Several Lifted Abstract Domains for Static Analysis of Numerical Program Families

Aleksandar S. Dimovski[a], Sven Apel[b], Axel Legay[c]

[a]*Mother Teresa University, 12 Udarna Brigada 2a, 1000 Skopje, Mkd*
[b]*Saarland University, Campus E1.1, 66123 Saarbrücken, Germany*
[c]*Université catholique de Louvain, 1348 Ottignies-Louvain-la-Neuve, Belgium*

Abstract

Lifted (*family-based*) *static analysis based on abstract interpretation* is capable of 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. The elements of the underlying lifted domain are tuples, which maintain one property per system variant. Still, explicit property enumeration in tuples, one by one for all variants, immediately yields combinatorial explosion. This is particulary apparent in the case of program families that, apart from Boolean features, contain also numerical features with large domains, thus giving rise to astronomical configuration spaces.

The key for an efficient lifted analysis is a proper handling of variability-specific constructs of the language (e.g., feature-based runtime tests and `#if` directives). In this work, we introduce new symbolic representations of the lifted domain that can efficiently analyze program families with numerical features. This makes sharing between property elements corresponding to different variants explicitly possible. In the first approach, elements of the new lifted domain are *decision trees*, in which decision nodes are labeled with linear constraints defined over numerical features and the leaf nodes belong to an existing single-program analysis domain. The lifted domain is parametric in the choice of the domains for representing linear constraints and leaf nodes. Furthermore, we propose another alternative approach for efficient lifted analysis. We encode a program family with numerical features as a family with only Boolean features, and then use a BDD lifted domain to analyze the resulting program family.

To illustrate the potential of our representations, we have implemented an experimental lifted static analyzer, called SPLNum^{2}Analyzer, for infer-

ring invariants of `#if`-annotated C programs. The tool implements all three approaches for lifted analysis based on abstract interpretation: tuple-based, decision tree-based, and BDD-based. It uses existing numerical abstract domains (e.g., intervals, octagons, polyhedra) from the APRON library as parameters. An empirical evaluation on benchmarks from SV-COMP and BusyBox yields promising results indicating that our tool can be successfully used for analyzing program families with very large configuration spaces.

Keywords: Static Analysis, Abstract Interpretation, Software Product Lines, Lifted Domains: Tuples, Decision Trees, and BDDs

---

1. Introduction

Many software systems today are configurable [1]: they use *features* (or configurable options) to control presence and absence of functionality. Different family members, called variants, are derived by switching features on and off, while reuse of common code is maximized, leading to productivity gains, shorter time to market, greater market coverage, etc. Program families (a.k.a. Software Product Lines) are commonly seen in the development of commercial embedded software, such as cars, phones, avionics, medicine, robotics, 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 consider here program families implemented using `#if` directives from the C preprocessor `CPP` [2]. They use `#if`-s to specify under which conditions parts of code should be included or excluded from a variant.

Classic program families use only Boolean features that have two values: on and off. However, Boolean features are insufficient for real-world program families, as there exist features that have a range of numbers as possible values. These features are called *numerical features* [3, 4]. For instance, Linux kernel, BusyBox, Apache web server, Java Garbage Collector are real-world program families that contain numerical features, also called *numerical program families*. Their analysis is very challenging, due to the fact that from only a few features, huge number of variants can be derived.

Software verification and static analysis of program families address the problem of checking that all variants of a given family satisfy certain properties [5]. In this paper, we are concerned with the verification of numerical

program families using abstract interpretation-based static analysis. *Abstract interpretation* [6, 7] is a general theory for approximating the semantics of programs. It provides sound (all positive answers are correct) and efficient (with a good tradeoff between precision and cost) static analyses of run-time properties of real programs. It has been used as the foundation for various successful industrial-scale static analyzers, such as ASTREE [8]. Still, 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) in practice. 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 [9, 10]. Therefore, we use so-called *lifted* (a.k.a. *family-based or variability-aware*) *static analyses* [9, 10, 11], which analyze all variants of the family simultaneously, without generating any of the variants explicitly. They take as input the common code base, which encodes all variants of a program family, and produce analysis results corresponding to all variants. The standard abstract interpretation-based lifted analysis [9, 11] uses a lifted domain that represents a n-fold product of an existing single-program analysis domain for expressing program properties (where n is the number of valid configurations). That is, the lifted domain maintains one property element per valid variant in tuples. This explicit property enumeration in tuples is 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 (or even faster) with the number of features. This problem has been successfully addressed for program families that contain only Boolean features [12, 13, 14, 15], by using sharing through binary decision diagrams (BDDs). However, the fundamental limitation of existing lifted analysis techniques is that they are not able to handle numerical features.

To overcome this limitation, we present two new approaches for effectively analyzing program families with both Boolean and numerical features by means of abstract interpretation. The first approach is inspired by the decision tree abstract domain proposed by Urban and Mine [16, 17, 18] for proving program termination. In particular, elements of our lifted domain are *decision trees*, in which decision nodes are labelled with linear constraints over features, whereas leaf nodes belong to a single-program analysis domain. The decision trees recursively partition the space of configurations (i.e., the space of possible combinations of feature values), whereas the program prop-

erties at the leaves provide analysis information corresponding to each partition, i.e. to the variants (configurations) that satisfy the constraints along the path to the given leaf node. The partitioning is dynamic, which means that partitions are split by feature-based tests (at `#if` directives), and joined when merging the corresponding control flows again. In terms of decision trees, this means that new decision nodes are added by feature-based tests and removed when merging control flows. In fact, the partitioning of the set of configurations is semantics-based, which means that linear constraints over numerical features that occur in decision nodes are automatically inferred by the analysis and do not necessarily occur syntactically in the code base. The decision tree lifted domain is parametric in the choice of numerical domain which underlies the linear constraints over numerical features labelling decision nodes, and the choice of the single-program analysis domain for leaf nodes. In fact, in our implementation, we also use numerical domains for leaf nodes, which encode linear constraints over program variables. The numerical domains, such as intervals [6], octagons [19], polyhedra [20], are widely used in practice to maintain information about the set of possible values of variables along with the possible relations between them. To implement the decision tree lifted domain, we adapt the operations of the decision tree termination domain [17, 18] for implementing decision nodes and use the numerical domains from the APRON library [21] for implementing leaf nodes.

The second approach for lifted analysis uses a boolean encoding of numerical program families and a binary decision diagram (BDD) lifted domain introduced by [15]. In particular, we first define a syntactic transformation to convert a numerical program family into a Boolean program family that contains only Boolean features. This is done by replacing every atomic linear constraint over numerical features by a fresh Boolean feature. Then, we use the BDD lifted domain [15] to analyze the resulting Boolean program family. To implement the BDD lifted domain, we use the BDDAPRON library [22] that represents the power domain of Boolean formulae and any APRON domain. The operations provided by the BDDAPRON library are highly optimized, which often leads to better time performances of this approach. For example, the library of our decision tree lifted domain is still a prototype implementation and many operations can be further optimized and improved. In fact, the efficient implementation is the main motivation for introducing the BDD-based approach to analyze numerical program families.

Another approach for efficient lifted analysis would be to replace compile-time variability with run-time variability (non-determinism) [23]. In particu-

lar, a given program family is transformed into a single program by encoding features with ordinary program variables that are non-deterministically initialized to any value from their domain and by encoding static configuration options (`#if` directives) with conditional `if` statements. The resulting single programs, called *variability simulators*, can be analyzed using off-the-shelf single-program analyzers. For example, we can use the numerical abstract domains for this aim. However, the convexity of these numerical domains makes them infer conjunctions of linear constraints over variables. This leads to imprecisions and rough approximations in the analysis, which often results in failure of showing the required program properties. The lifted domains proposed here can remedy this shortcoming by considering disjunctions arising from features. The elements of lifted domains partition the space of all possible values of features inducing disjunctions into the base domain defined over program variables. This way, we obtain more precise, but also more expensive lifted analysis.

We have implemented a tool, called SPLNum^{2}Analyzer, [1] that performs a *forward reachability analysis* of C program families with Boolean and numerical features using three lifted domains: tuples, decision trees, and BDDs. The tool computes a set of possible numerical invariants, which represent linear constraints over program variables, in all program locations. We can use the invariants inferred from the implemented lifted static analyzer to check invariance properties of C program families, such as assertions, buffer overflows, null pointer references, division by zero, etc [8].

In summary, we make several contributions:

- We propose a new, parameterized lifted analysis domain based on decision trees for analyzing numerical program families.
- We propose a new lifted analysis based on Boolean encoding of numerical program families and BDD-based lifted domain.
- We implement a prototype lifted static analyzer, SPLNum^{2}Analyzer, that performs a forward analysis of `#if`-enriched C programs, where numerical domains from the APRON library are used as parameters.

[1] Num^2 in the name of the tool refers to its ability to both handle Numerical features and to perform Numerical client analysis of SPLs (program families).

- We evaluate our approaches for automatic inference of program invariants. We compare performances of our lifted analyzers based on tuples, decision trees, and BDDs; as well as the approach based on single-program analysis and variability simulator.

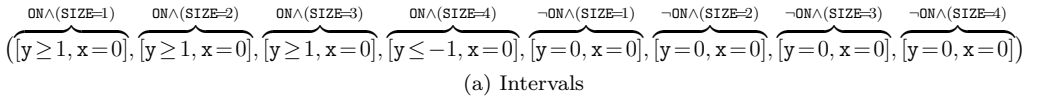
This work extends and revises the conference article [24]. We make the following extensions here: (1) we provide another approach for lifted analysis using Boolean encoding and BDD-based lifted domain; (2) we expand the evaluation by implementing the new BDD-based approach, by considering more benchmarks, and by extending the performance results including a comparison with a single-program analysis of variability simulators; (3) we provide formal proofs for all main results in the work; (4) we provide additional illustrations, explanations, and examples. The paper proceeds with a motivating example that illustrates our new approaches for lifted analysis. The language for writing program families is introduced in Section 3. The basics of tuple-based lifted analysis are introduced in Section 4. Section 5 defines our new decision tree-based lifted analysis, while Section 6 describes the alternative approach via 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

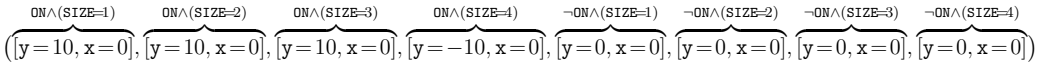
To illustrate the potential of our new lifted domains, we now consider a motivating example using the following program family SIMPLE:

```
1   int x := 10, y := 0;
2   while (x ≠ 0) {
3       x := x-1;
4       #if (SIZE ≤ 3) y := y+1; #else y := y-1; #endif
5       #if (¬ON) y := 0; #else skip; #endif 6}
7   assert (y > 1);
```

The set $\mathbb{F}$ of features is $\{\texttt{ON}, \texttt{SIZE}\}$, where `ON` is a Boolean feature and `SIZE` is a numerical feature whose domain is $[1, 4] = \{1, 2, 3, 4\}$. Thus, the set of valid configurations is $\mathbb{K} = \{\texttt{ON} \wedge (\texttt{SIZE}=1), \texttt{ON} \wedge (\texttt{SIZE}=2), \texttt{ON} \wedge (\texttt{SIZE}=3), \texttt{ON} \wedge (\texttt{SIZE}=4), \neg\texttt{ON} \wedge (\texttt{SIZE}=1), \neg\texttt{ON} \wedge (\texttt{SIZE}=2), \neg\texttt{ON} \wedge (\texttt{SIZE}=3), \neg\texttt{ON} \wedge (\texttt{SIZE}=4)\}$. The code of SIMPLE contains two `#if` directives, which change the value assigned to `y`, depending on how features from $\mathbb{F}$ are set at compile-time. For each configuration from $\mathbb{K}$, a different variant (single program)



(a) Intervals

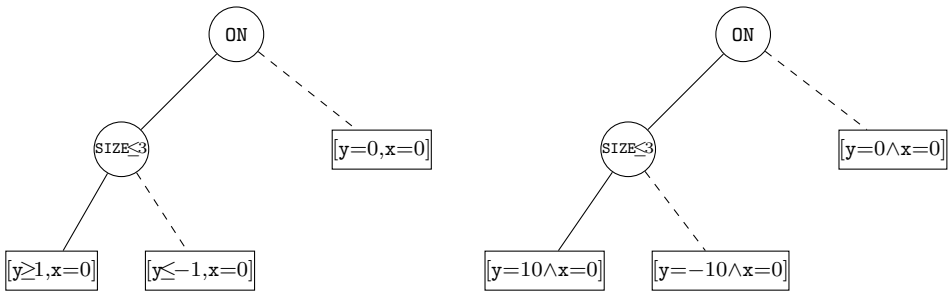


(b) Polyhedra.

Figure 1: Tuple-based analyses results at program location ⑦ of SIMPLE.

can be generated by appropriately resolving `#if`-s. For example, the variant corresponding to configuration $\texttt{ON} \wedge (\texttt{SIZE} = 1)$ will have `ON` and `SIZE` set to true and 1, so that the assignment `y := y+1` and `skip` in program locations ④ and ⑤, respectively, will be included in this variant. The variant for configuration $\neg\texttt{ON} \wedge (\texttt{SIZE} = 4)$ will have features `ON` and `SIZE` set to false and 4, so the assignments `y := y-1` and `y := 0` in program locations ④ and ⑤, respectively, will be included in it. There are $|\mathbb{K}| = 8$ variants that can be derived from the family SIMPLE.

Assume that we want to perform *lifted analysis* of SIMPLE using the numerical domains: *intervals* [6] and *polyhedra* [20]. The standard lifted domain used in the literature [9, 11] is defined as the Cartesian product of $|\mathbb{K}|$ copies of the basic analysis domain (e.g. interval, polyhedra). Hence, elements of the lifted domain are tuples containing one component for each valid configuration from $\mathbb{K}$, where each component represents a linear constraint over program variables (`x` and `y` in this case). The analysis results at location ⑦ of SIMPLE obtained using the lifted *interval and polyhedra* analyses are 8-sized tuples shown in Fig. 1a and Fig. 1b, respectively. Note that the first component of a tuple in Fig. 1 corresponds to configuration $\texttt{ON} \wedge (\texttt{SIZE} = 1)$, the second to $\texttt{ON} \wedge (\texttt{SIZE} = 2)$, the third to $\texttt{ON} \wedge (\texttt{SIZE} = 3)$, and so on. We can see in Fig. 1 that the interval analysis discovers imprecise (approximative) results for the variable `y`: $(\texttt{y} \geq 1)$ for configurations $\texttt{ON} \wedge (\texttt{SIZE} = 1)$, $\texttt{ON} \wedge (\texttt{SIZE} = 2)$, and $\texttt{ON} \wedge (\texttt{SIZE} = 3)$, as well as $(\texttt{y} \leq -1)$ for configuration $\texttt{ON} \wedge (\texttt{SIZE} = 4)$. This is due to the fact that the interval analysis cannot reason about the relations between variables `x` and `y`. Using this result at location ⑦, we can not answer whether the given assertion $(\texttt{y} > 1)$ is valid or fails for the configurations $\texttt{ON} \wedge (\texttt{SIZE} = 1)$, $\texttt{ON} \wedge (\texttt{SIZE} = 2)$, and $\texttt{ON} \wedge (\texttt{SIZE} = 3)$, whereas we can successfully conclude that the assertion fails for all other configurations. However, the polyhedra lifted analysis reports the most precise results for both `x` and `y` in all configurations: $(\texttt{y} = 10)$ for



(a) Intervals (b) Polyhedra.

Figure 2: Decision tree-based analyses results at program location ⑦ of SIMPLE (solid edges = true, dashed edges = false). Each constraint is satisfied by the left subtree of the decision node, while the right subtree satisfies its negation.

configurations $\texttt{ON} \wedge (\texttt{SIZE} = 1)$, $\texttt{ON} \wedge (\texttt{SIZE} = 2)$, and $\texttt{ON} \wedge (\texttt{SIZE} = 3)$, as well as $(\texttt{y} = -10)$ for $\texttt{ON} \wedge (\texttt{SIZE} = 4)$. This is due to the fact that the polyhedra domain is fully relational and is able to track all (linear) relations between program variables. Using the polyhedra analysis, we can answer that the assertion is valid for $\texttt{ON} \wedge (\texttt{SIZE} = 1)$, $\texttt{ON} \wedge (\texttt{SIZE} = 2)$, and $\texttt{ON} \wedge (\texttt{SIZE} = 3)$.

If we perform lifted interval and polyhedra analyses based on the decision tree domain proposed in this work, then the corresponding decision trees inferred at the final program location ⑦ of SIMPLE are depicted in Fig. 2a and Fig. 2b, respectively. Notice that the inner nodes of the decision tree in Fig. 2 are labeled with Boolean features (ON) and *interval* linear constraints over numerical features (SIZE), while the leaves are labeled with the elements from the property domain we use (interval and polyhedra linear constraints over program variables x and y). The edges of decision trees are labeled with the truth value of the decision on the parent node; we use solid edges for true (i.e. the constraint in the parent node is satisfied) and dashed edges for false (i.e. the negation of the constraint in the parent node is satisfied). As decision nodes partition the space of valid configurations $\mathbb{K}$, we implicitly assume the correctness of linear constraints that take into account domains of numerical features. For example, the node with constraint $(\texttt{SIZE} \leq 3)$ is satisfied when $(\texttt{SIZE} \leq 3) \wedge (1 \leq \texttt{SIZE} \leq 4)$, whereas its negation is satisfied when $(\texttt{SIZE} > 3) \wedge (1 \leq \texttt{SIZE} \leq 4)$. The constraint $(1 \leq \texttt{SIZE} \leq 4)$ represents the domain $[1, 4]$ of SIZE. We can see that decision trees offer more possibilities for sharing and interaction between analysis properties corresponding to different configurations, they provide symbolic and compact representation of lifted analysis elements. For example, Fig. 2b presents polyhedra properties

of the two program variables x and y, which are partitioned with respect to features ON and SIZE. When (ON $\wedge$ (SIZE ≤ 3)) is true the shared property is (y=10, x=0), whereas when (ON $\wedge \neg$(SIZE ≤ 3)) is true the shared property is (y=-10, x=0). When $\neg$ON is true, the property is independent from the value of SIZE, hence a node with a constraint over SIZE is not needed. Therefore, the cases when $\neg$ON is true are all identical, and share the same leaf node (y=0, x=0). This invariant can be written as the following disjunctive property in first order logic: $\big(\texttt{ON} \wedge (\texttt{SIZE} \leq 3) \wedge \texttt{y} = 10 \wedge \texttt{x} = 0\big) \vee \big(\texttt{ON} \wedge \neg(\texttt{SIZE} \leq 3) \wedge \texttt{y} = -10 \wedge \texttt{x} = 0\big) \vee \big(\neg\texttt{ON} \wedge \texttt{y} = 0, \texttt{x} = 0\big)$.

