Lifted Static Analysis of Dynamic Program Families by Abstract Interpretation

Aleksandar S. Dimovski
Mother Teresa University, st. Mirche Acev nr. 4, 1000 Skopje, North Macedonia

Sven Apel
Saarland University, Saarland Informatics Campus E1.1, 66123 Saarbrücken, Germany

Abstract

Program families (software product lines) are increasingly adopted by industry for building families of related software systems. A program family offers a set of *features* (configured options) to control the presence and absence of software functionality. Features in program families are often assigned at compile-time, so their values can only be *read* at run-time. However, today many program families and application domains demand run-time adaptation, reconfiguration, and post-deployment tuning. Dynamic program families (dynamic software product lines) have emerged as an attempt to handle variability at run-time. Features in dynamic program families can be controlled by ordinary program variables, so *reads* and *writes* to them may happen at run-time.

Recently, a decision tree lifted domain for analyzing traditional program families with numerical features has been proposed, in which decision nodes contain linear constraints defined over numerical features and leaf nodes contain analysis properties defined over program variables. Decision nodes partition the configuration space of possible feature values, while leaf nodes provide analysis information corresponding to each partition of the configuration space. As features are statically assigned at compile-time, decision nodes can be added, modified, and deleted only when analyzing read accesses of features. In this work, we extend the decision tree lifted domain so that it can be used to efficiently analyze dynamic program families with numerical features. Since features can now be changed at run-time, decision nodes can be modified when handling read and write accesses of feature variables. For this purpose, we define extended transfer functions for assignments and tests as well as a special widening operator to ensure termination of the lifted analysis. To illustrate the potential of this approach, we have implemented a lifted static analyzer, called DSPLNum^{2}Analyzer, for inferring numerical invariants of dynamic program families written in C. An empirical evaluation on benchmarks from SV-COMP indicates that our tool is effective and provides a flexible way of adjusting the precision/cost ratio in static analysis of dynamic program families.

2012 ACM Subject Classification Software and its engineering → Software functional properties; Software and its engineering → Software creation and management; Theory of computation → Logic

Keywords and phrases Dynamic program families, Static analysis, Abstract interpretation, Decision tree lifted domain

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2021.14

1 Introduction

A *program family* (software product line) is a set of similar programs, called *variants*, that is built from a common code base [39]. The variants of a program family can be distinguished in terms of *features*, which describe the commonalities and variability between the variants. Program families are commonly seen in the development of commercial embedded and critical system domains, such as cars, phones, avionics, medicine, robotics, etc. [1]. There are several techniques for implementing program families. Often traditional program families [11] support static feature binding and require to know the values of features at compile-time. For example, `#if` directives from the C preprocessor `CPP` represent the most common implementation mechanism in practice [34]. At compile-time, a variant is derived by assigning

35th European Conference on Object-Oriented Programming (ECOOP 2021).
Editors: Manu Sridharan and Anders Møller; Article No. 14; pp. 14:1–14:28
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

concrete values to a set of features relevant for it, and only then is this variant compiled or interpreted. However, in an increasingly dynamic world, the increasing need for adaptive software demands highly configurable and adaptive variability mechanisms, many of them managed at run-time. Recent development approaches such as dynamic program families (dynamic software product lines) [29, 28, 41, 7] support dynamic feature binding, and so features can be assigned at run-time. This provides high flexibility to tailor a variant with respect to available resources and user preferences on demand. Dynamic binding is often necessary in long-running systems that cannot be stopped but have to adapt to changing requirements [27]. For example, for a mobile device, we can decide at run-time which values of features are actually required according to the location of the device. Hence, a dynamic program family adapts to dynamically changing requirements by reconfiguring itself, which may result in an infinite configuration process [10].

In this paper, we devise an approach to perform static analysis by abstract interpretation of dynamic program families. *Abstract interpretation* [12, 38] is a powerful framework for approximating the semantics of programs. It provides static analysis techniques that analyze the program's source code directly and without intervention at some level of abstraction. The obtained static analyses are sound (all reported correct programs are indeed correct) and efficient (with a good trade-off between precision and cost). However, static analysis of program families is harder than static analysis of single programs, because the number of possible variants can be very large (often huge) in practice. Recently, researchers have addressed this problem by designing aggregate *lifted* (family-based) *static analyses* [5, 36, 47], which analyze all variants of the family simultaneously in a single run. These techniques take as input the common code base, which encodes all variants of a program family, and produce precise analysis results for all variants. Lifted static analysis by abstract interpretation of traditional (static) program families with numerical features has been introduced recently [21]. The elements of the lifted abstract domain are *decision trees*, in which the decision nodes are labelled with linear constraints over numerical features, whereas the 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 properties 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. Since features are statically bound at compile-time and only appear in presence conditions of `#if` directives, new decision nodes can only be added by feature-based presence conditions (at `#if` directives), and existing decision nodes can be removed when merging the corresponding control flows again. The fundamental limitation of this decision tree lifted domain [21] (as well as other lifted domains [4, 36, 47]) is that it cannot handle dynamically bound features that can be changed at run-time.

To improve over the state-of-the-art, we devise a novel decision tree lifted domain for analyzing dynamic program families with numerical features. Since features can now be dynamically reconfigured and bound at run-time, linear constraints over features that occur in decision nodes can be dynamically changed during the analysis. This requires extended transfer functions for assignments and tests that can freely modify decision nodes and leafs. Moreover, we need a special widening operator applied on linear constraints in decision nodes as well as on analysis properties in leaf nodes to ensure that we obtain finite decision trees. This way, we minimize the cost of the lifted analysis and ensure its termination.

The resulting decision tree lifted domain is parametric in the choice of the numerical domain that underlies the linear constraints over numerical features labelling decision nodes, and the choice of the single-program analysis domain for leaf nodes. In our implementation, we also use numerical domains for leaf nodes, which encode linear constraints over both program and feature variables. We use well-known numerical domains, including intervals [12], octagons [37], polyhedra [16], from the APRON library [33], to obtain a concrete decision tree-based implementation of the lifted abstract domain. To demonstrate the feasibility of our approach, we have implemented a *lifted analysis* of dynamic program families written in C for the automatic inference of numerical invariants. Our tool, called DSPLNUM^{2}ANALYZER[1], computes a set of possible numerical invariants, which represent linear constraints over program and feature variables. We can use the implemented lifted static analyzer to check invariance properties of dynamic program families in C, such as assertions, buffer overflows, null pointer references, division by zero, etc. [14].

Since features behave as ordinary program variables in dynamic program families, they can be also analyzed using off-the-shelf single-program analyzers. For example, we can use numerical abstract domains from the APRON library [33] for analyzing dynamic program families. However, these domains infer a conjunction of linear constraints over variables to record the information of all possible values of variables and relationships between them. The absence of disjunctions may result in rough approximations and very weak analysis results, which may lead to imprecisions and the failure of showing the required program properties. The decision tree lifted domain proposed here overcomes these limitations of standard single-program analysis domains by adding weak forms of disjunctions arising from feature-based program constructs. The elements of the decision tree lifted domain partition the space of possible values of features inducing disjunctions into the leaf domain.

In summary, we make several contributions:

- We propose a new parameterized decision tree lifted domain suited for handling program families with dynamically bound features.
- We develop a lifted static analyzer, DSPLNUM^{2}ANALYZER, in which the lifted domain is instantiated to numerical domains from the APRON library.
- We evaluate our approach for lifted static analysis of dynamic program families written in C. We compare (precision and time) performances of our decision tree-based approach with the single-program analysis approach; and we show their concrete application in assertion checking. Our lifted analysis provides an acceptable precision/cost tradeoff: we obtain invariants with a higher degree of precision within a reasonable amount of time than when using single-program analysis.

2 Motivating Example

We now illustrate the decision tree lifted domain through several motivating examples. The code base of the program family SFAMILY is given in Fig. 1. SFAMILY contains one numerical feature `A` whose domain is $[0, 99] = \{0, 1, \ldots, 99\}$. Thus, there are a hundred valid configurations $\mathbb{K} = \{(\mathtt{A}=0), (\mathtt{A}=1), \ldots, (\mathtt{A}=99)\}$. The code of SFAMILY contains one `#if` directive that changes the current value of program variable `y` depending on how feature `A` is set at compile-time. For each configuration from $\mathbb{K}$, a variant (single program) can be generated by appropriately resolving the `#if` directive. For example, the variant corresponding to configuration $(\mathtt{A}=0)$ will have the assignment `y := y+1` included in location ③, whereas the variant corresponding to configuration $(\mathtt{A}=10)$ will have the assignment `y := y-1` included in location ③.

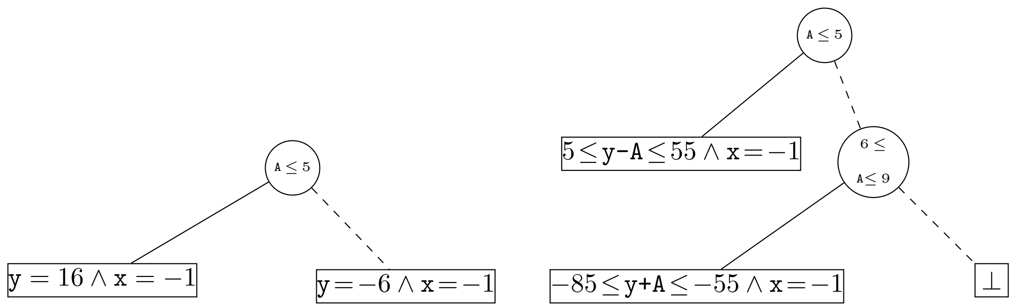
[1] NUM^2 in the name of the tool refers to its ability to both handle NUMerical features and to perform NUMerical client analysis of dynamic program families (DSPLs).

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

Figure 1 Program family sFAMILY.

```
① int x := 10, y := 5;
② A := [0,9];
③ while (x ≥ 0) {
④     if (A ≤ 5) then y := y+A;
⑤                else y := y-A;
⑥     x := x-1; }
⑦ if (A ≤ 5) then assert (y ≥ 5);
⑧            else assert (y ≤ −60);
```

Figure 2 Dynamic program family dFAMILY.



(a) sFAMILY. **(b)** dFAMILY.

Figure 3 Inferred decision trees at final program locations (solid edges = true, dashed edges = false).

Assume that we want to perform *lifted polyhedra analysis* of sFAMILY using the decision tree lifted domain introduced in [21]. The decision tree inferred at the final location of sFAMILY is shown in Fig. 3a. Notice that inner decision nodes (resp., leaves) of the decision tree in Fig. 3a are labeled with *Polyhedra* linear constraints over feature $\mathtt{A}$ (resp., over program variables $\mathtt{x}$ and $\mathtt{y}$). 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). We observe that decision trees offer good possibilities for sharing and interaction between analysis properties corresponding to different configurations, and so they provide compact representation of lifted analysis elements. For example, the decision tree in Fig. 3a shows that when $(\mathtt{A} \leq 5)$ the shared property in the final location is $(\mathtt{y} = 16, \mathtt{x} = -1)$, whereas when $(\mathtt{A} > 5)$ the shared property is $(\mathtt{y} = -6, \mathtt{x} = -1)$. Hence, the decision tree-based approach uses only two leaves (program properties), whereas the brute force enumeration approach that analyzes all variants one by one will use a hundred program properties. This ability for sharing is the key motivation behind the usage of decision trees in lifted analysis.

Consider the code base of the dynamic program family dFAMILY in Fig. 2. Similarly to sFAMILY, dFAMILY contains one feature $\mathtt{A}$ with domain $[0, 99]$. However, feature $\mathtt{A}$ in sFAMILY can only be read and occurs only in presence conditions of `#if`-s. In contrast, feature $\mathtt{A}$ in dFAMILY can also be assigned and occurs freely in the code as any other program variable (see locations ②, ④, ⑤, and ⑦). To perform *lifted polyhedra analysis* of dFAMILY, we need to extend the decision tree lifted domain for traditional program families [21], so that it takes into account the new possibilities of features in dynamic program families. The decision tree inferred in program location ⑦ of dFAMILY is depicted in

Fig. 3b. It can be written as the following disjunctive property in first order logic:

$$\big(0 \leq \mathtt{A} \leq 5 \wedge 5 \leq \mathtt{y}\text{-}\mathtt{A} \leq 55 \wedge \mathtt{x} = -1\big) \vee \big(6 \leq \mathtt{A} \leq 9 \wedge -85 \leq \mathtt{y}\text{+}\mathtt{A} \leq -55 \wedge \mathtt{x} = -1\big) \vee \big(9 < \mathtt{A} \leq 99 \wedge \bot\big)$$

This invariant successfully confirms the validity of the given assertion. Note that, the leaf node $\bot$ abstracts only the empty set of (concrete) program states and so it describes unreachable program locations. Hence, $\bot$ in Fig. 3b means that the assertion at location ⑦ is unreachable when $(\mathtt{A} > 9)$. Also, 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 features. For example, the decision node $(\mathtt{A} \leq 5)$ is satisfied when $(\mathtt{A} \leq 5) \wedge (0 \leq \mathtt{A} \leq 99)$, whereas its negation is satisfied when $(\mathtt{A} > 5) \wedge (0 \leq \mathtt{A} \leq 99)$. The constraint $(0 \leq \mathtt{A} \leq 99)$ represents the domain of $\mathtt{A}$.

