final}}) = a_{\texttt{final}}$, and $a^{sat}_{\texttt{final}} = \text{FILTER}_{\mathbb{A}}(e_{\texttt{f}}, a_{\texttt{final}})$, where $e_{\texttt{f}}$ is the expression we want to hold at location $l_{\texttt{final}}$. We want to design a backward abstract interpreter that propagates backwards the invariants ensuring that the expression $e_{\texttt{f}}$ is satisfied, $a^{sat}_{\texttt{final}}$. The backward interpreter refines the invariants found by the forward abstract interpreter $\overrightarrow{F}^{\#}$. Thus, it takes two elements of $\mathbb{A}$ as inputs: an invariant found by $\overrightarrow{F}^{\#}$ to refine and an invariant to propagate backwards. It is based on a family of *backward abstract transfer functions* $\overleftarrow{\delta}_{l,l'} : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$, which map a precondition at label l found by $\overrightarrow{F}^{\#}$ to refine and a postcondition at label l' into a refined precondition at label l. The definition of $\overleftarrow{\delta}_{l,l'}$ is:

do-nothing $l_0 : \mathtt{skip}; l_1$ **::** $\overleftarrow{\delta}_{l_0,l_1}(a, a') = a \sqcap a'$.

assignment $l_0 : \mathtt{x} \ \mathtt{:=}\ e; l_1$ **::** $\overleftarrow{\delta}_{l_0,l_1}(a, a') = \text{B-ASSIGN}_{\mathbb{A}}(\mathtt{x:=}e, a, a')$.

conditional $l_0 : \mathtt{if}\ (e)\ \mathtt{then}\ \{l_0^t : s; l_1^t\}\ \mathtt{else}\ \{l_0^f : s'; l_1^f\}; l_1$ **::** $\overleftarrow{\delta}_{l_0,l_0^t}(a, a') = a \sqcap (a' \sqcup \text{FILTER}_{\mathbb{A}}(\neg e, \top_{\mathbb{A}}))$, $\overleftarrow{\delta}_{l_0,l_0^f}(a, a') = a \sqcap (a' \sqcup \text{FILTER}_{\mathbb{A}}(e, \top_{\mathbb{A}}))$, $\overleftarrow{\delta}_{l_1^t,l_1}(a, a') = a \sqcap a'$, $\overleftarrow{\delta}_{l_1^f,l_1}(a, a') = a \sqcap a'$.

loop $l_0 : \mathtt{while}\ (e)\ \mathtt{do}\ \{l_0^t : s; l_1^t\}; l_1$ **::** $\overleftarrow{\delta}_{l_0,l_0^t}(a, a') = a \sqcap (a' \sqcup \text{FILTER}_{\mathbb{A}}(\neg e, \top_{\mathbb{A}}))$, $\overleftarrow{\delta}_{l_0,l_1}(a, a') = a \sqcap (a' \sqcup \text{FILTER}_{\mathbb{A}}(e, \top_{\mathbb{A}}))$, and $\overleftarrow{\delta}_{l_1^t,l_0}(a, a') = a \sqcap a'$.

assertion $l_0 : \mathtt{assert}(e); l_1$ **::** $\overleftarrow{\delta}_{l_0,l_1}(a, a') = a \sqcap (a' \sqcup \text{FILTER}_{\mathbb{A}}(\neg e, \top_{\mathbb{A}}))$.

The requirement for soundness of $\{\overleftarrow{\delta}_{l,l'} \mid l, l' \in \mathbb{L}\}$ is written as: $\forall a, a' \in \mathbb{A}, \forall \rho \in \gamma_{\mathbb{A}}(a), \rho' \in \gamma_{\mathbb{A}}(a'), (l, \rho) \longrightarrow (l', \rho') \implies \rho \in \gamma_{\mathbb{A}}(\overleftarrow{\delta}_{l,l'}(a, a'))$. That is, a is refined into a stronger precondition by taking into account the postcondition a'.

Suppose that $\mathcal{F}^{sat} = \overrightarrow{\mathcal{I}}^{\#}[l_{\mathtt{final}} \mapsto a^{sat}_{\mathtt{final}}]$ be an invariant that enforces $a^{sat}_{\mathtt{final}}$ to hold at location $l_{\mathtt{final}}$, and everywhere else $\mathcal{F}^{sat}$ coincides with the invariant $\overrightarrow{\mathcal{I}}^{\#}$ found by the forward abstract interpreter. The backward abstract interpreter is defined as:

$$\overleftarrow{F}^{\#} = \lambda F.\lambda(l \in \mathbb{L}). \sqcap_{\mathbb{A}} \{\overleftarrow{\delta}_{l,l'}(F(l), F(l')) \mid l' \in \mathbb{L}\}$$

such that the results of the backward analyzer is: $\overleftarrow{C}^{\#}_{sat} = \mathtt{gfp}^{\#}_{\mathcal{F}^{sat}} \overleftarrow{F}^{\#}$. The necessary preconditions that the given expression $e_{\mathtt{f}}$ is satisfied in $l_{\mathtt{input}}$ is $a^{sat}_{\mathtt{input}} = \overleftarrow{C}^{\#}_{sat}(l_{\mathtt{input}})$. We can now compute the over-approximated set $\mathbb{E}^{\#}_{sat}$ of input environments $\mathbb{E}_{sat}$, which *may* lead to satisfaction of the expression $e_{\mathtt{f}}$, as: $\mathbb{E}^{\#}_{sat} = \mathbb{E} \cap \gamma_{\mathbb{A}}(a^{sat}_{\mathtt{input}})$, such that $\mathbb{E}^{\#}_{sat} \supseteq \mathbb{E}_{sat}$. Note that input environments from $\mathbb{E} \backslash \mathbb{E}^{\#}_{sat}$ will definitely lead to failure of the given expression $e_{\mathtt{f}}$.

4.3. Numerical Property Domains

So far, we have presented a generic static analyzer parameterized by the choice of an abstract domain $\mathbb{A}$ equipped with a set of sound operators and transfer functions. Now, we present several such numerical abstract domains of intervals [3], octagons [12], and polyhedra [13]. Note that each domain employs data structures and algorithms specific to the shape of invariants it

represents and manipulates. For instance, we will see that interval domain employs algorithms from interval arithmetic; octagon domain from matrices and graphs; while polyhedra domain from the theory of convex polyhedra.

Intervals. The *Interval domain* [3] (also called *Box domain*), denoted as $\langle I, \sqsubseteq_I \rangle$, is a non-relational numerical property domain, which abstracts each variable independently but do not take variable relationships into account. It identifies the range of possible values for every variable as an interval. The property elements are: $\{\perp_I\} \cup \{[l,h] \mid l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leq h\}$, where the least element (bottom) $\perp_I$ denotes the empty interval and the greatest element (top) is $\top_I = [-\infty, +\infty]$. The abstract operations of the *Interval domain* are defined in [3]. Interval analysis is very cheap, that is, all the domain operations can be performed in linear time and space in the number of variables.

We now give precise definitions of some operations. The concretization function γ_I, which assigns a concrete meaning to each element from I, is:

$$\gamma_I(\perp_I) \;=\; \emptyset, \qquad \gamma_I([l,h]) \;=\; \{n \in \mathbb{Z} \mid l \leq n \leq h\}$$

The partial ordering $\sqsubseteq_I$, and meet $\sqcap_I$ are defined as:

$$\begin{aligned} &[l_1,h_1] \sqsubseteq_I [l_2,h_2] \quad \equiv_{def} \quad l_2 \leq l_1 \wedge h_1 \leq h_2, \\ &[l_1,h_1] \;\sqcap_I\; [l_2,h_2] = [max\{l_1,l_2\}, min\{h_1,h_2\}] \end{aligned}$$

The interval domain has infinite strictly ascending chains so we need to define widening operators in order to enforce convergence of the fixed point of `while` loops. The standard widening consists in replacing any unstable upper bound with $+\infty$ and any unstable lower bound with $-\infty$ [3]:

$$[l_1,h_1] \nabla_I [l_2,h_2] = \Big[\begin{cases} l_1, & \text{if } l_1 \leq l_2 \\ -\infty, & \text{otherwise} \end{cases}, \begin{cases} h_1, & \text{if } h_1 \geq h_2 \\ +\infty, & \text{otherwise} \end{cases} \Big]$$

In order to improve the precision of loop analysis, we can apply the narrowing after stabilization with widening is achieved. This is a simple choice:

$$[l_1,h_1] \Delta_I [l_2,h_2] = \Big[\begin{cases} l_2, & \text{if } l_1 = -\infty \\ l_1, & \text{otherwise} \end{cases}, \begin{cases} h_2, & \text{if } h_1 = +\infty \\ h_1, & \text{otherwise} \end{cases} \Big]$$

Let $a \in Var \to I$ be an abstract state which maps each variable `x` to an interval. The transfer function FILTER_I abstracts tests (expressions) in

while-s and if-s by restricting the input abstract store so that it satisfies the given test. We can handle some simple cases precisely, and we use the identity as a sound abstraction for the other cases. For example, we have:

$$\text{FILTER}_I(\texttt{x} \leq n : \mathit{Exp}, a : \mathit{Var} \to I) = \begin{cases} a[\texttt{x} \mapsto [l, \mathit{min}(h, n)]], & \text{if } l \leq n \\ \bot_I, & \text{if } l > n \end{cases}$$

where $a(\texttt{x}) = [l, h]$. The transfer function for assignments is:

$$\text{ASSIGN}_I(\texttt{x:=}e : \mathit{Stm}, a : \mathit{Var} \to I) = a[\texttt{x} \mapsto [\![e]\!]_I a]$$

where $[\![e]\!]_I a$ is the value obtained by abstract evaluation of e in the store a.

The backward assignment for the invertible cases [6], `x := x+`$[l, h]$ and `x := -x+`$[l, h]$, is defined by forward assignments (for all numerical domains):

$$\text{B-ASSIGN}_{\mathbb{A}}(\texttt{x:=x+}[l, h] : \mathit{Stm}, a : \mathbb{A}, a' : \mathbb{A}) = \text{ASSIGN}_{\mathbb{A}}(\texttt{x:=x+}[\texttt{-}h, \texttt{-}l], a') \sqcap a$$
$$\text{B-ASSIGN}_{\mathbb{A}}(\texttt{x:=-x+}[l, h] : \mathit{Stm}, a : \mathbb{A}, a' : \mathbb{A}) = \text{ASSIGN}_{\mathbb{A}}(\texttt{x:=-x+}[l, h], a') \sqcap a$$

We refer to [6, 12] for the definition of other cases.

Octagons. The *Octagon domain* [12], denoted as $\langle O, \sqsubseteq_O \rangle$, is a weakly relational numerical property domain, where property elements are conjunctions of linear inequalities of the form $\pm \texttt{x}_j \pm \texttt{x}_i \leq c$ between variables $\texttt{x}_i$ and $\texttt{x}_j$. Abstract operations of the *Octagon domain* are defined in [12]. The octagon analysis has a cubic time cost per domain operation. Thus, it represents a trade-off between the interval analysis, which is very cheap but quite imprecise, and the polyhedra analysis, which is very expressive but costly.