The third approach for analyzing numerical program families is based on the BDD lifted domain [15]. First, the code of SIMPLE is transformed by replacing the linear constraint (SIZE ≤ 3) with a fresh Boolean feature SIZE3 (see the result of this transformation in Fig. 8). Then, the BDD lifted domain proposed in [15] is used to analyze the resulting Boolean program family. BDDs inferred at the final program location (7) are the same as in the case of decision trees in Fig. 2, but now the linear constraint in the decision node (SIZE ≤ 3) is replaced with SIZE3 (see also Fig. 9). That is, when (ON $\wedge$ SIZE3) holds the shared property is (y=10, x=0), whereas when (ON $\wedge \neg$SIZE3) holds the shared property is (y= $-$ 10, x=0).

In summary, we observe that decision tree-based and BDD-based representations use only three leaf nodes (properties), whereas the tuple-based representation uses eight properties. This ability for sharing is the key motivation behind the usage of new representations.

Alternatively, the program family SIMPLE can be transformed into a single program, called variability simulator [23], which can be analyzed using the single-program APRON polyhedra domain. More specifically, variability simulator of SIMPLE is obtained by encoding features as ordinary program variables that are non-deterministically initialized, that is: int SIZE $= [1, 4]$, ON $= [0, 1]$, and by encoding #if directives as ordinary if statements. The single-program polyhedra analysis will infer the invariant: $-10 \leq \texttt{y} \leq 10$ at location (7). However, this invariant is not strong enough to establish the validity of the given assertion. Although the assertion is valid for variants satisfying ON $\wedge$ (SIZE ≤ 3), convex numerical domains that only infer conjunctions of constraints will not be able to establish the validity. As we have seen, lifted domains can remedy this problem by allowing disjunctions determined by values of features.

3. A Language for Program Families

Let $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite and totaly ordered set of *numerical features* available in a program family. For each feature $A \in \mathbb{F}$, $\mathrm{dom}(A) \subseteq \mathbb{Z}$ denotes the set of possible values that can be assigned to A. Note that any Boolean feature can be represented as a numerical feature $B \in \mathbb{F}$ with $\mathrm{dom}(B) = \{0, 1\}$, such that 0 means that feature B is disabled while 1 means that B is enabled. A valid combination of feature's values represents a *configuration* k, which specifies one *variant* of a program family. It is given as a *valuation function* $k : \mathbb{F} \to \mathbb{Z}$, which is a mapping that assigns a value from $\mathrm{dom}(A)$ to each feature A, i.e. $k(A) \in \mathrm{dom}(A)$ for any $A \in \mathbb{F}$. We assume that only a subset $\mathbb{K}$ of all possible configurations are *valid*. Note that $\mathrm{dom}(A)$ is finite for each $A \in \mathbb{F}$, thus $\mathbb{K}$ is also a finite set. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathbb{K}$ can be represented by a formula: $(A_1 = k(A_1)) \wedge \ldots \wedge (A_n = k(A_n))$. We often abbreviate $(B = 1)$ with B and $(B = 0)$ with $\neg B$, for a Boolean feature $B \in \mathbb{F}$. The set of valid configurations $\mathbb{K}$ can also be represented as a formula: $\vee_{k \in \mathbb{K}} k$.

The set of valid configurations $\mathbb{K}$ of a program family with numerical features is typically described by a *numerical feature model* [3], [2] i.e. a tree-like structure that describes which combinations of feature's values and relationships among them are valid. In this work we disregard syntactic representations of the set $\mathbb{K}$, as we are concerned with behavioural analysis of program families rather than with implementation details of $\mathbb{K}$. Therefore, we use the set- and logic-theoretic views of $\mathbb{K}$, which is convenient for our purpose.

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

$$\theta ::= \mathrm{true} \mid ef_1 \bowtie ef_2 \mid \neg\theta \mid \theta_1 \wedge \theta_2 \mid \theta_1 \vee \theta_2, \qquad ef ::= n \mid A \mid ef_1 \boxplus ef_2$$

where $A \in \mathbb{F}$, $n \in \mathbb{Z}$, $\boxplus \in \{+, -, *\}$, and $\bowtie \in \{=, <\}$. When a configuration $k \in \mathbb{K}$ satisfies a feature expression $\theta \in \mathit{FeatExp}(\mathbb{F})$, 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 $k \models \theta$.

[2] Other terms that appear (e.g. [25]) as a way to extend classical Boolean feature models with non-Boolean features are *extended, advanced,* or *attributed feature models.*

Example 1 *Let us revisit the program family* SIMPLE *from Section 2. The set* $\mathbb{F}$ *of features is* $\{B, \mathit{SIZE}\}$*, where* $\mathrm{dom}(\mathit{SIZE}) = [1, 4]$*. There are eight possible valid configurations* $\mathbb{K} = \{\mathit{ON} \wedge (\mathit{SIZE} = 1), \mathit{ON} \wedge (\mathit{SIZE} = 2), \mathit{ON} \wedge (\mathit{SIZE} = 3), \mathit{ON} \wedge (\mathit{SIZE} = 4), \neg\mathit{ON} \wedge (\mathit{SIZE} = 1), \neg\mathit{ON} \wedge (\mathit{SIZE} = 2), \neg\mathit{ON} \wedge (\mathit{SIZE} = 3), \neg\mathit{ON} \wedge (\mathit{SIZE} = 4)\}$*. For the feature expression* $(\mathit{SIZE} \leq 3)$*, we have* $[\![(\mathit{SIZE} \leq 3)]\!] = \{\mathit{ON} \wedge (\mathit{SIZE} = 1), \mathit{ON} \wedge (\mathit{SIZE} = 2), \mathit{ON} \wedge (\mathit{SIZE} = 3), \neg\mathit{ON} \wedge (\mathit{SIZE} = 1), \neg\mathit{ON} \wedge (\mathit{SIZE} = 2), \neg\mathit{ON} \wedge (\mathit{SIZE} = 3)\}$*. Therefore, it holds that* $\mathit{ON} \wedge (\mathit{SIZE} = 2) \models (\mathit{SIZE} \leq 3)$ *and* $\mathit{ON} \wedge (\mathit{SIZE} = 4) \not\models (\mathit{SIZE} \leq 3)$*, where* $\mathit{ON} \wedge (\mathit{SIZE} = 2) \in \mathbb{K}$*,* $\mathit{ON} \wedge (\mathit{SIZE} = 4) \in \mathbb{K}$*, and* $(\mathit{SIZE} \leq 3) \in \mathit{FeatExp}(\mathbb{F})$*.* □

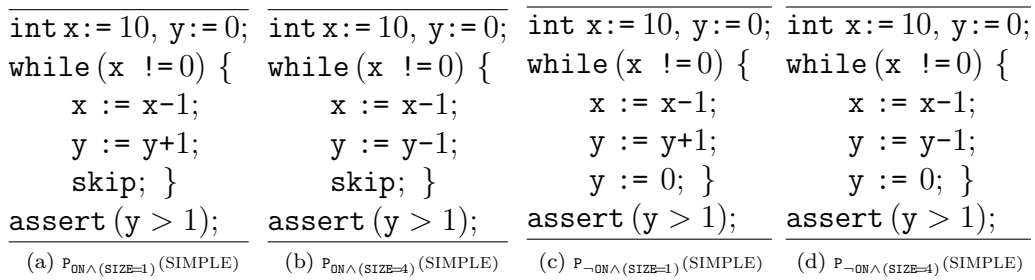
We consider a simple sequential non-deterministic programming language, which will be used to exemplify our work. The variables are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. Note that our implementation, described in Section 7, actually supports a subset of the C language enriched with `#if`-s, which is sufficient to handle realistic program families. To encode multiple variants, a new compile-time conditional statement is included. The new statement "`#if` (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 syntax is:

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

where n ranges over integers, $[n, n']$ over integer intervals, `x` over program variables *Var*, and $\oplus$ over binary arithmetic-logic 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 non-determinism useful to model user input or to approximate expressions. The set of all statements s is denoted by *Stm*; the set of all expressions e is denoted by *Exp*.

Remark *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 constructs can be desugared and represented only by* `#if` *construct.*

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 `#if`-s) corresponding to k as the



(a) $\mathtt{P}_{\mathtt{ON}\wedge(\mathtt{SIZE}=1)}$(SIMPLE) (b) $\mathtt{P}_{\mathtt{ON}\wedge(\mathtt{SIZE}=4)}$(SIMPLE) (c) $\mathtt{P}_{\neg\mathtt{ON}\wedge(\mathtt{SIZE}=1)}$(SIMPLE) (d) $\mathtt{P}_{\neg\mathtt{ON}\wedge(\mathtt{SIZE}=4)}$(SIMPLE)

Figure 3: Different variants of the program family SIMPLE from Section 2.

output. Second, the obtained variant is evaluated using the standard single-program semantics [11]. 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 "$\mathtt{\#if}\ (\theta)\ s\ \mathtt{\#endif}$", 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:

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

Notice that since any $k \in \mathbb{K}$ is a valuation function, we have that either $k \models \theta$ holds or $k \not\models \theta$ (which is equivalent to $k \models \neg\theta$) holds, for any $\theta \in \mathit{FeatExp}(\mathbb{F})$. For example, $\mathtt{P}_{\mathtt{ON}\wedge(\mathtt{SIZE}=1)}(\text{SIMPLE})$, $\mathtt{P}_{\mathtt{ON}\wedge(\mathtt{SIZE}=4)}(\text{SIMPLE})$, $\mathtt{P}_{\neg\mathtt{ON}\wedge(\mathtt{SIZE}=1)}(\text{SIMPLE})$, and $\mathtt{P}_{\neg\mathtt{ON}\wedge(\mathtt{SIZE}=4)}(\text{SIMPLE})$, shown in Fig. 3a, Fig. 3b, Fig. 3c, and Fig. 3d, respectively, are variants derived from SIMPLE defined in Section 2.

4. Lifted Analysis based on Tuples

In this section, we introduce the product lifted domain.

4.1. Lifted Analysis with Tuples

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 the variability of program families into account.

Lifted analysis as defined by Midtgaard et al. [11] rely on a lifted domain that is $|\mathbb{K}|$-*fold product* of an existing single-program abstract domain $\mathbb{A}$ defined over the set of program variables *Var*. The abstract domain

$\mathbb{A}$ employs data structures and algorithms specific to the shape of invariants (analysis properties) it represents and manipulates. We assume that the single-program abstract domain $\mathbb{A}$ is equipped with sound operators for concretization $\gamma_{\mathbb{A}}$, ordering $\sqsubseteq_{\mathbb{A}}$, least upper bound (called join) $\sqcup_{\mathbb{A}}$, greatest upper bound (called meet) $\sqcap_{\mathbb{A}}$, the least element (called bottom) $\bot_{\mathbb{A}}$, the greatest element (called 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 forward assignments $\text{ASSIGN}_{\mathbb{A}}$. More specifically, the concretization function $\gamma_{\mathbb{A}}$ assigns a concrete meaning to each element in the abstract domain $\mathbb{A}$, ordering $\sqsubseteq_{\mathbb{A}}$ conveys the idea of approximation since some analysis results may be coarser than some other results, whereas join $\sqcup_{\mathbb{A}}$ and meet $\sqcap_{\mathbb{A}}$ convey the idea of convergence since a new abstract element is computed when merging control flows. To analyze loops effectively and efficiently, the convergence acceleration operators such as widening $\nabla_{\mathbb{A}}$ and narrowing $\triangle_{\mathbb{A}}$ are used. Transfer functions give abstract semantics of expressions and statements at the level of single-programs. Hence, $\text{FILTER}_{\mathbb{A}}(a : \mathbb{A}, e : \mathit{Exp})$ returns an abstract element from $\mathbb{A}$ obtained by restricting the input abstract element a to satisfy the given test e, whereas $\text{ASSIGN}_{\mathbb{A}}(a : \mathbb{A}, \texttt{x:=}e : \mathit{Stm})$ returns an updated version of the input abstract element a by abstractly evaluating assignment $\texttt{x:=}e$ in it.

Numerical Property Domains. In this work, we will instantiate $\mathbb{A}$ with one of the numerical property domains $\langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$ [7], such as intervals, octagons, and polyhedra. They differ in expressive power and computational complexity. The *Interval domain* [6], denoted as $\langle I, \sqsubseteq_I \rangle$, is a non-relational numerical domain that identifies the range of possible values for every variable as an interval. The elements are: $\{\bot_I\} \cup \{[l, h] \mid l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leq h\}$. The *Octagon domain* [19], denoted as $\langle O, \sqsubseteq_O \rangle$, is a weakly-relational numerical domain, where elements are conjunctions of linear constraints of the form $\pm\texttt{x}_j \pm \texttt{x}_i \geq \beta$ between variables $\texttt{x}_i$ and $\texttt{x}_j$, and $\beta \in \mathbb{Z}$. The *Polyhedra domain* [20], denoted as $\langle P, \sqsubseteq_P \rangle$, is a fully relational numerical property domain. It expresses conjunctions of linear constraints of the form $\alpha_1\texttt{x}_1 + \ldots + \alpha_n\texttt{x}_n + \beta \geq 0$, where $\texttt{x}_1$, ..., $\texttt{x}_n$ are variables and $\alpha_i, \beta \in \mathbb{Z}$.

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 configuration of $\mathbb{K}$.

Example 2 *Consider the tuple in Fig. 1a, in which components are properties from the Interval domain and* $\mathbb{K} = \{\texttt{ON} \wedge (\texttt{SIZE} = 1), \texttt{ON} \wedge (\texttt{SIZE} =$

$2), \mathit{ON} \wedge (\mathit{SIZE} = 3), \mathit{ON} \wedge (\mathit{SIZE} = 4), \neg\mathit{ON} \wedge (\mathit{SIZE} = 1), \neg\mathit{ON} \wedge (\mathit{SIZE} = 2), \neg\mathit{ON} \wedge (\mathit{SIZE} = 3), \neg\mathit{ON} \wedge (\mathit{SIZE} = 4)\}$. *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. 1a can be written as the following disjunctive property:*

$$\begin{array}{l}
\big(\mathit{ON} \wedge (\mathit{SIZE}{=}1) \wedge [\boldsymbol{y} \geq 1, \boldsymbol{x}{=}0]\big) \vee \big(\mathit{ON} \wedge (\mathit{SIZE}{=}2) \wedge [\boldsymbol{y} \geq 1, \boldsymbol{x}{=}0]\big) \vee \\
\big(\mathit{ON} \wedge (\mathit{SIZE}{=}3) \wedge [\boldsymbol{y} \geq 1, \boldsymbol{x}{=}0]\big) \vee \big(\mathit{ON} \wedge (\mathit{SIZE}{=}4) \wedge [\boldsymbol{y} \leq -1, \boldsymbol{x}{=}0]\big) \vee \\
\big(\neg\mathit{ON} \wedge (\mathit{SIZE}{=}1) \wedge [\boldsymbol{y}{=}0, \boldsymbol{x}{=}0]\big) \vee \big(\neg\mathit{ON} \wedge (\mathit{SIZE}{=}2) \wedge [\boldsymbol{y}{=}0, \boldsymbol{x}{=}0]\big) \vee \\
\big(\neg\mathit{ON} \wedge (\mathit{SIZE}{=}3) \wedge [\boldsymbol{y}{=}0, \boldsymbol{x}{=}0]\big) \vee \big(\neg\mathit{ON} \wedge (\mathit{SIZE}{=}4) \wedge [\boldsymbol{y}{=}0, \boldsymbol{x}{=}0]\big)
\end{array}$$

Lifted Operations. Given a tuple (i.e., an element of the product lifted domain) $\overline{a} \in \mathbb{A}^{\mathbb{K}}$, the projection π_k selects the k^{th} component of $\overline{a}$. All lifted operations are defined by lifting the corresponding operations of the domain $\mathbb{A}$ configuration-wise.

$$\begin{array}{ll}
\overline{\gamma}(\overline{a}) = \prod_{k\in\mathbb{K}}(\gamma_{\mathbb{A}}(\pi_k(\overline{a}))), & \overline{a_1} \,\dot{\sqsubseteq}\, \overline{a_2} \equiv \pi_k(\overline{a_1}) \sqsubseteq_{\mathbb{A}} \pi_k(\overline{a_2}), \forall k \in \mathbb{K} \\
\overline{a_1} \,\dot{\sqcup}\, \overline{a_2} = \prod_{k\in\mathbb{K}}(\pi_k(\overline{a_1}) \sqcup_{\mathbb{A}} \pi_k(\overline{a_2})), & \overline{a_1} \,\dot{\sqcap}\, \overline{a_2} = \prod_{k\in\mathbb{K}}(\pi_k(\overline{a_1}) \sqcap_{\mathbb{A}} \pi_k(\overline{a_2})) \\
\dot{\top} = \prod_{k\in\mathbb{K}} \top_{\mathbb{A}} = (\top_{\mathbb{A}}, \ldots, \top_{\mathbb{A}}), & \dot{\bot} = \prod_{k\in\mathbb{K}} \bot_{\mathbb{A}} = (\bot_{\mathbb{A}}, \ldots, \bot_{\mathbb{A}}) \\
\overline{a_1} \,\dot{\nabla}\, \overline{a_2} = \prod_{k\in\mathbb{K}}(\pi_k(\overline{a_1}) \nabla_{\mathbb{A}} \pi_k(\overline{a_2})), & \overline{a_1} \,\dot{\triangle}\, \overline{a_2} = \prod_{k\in\mathbb{K}}(\pi_k(\overline{a_1}) \triangle_{\mathbb{A}} \pi_k(\overline{a_2}))
\end{array}$$