Alternatively, dynamic program family DFAMILY can be analyzed using the off-the-shelf (single-program) APRON polyhedra domain [33], such that feature $\mathtt{A}$ is considered as an ordinary program variable. In this case, we obtain the invariant: $\mathtt{A}\text{+}\mathtt{y} \leq 66 \wedge \mathtt{A}\text{-}\mathtt{y} \geq -54$ at location ⑦. However, this invariant is not strong enough to establish the validity of the given assertion. This is because the different partitions of the set of valid configurations have different behaviours and this single-program domain do not consider them separately. Therefore, this domain is less precise than the decision tree lifted domain that takes those differences into account.

3 A Language for Dynamic Program Families

Let $\mathbb{F} = \{A_1, \dots, A_n\}$ be a finite and totaly ordered set of *numerical features* available in a dynamic 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. An assignment of values to all features represents a *configuration* k, which specifies one *variant* of a program family. It is given as a *valuation function* $k : \mathbb{K} = \mathbb{F} \to \mathbb{Z}$, 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*. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathbb{K}$ can be represented by a formula: $(A_1 = k(A_1)) \wedge \ldots \wedge (A_n = k(A_n))$. Given a Boolean feature $B \in \mathbb{F}$, we often abbreviate $(B = 1)$ with formula B and $(B = 0)$ with formula $\neg B$. The set of valid configurations $\mathbb{K}$ can be also represented as a formula: $\vee_{k \in \mathbb{K}} k$.

We consider a simple sequential non-deterministic programming language, which will be used to exemplify our work. The program variables *Var* are statically allocated and the only data type is the set $\mathbb{Z}$ of mathematical integers. To introduce dynamic variability into the language, apart from reading the current values of features, it is possible to write into features. The new statement "$\mathtt{A}\texttt{:=}ae$" has a possibility to update the current configuration (variant) $k \in \mathbb{K}$ by assigning a new arithmetic expression ae to feature A. This is known as *run-time reconfiguration* [7]. We write $k[A \mapsto n]$ for the updated configuration that is identical to k but feature A is mapped to value n. The syntax of the language is:

$$\begin{aligned}
s &::= \mathtt{skip} \mid \mathtt{x}\texttt{:=}ae \mid s;s \mid \mathtt{if}\,(be)\,\mathtt{then}\,s\,\mathtt{else}\,s \mid \mathtt{while}\,(be)\,\mathtt{do}\,s \mid \mathtt{A}\texttt{:=}ae,\\
ae &::= n \mid [n, n'] \mid \mathtt{x} \in \mathit{Var} \mid A \in \mathbb{F} \mid ae \oplus ae,\\
be &::= ae \bowtie ae \mid \neg be \mid be \wedge be \mid be \vee be
\end{aligned}$$

where n ranges over integers $\mathbb{Z}$, $[n, n']$ over integer intervals, $\mathtt{x}$ over program variables *Var*, A over numerical features $\mathbb{F}$, and $\oplus \in \{+, -, *, /\}$, $\bowtie \in \{<, \leq, =, \neq\}$. Integer intervals $[n, n']$

denote a random choice of an integer in the interval. The set of all statements s is denoted by *Stm*; the set of all arithmetic expressions ae is denoted by *AExp*; the set of all boolean expressions be is denoted by *BExp*.

Semantics.

We now define the semantics of a dynamic program family. A *store* $\sigma : \Sigma = \mathit{Var} \to \mathbb{Z}$ is a mapping from program variables to values, whereas a *configuration* $k : \mathbb{K} = \mathbb{F} \to \mathbb{Z}$ is a mapping from numerical features to values. A *program state* $s = \langle \sigma, k \rangle : \Sigma \times \mathbb{K}$ is a pair consisting of a store $\sigma \in \Sigma$ and a configuration $k \in \mathbb{K}$. The semantics of arithmetic expressions $[\![ae]\!] : \Sigma \times \mathbb{K} \to \mathcal{P}(\mathbb{Z})$ is the set of possible values for expression ae in a given state. It is defined by induction on ae as a function from a store and a configuration to a set of values:

$$\begin{array}{l}[\![n]\!]\langle\sigma,k\rangle = \{n\},\ [\![[n,n']]\!]\langle\sigma,k\rangle = \{n,\dots,n'\},\ [\![\mathtt{x}]\!]\langle\sigma,k\rangle = \{\sigma(\mathtt{x})\}, \\ [\![A]\!]\langle\sigma,k\rangle = \{k(A)\},\ [\![ae_0 \oplus ae_1]\!]\langle\sigma,k\rangle = \{n_0 \oplus n_1 \mid n_0 \in [\![ae_0]\!]\langle\sigma,k\rangle, n_1 \in [\![ae_1]\!]\langle\sigma,k\rangle\}\end{array}$$

Similarly, the semantics of boolean expressions $[\![be]\!] : \Sigma \times \mathbb{K} \to \mathcal{P}(\{\text{true}, \text{false}\})$ is the set of possible truth values for expression be in a given state.

$$\begin{array}{l}[\![ae_0 \bowtie ae_1]\!]\langle\sigma,k\rangle = \{n_0 \bowtie n_1 \mid n_0 \in [\![ae_0]\!]\langle\sigma,k\rangle, n_1 \in [\![ae_1]\!]\langle\sigma,k\rangle\} \\ [\![\neg be]\!]\langle\sigma,k\rangle = \{\neg t \mid t \in [\![be]\!]\langle\sigma,k\rangle\}, \\ [\![be_0 \wedge be_1]\!]\langle\sigma,k\rangle = \{t_0 \wedge t_1 \mid t_0 \in [\![be_0]\!]\langle\sigma,k\rangle, t_1 \in [\![be_1]\!]\langle\sigma,k\rangle\} \\ [\![be_0 \vee be_1]\!]\langle\sigma,k\rangle = \{t_0 \vee t_1 \mid t_0 \in [\![be_0]\!]\langle\sigma,k\rangle, t_1 \in [\![be_1]\!]\langle\sigma,k\rangle\}\end{array}$$

We define an *invariance semantics* [12, 38] on the complete lattice $\langle \mathcal{P}(\Sigma\times\mathbb{K}), \subseteq, \cup, \cap, \emptyset, \Sigma\times\mathbb{K}\rangle$ by induction on the syntax of programs. It works on *sets of states*, so the property of interest is the possible sets of stores and configurations that may arise at each program location. In Fig. 4, we define the invariance semantics $[\![s]\!] : \mathcal{P}(\Sigma \times \mathbb{K}) \to \mathcal{P}(\Sigma \times \mathbb{K})$ of each program statement. The states resulting from the invariance semantics are built forward: each function $[\![s]\!]$ takes as input a set of states (i.e. pairs of stores and configurations) $S \in \mathcal{P}(\Sigma \times \mathbb{K})$ and outputs the set of possible states at the final location of the statement. The operation $k[A \mapsto n]$ (resp., $\sigma[\mathtt{x} \mapsto n]$) is used to update a configuration from $\mathbb{K}$ (resp., a store from Σ). Note that a `while` statement is given in a standard fixed-point formulation [12], where the fixed-point functional $\phi : \mathcal{P}(\Sigma \times \mathbb{K}) \to \mathcal{P}(\Sigma \times \mathbb{K})$ accumulates the possible states after another `while` iteration from a given set of states X.

However, the invariance semantics $[\![s]\!]$ is not computable since our language is Turing complete. In the following, we present sound decidable abstractions of $[\![s]\!]$ by means of decision tree-based abstract domains.

4 Decision Trees Lifted Domain

Lifted analyses are designed by *lifting* existing single-program analyses to work on program families, rather than on individual programs. Lifted analysis for traditional program families introduced in [21] relies on a *decision tree lifted domain*. The leaf nodes of decision trees belong to an existing single-program analysis domain, and are separated by linear constraints over numerical features, organized in decision nodes. In Section 4.1, we first recall basic elements of the decision tree lifted domain [21] that can be reused for dynamic program families. Then, in Section 4.2 we consider extended transfer functions for assignments and tests when features can freely occur in them, whereas in Section 4.3 we define the extrapolation widening operator for this lifted domain. Finally, we define the abstract invariance semantics based on this domain and show its soundness in Section 4.4.

$$
\begin{aligned}
[\![\texttt{skip}]\!]S &= S\\
[\![\texttt{x := } ae]\!]S &= \{\langle \sigma[\texttt{x} \mapsto n], k\rangle \mid \langle \sigma, k\rangle \in S, n \in [\![ae]\!]\langle \sigma, k\rangle\}\\
[\![s_1 \texttt{ ; } s_2]\!]S &= [\![s_2]\!]([\![s_1]\!]S)\\
[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]S &= [\![s_1]\!]\{\langle \sigma, k\rangle \in S \mid \text{true} \in [\![be]\!]\langle \sigma, k\rangle\} \cup\\
&\quad [\![s_2]\!]\{\langle \sigma, k\rangle \in S \mid \text{false} \in [\![be]\!]\langle \sigma, k\rangle\}\\
[\![\texttt{while } be \texttt{ do } s]\!]S &= \{\langle \sigma, k\rangle \in \text{lfp}\, \phi \mid \text{false} \in [\![be]\!]\langle \sigma, k\rangle\}\\
\phi(X) &= S \cup [\![s]\!]\{\langle \sigma, k\rangle \in X \mid \text{true} \in [\![be]\!]\langle \sigma, k\rangle\}\\
[\![A \texttt{ := } ae]\!]S &= \{\langle \sigma, k[A \mapsto n]\rangle \mid \langle \sigma, k\rangle \in S, n \in [\![ae]\!]\langle \sigma, k\rangle, k[A \mapsto n] \in \mathbb{K}\}
\end{aligned}
$$

Figure 4 Invariance semantics $[\![s]\!] : \mathcal{P}(\Sigma \times \mathbb{K}) \to \mathcal{P}(\Sigma \times \mathbb{K})$.

4.1 Basic elements

Abstract domain for leaf nodes.

We assume that a single-program numerical domain $\mathbb{D}$ defined over a set of variables V is equipped with sound operators for concretization $\gamma_{\mathbb{D}}$, ordering $\sqsubseteq_{\mathbb{D}}$, join $\sqcup_{\mathbb{D}}$, meet $\sqcap_{\mathbb{D}}$, the least element (called bottom) $\bot_{\mathbb{D}}$, the greatest element (called top) $\top_{\mathbb{D}}$, widening $\nabla_{\mathbb{D}}$, and narrowing $\triangle_{\mathbb{D}}$, as well as sound transfer functions for tests (boolean expressions) $\text{FILTER}_{\mathbb{D}}$ and forward assignments $\text{ASSIGN}_{\mathbb{D}}$. The domain $\mathbb{D}$ employs data structures and algorithms specific to the shape of invariants (analysis properties) it represents and manipulates. More specifically, the concretization function $\gamma_{\mathbb{D}}$ assigns a concrete meaning to each element in $\mathbb{D}$, ordering $\sqsubseteq_{\mathbb{D}}$ conveys the idea of approximation since some analysis results may be coarser than some other results, whereas join $\sqcup_{\mathbb{D}}$ and meet $\sqcap_{\mathbb{D}}$ 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 $\nabla_{\mathbb{D}}$ and narrowing $\triangle_{\mathbb{D}}$ are used. Transfer functions give abstract semantics of expressions and statements. Hence, $\text{ASSIGN}_{\mathbb{D}}(d : \mathbb{D}, \texttt{x:=}ae : \mathit{Stm})$ returns an updated version of d by abstractly evaluating $\texttt{x:=}ae$ in it, whereas $\text{FILTER}_{\mathbb{D}}(d : \mathbb{D}, be : \mathit{BExp})$ returns an abstract element from $\mathbb{D}$ obtained by restricting d to satisfy test be. In practice, the domain $\mathbb{D}$ will be instantiated with some of the known numerical domains, such as Intervals $\langle I, \sqsubseteq_I\rangle$ [12], Octagons $\langle O, \sqsubseteq_O\rangle$ [46], and Polyhedra $\langle P, \sqsubseteq_P\rangle$ [16]. The elements of I are intervals of the form: $\pm x \geq \beta$, where $x \in V, \beta \in \mathbb{Z}$; the elements of O are conjunctions of octagonal constraints of the form $\pm x_1 \pm x_2 \geq \beta$, where $x_1, x_2 \in V, \beta \in \mathbb{Z}$; while the elements of P are conjunctions of polyhedral constraints of the form $\alpha_1 x_1 + \ldots + \alpha_k x_k + \beta \geq 0$, where $x_1, \ldots x_k \in V, \alpha_1, \ldots, \alpha_k, \beta \in \mathbb{Z}$.