Each property element is encoded as *Difference Bound Matrix* (DBM) $\mathbf{m}$ which is a $2n \times 2n$ matrix, where n is the total number of program variables. For each variable $x_i \in \mathit{Var}$, we consider two versions x'_{2i-1} and x'_{2i} which correspond to $+x_i$ and $-x_i$ respectively. The element m_{ij} at row i and column j of $\mathbf{m}$ ($1 \leq i \leq 2n, 1 \leq j \leq 2n$), denotes the constraint $x'_j - x'_i \leq m_{ij}$. The concretization function γ_O is defined as:

$$\gamma_O(\mathbf{m}) = \{(v_1, \ldots, v_n) \in \mathbb{Z}^n \mid (v_1, -v_1, \ldots, v_n, -v_n) \in \gamma_{DBM}(\mathbf{m})\}$$
$$\gamma_{DBM}(\mathbf{m}) = \{(v_1, \ldots, v_{2n}) \in \mathbb{Z}^{2n} \mid \forall i, j. v_j - v_i \leq m_{ij}\}$$

The structure $\langle DBM, \sqsubseteq_{DBM}, \sqcup_{DBM}, \sqcap_{DBM}, \bot_{DBM}, \top_{DBM} \rangle$ is a lattice, where $\sqsubseteq_{DBM}$, $\sqcup_{DBM}$, and $\sqcap_{DBM}$ are defined element-wise, by extending the regular arithmetic order $\leq$ on $\mathbb{Z}$. The top element $\top_{DBM}$ has all its elements $+\infty$.

As for the interval domain, the widening ∇_O puts unstable bounds to infinity, while the narrowing Δ_O refines an upper bound only if it is infinity.

$$\forall i,j. \big[\mathbf{m}\nabla_O\mathbf{n}\big]_{ij} = \begin{cases} m_{ij}, & \text{if } n_{ij} \leq m_{ij} \\ +\infty, & \text{otherwise} \end{cases}, \quad \forall i,j. \big[\mathbf{m}\Delta_O\mathbf{n}\big]_{ij} = \begin{cases} n_{ij}, & \text{if } m_{ij} = +\infty \\ m_{ij}, & \text{otherwise} \end{cases}$$

We can model exactly tests of the form $\pm\mathtt{x}_j \pm \mathtt{x}_i \leq c$ and assignments of the form $\mathtt{x}_j := \pm\, \mathtt{x}_i + c$ [12]. For other tests and assignments we revert to the sound identity and to the sound non-deterministic assignment, respectively.

Polyhedra. The *Polyhedra domain* [13], denoted as $\langle P, \sqsubseteq_P \rangle$, is a fully relational numerical property domain, which allows manipulating conjunctions of linear inequalities of the form $\alpha_1\mathtt{x}_1 + \ldots + \alpha_n\mathtt{x}_n \geq \beta$, where $\mathtt{x}_1, \ldots, \mathtt{x}_n$ are variables and $\alpha_i, \beta \in \mathbb{Q}$ (rationals). The abstract operations of the *Polyhedra domain* are defined in [13]. Polyhedra analysis is very expensive, that is, it has time and memory cost exponential in the number of variables in practice.

A property element is represented as a conjunction of linear constraints given in the matrix form $\langle\!|\mathbf{A}, \vec{\mathbf{b}}|\!\rangle$ which consists of a matrix $\mathbf{A} \in \mathbb{Q}^{m \times n}$ and a vector $\vec{\mathbf{b}} \in \mathbb{Q}^m$, where n is the number of variables and m is the number of constraints. This is called the constraint representation of polyhedra elements, and there is another so-called generator representation. Some domain operations can be performed more efficiently using the generator representation only, others based on the constraint representation, and some making use of both. We now present some operations defined using the constraint representation. The concretization function is:

$$\gamma_P(\langle\!|\mathbf{A}, \vec{\mathbf{b}}|\!\rangle) = \{\vec{\mathbf{v}} \in \mathbb{Q}^n \mid \mathbf{A} \cdot \vec{\mathbf{v}} \geq \vec{\mathbf{b}}\}$$

The meet $\sqcap_P$, and the widening ∇_P are defined as:

$$\begin{aligned} &\langle\!|\mathbf{A}_1, \vec{\mathbf{b}_1}|\!\rangle \sqcap_P \langle\!|\mathbf{A}_2, \vec{\mathbf{b}_2}|\!\rangle = \langle\!|\tbinom{\mathbf{A}_1}{\mathbf{A}_2}, \tbinom{\vec{\mathbf{b}_1}}{\vec{\mathbf{b}_2}}|\!\rangle \\ &\langle\!|\mathbf{A}_1, \vec{\mathbf{b}_1}|\!\rangle \nabla_P \langle\!|\mathbf{A}_2, \vec{\mathbf{b}_2}|\!\rangle = \{c \in \langle\!|\mathbf{A}_1, \vec{\mathbf{b}_1}|\!\rangle \mid \langle\!|\mathbf{A}_2, \vec{\mathbf{b}_2}|\!\rangle \sqsubseteq_P \{c\}\} \end{aligned}$$

where c represents one constraint from $\langle\!|\mathbf{A}_1, \vec{\mathbf{b}_1}|\!\rangle$. FILTER_P handles precisely affine inequality tests by adding them to the input polyhedra.

$$\text{FILTER}_P(\sum_i \alpha_i\mathtt{x}_i \geq \beta : \mathit{Exp}, \langle\!|\mathbf{A}, \vec{\mathbf{b}}|\!\rangle : P) = \langle\!|\begin{pmatrix} \mathbf{A} \\ \alpha_1 \ldots \alpha_n \end{pmatrix}, \begin{pmatrix} \vec{\mathbf{b}} \\ \beta \end{pmatrix}|\!\rangle$$

In all other cases, FILTER_P performs the sound identity operation. Likewise, ASSIGN_P handles exactly affine assignments, and it performs the sound non-deterministic assignment for all other (non-affine) cases.

5. Lifted Analysis via Products

Lifted analyses are designed by *lifting* existing single-program analyses to work on program families, rather than on individual programs. They directly analyze program families, without preprocessing them by taking into account variability-specific aspects of program families. In this section, we recall a lifted analysis based on the lifted domain that is the $|\mathbb{K}|$-fold product of an existing (single-program) analysis domain $\mathbb{A}$ [8]. From now on, we assume that the domain $\mathbb{A}$ is equipped with sound operators and transfer functions.

Lifted Domain. The *lifted domain* is defined as $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq}, \dot{\sqcup}, \dot{\sqcap}, \dot{\bot}, \dot{\top} \rangle$, where $\mathbb{A}^{\mathbb{K}}$ is shorthand for the $|\mathbb{K}|$-fold product $\prod_{k \in \mathbb{K}} \mathbb{A}$, that is, there is one separate copy of $\mathbb{A}$ for each valid configuration of $\mathbb{K}$.

Example 2. *Consider the tuple in Fig. 2a, in which components are analysis properties from the Interval domain and* $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$. *Note that to simplify the presentation, we write* $\boldsymbol{x} \geq n$ *short for* $\boldsymbol{x} \mapsto [n, +\infty]$, $\boldsymbol{x} \leq n$ *for* $\boldsymbol{x} \mapsto [-\infty, n]$, $n \leq \boldsymbol{x} \leq n'$ *for* $\boldsymbol{x} \mapsto [n, n']$, *and* $\boldsymbol{x} = n$ *for* $\boldsymbol{x} \mapsto [n, n]$. *In first order logic, the tuple in Fig. 2a can be written as the following disjunctive property:*

$$\begin{gathered}\big(A \wedge B \wedge [\boldsymbol{y} \geq 0, \boldsymbol{x} = 0]\big) \vee \big(A \wedge \neg B \wedge [\boldsymbol{y} \geq 0, \boldsymbol{x} = 0]\big) \vee \\ \big(\neg A \wedge B \wedge [\boldsymbol{y} \geq 0, \boldsymbol{x} = 0]\big) \vee \big(\neg A \wedge \neg B \wedge [0 \leq \boldsymbol{y} \leq 9, \boldsymbol{x} = 0]\big)\end{gathered} \tag{2}$$

Abstract Operations. Given a tuple (lifted domain element) $\overline{a} \in \mathbb{A}^{\mathbb{K}}$, the projection π_k selects the kth component of $\overline{a}$.

Concretization function: Given a tuple $\overline{a} \in \mathbb{A}^{\mathbb{K}}$, the concretization function $\overline{\gamma}$ lifts configuration-wise the function $\gamma_{\mathbb{A}}$ of the domain $\mathbb{A}$, defined as: $\overline{\gamma}(\overline{a}) = \prod_{k \in \mathbb{K}} \gamma_{\mathbb{A}}(\pi_k(\overline{a}))$.

Ordering: Given two tuples $\overline{a_1}, \overline{a_2} \in \mathbb{A}^{\mathbb{K}}$, their approximation ordering $\overline{a_1} \ \dot{\sqsubseteq} \ \overline{a_2}$ is computed by lifting configuration-wise the ordering $\sqsubseteq_{\mathbb{A}}$ of the domain $\mathbb{A}$: $\overline{a_1} \ \dot{\sqsubseteq} \ \overline{a_2} \ \equiv_{def} \ \pi_k(\overline{a_1}) \sqsubseteq_{\mathbb{A}} \pi_k(\overline{a_2})$ for all $k \in \mathbb{K}$.

Join, Meet: Similarly, we lift configuration-wise all other elements of the lattice $\mathbb{A}$. Given $\overline{a_1}, \overline{a_2} \in \mathbb{A}^{\mathbb{K}}$, their join $\overline{a_1} \ \dot{\sqcup} \ \overline{a_2}$ and meet $\overline{a_1} \ \dot{\sqcap} \ \overline{a_2}$ are:

$$\overline{a_1} \ \dot{\sqcup} \ \overline{a_2} = \textstyle\prod_{k \in \mathbb{K}} (\pi_k(\overline{a_1}) \sqcup_{\mathbb{A}} \pi_k(\overline{a_2})), \quad \overline{a_1} \ \dot{\sqcap} \ \overline{a_2} = \textstyle\prod_{k \in \mathbb{K}} (\pi_k(\overline{a_1}) \sqcap_{\mathbb{A}} \pi_k(\overline{a_2}))$$

Top, Bottom: The top $\dot{\top}$ and bottom $\dot{\bot}$ elements are defined as: $\dot{\top} = \prod_{k \in \mathbb{K}} \top_{\mathbb{A}} = (\top_{\mathbb{A}}, \dots, \top_{\mathbb{A}})$, $\dot{\bot} = \prod_{k \in \mathbb{K}} \bot_{\mathbb{A}} = (\bot_{\mathbb{A}}, \dots, \bot_{\mathbb{A}})$

Widening, Narrowing: The widening $\dot{\nabla}$ and the narrowing $\dot{\Delta}$ are:

$$\overline{a_1} \dot{\nabla} \overline{a_2} = \textstyle\prod_{k \in \mathbb{K}} (\pi_k(\overline{a_1}) \nabla_{\mathbb{A}} \pi_k(\overline{a_2})), \quad \overline{a_1} \dot{\Delta} \overline{a_2} = \textstyle\prod_{k \in \mathbb{K}} (\pi_k(\overline{a_1}) \Delta_{\mathbb{A}} \pi_k(\overline{a_2}))$$

Transfer Functions. We now define transfer functions for tests and assignments. There are two types of tests: *expression-based tests* that occur in `while`-s and `if`-s, and *feature-based tests* that occur in `#ifdef`-s.