Lifted Transfer Functions. We now introduce lifted transfer functions for giving abstract semantics of expressions and statements at the level of program families. We define lifted transfer functions for tests, forward assignments ($\overline{\texttt{ASSIGN}}$), and `#if`-s ($\overline{\texttt{IFDEF}}$). There are two types of tests: *expression-based tests*, denoted $\overline{\texttt{FILTER}}$, that occur in `while` and `if` statements, and *feature-based tests*, denoted $\overline{\texttt{FEAT-FILTER}}$, that occur in `#if`-s. Each lifted transfer function takes as input a tuple from $\mathbb{A}^{\mathbb{K}}$ representing the invariant (i.e., property) before evaluating the statement (resp., expression) to handle, and returns a tuple from $\mathbb{A}^{\mathbb{K}}$ representing the invariant after evaluating the given statement (resp., expression).

$$\begin{array}{l}
\overline{\texttt{FILTER}}(\overline{a} : \mathbb{A}^{\mathbb{K}},\ e : \mathit{Exp}) = \prod_{k\in\mathbb{K}}(\texttt{FILTER}_{\mathbb{A}}(\pi_k(\overline{a}), e)) \\
\overline{\texttt{FEAT-FILTER}}(\overline{a} : \mathbb{A}^{\mathbb{K}},\ \theta : \mathit{FeatExp}(\mathbb{F})) = \prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\overline{a}), & \text{if } k \models \theta \\ \bot_{\mathbb{A}}, & \text{if } k \not\models \theta \end{cases} \\
\overline{\texttt{ASSIGN}}(\overline{a} : \mathbb{A}^{\mathbb{K}},\ \texttt{x:=}e : \mathit{Stm}) = \prod_{k\in\mathbb{K}}(\texttt{ASSIGN}_{\mathbb{A}}(\pi_k(\overline{a}), \texttt{x:=}e)) \\
\overline{\texttt{IFDEF}}(\overline{a} : \mathbb{A}^{\mathbb{K}},\ \texttt{\#if}\ (\theta)\ s\ \texttt{\#endif} : \mathit{Stm}) = \overline{[\![s]\!]}(\overline{\texttt{FEAT-FILTER}}(\overline{a},\ \theta)) \,\dot{\sqcup}\, \overline{\texttt{FEAT-FILTER}}(\overline{a},\ \neg\theta)
\end{array}$$

where $\overline{[\![s]\!]}(\overline{a})$ represents the lifted transfer function in $\mathbb{A}^{\mathbb{K}}$ for statement s. $\overline{\texttt{FILTER}}$ and $\overline{\texttt{ASSIGN}}$ are defined by applying $\texttt{FILTER}_{\mathbb{A}}$ and $\texttt{ASSIGN}_{\mathbb{A}}$ independently on each component of the input tuple $\overline{a}$. $\overline{\texttt{FEAT-FILTER}}$ keeps those components k of the input tuple $\overline{a}$ that satisfy θ, otherwise it replaces the other components of $\overline{a}$ that do not satisfy θ with $\bot_{\mathbb{A}}$. $\overline{\texttt{IFDEF}}$ captures the effect of analyzing the statement s in those components k of $\overline{a}$ that satisfy θ, otherwise it is an identity for the other components that do not satisfy θ.

Lifted Analysis. The operators and transfer functions of the lifted domain $\mathbb{A}^{\mathbb{K}}$ are combined together to analyze program families. Initially, we build a tuple where all components are set to $\top_{\mathbb{A}}$ for the first program location. The $\dot{\top}$ analysis properties are propagated forward from the first program location towards the final program location taking assignments, `#if`-s, and (expression- and feature-based) tests into account with join and widening around `while`-s. We apply delayed widening [26], which means we start extrapolating by widening only after some fixed number of iterations we analyze the loop's body. We improve the precision of the solution obtained by delayed widening by further applying a narrowing operator [26]. The soundness and correctness of the lifted analysis based on $\langle \mathbb{A}^{\mathbb{K}}, \dot{\sqsubseteq} \rangle$ follows immediately from the soundness of all operators and transfer functions of $\langle \mathbb{A}, \sqsubseteq_{\mathbb{A}} \rangle$ (see the proof in [11]).

Example 3 *Consider the program family* SIMPLE *from Section 2. We want to perform lifted* interval *analysis of* SIMPLE *using the lifted domain $I^{\mathbb{K}}$. The final analysis results at program locations from ① to ⑦ are shown in Fig. 4. They represent 8-sized tuples, which contain one interval property (store) for each configuration.* □

5. Lifted Analysis based on Decision Trees

In this section, we introduce a new lifted domain that relies on *decision trees*. Its elements are disjunctions of the leaf nodes that belong to an existing single-program analysis domain $\mathbb{A}$ defined over program variables *Var*. The leaf nodes are separated by linear constraints over features, organized in the decision nodes. Hence, we encapsulate the set of valid configurations $\mathbb{K}$ into the decision nodes where each top-down path represents one or several configurations from $\mathbb{K}$ that satisfy the constraints encountered along the given path. We store in each leaf node the property from an existing

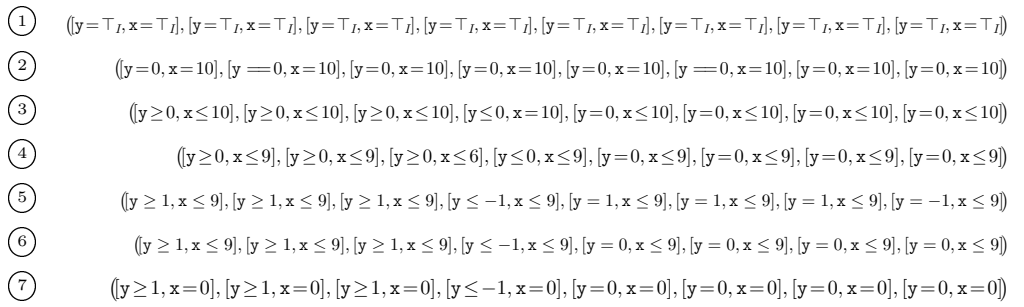


Figure 4: Tuple-based (interval) analyses results at all program locations of SIMPLE.

single-program analysis domain $\mathbb{A}$ generated from the variants representing the corresponding configurations.

We assume $\mathbb{F} = \{A_1, \ldots, A_n\}$ be a finite and totally ordered set of numerical features, such that the ordering is $A_1 > A_2 > \ldots > A_n$. We develop an efficient lifted analysis for numerical program families, which is based on a lifted domain of decision trees.

Domain for Decision Nodes. We define the family of abstract domains for linear constraints $\mathbb{C}_{\mathbb{D}}$, which are parameterized by any of the numerical domains $\mathbb{D}$ (intervals I, octagons O, polyhedra P). We use $\mathbb{C}_I = \{\pm A_i \geq \beta \mid A_i \in \mathbb{F}, \beta \in \mathbb{Z}\}$ to denote the finite set of *interval constraints*, $\mathbb{C}_O = \{\pm A_i \pm A_j \geq \beta \mid A_i, A_j \in \mathbb{F}, \beta \in \mathbb{Z}\}$ to denote the finite set of *octagonal constraints*, and $\mathbb{C}_P = \{\alpha_1 A_1 + \ldots + \alpha_n A_n + \beta \geq 0 \mid A_1, \ldots A_n \in \mathbb{F}, \alpha_1, \ldots, \alpha_n, \beta \in \mathbb{Z}, \gcd(|\alpha_1|, \ldots, |\alpha_n|, |\beta|) = 1\}$ to denote the finite set of *polyhedral constraints*. We have $\mathbb{C}_I \subseteq \mathbb{C}_O \subseteq \mathbb{C}_P$.

The finite set $\mathbb{C}_{\mathbb{D}}$ of linear constraints over features $\mathbb{F}$ is constructed by the underlying domain $\langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$ using the Galois connection $\langle \mathcal{P}(\mathbb{C}_{\mathbb{D}}), \sqsubseteq_{\mathbb{D}} \rangle \underset{\alpha_{\mathbb{C}_{\mathbb{D}}}}{\overset{\gamma_{\mathbb{C}_{\mathbb{D}}}}{\leftrightarrows}} \langle \mathbb{D}, \sqsubseteq_{\mathbb{D}} \rangle$, where $\mathcal{P}(\mathbb{C}_{\mathbb{D}})$ is the power set of $\mathbb{C}_{\mathbb{D}}$. The concretization function $\gamma_{\mathbb{C}_{\mathbb{D}}} : \mathbb{D} \to \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ maps an interval (resp., an octagon, a polyhedron) that represents a conjunction of constraints to a finite set of interval (resp., octagonal, polyhedral) constraints. We have $\gamma_{\mathbb{C}_{\mathbb{D}}}(\top_{\mathbb{D}}) = \emptyset$ and $\gamma_{\mathbb{C}_{\mathbb{D}}}(\bot_{\mathbb{D}}) = \{\bot_{\mathbb{C}_{\mathbb{D}}}\}$, where $\bot_{\mathbb{C}_{\mathbb{D}}}$ is an unsatisfiable constraint such as $-1 \geq 0$.

The domain of decision nodes is $\mathbb{C}_{\mathbb{D}}$. We impose a total order $<_{\mathbb{C}_{\mathbb{D}}}$ on $\mathbb{C}_{\mathbb{D}}$ to be the lexicographic order on the coefficients $\alpha_1, \ldots, \alpha_n$ and constant

α_{n+1} of the linear constraints [17], such that:

$$(\alpha_1 \cdot A_1 + \ldots + \alpha_n \cdot A_n + \alpha_{n+1} \geq 0) <_{\mathbb{C}_\mathbb{D}} (\alpha'_1 \cdot A_1 + \ldots + \alpha'_n \cdot A_n + \alpha'_{n+1} \geq 0)$$
$$\iff \exists j > 0. \forall i < j. (\alpha_i = \alpha'_i) \wedge (\alpha_j < \alpha'_j)$$

The negation of linear constraints is formed as: $\neg(\alpha_1 A_1 + \ldots \alpha_n A_n + \beta \geq 0) = -\alpha_1 A_1 - \ldots - \alpha_n A_n - \beta - 1 \geq 0$. For example, the negation of $A - 3 \geq 0$ is the constraint $-A + 2 \geq 0$ (i.e., $A \leq 2$). To ensure canonical representation of decision trees, a linear constraint c and its negation $\neg c$ cannot both appear as nodes in a decision tree. Hence, we only keep the largest constraint with respect to $<_{\mathbb{C}_\mathbb{D}}$ between c and $\neg c$.

Lifted Domain for Decision Trees. A *decision tree* $t \in \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{A})$ over the sets $\mathbb{C}_\mathbb{D}$ of linear constraints defined over features $\mathbb{F}$ and the leaf abstract domain $\mathbb{A}$ defined over program variables *Var* is either a leaf node $\langle\!|a|\!\rangle$ with $a \in \mathbb{A}$, or $[\![c : tl, tr]\!]$, where $c \in \mathbb{C}_\mathbb{D}$ (denoted by $t.c$) is the *smallest* constraint with respect to $<_{\mathbb{C}_\mathbb{D}}$ appearing in the tree t, tl (denoted by $t.l$) is the left subtree of t representing its *true branch*, and tr (denoted by $t.r$) is the right subtree of t representing its *false branch*. The path along a decision tree establishes the set of configurations (those that satisfy the encountered constraints), and the leaf nodes represent the analysis properties for the corresponding configurations.

Example 4 *The following two decision trees t_1 and t_2 have decision nodes labelled with Interval linear constraints over the numerical feature* `SIZE` *with domain $\{1, 2, 3, 4\}$, whereas leaf nodes are Interval linear constraints over the integer program variable y:*

$$t_1 = [\![\texttt{SIZE} \geq 4 : \langle\!|[y \geq 2]|\!\rangle, \langle\!|[y = 0]|\!\rangle]\!], \; t_2 = [\![\texttt{SIZE} \geq 2 : \langle\!|[y \geq 0]|\!\rangle, \langle\!|[y \leq 0]|\!\rangle]\!]$$

Lifted Operations. The *concretization function* $\gamma_\mathbb{T}$ of a decision tree $t \in \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{A})$ returns $\gamma_\mathbb{A}(a)$ for $k \in \mathbb{K}$, where k satisfies the set $C \in \mathcal{P}(\mathbb{C}_\mathbb{D})$ of constraints accumulated along the top-down path to the leaf node $a \in \mathbb{A}$. Formally, $\gamma_\mathbb{T}(t) = \overline{\gamma}_\mathbb{T}[\mathbb{K}](t)$. The function $\overline{\gamma}_\mathbb{T}$ accumulates into a set C (initially equal to $\mathbb{K} = \vee_{k \in \mathbb{K}} k$) constraints along the paths up to a leaf node:

$$\overline{\gamma}_\mathbb{T}[C](\langle\!|a|\!\rangle) = \prod_{k \models C} \gamma_\mathbb{A}(a),$$
$$\overline{\gamma}_\mathbb{T}[C]([\![c : tl, tr]\!]) = \overline{\gamma}_\mathbb{T}[C \cup \{c\}](tl) \times \overline{\gamma}_\mathbb{T}[C \cup \{\neg c\}](tr)$$

Note that $k \models C$ is equivalent with $\alpha_{\mathbb{C}_\mathbb{D}}(\{k\}) \sqsubseteq_\mathbb{D} \alpha_{\mathbb{C}_\mathbb{D}}(C)$, where $\alpha_{\mathbb{C}_\mathbb{D}}(C)$ represents a conjunction of linear constraints from the set C. Therefore, we can check $k \models C$ using the operation $\sqsubseteq_\mathbb{D}$ of the numerical domain $\mathbb{D}$.

Other binary operations of $\mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{A})$ are based on Algorithm 1 for *tree unification* [17], which finds a common refinement (labelling) of two trees t_1 and t_2 by calling function $\texttt{UNIFICATION}(t_1, t_2, \mathbb{K})$. It possibly adds new constraints as decision nodes (Lines 5–7, Lines 11–13), or removes constraints that are redundant (Lines 3,4,9,10,15,16). The function UNIFICATION accumulates into the set $C \in \mathcal{P}(\mathbb{C}_\mathbb{D})$ (initialized to $\mathbb{K}$), constraints encountered along the paths of the decision tree. We use the function isRedundant(c, C) to check whether a linear constraint $c \in \mathbb{C}_\mathbb{D}$ is redundant with respect to a set C by testing $\alpha_{\mathbb{C}_\mathbb{D}}(C) \sqsubseteq_\mathbb{D} \alpha_{\mathbb{C}_\mathbb{D}}(\{c\})$. Note that the tree unification does not lose any information.

Algorithm 1: $\texttt{UNIFICATION}(t_1, t_2, C)$

1 **if** isLeaf$(t_1) \wedge$ isLeaf(t_2) **then return** (t_1, t_2);
2 **if** isLeaf$(t_1) \vee$ (isNode$(t_1) \wedge$ isNode$(t_2) \wedge t_2.c <_{\mathbb{C}_\mathbb{D}} t_1.c)$ **then**
3 **if** isRedundant$(t_2.c, C)$ **then return** $\texttt{UNIFICATION}(t_1, t_2.l, C)$;
4 **if** isRedundant$(\neg t_2.c, C)$ **then return** $\texttt{UNIFICATION}(t_1, t_2.r, C)$;
5 $(l_1, l_2) = \texttt{UNIFICATION}(t_1, t_2.l, C \cup \{t_2.c\})$;
6 $(r_1, r_2) = \texttt{UNIFICATION}(t_1, t_2.r, C \cup \{\neg t_2.c\})$;
7 **return** $(\llbracket t_2.c : l_1, r_1 \rrbracket, \llbracket t_2.c : l_2, r_2 \rrbracket)$;
8 **if** isLeaf$(t_2) \vee$ (isNode$(t_1) \wedge$ isNode$(t_2) \wedge t_1.c <_{\mathbb{C}_\mathbb{D}} t_2.c)$ **then**
9 **if** isRedundant$(t_1.c, C)$ **then return** $\texttt{UNIFICATION}(t_1.l, t_2, C)$;
10 **if** isRedundant$(\neg t_1.c, C)$ **then return** $\texttt{UNIFICATION}(t_1.r, t_2, C)$;
11 $(l_1, l_2) = \texttt{UNIFICATION}(t_1.l, t_2, C \cup \{t_1.c\})$;
12 $(r_1, r_2) = \texttt{UNIFICATION}(t_1.r, t_2, C \cup \{\neg t_1.c\})$;
13 **return** $(\llbracket t_1.c : l_1, r_1 \rrbracket, \llbracket t_1.c : l_2, r_2 \rrbracket)$;
14 **else**
15 **if** isRedundant$(t_1.c, C)$ **then return** $\texttt{UNIFICATION}(t_1.l, t_2.l, C)$;
16 **if** isRedundant$(\neg t_1.c, C)$ **then return** $\texttt{UNIFICATION}(t_1.r, t_2.r, C)$;
17 $(l_1, l_2) = \texttt{UNIFICATION}(t_1.l, t_2.l, C \cup \{t_1.c\})$;
18 $(r_1, r_2) = \texttt{UNIFICATION}(t_1.r, t_2.r, C \cup \{\neg t_1.c\})$;
19 **return** $(\llbracket t_1.c : l_1, r_1 \rrbracket, \llbracket t_1.c : l_2, r_2 \rrbracket)$;

Example 5 *Consider decision trees t_1 and t_2 from Example 4. After tree unification $\texttt{UNIFICATION}(t_1, t_2, \mathbb{K})$, the resulting decision trees are:*

$$t_1 = \llbracket \mathit{SIZE} \geq 4 : \langle\!\langle [y \geq 2] \rangle\!\rangle, \llbracket \mathit{SIZE} \geq 2 : \langle\!\langle [y = 0] \rangle\!\rangle, \langle\!\langle [y = 0] \rangle\!\rangle \rrbracket \rrbracket,$$
$$t_2 = \llbracket \mathit{SIZE} \geq 4 : \langle\!\langle [y \geq 0] \rangle\!\rangle, \llbracket \mathit{SIZE} \geq 2 : \langle\!\langle [y \geq 0] \rangle\!\rangle, \langle\!\langle [y \leq 0] \rangle\!\rangle \rrbracket \rrbracket$$