We will sometimes write $\mathbb{D}_V$ to explicitly denote the set of variables V over which domain $\mathbb{D}$ is defined. In this work, we use domains $\mathbb{D}_{\mathit{Var} \cup \mathbb{F}}$ for leaf nodes of decision trees that are defined over both program and feature variables. The abstraction for numerical domains $\langle \mathbb{D}_{\mathit{Var} \cup \mathbb{F}}, \sqsubseteq_{\mathbb{D}}\rangle$ is formally defined by the concretization-based abstraction $\langle \mathcal{P}(\Sigma \times \mathbb{K}), \subseteq\rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle \mathbb{D}_{\mathit{Var} \cup \mathbb{F}}, \sqsubseteq_{\mathbb{D}}\rangle$. We refer to [38] for a more detailed discussion of the definition of $\gamma_{\mathbb{D}}$ as well as other abstract operations and transfer functions for Intervals, Octagons, and Polyhedra.

Abstract domain for decision nodes.

We introduce a family of abstract domains for linear constraints $\mathbb{C}_{\mathbb{D}}$ defined over features $\mathbb{F}$, which are parameterized by any of the numerical domains $\mathbb{D}$ (intervals I, octagons O, polyhedra P). For example, the finite set of *polyhedral constraints* is $\mathbb{C}_P = \{\alpha_1 A_1 + \ldots +$

$\alpha_k A_k + \beta \geq 0 \mid A_1, \dots A_k \in \mathbb{F}, \alpha_1, \dots, \alpha_k, \beta \in \mathbb{Z}, \gcd(|\alpha_1|, \dots, |\alpha_k|, |\beta|) = 1\}$. The finite set $\mathbb{C}_\mathbb{D}$ of linear constraints over features $\mathbb{F}$ is constructed by the underlying numerical 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 a conjunction of constraints from $\mathbb{D}$ to a finite set of constraints in $\mathcal{P}(\mathbb{C}_\mathbb{D})$.

The domain of decision nodes is $\mathbb{C}_\mathbb{D}$. We assume the set of features $\mathbb{F} = \{A_1, \dots, A_n\}$ to be totally ordered, such that the ordering is $A_1 > \dots > A_n$. We impose a total order $<_{\mathbb{C}_\mathbb{D}}$ on $\mathbb{C}_\mathbb{D}$ to be the lexicographic order on the coefficients $\alpha_1, \dots, \alpha_n$ and constant α_{n+1} of the linear constraints, such that:

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

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

Abstract domain for decision trees.

A *decision tree* $t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}_\mathbb{F}}, \mathbb{D}_{Var \cup \mathbb{F}})$ over the sets $\mathbb{C}_{\mathbb{D}_\mathbb{F}}$ of linear constraints defined over $\mathbb{F}$ and the leaf abstract domain $\mathbb{D}_{Var \cup \mathbb{F}}$ defined over $Var \cup \mathbb{F}$ is: either a leaf node $\ll d \gg$ with $d \in \mathbb{D}_{Var \cup \mathbb{F}}$, or $[\![c : tl, tr]\!]$, where $c \in \mathbb{C}_{\mathbb{D}_\mathbb{F}}$ (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 1.** The following two decision trees t_1 and t_2 have decision and leaf nodes labelled with polyhedral linear constraints defined over numerical feature `A` with domain $\mathbb{Z}$ and over integer program variable y, respectively:

$$t_1 = [\![\mathtt{A} \geq 4 : \ll[y \geq 2]\gg, \ll[y = 0]\gg]\!], \;\; t_2 = [\![\mathtt{A} \geq 2 : \ll[y \geq 0]\gg, \ll[y \leq 0]\gg]\!] \qquad ◀$$

Abstract Operations.

We define the following concretization-based abstraction $\langle \mathcal{P}(\Sigma \times \mathbb{K}), \subseteq \rangle \xleftarrow{\gamma_\mathbb{T}} \langle \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{D}), \sqsubseteq_\mathbb{T} \rangle$. The *concretization function* $\gamma_\mathbb{T}$ of a decision tree $t \in \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{D})$ returns a set of pairs $\langle \sigma, k \rangle$, such that $\langle \sigma, k \rangle \in \gamma_\mathbb{D}(d)$ and k satisfies the set $C \in \mathcal{P}(\mathbb{C}_\mathbb{D})$ of constraints accumulated along the top-down path to the leaf node $d \in \mathbb{D}$. More formally, the concretization function $\gamma_\mathbb{T}(t) : \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{D}) \to \mathcal{P}(\Sigma \times \mathbb{K})$ is defined as:

$$\gamma_\mathbb{T}(t) = \overline{\gamma}_\mathbb{T}[\mathbb{K}](t)$$

where $\mathbb{K} \in \mathcal{P}(\mathbb{C}_\mathbb{D})$ is the set of configurations, i.e. the set of constraints over $\mathbb{F}$ taking into account the domains of features. Function $\overline{\gamma}_\mathbb{T} : \mathcal{P}(\mathbb{C}_\mathbb{D}) \to \mathbb{T}(\mathbb{C}_\mathbb{D}, \mathbb{D}) \to \mathcal{P}(\Sigma \times \mathbb{K})$ is defined as:

$$\overline{\gamma}_\mathbb{T}[C](\ll d \gg) = \{\langle \sigma, k \rangle \mid \langle \sigma, k \rangle \in \gamma_\mathbb{D}(d), k \models C\},$$
$$\overline{\gamma}_\mathbb{T}[C]([\![c : tl, tr]\!]) = \overline{\gamma}_\mathbb{T}[C \cup \{c\}](tl) \;\cup\; \overline{\gamma}_\mathbb{T}[C \cup \{\neg c\}](tr)$$

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

1 **if** $\text{isLeaf}(t_1) \wedge \text{isLeaf}(t_2)$ **then return** (t_1, t_2);
2 **if** $\text{isLeaf}(t_1) \vee (\text{isNode}(t_1) \wedge \text{isNode}(t_2) \wedge t_2.c <_{\mathbb{C}_{\mathbb{D}}} t_1.c)$ **then**
3   **if** $\text{isRedundant}(t_2.c, C)$ **then return** $\texttt{UNIFICATION}(t_1, t_2.l, C)$;
4   **if** $\text{isRedundant}(\neg t_2.c, C)$ **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** $([\![t_2.c : l_1, r_1]\!], [\![t_2.c : l_2, r_2]\!])$;
8 **if** $\text{isLeaf}(t_2) \vee (\text{isNode}(t_1) \wedge \text{isNode}(t_2) \wedge t_1.c <_{\mathbb{C}_{\mathbb{D}}} t_2.c)$ **then**
9   **if** $\text{isRedundant}(t_1.c, C)$ **then return** $\texttt{UNIFICATION}(t_1.l, t_2, C)$;
10   **if** $\text{isRedundant}(\neg t_1.c, C)$ **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** $([\![t_1.c : l_1, r_1]\!], [\![t_1.c : l_2, r_2]\!])$;
14 else
15   **if** $\text{isRedundant}(t_1.c, C)$ **then return** $\texttt{UNIFICATION}(t_1.l, t_2.l, C)$;
16   **if** $\text{isRedundant}(\neg t_1.c, C)$ **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** $([\![t_1.c : l_1, r_1]\!], [\![t_1.c : l_2, r_2]\!])$;

Note that $k \models C$ is equivalent with $\alpha_{\mathbb{C}_{\mathbb{D}}}(\{k\}) \sqsubseteq_{\mathbb{D}} \alpha_{\mathbb{C}_{\mathbb{D}}}(C)$, thus we can check $k \models C$ using the abstract operation $\sqsubseteq_{\mathbb{D}}$ of the numerical domain $\mathbb{D}$.

Other binary operations rely on the algorithm for *tree unification* [45] given in Algorithm 1, which finds a common labelling of two trees t_1 and t_2 by forcing them to have the same structure. It accumulates into the set $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ (initially equal to $\mathbb{K}$) the linear constraints encountered along the paths of the decision trees possibly adding new constraints as decision nodes (Lines 5–7, Lines 11–13) or removing constraints that are redundant with respect to C (Lines 3,4,9,10,15,16). This is done by using the function $\text{isRedundant}(c, C)$, which checks whether the linear constraint $c \in \mathbb{C}_{\mathbb{D}}$ is redundant with respect to the 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.

▶ **Example 2.** After tree unification of t_1 and t_2 from Example 1, we obtain:

$$\begin{aligned} t_1 &= [\![\texttt{A} \geq 4 : \ll[y \geq 2]\gg, [\![\texttt{A} \geq 2 : \ll[y = 0]\gg, \ll[y = 0]\gg]\!]]\!], \\ t_2 &= [\![\texttt{A} \geq 4 : \ll[y \geq 0]\gg, [\![\texttt{A} \geq 2 : \ll[y \geq 0]\gg, \ll[y \leq 0]\gg]\!]]\!] \end{aligned}$$

Note that the tree unification adds a decision node for $\texttt{A} \geq 2$ to the right subtree of t_1, whereas it adds a decision node for $\texttt{A} \geq 4$ to t_2 and removes the redundant constraint $\texttt{A} \geq 2$ from the resulting left subtree of t_2. ◀

Some binary operations are performed leaf-wise on the unified decision trees. Given two unified decision trees t_1 and t_2, their ordering $t_1 \sqsubseteq_{\mathbb{T}} t_2$, join $t_1 \sqcup_{\mathbb{T}} t_2$, and meet $t_1 \sqcap_{\mathbb{T}} t_2$ are defined recursively:

$$\begin{array}{ll} \ll d_1 \gg \sqsubseteq_{\mathbb{T}} \ll d_2 \gg = d_1 \sqsubseteq_{\mathbb{D}} d_2, & [\![c : tl_1, tr_1]\!] \sqsubseteq_{\mathbb{T}} [\![c : tl_2, tr_2]\!] = (tl_1 \sqsubseteq_{\mathbb{T}} tl_2) \wedge (tr_1 \sqsubseteq_{\mathbb{T}} tr_2) \\ \ll d_1 \gg \sqcup_{\mathbb{T}} \ll d_2 \gg = \ll d_1 \sqcup_{\mathbb{D}} d_2 \gg, & [\![c : tl_1, tr_1]\!] \sqcup_{\mathbb{T}} [\![c : tl_2, tr_2]\!] = [\![c : tl_1 \sqcup_{\mathbb{T}} tl_2, tr_1 \sqcup_{\mathbb{T}} tr_2]\!] \\ \ll d_1 \gg \sqcap_{\mathbb{T}} \ll d_2 \gg = \ll d_1 \sqcap_{\mathbb{D}} d_2 \gg, & [\![c : tl_1, tr_1]\!] \sqcap_{\mathbb{T}} [\![c : tl_2, tr_2]\!] = [\![c : tl_1 \sqcap_{\mathbb{T}} tl_2, tr_1 \sqcap_{\mathbb{T}} tr_2]\!] \end{array}$$

The top is a tree with a single $\top_{\mathbb{D}}$ leaf: $\top_{\mathbb{T}} = \ll \top_{\mathbb{D}} \gg$, while the bottom is a tree with a single $\bot_{\mathbb{D}}$ leaf: $\bot_{\mathbb{T}} = \ll \bot_{\mathbb{D}} \gg$.

▶ **Example 3.** Consider the unified trees t_1 and t_2 from Example 2. We have that $t_1 \sqsubseteq_{\mathbb{T}} t_2$ holds, $t_1 \sqcup_{\mathbb{T}} t_2 = [\![\mathtt{A} \geq 4 : \ll [y \geq 0] \gg, [\![\mathtt{A} \geq 2 : \ll [y \geq 0] \gg, \ll [y \leq 0] \gg]\!]]\!]$, and $t_1 \sqcap_{\mathbb{T}} t_2 = [\![\mathtt{A} \geq 4 : \ll [y \geq 2] \gg, [\![\mathtt{A} \geq 2 : \ll [y = 0] \gg, \ll [y = 0] \gg]\!]]\!]$. ◀

The concretization function $\gamma_{\mathbb{T}}$ is monotonic with respect to the ordering $\sqsubseteq_{\mathbb{T}}$.

