# Lifted Static Analysis using a Binary Decision Diagram Abstract Domain

Aleksandar S. Dimovski
aleksandar.dimovski@unt.edu.mk
Faculty of Informatics, Mother Teresa University
Skopje, Republic of North Macedonia

## Abstract

Many software systems are today variational. They can produce a potentially large variety of related programs (variants) by selecting suitable configuration options (features) at compile time. Specialized *variability-aware* (*lifted*, *family-based*) static analyses allow analyzing all variants of the family, 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 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 families with a large number of variants.

In this work, we propose a more efficient lifted static analysis 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 features, and the leaf nodes belong to an existing single-program analysis domain. We have developed a lifted static analysis which 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. Through experiments applied to C program families, we show that our new BDD-based approach outperforms the old tuple-based approach for lifted analysis.

***CCS Concepts*** • **Software and its engineering → Software organization and properties**; *Software functional properties*; Formal methods; Automated static analysis.

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

**ACM Reference Format:**
Aleksandar S. Dimovski. 2019. 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 '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357765.3359518

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*GPCE '19, October 21–22, 2019, Athens, Greece*
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6980-0/19/10…$15.00
https://doi.org/10.1145/3357765.3359518

## 1 Introduction

Variability is becoming increasingly common in today's software systems. Many software projects adopt the *Software Product Line* (SPL) methodology [10] 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 configured 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. Many industrial SPLs are typically implemented using annotative approaches such as conditional compilation (e.g., `#ifdef`-s from the C-preprocessor [30]).

In many of the application domains, a rigourous verification and analysis of program families is of paramount importance. Among the methods included in current practices, static program analysis by abstract interpretation [11, 35] is a powerful technique for automatic verification of software systems. Unfortunately, static analysis of program families is hard because the number of possible variants can be very large (often huge). Hence, the simplest brute-force approach that uses a preprocessor to generate all variants of a family and then applies an existing single-program analysis to each individual variant, one-by-one, is very inefficient. This approach has to compile (preprocess and build control

flow graph) and execute the fixed point iterative algorithm once for each possible variant. To overcome this problem, *variability-aware* (lifted, family-based) static analyses have been proposed [4, 33], which are able to handle programs involving Boolean features and numerical variables. 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 code base, which encodes all variants of a program family, and produce precise disjunctive analysis results corresponding to all variants by using a lifted analysis domain, which represents $n$-fold product of a single-program analysis domain used for expressing program properties (where $n$ is the number of valid configurations). 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. [1] This is known as 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 analysis due to the variability-specific constructs of the language (e.g. Boolean features, `#ifdef`-s, etc). One possible solution is to enable weak forms of disjunctions in order to improve expressivity while minimizing the cost of the analysis [7, 14, 38]. 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 diagrams (BDDs) represent an instance of the reduced cardinal power of domains [12, Sect. 10.2], which map the values of Boolean features (represented in decision nodes) to an analysis property (represented in leaf nodes) for the variant specified by the values of features along the path leading to the leave. These decision diagram domains are particularly well suited for representing disjunctive properties (which is the key aspect of a lifted analysis domain). The lifted analysis 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.

[1] For example, the Linux kernel currently provides more than 10, 000 configurable features, which leads up to $2^{10,000}$ distinct variants.

On the practical side, we have developed a prototype lifted analyzer which uses the BDDAPRON library [28] to implement the binary decision diagram domain. BDDAPRON uses any property domain from the APRON library [29] 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 a forward reachability analysis of C program families for the automatic inference of *invariants* in all program points. The tool computes a set of possible invariants, thanks notably to the design of numerical property domains, which allow to represent the information about the possible values of individual variables (by using interval domain, which is non-relational), as well as relations between variables: constraints between two variables (by using octagon domain, which is weakly relational), and constraints between all variables (by using polyhedra domain, which is fully relational). 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 C program families, which represent majority of industrial embedded code. In particular, we are able to check invariance properties, such as assertions, buffer overflows, division by zero, etc [13]. This work makes several contributions:

- We propose a new lifted analysis domain based on BDDs, which is well suited for handling the disjunctive properties that come from the variability.
- We develop a lifted static analyzer in which the lifted analysis domains are instantiated to numerical property domains from the APRON library. Hence, we can use it for program verification and proving program invariants of C program families.
- Finally, we evaluate our approach for lifted static analysis of C program families by comparing implementations which use the tuple-based lifted domain and BDD-based lifted domain.

## 2 Motivating Example

To better illustrate the issues we are addressing in this work, we now present a motivating example based on the following program family $P$:

```
①  int x := 10, y := 0;
②  while (x != 0) {
③      x := x-1;
④      #if (A) y := y+1; #endif
⑤      #if (B) y := y+1; #endif
⑥  } ⑦
```

The set of (Boolean) features in the above 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 family $P$ contains two `#if` statements, which increase the variable `y` by 1,

depending on which features from $\mathbb{F}$ are enabled. For each configuration from $\mathbb{K}$ a different variant (single program) can be generated by appropriately resolving `#if`-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 program points ④ and ⑤ 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 program points ④ and ⑤ will not be included in it. There are $|\mathbb{K}| = 4$ generated variants in total, which are shown in Fig. 3.

Assume that we want to perform *lifted analyses* on the family $P$ using the numerical property domains: intervals, octagons, and polyhedra. The standard lifted analysis domain from [4, 33] is defined as cartesian product of $\mathbb{K}$ copies of the basic domain, which corresponds to the client analysis we want to perform. Thus, the lifted domain will have elements which are tuples containing one component for every valid configuration of $\mathbb{K}$. The lifted analyses results in the final program point ⑦ of $P$ obtained using the lifted *interval, octagon, and polyhedra* analyses are the 4-sized tuples shown in Fig. 1a, Fig. 1b, and Fig. 1c, respectively. Note that the first component of a tuple in Fig. 1 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. 1, we can see that the interval analysis discovers imprecise (approximative) results about the variable y for configurations $A \wedge B$, $A \wedge \neg B$ and $\neg A \wedge B$, that is $y \geq 0$, since it is not able to reason about the relations between the variables x and y. The octagon analysis gives more precise (less approximative) results about y. That is, $y = 10$ for configurations $A \wedge \neg B$ and $\neg A \wedge B$ as well as $y = 0$ for $\neg A \wedge \neg B$ are the most precise (exact) results. But, we obtain an imprecise (approximative) result $y \geq 10$ for the configuration $A \wedge B$, since the relations which can be tracked using the octagon analysis are limited. Finally, the polyhedra analysis 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 (linear) relations between variables.