Note that `UNIFICATION` *adds a decision node for* $\texttt{SIZE} \geq 2$ *to the right subtree of* t_1*, whereas it adds a decision node for* $\texttt{SIZE} \geq 4$ *to* t_2 *and removes the redundant constraint* $\texttt{SIZE} \geq 2$ *from the resulting left subtree of* t_2*.* □

All binary operations are performed leaf-wise on the unified decision trees. Given two unified decision trees t_1 and t_2, their ordering $t_1 \sqsubseteq_{\mathbb{T}} t_2$ is:

$$\langle\!|a_1|\!\rangle \sqsubseteq_{\mathbb{T}} \langle\!|a_2|\!\rangle = a_1 \sqsubseteq_{\mathbb{A}} a_2,$$
$$[\![c : tl_1, tr_1]\!] \sqsubseteq_{\mathbb{T}} [\![c : tl_2, tr_2]\!] = (tl_1 \sqsubseteq_{\mathbb{T}} tl_2) \wedge (tr_1 \sqsubseteq_{\mathbb{T}} tr_2)$$

while their join $t_1 \sqcup_{\mathbb{T}} t_2$ is defined as:

$$\langle\!|a_1|\!\rangle \sqcup_{\mathbb{T}} \langle\!|a_2|\!\rangle = \langle\!|a_1 \sqcup_{\mathbb{A}} a_2|\!\rangle$$
$$[\![c : tl_1, tr_1]\!] \sqcup_{\mathbb{T}} [\![c : tl_2, tr_2]\!] = [\![c : tl_1 \sqcup_{\mathbb{T}} tl_2, tr_1 \sqcup_{\mathbb{T}} tr_2]\!]$$

Similarly to $t_1 \sqcup_{\mathbb{T}} t_2$, we compute meet (resp., widening and narrowing) $t_1 \sqcap_{\mathbb{T}} t_2$ (resp., $t_1 \nabla_{\mathbb{T}} t_2$ and $t_1 \triangle_{\mathbb{T}} t_2$) of two unified decision trees t_1 and t_2, in such a way that instead of the join operator $\sqcup_{\mathbb{A}}$ we use the meet operator $\sqcap_{\mathbb{A}}$ (resp., widening $\nabla_{\mathbb{A}}$ and narrowing $\triangle_{\mathbb{A}}$) of the abstract domain $\mathbb{A}$. The top element is a tree with a single $\top_{\mathbb{A}}$ leaf: $\top_{\mathbb{T}} = \langle\!|\top_{\mathbb{A}}|\!\rangle$, while the bottom element is a tree with a single $\bot_{\mathbb{A}}$ leaf: $\bot_{\mathbb{T}} = \langle\!|\bot_{\mathbb{A}}|\!\rangle$.

Example 6 *Consider the unified decision trees* t_1 *and* t_2 *from Example 5. We can see that* $t_1 \sqsubseteq_{\mathbb{T}} t_2$ *holds, and*

$$t_1 \sqcup_{\mathbb{T}} t_2 = [\![\texttt{SIZE} \geq 4 : \langle\!|[y \geq 0]|\!\rangle, [\![\texttt{SIZE} \geq 2 : \langle\!|[y \geq 0]|\!\rangle, \langle\!|[y \leq 0]|\!\rangle]\!]]$$
$$t_1 \sqcap_{\mathbb{T}} t_2 = [\![\texttt{SIZE} \geq 4 : \langle\!|[y \geq 2]|\!\rangle, [\![\texttt{SIZE} \geq 2 : \langle\!|[y = 0]|\!\rangle, \langle\!|[y = 0]|\!\rangle]\!]]$$

Lifted Transfer Functions. Transfer functions for assignments ($\texttt{ASSIGN}_{\mathbb{T}}$) and expression-based tests ($\texttt{FILTER}_{\mathbb{T}}$) modify only leaf nodes of a decision tree. In contrast, transfer functions for variability-specific constructs, such as feature-based tests ($\texttt{FEAT-FILTER}_{\mathbb{T}}$) and `#if`-s ($\texttt{IFDEF}_{\mathbb{T}}$) add, modify, or delete decision nodes of a decision tree. This is due to the fact that the analysis information about program variables is located only in leaf nodes, while the analysis information about features is located in decision nodes.

Transfer function $\texttt{ASSIGN}_{\mathbb{T}}$ for handling an assignment `x:=`e in the input tree t is described by Algorithm 2. Note that `x` is a program variable, and e may contain only program variables. We apply to each leaf node a of t transfer function $\texttt{ASSIGN}_{\mathbb{A}}$, which substitutes expression e for `x` in a. Similarly, transfer function $\texttt{FILTER}_{\mathbb{T}}$ for handling expression-based tests $e \in Exp$ is implemented by applying $\texttt{FILTER}_{\mathbb{A}}$ leaf-wise as described by Algorithm 3.

Algorithm 2: $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}e)$

1 **if** isLeaf(t) **then return** $\langle\!|\texttt{ASSIGN}_{\mathbb{A}}(t, \texttt{x:=}e)|\!\rangle$;
2 **return** $[\![t.c : \texttt{ASSIGN}_{\mathbb{T}}(t.l, \texttt{x:=}e), \texttt{ASSIGN}_{\mathbb{T}}(t.r, \texttt{x:=}e)]\!]$;

Algorithm 3: $\texttt{FILTER}_{\mathbb{T}}(t, e)$

1 **if** isLeaf(t) **then return** $\langle\!|\texttt{FILTER}_{\mathbb{A}}(t, e)|\!\rangle$;
2 **return** $[\![t.c : \texttt{FILTER}_{\mathbb{T}}(t.l, e), \texttt{FILTER}_{\mathbb{T}}(t.r, e)]\!]$;

Transfer function $\texttt{FEAT-FILTER}_{\mathbb{T}}$ for feature-based tests θ is described by Algorithm 4. It reasons by induction on the structure of θ (we assume negation is applied to atomic propositions). When θ is an atomic constraint over numerical features (Lines 2,3), we use $\texttt{FILTER}_{\mathbb{D}}$ to approximate θ, thus producing a set of constraints J, which are then added to the tree t, possibly discarding all paths of t that do not satisfy θ. This is done by calling function $\texttt{RESTRICT}(t, \mathbb{K}, J)$, which adds linear constraints from J to t in ascending order with respect to $<_{\mathbb{C}_{\mathbb{D}}}$ as shown in Algorithm 5. Note that θ may not be representable exactly in $\mathbb{C}_{\mathbb{D}}$ (e.g., in the case of non-linear constraints over $\mathbb{F}$), so $\texttt{FILTER}_{\mathbb{D}}$ may produce a set of constraints approximating it. When θ is a conjunction (resp., disjunction) of two feature expressions (Lines 4,5) (resp., (Lines 6,7)), the resulting decision trees are merged by operation meet $\sqcap_{\mathbb{T}}$ (resp., join $\sqcup_{\mathbb{T}}$). Function $\texttt{RESTRICT}(t, C, J)$, described in Algorithm 5, takes as input a decision tree t, a set C of linear constraints accumulated along paths up to a node, and a set J of linear constraints in canonical form that need to be added to t. For each constraint $j \in J$, there exists a boolean b_j that shows whether the tree should be constrained with respect to j (b_j is set to true) or with respect to $\neg j$ (b_j is set to false). When J is not empty, the linear constraints from J are added to t in ascending order with respect to $<_{\mathbb{C}_{\mathbb{D}}}$. At each iteration, the smallest linear constraint j is extracted from J (Line 9), and is handled appropriately based on whether j is smaller (Line 11–15), or greater or equal (Line 17–21) to the constraint at the node of t we currently consider.

Finally, transfer function $\texttt{IFDEF}_{\mathbb{T}}$ is defined as:

$$\texttt{IFDEF}_{\mathbb{T}}(t, \texttt{\#if}\ (\theta)\ s) = [\![s]\!]_{\mathbb{T}}(\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta)) \sqcup_{\mathbb{T}} \texttt{FEAT-FILTER}_{\mathbb{T}}(t, \neg\theta)$$

where $[\![s]\!]_{\mathbb{T}}(t)$ denotes transfer function in $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ for statement s.

Algorithm 4: $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta)$

```
1  switch θ do
2      case (e_{F_Z} ⋈ e_{F_Z}) || (¬(e_{F_Z} ⋈ e_{F_Z})) do
3          J = FILTER_D(⊤_D, θ); return RESTRICT(t, K, J)
4      case θ_1 ∧ θ_2 do
5          return FEAT-FILTER_T(t, θ_1) ⊓_T FEAT-FILTER_T(t, θ_2)
6      case θ_1 ∨ θ_2 do
7          return FEAT-FILTER_T(t, θ_1) ⊔_T FEAT-FILTER_T(t, θ_2)
```

Note that after applying a transfer function to analyze a statement, the obtained decision tree must be normalized (sorted) to remove possible multiple occurrences of a constraint c and possible occurrences of both c and $\neg c$. Decision trees also may contain some redundancy that can be exploited to further compress them [17]. Function $\texttt{COMPRESS}_{\mathbb{T}}(t, C)$, described by Algorithm 6, is applied to decision trees t that satisfy a set of constraints C in order to compress (reduce) their representation. We use five different optimizations. First, if constraints on a path to some leaf are unsatisfiable, we eliminate that leaf node (Lines 10–12). Second, if a decision node contains two same leaves, then we keep only one leaf and we also eliminate the decision node (Lines 8,9). Third, we generalize the previous optimization. If a decision node contains two same subtrees, then we keep only one subtree and we also eliminate the decision node (Lines 13,14). Fourth, if a decision node contains a left leaf and a right subtree, such that its left leaf is the same with the left leaf of its right subtree and the constraint in the decision node is less or equal to the constraint in the root of its right subtree, then we can eliminate the decision node and its left leaf (Lines 15,16). Finally, we have an analogous rule when a decision node contains a left subtree and a right leaf (Lines 17,18). When we want to compress a decision tree t obtained using lifted operations and transfer functions, we call the function $\texttt{COMPRESS}_{\mathbb{T}}(t, \mathbb{K})$. The set of constraints is initialized to $\mathbb{K}$, i.e. we initially include only those constraints that represent domains of numerical features.

Lifted Analysis. Lifted operations and transfer functions of $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ can now be used to analyze numerical program families. For each statement s, we define function $\texttt{L-ANALYSE}_{\mathbb{T}}(t : \mathbb{T}, s : \mathit{Stm})$ that takes as input a

Algorithm 5: `RESTRICT`(t, C, J)

if isEmpty(J) **then**
  if isLeaf(t) **then** return t;
  if isRedundant$(t.c, C)$ **then** return `RESTRICT`$(t.l, C, J)$;
  if isRedundant$(\neg t.c, C)$ **then** return `RESTRICT`$(t.r, C, J)$;
  $l =$ `RESTRICT`$(t.l, C \cup \{t.c\}, J)$;
  $r =$ `RESTRICT`$(t.r, C \cup \{\neg t.c\}, J)$;
  return $(\llbracket t.c : l, r \rrbracket)$;
else
  $j = min_{<_{\mathbb{C}_\mathbb{D}}}(J)$;
  if isLeaf$(t) \vee$ (isNode$(t) \wedge j <_{\mathbb{C}_\mathbb{D}} t.c$) **then**
    if isRedundant(j, C) **then** return `RESTRICT`$(t, C, J \backslash \{j\})$;
    if isRedundant$(\neg j, C)$ **then** return $\langle\!| \bot_{\mathbb{A}} |\!\rangle$;
    if $j =_{\mathbb{C}_\mathbb{D}} t.c$ **then** (**if** b_j **then** $t = t.l$; **else** $t = t.r$) ;
    if b_j **then return** $(\llbracket j :$ `RESTRICT`$(t, C \cup \{j\}, J \backslash \{j\}), \langle\!| \bot_{\mathbb{A}} |\!\rangle \rrbracket)$;
    else return $(\llbracket j : \langle\!| \bot_{\mathbb{A}} |\!\rangle,$ `RESTRICT`$(t, C \cup \{j\}, J \backslash \{j\}) \rrbracket)$;
  else
    if isRedundant$(t.c, C)$ **then** return `RESTRICT`$(t.l, C, J)$;
    if isRedundant$(\neg t.c, C)$ **then** return `RESTRICT`$(t.r, C, J)$;
    $l =$ `RESTRICT`$(t.l, C \cup \{t.c\}, J)$;
    $r =$ `RESTRICT`$(t.r, C \cup \{\neg t.c\}, J)$;
    return $(\llbracket t.c : l, r \rrbracket)$;

Algorithm 6: $\texttt{COMPRESS}_{\mathbb{T}}(t, C)$

```
1  switch t do
2      case ⟨|n|⟩ do
3          return ⟨|n|⟩;
4      case [[t.c : l, r]] do
5          l' = COMPRESS_T(t.l, C ∪ {t.c}) ;
6          r' = COMPRESS_T(t.r, C ∪ {¬t.c}) ;
7          switch l', r' do
8              case ⟨|n'_l|⟩, ⟨|n'_r|⟩ when n'_l = n'_r do
9                  return ⟨|n'_l|⟩;
10             case ⟨|n'_l|⟩, ⟨|n'_r|⟩ do
11                 if UNSAT(C ∪ {t.c}) then return ⟨|n'_r|⟩;
12                 if UNSAT(C ∪ {¬t.c}) then return ⟨|n'_l|⟩;
13             case [[c_1 : l_1, r_1]], [[c_2 : l_2, r_2]] when c_1 = c_2 ∧ l_1 = l_2 ∧ r_1 = r_2 do
14                 return [[c_1 : l_1, r_1]];
15             case ⟨|n'_l|⟩, [[c_2 : l_2, r_2]] when ⟨|n'_l|⟩ = l_2 ∧ c ≤ c_2 do
16                 return [[c_2 : l_2, r_2]];
17             case [[c_1 : l_1, r_1]], ⟨|n'_r|⟩ when ⟨|n'_r|⟩ = r_1 ∧ c_1 ≤ c do
18                 return [[c_1 : l_1, r_1]];
19             case default: do
20                 return [[t.c : l', r']];
```

decision tree t corresponding to the initial location of statement s, and outputs a decision tree corresponding to the final location of s. Function $\texttt{L-ANALYSE}_{\mathbb{T}}(t, s)$ is described by Algorithm 7. For a `while` loop, $\phi^{\nabla}_{\mathbb{T}}(x) = t \sqcup_{\mathbb{T}} \texttt{L-ANALYSE}_{\mathbb{T}}(\texttt{FILTER}_{\mathbb{T}}(x, e), s)$ and $\texttt{lfp}\, \phi^{\nabla}_{\mathbb{T}}$ is the limit of the following increasing chain defined by delayed widening:

$$y_0 = \bot_{\mathbb{T}}; \quad y_{n+1} = \phi^{\nabla}_{\mathbb{T}}(y_n), \text{ if } n < N; \quad y_{n+1} = y_n \,\nabla_{\mathbb{T}}\, \phi^{\nabla}_{\mathbb{T}}(y_n), \text{ if } n \geq N \quad (1)$$

for some fixed number N denoting the widening delay. The lifted analysis of a program s is defined as $\texttt{L-ANALYSE}_{\mathbb{T}}(t_{in}, s)$, where t_{in} is taken as input tree in the initial location of s. The tree t_{in} has only one leaf node $\top_{\mathbb{A}}$, and decision nodes define the set $\mathbb{K}$. Note that if there are no constraints on $\mathbb{K}$

so that $\vee_{k\in\mathbb{K}} k \equiv \text{true}$, then $t_{in} = \top_{\mathbb{T}}$. In this way, we collect the possible invariants in the form of decision trees at all locations.

We establish correctness of the lifted analysis based on $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ by showing that it produces identical results with the product lifted domain $\mathbb{A}^{\mathbb{K}}$. Let $[\![s]\!]_{\mathbb{T}}$ and $\overline{[\![s]\!]}$ denote transfer functions of statement s in $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ and $\mathbb{A}^{\mathbb{K}}$. Note that $[\![s]\!]_{\mathbb{T}}(t) = \texttt{L-ANALYSE}_{\mathbb{T}}(t, s)$. Given $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ and $\overline{a} \in \mathbb{A}^{\mathbb{K}}$, we denote by $t \equiv \overline{a}$ iff the tree t represents a compact decision tree representation of the tuple $\overline{a}$, where all analysis equivalent results in $\overline{a}$ are shared by the same leaf nodes in t. Note that $t_{in} \equiv \overline{a}_{in}$. Moreover, $\gamma_{\mathbb{T}}(t) = \overline{\gamma}(\overline{a})$ when $t \equiv \overline{a}$. We first show the following auxiliary result.