▶ **Lemma 4.** $\forall t_1, t_2 \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$: $t_1 \sqsubseteq_{\mathbb{T}} t_2 \implies \gamma_{\mathbb{T}}(t_1) \subseteq \gamma_{\mathbb{T}}(t_2)$.

Proof. Let $t_1, t_2 \in \mathbb{T}$ such that $t_1 \sqsubseteq_{\mathbb{T}} t_2$. The ordering $\sqsubseteq_{\mathbb{T}}$ between decision trees is implemented by first calling the tree unification algorithm, and then by comparing the decision trees "leaf-wise". Tree unification forces the same structure on decision trees, so all paths to the leaf nodes coincide between the unified decision trees. Let $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ denote the set of linear constraints satisfied along a path of the unified decision trees, and let $d_1, d_2 \in \mathbb{D}_{Var \cup \mathbb{F}}$ denote the leaf nodes reached following the path C within the first and the second decision tree. Since $t_1 \sqsubseteq_{\mathbb{T}} t_2$, we have that $d_1 \sqsubseteq_{\mathbb{D}} d_2$ and so $\gamma_{\mathbb{D}}(d_1) \subseteq \gamma_{\mathbb{D}}(d_2)$. The proof follows from: $\{\langle \sigma, k \rangle \mid \langle \sigma, k \rangle \in \gamma_{\mathbb{D}}(d_1), k \models C\} \subseteq \{\langle \sigma, k \rangle \mid \langle \sigma, k \rangle \in \gamma_{\mathbb{D}}(d_2), k \models C\}$. ◀

Basic Transfer functions.

We define basic lifted transfer functions for forward assignments ($\text{ASSIGN}_{\mathbb{T}}$) and tests ($\text{FILTER}_{\mathbb{T}}$), when only program variables occur in given assignments and tests (boolean expressions). Those basic transfer functions $\text{ASSIGN}_{\mathbb{T}}$ and $\text{FILTER}_{\mathbb{T}}$ modify only leaf nodes since the analysis information about program variables is located in leaf nodes while the information about features is located in both decision nodes and leaf nodes.

Algorithm 2 $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$ **when** $\text{vars}(ae) \subseteq \textit{Var}$

```
if isLeaf(t) then return ≪ASSIGN_D_{Var∪F}(t, x:=ae)≫;
if isNode(t) then
    l = ASSIGN_T(t.l, x:=ae, C ∪ {t.c});
    r = ASSIGN_T(t.r, x:=ae, C ∪ {¬t.c});
    return ⟦t.c : l, r⟧
```

Algorithm 3 $\texttt{FILTER}_{\mathbb{T}}(t, be, C)$ **when** $\text{vars}(be) \subseteq \textit{Var}$

```
if isLeaf(t) then return ≪FILTER_D_{Var∪F}(t, be)≫;
if isNode(t) then
    l = FILTER_T(t.l, be, C ∪ {t.c});
    r = FILTER_T(t.r, be, C ∪ {¬t.c});
    return ⟦t.c : l, r⟧
```

Basic transfer function $\text{ASSIGN}_{\mathbb{T}}$ for handling an assignment $\texttt{x:=}ae$ is described by Algorithm 2. Note that $\mathtt{x} \in \textit{Var}$ is a program variable, and $ae \in \textit{AExp}$ may contain only program variables, i.e. the set of variables that occur in ae is $\text{vars}(ae) \subseteq \textit{Var}$. $\text{ASSIGN}_{\mathbb{T}}$ descends along the paths of the decision tree t up to a leaf node d, where $\text{ASSIGN}_{\mathbb{D}_{Var \cup \mathbb{F}}}$ is invoked to substitute expression ae for variable $\mathtt{x}$ in d. Similarly, basic transfer function

Algorithm 4 $\texttt{FILTER}_{\mathbb{T}}(t, be, C)$ **when** $\mathbf{vars}(be) \subseteq \mathbb{F}$

1 **switch** be **do**
2   **case** $(ae_0 \bowtie ae_1) \mid\mid (\neg(ae_0 \bowtie ae_1))$ **do**
3         $J = \texttt{FILTER}_{\mathbb{D}_{\mathbb{F}}}(\top_{\mathbb{D}_{\mathbb{F}}}, be)$; **return** $\texttt{RESTRICT}(t, C, J)$
4   **case** $be_1 \wedge be_2$ **do**
5   **return** $\texttt{FILTER}_{\mathbb{T}}(t, be_1, C) \sqcap_{\mathbb{T}} \texttt{FILTER}_{\mathbb{T}}(t, be_2, C)$
6   **case** $be_1 \vee be_2$ **do**
7   **return** $\texttt{FILTER}_{\mathbb{T}}(t, be_1, C) \sqcup_{\mathbb{T}} \texttt{FILTER}_{\mathbb{T}}(t, be_2, C)$

$\text{FILTER}_{\mathbb{T}}$ for handling tests $be \in \mathit{BExp}$ when $\text{vars}(be) \subseteq \mathit{Var}$, given in Algorithm 3, is implemented by applying $\text{FILTER}_{\mathbb{D}_{\mathit{Var}\cup\mathbb{F}}}$ leaf-wise, so that be is satisfied by all leaves.

Note that, in program families with static feature binding, features occur only in presence conditions (tests) of `#if` directives. Thus, special transfer functions $\text{FEAT-FILTER}_{\mathbb{T}}$ for feature-based tests and $\text{IFDEF}_{\mathbb{T}}$ for `#if` directives are defined in [21], which can add, modify, or delete decision nodes of a decision tree. Therefore, the basic transfer function $\text{FILTER}_{\mathbb{T}}$ for handling tests $be \in \mathit{BExp}$ when $\text{vars}(be) \subseteq \mathbb{F}$ coincides with $\text{FEAT-FILTER}_{\mathbb{T}}$ in [21], and is given in Algorithm 4. It reasons by induction on the structure of be. When be is a comparison of arithmetic expressions (Lines 2,3), we use $\text{FILTER}_{\mathbb{D}_{\mathbb{F}}}$ to approximate be, 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 be. This is done by calling function $\texttt{RESTRICT}(t, C, J)$, 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 be may not be representable exactly in $\mathbb{C}_{\mathbb{D}}$ (e.g., in the case of non-linear constraints over $\mathbb{F}$), so $\text{FILTER}_{\mathbb{D}_{\mathbb{F}}}$ may produce a set of constraints approximating it. When be 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 $\sqcup_{\mathbb{T}}$).

The above transfer function and some of the remaining operations rely on function `RESTRICT` given in Algorithm 5 for constraining a decision tree t with respect to a given set J of linear constraints over $\mathbb{F}$. The subtrees whose paths from the root satisfy these constraints are preserved, while leafs of the other subtrees are replaced with bottom $\bot_{\mathbb{D}}$. Function $\texttt{RESTRICT}(t, C, J)$ takes as input a decision tree t, a set C of constraints accumulated along paths up to a node, 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). At each iteration, the smallest linear constraint j is extracted from J (Line 9), and is handled appropriately based on whether j is smaller or equal (Line 11–15), or greater (Line 17–21) to the constraint at the node of t we currently consider.

4.2 Extended transfer functions

We now define extended transfer functions $\text{ASSIGN}_{\mathbb{T}}$ and $\text{FILTER}_{\mathbb{T}}$ where assignments and tests may contain both feature and program variables.

Assignments.

Transfer function $\text{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$, when $\text{vars}(ae) \subseteq \mathit{Var} \cup \mathbb{F}$, is given in Algorithm 6. It accumulates the constraints along the paths in the decision tree t in a set of constraints

Algorithm 5 $\texttt{RESTRICT}(t, C, J)$

---

if isEmpty(J) **then**
 if isLeaf(t) **then** return t;
 if isRedundant$(t.c, C)$ **then** return $\texttt{RESTRICT}(t.l, C, J)$;
 if isRedundant$(\neg t.c, C)$ **then** return $\texttt{RESTRICT}(t.r, C, J)$;
 $l = \texttt{RESTRICT}(t.l, C \cup \{t.c\}, J)$;
 $r = \texttt{RESTRICT}(t.r, C \cup \{\neg t.c\}, J)$;
 return $(\llbracket t.c : l, r \rrbracket)$**;**
else
 $j = min_{<_{\mathbb{C}_{\mathbb{D}}}}(J)$;
 if isLeaf$(t) \vee ($isNode$(t) \wedge j \leq_{\mathbb{C}_{\mathbb{D}}} t.c)$ **then**
  if isRedundant(j, C) **then** return $\texttt{RESTRICT}(t, C, J \backslash \{j\})$;
  if isRedundant$(\neg j, C)$ **then** return $\ll \perp_{\mathbb{A}} \gg$;
  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** $(\llbracket j : \texttt{RESTRICT}(t, C \cup \{j\}, J \backslash \{j\}), \ll \perp_{\mathbb{A}} \gg \rrbracket)$ **;**
  else return $(\llbracket j : \ll \perp_{\mathbb{A}} \gg, \texttt{RESTRICT}(t, C \cup \{\neg j\}, J \backslash \{j\}) \rrbracket)$ **;**
 else
  if isRedundant$(t.c, C)$ **then** return $\texttt{RESTRICT}(t.l, C, J)$;
  if isRedundant$(\neg t.c, C)$ **then** return $\texttt{RESTRICT}(t.r, C, J)$;
  $l = \texttt{RESTRICT}(t.l, C \cup \{t.c\}, J)$;
  $r = \texttt{RESTRICT}(t.r, C \cup \{\neg t.c\}, J)$;
  return $(\llbracket t.c : l, r \rrbracket)$**;**

$C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ (Lines 8–10), which is initialized to $\mathbb{K}$, up to the leaf nodes in which assignment is performed by $\text{ASSIGN}_{\mathbb{D}_{Var \cup \mathbb{F}}}$. That is, we first merge constraints from the leaf node t defined over $Var \cup \mathbb{F}$ and constraints from decision nodes $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}_{\mathbb{F}}})$ defined over $\mathbb{F}$, by using $\uplus_{Var \cup \mathbb{F}}$ operator. Thus, we obtain an abstract element from $\mathbb{D}_{Var \cup \mathbb{F}}$ on which the assignment operator of the domain $\mathbb{D}_{Var \cup \mathbb{F}}$ is applied (Line 2).

Transfer function $\text{ASSIGN}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$, when $\text{vars}(ae) \subseteq Var \cup \mathbb{F}$, is implemented by Algorithm 7. It calls the auxiliary function $\text{ASSIGN-AUX}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$, which performs the assignment on each leaf node t merged with the set of linear constraints C collected along the path to the leaf (Line 6). The obtained result d' is a new leaf node (Line 7), and furthermore it is projected on feature variables using $\restriction_{\mathbb{F}}$ operator to generate a new set of constraints $J = \gamma_{\mathbb{C}_{\mathbb{D}}}(d' \restriction_{\mathbb{F}})$ that needs to be substituted to C in the decision tree (Lines 8–13). The substitution is done at each decision node, such that new sets of constraints J_1 and J_2 are collected from its left and right subtrees, and then they are used as constraints in the given decision node instead of $t.c$ and $\neg t.c$. Let $J = J_1 \cap J_2$ be the common (overlapping) set of constraints that arise due to non-determinism (Line 11). When both $J_1 \backslash J$ and $J_2 \backslash J$ are empty, the left and the right subtrees are joined (Line 12). Otherwise, the corresponding tree is constructed using sets $J_1 \backslash J$ and $J_2 \backslash J$ and together with the set J are propagated to the parent node (Line 13). Note that, if some of the sets of constraints J, $J_1 \backslash J$, and $J_2 \backslash J$ is empty in the returned trees in Lines 12-13, then it is considered as a true constraint so that its true branch is always taken.