On the other hand, if we perform lifted analyses based on the binary decision diagram domain proposed here, then the analyses results in the final program point ⑦ of $P$ obtained using interval, octagon, and polyhedra domains are shown in Fig. 2a, Fig. 2b, and Fig. 2c, respectively. Note that the inner nodes of a binary decision diagram (BDD) in Fig. 2 are labeled with features from $\mathbb{F}$, the leaves are labeled with the elements from the property domain we use, and the edges 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 binary decision diagrams 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. 2a presents interval properties of two variables x and y, which are partitioned with respect to the 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 (properties for x and y), while the tuple-based representation in Fig. 1a uses four. 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 $\mathbb{F} = \{A_1, \dots, A_n\}$ 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* (valid product) 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 formula: $k(A_1) \wedge \dots \wedge k(A_n)$, where $k(A_i) = A_i$ if $A_i \in k$, and $k(A_i) = \neg A_i$ if $A_i \notin k$ for $1 \leq i \leq n$. We will use both representations interchangeably.

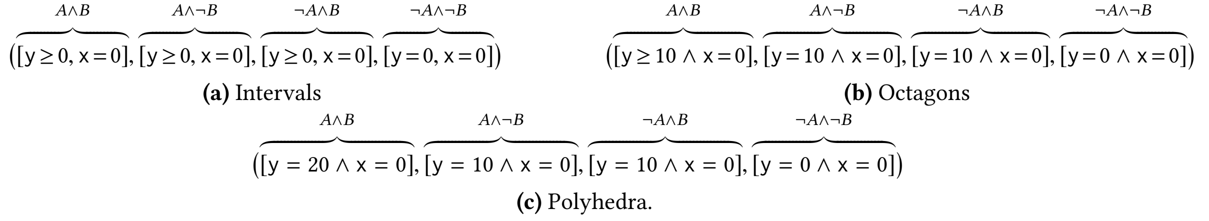
We define *feature expressions*, denoted $FeatExp(\mathbb{F})$, as the set of well-formed propositional logic formulae over $\mathbb{F}$ generated by the grammar:

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

We will use $\theta \in FeatExp(\mathbb{F})$ to define presence conditions in program families. We write $[\![\theta]\!]$ to denote the set of variants from $\mathbb{K}$ that satisfy $\theta$, i.e. $k \in [\![\theta]\!]$ iff $k \models \theta$, where $\models$ is the standard satisfaction relation of propositional logic. For example, given $\mathbb{F} = \{A, B\}$ with all four possible variants being valid $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$ (or, equivalently using sets $\mathbb{K} = \{\{A, B\}, \{A\}, \{B\}, \emptyset\}$), for the feature expression $A \vee B$ we have: $[\![A \vee B]\!] = \{A \wedge B, A \wedge \neg B, \neg A \wedge B\}$.

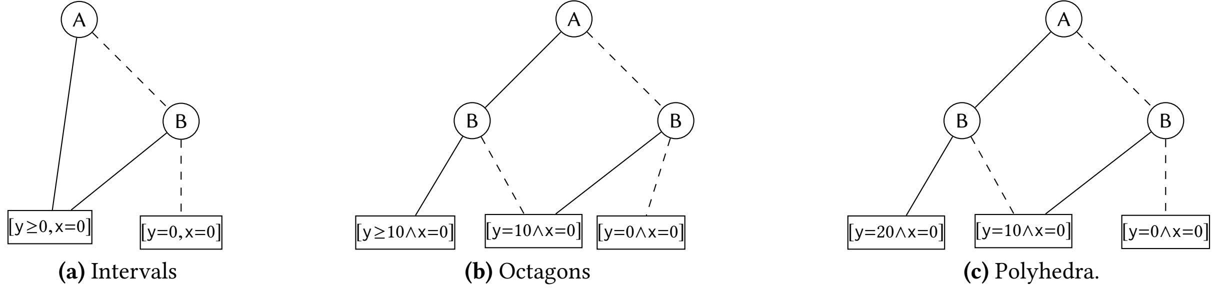
We consider the language $\overline{\text{IMP}}$ for writing program families, which will be used to exemplify our work. Still, the introduced methodology is not limited to $\overline{\text{IMP}}$ or its features. In fact, we evaluate our approach on program families written in C. $\overline{\text{IMP}}$ is an extension of the imperative language IMP [35] often used in semantic studies. $\overline{\text{IMP}}$ adds a compile-time conditional statement for encoding multiple variants of a program. The new statement "`#if` $(\theta)$ $s$" contains a feature expression $\theta \in FeatExp(\mathbb{F})$ as a presence condition, such that only if $\theta$ is satisfied by a configuration $k \in \mathbb{K}$ then the statement $s$ will be included in the variant corresponding to $k$. The syntax of the language is given by:

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



**(a)** Intervals **(b)** Octagons **(c)** Polyhedra.

**Figure 1.** Tuple-based analyses results at the program point ⑦ of $P$.



**(a)** Intervals **(b)** Octagons **(c)** Polyhedra.

**Figure 2.** BDD-based analyses results at the program point ⑦ of $P$ (solid edges = true, dashed edges = false).

where $n$ ranges over integers, $[n, n']$ ranges over integer intervals, x ranges over variable names *Var*, and $\oplus$ over binary arithmetic operators. Integer intervals $[n, n']$ have constant and possibly infinite bounds, and denote a random choice of an integer in the interval. This provides a notion of nondeterminism useful to model user input or to approximate expressions. The set of all generated statements $s$ is denoted by *Stm*, while the set of all expressions $e$ is denoted by *Exp*.

The $\overline{\text{IMP}}$ programs are evaluated in two stages. First, a *preprocessor* takes as input an $\overline{\text{IMP}}$ program and a configuration $k \in \mathbb{K}$, and outputs a variant, i.e. a single IMP program without `#if`-s, corresponding to $k$. Second, the obtained variant is evaluated using the standard IMP semantics [35]. The first stage is specified by the projection function $\mathsf{P}_k$, which copies all basic statements of $\overline{\text{IMP}}$ that are also in IMP and recursively pre-processes all sub-statements of compound statements. Hence, $\mathsf{P}_k(\mathtt{skip}) = \mathtt{skip}$ and $\mathsf{P}_k(s;s') = \mathsf{P}_k(s);\mathsf{P}_k(s')$. The interesting case is "`#if` $(\theta)$ $s$" statement, where the statement $s$ is included in the resulting variant iff $k \models \theta$, otherwise the statement $s$ is removed. That is,

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

For example, the variants $\mathsf{P}_{A\wedge B}(P)$, $\mathsf{P}_{A\wedge\neg B}(P)$, $\mathsf{P}_{\neg A\wedge B}(P)$, and $\mathsf{P}_{\neg A\wedge\neg B}(P)$ shown in Fig. 3a, Fig. 3b, Fig. 3c, and Fig. 3d, respectively, are derived from the program family $P$ defined in Section 2.

## 4 Numerical Property Domains

There exist various numerical property domains, which can be used for automatic discovery of numerical properties of program variables. They differ in expressive power and computational complexity. In the following, we briefly recall the well-known numerical property domains of intervals [11], octagons [34], and polyhedra [15]. They are the foundation upon which we implement in practice new lifted analyses domains introduced in Sections 5 and 6.