Lemma 7 *Let $t_1 \equiv \overline{a_1}$ and $t_2 \equiv \overline{a_2}$. Then, $t_1 \nabla_{\mathbb{T}} t_2 \ \equiv \ \overline{a_1} \dot{\nabla} \overline{a_2}$.*

Proof *The operator $\dot{\nabla}$ on $\mathbb{A}^{\mathbb{K}}$, defined as $\overline{a_1} \ \dot{\nabla} \ \overline{a_2} = \prod_{k\in\mathbb{K}} (\pi_k(\overline{a_1}) \nabla_{\mathbb{A}} \pi_k(\overline{a_2}))$, is a widening operator [26]. Let t_1 and t_2 represent decision tree representations of $\overline{a_1}$ and $\overline{a_2}$. By definition of $\nabla_{\mathbb{T}}$, t_1 and t_2 are first unified, and then $\nabla_{\mathbb{A}}$ is performed leaf-wise on unified trees. Recall that the tree unification does not lose any information. It can only replicate some leaf nodes in original t_1 and t_2, so that the structure (decision nodes) of t_1 and t_2 are unified. Let C be a set of constraints collected along the path to leaf nodes a_1 and a_2 of t_1 and t_2, respectively. Then, $t_1 \nabla_{\mathbb{T}} t_2$ will have a leaf node $a_1 \nabla_{\mathbb{A}} a_2$ and constraints from C are in decision nodes on the path from the root to that node. On the other hand, we will obtain the same result $a_1 \nabla_{\mathbb{A}} a_2$ for those components of $\overline{a_1} \ \dot{\nabla} \ \overline{a_2}$ that satisfy constraints C. Therefore, $t_1 \nabla_{\mathbb{T}} t_2 \ \equiv \ \overline{a_1} \dot{\nabla} \overline{a_2}$.*

Remark *The widening operator of the decision tree termination domain [17] first performs left unification of t_1 and t_2, which forces the structure of t_1 on t_2, and then applies leaf-wise the widening operator of the leaf domain. There may be information loss by applying the left unification. However, decision nodes of the termination domain [17] are constraints defined over program variables that split the (potentially infinite) memory space, whereas decision nodes of our lifted domain are constraints defined over feature variables that split the (finite) configuration space $\mathbb{K}$. Feature variable do not exhibit infinite behaviour in our language, since they are statically bound at compile-time and so their static values can be only read at run-time. Therefore, our widening operator can use the more precise tree unification algorithm.*

Lemma 8 (Termination) *The increasing chain $\{y_i\}_{i\geq 0}$ defined in Eqn. 1 stabilizes after a finite number of times: $\exists k \geq 0. y_k = y_{k+1}$.*

Proof *Each element $y_{n+1}, n \geq N$ is obtained by applying the widening operator $\nabla_{\mathbb{T}}$ on y_n that describes the result after n loop iterations and the result obtained after applying one more loop iteration $\phi^{\nabla}_{\mathbb{T}}(y_n)$. The widening operator first applies* `UNIFICATION`$(y_n, \phi^{\nabla}_{\mathbb{T}}(y_n), \mathbb{K})$ *that will produce two unified trees with the same labelling of decision nodes. This operation will (potentially) add new paths in the unified trees, thus refining the partitions of the configuration set $\mathbb{K}$. Since $\mathbb{K}$ is a finite set, there is no possibility to generate an infinite sequence of partition refinements of $\mathbb{K}$. In the extreme case, there will be one path corresponding to each configuration from $\mathbb{K}$ in the unified trees. Subsequently, on the unified trees y_n and $\phi^{\nabla}_{\mathbb{T}}(y_n)$, we apply the widening operator of the leaf domain $\nabla_{\mathbb{A}}$ leaf-wise. Basically, the leaves of unified trees y_n will stabilize first, and then their decision nodes will stabilize as well. First, no infinite sequences of partition refinements of $\mathbb{K}$ will be possible, since $\mathbb{K}$ is finite. Second, new paths (potentially) added to the left unified tree y_n with stabilized leaves will duplicate some existing leaves, which will be redundant and reduced using the* `COMPRESS` *function.*

Theorem 9 *Let $t \equiv \overline{a}$. Then, $[\![s]\!]_{\mathbb{T}}(t) \equiv \overline{[\![s]\!]}(\overline{a})$.*

Proof *The proof is by induction on the structure of s. We consider the two most interesting cases.*

Case `x:=`e`.` $\overline{\texttt{ASSIGN}}(\overline{a},$ `x:=`$e)$ *applies* $\texttt{ASSIGN}_{\mathbb{A}}(a,$ `x:=`$e)$ *to each component $a = \pi_k(\overline{a})$ of $\overline{a}$.* $\texttt{ASSIGN}_{\mathbb{T}}(t,$ `x:=`$e)$ *applies* $\texttt{ASSIGN}_{\mathbb{A}}(a,$ `x:=`$e)$ *to each leaf a in the tree t. The proof follows by correctness of $t \equiv \overline{a}$.*

Case `#if` (θ) s `#endif.` *Transfer functions for* `#if` *are identical in both lifted domains. We only need to show that functions* $\overline{\texttt{FEAT-FILTER}}(\overline{a}, \theta)$ *and* $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta)$ *are identical. This can be shown by induction on θ. Assume that θ is an atomic constraint.* $\overline{\texttt{FEAT-FILTER}}(\overline{a}, \theta)$ *keeps only those components $\pi_k(\overline{a})$ of $\overline{a}$ such that $k \models \theta$. On the other hand,* $\texttt{FEAT-FILTER}_{\mathbb{T}}(t, \theta)$ *first produces all linear constraints in $\mathbb{D}$ that satisfy θ, and then adds them in the tree t. Thus, it keeps only those leaf nodes that satisfy the newly generated constraints from θ.*

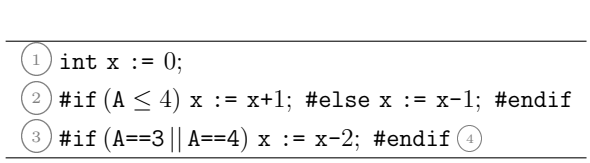
Case `while` (e) `do` s`.` *Transfer functions for* `while` *are identical in both lifted domains (see Eqn 1). The proof follows by Lemma 7, Lemma 8, and structural induction.*

Algorithm 7: $\texttt{L-ANALYSE}_{\mathbb{T}}(t, s)$

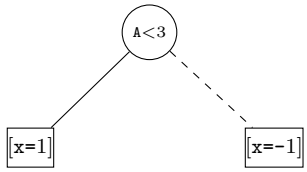
```
switch s do
    case skip do
        return t;
    case x:=e do
        return ASSIGN_T(t, x:=e);
    case if (e) then s1 else s2 do
        return L-ANALYSE_T(FILTER_T(t, e), s1) ⊔_T
        L-ANALYSE_T(FILTER_T(t, ¬e), s2);
    case s1; s2 do
        return L-ANALYSE_T(L-ANALYSE_T(t, s1), s2);
    case while (e) do s do
        return FILTER_T(lfp φ^▽_T, ¬e);
    case #if (θ) s #endif do
        return IFDEF_T(t, #if (θ) s #endif);
```

Example 10 *Let us consider the code base of a program family P given in Fig. 5a. It contains only one numerical feature* `A` *with domain* $[0, 99]$. *The decision tree inferred at the final program location* ④ *is depicted in Fig. 5b. It uses the interval domain for both decision and leaf nodes. Note that the constraint* $(\texttt{A} < 3)$ *does not explicitly appear in the code base, but we obtain it in the decision tree representation as a result of applying the functions* $\texttt{IFDEF}_{\mathbb{T}}$ *and* $\texttt{COMPRESS}_{\mathbb{T}}$. *This shows that partitioning of the configuration space* $\mathbb{K}$ *induced by decision trees is semantics-based rather than syntactic-based. That is, linear constraints labelling decision nodes are automatically inferred by the analysis and do not necessarily appear in the code.* □

Example 11 *Let us consider the code base of a program family P' given in Fig. 6a. It contains one numerical feature* `A` *with domain* $[1, 4]$ *and a non-linear feature expression* $\texttt{A}*\texttt{A} < 9$. *At location* ②, $\texttt{FEAT-FILTER}_{\mathbb{T}}(\langle\!|\, \texttt{x} = 0 \,|\!\rangle, \texttt{A}* \texttt{A} < 9)$ *returns an over-approximating tree* $\langle\!|\, \texttt{x} = 0 \,|\!\rangle$, *while* $\texttt{FEAT-FILTER}_{\mathbb{T}}(\langle\!|\, \texttt{x} = 0 \,|\!\rangle, \neg(\texttt{A} * \texttt{A} < 9))$ *returns* $[\![\texttt{A} \geq 3, \langle\!|\, \texttt{x} = 0 \,|\!\rangle, \langle\!|\, \bot_I \,|\!\rangle]\!]$. *In effect, we obtain an over-approximating result at the final program location* ③ *as shown in Fig. 6b. Note that when* $(\texttt{A} \geq 3)$ *the leaf is a join of* $\texttt{ASSIGN}_I(\langle\!|\, \texttt{x} = 0 \,|\!\rangle, \texttt{x := x+1})$ *and*

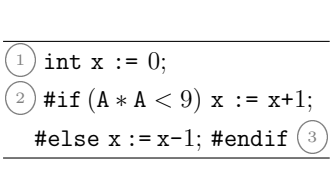


(a) Code base for program family P.

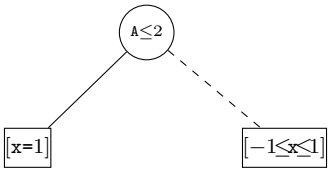


(b) Decision tree representing the result at location ④ of P.

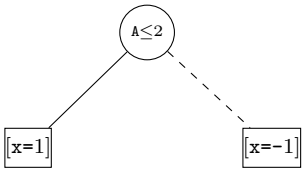
Figure 5: Lifted analysis of an example program family P.



(a) Code base for P'.



(b) Over-approximating decision tree at location ③ of P'.



(c) Precise decision tree at location ③ of P'.

Figure 6: Lifted analysis of an example program family P'.

$\mathit{ASSIGN}_I(\langle\!|\, \mathtt{x} = 0 \,|\!\rangle, \mathtt{x := x-1})$*, while when* $(\mathtt{A} \leq 2)$ *the leaf is* $\mathit{ASSIGN}_I(\langle\!|\, \mathtt{x} = 0 \,|\!\rangle, \mathtt{x := x+1})$*. The precise result at the program location* ③*, which can be obtained in case we have numerical domains that can handle non-linear constraints, is given in Fig. 6c. We observe that when* $\neg(\mathtt{A} \leq 2)$*, i.e. when* $(\mathtt{A} \geq 3)$*, we obtain an over-approximating analysis result* $(-1 \leq \mathtt{x} \leq 1$ *instead of* $\mathtt{x} = -1)$ *due to the over-approximation of the non-linear feature expression* $\mathtt{A} * \mathtt{A} < 9$ *in the numerical domains we use.* □

Example 12 *In Fig. 7 we depict decision trees inferred by performing lifted analysis based on the domain* $\mathbb{T}(\mathbb{C}_I, I)$ *of* SIMPLE *from Section 2, where the ordering of features is* $\mathit{ON} < \mathit{SIZE}$ *and so* $\mathit{ON} <_{\mathbb{C}_I} (\mathit{SIZE} \leq 3)$*. We can see that the number of interval properties needed at locations from* ① *to* ⑦ *is significantly less than 8 properties we need when the product lifted domain is used (see Example 3). In particular, only one interval property is needed at locations* ① *and* ②*, four interval properties are used at location* ⑤*, while three interval properties are needed at all other locations.* □

6. Lifted Analysis based on Binary Decision Diagrams

In this section, we describe another way to analyze numerical program families. First, we describe a syntactic transformation that encodes numerical

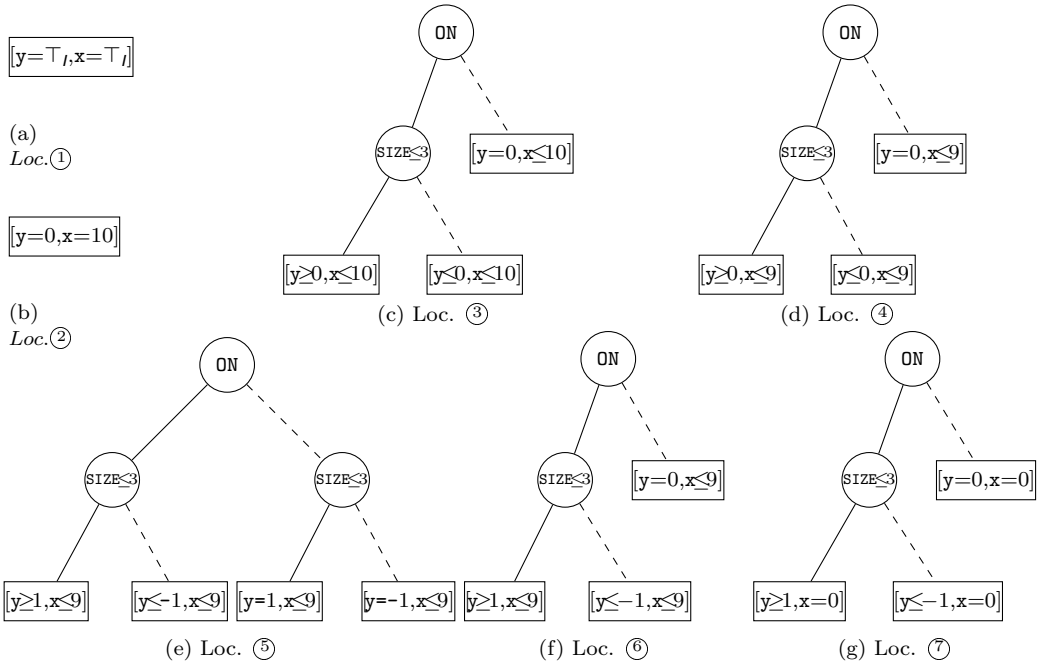


Figure 7: Decision tree invariants at program locations from ① to ⑦ of SIMPLE.

features as a set of Boolean features, that is, it converts a numerical program family into a Boolean program family that contains only Boolean features. Then, we briefly recall the BDD lifted domain introduced by [15] to analyze the transformed Boolean program families.

6.1. Boolean encoding

Our aim is to transform an input program family s with a set of numerical features $\mathbb{F} = \{A_1, \ldots, A_m\}$ into an output program family s' with a set of Boolean features $\mathbb{F}' = \{B_1, \ldots, B_n\}$. The set of configurations $\mathbb{K}'$ in s' includes all possible combinations of feature values (in total $2^{|\mathbb{F}'|}$).

Let $\sigma : \mathit{FeatExp}(\mathbb{F}) \to \mathbb{F}'$ be an *environment function* that maps atomic feature expressions (constraints) over $\mathbb{F}$ into Boolean features from $\mathbb{F}$'. We now define rewrite rules for eliminating atomic constraints of the form $ef \bowtie ef'$ from a numerical program family. The rewrite rules are:

$$ef \bowtie ef' \;\rightsquigarrow\; B, \text{ when } ef \bowtie ef', \neg(ef \bowtie ef') \notin \mathit{dom}(\sigma) \qquad \text{(R-1)}$$

where B is a fresh Boolean feature. The set $\mathbb{F}'$ and the function σ are updated to $\mathbb{F}' \cup \{B\}$ and $\sigma[ef \bowtie ef' \mapsto B]$, respectively. The rule (R-1) states

that, if the current numerical program family being transformed matches the abstract syntax tree node of the shape $ef \bowtie ef'$, and the atomic constraint $ef \bowtie ef'$ or its negation are not in the domain of σ, then replace $ef \bowtie ef'$ by a fresh Boolean feature B. The rules (R-2) and (R-3) handle the cases when atomic constraint $ef \bowtie ef'$ or its negation are already in the domain of σ.

$$ef \bowtie ef' \;\; \leadsto \;\; \sigma(ef \bowtie ef'), \text{ when } ef \bowtie ef' \in dom(\sigma) \qquad \text{(R-2)}$$

$$ef \bowtie ef' \;\; \leadsto \;\; \neg\sigma(ef \bowtie ef'), \text{ when } \neg(ef \bowtie ef') \in dom(\sigma) \qquad \text{(R-3)}$$

We write $\texttt{Rewrite}(s)$ to be the final transformed Boolean program family s' obtained by repeatedly applying rules (R-1)–(R-3) on s and on its transformed versions until we reach a point at which no rule can be applied, i.e. when all atomic constraints $fe \bowtie fe'$ are eliminated. Initially, $\mathbb{F}' = \emptyset$ and $\sigma = []$ are the empty set and environment, respectively. Note that the set of configurations $\mathbb{K}'$ is $2^{\mathbb{F}'}$ for the resulting Boolean program family $\texttt{Rewrite}(s)$.

The following result shows that for any configuration of s, there exists a configuration of $\texttt{Rewrite}(s)$, such that the same variants are derived.

Theorem 13 *Let s be a numerical program family and let $s' = \texttt{Rewrite}(s)$. Let σ be the resulting environment function, s.t. atomic constraints $fe_1 \bowtie fe'_1, \ldots, fe_n \bowtie fe'_n$ correspond to Boolean features $B_1, \ldots, B_n$. For $\forall k \in \mathbb{K}$, $\exists k' \in \mathbb{K}'$, such that $k'(B_i) \iff (k \models fe_i \bowtie fe'_i)$ for all $1 \leq i \leq n$ and $P_k(s) = P_{k'}(s')$.*

Proof *The proof is by structural induction on statements. The only interesting case is $\texttt{\#if}\,(\theta)\; s1\; \texttt{\#endif}$, since in all other cases we have identity transformations. θ is a feature expression in which may occur the atomic constraints $fe_1 \bowtie fe'_1, \ldots, fe_n \bowtie fe'_n$, so it holds that $k \models \theta$ iff $k' \models \theta\,\sigma$. Here, $\theta\,\sigma$ represents a feature expression over Boolean features $B_1, \ldots, B_n$ obtained by replacing atomic constraints from θ according to the environment function σ. Then, $P_k(\texttt{\#if}\,(\theta)\; s1\; \texttt{\#endif}) = P_{k'}(\texttt{Rewrite}(\texttt{\#if}\,(\theta)\; s1\; \texttt{\#endif}))$ follows immediately.*

Example 14 *The numerical program family* SIMPLE *from Section 2 is encoded as the Boolean program family* SIMPLE^{bool} *shown in Fig. 8, where $\sigma(\texttt{SIZE} \leq 3) = \texttt{SIZE3}$. The sets of features and configurations are $\mathbb{F}^{bool} = \{\texttt{ON}, \texttt{SIZE3}\}$ and $\mathbb{K}^{bool} = \{\texttt{ON} \wedge \texttt{SIZE3}, \texttt{ON} \wedge \neg\texttt{SIZE3}, \neg\texttt{ON} \wedge \texttt{SIZE3}, \neg\texttt{ON} \wedge \neg\texttt{SIZE3}\}$. Note that, $P_{\texttt{ON}\wedge(\texttt{SIZE}=1)}(\text{SIMPLE}) = P_{\texttt{ON}\wedge\texttt{SIZE3}}(\text{SIMPLE}^{bool})$ and $P_{\neg\texttt{ON}\wedge(\texttt{SIZE}=4)}(\text{SIMPLE}) = P_{\neg\texttt{ON}\wedge\neg\texttt{SIZE3}}(\text{SIMPLE}^{bool})$.*

```
1  int x := 10, y := 0;
2  while (x ≠ 0) {
3    x := x-1;
4    #if (SIZE3) y := y+1; #else y := y-1; #endif
5    #if (¬ON) y := 0; #else skip; #endif 6}
7  assert (y > 1);
```

Figure 8: The Boolean program family SIMPLE^{bool}.