Algorithm 6 $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$ **when vars**$(ae) \subseteq \textbf{\textit{Var}} \cup \mathbb{F}$

if isLeaf(t) **then**
  $d' = \texttt{ASSIGN}_{\mathbb{D}_{Var \cup \mathbb{F}}}(t \uplus_{Var \cup \mathbb{F}} \alpha_{\mathbb{C}_{\mathbb{D}}}(C), \texttt{x:=}ae)$;
  return $\ll d' \gg$
if isNode(t) **then**
  $l = \text{ASSIGN}_{\mathbb{T}}(t.l, \texttt{x:=}ae, C \cup \{t.c\})$;
  $r = \text{ASSIGN}_{\mathbb{T}}(t.r, \texttt{x:=}ae, C \cup \{\neg t.c\})$;
  return $[\![t.c : l, r]\!]$

Algorithm 7 $\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$ **when vars**$(ae) \subseteq \textbf{\textit{Var}} \cup \mathbb{F}$

(t,d) = $\texttt{ASSIGN-AUX}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$
return t

Function $\texttt{ASSIGN-AUX}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$**:**
  if isLeaf(t) **then**
    $d' = \texttt{ASSIGN}_{\mathbb{D}_{Var \cup \mathbb{F}}}(t \uplus_{Var \cup \mathbb{F}} \alpha_{\mathbb{C}_{\mathbb{D}}}(C), \texttt{A:=}ae)$
    return $(\ll d' \gg, \gamma_{\mathbb{C}_{\mathbb{D}}}(d' \upharpoonright_{\mathbb{F}}))$
  if isNode(t) **then**
    $(t_1, J_1) = \texttt{ASSIGN-AUX}_{\mathbb{T}}(t.l, \texttt{A:=}ae, C \cup \{t.c\})$
    $(t_2, J_2) = \texttt{ASSIGN-AUX}_{\mathbb{T}}(t.r, \texttt{A:=}ae, C \cup \{\neg t.c\})$
    $J = J_1 \cap J_2$
    if isEmpty$(J_1 \backslash J) \wedge$ isEmpty$(J_2 \backslash J)$ **then return** $\big([\![J, t_1 \sqcup_{\mathbb{T}} t_2, \perp_{\mathbb{T}}]\!], \emptyset\big)$
    else return $\big([\![J_1 \backslash J, t_1, [\![J_2 \backslash J, t_2, \perp_{\mathbb{T}}]\!]]\!], J\big)$

Tests.

Transfer function $\text{FILTER}_{\mathbb{T}}(t, be, C)$, when $\text{vars}(be) \subseteq Var \cup \mathbb{F}$, is described by Algorithm 8. Similarly to $\text{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$ in Algorithm 6, it accumulates the constraints along the paths in the decision tree t in a set of constraints $C \in \mathcal{P}(\mathbb{C}_{\mathbb{D}})$ up to the leaf nodes (Lines 6–9). When t is a leaf node, test be is handled using $\text{FILTER}_{\mathbb{D}_{Var \cup \mathbb{F}}}$ applied on an abstract element from $\mathbb{D}_{Var \cup \mathbb{F}}$ obtained by merging constraints in the leaf node and decision nodes along the path to the leaf (Lines 2). The obtained result d' represents a new leaf node, and furthermore d' is projected on feature variables using $\upharpoonright_{\mathbb{F}}$ operator to generate a new set of constraints J that is added to the given path to d' (Lines 3–5).

Note that the trees returned by $\text{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, C)$, $\text{ASSIGN}_{\mathbb{T}}(t, \texttt{A:=}ae, C)$, and $\text{FILTER}_{\mathbb{T}}(t, be, C)$ are sorted (normalized) to remove possible multiple occurrences of a constraint c, possible occurrences of both c and $\neg c$, and possible ordering inconsistences. Moreover, the obtained decision trees may contain some redundancy that can be exploited to further compress them. We use several optimizations [21, 45]. E.g., if constraints on a path to some leaf are unsatisfiable, we eliminate that leaf node; if a decision node contains two same subtrees, then we keep only one subtree and we also eliminate the decision node, etc.

Algorithm 8 $\texttt{FILTER}_{\mathbb{T}}(t, be, C)$ **when** $\mathbf{vars}(be) \subseteq \boldsymbol{Var} \cup \mathbb{F}$

if isLeaf(t) **then**
 $d' = \texttt{FILTER}_{\mathbb{D}_{Var\cup\mathbb{F}}}(t \uplus_{Var\cup\mathbb{F}} \alpha_{\mathbb{C}_{\mathbb{D}}}(C), be)$;
 $J = \gamma_{\mathbb{C}_{\mathbb{D}}}(d' \restriction_{\mathbb{F}})$;
 if isRedundant(J, C) **then return** $\ll d' \gg$;
 else return $\texttt{RESTRICT}(\ll d' \gg, C, J \backslash C)$;
if isNode(t) **then**
 $l = \text{FILTER}_{\mathbb{T}}(t.l, be, C \cup \{t.c\})$;
 $r = \text{FILTER}_{\mathbb{T}}(t.r, be, C \cup \{\neg t.c\})$;
 return $[\![t.c : l, r]\!]$

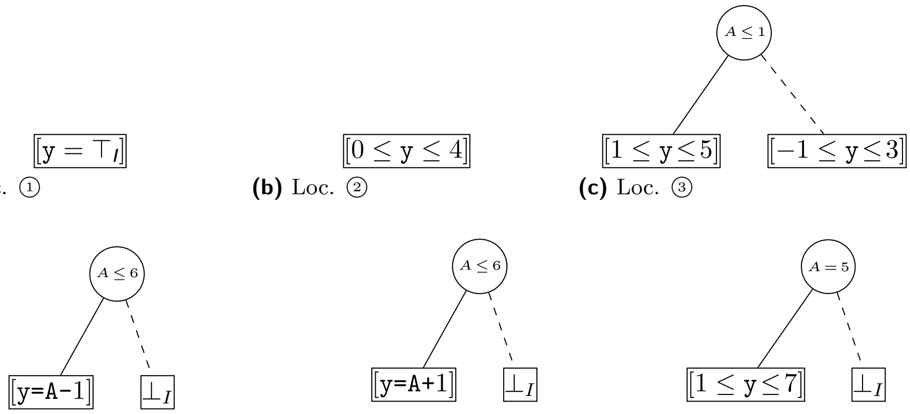
▶ **Example 5.** Let us consider the following dynamic program family P':

① `int y :=` $[0,4]$;
② `if` $(A < 2)$ `y := y+1; else y := y-1;`
③ A `:= y+1;`
④ `y :=` A`+1;`
⑤ A `:= 5;` ⑥

The code base of P' contains only one program variable $Var = \{\texttt{y}\}$ and one numerical feature $\mathbb{F} = \{A\}$ with domain $dom(A) = [0, 99]$. In Fig. 5 we depict decision trees inferred by performing *polyhedral* lifted analysis using the lifted domain $\mathbb{T}(\mathbb{C}_P, P)$. We use $\text{FILTER}_{\mathbb{T}}$ from Algorithm 4 to analyze statement at location ② and infer the decision tree at location ③. Then, we use $\texttt{ASSIGN}_{\mathbb{T}}$ from Algorithm 7 to analyze the statement A `:= y+1` at ③ and infer the tree at location ④. Note that, by using the left and right leafs in the input tree at ③, we generate constraint sets $J_1 = (2 \leq A \leq 6)$ and $J_2 = (0 \leq A \leq 4)$ with the same leaf nodes [`y=`A`-1`]. After applying reductions, we obtain the tree at location ④. Recall that we implicitly assume the correctness of linear constraints $\mathbb{K}$ that take into account domains of features. Hence, node $(A \leq 6)$ is satisfied when $(A \leq 6) \wedge (0 \leq A \leq 99)$, where constraint $(0 \leq A \leq 99)$ represents the domains of A. Finally, statement `y :=` A`+1` at location ④ is analyzed using Algorithm 6 such that all leafs in the input tree are updated accordingly, whereas statement A `:= 5` at location ⑤ is analyzed using Algorithm 7 such that all leafs in the input tree along the paths to them are joined to create new leaf that satisfies $(A = 5)$.

4.3 Widening

The widening operator $\nabla_{\mathbb{T}}$ is necessary in order to extrapolate an analysis property over configurations (values of features) and stores (values of program variables) on which it is not yet defined. Hence, it provides a way to handle (potentially) infinite reconfiguration of features inside loops. The widening $t_1 \nabla_{\mathbb{T}} t_2$ is implemented by calling function $\texttt{WIDEN}_{\mathbb{T}}(t_1, t_2, \mathbb{K})$, where t_1 and t_2 are two decision trees and $\mathbb{K}$ is the set of valid configurations. Function $\texttt{WIDEN}_{\mathbb{T}}$, given in Algorithm 9, first calls function `LEFT_UNIFICATION` (Line 1) that performs widening of the configuration space (i.e., decision nodes), and then extrapolates the value of leafs by calling function `WIDEN_LEAF` (Line 2). Function `LEFT_UNIFICATION` (Lines 4–17) limits the size of decision trees, and thus avoids infinite sequences of partition refinements. It forces the structure of t_1 on t_2. This way, there may be information loss by applying this function. `LEFT_UNIFICATION` accumulates into a set C (initially equal to $\mathbb{K}$) the linear constraints along the paths in the first decision tree, possibly adding nodes to the second



(a) Loc. ① **(b)** Loc. ② **(c)** Loc. ③

(d) Loc. ④ **(e)** Loc. ⑤ **(f)** Loc. ⑥

Figure 5 Decision tree-based (polyhedral) invariants at program locations from ① to ⑥ of P'.

tree (Lines 10–17), or removing decision nodes from the second tree in which case the left and the right subtree are joined (Lines 6–9), or removing constraints that are redundant (Lines 7,8 and 11,12). Finally, function `WIDEN_LEAF` (Line 18–23) applies the widening $\nabla_{\mathbb{D}}$ leaf-wise on the left unified decision trees.

▶ **Example 6.** Consider the following two decision trees t_1 and t_2:

$$t_1 = [\![\mathtt{A}>1 : [\![\mathtt{A}>5 : \ll[y \ge 0]\gg, \ll[y \le 0]\gg]\!], \ll[y=0]\gg]\!]$$
$$t_2 = [\![\mathtt{A}>2 : \ll[y=1]\gg, \ll[y>1]\gg]\!]$$

After applying the left unification of t_1 and t_2, the tree t_2 becomes:

$$t_2 = [\![\mathtt{A}>1 : [\![\mathtt{A}>5 : \ll[y=1]\gg, \ll[y \ge 1]\gg]\!], \ll[y>1]\gg]\!]$$

Note that when $(\mathtt{A}>1)$ and $\neg(\mathtt{A}>5)$, the left and right leafs of the input t_2 are joined, thus yielding the leaf $[y \ge 1]$ in the left-unified t_2. This represents an example of information-loss in a left-unified tree. After applying the leaf-wise widening of t_1 and left-unified t_2, we obtain:

$$t = [\![\mathtt{A}>1 : [\![\mathtt{A}>5 : \ll[y \ge 0]\gg, \ll\top\gg]\!], \ll[y \ge 0]\gg]\!] \qquad ◀$$

4.4 Soundness

The operations and transfer functions of the decision tree lifted domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$ can now be used to define the abstract invariance semantics. In Fig. 6, we define the *abstract invariance semantics* $[\![s]\!]^\natural : \mathbb{T} \to \mathbb{T}$ for each statement s. Function $[\![s]\!]^\natural$ takes as input a decision tree over-approximating the set of reachable states at the initial location of statement s, and outputs a decision tree that over-approximates the set of reachable states at the final location od s. For a `while` loop, $\mathrm{lfp}^\natural\, \phi^\natural$ is the limit of the following increasing chain defined by the widening operator $\nabla_{\mathbb{T}}$ (note that, $t_1 \nabla_{\mathbb{T}} t_2 = \mathtt{WIDEN}_{\mathbb{T}}(t_1, t_2, \mathbb{K})$):

$$y_0 = \bot_{\mathbb{T}}, \quad y_{n+1} = y_n \nabla_{\mathbb{T}} \phi^\natural(y_n)$$

The lifted analysis (abstract invariance semantics) of a dynamic program family s is defined as $[\![s]\!]^\natural t_{in}$, where the input tree t_{in} at the initial location has only one leaf node $\top_{\mathbb{D}}$ and decision nodes define the set $\mathbb{K}$. Note that $t_{in} = \ll\top_{\mathbb{D}}\gg$ if there are no constraints in $\mathbb{K}$. This way, by calculating $[\![s]\!]^\natural t_{in}$ we collect the possible invariants in the form of decision trees at all program locations.