Expression-based tests: The transfer function $\overline{\text{FILTER}}$ for the expression tests "e" in `while` and `if` statements is designed to handle the test "e" independently on each configuration component $k \in \mathbb{K}$ using the transfer function $\text{FILTER}_{\mathbb{A}}$ of the domain $\mathbb{A}$. That is,

$$\overline{\text{FILTER}}(e : \mathit{Exp}, \overline{a} : \mathbb{A}^{\mathbb{K}}) = \prod_{k\in\mathbb{K}}(\text{FILTER}_{\mathbb{A}}(e, \pi_k(\overline{a})))$$

Feature-based tests: The transfer function $\overline{\text{F-FILTER}}$ for the feature expression tests "θ" in `#ifdef`-s is designed to check the satisfaction of $k \models \theta$ for each configuration component $k \in \mathbb{K}$. If $k \models \theta$ holds, then we keep the corresponding component element, otherwise we replace it with $\bot_{\mathbb{A}}$. That is,

$$\overline{\text{F-FILTER}}(\theta : \mathit{FeatExp}(\mathbb{F}), \overline{a} : \mathbb{A}^{\mathbb{K}}) = \prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\overline{a}), & \text{if } k \models \theta \\ \bot_{\mathbb{A}}, & \text{if } k \not\models \theta \end{cases}$$

Assignments: The transfer functions $\overline{\text{ASSIGN}}$ and $\overline{\text{B-ASSIGN}}$ that handle the assignment "$\texttt{x:=}e$" in the input tuple $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ are defined using $\text{ASSIGN}_{\mathbb{A}}$ and $\text{B-ASSIGN}_{\mathbb{A}}$, respectively, which are independently applied on each component of $\overline{a}$. We have:

$$\overline{\text{ASSIGN}}(\texttt{x:=}e : \mathit{Stm}, \overline{a} : \mathbb{A}^{\mathbb{K}}) = \prod_{k\in\mathbb{K}}(\text{ASSIGN}_{\mathbb{A}}(\texttt{x:=}e, \pi_k(\overline{a})))$$
$$\overline{\text{B-ASSIGN}}(\texttt{x:=}e : \mathit{Stm}, \overline{a}, \overline{a'} : \mathbb{A}^{\mathbb{K}}) = \prod_{k\in\mathbb{K}}(\text{B-ASSIGN}_{\mathbb{A}}(\texttt{x:=}e, \pi_k(\overline{a}), \pi_k(\overline{a'})))$$

***`#ifdef`** statements:* Given the (lifted) forward $\overrightarrow{[\![s]\!]}$ and backward transfer functions $\overleftarrow{[\![s]\!]}$ for statement s, the forward $\overline{\text{IFDEF}}$ and backward transfer functions $\overline{\text{B-IFDEF}}$ for "`#ifdef` (θ) s `#endif`" are defined as:

$$\overline{\text{IFDEF}}(\texttt{\#ifdef } (\theta)\ s\ \texttt{\#endif} : \mathit{Stm}, \overline{a} : \mathbb{A}^{\mathbb{K}}) =$$
$$\overrightarrow{[\![s]\!]}(\overline{\text{F-FILTER}}(\theta, \overline{a})) \mathbin{\dot\sqcup} \overline{\text{F-FILTER}}(\neg\theta, \overline{a})$$
$$\overline{\text{B-IFDEF}}(\texttt{\#ifdef } (\theta)\ s\ \texttt{\#endif} : \mathit{Stm}, \overline{a} : \mathbb{A}^{\mathbb{K}}, \overline{a'} : \mathbb{A}^{\mathbb{K}}) =$$
$$\overline{a} \mathbin{\dot\sqcap} \big(\overleftarrow{[\![s]\!]}(\overline{a'}) \mathbin{\dot\sqcup} \overline{\text{F-FILTER}}(\neg\theta, \dot\top)\big) \mathbin{\dot\sqcap} \big(\overline{a'} \mathbin{\dot\sqcup} \overline{\text{F-FILTER}}(\theta, \dot\top)\big)$$

For the backward transfer function $\overline{\text{B-IFDEF}}$, the postcondition $\overline{a'}$ is obtained because either (1) θ is satisfied in the location before `#ifdef`, so we get the respective precondition by calculating $\overleftarrow{[\![s]\!]}(\overline{a'}) \mathbin{\dot\sqcup} \overline{\text{F-FILTER}}(\neg\theta, \dot\top)$; or

(2) $\neg\theta$ is satisfied in the location before `#ifdef`, so we get the respective precondition by calculating $\overline{a' \dot{\sqcup} \text{F-FILTER}(\theta, \dot{\top})}$. Finally, we find the new, stronger precondition by calculating meet of the precondition $\overline{a}$ found by forward analysis and the two preconditions found by back-propagation (1) and (2).

Lifted Analysis. In the first iteration of the forward lifted analysis, we construct tuples based on the information we have for $\mathbb{K}$. Initially, we build a tuple $\overline{a}_{in} = \dot{\top}$ in which all components are set to $\top_{\mathbb{A}}$ for the first location, whereas for the other locations all components are set to $\bot_{\mathbb{A}}$. The operators of the lifted domain $\mathbb{A}^{\mathbb{K}}$ and forward transfer functions are combined together to analyze program families. The $\overline{a}_{in} = \dot{\top}$ analysis property is then propagated forward from the first location towards the final location taking assignments and tests into account with join and widening around `while`-s. We apply so-called delayed widening, which means we start extrapolating by widening only after some fixed number of iterations we analyze the loop. This way the widening can make a better guess about the `while` behavior. For the backward lifted analysis, we start from a program location $l_{\texttt{final}}$ where we want some invariant $\overline{a}^{sat}_{\texttt{final}}$ to hold. Then, we back-propagate necessary preconditions for the satisfaction of the given invariant $\overline{a}^{sat}_{\texttt{final}}$ from location $l_{\texttt{final}}$ to $l_{\texttt{input}}$ using operators and backward transfer functions of $\mathbb{A}^{\mathbb{K}}$.

As a consequence of the soundness of all operators and transfer functions of the abstract domain $\mathbb{A}$, we can establish the soundness and correctness of the lifted analyses based on tuples (proved in [8]).

Example 3. *Consider the program family P from Section 2. We want to perform* polyhedra *forward lifted analysis of P using the product lifted domain. In order to enforce convergence of the analysis, we apply the widening operator at the loop head, that is, at the location before the* `while` *test. The invariants inferred by our static analysis at program locations from ① to ⑧ are shown in Fig. 4. They represent 4-sized tuples, which contain four polyhedra properties (invariants), one for each configuration.*

Assume that we want the invariant ($\boldsymbol{y} \leq 15$) to hold at location ⑧. The necessary preconditions inferred by polyhedra backward lifted analysis of P at some selected locations from ⑧ to l_{input} are shown in Fig. 5. We can see that $\bot_P$ is found at l_{input} for configuration $A \wedge B$, which means that no input value for $\boldsymbol{y}$ *will make the given invariant hold at location ⑧. On the other hand, precondition $0 \leq \boldsymbol{y} \leq 9$ is found at l_{input} for $\neg A \wedge \neg B$, which means that for all input values of* $\boldsymbol{y}$ *the given invariant will hold at ⑧.* □

$$
\begin{array}{ll}
① & ([\mathtt{y}=\top_P, \mathtt{x}=\top_P], [\mathtt{y}=\top_P, \mathtt{x}=\top_P], [\mathtt{y}=\top_P, \mathtt{x}=\top_P], [\mathtt{y}=\top_P, \mathtt{x}=\top_P]) \\
l_{\mathtt{input}} & ([0\leq\mathtt{y}\leq 9, \mathtt{x}=\top_P], [0\leq\mathtt{y}\leq 9, \mathtt{x}=\top_P], [0\leq\mathtt{y}\leq 9, \mathtt{x}=\top_P], [0\leq\mathtt{y}\leq 9, \mathtt{x}=\top_P]) \\
③ & ([0\leq\mathtt{y}\leq 9, \mathtt{x}=10], [0\leq\mathtt{y}\leq 9, \mathtt{x}=10], [0\leq\mathtt{y}\leq 9, \mathtt{x}=10], [0\leq\mathtt{y}\leq 9, \mathtt{x}=10]) \\
④ & ([20\leq\mathtt{y}+2\mathtt{x}\leq 29, 0\leq\mathtt{x}\leq 10], [10\leq\mathtt{y}+\mathtt{x}\leq 19, 0\leq\mathtt{x}\leq 10], [10\leq\mathtt{y}+\mathtt{x}\leq 19, 0\leq\mathtt{x}\leq 10], [0\leq\mathtt{y}\leq 9, 0\leq\mathtt{x}\leq 10]) \\
⑤ & ([18\leq\mathtt{y}+2\mathtt{x}\leq 27, -1\leq\mathtt{x}\leq 9], [9\leq\mathtt{y}+\mathtt{x}\leq 18, -1\leq\mathtt{x}\leq 9], [9\leq\mathtt{y}+\mathtt{x}\leq 18, -1\leq\mathtt{x}\leq 9], [0\leq\mathtt{y}\leq 9, -1\leq\mathtt{x}\leq 9]) \\
⑥ & ([19\leq\mathtt{y}+2\mathtt{x}\leq 28, -1\leq\mathtt{x}\leq 9], [10\leq\mathtt{y}+\mathtt{x}\leq 19, -1\leq\mathtt{x}\leq 9], [9\leq\mathtt{y}+\mathtt{x}\leq 18, -1\leq\mathtt{x}\leq 9], [0\leq\mathtt{y}\leq 9, -1\leq\mathtt{x}\leq 9]) \\
⑦ & ([20\leq\mathtt{y}+2\mathtt{x}\leq 29, -1\leq\mathtt{x}\leq 9], [10\leq\mathtt{y}+\mathtt{x}\leq 19, -1\leq\mathtt{x}\leq 9], [10\leq\mathtt{y}+\mathtt{x}\leq 19, -1\leq\mathtt{x}\leq 9], [0\leq\mathtt{y}\leq 9, -1\leq\mathtt{x}\leq 9]) \\
⑧ & ([20\leq\mathtt{y}\leq 29, \mathtt{x}=0], [10\leq\mathtt{y}\leq 19, \mathtt{x}=0], [10\leq\mathtt{y}\leq 19, \mathtt{x}=0], [0\leq\mathtt{y}\leq 9, \mathtt{x}=0])
\end{array}
$$

Figure 4: Tuple-based forward polyhedra invariants at locations from ① to ⑧ of P.

$$
\begin{array}{ll}
⑧ & ([\bot_P], [10\leq\mathtt{y}\leq 15, \mathtt{x}=0], [10\leq\mathtt{y}\leq 15, \mathtt{x}=0], [0\leq\mathtt{y}\leq 9, \mathtt{x}=0]) \\
⑦ & ([\bot_P], [10\leq\mathtt{y}+\mathtt{x}\leq 15, -1\leq\mathtt{x}\leq 9], [10\leq\mathtt{y}+\mathtt{x}\leq 15, -1\leq\mathtt{x}\leq 9], [0\leq\mathtt{y}\leq 9, -1\leq\mathtt{x}\leq 9]) \\
③ & ([\bot_P], [10\leq\mathtt{y}+\mathtt{x}\leq 15, \mathtt{x}=10], [10\leq\mathtt{y}+\mathtt{x}\leq 15, \mathtt{x}=10], [0\leq\mathtt{y}\leq 9, \mathtt{x}=10]) \\
l_{\mathtt{input}} & ([\bot_P], [0\leq\mathtt{y}\leq 5], [0\leq\mathtt{y}\leq 5], [0\leq\mathtt{y}\leq 9])
\end{array}
$$

Figure 5: Tuple-based backward polyhedra invariants at selected locations of P.

6. Lifted Analysis via Binary Decision Diagrams

In this section, we propose a new efficient lifted analysis by introducing the lifted domain of binary decision diagrams (BDDs), denoted as $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$. We exploit the well-known efficiency of BDDs, introduced by Bryant [19] for representing Boolean functions, and adapt them to represent the lifted domain $\mathbb{A}^{\mathbb{K}}$ more concisely. The elements of the domain $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ are disjunctions of leaf nodes that belong to an existing (single-program) analysis domain $\mathbb{A}$, which are separated by the values of Boolean features from $\mathbb{F}$ organized in the decision nodes. Therefore, we encapsulate the set $\mathbb{K}$ into the decision nodes of a BDD where each top-down path represents one or several configurations from $\mathbb{K}$, and we store in each leaf node the property generated from the variants derived by the corresponding configurations.