***Intervals.*** The *Interval domain* [11] (also called *Box domain*), denoted as $\langle I, \sqsubseteq_I \rangle$, is a non-relational numerical property domain, which abstracts each variable independently. It identifies the range of possible values for every variable as an interval. The property elements are: $\{\bot_I\} \cup \{[l, h] \mid l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leq h\}$, where the least element (bottom) $\bot_I$ denotes the empty interval and the greatest element (top) is $\top_I = [-\infty, +\infty]$.

The abstract operations of the *Interval domain* are defined in [11], they are: concretization function $\gamma_I$, partial ordering $\sqsubseteq_I$, least upper bound (join) $\sqcup_I$, greatest lower bound (meet) $\sqcap_I$, widening $\nabla_I$, narrowing $\triangle_I$, transfer functions for tests $\text{FILTER}_I$ and assignments $\text{ASSIGN}_I$. 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 $\gamma_I$, which assigns a concrete meaning to each element from $I$, is defined as:

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

The partial ordering $\sqsubseteq_I$ is defined as:

$$[l_1, h_1] \sqsubseteq_I [l_2, h_2] \ \equiv_{def} \ l_2 \leq l_1 \wedge h_1 \leq h_2$$

```
int x := 10, y := 0;
while(x !=0) {
    x := x-1;
    y := y+1;
    y := y+1;
}
```

(a) $\mathsf{P}_{A\wedge B}(P)$

```
int x := 10, y := 0;
while(x !=0) {
    x := x-1;
    y := y+1;
}
```

(b) $\mathsf{P}_{A\wedge\neg B}(P)$

```
int x := 10, y := 0;
while(x !=0) {
    x := x-1;
    y := y+1;
}
```

(c) $\mathsf{P}_{\neg A\wedge B}(P)$

```
int x := 10, y := 0;
while(x !=0) {
    x := x-1;
}
```

(d) $\mathsf{P}_{\neg A\wedge\neg B}(P)$

**Figure 3.** Different variants of the program family $P$ from Section 2.

The least upper bound (join), $\sqcup_I$, and the greatest lower bound (meet), $\sqcap_I$, are:

$$[l_1,h_1] \sqcup_I [l_2,h_2] = [min\{l_1,l_2\}, max\{h_1,h_2\}],$$
$$[l_1,h_1] \sqcap_I [l_2,h_2] = [max\{l_1,l_2\}, min\{h_1,h_2\}]$$

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 $-\infty$ [11]:

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

In order to improve the precision of loop analysis, we can apply the narrowing after stabilization with widening is achieved. This is an example narrowing for intervals:

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

Let $a \in Var \rightarrow I$ be an abstract state which maps each variable x to an interval. The transfer function $\text{FILTER}_I$ abstracts tests (expressions) in `while`-s and `if`-s by restricting the input abstract store so that it satisfies the given test. For example, this is one simple case:

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

where $a(\mathsf{x}) = [l,h]$. The transfer function $\text{ASSIGN}_I$ which abstracts assignments is:

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

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

***Octagons.*** The *Octagon domain* [34], 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 $\pm\mathsf{x}_j \pm \mathsf{x}_i \leq c$ between program variables $\mathsf{x}_i$ and $\mathsf{x}_j$.

The abstract operations of the *Octagon domain* are defined in [34]. 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 quite 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 $x_i \in Var$, 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 $\gamma_O$ is defined as:

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

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 elementwise, and $\top_{DBM}$ has all its elements set to $+\infty$. Thus, we can use its operators to define the octagon analysis.

As for the interval domain, the widening $\nabla_O$ puts unstable bounds to infinity, while the narrowing $\triangle_O$ refines an upper bound only if it is infinity.

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

$$\forall i,j.\left[\mathbf{m}\triangle_O\mathbf{n}\right]_{ij} = \begin{cases} n_{ij}, & \text{if } m_{ij} = +\infty \\ m_{ij}, & \text{otherwise} \end{cases}$$

The transfer functions for tests $\text{FILTER}_O$ and assignments $\text{ASSIGN}_O$ are similar [34]. We show here only the simplest case of handling the non-deterministic assignment $\mathsf{x}_j$:=? (or $\mathsf{x}_j$:=$[-\infty,+\infty]$).

$$\forall k,l.\left[\text{ASSIGN}_I(\mathbf{m} : DBM,\ \mathsf{x}_j:=? : Stm)\right]_{kl} = \begin{cases} +\infty, & \text{if } k \in \{2j-1,2j\} \vee l \in \{2j-1,2j\} \\ m_{kl}, & \text{otherwise} \end{cases}$$

***Polyhedra.*** The *Polyhedra domain* [15], denoted as $\langle P, \sqsubseteq_P \rangle$, is a fully relational numerical property domain, which allows manipulating conjunctions of linear inequalities of the form $\alpha_1\mathsf{x}_1 + \ldots + \alpha_n\mathsf{x}_n \geq \beta$, where $\mathsf{x}_1,\ldots,\mathsf{x}_n$ are program variables and $\alpha_i, \beta \in \mathbb{R}$ (reals). The abstract operations of the *Polyhedra domain* are defined in [15]. 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 $\mathbf{A} \in \mathbb{R}^{m\times n}$ and a vector $\vec{\mathbf{b}} \in \mathbb{R}^m$, 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. One representation can be converted to the other one using the Chernikova's algorithm [8]. 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:

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

The meet $\sqcap_P$ is defined as:

$$\langle\!\langle \mathbf{A}_1, \vec{\mathbf{b}}_1 \rangle\!\rangle \sqcap_P \langle\!\langle \mathbf{A}_2, \vec{\mathbf{b}}_2 \rangle\!\rangle = \langle\!\langle \binom{\mathbf{A}_1}{\mathbf{A}_2}, \binom{\vec{\mathbf{b}}_1}{\vec{\mathbf{b}}_2} \rangle\!\rangle$$

We also need widening since the polyhedra domain has infinite strictly increasing chains.

$$\langle\!\langle \mathbf{A}_1, \vec{\mathbf{b}}_1 \rangle\!\rangle \nabla_P \langle\!\langle \mathbf{A}_2, \vec{\mathbf{b}}_2 \rangle\!\rangle = \{c \in \langle\!\langle \mathbf{A}_1, \vec{\mathbf{b}}_1 \rangle\!\rangle \mid \langle\!\langle \mathbf{A}_2, \vec{\mathbf{b}}_2 \rangle\!\rangle \sqsubseteq_P \{c\}\}$$

where $c$ represents one constraint from $\langle\!\langle \mathbf{A}_1, \vec{\mathbf{b}}_1 \rangle\!\rangle$. The transfer function $\text{FILTER}_P$ abstracts affine inequality tests (expressions) by adding them to the input polyhedra.

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

## 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 $\overline{\text{IMP}}$ program families, without preprocessing them by taking into account variability-specific aspects of program families.