Algorithm 9 $\texttt{WIDEN}_{\mathbb{T}}(t_1, t_2, C)$

```
(t1, t2) = LEFT_UNIFICATION(t1, t2, C)
return WIDEN_LEAF(t1, t2, C)

Function LEFT_UNIFICATION(t1, t2, C):
    if isLeaf(t1) ∧ isLeaf(t2) then return (t1, t2)
    if isLeaf(t1) ∨ (isNode(t1) ∧ isNode(t2) ∧ t2.c <_CD t1.c) then
        if isRedundant(t2.c, C) then return LEFT_UNIFICATION(t1, t2.l, C)
        if isRedundant(¬t2.c, C) then return LEFT_UNIFICATION(t1, t2.r, C)
        return LEFT_UNIFICATION(t1, t2.l ⊔_T t2.r, C)
    if isLeaf(t2) ∨ (isNode(t1) ∧ isNode(t2) ∧ t1.c ≤_CD t2.c) then
        if isRedundant(t1.c, C) then return UNIFICATION(t1.l, t2, C)
        if isRedundant(¬t1.c, C) then return UNIFICATION(t1.r, t2, C)
        if t1.c <_CD t2.c then  t21 = t2; t22 = t2;
        else t21 = t2.l; t22 = t2.r;
        (l1, l2) = UNIFICATION(t1.l, t21, C ∪ {t1.c})
        (r1, r2) = UNIFICATION(t1.r, t22, C ∪ {¬t1.c})
        return ([[t1.c : l1, r1]], [[t1.c : l2, r2]])

Function WIDEN_LEAF(t1, t2, C):
    if isLeaf(t1) ∧ isLeaf(t2) then return (≪t1 ∇_D t2≫)
    if isNode(t1) ∧ isNode(t2) then
        l = WIDEN_LEAF(t1.l, t2.l, C ∪ {t1.c})
        r = WIDEN_LEAF(t1.r, t2.r, C ∪ {¬t1.c})
        return ([[t1.c : l, r]])
```

$$
\begin{aligned}
[\![\texttt{skip}]\!]^\natural t &= t\\
[\![\texttt{x := } ae]\!]^\natural t &= \texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, \mathbb{K})\\
[\![s_1 \texttt{ ; } s_2]\!]^\natural t &= [\![s_2]\!]^\natural([\![s_1]\!]^\natural t)\\
[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]^\natural t &= [\![s_1]\!]^\natural(\texttt{FILTER}_{\mathbb{T}}(t, be, \mathbb{K})) \sqcup_{\mathbb{T}} [\![s_2]\!]^\natural(\texttt{FILTER}_{\mathbb{T}}(t, \neg be, \mathbb{K}))\\
[\![\texttt{while } be \texttt{ do } s]\!]^\natural t &= \texttt{FILTER}_{\mathbb{T}}(\mathrm{lfp}^\natural\, \phi^\natural, \neg be, \mathbb{K})\\
\phi^\natural(x) &= t \sqcup_{\mathbb{T}} [\![s]\!]^\natural(\texttt{FILTER}_{\mathbb{T}}(x, be, \mathbb{K}))\\
[\![A \texttt{ := } ae]\!]^\natural t &= \texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{A:=}ae, \mathbb{K})
\end{aligned}
$$

Figure 6 Abstract invariance semantics $[\![s]\!]^\natural : \mathbb{T} \to \mathbb{T}$.

We can establish soundness of the abstract invariant semantics $[\![s]\!]^\natural t_{in} \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$ with respect to the invariance semantics $[\![s]\!]\langle\Sigma, \mathbb{K}\rangle \in \mathcal{P}(\Sigma \times \mathbb{K})$, where $\langle\Sigma, \mathbb{K}\rangle = \{\langle\sigma, k\rangle \mid \sigma \in \Sigma, k \in \mathbb{K}\}$, by showing that $[\![s]\!]\langle\Sigma, \mathbb{K}\rangle \subseteq \gamma_{\mathbb{T}}\big([\![s]\!]^\natural t_{in}\big)$. This is done by proving the following result. [2]

▶ **Theorem 7** (Soundness). $\forall t \in \mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D}) : [\![s]\!]\gamma_{\mathbb{T}}(t) \subseteq \gamma_{\mathbb{T}}\big([\![s]\!]^\natural t\big)$.

Proof. The proof is by structural induction on s. We consider the most interesting cases.

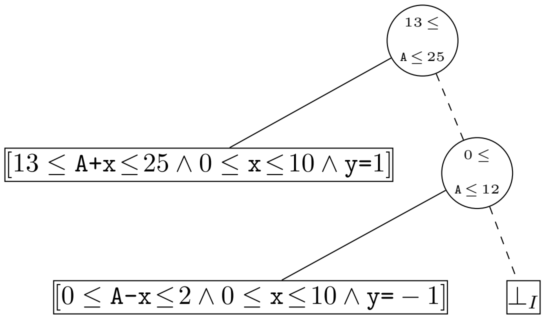
Case `skip`. $[\![\texttt{skip}]\!]\gamma_{\mathbb{T}}(t) = \gamma_{\mathbb{T}}(t) = \gamma_{\mathbb{T}}([\![\texttt{skip}]\!]^\natural t)$.

Case $\texttt{x:=}ae$. Let $\langle\sigma, k\rangle \in \gamma_{\mathbb{T}}(t)$. By definition of $[\![\texttt{x := } ae]\!]$ in Fig. 4, it holds that $\langle\sigma[\texttt{x} \mapsto n], k\rangle \in [\![\texttt{x := } ae]\!]\gamma_{\mathbb{T}}(t)$ for all $n \in [\![ae]\!]\langle\sigma, k\rangle$. Since $\langle\sigma, k\rangle \in \gamma_{\mathbb{T}}(t)$, there must be a leaf node d of t and a set of constraints C collected along the path to d, such that $\langle\sigma, k\rangle \in \gamma_{\mathbb{D}}(d) \wedge k \models C$. By definition of the abstraction $\langle\mathcal{P}(\Sigma \times \mathbb{K}), \subseteq\rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle\mathbb{D}_{Var\cup\mathbb{F}}, \sqsubseteq_{\mathbb{D}}\rangle$, the soundness of $\texttt{ASSIGN}_{\mathbb{D}_{Var\cup\mathbb{F}}}$, and by definition of $\texttt{ASSIGN}_{\mathbb{T}}$ (cf. Algorithms 2 and 6), it must hold $\langle\sigma[\texttt{x} \mapsto n], k\rangle \in \gamma_{\mathbb{T}}(\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x := } ae, \mathbb{K}))$ due to the fact that Algorithms 2 and 6 invoke $\texttt{ASSIGN}_{\mathbb{D}_{Var\cup\mathbb{F}}}$ for every leaf node of t that may be merged with linear constraints from decision nodes found on the path from the root to that leaf. Thus, we conclude $[\![\texttt{x := } ae]\!]\gamma_{\mathbb{T}}(t) \subseteq \gamma_{\mathbb{T}}(\texttt{ASSIGN}_{\mathbb{T}}(t, \texttt{x:=}ae, \mathbb{K})) = \gamma_{\mathbb{T}}([\![\texttt{x := } ae]\!]^\natural t)$.

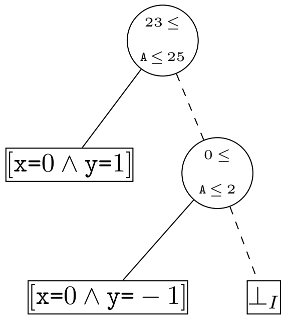
Case $\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2$. Let $\langle\sigma, k\rangle \in \gamma_{\mathbb{T}}(t)$ and $\langle\sigma', k'\rangle \in [\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]\{\langle\sigma, k\rangle\}$. By structural induction, we have that $[\![s_1]\!]\gamma_{\mathbb{T}}(t') \subseteq \gamma_{\mathbb{T}}([\![s_1]\!]^\natural t')$ and $[\![s_2]\!]\gamma_{\mathbb{T}}(t') \subseteq \gamma_{\mathbb{T}}([\![s_2]\!]^\natural t')$ for any t'. By definition of $[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]$ in Fig. 4, we have that $\langle\sigma', k'\rangle \in [\![s_1]\!]\{\langle\sigma, k\rangle\}$ if true $\in [\![be]\!]\langle\sigma, k\rangle$ or $\langle\sigma', k'\rangle \in [\![s_2]\!]\{\langle\sigma, k\rangle\}$ if false $\in [\![be]\!]\langle\sigma, k\rangle$. Since $\langle\sigma, k\rangle \in \gamma_{\mathbb{T}}(t)$, there must be a leaf node d of t and a set of constraints C collected along the path to d, such that $\langle\sigma, k\rangle \in \gamma_{\mathbb{D}}(d) \wedge k \models C$. By definition of the abstraction $\langle\mathcal{P}(\Sigma \times \mathbb{K}), \subseteq\rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle\mathbb{D}_{Var\cup\mathbb{F}}, \sqsubseteq_{\mathbb{D}}\rangle$, the soundness of $\texttt{FILTER}_{\mathbb{D}_{Var\cup\mathbb{F}}}$, and by definition of $\texttt{FILTER}_{\mathbb{T}}$ (cf. Algorithms 2, 4, and 8), it must hold that $\langle\sigma, k\rangle \in \gamma_{\mathbb{T}}(\texttt{FILTER}_{\mathbb{T}}(t, be, \mathbb{K}))$ or $\langle\sigma, k\rangle \in \gamma_{\mathbb{T}}(\texttt{FILTER}_{\mathbb{T}}(t, \neg be, \mathbb{K}))$ due to the fact that these Algorithms invoke $\texttt{FILTER}_{\mathbb{D}_{Var\cup\mathbb{F}}}$ for every leaf node of t that may be merged with linear constraints from decision nodes found on the path from the root to that leaf. Thus, by structural induction, we have $\langle\sigma', k'\rangle \in \gamma_{\mathbb{T}}([\![s_1]\!]^\natural\texttt{FILTER}_{\mathbb{T}}(t, be, \mathbb{K}))$ or $\langle\sigma', k'\rangle \in \gamma_{\mathbb{T}}([\![s_2]\!]^\natural\texttt{FILTER}_{\mathbb{T}}(t, \neg be, \mathbb{K}))$, and so $\langle\sigma', k'\rangle \in \gamma_{\mathbb{T}}([\![s_1]\!]^\natural\texttt{FILTER}_{\mathbb{T}}(t, be, \mathbb{K}) \sqcup_{\mathbb{T}} [\![s_2]\!]^\natural\texttt{FILTER}_{\mathbb{T}}(t, \neg be, \mathbb{K}))$. Thus, we conclude that $[\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]\gamma_{\mathbb{T}}(t) \subseteq \gamma_{\mathbb{T}}([\![s_1]\!]^\natural\texttt{FILTER}_{\mathbb{T}}(t, be, \mathbb{K}) \sqcup_{\mathbb{T}} [\![s_2]\!]^\natural\texttt{FILTER}_{\mathbb{T}}(t, \neg be, \mathbb{K})) = \gamma_{\mathbb{T}}([\![\texttt{if } be \texttt{ then } s_1 \texttt{ else } s_2]\!]^\natural t)$.

Case $\texttt{while } e \texttt{ do } s$. We show that, given a $t \in \mathbb{T}$, for all $x \in \mathbb{T}$, we have: $\phi(\gamma_{\mathbb{T}}(x)) \subseteq \gamma_{\mathbb{T}}(\phi^\natural(x))$. By structural induction, we have $[\![s]\!]\gamma_{\mathbb{T}}(x) \subseteq \gamma_{\mathbb{T}}([\![s]\!]^\natural x)$.

[2] Note that $\gamma_{\mathbb{T}}(t_{in}) = \langle\Sigma, \mathbb{K}\rangle$.



■ **Figure 7** Invariant at loc. 5 of P''.



■ **Figure 8** Invariant at loc. 9 of P''.