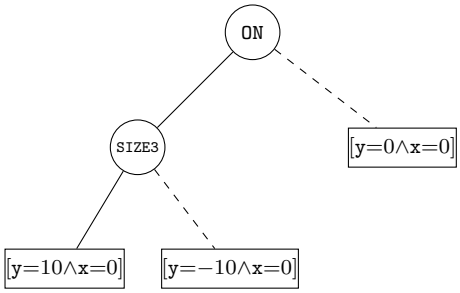


Figure 9: BDD-based (polyhedra) result at location (7) of SIMPLE^{bool}.

Remark *The opposite of the result in Theorem 13 does not hold. That is, there may exists a configuration of* $\texttt{Rewrite}(s)$ *for which no corresponding configuration of* s *exists with the same derived variant.*

Example 15 *Consider the following numerical program family* s*:*

$$\begin{array}{l}\texttt{int } x := 0;\\ \texttt{\#if}\,(A > 5)\ x := x+2;\ \texttt{\#else}\ x := x-2;\ \texttt{\#endif}\\ \texttt{\#if}\,(A > 7)\ x := x+1;\ \texttt{\#else}\ x := x-1;\ \texttt{\#endif}\end{array}$$

It has one numerical feature A *with domain* $[0, 99]$*, and the configuration set is* $\mathbb{K} = \{(A = i) \mid i \in [0, 99]\}$*. The Boolean program family* $\texttt{Rewrite}(s)$ *is:*

$$\begin{array}{l}\texttt{int } x := 0;\\ \texttt{\#if}\,(B_1)\ x := x+2;\ \texttt{\#else}\ x := x-2;\ \texttt{\#endif}\\ \texttt{\#if}\,(B_2)\ x := x+1;\ \texttt{\#else}\ x := x-1;\ \texttt{\#endif}\end{array}$$

It has two Boolean features B_1 *and* B_2*, the configuration set is* $\mathbb{K}' = \{(B_1 \wedge B_2), (B_1 \wedge \neg B_2), (\neg B_1 \wedge B_2), (\neg B_1 \wedge \neg B_2)\}$*, and the environment function is* $\sigma = [(A > 5) \mapsto B_1, (A > 7) \mapsto B_2]$*. For each configuration* $k \in \mathbb{K}$*, there exists* $k' \in \mathbb{K}'$*, s.t.* $P_k(s) = P_{k'}(\texttt{Rewrite}(s))$*. However, for* $k' = (\neg B_1 \wedge B_2) \in \mathbb{K}'$*, there is no* $k \in \mathbb{K}$*, s.t.* $P_k(s) = P_{k'}(\texttt{Rewrite}(s))$*. Note that the derived variant* $P_{\neg B_1 \wedge B_2}(\texttt{Rewrite}(s))$ *is:* $\texttt{int } x := 0; x := x-2; x := x+1$*.* □

6.2. A BDD Lifted Domain

We now show how to analyze a Boolean program family [15]. Let $\mathbb{F}' = \{B_1, \ldots, B_k\}$ be a finite and totaly ordered set of *Boolean features* available in the program family, such that the ordering is $B_1 < \ldots < B_k$. Each configuration $k \in \mathbb{K}'$ can be represented either by a subset of features: $k \subseteq \mathbb{F}'$;

or by a formula: $(k(B_1) \wedge \ldots \wedge k(B_k))$, where $k(B_i) = B_i$ if $B_i \in k$, and $k(B_i) = \neg B_i$ if $B_i \notin k$ for $1 \leq i \leq n$. The set of *feature expressions* $\mathit{FeatExp}(\mathbb{F}')$ is: $\theta ::= \text{true} \mid B \in \mathbb{F}' \mid \neg\theta \mid \theta_1 \wedge \theta_2 \mid \theta_1 \vee \theta_2$.

Lifted Domain. We first consider a simpler form of binary decision diagrams called *binary decision trees* (BDTs). A *binary decision tree* (BDT) $t \in \mathbb{T}(\mathbb{F}', \mathbb{A})$ over the set $\mathbb{F}'$ of Boolean features and the leaf abstract domain $\mathbb{A}$ is either a leaf node $\langle\!|a|\!\rangle$, with a an element of $\mathbb{A}$ and $\mathbb{F}' = \emptyset$, or $[\![B : tl, tr]\!]$, where B 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 $tl, tr \in \mathbb{T}(\mathbb{F}' \backslash \{B\}, \mathbb{A})$. The left and right subtrees are either both leaf nodes or both rooted at decision nodes labeled with the same feature.

However, BDTs contain some redundancy. There are three optimizations we can apply to BDTs in order to reduce their representation [27, 28]:

(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 A_j, remove A_i by sending all its incoming edges to A_j.

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

If we apply reductions (**1**)-(**3**) to a binary decision tree $t \in \mathbb{T}(\mathbb{F}', \mathbb{A})$ until no further reductions are possible, then the result is a reduced *binary decision diagram* $d \in \mathbb{D}(\mathbb{F}', \mathbb{A})$. Thanks to the sharing of information enabled by the reductions (**1**)-(**3**), BDDs are quite compact representation of disjunctive analysis properties. Moreover, if the ordering on the Boolean variables from $\mathbb{F}$ occurring on any path is fixed, the resulting BDDs have a *canonical form.*

Lifted Operations. The operations on $\mathbb{D}(\mathbb{F}', \mathbb{A})$ are implemented by recursive traversal of the operand BDDs and by using hashtables to store and reuse already computed subtrees [27]. The basic operations are:

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

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

- `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`, operations and transfer functions from $\mathbb{A}$ are lifted to $\mathbb{D}(\mathbb{F}', \mathbb{A})$.

The *concretization function* $\gamma_{\mathbb{D}}$ of a binary decision diagram $d \in \mathbb{D}(\mathbb{F}', \mathbb{A})$ returns $\gamma_{\mathbb{A}}(a)$ for $k \in \mathbb{K}'$, where k satisfies the constraints accumulated along the top-down path to the leaf node $a \in \mathbb{A}$. Formally, $\gamma_{\mathbb{D}}(d) = \overline{\gamma}_{\mathbb{D}}[\vee_{k \in \mathbb{K}'} k](d)$, where the function $\overline{\gamma}_{\mathbb{D}}$ is defined recursively as:

$$\begin{array}{l} \overline{\gamma}_{\mathbb{D}}[\theta](\langle\!|a|\!\rangle) = \prod_{k \models \theta} \gamma_{\mathbb{A}}(a), \\ \overline{\gamma}_{\mathbb{D}}[\theta](\llbracket B : tl, tr \rrbracket) = \overline{\gamma}_{\mathbb{D}}[\theta \wedge B](tl) \times \overline{\gamma}_{\mathbb{D}}[\theta \wedge \neg B](tr) \end{array}$$

Given two BDDs $d_1, d_2 \in \mathbb{D}(\mathbb{F}', \mathbb{A})$, their ordering $\sqsubseteq_{\mathbb{D}}$ and join $\sqcup_{\mathbb{D}}$ are:

$$\begin{array}{lcl} d_1 \sqsubseteq_{\mathbb{D}} d_2 & \equiv_{def} & \texttt{apply}_2(\lambda(a_1, a_2).a_1 \sqsubseteq_{\mathbb{A}} a_2, d_1, d_2) \equiv \text{true} \\ d_1 \sqcup_{\mathbb{D}} d_2 & = & \texttt{apply}_2(\lambda(a_1, a_2).a_1 \sqcup_{\mathbb{A}} a_2, d_1, d_2) \end{array}$$

Similarly, we compute the other binary operations. The BDDs $\top_{\mathbb{D}}$ and $\bot_{\mathbb{D}}$ have only one leaf node $\top_{\mathbb{A}}$ and $\bot_{\mathbb{A}}$, respectively.

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

Transfer function $\texttt{FILTER}_{\mathbb{D}}$ for expression tests e is implemented by handling e at each leaf node a of the input BDD d using $\texttt{apply}_1$. That is,

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

Transfer function $\texttt{FEAT-FILTER}_{\mathbb{D}}$ for feature expression tests θ in `#if`-s is implemented using the `meet_condition` operation:

$$\texttt{FEAT-FILTER}_{\mathbb{D}}(d : \mathbb{D}(\mathbb{F}', \mathbb{A}), \theta : \mathit{FeatExp}(\mathbb{F}')) = \texttt{meet\_condition}(d, \theta)$$

Transfer functions $\texttt{ASSIGN}_{\mathbb{D}}$ for assignment `x:=`e are implemented by applying $\texttt{ASSIGN}_{\mathbb{A}}$ leaf-wise using $\texttt{apply}_1$.

$$\texttt{ASSIGN}_{\mathbb{D}}(d : \mathbb{D}(\mathbb{F}', \mathbb{A}), \texttt{x:=}e : \mathit{Stm}) = \texttt{apply}_1(\lambda a.\texttt{ASSIGN}_{\mathbb{A}}(a, \texttt{x:=}e), d)$$

Given the (lifted) transfer function $\overrightarrow{\llbracket s \rrbracket}$ for statement s, $\texttt{IFDEF}_{\mathbb{D}}$ is:

$$\begin{array}{l} \texttt{IFDEF}_{\mathbb{D}}(d : \mathbb{D}(\mathbb{F}', \mathbb{A}), \texttt{\#if } (\theta)\ s\ \texttt{\#endif} : \mathit{Stm}) = \\ \qquad \overrightarrow{\llbracket s \rrbracket}(\texttt{FEAT-FILTER}_{\mathbb{D}}(d, \theta)) \sqcup_{\mathbb{D}} \texttt{FEAT-FILTER}_{\mathbb{D}}(d, \neg\theta) \end{array}$$

Lifted Analysis. In the first iteration, we build a BDD with only one leaf node $\top_{\mathbb{A}}$ for the first program location. Thus, $d_{in} = \texttt{meet\_condition}(\top_{\mathbb{D}}, \vee_{k \in \mathbb{K}'} k)$. Note that, in the case of $\texttt{Rewrite}(s)$, $\mathbb{K}' = 2^{\mathbb{F}'}$ and $\vee_{k \in \mathbb{K}'} k \equiv \text{true}$, so the initial BDD is $d_{in} = \top_{\mathbb{D}}$. The operators and transfer functions of the lifted domain $\mathbb{D}(\mathbb{F}', \mathbb{A})$ are combined together to analyze Boolean program families.

Once the BDD-based lifted analysis infers invariants in all locations of the Boolean program family $\texttt{Rewrite}(s)$, we want to find the corresponding invariants of the numerical program family s. Given a BDD $d \in \mathbb{D}(\mathbb{F}', \mathbb{A})$, we define $\texttt{encode}(d)$ to be a decision tree from $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ obtained by replacing the Boolean features $B \in \mathbb{F}'$ occurring in decision nodes of d with constraints $fe \bowtie fe'$ such that $\sigma(fe \bowtie fe') = B$. The obtained decision tree is then normalized and compressed using $\texttt{COMPRESS}_{\mathbb{T}}$ (see Algorithm 6).s

Let $[\![s]\!]_{\mathbb{T}}$ and $[\![s]\!]_{\mathbb{D}}$ denote transfer functions of statement s in $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ and $\mathbb{D}(\mathbb{F}', \mathbb{A})$, respectively. Note that $t_{in} = \texttt{encode}(d_{in})$.

Theorem 16 $[\![s]\!]_{\mathbb{T}}(\texttt{encode}(d)) = \texttt{encode}\big([\![\texttt{Rewrite}(s)]\!]_{\mathbb{D}}(d)\big)$.

Proof *It follows from the correctness of* `Rewrite` *(see Theorem 13), the definitions of* $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$ *and* $\mathbb{D}(\mathbb{F}', \mathbb{A})$ *domains, and the definition of* `encode`.

Example 17 *If we perform polyhedra lifted analysis of* SIMPLE^{bool} *in Fig. 8 using the lifted domain* $\mathbb{D}(\mathbb{F}', P)$*, where the ordering of features is* `ON` $<$ `SIZE3`*, the obtained BDD at location* (7) *is shown in Fig. 9. By applying the function* `encode` *on it, we obtain the decision tree in Fig. 2b.* □

Example 18 *Reconsider program families* s *and* $\texttt{Rewrite}(s)$ *from Example 15. The polyhedra lifted analysis of* s *using* $\mathbb{T}(\mathbb{C}_P, P)$ *domain and of* $\texttt{Rewrite}(s)$ *using* $\mathbb{D}(\mathbb{F}', P)$ *domain infer invariants given in Figs. 10 and 11 in the final location, respectively. By applying* `encode` *function on the BDD in Fig. 11 we obtain the decision tree in Fig. 10, since the constraint leading to the leaf node* $(\boldsymbol{x} = -1)$*, that is* $(\neg(\boldsymbol{A} > 5) \wedge (\boldsymbol{A} > 7))$*, is unsatisfiable, so it will be eliminated by* $\texttt{COMPRESS}_{\mathbb{T}}$. □

7. Implementation and Evaluation

We evaluate our approaches for lifted analysis phrased in the abstract interpretation framework. It consists of running the proposed tuple-based, tree-based, and BDD-based lifted analyses on a dozen of `#if`-annotated C

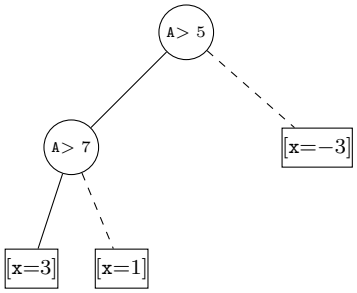


Figure 10: Decision tree-based invariant at final location of s.

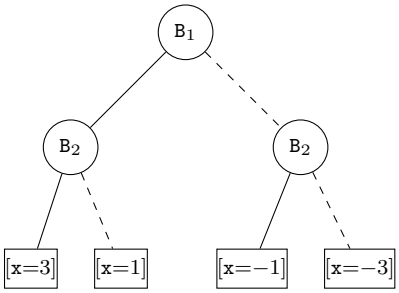


Figure 11: BDD-based invariant at final location of `Rewrite`(s).

case studies. The evaluation aims to show that we can use our lifted analyses with improved representation to efficiently analyze various program families. To do that, we ask the following research questions:

RQ1: How efficient are the new decision tree-based and BDD-based lifted analyses compared to the standard tuple-based lifted analysis?

RQ2: Can the decision tree-based and BDD-based lifted analyses turn some previously infeasible lifted analysis tasks into feasible ones?

RQ3: What are the time and precision performances of our lifted analyses compared to the single-program analysis of variability simulator?

RQ4: Can we find practical application scenarios of using our lifted analyses to efficient verification and program synthesis?

Implementation. We have developed a prototype lifted static analyzer, called SPLNum^{2}Analyzer, which uses lifted domains of tuples $\mathbb{A}^{\mathbb{K}}$, decision trees $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{A})$, and BDDs $\mathbb{D}(\mathbb{F}', \mathbb{A})$. The abstract domain $\mathbb{A}$ for encoding properties of tuple components and leaf nodes as well as the abstract domain $\mathbb{D}$ for encoding linear constraints over numerical features are based on intervals, octagons, and polyhedra numerical domains. Their abstract operations and transfer functions are provided by the APRON library [21]. The operations and transfer functions for BDD lifted domain that combines Boolean formulae and APRON domains are provided by the BDDAPRON library [22]. Our proof-of-concept implementation is written in OCaml and consists of around 7K lines of code. The current front-end of the tool accepts programs written in a (subset of) C with `#if` directives, but without `struct` and `union` types. It currently provides only a limited support for arrays and pointers,

though an extension is possible. The only basic data type is mathematical integers. SPLNum^{2}Analyzer automatically infers numerical invariants in all program locations corresponding to all variants in the given family. 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 that assigns an element from the lifted domain to every program location. It is possible to tune the precision of the analysis by choosing the underlying numerical domain (intervals, octagons, polyhedra).

Experimental setup and Benchmarks. 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. All times are reported as average over five independent executions. The implementation, benchmarks, and all results obtained from our experiments are available from [29]: `https://github.com/aleksdimovski/SPLNUM2Analyzer`. In our experiments, we use two instances of our lifted analyses via tuples: $\overline{\mathcal{A}}_{\Pi}(I)$ and $\overline{\mathcal{A}}_{\Pi}(P)$, via decision trees: $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, and via BDDs: $\overline{\mathcal{A}}_{\mathbb{D}}(I)$ and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$, which use intervals and polyhedra domains as parameters, respectively.

SPLNum^{2}Analyzer was evaluated on a dozen of C numerical programs collected from several categories of the 9th International Competition on Software Verification (SV-COMP 2020) [3] as well as from the real-world BusyBox project [4] . The categories from SV-COMP are: `product lines`, `loops`, `loop-invgen` (`invgen` for short), `loop-lit` (`lit` for short), `termination-crafted` (`craft`), and `termination-restricted` (`restr`). 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 integer variables that our tool can handle, and then we have manually added variability (features and `#if` directives) in each of them. We have generated program families with small and moderate configuration sizes, since they are very common in practice. We have generated presence conditions with different complexities, from only atomic constraints to more complex feature expressions, and `#if` directives are inserted in various locations of the code. In the case of BusyBox, we have first selected some program families with numerical features, and then we have simplified them so that can be handled

[3] `https://sv-comp.sosy-lab.org/2020/`

[4] `https://busybox.net`

by our tool. For example, any reference to a pointer or a library function is replaced with $? = [-\infty, +\infty]$. In the case of the `product lines` category, we have selected several specifications from the e-mail and elevator systems. Tables 1 and 3 presents characteristics of the selected benchmarks in our empirical study. We list: the file name (Benchmark), the category where it is located (folder), number of features ($|\mathbb{F}|$), number of valid configurations ($|\mathbb{K}|$), and number of lines of code (LOC).