Lifted Domain. We first consider a simpler form of binary decision diagrams called *binary decision trees* (BDTs). A *binary decision tree* (BDT) $t \in \mathbb{T}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ over the set $\mathbb{F}$ of features, the set $\mathbb{K}$ of valid configurations,

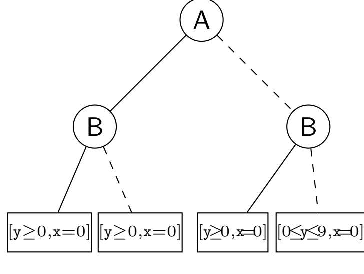

Figure 6: A BDT.

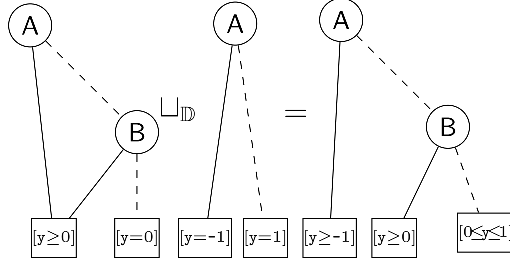

Figure 7: The join of two BDDs.

and the leaf abstract domain $\mathbb{A}$ is either a leaf node $\langle\!|a|\!\rangle$, with $a \in \mathbb{A}$ and $\mathbb{F} = \mathbb{K} = \emptyset$, or $[\![A : tl, tr]\!]$, where A is *the smallest element* of $\mathbb{F}$ with respect to its ordering, tl is the left subtree of t representing its true branch, and tr is the right subtree of t representing its false branch, such that $tl, tr \in \mathbb{T}(\mathbb{F}\backslash\{A\}, \mathbb{K}_{\backslash\{A\}}, \mathbb{A})$. Note that, $\mathbb{F} = \{A_1, \ldots, A_n\}$ is a totally ordered set with ordering: $A_1 < \ldots < A_n$, and $\mathbb{K}_{\backslash\{A\}}$ denotes the removal of feature A from each configuration in $\mathbb{K}$. The left and right subtrees are either both leafs or both rooted at decision nodes labeled with the same feature.

Example 4. *The binary decision tree in Fig. 6 has decision nodes labeled with features A and B, and leaf nodes are Interval properties. In first order logic, this tree expresses the same formula as one in Eqn. (2), Example 2.* $\square$

However, BDTs contain some redundancy. There are three optimizations we can apply to BDTs in order to reduce their representation [19, 20]:

(1) Removal of duplicate leaves. If a tree contains more than one same leaf, we redirect all edges that point to such leaves to just one of them.

(2) Removal of redundant tests. If both outgoing edges of a node A_i point to the same node A_j, we eliminate A_i by sending all its incoming edges to A_j.

(3) Removal of duplicate non-leaves. If two nodes A_i and A_j are the roots of identical subtrees, we eliminate A_i by sending all its incoming edges to A_j.

If we apply reductions (**1**)-(**3**) to a binary decision tree $t \in \mathbb{T}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ until no further reductions are possible, then the result is a *reduced binary decision diagram*. Thanks to the sharing of information enabled by the reductions

(**1**)-(**3**), BDDs are quite compact representation of disjunctive analysis properties from $\mathbb{A}^{\mathbb{K}}$. Moreover, if the ordering on the Boolean variables from $\mathbb{F}$ occurring on any path is fixed to the ordered list $[A_1, \dots, A_n]$, then we obtain a *reduced ordered binary decision diagram* $d \in \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$. [3] Notice that reduced ordered binary decision diagrams (BDDs) have a *canonical form*, which means that any disjunctive analysis property from the lifted domain $\mathbb{A}^{\mathbb{K}}$ can be represented in an *unique way* by a BDD from $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$. Furthermore, by applying the abstract operations and transfer functions from $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ on BDDs in canonical forms, we also obtain BDDs in canonical form.

Example 5. *After applying reductions (**1**)-(**3**) to the binary decision tree in Fig. 6, the resulting reduced ordered binary decision diagram (BDD) is shown in Fig. 3a.* □

Abstract Operations. The abstract operations on $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ are implemented by recursive traversal of the operand BDDs and by using hashtables to store and reuse already computed subtrees [19]. The basic operations are:

- $\texttt{apply}_2(\texttt{op}, d_1, d_2)$ which lifts any binary operation `op` from the domain $\mathbb{A}$ to BDDs, thus computing the reduced ordered binary decision diagram of "$d_1 \, \texttt{op} \, d_2$".

- $\texttt{apply}_1(\texttt{op}, d)$ which applies any unary operation `op` from the domain $\mathbb{A}$ to the leaf nodes of the BDD d, thus computing the reduced ordered binary decision diagram of "$\texttt{op} \, d$".

- $\texttt{meet_condition}(d, b)$ which restricts the top-down paths (Boolean part) of the BDD d to those paths that satisfy the condition b.

With the help of $\texttt{apply}_2$, $\texttt{apply}_1$, and `meet_condition`, abstract operations and transfer functions from $\mathbb{A}$ are lifted to $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$.

Concretization function: Given a BDD $d \in \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$, the concretization function $\gamma_{\mathbb{D}}$ returns $\gamma_{\mathbb{A}}(a)$ for $k \in \mathbb{K}$, where k satisfies the constraints reached along the top-down path to the leaf node $a \in \mathbb{A}$. More formally, $\gamma_{\mathbb{D}}(d) = \overline{\gamma}_{\mathbb{D}}[\mathbb{K}](d)$, where function $\overline{\gamma}_{\mathbb{D}}$ is defined as:

$$\overline{\gamma}_{\mathbb{D}}[C](\langle\!| a |\!\rangle) = \textstyle\prod_{k \models C} \gamma_{\mathbb{A}}(a), \quad \overline{\gamma}_{\mathbb{D}}[C](\llbracket A : tl, tr \rrbracket) = \overline{\gamma}_{\mathbb{T}}[C \wedge A](tl) \times \overline{\gamma}_{\mathbb{T}}[C \wedge \neg A](tr)$$

[3] We abbreviate here a reduced ordered binary decision diagram with only BDD, but this term is also abbreviated with ROBDD in the literature [19, 20]

Ordering: The approximation ordering $d_1 \sqsubseteq_{\mathbb{D}} d_2$ is defined as:

$$d_1 \sqsubseteq_{\mathbb{D}} d_2 \quad \equiv_{def} \quad \texttt{apply}_2(\lambda(a_1, a_2).a_1 \sqsubseteq_{\mathbb{A}} a_2, d_1, d_2)$$

If the resulting BDD of the above operation is the constant true, then $d_1 \sqsubseteq_{\mathbb{D}} d_2$ holds.

Join, Meet: Similarly, we compute the other binary operations. For join $d_1 \sqcup_{\mathbb{D}} d_2$ and meet $d_1 \sqcap_{\mathbb{D}} d_2$, we have:

$$\begin{aligned} d_1 \sqcup_{\mathbb{D}} d_2 &= \texttt{apply}_2(\lambda(a_1, a_2).a_1 \sqcup_{\mathbb{A}} a_2, d_1, d_2), \\ d_1 \sqcap_{\mathbb{D}} d_2 &= \texttt{apply}_2(\lambda(a_1, a_2).a_1 \sqcap_{\mathbb{A}} a_2, d_1, d_2) \end{aligned}$$

For example, Fig. 7 shows the join of two BDDs from $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$. We can observe that a loss of precision in the join operation can also have a negative effect on the overall sharing possibilities. Thus, although both arguments of the join in Fig. 7 have two leaf nodes, the resulting BDD has three leaves.

Top, Bottom: The BDDs $\top_{\mathbb{D}}$ and $\bot_{\mathbb{D}}$ representing the top and bottom elements in $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ have only one leaf node $\top_{\mathbb{A}}$ and $\bot_{\mathbb{A}}$, respectively.

Widening, Narrowing: We have:

$$\begin{aligned} d_1 \nabla_{\mathbb{D}} d_2 &= \texttt{apply}_2(\lambda(a_1, a_2).a_1 \nabla_{\mathbb{A}} a_2, d_1, d_2), \\ d_1 \Delta_{\mathbb{D}} d_2 &= \texttt{apply}_2(\lambda(a_1, a_2).a_1 \Delta_{\mathbb{A}} a_2, d_1, d_2) \end{aligned}$$

Transfer Functions. We proceed by defining transfer functions for expression-based and feature-based tests as well as for assignments and `#ifdef`-s.

Expression-based tests: The transfer function $\text{FILTER}_{\mathbb{D}}$ for the expression tests "e" in `while`-s and `if`-s is implemented by handling "e" at each leaf node of the input BDD using $\texttt{apply}_1$. That is,

$$\text{FILTER}_{\mathbb{D}}(e : \mathit{Exp}, d : \mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})) = \texttt{apply}_1(\lambda a.\text{FILTER}_{\mathbb{A}}(e, a), d)$$

Feature-based tests: The transfer function $\text{F-FILTER}_{\mathbb{D}}$ for the feature expression tests "θ" in `#ifdef`-s is implemented using the `meet_condition` operation. We have

$$\text{F-FILTER}_{\mathbb{D}}(\theta : \mathit{FeatExp}(\mathbb{F}), d : \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})) = \texttt{meet_condition}(d, \theta)$$

Assignments: The transfer functions $\text{ASSIGN}_{\mathbb{D}}$ and $\text{B-ASSIGN}_{\mathbb{D}}$ for the assignment "$\texttt{x:=}e$" are implemented by applying $\text{ASSIGN}_{\mathbb{A}}$ and $\text{B-ASSIGN}_{\mathbb{A}}$, respectively, at each leaf node of the input BDD using $\texttt{apply}_1$.

$$\begin{aligned} &\text{ASSIGN}_{\mathbb{D}}(\texttt{x:=}e : \mathit{Stm}, d : \mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})) = \texttt{apply}_1(\lambda a.\text{ASSIGN}_{\mathbb{A}}(\texttt{x:=}e, a), d) \\ &\text{B-ASSIGN}_{\mathbb{D}}(\texttt{x:=}e, d, d') = \texttt{apply}_2(\lambda(a, a').\text{B-ASSIGN}_{\mathbb{A}}(\texttt{x:=}e, a, a'), d, d') \end{aligned}$$

#ifdef statements: Given the (lifted) forward $\overrightarrow{[\![s]\!]}_{\mathbb{D}}$ and backward transfer functions $\overleftarrow{[\![s]\!]}_{\mathbb{D}}$ for statement s, the forward $\mathrm{IFDEF}_{\mathbb{D}}$ and backward transfer functions $\mathrm{B\text{-}IFDEF}_{\mathbb{D}}$ for "`#ifdef` (θ) s `#endif`" are defined as:

$$\begin{array}{l}\mathrm{IFDEF}_{\mathbb{D}}(\texttt{\#ifdef }(\theta)\ s\ \texttt{\#endif} : Stm, d : \mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})) = \\ \qquad\qquad \overrightarrow{[\![s]\!]}_{\mathbb{D}}(\text{F-FILTER}_{\mathbb{D}}(\theta, d)) \sqcup_{\mathbb{D}} \text{F-FILTER}_{\mathbb{D}}(\neg\theta, d) \\ \mathrm{B\text{-}IFDEF}_{\mathbb{D}}(\texttt{\#ifdef }(\theta)\ s\ \texttt{\#endif} : Stm, d : \mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A}), d' : \mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})) = \\ \quad d \sqcap_{\mathbb{D}} \big(\overleftarrow{[\![s]\!]}_{\mathbb{D}}(d') \sqcup_{\mathbb{D}} \text{F-FILTER}_{\mathbb{D}}(\neg\theta, \top_{\mathbb{D}})\big) \sqcap_{\mathbb{D}} \big(d' \sqcup_{\mathbb{D}} \text{F-FILTER}_{\mathbb{D}}(\theta, \top_{\mathbb{D}})\big)\end{array}$$

Lifted Analysis. A *path* in a BDD corresponds to one or several configurations. We say that a path is *valid* if the corresponding configurations are valid and belong to $\mathbb{K}$. In the first iteration of the forward lifted analysis, we build BDDs with only one leaf node that can be reached along only valid paths. For the first program location the leaf node is $\top_{\mathbb{A}}$, whereas for the other program locations the leaf node is $\bot_{\mathbb{A}}$. Thus, in the first iteration, the BDD for the first location is $d_{in} = \texttt{meet_condition}(\top_{\mathbb{D}}, \vee_{k\in\mathbb{K}} k)$, whereas for the other locations is $\texttt{meet_condition}(\bot_{\mathbb{D}}, \vee_{k\in\mathbb{K}} k)$. Note that, if $\mathbb{K} = 2^{\mathbb{F}}$ then $\vee_{k\in\mathbb{K}} k \equiv$ true, so $d_{in} = \top_{\mathbb{D}}$ for the first location and $\bot_{\mathbb{D}}$ for the others.