Let $\langle \sigma, k\rangle \in \gamma_{\mathbb{T}}(x)$ and $\langle \sigma', k'\rangle \in \phi(\gamma_{\mathbb{T}}(x))$. By definition of $\phi(x)$ in Fig. 4, we have that $\langle \sigma', k'\rangle \in [\![s]\!]\{\langle \sigma, k\rangle\}$ and true $\in [\![be]\!]\langle \sigma, k\rangle$. By definition of the abstraction $\langle \mathcal{P}(\Sigma \times \mathbb{K}), \subseteq\rangle \xleftarrow{\gamma_{\mathbb{D}}} \langle \mathbb{D}_{Var\cup\mathbb{F}}, \sqsubseteq_{\mathbb{D}}\rangle$, the soundness of $\texttt{FILTER}_{\mathbb{D}_{Var\cup\mathbb{F}}}$, and by definition of $\texttt{FILTER}_{\mathbb{T}}$ (cf. Algorithms 2, 4, and 8), it must hold that $\langle \sigma, k\rangle \in \gamma_{\mathbb{T}}(\texttt{FILTER}_{\mathbb{T}}(x, be, \mathbb{K}))$ by using similar arguments to '`if`' case. Thus, by structural induction, we have $\langle \sigma', k'\rangle \in \gamma_{\mathbb{T}}([\![s]\!]^{\natural}\texttt{FILTER}_{\mathbb{T}}(x, be, \mathbb{K}))$, and so $\langle \sigma', k'\rangle \in \gamma_{\mathbb{T}}(\phi^{\natural}(x))$. We conclude $\phi(\gamma_{\mathbb{T}}(x)) \subseteq \gamma_{\mathbb{T}}(\phi^{\natural}(x))$. The proof that $[\![\texttt{while}\ e\ \texttt{do}\ s]\!]\gamma_{\mathbb{T}}(t) \subseteq \gamma_{\mathbb{T}}([\![\texttt{while}\ e\ \texttt{do}\ s]\!]^{\natural}(t))$ follows from the definition of $\nabla_{\mathbb{T}}$ (cf. Algorithm 9) that invokes the sound $\nabla_{\mathbb{D}_{Var\cup\mathbb{F}}}$ operator on leaf nodes. ◀

▶ **Example 8.** Let us consider the following dynamic program family P'':

```
(1) A := [10, 15];
(2) int x := 10, y;
(3) if (A>12) then y := 1 else y := −1;
(4) while (5) (x  > 0) {
(6)     A := A+y;
(7)     x := x-1;
(8) } (9)
```

which contains one feature `A` with domain [0,99]. Initially, `A` can have a value from [10,15]. We can calculate the abstract invariant semantics $[\![P'']\!]^{\natural}$, thus obtaining invariants from $\mathbb{T}$ in all locations. We show the inferred invariants in locations 5 and 9 in Figs. 7 and 8, respectively. The decision tree at the final location 9 shows that we have $\texttt{x=}0 \wedge \texttt{y=}1$ when $23 \leq \texttt{A} \leq 25$ and $\texttt{x=}0 \wedge \texttt{y=-}1$ when $0 \leq \texttt{A} \leq 2$ on program exit. On the other hand, if we analyze P'' using single-program polyhedra analysis, where `A` is considered as an ordinary program variable, we obtain the following less precise invariant on program exit: $\texttt{x=}0 \wedge -1 \leq \texttt{y} \leq 1 \wedge 5 \leq 2\texttt{A} - 5\texttt{y} \leq 45$. ◀

5 Evaluation

We evaluate our decision tree-based approach for analyzing dynamic program families by comparing it with the single-program analysis approach, in which dynamic program families are considered as single programs and features as ordinary program variables. The evaluation aims to show that our decision tree-based approach can effectively analyze dynamic program families and that it achieves a good precision/cost tradeoff with respect to the single-program analysis. Specifically, we ask the following research questions:

RQ1: How precise are inferred invariants of our decision tree-based approach compared to single-program analysis?
RQ2: How time efficient is our decision tree-based approach compared to single-program analysis?
RQ3: Can we find practical application scenarios of using our approach to effectively analyze dynamic program families?

Implementation

We have developed a prototype lifted static analyzer, called DSPLNum^{2}Analyzer, which uses the lifted domain of decision trees $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$. The abstract operations and transfer functions of the numerical domain $\mathbb{D}$ (e.g., intervals, octagons, and polyhedra) are provided by the APRON library [33]. Our proof-of-concept implementation is written in OCaml and consists of around 8K lines of code. The current front-end of the tool provides only a limited support for arrays, pointers, recursion, `struct` and `union` types, though an extension is possible. The only basic data type is mathematical integers, which is sufficient for our purposes. DSPLNum^{2}Analyzer automatically computes a decision tree from the lifted domain in every program location. The analysis proceeds by structural induction on the program syntax, iterating `while`-s until a fixed point is reached. We apply delayed widening [13], which means that we start extrapolating by widening only after some fixed number of iterations we explicitly analyze the loop's body. The precision of the obtained invariants for `while`-s is further improved by applying the narrowing operator [13]. We can tune the precision and time efficiency of the analysis by choosing the underlying numerical abstract domain (intervals, octagons, polyhedra), and by adjusting the widening delay. The precision of domains increases from intervals to polyhedra, but so does the computational complexity.

Experimental setup and Benchmarks

All experiments are executed on a 64-bit Intel®Core^{TM} i7-8700 CPU@3.20GHz $\times$ 12, Ubuntu 18.04.5 LTS, with 8 GB memory. All times are reported as averages over five independent executions. The implementation, benchmarks, and all results obtained from our experiments are available from [20]: `https://zenodo.org/record/4718697#.YJrDzagzbIU`. We use three instances of our lifted analyses via decision trees: $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, which use intervals, octagons, and polyhedra domains as parameters. We compare our approach with three instances of the single-program analysis based on numerical domains from the APRON library [33]: $\mathcal{A}(I)$, $\mathcal{A}(O)$, and $\mathcal{A}(P)$, which use intervals, octagons, and polyhedra domains, respectively. The default widening delay is 2.

The evaluation is performed on a dozen of C numerical programs collected from several categories of the 9th International Competition on Software Verification (SV-COMP 2020) [3]: `product lines`, `loops`, `loop-invgen` (`invgen` for short), `loop-lit` (`lit` for short), and `termination-crafted` (`crafted` for short). In the case of `product lines`, we selected the e-mail system [26], which has been used before to assess product-line verification in the product-line community [2, 3, 48]. The e-mail system has eight features: encryption, decryption, automatic forwarding, e-mail signatures, auto responder, keys, verify, and address book, which can be activated or deactivated at run-time. There are forty valid configurations that can be derived. For the other categories, we have first selected some numerical programs, and then we have considered some of their integer variables as features. Basically, we selected

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

Table 1 Performance results for single analysis $\mathcal{A}(I)$ vs. lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one and two features on selected e-mail variant simulators. All times are in seconds.

Benchmark	LOC	$\mathcal{A}(I)$, 0 feature			$\overline{\mathcal{A}}_{\mathbb{T}}(I)$, 1 feature			$\overline{\mathcal{A}}_{\mathbb{T}}(I)$, 2 features		
		Time	Unrea.	Rea.	Time	Unrea.	Mix	Time	Unrea.	Mix
`e-mail_spec0`	2645	16.2	80	48	29.3	80	48(1:1)	50.7	80	48(3:1)
`e-mail_spec6`	2660	18.8	6	26	23.6	16	16(1:1)	24.2	16	16(3:1)
`e-mail_spec8`	2665	14.6	12	20	19.1	12	20(1:1)	27.7	12	20(2:2)
`e-mail_spec11`	2660	15.2	160	96	24.7	160	96(1:1)	32.1	160	96(3:1)
`e-mail_spec27`	2630	14.5	384	128	28.4	384	128(1:1)	38.4	384	128(3:1)

those program variables as features that control configuration decisions and can influence the outcome of the given assertions. Tables 1 and 2 present characteristics of the selected benchmarks in our empirical study, such as: the file name (Benchmark), the category where it is located (folder), number of features ($|\mathbb{F}|$), total number of lines of code (LOC).

We use the analyses $\mathcal{A}(\mathbb{D})$ and $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ to evaluate the validity of assertions in the selected benchmarks. 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 $d \sqsubseteq_{\mathbb{D}} \mathrm{FILTER}_{\mathbb{D}}(d, be)$, meaning that the assertion is indeed valid regardless of approximations; (3) '*erroneous*' (invalid), if $d \sqsubseteq_{\mathbb{D}} \mathrm{FILTER}_{\mathbb{D}}(d, \neg be)$, 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 the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$, we may also obtain *mixed* assertions when different leaf nodes of the resulting decision trees yield different answers.

Results

E-mail system. We use a *variant simulator* that has been generated with *variability encoding* from the e-mail configurable system [26]. Variability encoding is a process of encoding compile-time (static) variability of a configurable system as run-time (dynamic) variability in the variant simulator [48, 32]. In this setting, compile-time features are encoded with global program variables, and static configuration choices (e.g., `#if`-s) are encoded with conditional statements in the target language (`if` statements). We consider five specifications of the e-mail system encoded as assertions in SV-COMP. As variant simulators use standard language constructs to express variability (`if` statements), they can be analyzed by standard single-program analyzers $\mathcal{A}(\mathbb{D})$. We also analyze the variant simulators using our lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$, where some of the feature variables are considered as real features. This way, our aim is to obtain more precise analysis results. For effectiveness, we consider only those feature variables that influence directly the specification as real features. Specifically, we consider variant simulators with one and two separate features, and five specifications: `spec0`, `spec6`, `spec8`, `spec11`, and `spec27`. For example, `spec0` checks whether a message to be forwarded is readable, while `spec27` checks whether the public key of a person who sent a message is available. For each specification, many assertions appear in the main function after inlining.

Table 1 shows the results of analyzing the selected e-mail simulators 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 Unrea., 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 Unrea., 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. Note that feature variables in variant simulators are non-deterministically initialized at the beginning of the program and then can be only read in guards of `if` statements, thus $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ may only find more precise answers than $\mathcal{A}(I)$ with respect to the reachability of assertions. That is, it may find more assertions that are unreachable in various variants. See the following paragraph 'Other benchmarks' for examples where 'I don't know' answers by $\mathcal{A}(I)$ are turned into definite ('correct'/'erroneous') answers by $\overline{\mathcal{A}}_{\mathbb{T}}(I)$. We can see in Table 1 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 128 'I don't know' assertions for `spec27`, while $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with one feature `Keys` finds 128 (1:1) mixed assertions such that each assertion is 'unreachable' when `Keys`=0 and 'I don't know' when `Keys`=1. By using $\overline{\mathcal{A}}_{\mathbb{T}}(I)$ with two features `Keys` and `Forward`, we obtain 128 (3:1) mixed assertions, with each assertion is 'unreachable' when $\texttt{Keys} = 0 \vee \texttt{Forward} = 0$. 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 2.2 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 configurations.

Other benchmarks. We now present the performance results for the benchmarks from other SV-COMP categories. The program `half_2.c` from `loop-invgen` category is given in Fig. 9a. When we perform a single-program analysis $\mathcal{A}(P)$, we obtain the 'I don't know' answer for the assertion. However, if n is considered as a feature and the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ is performed on the resulting dynamic program family, we yield that the assertion is: 'correct' when $n \geq 1$, 'erroneous' when $n \leq -2$, and 'I don't know' answer otherwise. We observe that the lifted analysis considers two different behaviors of `half_2.c` separately: the first when the loops are executed one or more times, and the second when the loops are not executed at all. Hence, we obtain definite answers, 'correct' and 'erroneous', for the two behaviors. The program `seq.c` from `loop-invgen` category is given in Fig. 9b. When `seq.c` is analyzed using $\mathcal{A}(P)$, we obtain 'I don't know' for the assertion. When $n0$ and $n1$ are considered as features with the domains $[-Max, +Max]$, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ gives more precise results for the assertion. In particular, the assertion is 'correct' when $(1 \leq n0 \leq Max \wedge 1 \leq n1 \leq Max)$ or $(-Max \leq n0 \leq 0 \wedge -Max \leq n1 \leq 0)$, whereas the assertion is 'erroneous' when $(n0 + n1 \leq 0 \wedge (n0 \geq 1 \vee n1 \geq 1))$ and we obtain 'I don't know' when $(n0 + n1 \geq 1 \wedge (n0 \leq 0 \vee n1 \leq 0))$. The program `sum01_bug02.c` from `loops` is given in Fig. 9c. $\mathcal{A}(P)$ reports 'I don't know' for the assertion, while $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, when n is a feature with domain $[0, Max]$, reports more precise answers: 'erroneous' when $n \geq 9$, 'correct' when $n = 0$, and 'I don't know' otherwise. $\mathcal{A}(P)$ reports 'I don't know' for the assertion in `count_up_down*.c` from `loops`, which is given in Fig. 9d. Still, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ when n is a feature with domain $[-Max, Max]$ reports: 'correct' answer when $n = 0$ at the final location, 'erroneous' when $n \leq -1$, and 'I don't know' otherwise. Similarly, $\mathcal{A}(P)$ reports 'I don't know' for the assertions in `hhk2008.c` and `gsv2008.c` from `loop-lit` (given in Figs. 9e and 9f). However, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ reports more precise answers in both cases. We consider *res* and *cnt* (resp., x) as features with domains $[-Max, Max]$ for `hhk2008.c` (resp., `gsv2008.c`), and

```
n:=[-Max,Max];
int k:=n, i:=0;
while (i<n) {
    k := k-1;
    i := i+2; }
int j:=0;
while (j<n/2) {
    k := k-1;
    j := j+1; }
assert(k≥−1);
```