Performance Results. We now present the performance results of our empirical study and discuss the implications. Table 1 shows the results of analyzing our benchmark files by using different versions of our lifted static analyses based on tuples, decision trees, and BDDs. For each numerical domain (Intervals and Polyhedra), there are four columns. In the first column, $\overline{\mathcal{A}}_{\Pi}(-)$, we report the running time in seconds to analyze the given benchmark using the lifted analysis based on tuples. In the second (resp., third) column, $\overline{\mathcal{A}}_{\mathbb{T}}(-)$ (resp., $\overline{\mathcal{A}}_{\mathbb{D}}(-)$), we report the speed up factor for the lifted analysis based on decision trees (resp., on BDDs) relative to the corresponding baseline lifted analysis based on tuples, that is, $\overline{\mathcal{A}}_{\mathbb{T}}(-)$ vs. $\overline{\mathcal{A}}_{\Pi}(-)$ (resp., $\overline{\mathcal{A}}_{\mathbb{D}}(-)$ vs. $\overline{\mathcal{A}}_{\Pi}(-)$. In the fourth column, Res, we show whether the results inferred by $\overline{\mathcal{A}}_{\mathbb{D}}(-)$ are identical (denoted by ✓) or approximate (denoted by $\approx$) with respect to the results of $\overline{\mathcal{A}}_{\mathbb{T}}(-)$ and $\overline{\mathcal{A}}_{\Pi}(-)$. The approximation occurs due to the possibility that $\overline{\mathcal{A}}_{\mathbb{D}}(-)$ reports results for variants that are not valid (see Section 6). The performance results confirm that sharing is indeed effective and especially so for large values of $|\mathbb{K}|$. On our benchmarks, it translates to speed ups (i.e., $\overline{\mathcal{A}}_{\mathbb{T}}(-)$ vs. $\overline{\mathcal{A}}_{\Pi}(-)$) that range from 1.5 to 5.2 times when $|\mathbb{K}| < 100$, and from 4.6 to 22 times when $|\mathbb{K}| > 100$ (addresses **RQ1**). We observe even bigger speed ups in case of BDDs (i.e., $\overline{\mathcal{A}}_{\mathbb{D}}(-)$ vs. $\overline{\mathcal{A}}_{\Pi}(-)$) that range from 5.7 to 23 times when $|\mathbb{K}| < 100$, and from 20 to 84 times when $|\mathbb{K}| > 100$ (addresses **RQ1**). Of course, the performance and speed up also depend on how much variability sharing is possible for a given benchmark as well as from the form of feature expressions present in it. In summary, we conclude that the analysis that combines BDDs and intervals, $\overline{\mathcal{A}}_{\mathbb{D}}(I)$, is the fastest version. Moreover, the decision tree-based $\overline{\mathcal{A}}_{\mathbb{T}}(-)$ and the BDD-based lifted analyses $\overline{\mathcal{A}}_{\mathbb{D}}(-)$ outperform the tuple-based lifted analysis $\overline{\mathcal{A}}_{\Pi}(-)$ on all benchmarks. Of course, all versions that use polyhedra are slower than intervals, but they produce more precise invariants. Furthermore, the BDD-based domain $\overline{\mathcal{A}}_{\mathbb{D}}(I)$ may infer approximate results in some cases.

Table 1: Performance results for lifted analyses based on tuples (which are used as baseline) vs. lifted analyses based on decision trees vs. lifted analyses based on Boolean encoding and BDDs performed on selected benchmarks from SV-COMP 2020 and BusyBox. All times are in seconds.

Bench.	folder	$\lvert\mathbb{K}\rvert$	loc	Intervals				Polyhedra			
				$\overline{\mathcal{A}}_{\Pi}$	$\overline{\mathcal{A}}_{\mathbb{T}}$	$\overline{\mathcal{A}}_{\mathbb{D}}$	Res	$\overline{\mathcal{A}}_{\Pi}$	$\overline{\mathcal{A}}_{\mathbb{T}}$	$\overline{\mathcal{A}}_{\mathbb{D}}$	Res
half_2.c	invgen	36	60	0.03	2.1×	12×	≈	0.12	5.1×	23×	≈
heapsort.c	invgen	36	60	0.12	2.5×	19×	✓	0.55	2.3×	17×	✓
seq.c	invgen	125	40	0.51	12×	54×	≈	2.17	15×	72×	≈
eq1.c	loops	36	20	0.07	4.1×	15×	✓	0.24	5.2×	24×	✓
eq2.c	loops	25	20	0.03	1.7×	8.8×	✓	0.10	2.2×	14×	✓
sum01*.c	loops	25	20	0.04	1.8×	8×	✓	0.16	2.4×	9.7×	✓
count_up_do*.c	loops	49	30	0.02	1.9×	7×	≈	0.07	2.4×	17×	≈
hhk2008.c	lit	216	30	0.34	13×	62×	✓	1.18	16×	59×	✓
gsv2008.c	lit	25	25	0.02	1.5×	5.7×	✓	0.10	2.2×	11×	✓
gcnr2008.c	lit	25	30	0.05	2.2×	8.5×	≈	0.32	3.4×	12×	✓
GCD4.c	restr	125	30	0.51	12×	84×	≈	2.10	15×	74×	≈
UpAndDown.c	restr	25	30	0.06	2×	15×	≈	0.23	3.1×	13×	≈
java_Sequen.c	restr	25	25	0.02	1.9×	6.5×	≈	0.07	2.6×	9×	✓
Toulouse*.c	craft	125	75	0.38	7.7×	27×	✓	1.66	11×	54×	✓
Mysore.c	craft	125	35	0.12	4.6×	26×	≈	0.40	6.4×	31×	✓
copyfd.c	BusyBox	16	84	0.07	4.2×	8.5×	✓	0.36	5.2×	9.2×	✓
real_path.c	BusyBox	128	45	3.03	12×	20×	✓	12.5	22×	31×	✓
e-mail_spec6	product	4	2660	42.3	1.2×	10×	✓		infeasible		
e-mail_spec8	product	4	2665	40.3	1.1×	12×	✓		infeasible		
elevator_spec1	product	4	3035	45.9	3×	39×	✓		infeasible		
elevator_spec2	product	8	3055	0.40	7×	20×	✓	6.75	7×	38×	✓

Computational tractability. The tuple-based lifted analysis $\overline{\mathcal{A}}_{\Pi}(-)$ may become very slow or even infeasible for very large configuration spaces $|\mathbb{K}|$. In that case, the decision tree-based $\overline{\mathcal{A}}_{\mathbb{T}}(-)$ and BDD-based $\overline{\mathcal{A}}_{\mathbb{D}}(-)$ lifted analyses can be used to obtain feasible lifted analyses.

In order to confirm the above statement, we have tested the limits of $\overline{\mathcal{A}}_{\Pi}(P)$, $\overline{\mathcal{A}}_{\mathbb{T}}(-)$, and $\overline{\mathcal{A}}_{\mathbb{D}}(-)$ lifted analyses. We took a method, $\texttt{test}_n^k()$, which contains n numerical features $\texttt{A}_1, \ldots, \texttt{A}_n$, such that each numerical feature has domain of k elements, that is, $\mathrm{dom}(\texttt{A}_i) = [0, k{-}1] = \{0, \ldots, k{-}1\}$. The body of $\texttt{test}_n^k()$ consists of n sequentially composed `#if`-s of the form `#if` $(\texttt{A}_i = 0)$ `i := i+1 #else i := 0 #endif`. For example, the method $\texttt{test}_2^3()$ with two features $\texttt{A}_1$ and $\texttt{A}_2$, whose domain is $[0, 2]$, is:

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

Subject to the chosen configuration and the values assigned to the available features, variable `i` in location ④ can have a value in the range from value 2 when $\texttt{A}_1$ and $\texttt{A}_2$ are assigned to 0, to value 0 when $\texttt{A}_2 \geq 1$. The analysis results in program location ④ of $\texttt{test}_2^3()$ obtained using $\overline{\mathcal{A}}_{\Pi}(P)$, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ are shown in Fig. 12, Fig. 13, and Fig. 14, respectively. The tuple-based $\overline{\mathcal{A}}_{\Pi}(P)$ uses tuples with 9 interval properties (components), while the decision tree $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ and BDD-based $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ use only 3 interval properties (leaf nodes) which are shared between all 9 configurations. Notice that for the decision tree representation, the ordering of features is $\texttt{A}_2 < \texttt{A}_1$ and the constraints that represent domains of numerical features (inferred from $\mathbb{K}$) are implicitly included in the tree. For example, the leaf node `i=0` is reachable for variants satisfying $\neg(\texttt{A}_2 = 0) \wedge (0 \leq \texttt{A}_1 \leq 2) \wedge (0 \leq \texttt{A}_2 \leq 2)$. Notice that for the BDD representation, Boolean features $\texttt{B}_1$ and $\texttt{B}_2$ are used for atomic constraints $(\texttt{A}_1 = 0)$ and $(\texttt{A}_2 = 0)$, and the ordering of features is $\texttt{B}_2 < \texttt{B}_1$.

We have generated methods $\texttt{test}_n^k()$ by gradually increasing variability. The method $\texttt{test}_n^k()$ contains n numerical features $\texttt{A}_1, \ldots \texttt{A}_n$, each of them with domain $\{0, \ldots, k{-}1\}$, and it has n sequentially composed `#if`-s guarded by all existing features. In general, the size of tuples used by $\overline{\mathcal{A}}_{\Pi}(P)$ (i.e., the number of interval properties) is k^n, whereas the number of leaf nodes used by $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ (i.e., the number of interval properties) in the final program location is $n + 1$ (so it does not depend on k). The performance results of analyzing $\texttt{test}_n^k$, for different values of n and k, using $\overline{\mathcal{A}}_{\Pi}(P)$, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ are shown in Table 2. We observe that $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$

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

Figure 12: $\overline{\mathcal{A}}_{\Pi}(P)$ result at location ④ of $\texttt{test}_2^3()$.

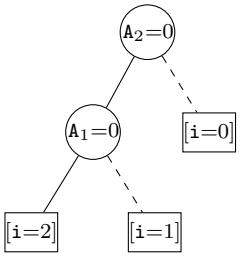


Figure 13: $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ result at location ④ of $\texttt{test}_2^3()$.

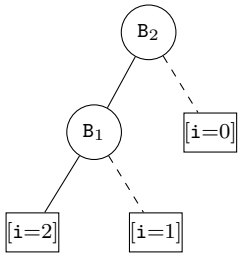


Figure 14: $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ result at location ④ of $\texttt{test}_2^3()$. Feature $\texttt{B}_2$ corresponds to constraint $\texttt{A}_2 = 0$, and $\texttt{B}_1$ corresponds to $\texttt{A}_1 = 0$.

Table 2: The performance results of analyzing $\texttt{test}_n^k$ (`infeasible` = analysis fails to terminate within a timeout limit of 30 minutes). All times are in seconds.

n	$k=3$			$k=5$			$k=7$		
	$\overline{\mathcal{A}}_{\Pi}(P)$	$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	$\overline{\mathcal{A}}_{\mathbb{D}}(P)$	$\overline{\mathcal{A}}_{\Pi}(P)$	$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	$\overline{\mathcal{A}}_{\mathbb{D}}(P)$	$\overline{\mathcal{A}}_{\Pi}(P)$	$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	$\overline{\mathcal{A}}_{\mathbb{D}}(P)$
5	0.212	0.138	0.003	3.538	0.139	0.003	22.37	0.139	0.003
6	0.939	0.301	0.004	26.29	0.301	0.004	infeasible	0.309	0.004
7	3.882	0.610	0.006	infeasible	0.610	0.006	infeasible	0.610	0.006
10	292.2	5.37	0.011	infeasible	5.39	0.011	infeasible	5.47	0.011
11	infeasible	13.81	0.012	infeasible	13.86	0.012	infeasible	13.85	0.012
14	infeasible	323.1	0.028	infeasible	442.2	0.029	infeasible	454.2	0.029

yield trees that provide quite compact and symbolic representation of lifted analysis results. Since the configurations with equivalent analysis results are nicely encoded using decision nodes, the performances of $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ and $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ do not depend on the domain size of features k, but only depend on the number of features n. On the other hand, the performance of $\overline{\mathcal{A}}_{\Pi}(P)$ heavily depends on k. Thus, within a timeout limit of 15 minutes, the analysis $\overline{\mathcal{A}}_{\Pi}(P)$ terminates for $\texttt{test}_{10}^3$, $\texttt{test}_6^5$, and $\texttt{test}_5^7$, whereas it fails to terminate for the corresponding methods with higher number of n.

For $k = 3$ and $n = 10$ (thus, $|\mathbb{K}| = 3^{10}$ =59,049) configurations, the analysis $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ (resp., $\overline{\mathcal{A}}_{\mathbb{D}}(P)$) gives speed up of 54 times (resp., 10^5 times) com-

pared to $\overline{\mathcal{A}}_{\Pi}(P)$, whereas for $k = 3$ and $n = 11$ (thus, $|\mathbb{K}| = 3^{11}$ =177,147), $\overline{\mathcal{A}}_{\Pi}(P)$ crashes with an out-of-memory error, but $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ terminates in less than 13.8 seconds while $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ in 0.01 seconds. For $k = 5$ and $n = 6$ (thus, $|\mathbb{K}| = 5^6$ =15,625) configurations, the analysis $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ (resp., $\overline{\mathcal{A}}_{\mathbb{D}}(P)$) gives speed up of 89 times (resp., 6500 times) compared to $\overline{\mathcal{A}}_{\Pi}(P)$, whereas for $k = 5$ and $n = 7$ (thus, $|\mathbb{K}| = 5^7$ =78,125), $\overline{\mathcal{A}}_{\Pi}(P)$ crashes with an out-of-memory error, but $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ terminates in less than 0.61 seconds while $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ in 0.006 seconds. For $k = 7$ and $n = 5$ (thus, $|\mathbb{K}| = 7^5$ =16,807) configurations, the analysis $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ (resp., $\overline{\mathcal{A}}_{\mathbb{D}}(P)$) gives speed up of 161 times (resp., 7000 times) compared to $\overline{\mathcal{A}}_{\Pi}(P)$, whereas for $k = 7$ and $n = 6$ (thus, $|\mathbb{K}| = 7^6$ =117,649), $\overline{\mathcal{A}}_{\Pi}(P)$ crashes with an out-of-memory error, but $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ terminates in less than 0.3 seconds while $\overline{\mathcal{A}}_{\mathbb{D}}(P)$ in 0.004 seconds. Notice that the BDD approach is particularly well suited for analyzing this example. In summary, we can conclude that decision trees and 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**).

Lifted vs. single-program analysis. The e-mail system has eight features: encryption, decryption, automatic forwarding, e-mail signatures, auto responder, keys, verify, and address book. There are forty valid configurations that can be derived. A variant simulator (single-program) generated with variability encoding from the e-mail system, where all features are considered as program variables, is analyzed using the interval domain $\mathcal{A}(I)$. We use our decision tree lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ to analyze program families where some of the features are considered as real ones. For effectiveness, we consider as real only those features that influence directly the specification. In particular, we consider program families with one and two separate features, and four specifications: `spec6`, `spec8`, `spec11`, and `spec27`. For each specification, many assertions appear in the main function after inlining. To assess the precision of $\mathcal{A}(I)$ and $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, we evaluate the outcomes of the assertions. Let $d \in \mathbb{D}$ be a numerical invariant found before the assertion `assert`(be). An analysis can establish that the assertion is: (1) '*unreachable*', if $d = \bot_{\mathbb{D}}$; (2) '*correct*' (valid), if $d \sqsubseteq_{\mathbb{D}} \text{FILTER}_{\mathbb{D}}(d, be)$, meaning that the assertion is indeed valid regardless of approximations; (3) '*erroneous*' (invalid), if $d \sqsubseteq_{\mathbb{D}} \text{FILTER}_{\mathbb{D}}(d, \neg be)$, meaning that the assertion is indeed invalid; and (4) '*I don't know*', otherwise, meaning that the approximations introduced due to abstraction prevent the analyzer from giving a definite answer. We

say that an assertion is *reachable* if one of the answers (2), (3), or (4) is obtained. In the case of lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, we may obtain *mixed* assertions when different leaves of the resulting decision trees yield different answers.

Table 3 shows the results of analyzing the e-mail system using $\mathcal{A}(I)$ and $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one and two features. In the case of $\mathcal{A}(I)$, we report the number of assertions that are found 'unreachable', denoted by UNR, and reachable ('correct'/'erroneous'/'I don't know'), denoted by REA. In the case of $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, we report the number of 'unreachable' assertions, denoted by UNR, and mixed assertions, denoted by MIX. When a reachable ('correct'/'erroneous'/'I don't know') assertion is reported by $\mathcal{A}(I)$, the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ may give more precise answer by providing the information for which variants that assertion is reachable and for which is unreachable. We denote by $(n : m)$ the fact that one assertion is unreachable in n variants and reachable in m variants. We can see in Table 3 that, for all reachable assertions found by $\mathcal{A}(I)$, we obtain more precise answers using the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(I)$. For example, $\mathcal{A}(I)$ finds 6 unreachable and 26 reachable ('correct') assertions for `spec6`, while $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one feature `Encrypt` (i.e. two variants) finds 16 unreachable assertions and 16 (1:1) mixed assertions such that each mixed assertion is 'unreachable' when `Encrypt`=0 and 'correct' when `Encrypt`=1. By using $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with two features `Encrypt` and `Decrypt` (i.e. four variants), we obtain 16 unreachable assertions and 16 (2:2) mixed assertions. Similar analysis results are obtained for the other specifications. For all specifications, the analysis time increases by considering more features. In particular, we find that $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one feature is in average 1.6 times slower than $\mathcal{A}(I)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with two features is in average 2.5 times slower than $\mathcal{A}(I)$. However, we also obtain more precise information when using $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with respect to the reachability of assertions in various variants (addresses **RQ3** and **RQ4**).