The operators of the lifted domain $\mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})$ and transfer functions are combined together to analyze forward and backward program families. The non-$\bot_{\mathbb{D}}$ analysis properties are propagated forward towards the final location taking assignments and tests into account with join and widening around `while`-s. For the backward analysis, the invariant $d^{sat}_{\texttt{final}}$ we want to establish at location $l_{\texttt{final}}$ is propagated backward towards the initial location.

We establish correctness of the lifted analysis based on $\mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})$ by showing that it produces identical results with tuple-based domain $\mathbb{A}^{\mathbb{K}}$. Let $\overrightarrow{[\![s]\!]}_{\mathbb{D}}$ and $\overrightarrow{[\![s]\!]}$ denote forward transfer functions of statement s in $\mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})$ and $\mathbb{A}^{\mathbb{K}}$ respectively, while $\overleftarrow{[\![s]\!]}_{\mathbb{D}}$ and $\overleftarrow{[\![s]\!]}$ denote backward transfer functions of s in $\mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})$ and $\mathbb{A}^{\mathbb{K}}$ respectively.

Theorem 6. *Let $\overline{a}_{in} \in \mathbb{A}^{\mathbb{K}}, d_{in} \in \mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})$ be initial invariants in the first program location, and let $\overline{a}^{sat}_{final} \in \mathbb{A}^{\mathbb{K}}, d^{sat}_{final} \in \mathbb{D}(\mathbb{F},\mathbb{K},\mathbb{A})$ be final invariants we want to establish. We have:*

$$\gamma_{\mathbb{D}}\big(\overrightarrow{[\![s]\!]}_{\mathbb{D}}(d_{in})\big) = \overline{\gamma}\big(\overrightarrow{[\![s]\!]}(\overline{a}_{in})\big); \quad \gamma_{\mathbb{D}}\big(\overleftarrow{[\![s]\!]}_{\mathbb{D}}(d^{sat}_{final})\big) = \overline{\gamma}\big(\overleftarrow{[\![s]\!]}(\overline{a}^{sat}_{final})\big)$$

Proof 1. *The proof is by induction on structure of s. Assume $\gamma_{\mathbb{D}}(d) = \overline{\gamma}(\overline{a})$ (*). We consider the two most interesting cases for forward lifted analysis.*

Case `x:=`e**.** $\overline{ASSIGN}(\texttt{x:=}e, \overline{a})$ *applies* $ASSIGN_{\mathbb{A}}(\texttt{x:=}e, d)$ *to each component of* $\overline{a}$*. On the other hand,* $ASSIGN_{\mathbb{D}}(\texttt{x:=}e, d)$ *applies* $ASSIGN_{\mathbb{A}}(\texttt{x:=}e, d)$ *to each leaf* a *in* d*. The proof follows by correctness of assumption (*).*

Case `#if` (θ) s `#endif`**.** *Transfer functions for* `#if` *are identical in both lifted domains. We only need to show that* $\overline{F\text{-}FILTER}(\theta, \overline{a})$ *and* $F\text{-}FILTER_{\mathbb{D}}(\theta, d)$ *are identical. This can be shown by induction on* θ*. Assume that* θ *is an atomic constraint.* $\overline{F\text{-}FILTER}(\theta, \overline{a})$ *keeps only those components* k *of* $\overline{a}$ *such that* $k \models \theta$*. On the other hand,* $F\text{-}FILTER_{\mathbb{D}}(\theta, d)$ *first restricts all top-down paths of the BDD* d *to satisfy the condition* θ*. Thus, it keeps only those leaf nodes that satisfy the newly generated constraints from* θ*. The other cases are similar.*

Similarly, we prove the corresponding cases for backward lifted analysis.

Example 7. *Consider the program family* P *from Section 2. We want to perform interval forward lifted analysis of* P *using the lifted domain* $\mathbb{D}(\mathbb{F}, \mathbb{K}, I)$*, where* $\mathbb{F} = \{A, B\}$*,* $\mathbb{K} = 2^{\{A,B\}}$*, and the ordering of features is* $A < B$*. The final analysis results at program locations from* ① *to* ⑧ *are shown in Fig. 8. Compared to the analysis results obtained using the lifted domain* $I^{\mathbb{K}}$ *(see Section 2 and Example 3), which represent 4-sized tuples, it is obvious that results based on the lifted domain* $\mathbb{D}(\mathbb{F}, \mathbb{K}, I)$ *have a lot of sharing of the redundant information in all program locations. For example, in program locations from* ① *to* ④*, there is only one interval property (leaf node) that is shared by all four configurations. In locations* ⑤ *and* ⑧ *there are two interval properties, while in* ⑥ *and* ⑦ *there are three interval properties.*

Similar possibilities for sharing are observed if we perform backward lifted analysis using the lifted domain $\mathbb{D}(\mathbb{F}, \mathbb{K}, I)$*.* □

7. Implementation and Evaluation

We now evaluate our approach for speeding up lifted analysis based on abstract interpretation. It consists of running the proposed tuple-based and BDD-based lifted analyses on several `#ifdef`-enriched C case studies. In particular, we ask the following research questions here:

RQ1: How efficient are our BDD-based lifted analyses compared to the corresponding tuple-based lifted analyses?

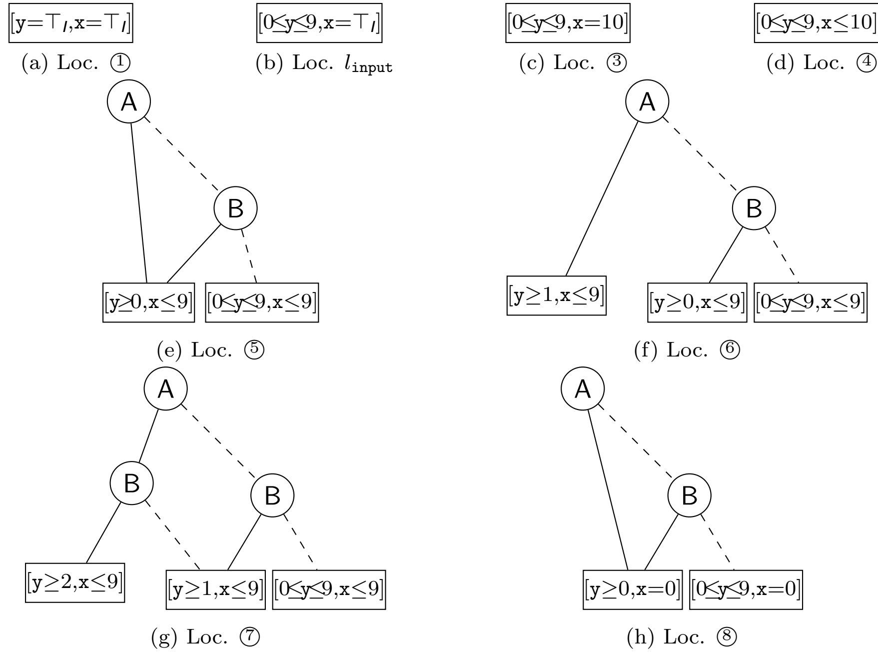

Figure 8: BDD-based forward interval properties at locations from ① to ⑧ of P.

RQ2: Can BDD-based lifted analyses turn some previously infeasible tuple-based lifted analyses tasks into feasible ones?

RQ3: Can we find practical application scenarios of using our lifted analyses to efficiently verify and analyze `#ifdef`-enriched C programs?

Implementation. We have implemented our lifted abstract domains of tuples $\mathbb{A}^{\mathbb{K}}$ and binary decision diagrams $\mathbb{D}(\mathbb{K}, \mathbb{F}, \mathbb{A})$ into a prototype static analyzer. The abstract domains $\mathbb{A}$ for encoding properties of leaf nodes are based on intervals, octagons, and polyhedra. The operators and transfer functions for the domains $\mathbb{A}$: intervals, octagons, and polyhedra, are provided by the APRON library [11]. The operators and transfer functions for the binary decision diagram domains that combine Boolean formulae and APRON domains are provided by the BDDAPRON library [10]. The prototype tool is written in OCaml and consists of around 6K lines of code. It accepts programs written in a subset of C with `#ifdef` directives, but without `struct` and `union` types. It provides only a limited support of arrays and pointers, and the only basic data type is mathematical integers. As output, the tool infers numerical invariants in all program locations. The tool performs either

one forward reachability analysis for inferring invariants, or a combination of preliminary forward and a backward analysis for inferring necessary preconditions that a given invariant holds. The analyses proceed by structural induction on the program syntax, iterating `while`-s until a fixed point is reached. They compute the unique solutions which to every program location assign an element from the lifted domain.

Experimental setup. All experiments are executed on a 64-bit Intel®Core^{TM} i7-8700 CPU@3.20GHz × 12, Ubuntu 18.04.5 LTS, with 8 GB memory. The reported times represent the average runtime of five independent executions. We report the times (in *seconds*) needed for actual forward analysis to be performed. All reported times represent CPU-times measured via Sys.time function of OCaml. The implementation, benchmarks, and all results obtained from our experiments are available from: `https://github.com/aleksdimovski/lifted_analyzer` and `https://zenodo.org/record/4090354#.X4gKCNUzbIU`. In our experiments, we use three instances of our lifted analyses based on BDDs: $\overline{\mathcal{A}}_{\mathbb{D}}(I)$, $\overline{\mathcal{A}}_{\mathbb{D}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ which use intervals, octagons, and polyhedra domains for the leaf nodes, respectively. We also consider three lifted analyses based on tuples: $\overline{\mathcal{A}}_{\Pi}(I)$, $\overline{\mathcal{A}}_{\Pi}(O)$, and $\overline{\mathcal{A}}_{\Pi}(P)$, which use intervals, octagons, and polyhedra domains for the component elements.