In this section, we introduce lifted analyses based on the lifted domain that is $|\mathbb{K}|$-fold product of an existing (single-program) analysis domain $\mathbb{A}$. From now on, we assume that the single-program analysis domain $\mathbb{A}$ is equipped with sound operators for concretization $\gamma_{\mathbb{A}}$, ordering $\sqsubseteq_{\mathbb{A}}$, join $\sqcup_{\mathbb{A}}$, meet $\sqcap_{\mathbb{A}}$, bottom $\bot_{\mathbb{A}}$, top $\top_{\mathbb{A}}$, widening $\nabla_{\mathbb{A}}$, and narrowing $\triangle_{\mathbb{A}}$, as well as sound transfer functions for tests $\text{FILTER}_{\mathbb{A}}$ and assignments $\text{ASSIGN}_{\mathbb{A}}$. For example, in the implementation we will use for $\mathbb{A}$ one of the numerical property domains introduced in Section 4.

***Lifted Domain.*** The *lifted analysis 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 5.1.** Consider the tuple in Fig. 1a, in which components are analysis properties from the Interval domain and $\mathbb{K} = \{A \land B, A \land \neg B, \neg A \land B, \neg A \land \neg B\}$. Note that to simplify the presentation, we write $\mathsf{x} \geq n$ short for $\mathsf{x} \mapsto [n, +\infty]$, $\mathsf{x} \leq n$ short for $\mathsf{x} \mapsto [-\infty, n]$, $n \leq \mathsf{x} \leq n'$ short for $\mathsf{x} \mapsto [n, n']$, and $\mathsf{x} = n$ short for $\mathsf{x} \mapsto [n, n]$. In first order logic, the tuple in Fig. 1a can be written as the following disjunctive property:

$$\big(A \land B \land [\mathsf{y} \geq 0, \mathsf{x} = 0]\big) \lor \big(A \land \neg B \land [\mathsf{y} \geq 0, \mathsf{x} = 0]\big) \lor \big(\neg A \land B \land [\mathsf{y} \geq 0, \mathsf{x} = 0]\big) \lor \big(\neg A \land \neg B \land [\mathsf{y} = 0, \mathsf{x} = 0]\big) \quad (1)$$

***Abstract Operations.*** Given a tuple (lifted domain element) $\overline{a} \in \mathbb{A}^{\mathbb{K}}$, the projection $\pi_k$ selects the $k^{\text{th}}$ component of $\overline{a}$.

*Concretization function:* Given a configuration $k \in \mathbb{K}$, the concretization function $\overline{\gamma}^k$ of a tuple $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ depends on the concretization function $\gamma_{\mathbb{A}}$ of the domain $\mathbb{A}$, and is defined as: $\overline{\gamma}^k(\overline{a}) = \gamma_{\mathbb{A}}(\pi_k(\overline{a}))$.

*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 defined as: $\overline{a_1} \dot{\sqcup} \overline{a_2} = \prod_{k \in \mathbb{K}} (\pi_k(\overline{a_1}) \sqcup_{\mathbb{A}} \pi_k(\overline{a_2}))$, and $\overline{a_1} \dot{\sqcap} \overline{a_2} = \prod_{k \in \mathbb{K}} (\pi_k(\overline{a_1}) \sqcap_{\mathbb{A}} \pi_k(\overline{a_2}))$.

*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 $\dot{\triangle}$ are defined as: $\overline{a_1} \dot{\nabla} \overline{a_2} = \prod_{k \in \mathbb{K}} (\pi_k(\overline{a_1}) \nabla_{\mathbb{A}} \pi_k(\overline{a_2}))$, $\overline{a_1} \dot{\triangle} \overline{a_2} = \prod_{k \in \mathbb{K}} (\pi_k(\overline{a_1}) \triangle_{\mathbb{A}} \pi_k(\overline{a_2}))$.

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

*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,

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

*Feature-based tests:* The transfer function $\overline{\text{F-FILTER}}$ for the feature expression tests "$\theta$" in `#if`-s is designed to check the satisfaction of $k \models \theta$ [2] 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, we have

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

*Assignments:* The transfer function $\overline{\text{ASSIGN}}$ that handle the assignment "$\mathsf{x}$:=$e$" in the input tuple $\overline{a} \in \mathbb{A}^{\mathbb{K}}$ is defined using $\text{ASSIGN}_{\mathbb{A}}$ which is independently applied on each component of $\overline{a}$. We have

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

[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 $\theta \in FeatExp(\mathbb{F})$.

① $([\mathsf{y}=\top_I, \mathsf{x}=\top_I], [\mathsf{y}=\top_I, \mathsf{x}=\top_I], [\mathsf{y}=\top_I, \mathsf{x}=\top_I], [\mathsf{y}=\top_I, \mathsf{x}=\top_I])$

② $([\mathsf{y}=0, \mathsf{x}=10], [\mathsf{y}=0, \mathsf{x}=10], [\mathsf{y}=0, \mathsf{x}=10], [\mathsf{y}=0, \mathsf{x}=10])$

③ $([\mathsf{y}\geq 0, \mathsf{x}\leq 10], [\mathsf{y}\geq 0, \mathsf{x}\leq 10], [\mathsf{y}\geq 0, \mathsf{x}\leq 10], [\mathsf{y}=0, \mathsf{x}\leq 10])$

④ $([\mathsf{y}\geq 0, \mathsf{x}\leq 9], [\mathsf{y}\geq 0, \mathsf{x}\leq 9], [\mathsf{y}\geq 0, \mathsf{x}\leq 9], [\mathsf{y}=0, \mathsf{x}\leq 9])$

⑤ $([\mathsf{y}\geq 1, \mathsf{x}\leq 9], [\mathsf{y}\geq 1, \mathsf{x}\leq 9], [\mathsf{y}\geq 0, \mathsf{x}\leq 9], [\mathsf{y}=0, \mathsf{x}\leq 9])$

⑥ $([\mathsf{y}\geq 2, \mathsf{x}\leq 9], [\mathsf{y}\geq 1, \mathsf{x}\leq 9], [\mathsf{y}\geq 1, \mathsf{x}\leq 9], [\mathsf{y}=0, \mathsf{x}\leq 9])$

⑦ $([\mathsf{y}\geq 0, \mathsf{x}=0], [\mathsf{y}\geq 0, \mathsf{x}=0], [\mathsf{y}\geq 0, \mathsf{x}=0], [\mathsf{y}=0, \mathsf{x}=0])$

**Figure 4.** Tuple-based (interval) analyses results at the program points from ① to ⑦ of $P$.

`#if` *statements:* Given the (lifted) transfer function $\overline{[\![s]\!]}$ for the statement $s$, the transfer function $\overline{\text{IFDEF}}$ for "`#if` $(\theta)$ $s$" is defined as:

$$\overline{\text{IFDEF}}(\overline{a} : \mathbb{A}^{\mathbb{K}}, \texttt{\#if } (\theta)\ s : Stm) = \overline{[\![s]\!]}(\overline{\text{F-FILTER}}(\overline{a}, \theta)) \dot{\sqcup} \overline{\text{F-FILTER}}(\overline{a}, \neg\theta)$$

***Lifted Analysis.*** In the first iteration of the analysis, we construct tuples based on the information we have for $\mathbb{K}$. For the first program point, we build a tuple where all components are set to $\top_{\mathbb{A}}$, whereas for the other program points all components are set to $\bot_{\mathbb{A}}$.