Practical applications. To illustrate the effectiveness of SPLNUM^{2}ANALYZER, we consider some practical applications to program synthesis, and we present the results produced by our analyzer (addresses **RQ4**).

A *sketch* [30] is a partial program with missing integer expressions called *holes* to be discovered by the synthesizer. The integer hole is a placeholder that the program synthesizer must replace with a suitable integer constant. The synthesizer ensures that the resulting code will avoid any assertion fail-

Table 3: Performance results for single analysis $\mathcal{A}(I)$ vs. lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one and two features on the e-mail system from `product lines` category. All times are in seconds.

Benchmark	LOC	$\mathcal{A}(I)$, 0 feat			$\overline{\mathcal{A}}_{\mathbb{T}}(I)$, 1 feat			$\overline{\mathcal{A}}_{\mathbb{T}}(I)$, 2 feat		
		TIME	UNR.	REA.	TIME	UNR.	MIX	TIME	UNR.	MIX
`e-mail_spec6`	2660	22.7	6	26	32.5	16	16(1:1)	42.3	16	16(2:2)
`e-mail_spec8`	2665	19.8	0	32	27.6	0	32(1:1)	40.3	0	32(2:2)
`e-mail_spec11`	2660	20.6	160	96	38.8	160	96(1:1)	60.1	160	96(3:1)
`e-mail_spec27`	2630	19.6	384	128	35.1	384	128(1:1)	59.4	384	128(2:2)

ures under all possible inputs. Consider the following sketch:

```
1  int x := 10, y := 0;
2  while (x > ??) {
3      x := x-1; y := y+1; }
4  assert (y > 2);
```

which contains one integer hole denoted by `??`. The synthesizer should replace the hole `??` with a constant from $\mathbb{Z}$, such that the synthesized program satisfies the given assertion. A sketch can be represented as a program family, such that all possible realisations of holes in the sketch correspond to possible variants in the program family. For example, the above sketch can be encoded as a program family that contains one numerical feature `A` with domain $[0, max]$ [5] by replacing the hole `??` with feature `A`. Note that the expression $\mathtt{x} > \mathtt{A}$ is abbreviation for:

$$\texttt{\#if}\ (\texttt{A=}0)\ (\texttt{x}>0)\ \texttt{\#elif}\ \ldots\ \texttt{\#elif}\ (\texttt{A=}max)\ (\texttt{x}>max)\ \texttt{\#endif}\ \ldots\ \texttt{\#endif}$$

However, instead of performing the above syntactic transformation, we can extend our tool so that it can directly handle statements and expressions that only read the values of numerical features.

By performing lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ of the resulting program family, at location (4) we obtain the decision tree $[\![\mathtt{A} \leq 9 : \langle\!|\mathtt{A+y=10} \wedge \mathtt{A=x}|\!\rangle, \langle\!|\mathtt{y=0} \wedge \mathtt{x=10}|\!\rangle]\!]$. We can check that the assertion $(\mathtt{y} > 2)$ is valid only for $(0 \leq \mathtt{A} \leq 7)$. Hence,

[5] Note that max represents some maximal representable integer.

we can correctly replace the hole `??` in the original sketch with an integer from [0,7], so that the assertion will always be valid. This way, we have shown that our lifted analyzer can be used to synthesize the hole `??` in the above sketch (addresses **RQ4**). The decision tree-based representation is the most suitable for resolving program sketches due to the fact that linear constraints labelling decision nodes are automatically inferred during the analysis. Since our lifted analyzer is based on numerical domains $\mathbb{D}$, it can be used mainly for synthesizing numerical programs. The existing sketching approach [30], which uses SAT-based inductive synthesis, is more general, though it is mainly used for synthesizing bit-manipulating programs. For more detailed explanation, we refer to [31].

Discussion. Our experiments demonstrate that the decision tree-based $\overline{\mathcal{A}}_{\mathbb{T}}(-)$ and the BDD-based lifted analyses $\overline{\mathcal{A}}_{\mathbb{D}}(-)$ outperform the tuple-based lifted analysis $\overline{\mathcal{A}}_{\Pi}(-)$. Moreover, the BDD-based $\overline{\mathcal{A}}_{\mathbb{D}}(-)$ is often faster than the decision tree-based $\overline{\mathcal{A}}_{\mathbb{T}}(-)$. However, there are examples when $\overline{\mathcal{A}}_{\mathbb{T}}(-)$ is faster. Recall that the partitioning of the configuration space induced by our decision trees is semantics-based rather than syntactic-based. For instance, if we consider Example 10, our decision tree-based approach constructs a decision tree with only one decision node such that the linear constraint in that node is automatically inferred by the analysis and does not appear syntactically in the code. On the other hand, if we use Boolean encoding, we would need three Boolean features and a more complex BDD, thus resulting in a slower BDD-based analysis. Furthermore, note that the BDD domain (implemented via the BDDAPRON library [22]) is very optimized and efficiently implemented, whereas the decision tree domain is still a prototype. Many abstract operations and transfer functions of the decision tree domain can be further optimized, thus making its performance to improve. Finally, we have seen that the decision tree domain can be successfully applied to resolving program sketches, whereas the BDD domain is not suitable for this application. Moreover, the BDD-based analysis infers approximate results in some cases. This way, we conclude that both newly proposed lifted domains (decision trees and BDDs) are useful in practice for different application scenarios.

We have seen that there is a whole spectrum of possible techniques for analyzing program families. At one end of the spectrum are lifted analyses, which are fully-disjunctive with respect to features variables. They are based on single-program domains with full partitioning of the configuration space $\mathbb{K}$. At the other end of the spectrum is a single-program analysis of variability

simulators. It represents a non-disjunctive analysis with no partitioning of the configuration space $\mathbb{K}$. We can also construct a range of hybrid techniques between fully-disjunctive lifted analysis and non-disjunctive single-program analysis. They represent disjunctive analyses only with respect to some (but not all) feature variables with partitioning of some subset of $\mathbb{K}$.

Threats to validity. Our current tool supports a non-trivial subset of C, and the missing constructs (e.g. pointers, `struct` and `union` types) are largely orthogonal to the solution (lifted domains). In particular, these features complicate the abstract 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 lifted domains we introduce in this work. Therefore, supporting these constructs would not provide any new insights to our evaluation.

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 (from 1 to 3 features, and from 16 to 216 configurations), 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 (more than 4 features and thousand configurations), the tuple-based lifted analysis is very likely to become infeasible, as demonstrated by the simple $\texttt{test}_n^k$ method. We have also inserted presence conditions with different complexities, from atomic to more complex feature expressions, and `#if`-s are placed in different locations of the code (e.g., initialization part, inside loops, etc.). However, the experiments are also performed on more realistic program families from BusyBox project and `product-line` category of SV-COMP.

8. Related Work

Decision-tree abstract domains have been used in the abstract interpretation community recently [32, 33, 34, 16]. Decision trees have been applied for the disjunctive refinement of interval (boxes) domain [32]. That is, each element of the new domain is a propositional formula over interval linear constraints. Segmented decision tree abstract domains has also been defined in the literature [33, 34] to enable path dependent static analysis. Their elements contain decision nodes that are determined either by values of program variables [33] or by the branch (`if`) conditions [34], whereas the leaf

nodes are numerical properties. Urban and Mine [16, 17] 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. The decision tree lifted domain proposed here extends this termination domain by adapting it in the context of product-lines analysis. Logico-numerical abstract domains implemented using BDDAPRON and specifically designed acceleration methods are used in [35] to verify synchronous data-flow programs with Boolean and numerical variables, such as LUSTRE programs. Both termination and logico-numerical domains can be considered as single-program domains that use partitioning techniques with respect to the total memory space and the Boolean state subspace, respectively. Similarly, the lifted domains proposed here can be considered as single-program domains that use partitioning with respect to the feature variables in order to partition the configuration space $\mathbb{K}$. The APRON library has been developed by Jeannet and Mine [21] to support the application of numerical abstract domains in static analysis, while the BDDAPRON library has been developed by Jeannet [22] to implement a relational inter-procedural analysis of concurrent programs. The ELINA library [36] represents an another efficient implementation of numerical abstract domains.

There are two styles of static analysis: *a dataflow analysis from the monotone framework* developed by Kildall [37] that is algorithmically defined on syntactic CFGs, and *an abstract interpretation-based static analysis* developed by Cousot and Cousot [6] that is more general and semantically defined. A central limitation of the monotone framework approach is the requirement that domains have finite height, which is overcame by the abstract interpretation approach using widening and narrowing techniques [6, 26]. Brabrand et al. [9] 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 based on using variational data structures [10] (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$^{\text{LIFT}}$ [14] is an implementation of the lifted dataflow analysis formulated within the

IFDS framework. 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 analyses considered here, are not distributive and cannot be encoded in IFDS.

Midtgaard et al. [11] 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*. The method uses the calculational approach to abstract interpretation of Cousot [38] in order to derive a lifted analysis which is correct by construction. This approach improves over the "brute force" strategy since a lot of caching effects and improvements are possible for it. For example, it compiles and executes the fixed point iterative algorithm once per whole family, configuration satisfiability tests $k \models \theta$ can be memoized, many transfer functions that act identically for all configurations can be executed more efficiently, and some simple forms of sharing analysis equivalent information can be implemented via bit vectors or formulae. Moreover, so-called variability abstractions [39, 40] can be applied as another way to speed up lifted analysis by deriving abstract lifted analysis. They tame the combinatorial explosion of the number of configurations and reduce it to something more tractable by manipulating the configuration space. A more efficient implementation of lifted static analysis by abstract interpretation is obtained by improving representation via sharing by using binary decision diagram (BDD) domains [15, 41]. BDDs consist of decision nodes that are labelled with Boolean features, and leaf nodes that belong to an existing single-program analysis domain. However, the above lifted static analyses based on abstract interpretation are applied to classical program families with only Boolean features. On the other hand, in this work we consider `#if`-enriched C program families with both Boolean and numerical features, which represent the majority of industrial embedded code. Other applications of decision tree lifted domain are for resolving program sketches [31] and for analyzing dynamic program families [42].

Several *other successful lifted techniques* for analyzing classical SPLs that contain only Boolean features are based on sharing through binary decision diagrams (BDDs). SPLverifier [12, 13] performs software model checking of program families based on variability encoding (to transform compile-time to run-time variability) and BDDs (to represent variability information in states). VarexJ [43] performs dynamic analysis of program families based on

variability-aware execution, while SuperC [44] is a variability-aware parser, which can parse languages with preprocessor annotations thus producing ASTs with variability nodes. Both use BDDs to represent feature expressions.

Compared to classical SPLs with only Boolean features, SPLs with numerical features have larger and more complex configuration spaces due to increased variability. One of the earliest attempts *to analyze SPLs with non-Boolean features* was in the context of lifted model checking by Cordy et al. [25]. They generalize the classical lifted model checking algorithms [45, 46, 47] to support numerical features and multi-features. To handle arithmetic constraints they use SMT solvers. However, that approach [25] works 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.

9. Conclusion

In this work, we employ decision trees, BDDs, and widely-known numerical abstract domains for automatic inference of invariants in all program locations of C program families that contain both Boolean and numerical features. In this way, we obtain several lifted domains for handling numerical features. Using experimental evidence, we have shown that our lifted analyses are effective and perform well on a wide variety of benchmarks. BDD-based lifted analysis has the best time performance. Decision tree-based analysis outperforms the tuple-based lifted analysis and has some interesting practical applications, like in program sketching.

In the future, we would like to extend the lifted domain to also support non-linear constraints, such as congruences and non-linear functions (e.g. polynomials, exponentials) [48], and to handle more complex heap-manipulating program families [49]. We also want to try other libraries that support numerical abstract domains, such as ELINA [36], and estimate their performance in the context of lifted analysis. We can also define a backward lifted analysis in combination with a preliminary forward lifted analysis to infer the necessary preconditions in order a given assertion to be satisfied or violated. The obtained preconditions in the form of linear constraints can be analyzed using model counting techniques to quantify how likely is an input or a variant to satisfy them [50, 51].

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] C. Henard, M. Papadakis, M. Harman, Y. L. Traon, Combining multi-objective search and constraint solving for configuring large software product lines, in: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1, IEEE Computer Society, 2015, pp. 517–528. doi:10.1109/ICSE.2015.69.
URL https://doi.org/10.1109/ICSE.2015.69

[4] D. Munoz, J. Oh, M. Pinto, L. Fuentes, D. S. Batory, Uniform random sampling product configurations of feature models that have numerical features, in: Proceedings of the 23rd International Systems and Software Product Line Conference, SPLC 2019, Volume A, ACM, 2019, pp. 39:1–39:13. doi:10.1145/3336294.3336297.
URL https://doi.org/10.1145/3336294.3336297

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

[6] 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 Symposium on Principles of Programming Languages, ACM, 1977, pp. 238–252. doi:10.1145/512950.512973.
URL http://doi.acm.org/10.1145/512950.512973

[7] 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

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival, Why does astrée scale up?, Formal Methods in System Design 35 (3) (2009)

229–264. doi:10.1007/s10703-009-0089-6.
URL https://doi.org/10.1007/s10703-009-0089-6

[9] 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.

[10] 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

[11] 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

[12] S. Apel, H. Speidel, P. Wendler, A. von Rhein, D. Beyer, Detection of feature interactions using feature-aware verification, in: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 372–375. doi:10.1109/ASE.2011.6100075.
URL http://dx.doi.org/10.1109/ASE.2011.6100075

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

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

[15] 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

[16] 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

[17] C. Urban, Static analysis by abstract interpretation of functional temporal properties of programs, Ph.D. thesis, École Normale Supérieure, Paris, France (2015).
URL https://tel.archives-ouvertes.fr/tel-01176641

[18] C. Urban, Function: An abstract domain functor for termination - (competition contribution), in: Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015. Proceedings, Vol. 9035 of LNCS, Springer, 2015, pp. 464–466. doi:10.1007/978-3-662-46681-0_46.
URL https://doi.org/10.1007/978-3-662-46681-0_46

[19] 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

[20] 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

[21] B. Jeannet, A. Miné, Apron: A library of numerical abstract domains for static analysis, in: Computer Aided Verification, 21st International Conference, CAV 2009. Proceedings, 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

[22] 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

[23] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, S. Apel, Variability encoding: From compile-time to load-time variability, J. Log. Algebraic Methods Program. 85 (1) (2016) 125–145. doi:10.1016/j.jlamp.2015.06.

007.
URL https://doi.org/10.1016/j.jlamp.2015.06.007

[24] 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.
URL https://arxiv.org/abs/2012.05863

[25] M. Cordy, P. Schobbens, P. Heymans, A. Legay, Beyond boolean product-line model checking: dealing with feature attributes and multi-features, in: 35th International Conference on Software Engineering, ICSE '13, IEEE Computer Society, 2013, pp. 472–481. doi:10.1109/ICSE.2013.6606593.
URL https://doi.org/10.1109/ICSE.2013.6606593

[26] P. Cousot, R. Cousot, Comparing the galois connection and widening/-narrowing approaches to abstract interpretation, in: Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Proceedings, Vol. 631 of LNCS, Springer, 1992, pp. 269–295. doi:10.1007/3-540-55844-6_142.
URL https://doi.org/10.1007/3-540-55844-6_142

[27] 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

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

[29] A. S. Dimovski, S. Apel, A. Legay, Tool artifact "SPLNum^{2}Analyzer", Zenodo (2021). doi:10.5281/zenodo.4796015.
URL https://zenodo.org/record/4796015#.YK1pAqgzbIU

[30] A. Solar-Lezama, Program sketching, STTT 15 (5-6) (2013) 475–495. doi:10.1007/s10009-012-0249-7.
URL https://doi.org/10.1007/s10009-012-0249-7

[31] A. S. Dimovski, S. Apel, A. Legay, Program sketching using lifted analysis for numerical program families, in: NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings, Vol. 12673 of LNCS, Springer, 2021, pp. 95–112. doi:10.1007/978-3-030-76384-8_7.
URL https://doi.org/10.1007/978-3-030-76384-8_7

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

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

[38] P. Cousot, The calculational design of a generic abstract interpreter, in: M. Broy, R. Steinbrüggen (Eds.), Calculational System Design, NATO ASI Series F. IOS Press, Amsterdam, 1999, pp. 1–88.

[39] A. S. Dimovski, C. Brabrand, A. Wasowski, Variability abstractions: Trading precision for speed in family-based analyses, in: 29th European Conference 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

[40] 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

[41] A. S. Dimovski, A binary decision diagram lifted domain for analyzing program families, Journal of Computer Languages 63 (2021) 101032. doi:https://doi.org/10.1016/j.cola.2021.101032.
URL https://www.sciencedirect.com/science/article/pii/S2590118421000113

[42] A. S. Dimovski, S. Apel, Lifted static analysis of dynamic program families by abstract interpretation, in: 35th European Conference on Object-Oriented Programming, ECOOP 2021, Vol. 194 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2021, pp. 14:1–14:28.
URL http://dx.doi.org/10.4230/LIPIcs.ECOOP.2021.xxx

[43] 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 International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016, ACM, 2016, pp. 483–494. doi:10.1145/2970276.2970322.
URL http://doi.acm.org/10.1145/2970276.2970322

[44] P. Gazzillo, R. Grimm, Superc: parsing all of C by taming the preprocessor, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, 2012, pp. 323–334. doi:10.1145/2254064.2254103.
URL http://doi.acm.org/10.1145/2254064.2254103

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

[46] A. S. Dimovski, A. Legay, A. Wasowski, Generalized abstraction-refinement for game-based CTL lifted model checking, Theor. Comput. Sci. 837 (2020) 181–206. doi:10.1016/j.tcs.2020.06.011.
URL https://doi.org/10.1016/j.tcs.2020.06.011

[47] 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

[48] A. R. Bradley, Z. Manna, H. B. Sipma, The polyranking principle, in: Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Proceedings, Vol. 3580 of LNCS, Springer, 2005, pp. 1349–1361. doi:10.1007/11523468_109.
URL https://doi.org/10.1007/11523468_109

[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] 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

[51] 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.