Benchmarks. For our experiment, we use a dozen of C programs extracted from five different folders (categories) of the 8th International Competition on Software Verification (SV-COMP 2019) (`https://sv-comp.sosy-lab.org/2019/`) as well as from the real-world BusyBox project (`https://busybox.net`). The folders from SV-COMP we consider are: `loops`, `loop-invgen` (`invgen` for short), `loop-lit` (`lit` for short), `termination-crafted` (`crafted` for short), and `termination-restricted` (`restrict` for short). Due to the limitations in the current front-end of the tool we were not able to analyze those programs that heavily use pointers, `struct` and `union` types. In the case of SV-COMP, we have first selected some numerical programs with integers that our tool can handle, and then we have manually added variability (features and `#ifdef` directives) in each of them. We have experimented with benchmarks that contain four or five features, since the ability for sharing and the speed ups of BDD-based lifted analysis should be visible for them. For program families with higher number of features, the tuple-based lifted analysis very quickly becomes impractically slow (see the paragraph

"From infeasible to feasible analyses"). The presence conditions in generated `#ifdef`-s are with different complexities, from atomic to more complex feature expressions. The `#ifdef`-s are inserted in different locations in the code. More specifically, sometimes they are inserted in the input sections so that input variables are initialized with different intervals based on the enabled presence conditions (e.g., see program families P_1 and P_2 from the paragraph "Application scenarios"). In some cases, the `#ifdef`-s are inserted in the middle of the code, like in the bodies of `while`-s and `if`-s (e.g., see program family P from Section 2). Some `#ifdef`-s are also inserted in the end of the code, like in the final assertions to be checked. In the case of BusyBox, we have first selected some programs with Boolean features, and then we have simplified those programs so that our tool can handle them. For example, any reference to a pointer or a library function is replaced with $? = [-\infty, +\infty]$. All features are unconstrained, so the set of valid configurations is $\mathbb{K} = 2^{\mathbb{F}}$. Finally, we have analyzed selected programs using our prototype lifted static analyzer. Table 1 presents characteristics of the selected benchmarks we analyzed: the file name (Benchmark), folder where it is located (folder), number of features ($|\mathbb{F}|$), and total number of lines of code (LOC).

Performance. We now present the performance results of our empirical study and discuss implications. Table 1 compares the performances of analyzing our benchmarks by using different versions of our lifted analyses based on BDDs and on tuples. For each analysis version based on BDDs, there are two columns. In the first column, Time, we report the running time in seconds to analyze the given benchmark using the following analyses versions based on BDDs: $\overline{\mathcal{A}}_{\mathbb{D}}(I)$, $\overline{\mathcal{A}}_{\mathbb{D}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$. In the second columnn, Improve, we report how many times a BDD-based analysis is faster than the corresponding baseline analysis based on tuples ($\overline{\mathcal{A}}_{\mathbb{D}}(I)$ vs. $\overline{\mathcal{A}}_{\Pi}(I)$, $\overline{\mathcal{A}}_{\mathbb{D}}(O)$ vs. $\overline{\mathcal{A}}_{\Pi}(O)$, and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ vs. $\overline{\mathcal{A}}_{\Pi}(P)$). The performance results match expectations. They confirm that sharing is indeed effective and especially so for large values of $|\mathbb{K}|$. All BDD-based versions achieve significant speed-ups compared to the tuple-based versions, which range from 2.6 to 14.4 times for programs with four features and from 5.3 to 11.8 times for programs with five features (addresses **RQ1**). Of course, the speed up depends on how much sharing is possible for a given program. We can see that $\overline{\mathcal{A}}_{\mathbb{D}}(I)$ is the fastest, then it comes $\overline{\mathcal{A}}_{\mathbb{D}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ is the slowest but the most precise version.

Table 1: Performance results for lifted static analyses based on binary decision diagrams vs. lifted static analyses based on tuples (which are used as baseline). Times in sec.

Bench.	folder	$\lvert\mathbb{F}\rvert$	LOC	$\overline{\mathcal{A}}_{\mathbb{D}}(I)$		$\overline{\mathcal{A}}_{\mathbb{D}}(O)$		$\overline{\mathcal{A}}_{\mathbb{D}}(P)$	
				TIME	IMPROVE	TIME	IMPROVE	TIME	IMPROVE
down.c	invgen	4	25	0.0041	4.4×	0.0067	7.7×	0.0087	7×
half2.c	invgen	4	30	0.0047	4.6×	0.0116	7.9×	0.0133	6.4×
heapsort.c	invgen	4	60	0.0124	7.6×	0.0425	14.4×	0.0547	12.7×
seq.c	invgen	4	40	0.0076	5.7×	0.0355	8.6×	0.0333	6.3×
eq1.c	loops	4	20	0.0109	2.6×	0.0196	4.5×	0.0252	4.2×
eq2.c	loops	5	20	0.0064	7.7×	0.0127	11.8×	0.0138	11.4×
sum01*.c	loops	4	20	0.0084	5.2×	0.0236	9.9×	0.0269	7.5×
count_up_down*.c	loops	4	30	0.0023	4.2×	0.0036	5.8×	0.0051	5.9×
hhk2008.c	lit	5	30	0.0048	6.6×	0.0101	10.7×	0.0135	8.9×
gsv2008.c	lit	5	25	0.0040	7.5×	0.0084	11.1×	0.0095	10.7×
gcnr2008.c	lit	4	30	0.0085	3.0×	0.0123	6.2×	0.0301	6.3×
bhmr2007.c	lit	4	30	0.0043	5.6×	0.0161	8.4×	0.0116	7.4×
GCD4.c	restrict	4	30	0.0031	6.3×	0.0042	10.1×	0.0085	8.7×
UpAndDown.c	restrict	4	30	0.0053	4.8×	0.0077	5.3×	0.0207	4.7×
Log.c	restrict	4	35	0.0041	5.7×	0.0081	9.7×	0.0101	8.8×
java_Sequence.c	restrict	4	25	0.0042	3×	0.0065	4.1×	0.0085	4.6×
Toulouse*.c	crafted	4	75	0.0078	4.1×	0.0116	4.4×	0.0201	5.2×
TelAviv*.c	crafted	4	50	0.0035	3.1×	0.0037	3.4×	0.0078	4.7×
Mysore.c	crafted	5	35	0.0046	5.3×	0.0054	5.8×	0.0118	7.4×
realpath.c	BusyBox	4	50	0.0030	3.9×	0.0057	4.8×	0.0091	4.7×
copyfd.c	BusyBox	4	90	0.0079	5.6×	0.0264	8.3×	0.0347	6.7×

From infeasible to feasible analyses. For very large values of $\lvert\mathbb{K}\rvert$, tuple-based lifted analyses may become impractically slow or even infeasible since they work on $\lvert\mathbb{K}\rvert$-sized tuples. In that case, we can use BDD-based lifted analyses with improved representation via sharing to obtain feasible lifted analyses.

As an experiment, we have tested the limits of the tuple-based lifted analysis $\overline{\mathcal{A}}_{\Pi}(P)$. We took a method, $\texttt{foo}_n()$, which contains n features $\texttt{A}_1, \ldots, \texttt{A}_n$ and n sequentially composed `#ifdef`-s of the form `#ifdef` $(\texttt{A}_i)$ `i := i+1` `#endif`. For example, the method $\texttt{foo}_3()$ with three features $\texttt{A}_1$, $\texttt{A}_2$, and $\texttt{A}_3$ is:

$$\Big(\overbrace{[\texttt{i}=3]}^{\texttt{A}_1\wedge\texttt{A}_2\wedge\texttt{A}_3}, \overbrace{[\texttt{i}=2]}^{\texttt{A}_1\wedge\texttt{A}_2\wedge\neg\texttt{A}_3}, \overbrace{[\texttt{i}=2]}^{\texttt{A}_1\wedge\neg\texttt{A}_2\wedge\texttt{A}_3}, \overbrace{[\texttt{i}=1]}^{\texttt{A}_1\wedge\neg\texttt{A}_2\wedge\neg\texttt{A}_3},$$
$$\overbrace{[\texttt{i}=2]}^{\neg\texttt{A}_1\wedge\texttt{A}_2\wedge\texttt{A}_3}, \overbrace{[\texttt{i}=1]}^{\neg\texttt{A}_1\wedge\texttt{A}_2\wedge\neg\texttt{A}_3}, \overbrace{[\texttt{i}=1]}^{\neg\texttt{A}_1\wedge\neg\texttt{A}_2\wedge\texttt{A}_3}, \overbrace{[\texttt{i}=0]}^{\neg\texttt{A}_1\wedge\neg\texttt{A}_2\wedge\neg\texttt{A}_3}\Big)$$

Figure 9: $\overline{\mathcal{A}}_{\Pi}(P)$ results at loc. ⑤ of $\texttt{foo}_3()$.

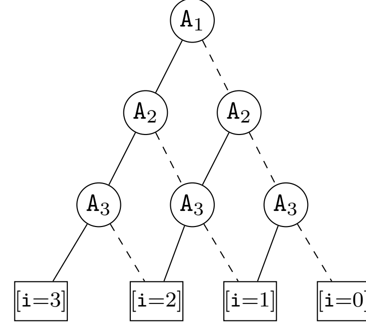

Figure 10: $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ results at loc. ⑤ of $\texttt{foo}_3()$.

Table 2: The performance results of analyzing $\texttt{foo}_n$.

n	$\overline{\mathcal{A}}_{\Pi}(P)$	$\overline{\mathcal{A}}_{\mathbb{D}}(P)$	Improve
3	0.0017	0.0007	2.4×
5	0.0103	0.0022	4.7×
10	0.5927	0.0511	11.6×
15	34.157	1.6939	20.2×
16	90.38	2.2196	40.7×
17	infeasible	2.5977	∞×
18	infeasible	3.0433	∞×

```
①    int i := 0;
②    #ifdef (A1) i := i+1; #endif
③    #ifdef (A2) i := i+1; #endif
④    #ifdef (A3) i := i+1; #endif ⑤
```

Depending on which features are enabled in a configuration, the variable `i` in location ⑤ can have a value in the range from 0 (when $\texttt{A}_1$, $\texttt{A}_2$, and $\texttt{A}_3$ are all disabled) to 3 (when $\texttt{A}_1$, $\texttt{A}_2$, and $\texttt{A}_3$ are all enabled). The analysis results in program location ⑤ obtained using $\overline{\mathcal{A}}_{\Pi}(P)$ and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ are shown in Fig. 9 and Fig. 10, respectively. The tuple-based $\overline{\mathcal{A}}_{\Pi}(P)$ uses 8 interval properties (components), while the BDD-based $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ uses only 4 interval properties (leaf nodes) that are shared between all 8 configurations.

We have gradually added unconstrained variability into $\texttt{foo}_3$ by adding optional features and by sequentially composing `#ifdef` statements guarded by all existing features. In general, the number of interval properties used by $\overline{\mathcal{A}}_{\Pi}(P)$ grows exponentially (that is, 2^n) with n, whereas the number

of interval properties used by $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ in the final program location grows linearly (that is, $n+1$) with n. The performance results of analyzing $\mathtt{foo}_n$, for different values of n, using $\overline{\mathcal{A}}_{\Pi}(P)$ and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ are shown in Table 2. Already for $|\mathbb{K}| = 2^{16}$ =65,536 configurations, the analysis $\overline{\mathcal{A}}_{\Pi}(P)$ took 90.4 seconds, while the analysis $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ took only 2.2 seconds thus giving speed up of 40.7 times. For $|\mathbb{K}| = 2^{17}$ =131,172, $\overline{\mathcal{A}}_{\Pi}(P)$ crashes with an out-of-memory error, while $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ ends in less than 2.6 seconds, producing a BDD with 18 leaf nodes: one node for each $\mathtt{i} \in \{0, \ldots, 17\}$. Hence, we can conclude that BDDs can not only greatly speed up lifted analyses, but also turn previously infeasible analyses into feasible by giving very compact representation of lifted analysis results, thus effectively eliminating the exponential blowup (addresses **RQ2**).