The operators of the lifted analysis domain $\mathbb{A}^{\mathbb{K}}$ and transfer functions are combined together to analyze program families. The non-$\bot$ analysis properties are then propagated forward from the first program point towards the final control point taking assignments and (expression- and feature-based) 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. 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 analysis based on tuples: we obtain correct analysis results for each valid variant.

**Example 5.2.** Consider the $\overline{\text{IMP}}$ program $P$ from Section 2. We want to perform *interval* lifted analysis of $P$ using the lifted domain $I^{\mathbb{K}}$. In order to enforce convergence of the analysis, we apply the widening operator at the loop head, that is, at the point before the `while` test. The final analysis results at program points from ① to ⑦ are shown in Fig. 4. They represent 4-sized tuples, which contain four interval properties (stores), one for each configuration. ◻

# 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 [5, 26] for representing formulae that combine Boolean variables and analysis properties. The elements

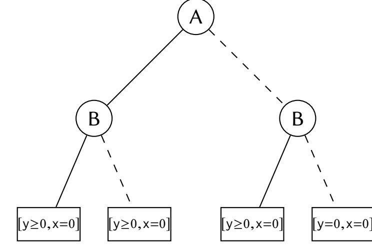



**Figure 5.** A BDT.

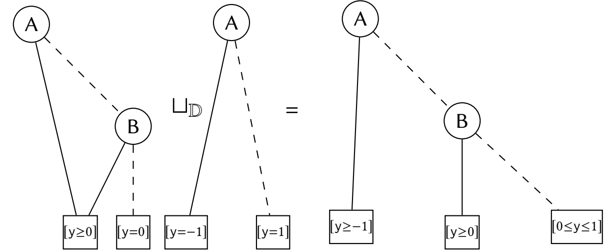



**Figure 6.** The join of two BDDs.

of the domain $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ are disjunctions of the leaf nodes that belong to an existing (single-program) analysis domain $\mathbb{A}$, which are separated by the values of Boolean features 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), which can be used as lifted analysis domains. 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, and the leaf abstract domain $\mathbb{A}$ is either a leaf node $\langle\!|p|\!\rangle$, with $p$ an element of $\mathbb{A}$ and $\mathbb{F} = \mathbb{K} = \emptyset$, or $[\![A : tl, tr]\!]$, where $A$ is the first element of $\mathbb{F}$, $tl$ is the left subtree of $t$ representing its true branch, and $tr$ is the right subtree of $t$ representing its false branch, such that $tl, tr \in \mathbb{T}(\mathbb{F}\backslash\{A\}, \mathbb{K}_{\backslash\{A\}}, \mathbb{A})$. $\mathbb{K}_{\backslash\{A\}}$ denotes the removal of $A$ from each configuration. The left and right subtrees are either both leaf nodes or both decision nodes labeled with the same feature.

**Example 6.1.** The binary decision tree in Fig. 5 has decision nodes labeled with features $A$ and $B$, and leaf nodes are Interval properties. In first order logic, the above tree expresses the same formula as the one in Eqn. (1), Example 5.1. ◻

However, the BDTs contain some redundancy. There are three optimizations we can apply to BDTs in order to reduce their representation [5, 26]:

(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* $d \in \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$. 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, then the resulting BDDs have a *canonical form*. This means that any disjunctive analysis property from the lifted domain $\mathbb{A}^{\mathbb{K}}$ can be represented in a unique way by a BDD from $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$.

**Example 6.2.** After applying reductions (**1**)-(**3**) to the binary decision tree in Fig. 5, the resulting reduced BDD is shown in Fig. 2a. □

***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 [5]. 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 BDD 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 BDD of "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 configuration $k \in \mathbb{K}$, the concretization function $\gamma_{\mathbb{D}}^k$ of a binary decision diagram $d \in \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ returns $\gamma_{\mathbb{A}}(a)$, where $a \in \mathbb{A}$ is the analysis property in the leaf node of $d$ reached along the top-down path representing the configuration $k$.

*Ordering:* Given two BDDs $d_1, d_2 \in \mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$, their approximation ordering $d_1 \sqsubseteq_{\mathbb{D}} d_2$ is defined as:

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

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. 6 shows the join of two BDDs from $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$.

*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:

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

***Transfer Functions.*** We now proceed by defining transfer functions for both expression-based and feature-based tests as well as for assignments and `#if`-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,

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

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

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

*Assignments:* The transfer function $\text{ASSIGN}_{\mathbb{D}}$ for the assignment "$\texttt{x:=}e$" is implemented by applying $\text{ASSIGN}_{\mathbb{A}}$ at each leaf node of the input BDD using $\texttt{apply}_1$.

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

`#if` *statements:* Given the (lifted) transfer function $\overline{[\![s]\!]}$ for the statement $s$, the transfer function $\text{IFDEF}_{\mathbb{D}}$ for "`#if` $(\theta)$ $s$" is defined as:

$$\begin{aligned} &\text{IFDEF}_{\mathbb{D}}(d\!:\!\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A}),\ \texttt{\#if}\ (\theta)\ s\!:\!Stm) = \\ &\qquad \overline{[\![s]\!]}(\text{F-FILTER}_{\mathbb{D}}(d,\ \theta)) \sqcup_{\mathbb{D}} \text{F-FILTER}_{\mathbb{D}}(d,\ \neg\theta) \end{aligned}$$

***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 analysis, we build BDDs with only one leaf node that can be reached along only valid paths. For the first program point the leaf node is $\top_{\mathbb{A}}$, whereas for the other program points the leaf node is $\bot_{\mathbb{A}}$. Thus, in the first iteration, the BDD for the first point is $\texttt{meet_condition}(\top_{\mathbb{D}}, \vee_{k \in \mathbb{K}} k)$, whereas for the other points 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 the BDD is $\top_{\mathbb{D}}$ for the first point and $\bot_{\mathbb{D}}$ for the others.

The operators of the abstract lifted domain $\mathbb{D}(\mathbb{F}, \mathbb{K}, \mathbb{A})$ and transfer functions are combined together to analyze program families. The non-$\bot_{\mathbb{D}}$ analysis properties are then propagated forward towards the final control point taking assignments and tests into account with join and widening around `while`-s. As a consequence of the soundness of all operators and transfer functions of the leaf domain $\mathbb{A}$, we can establish the soundness and correctness of the lifted analysis based on BDDs: we obtain correct analysis results for each variant corresponding to a valid configuration $k \in \mathbb{K}$.

**Example 6.3.** Consider the $\overline{\text{IMP}}$ program $P$ from Section 2. We want to perform interval lifted analysis of $P$ using the lifted domain $\mathbb{D}(\mathbb{F}, \mathbb{K}, I)$, where $\mathbb{F} = \{A, B\}$ and $\mathbb{K} = 2^{\{A,B\}}$. The final analysis results at program points from ① to ⑦ are shown in Fig. 7. Compared to the analysis results obtained using the lifted domain $I^{\mathbb{K}}$ in Fig. 4 (see Example 5.2), 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 points. For example, in program points from ① to ④, there is only one interval property (store). In points ⑤ and ⑦ there are two interval properties, while in ⑥ there are three interval properties. □

## 7 Evaluation

We now evaluate our approach for speeding up lifted analysis on several C case studies. The evaluation aims to show the following objectives:

- **O1:** The BDD-based lifted analyses outperform the corresponding tuple-based lifted analyses;
- **O2:** The BDD-based lifted analyses can even turn some previously infeasible tuple-based lifted analyses into feasible ones;
- **O3:** We can find practical application scenarios of using our lifted analyses to efficiently verify C program families.

***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 [29]. The operators and transfer functions for the binary decision diagram domains that combine Boolean formulae and APRON domains are provided by the BDDAPRON library [28]. The prototype tool is written in OCaml. It accepts programs written in a subset of C with `#ifdef` constructs, but without `struct` and `union` types. It provides only a limited support of arrays and pointers, and the only basic data types are integers. As output, the tool infers numeric invariants in all program points. We implement a forward reachability analysis that starts from the beginning of the program and an initial abstract property, then it goes forward to derive necessary conditions so that the executions reach particular program points. The analysis proceeds by structural induction on the program syntax, iterating `while`-s until a fixed point is reached. It computes the unique least solution which to every program point assigns an element from the lifted analysis domain.

***Experimental setup*** All experiments are executed on a 64-bit IntelCore$^{TM}$ i5 CPU, Linux Ubuntu VM, with 8 GB memory. The reported times represent the average runtime of five independent executions. We report the times (in *seconds*) needed for actual analyses to be performed. The implementation, benchmarks, and all results obtained from our experiments are available from: http://bit.ly/2SUck5n (also http://bit.ly/2Fu266X). 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, respectively.

***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) [3]. The folders we consider are: `loops`, `loop-invgen` (`invgen` for short), `loop-lit` (`lit` for short), `termination-crafted` (`crafted` for short), and finally we consider `termination-restricted` (`restrict` for short). We have selected some numeric programs with integers that our tool can handle. We have manually added variability in each of them, and then we have analyzed those programs using our prototype lifted analyzer. All added features are unconstrained, so the set of valid configurations is $\mathbb{K} = 2^{\mathbb{F}}$. Table 1 summarizes relevant characteristics for some selected benchmarks: the folder where it is located, the number of features, and the total number of lines of code (LOC).

***Performances*** Table 1 compares the performances of 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 program using the 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 analyses 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 results match expectations. All BDD-based versions achieve significant speed-ups compared to the tuple-based versions, which range from 2.6 to 13.5 times for programs with four features and from 5.3 to 11.8 times for programs with five features (addresses Objective (**O1**)). 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 version, then it comes $\overline{\mathcal{A}}_{\mathbb{D}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ is the slowest and the most precise version.

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

[3] https://sv-comp.sosy-lab.org/2019/

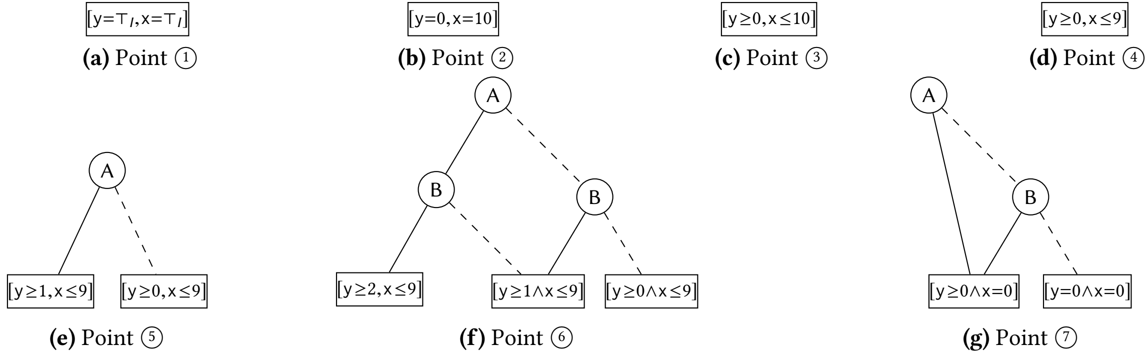



**Figure 7.** BDD-based analyses results at the program points from ① to ⑦ of $P$.

**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). All Times are in seconds.

| 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 | 0.0078 | 4.6× | 0.0163 | 7.5× | 0.0189 | 6.7× |
| half2.c | invgen | 4 | 30 | 0.0093 | 4.9× | 0.0240 | 7.8× | 0.0259 | 6.5× |
| heapsort.c | invgen | 4 | 60 | 0.0216 | 6.5× | 0.0887 | 13.5× | 0.0899 | 12.4× |
| seq.c | invgen | 4 | 40 | 0.0162 | 5.6× | 0.0721 | 9.1× | 0.0674 | 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 | 0.0046 | 5.6× | 0.0071 | 5.8× | 0.0103 | 5.9× |
| hhk2008.c | lit | 5 | 30 | 0.0100 | 7.9× | 0.0198 | 10.7× | 0.0269 | 8.9× |
| gsv2008.c | lit | 5 | 25 | 0.0079 | 7.5× | 0.0168 | 11.1× | 0.0189 | 10.7× |
| gcnr2008.c | lit | 4 | 30 | 0.0179 | 3.0× | 0.0345 | 6.2× | 0.0584 | 6.3× |
| bhmr2007.c | lit | 4 | 30 | 0.0085 | 5.6× | 0.0318 | 8.4× | 0.0231 | 7.4× |
| GCD4.c | restrict | 4 | 30 | 0.0056 | 6.3× | 0.0083 | 10.1× | 0.0170 | 8.7× |
| UpAndDown.c | restrict | 4 | 30 | 0.0106 | 4.8× | 0.0145 | 5.3× | 0.0414 | 4.7× |
| Log.c | restrict | 4 | 35 | 0.0082 | 5.7× | 0.0161 | 9.7× | 0.0201 | 8.8× |
| java_Sequence.c | restrict | 4 | 25 | 0.0085 | 3× | 0.0130 | 4.1× | 0.0177 | 4.6× |
| Toulouse*.c | crafted | 4 | 75 | 0.0143 | 4.1× | 0.0232 | 4.4× | 0.0392 | 5.2× |
| TelAviv*.c | crafted | 4 | 50 | 0.0070 | 3.1× | 0.0078 | 3.4× | 0.0142 | 4.7× |
| Mysore.c | crafted | 5 | 35 | 0.0092 | 5.3× | 0.0109 | 5.8× | 0.0235 | 7.4× |
| Copenhagen.c | crafted | 5 | 30 | 0.0041 | 7.6× | 0.0050 | 8.1× | 0.0087 | 11.1× |