Application scenarios. To illustrate the effectiveness of our approach, we consider some practical applications to program verification, and we present the results produced by our analyzer (addresses **RQ3**).

Let us consider the program family P_1:

```
①       #ifdef (A ∨ B ∨ C) int x := input([10, 20]);
②       #else int x := input([0, 20]); #endif
③       int y := input([0, 1]);
l_input  if (y ≥ 1) x := −x;
⑤       assert(x!=0); ⑥
```

which has three features A, B, and C. When at least one feature is on, the program stores a random value from $[10, 20]$ in x. Otherwise, when all features are off, it stores a random value from $[0, 20]$ in x. Then, depending on the input value of the variable y, x is negated or not. We want to check when the assertion at location ⑤ is valid. For example, later on in the program, there may be divisions by x (e.g. n/x). In this way, we can verify that there are no divisions-by-zero.

The lifted analysis using the Interval domain will take at location $l_{\mathtt{input}}$ the join of the then branch of conditional and its else branch. Hence, the BDD found in location ⑤ will have only one shared leaf node for all configurations $\mathtt{x} \in [-20, 20]$. Thus, the tool reports that the assertion may fail for all configurations. However, the lifted analysis using the Polyhedra domain will be more successful. The BDD found in location ⑤ will have only two leaf nodes (although there are 8 configurations), such that the shared invariant for configurations that satisfy $\mathtt{A} \vee \mathtt{B} \vee \mathtt{C}$ is: $10 \leq \mathtt{x}\text{+}30\mathtt{y} \leq 20 \land 0 \leq \mathtt{y} \leq 1$.

Hence, the BDD analysis will effectively conclude that the assertion is valid for configurations $A \vee B \vee C$, whereas it may fail for configuration $\neg A \wedge \neg B \wedge \neg C$.

Consider the following program family P_2:

```
①        #ifdef (A ∨ B) int j := input([0,9]);
②        #else int j := input([10,19]); #endif
l_input   int i := 0;
④        while (i < 100) {
⑤            i := i+1;
⑥            j := j+1; }
⑦        assert (j ≤ 105);
```

We want to prove the assertion ($j \leq 105$), since, for example, later on in the program there are references to an array using the index `105-j` (e.g. `a[105-j] := 0`). At location ⑦, a forward lifted analysis will find the invariant $100 \leq j \leq 109$ for configurations $A \vee B$ and the invariant $110 \leq j \leq 119$ for configurations $\neg A \wedge \neg B$. Hence, the assertion can be satisfied for configurations $A \vee B$ when $100 \leq j \leq 105$, and always fails for $\neg A \wedge \neg B$. We can perform a backward analyses to find the necessary preconditions on the input state at $l_{\texttt{input}}$ in order the assertion to hold. The tool will report that the necessary precondition for the assertion to hold at $l_{\texttt{input}}$ is: $0 \leq j \leq 5$ for configurations $A \vee B$, and it never holds for $\neg A \wedge \neg B$.

Threats to validity. The current tool has only limited support for arrays, pointers, recursion, `struct` and `union` types. Unfortunately, the most real-world program families contain the above features. Thus, we use as benchmarks either programs from single-program verification community (SV-COMP) in which we have manually added variability or more realistic program families (BusyBox) that have been simplified. However, the above features (arrays, pointers, etc) are largely orthogonal to the issue we are addressing here. In particular, these features complicate the semantics of single-programs and implementation of the domains for leaf nodes, but have no impact on the semantics of variability-specific constructs and the BDD lifted domain we introduce in this work. Therefore, if a real-world tool based on abstract interpretation, such as ASTREE [5], becomes freely available in the future, we can easily transfer our implementation to it. We perform lifted analysis of relatively small benchmarks (below 100 LOC). However, the focus of the BDD lifted domain is to combat the configuration space blow-up

of program families, not their LOC size. So, we expect to obtain similar or even better results for larger benchmarks.

Another threat to validity is the synthetic variability that has been manually added to the SV-COMP benchmarks. We have generated benchmarks with moderate feature use (4 or 5 features), due to the fact that they are very common in practice and any speed ups of lifted analysis should be visible for them. For benchmarks with intensive feature use (10 or more features), the tuple-based lifted analysis is very likely to become infeasible, as demonstrated by the simple $\texttt{foo}_n$ method, so the BDD lifted analysis will be significantly faster in that case. We have also inserted presence conditions with different complexities, from atomic to more complex feature expressions, and `#ifdef`-s are placed in different locations of the code.

8. Related Work

Disjunctive abstract domains have attracted considerable attention in abstract interpretation community recently [21, 22, 23, 24]. They enrich an abstract domain with disjunctions of several abstract elements. Decision trees have been applied for the disjunctive refinement of interval (boxes) domain [21], such that each element of the new domain is a propositional formula over interval linear constraints. Segmented decision tree abstract domains have also been defined in the literature [22, 23] to enable path dependent static analysis. Their elements contain decision nodes that are determined either by values of program variables [22] or by the branch (`if`) conditions [23], whereas the leaf nodes are numerical properties. Urban and Mine [24] use decision tree-based abstract domains to prove program termination. Decision nodes are labelled with linear constraints that split the memory space and leaf nodes contain affine ranking functions for proving program termination. Logico-numerical abstract domain implemented using BDDAPRON and specifically designed acceleration methods are used in [25] to verify synchronous data-flow programs with Boolean and numerical variables, such as LUSTRE programs. The BDDAPRON library has been developed by Jeannet [10] to implement a relational inter-procedural analysis of concurrent programs, whereas the APRON library has been developed by Jeannet and Mine [11] for the application of numerical domains in static analysis.

A combination *forward-backward* analyses based on abstract interpretation have also been used in practice for a long time [26, 27, 28, 14, 15]. Rival [26] uses forward-backward analysis to inspect more closely reported

alarms by `ASTREE`, which are then classified as true errors (bugs) or false alarms. Urban and Mine [28] use forward-backward analysis for the automatic inference of preconditions for program termination. Forward-backward analysis schemes have been used in [27] for the inference of safety properties of declarative synchronous programs. A combination of forward-backward analysis and model counting techniques [14, 15] are used to calculate the probability that a given assertion is valid or fails. In this work, we employ forward-backward analysis for analyzing program families.

Recently, two mainstream styles of static analysis have been a topic of considerable research in the SPL community: *a dataflow analysis from the monotone framework* developed by Kildall [29] that is algorithmically defined on syntactic CFGs, and *an abstract interpretation-based static analysis* developed by Cousot and Cousot [3] that is more general and semantically defined. Brabrand et. al. [7] lift a dataflow analysis from the *monotone framework*, resulting in a tuple-based lifted dataflow analysis that works on the level of families. The obtained lifted dataflow analyses are much faster than ones based on the "brute force" strategy, which generates and analyzes all variants one by one. Another efficient implementation of the lifted dataflow analysis from the monotone framework is by using variational data structures [30] (e.g., variational CFGs, variational data-flow facts) for achieving efficient dataflow computation. Several dataflow and control flow analysis (e.g., case termination, dangling switch, dead store, double free, freeing of static memory) are implemented and evaluated on some real-world systems. SPL^{LIFT} [31] is an efficient implementation of the lifted dataflow analysis formulated within the IFDS framework, which represents a subset of dataflow analyses with certain properties such as distributivity of transfer functions. Many dataflow analyses, including numerical analyses considered here, are not distributive and cannot be encoded in IFDS.

Midtgaard et. al. [8] have proposed a formal methodology for systematic derivation of tuple-based lifted static analyses from existing single-program analyses phrased in the abstract interpretation framework. There are two ways to speed up analyses: improving representation and increasing abstraction. In this paper, we investigate the former. The latter has also received attention in the field of lifted analysis [32, 33, 34]. Variability abstractions introduced in [32, 33, 34] aim to tame the combinatorial explosion of the number of configurations and reduce it to something more tractable by manipulating the configuration space. Such variability abstractions are used for deriving abstract lifted analyses, which enable deliberate trading of precision

for speed. However, the above tuple-based lifted analyses [7, 8, 32, 33, 34] are only applied to CIDE-based Java program families [35] using toy client analyses, such as reaching definitions and uninitialized variables. On the other hand, here we consider `#ifdef`-enriched C programs which represent the majority of industrial embedded code, as well as the most known numeric client analyses which enable verification of the most common invariance properties.

Various approaches have been proposed for lifting other existing analysis and verification techniques to work on the level of families (see [36] for a survey). Many family-based (lifted) approaches analyze entire families at once through sharing, by splitting where necessary and joining at fine granularity. Our lifted static analysis based on BDDs is an example of such analysis with sharing. There are other successful lifted techniques that use sharing through BDDs. SPLVerify [37] performs software model checking of program families based on variability encoding [38] which transforms compile-time to run-time variability [38], VarexJ [39] performs dynamic analysis of program families based on variability-aware execution, whereas SuperC [40] is a variability-aware parser, which can parse C language with preprocessor annotations thus producing ASTs with variability nodes. All of them use BDDs and standard BDD libraries (e.g. JavaBDD) to represent feature expressions. In this paper, we employ BDDs and widely-known numerical abstract domain libraries (APRON and BDDAPRON) for automatic inference of invariants of program families.

Lifted model checking has also been an active research field in recent years. One of the most known models of system families is by using the popular Feature Transition Systems (FTSs) [41]. Several specially designed lifted model checking algorithms for efficient verification of temporal properties of such models have been proposed [41, 42, 43, 44]. However, these approaches work on the level of models (i.e. high-level designs of SPLs), while our approach being based on abstract interpretation works directly on the level of source code. Specifically designed lifted model checking algorithms are used [45] for verifying symbolic game semantics models [46, 47] extracted from `#ifdef`-enriched imperative programs that contain undefined identifiers.

9. Conclusion

In this work we proposed lifted analysis domains based on tuples and binary decision diagrams, which are used for performing several lifted numeric analyses of program families. The BDD-based lifted domain provides a sym-

bolic and very compact representation of such lifted properties of program families, where the sharing of information is maximized. In effect, we obtain faster lifted analyses without losing any precision. We evaluate the proposed lifted domains on several C product lines. We experimentally demonstrate the effectiveness of BDD-based lifted domain.

In the future, we would like to extend the lifted abstract domain to also support non-Boolean (e.g., numerical) features [48]. We also plan to handle more complex heap-manipulating program families, so we would like to investigate the adaptability of such existing abstract domains [49] in our context. We also want to try other libraries that support numerical abstract domains, such as `ELINA` [50], and estimate their performance in this context.

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.

[2] C. Kästner, Virtual separation of concerns: Toward preprocessors 2.0, Ph.D. thesis, University of Magdeburg, Germany (May 2010).

[3] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the Fourth ACM Symp. on Principles of Programming Languages (POPL'77), ACM, 1977, pp. 238–252. doi:10.1145/512950.512973.
URL http://doi.acm.org/10.1145/512950.512973

[4] F. Nielson, H. R. Nielson, C. Hankin, Principles of Program Analysis, Springer-Verlag, Secaucus, USA, 1999.

[5] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, The astreé analyzer, in: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings, Vol. 3444 of LNCS, Springer, 2005, pp. 21–30. doi: 10.1007/978-3-540-31987-0_3.
URL https://doi.org/10.1007/978-3-540-31987-0_3

[6] A. Miné, Tutorial on static inference of numeric invariants by abstract interpretation, Foundations and Trends in Programming Languages 4 (3-

4) (2017) 120–372. doi:10.1561/2500000034.
URL https://doi.org/10.1561/2500000034

[7] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, P. Borba, Intraprocedural dataflow analysis for software product lines, T. Aspect-Oriented Software Development 10 (2013) 73–108.