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 $\mathsf{A}_1, \ldots, \mathsf{A}_n$ and $n$ sequentially composed `#if` statements of the form `#if` $(\mathsf{A}_i)$ `i := i+1`. For example, the method $\texttt{foo}_3()$ with three features $\mathsf{A}_1$, $\mathsf{A}_2$, and $\mathsf{A}_3$ is:

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

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

We have gradually added unconstrained variability into $\texttt{foo}_3$ by adding optional features and by sequentially composing `#if` 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 point grows linearly (that is, $n+1$) with $n$. The performance results of analyzing $\texttt{foo}_n$, 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 $\lvert\mathbb{K}\rvert = 2^{16}$ =65,536 configurations, the analysis $\overline{\mathcal{A}}_{\Pi}(P)$ took 181 seconds, while the analysis $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ took only 5.2 seconds thus giving speed

$$\big(\overbrace{[\mathtt{i}=3]}^{A_1 \wedge A_2 \wedge A_3}, \overbrace{[\mathtt{i}=2]}^{A_1 \wedge A_2 \wedge \neg A_3}, \overbrace{[\mathtt{i}=2]}^{A_1 \wedge \neg A_2 \wedge A_3}, \overbrace{[\mathtt{i}=1]}^{A_1 \wedge \neg A_2 \wedge \neg A_3}, \overbrace{[\mathtt{i}=2]}^{\neg A_1 \wedge A_2 \wedge A_3}, \overbrace{[\mathtt{i}=1]}^{\neg A_1 \wedge A_2 \wedge \neg A_3}, \overbrace{[\mathtt{i}=1]}^{\neg A_1 \wedge \neg A_2 \wedge A_3}, \overbrace{[\mathtt{i}=0]}^{\neg A_1 \wedge \neg A_2 \wedge \neg A_3}\big)$$

**Figure 8.** $\overline{\mathcal{A}}_{\Pi}(P)$ results at point ⑤ of $\mathtt{foo}_3$().

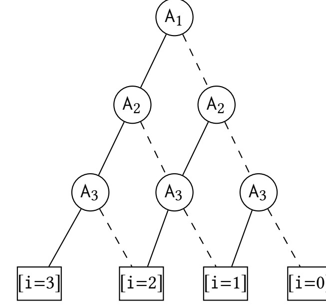



**Figure 9.** $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ results at point ⑤ of $\mathtt{foo}_3$().

**Table 2.** The performance results of analyzing $\mathtt{foo}_n$.

| n | $\overline{\mathcal{A}}_{\Pi}(P)$ | $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ | Improve |
|---|---|---|---|
| 3 | 0.0046 | 0.0017 | 2.7× |
| 5 | 0.0253 | 0.0047 | 5.4× |
| 10 | 1.6228 | 0.1304 | 12.4× |
| 15 | 85.525 | 3.7461 | 22.8× |
| 16 | 181.38 | 5.2260 | 34.7× |
| 17 | infeasible | 6.5258 | - |
| 18 | infeasible | 7.3149 | - |

up of 35 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 6.5 seconds, producing a BDD with 18 leaf nodes: one node for each i$\in \{0, \dots, 17\}$. Hence, BDDs can not only speed up analyses, but also turn previously infeasible analyses feasible (addresses Objective (**O2**)).

***Application scenarios.*** Let us consider the program $P_1$

```
①  #if (A) int x := [10, 20]; #endif
②  #if (¬A) int x := [0, 10]; #endif
③  int y := [0, 1];
④  if (y ≥ 1) x := −x;
⑤  assert(x!=0); ⑥
```

which has only one feature A. When A is on, the program stores a random value from [10, 20] in x. Otherwise, when A is off, it stores a random value from [0, 10] in x. Then, depending on the value of the non-deterministic variable y, x is negated or not. We want to show that the assertion at the program point ⑤, is correct for the config A. 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 point ④ the join of the then branch of conditional and its else branch. Hence, it will report in point ⑤ that $\mathtt{x} \in [-20, 20]$ for the config A and $\mathtt{x} \in [-10, 10]$ for the configuration ¬ A. Thus, the assertion will fail for both configurations. However, the lifted analysis using the Polyhedra domain will be more successful. The found invariant for the config A at point ⑤ is: $10 \leq \mathtt{x}+30\mathtt{y} \leq 20 \wedge 0 \leq \mathtt{y} \leq 1$. Hence, in this case, the analysis will correctly conclude that the assertion is correct for the configuration A (addresses Objective (**O3**)). However, the assertion may fail for the configuration ¬ A.

Let us consider the program $P_2$:

```
①  #if (A) int x := [−10, 10]; #endif
②  #if (¬A) int x := [0, 10]; #endif
③  int y := x;
④  if (x ≤ 0) y := −y;
⑤  assert(y<0); ⑥
```

which has one feature A. When A is on, the program stores a random value from [−10, 10] in x. Otherwise, when A is off, it stores a random value from [0, 10] in x. Then, we store in y the absolute value of x. In the program point ⑤, we want to check the given assertion. For example, later on in the program, there are references to an array using the index y (e.g. a[y] := 0). In this way, we want to verify that there are no array-out-of-bounds references.

Lifted analysis using the Interval domain is not able to deduce that the assertion is correct, since it finds that $-10 \leq \mathtt{y} \leq 10$ in point ⑤ for both configurations A and ¬ A. Still, the lifted analyses using Octagons and Polyhedra are able to prove that the assertion is correct, since they show that $\mathtt{y} \geq 0$ at ⑤ for both A and ¬ A (addresses Objective (**O3**)).

## 8 Related Work

We divide our discussion of related work into four categories: analyses based on disjunctive abstract domains, lifted analyses, other lifted techniques, and other types of families.

***Analyses based on disjunctive abstract domains.*** The use of disjunctive abstract domains in static analysis has attracted considerable attention recently. Decision trees have been used for the disjunctive refinement of the interval domain [25]. A segmented decision tree abstract domain where disjunctions are determined by value of variables is proposed in [14], whereas in [7] disjunctions are determined by the branch conditions. The Function analyzer [38] for proving program termination is also based on decision tree abstract domains for defining ranking functions, where decision nodes contain constraints that split the memory space and the leaves contain affine expressions. Logico-numerical

abstract domain implemented using BDDAPRON and specifically designed acceleration methods are used in [36] to verify synchronous data-flow programs with Boolean and numerical variables, such as LUSTRE programs. The BDDAPRON library has been developed by Jeannet [28] to implement a relational inter-procedural analysis of concurrent programs.

***Lifted analyses.*** Brabrand et. al. [4] lift a dataflow analysis from the monotone framework, resulting in a tuple-based lifted dataflow analysis that works on the level of families. Another efficient implementation of the lifted dataflow analysis formulated within the IFDS framework was proposed in SPL$^{\text{LIFT}}$ [3]. It has been shown that the running time of analyzing all variants in a family is close to the analysis of a single program. However, this technique is limited to work only for analyses phrased within the IFDS framework, a subset of dataflow analyses with certain properties, such as distributivity of transfer functions. Many dataflow analyses, including interval and octagon analyses, are not distributive and cannot be encoded in IFDS. A formal methodology for systematic derivation of tuple-based lifted static analyses from existing single-program analyses phrased in the abstract interpretation framework was proposed in [33]. 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 [17–20]. Variability abstractions introduced in [17, 19] 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. The works [18, 20] propose an automatic two-phase procedure for effective lifted analyses, where in the first phase a specifically designed pre-analysis is run to calculate suitable variability abstractions, and then in the second phase the calculated abstract lifted analysis is performed. However, the above tuple-based lifted analyses [4, 17, 18, 33] are only applied to CIDE-based Java program families [31] using toy client analyses, such as reaching definitions and uninitialized variables. On the other hand, here we consider `#ifdef`-based C program families 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.

***Other lifted techniques.*** Various approaches have been proposed for lifting other existing analysis and verification techniques to work on the level of families (see [37] for a survey). Besides the family-based (lifted) strategy, the survey [37] identifies *product-based* (all variants are analyzed one by one) and *sampling strategy* (only a random subset of variants is analyzed) as possible ways of analyzing product lines (see also [2]). 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 [39] performs software model checking of program families based on variability encoding which transforms compile-time to runtime variability [27], VarexJ [32] performs dynamic analysis of program families based on variability-aware execution, whereas SuperC [24] 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 numeric 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) [9]. Several specially designed lifted model checking algorithms for efficient verification of temporal properties of such models have been proposed [9, 16, 21, 22].

***Other types of program families.*** In this work, we consider annotation-based program families where variability is integrated with the common code base. Apart from C-Preprocessors [30], graphical CIDE [31] and the choice calculus [23] represent another methods to implement such annotation-based program families. The graphical CIDE (Colored IDE) is an Eclipse plug-in which annotates variability in program code using background colors, such that every feature is associated with a unique color. The choice calculus is a simple, formal language for representing variability in a way that maximizes sharing and minimizes redundancy, which is similar to the goals of the binary decision diagram domain used here. The annotation-based families contrast sharply with composition-based program families [1], where features are implemented as separate and composable units. In this approach, features are developed and tested independently, and then combined in a prescribed manner to produce the desired set of variants. They have also been interesting for analysis and verification [6].

## 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 symbolic 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.

## References


[1] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[2] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification: case studies and experiments. In *35th International Conference on Software Engineering, ICSE '13*, pages 482–491, 2013.

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

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

[5] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[6] M. Chechik, I. Stavropoulou, C. Disenfeld, and J. Rubin. FPH: efficient non-commutativity analysis of feature-based systems. In *Fundamental Approaches to Software Engineering, 21st Inter. Conference, FASE'18, Proceedings*, volume 10802 of *LNCS*, pages 319–336. Springer, 2018.

[7] J. Chen and P. Cousot. A binary decision tree abstract domain functor. In *Static Analysis - 22nd International Symposium, SAS 2015, Proceedings*, volume 9291 of *LNCS*, pages 36–53. Springer, 2015.

[8] N. V. Chernikova. Algorithm for finding a general formula for the non-negative solutions of a system of linear inequalities. *USSR Computational Mathematics and Mathematical Physics*, 5(2):228—-233, 1965.

[9] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. *IEEE Trans. Software Eng.*, 39(8):1069–1089, 2013.

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

[11] P. Cousot and 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)*, pages 238–252. ACM, 1977.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'79*, pages 269–282, 1979.

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

[14] P. Cousot, R. Cousot, and L. Mauborgne. A scalable segmented decision tree abstract domain. In *Time for Verification, Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 72–95. Springer, 2010.

[15] P. Cousot and 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)*, pages 84–96. ACM Press, 1978.

[16] A. S. Dimovski, A. S. Al-Sibahi, C. Brabrand, and A. Wasowski. Efficient family-based model checking via variability abstractions. *STTT*, 19(5):585–603, 2017.

[17] A. S. Dimovski, C. Brabrand, and A. Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conf. on Object-Oriented Programming, ECOOP 2015*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[18] A. S. Dimovski, C. Brabrand, and A. Wasowski. Finding suitable variability abstractions for family-based analysis. In *FM 2016: Formal Methods - 21st International Symposium, Proceedings*, volume 9995 of *LNCS*, pages 217–234. Springer, 2016.

[19] A. S. Dimovski, C. Brabrand, and A. Wasowski. Variability abstractions for lifted analysis. *Sci. Comput. Program.*, 159:1–27, 2018.

[20] A. S. Dimovski, C. Brabrand, and A. Wasowski. Finding suitable variability abstractions for lifted analysis. *Formal Asp. Comput.*, 31(2):231–259, 2019.

[21] A. S. Dimovski, A. Legay, and 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*, volume 11424 of *LNCS*, pages 192–209. Springer, 2019.

[22] A. S. Dimovski and A. Wasowski. From transition systems to variability models and from lifted model checking back to UPPAAL. In *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, volume 10460 of *LNCS*, pages 249–268. Springer, 2017.

[23] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol.*, 21(1):6:1–6:27, Dec. 2011.

[24] P. Gazzillo and R. Grimm. Superc: parsing all of C by taming the preprocessor. In J. Vitek, H. Lin, and F. Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 323–334. ACM, 2012.

[25] A. Gurfinkel and S. Chaki. Boxes: A symbolic abstract domain of boxes. In *Static Analysis - 17th International Symposium, SAS 2010. Proceedings*, volume 6337 of *LNCS*, pages 287–303. Springer, 2010.

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

[27] A. F. Iosif-Lazar, A. S. Al-Sibahi, A. S. Dimovski, J. E. Savolainen, K. Sierszecki, and A. Wasowski. Experiences from designing and validating a software modernization transformation (E). In *30th IEEE/ACM Inter. Conf. on Automated Software Engineering, ASE'15*, pages 597–607, 2015.

[28] B. Jeannet. Relational interprocedural verification of concurrent programs. In *Seventh IEEE Inter. Conf. on Software Engineering and Formal Methods, SEFM'09*, pages 83–92. IEEE Computer Society, 2009.

[29] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, 21st Inter. Conf., CAV'09.*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[30] C. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* PhD thesis, University of Magdeburg, Germany, May 2010.

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

[32] J. Meinicke, C. Wong, C. Kästner, T. Thüm, and 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*, pages 483–494. ACM, 2016.

[33] J. Midtgaard, A. S. Dimovski, C. Brabrand, and A. Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015.

[34] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

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

[36] P. Schrammel and B. Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *Static Analysis - 18th International Symposium, SAS 2011. Proceedings*, volume 6887, pages 233–248. Springer, 2011.

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

[38] C. Urban and A. Miné. A decision tree abstract domain for proving conditional termination. In *Static Analysis - 21st International Symposium, SAS 2014. Proceedings*, volume 8723 of *LNCS*, pages 302–318. Springer, 2014.

[39] A. von Rhein. *Analysis strategies for configurable systems.* PhD thesis, University of Passau, Germany, 2016.