[8] J. Midtgaard, A. S. Dimovski, C. Brabrand, A. Wasowski, Systematic derivation of correct variability-aware program analyses, Sci. Comput. Program. 105 (2015) 145–170. doi:10.1016/j.scico.2015.04.005.
URL http://dx.doi.org/10.1016/j.scico.2015.04.005

[9] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: POPL'79, 1979, pp. 269–282.

[10] B. Jeannet, Relational interprocedural verification of concurrent programs, in: Seventh IEEE Inter. Conf. on Software Engineering and Formal Methods, SEFM'09, IEEE Computer Society, 2009, pp. 83–92. doi:10.1109/SEFM.2009.29.
URL https://doi.org/10.1109/SEFM.2009.29

[11] B. Jeannet, A. Miné, Apron: A library of numerical abstract domains for static analysis, in: Computer Aided Verification, 21st Inter. Conf., CAV'09., Vol. 5643 of LNCS, Springer, 2009, pp. 661–667. doi:10.1007/978-3-642-02658-4_52.
URL https://doi.org/10.1007/978-3-642-02658-4_52

[12] A. Miné, The octagon abstract domain, Higher-Order and Symbolic Computation 19 (1) (2006) 31–100. doi:10.1007/s10990-006-8609-1.
URL https://doi.org/10.1007/s10990-006-8609-1

[13] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages (POPL'78), ACM Press, 1978, pp. 84–96. doi:10.1145/512760.512770.
URL https://doi.org/10.1145/512760.512770

[14] A. S. Dimovski, A. Legay, Computing program reliability using forward-backward precondition analysis and model counting, in: Fundamental Approaches to Software Engineering - 23rd International Conference,

FASE 2020, Proceedings, Vol. 12076 of LNCS, Springer, 2020, pp. 182–202. doi:10.1007/978-3-030-45234-6_9.
URL https://doi.org/10.1007/978-3-030-45234-6_9

[15] A. S. Dimovski, On calculating assertion probabilities for program families, Prilozi Contributions, Sec. Nat. Math. Biotech. Sci, MASA 41 (1) (2020) 13–23. doi:10.20903/csnmbs.masa.2020.41.1.153.

[16] A. S. Dimovski, Lifted static analysis using a binary decision diagram abstract domain, in: Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, ACM, 2019, pp. 102–114. doi:10.1145/3357765.3359518.
URL https://doi.org/10.1145/3357765.3359518

[17] A. Tarski, A lattice-theoretical fixpoint theorem and its applications, Pacific Journal of Mathematics 5 (2) (1955) 285–309.

[18] P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, J. Log. Program. 13 (2–3) (1992) 103–179.

[19] R. E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Trans. Computers 35 (8) (1986) 677–691. doi:10.1109/TC.1986.1676819.
URL https://doi.org/10.1109/TC.1986.1676819

[20] M. Huth, M. D. Ryan, Logic in computer science - modelling and reasoning about systems (2. ed.), Cambridge University Press, 2004.

[21] A. Gurfinkel, S. Chaki, Boxes: A symbolic abstract domain of boxes, in: Static Analysis - 17th International Symposium, SAS 2010. Proceedings, Vol. 6337 of LNCS, Springer, 2010, pp. 287–303. doi:10.1007/978-3-642-15769-1_18.
URL https://doi.org/10.1007/978-3-642-15769-1_18

[22] P. Cousot, R. Cousot, L. Mauborgne, A scalable segmented decision tree abstract domain, in: Time for Verification, Essays in Memory of Amir Pnueli, Vol. 6200 of LNCS, Springer, 2010, pp. 72–95. doi:10.1007/978-3-642-13754-9_5.
URL https://doi.org/10.1007/978-3-642-13754-9_5

[23] J. Chen, P. Cousot, A binary decision tree abstract domain functor, in: Static Analysis - 22nd International Symposium, SAS 2015, Proceedings, Vol. 9291 of LNCS, Springer, 2015, pp. 36–53. doi:10.1007/978-3-662-48288-9_3.
URL https://doi.org/10.1007/978-3-662-48288-9_3

[24] C. Urban, A. Miné, A decision tree abstract domain for proving conditional termination, in: Static Analysis - 21st International Symposium, SAS 2014. Proceedings, Vol. 8723 of LNCS, Springer, 2014, pp. 302–318. doi:10.1007/978-3-319-10936-7_19.
URL https://doi.org/10.1007/978-3-319-10936-7_19

[25] P. Schrammel, B. Jeannet, Logico-numerical abstract acceleration and application to the verification of data-flow programs, in: Static Analysis - 18th International Symposium, SAS 2011. Proceedings, Vol. 6887, Springer, 2011, pp. 233–248. doi:10.1007/978-3-642-23702-7_19.
URL https://doi.org/10.1007/978-3-642-23702-7_19

[26] X. Rival, Understanding the origin of alarms in astrée, in: Static Analysis, 12th International Symposium, SAS 2005, Proceedings, Vol. 3672 of LNCS, Springer, 2005, pp. 303–319. doi:10.1007/11547662_21.
URL https://doi.org/10.1007/11547662_21

[27] B. Jeannet, Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems, Formal Methods in System Design 23 (1) (2003) 5–37. doi:10.1023/A:1024480913162.
URL https://doi.org/10.1023/A:1024480913162

[28] C. Urban, Static analysis by abstract interpretation of functional temporal properties of programs. (analyse statique par interprétation abstraite de propriétés temporelles fonctionnelles des programmes), Ph.D. thesis, École Normale Supérieure, Paris, France (2015).
URL https://tel.archives-ouvertes.fr/tel-01176641

[29] G. A. Kildall, A unified approach to global program optimization, in: Conference Record of the ACM Symposium on Principles of Programming Languages, (POPL'73), 1973, pp. 194–206. doi:10.1145/512927.512945.
URL https://doi.org/10.1145/512927.512945

[30] A. von Rhein, J. Liebig, A. Janker, C. Kästner, S. Apel, Variability-aware static analysis at scale: An empirical study, ACM Trans. Softw. Eng. Methodol. 27 (4) (2018) 18:1–18:33. doi:10.1145/3280986.
URL https://doi.org/10.1145/3280986

[31] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, M. Mezini, Spl$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years, in: ACM SIGPLAN Conf. on PLDI '13, 2013, pp. 355–364.

[32] A. S. Dimovski, C. Brabrand, A. Wasowski, Variability abstractions: Trading precision for speed in family-based analyses, in: 29th European Conf. on Object-Oriented Programming, ECOOP 2015, Vol. 37 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, pp. 247–270. doi:10.4230/LIPIcs.ECOOP.2015.247.
URL http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.247

[33] A. S. Dimovski, C. Brabrand, A. Wasowski, Variability abstractions for lifted analysis, Sci. Comput. Program. 159 (2018) 1–27.

[34] A. S. Dimovski, C. Brabrand, A. Wasowski, Finding suitable variability abstractions for lifted analysis, Formal Asp. Comput. 31 (2) (2019) 231–259. doi:10.1007/s00165-019-00479-y.
URL https://doi.org/10.1007/s00165-019-00479-y

[35] C. Kästner, S. Apel, M. Kuhlemann, Granularity in software product lines, in: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), ACM, 2008, pp. 311–320.

[36] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A classification and survey of analysis strategies for software product lines, ACM Comput. Surv. 47 (1) (2014) 6.

[37] A. von Rhein, Analysis strategies for configurable systems, Ph.D. thesis, University of Passau, Germany (2016).

[38] A. F. Iosif-Lazar, A. S. Al-Sibahi, A. S. Dimovski, J. E. Savolainen, K. Sierszecki, A. Wasowski, Experiences from designing and validating a software modernization transformation (E), in: 30th IEEE/ACM Inter. Conf. on Automated Software Engineering, ASE'15, 2015, pp. 597–607. doi:10.1109/ASE.2015.84.
URL http://dx.doi.org/10.1109/ASE.2015.84

[39] J. Meinicke, C. Wong, C. Kästner, T. Thüm, G. Saake, On essential configuration complexity: measuring interactions in highly-configurable systems, in: Proceedings of the 31st IEEE/ACM Intern. Conf. on Automated Software Engineering, ASE'16, ACM, 2016, pp. 483–494. doi:10.1145/2970276.2970322.
URL http://doi.acm.org/10.1145/2970276.2970322

[40] P. Gazzillo, R. Grimm, Superc: parsing all of C by taming the preprocessor, in: J. Vitek, H. Lin, F. Tip (Eds.), ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012, ACM, 2012, pp. 323–334. doi:10.1145/2254064.2254103.
URL http://doi.acm.org/10.1145/2254064.2254103

[41] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, J. Raskin, Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking, IEEE Trans. Software Eng. 39 (8) (2013) 1069–1089. doi:10.1109/TSE.2012.86.
URL http://doi.ieeecomputersociety.org/10.1109/TSE.2012.86

[42] A. S. Dimovski, A. S. Al-Sibahi, C. Brabrand, A. Wasowski, Efficient family-based model checking via variability abstractions, STTT 19 (5) (2017) 585–603. doi:10.1007/s10009-016-0425-2.
URL https://doi.org/10.1007/s10009-016-0425-2

[43] A. S. Dimovski, A. Legay, A. Wasowski, Variability abstraction and refinement for game-based lifted model checking of full CTL, in: Fundamental Approaches to Software Engineering - 22nd International Conference, FASE 2019, Proceedings, Vol. 11424 of LNCS, Springer, 2019, pp. 192–209. doi:10.1007/978-3-030-16722-6_11.
URL https://doi.org/10.1007/978-3-030-16722-6_11

[44] A. S. Dimovski, Ctl* family-based model checking using variability abstractions and modal transition systems, Int. J. Softw. Tools Technol. Transf. 22 (1) (2020) 35–55. doi:10.1007/s10009-019-00528-0.
URL https://doi.org/10.1007/s10009-019-00528-0

[45] A. S. Dimovski, Verifying annotated program families using symbolic game semantics, Theor. Comput. Sci. 706 (2018) 35–53. doi:10.1016/

j.tcs.2017.09.029.
URL https://doi.org/10.1016/j.tcs.2017.09.029

[46] A. S. Dimovski, Program verification using symbolic game semantics, Theor. Comput. Sci. 560 (2014) 364–379. doi:10.1016/j.tcs.2014.01.016.
URL http://dx.doi.org/10.1016/j.tcs.2014.01.016

[47] A. Dimovski, R. Lazic, CSP representation of game semantics for second-order idealized algol, in: 6th International Conference on Formal Engineering Methods, ICFEM 2004, Proceedings, Vol. 3308 of LNCS, Springer, 2004, pp. 146–161. doi:10.1007/978-3-540-30482-1_18.
URL https://doi.org/10.1007/978-3-540-30482-1_18

[48] A. S. Dimovski, S. Apel, A. Legay, A decision tree lifted domain for analyzing program families with numerical features, in: Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Proceedings, Vol. 12649 of LNCS, Springer, 2021, pp. 67–86. doi:10.1007/978-3-030-71500-7_4.
URL https://doi.org/10.1007/978-3-030-71500-7_4

[49] B. E. Chang, X. Rival, Modular construction of shape-numeric analyzers, in: Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday, 2013., Vol. 129 of EPTCS, 2013, pp. 161–185. doi:10.4204/EPTCS.129.11.
URL https://doi.org/10.4204/EPTCS.129.11

[50] G. Singh, M. Püschel, M. T. Vechev, Making numerical program analysis fast, in: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015, ACM, 2015, pp. 303–313. doi:10.1145/2737924.2738000.
URL https://doi.org/10.1145/2737924.2738000