(a) *half_2.c*

```
n0:=[-Max,Max];
n1:=[-Max,Max];
int i0:=0, k=0;
while (i0<n0) {
    i0 := i0+1;
    k := k+1; }
int i1:=0;
while (i1<n1) {
    i1 := i1+1;
    k := k+1; }
int j1:=0;
while (j1<n0+n1) {
    j1 := j1+1;
    k := k-1; }
assert(k==0);
```

(b) *seq.c*

```
n:=[0,Max];
int a := 2;
int i, j:=10, sn=0;
for (i=1; i ≤ n; i++) {
    if (j>n) then
        sn := sn+a;
    j := j-1;
}
assert (sn ==  n*a);
```

(c) *sum01_bug02.c*

```
n:=[-Max,Max];
int x := n;
int y=0;
while (n>0) {
    n := n-1;
    y := y+1; }
}
assert (y ==  x);
```

(d) *count_up_down*.c*

```
res:=[-Max,Max];
cnt:=[-Max,Max];
int a:=res, b:=cnt;
while (cnt>0) {
    cnt := cnt-1;
    res := res+1; }
assert(res==a+b);
```

(e) *hhk2008.c*

```
x:=[-Max,Max];
x:=-50;
int y:=[-9,9];
while (x<0) {
    x := x+y;
    y := y+1; }
assert(y≤60+x);
```

(f) *gsv2008.c*

```
c:=[-Max,Max];
int x := [−Max, Max];
if (c ≥ 2) then {
  while (x+c ≥ 2) {
      x := x-c;
      c := c+1; }
}
assert (x ≤ −3);
```

(g) *Mysore.c*

```
x:=[-Max,Max];
y:=[-Max,Max];
int oldx;
while (x ≥ 0 ∧ y ≥ 0) }
    oldx := x;
    x := y-1; }
    y := oldx-1; }
assert (x+y ≤ 0);
```

(h) *Copenhagen.c*

Figure 9 Benchmarks from SV-COMP. All underlined variables are considered as features in the corresponding dynamic program families.

we obtain 'correct' answer when $cnt = 0$ for `hhk2008.c` (resp., when $x \geq 0$ for `gsv2008.c`), 'erroneous' answer when $cnt \leq -1$ for `hhk2008.c`, and 'I don't know' answer otherwise. Finally, $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ reports more precise answers than $\mathcal{A}(P)$ for `Mysore.c` and `Copenhagen.c` from `termination crafted` category (given in Figs. 9g and 9h).

Although for all benchmarks $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ infers more precise invariants, still $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ also takes more time than $\mathcal{A}(P)$, as expected. On our benchmarks, this translates to slow-downs (i.e., $\mathcal{A}(P)$ vs. $\overline{\mathcal{A}}_{\mathbb{T}}(P)$) of 4.9 times in average when $|\mathbb{F}| = 1$, and of 6.9 times in average when $|\mathbb{F}| = 2$. However, in some cases the more efficient version $\overline{\mathcal{A}}_{\mathbb{T}}(O)$, which uses octagons, can also provide more precise results than $\mathcal{A}(P)$. For example, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$ for `half_2.c` gives the precise 'erroneous' answer like $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ but gives 'I don't know' in all other cases, whereas $\overline{\mathcal{A}}_{\mathbb{T}}(O)$ for `count_up_down*.c` gives the precise 'erroneous' and 'unreachable' answers like $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ but it turns the 'correct' answer from $\overline{\mathcal{A}}_{\mathbb{T}}(P)$ into an 'I don't know'. On the other hand, for `gsv2008.c` and `Mysore.c`, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$ gives the same precise answers as $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, but twice faster. Furthermore, for `sum01*.c`, even $\overline{\mathcal{A}}_{\mathbb{T}}(I)$, which uses intervals, gives the same precise answers like $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, but with the similar time performance as $\mathcal{A}(P)$. Table 2 shows the running times of $\mathcal{A}(P)$, $\overline{\mathcal{A}}_{\mathbb{T}}(O)$, and $\overline{\mathcal{A}}_{\mathbb{T}}(P)$, as well as whether the corresponding analysis precisely evaluates the given assertion – denoted by Ans. (we use ✓ for yes, ≃ for partially yes, and × for no).

Table 2 Performance results for single analysis $\mathcal{A}(\mathbb{D})$ vs. lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ and $\overline{\mathcal{A}}_{\mathbb{T}}(O)$ on selected benchmarks from SV-COMP. All times are in seconds.

Benchmark	folder	$\lvert\mathbb{F}\rvert$	LOC	$\mathcal{A}(P)$		$\overline{\mathcal{A}}_{\mathbb{T}}(O)$		$\overline{\mathcal{A}}_{\mathbb{T}}(P)$	
				Time	Ans.	Time	Ans.	Time	Ans.
`half_2.c`	`invgen`	1	25	0.008	×	0.014	≃	0.017	✓
`seq.c`	`invgen`	2	30	0.015	×	0.084	✓	0.045	✓
`sum01*.c`	`loops`	1	15	0.008	×	0.009	✓	0.041	✓
`count_up_d*.c`	`loops`	1	15	0.002	×	0.008	≃	0.011	✓
`hhk2008.c`	`lit`	2	20	0.003	×	0.073	≃	0.032	✓
`gsv2008.c`	`lit`	1	20	0.002	×	0.007	✓	0.015	✓
`Mysore.c`	`crafted`	1	30	0.0008	×	0.002	✓	0.004	✓
`Copenhagen.c`	`crafted`	2	30	0.002	×	0.012	≃	0.021	✓

Discussion

Our experiments demonstrate that the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ is able to infer more precise numerical invariants than the single-program analysis $\mathcal{A}(\mathbb{D})$ while maintaining scalability (addresses **RQ1**). As the result of more complex abstract operations and transfer functions of the decision tree domain, we observe slower running times of $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ as compared to $\mathcal{A}(\mathbb{D})$. However, this is an acceptable precision/cost tradeoff, since the more precise numerical invariants inferred by $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ enables us to successfully answer many interesting assertions in all considered benchmarks (addresses **RQ2** and **RQ3**). Furthermore, our current tool is only a prototype implementation to experimentally confirm the suitability of our approach. Many abstract operations and transfer functions of the lifted domain can be further optimized, thus making the performances of the tool to improve.

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. If a real-world tool based on abstract interpretation (e.g. ASTREE [14]) becomes freely available, we can easily transfer our implementation to it.

6 Related Work

Decision-tree abstract domains have been a topic of research in the field of abstract interpretation in recent times [25, 15, 9, 46]. Decision trees have been applied for the disjunctive refinement of interval (boxes) domain [25]. That is, each element of the new domain is a propositional formula over interval linear constraints. Decision tree abstract domains has also been used to enable path dependent static analysis [15, 9] by handling disjunctive analysis properties. Binary decision tree domains [9] can express disjunctive properties depending on the boolean values of the branch (`if`) conditions (represented in decision nodes) with sharing of the properties of the other variables (represented in leaf nodes). Segmented decision tree abstract domains [15] are generalizations of binary decision tree domains and array segmentation, where the choices in decision nodes are made on the values of decision variables according to the ranges specified by a symbolic segmentation. A pre-analysis is used to find decision variables and their symbolic segmentation. The choices for a given decision variable are made only once along a given path. The decision tree lifted domain proposed here can be considered as a generalization of the segmented decision tree domain, where the choices for a given feature variable can be made several times along a given path and arbitrary linear constraints over feature variables can be used to represent the choices in decision nodes. Moreover, linear constraints labelling decision nodes here are semantically inferred during the static analysis and do not necessarily syntactically appear in the code. Urban and Mine [46] 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 APRON library has been developed by Jeannet and Mine [33] to support the application of numerical abstract domains in static analysis. The ELINA library [44] represents an another efficient implementation of numerical abstract domains.

Several *lifted analyses based on abstract interpretation* have been proposed recently [36, 23, 18, 19, 21] for analyzing traditional program families with `#ifdef`-s. A formal methodology for derivation of tuple-based lifted analyses from existing single-program analyses phrased in the abstract interpretation framework has been proposed by Midtgaard et. al. [36]. They use a lifted domain that is a $|\mathbb{K}|$-fold product of an existing single-program domain. That is, the elements of the lifted domain are tuples that contain one separate component for each configuration of $\mathbb{K}$. A more efficient lifted analysis by abstract interpretation obtained by improving representation via BDD-based lifted domains is proposed by Dimovski [18, 19]. The elements of the lifted domain are BDDs, in which decision nodes are labelled with Boolean features and leaf nodes belong to an existing single-program domain. BDDs offer more possibilities for sharing and interaction between analysis properties corresponding to different configurations. The above lifted analyses are applied to program families with only Boolean features. The work [21] extends prior approaches by using decision tree-based lifted domain for analyzing program families with numerical features. In this case, the elements of the lifted domain are decision trees, in which decision nodes are labelled with linear constraints over numerical features and leaf nodes belong to an existing single-program domain. This domain is also successfully applied to program synthesis for resolving program sketches [22]. Several other efficient implementations of the lifted dataflow analysis from *the monotone framework* (a-la Kildall) [35] have also been proposed in the SPL community. Brabrand et al. [5] have introduced a tuple-based lifted dataflow analysis, whereas an approach based on using variational data structures (e.g., variational CFGs, variational data-flow facts) [47] have been used for achieving efficient dataflow computation of some real-world systems. Finally, SPL$^{\text{LIFT}}$ [4] is an implementation of the lifted dataflow analysis formulated within the IFDS framework, which is a subset of dataflow analyses with certain properties, such as distributivity of transfer functions.

Dynamic program families (DSPLs) have been introduced by Hallsteinsen et al. [28] in 2008 as a technique that uses the principles of traditional SPLs to build variants adaptable at run-time. Since then, the research on DSPLs has been mainly focussed on developing mechanisms for implementing DSPLs and for defining suitable feature models.

There are many strategies for implementing variability in traditional SPLs, such as: annotative approach via the C-preprocessor's `#ifdef` construct [34], compositional approach via the feature-oriented programming (FOP) [40] and the delta-oriented programming (DOP) [43], etc. The extensions of FOP and DOP to support run-time reconfiguration and software evolution as found in DSPLs has been proposed by Rosenmuller et al. [42] and Damiani et al [17]. In this work, we extend the annotative approach via `#ifdef`-s to implement variability in DSPLs. The set of valid configurations $\mathbb{K}$ of a program family with Boolean and numerical features is typically described by a *numerical feature model*, which represents a tree-like structure that describes which combinations of feature's values and relationships among them are valid. Several works address the need to change the structural variability (feature model) at run-time. One approach [30] relies on the Common Variability Language (CVL) as an attempt for modelling variability transformations by allowing different types of substitutions to re-configure new versions of base models. Cetina et al. [8] also propose several strategies for modelling runtime transformations using CVL. Helleboogh et al. [31] use a meta-variability model to support dynamic feature models, where high-level constructs enable the addition and removal of variants on-the-fly to the base feature model. In this work, we disregard syntactic representations of the set $\mathbb{K}$ as feature model, as we are concerned with behavioural analysis of program families rather than with implementation details of $\mathbb{K}$. Therefore, we use the set-theoretic view of $\mathbb{K}$ that is syntactically fixed a priori. This is convenient for our purpose here. To the best of our knowledge, our work is pioneering in studying specifically designed behavioral analysis of dynamic program families.

7 Conclusion

In this work, we employ decision trees and widely-known numerical abstract domains for the automatic analysis of C program families that contain dynamically bound features. This way, we obtain a decision tree lifted domain for handling dynamic program families with numerical features. Based on a number of experiments on benchmarks from SV-COMP, we have shown that our lifted analysis is effective and performs well on a wide variety of cases by achieving a good precision/cots tradeoff. The lifted domain $\mathbb{T}(\mathbb{C}_{\mathbb{D}}, \mathbb{D})$ is very expressive since it can express weak forms of disjunctions arising from feature-based constructs.

In the future, we would like to extend the lifted abstract domain to also support non-linear constraints, such as congruences and non-linear functions (e.g. polynomials, exponentials) [6]. Note that the lifted analysis $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$ reports constraints defined over features for which a given assertion is valid, fails, or unreachable. The found constraints take into account the value of features at the location before the given assertion. By using a backward lifted analysis [24, 38], which propagates backwards the found constraints by $\overline{\mathcal{A}}_{\mathbb{T}}(\mathbb{D})$, we can infer the necessary preconditions (defined over features) in the initial state that will guarantee the assertion is always valid, fails, or unreachable.

References

1. Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.
2. Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. Detection of feature interactions using feature-aware verification. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 372–375, 2011. URL: `http://dx.doi.org/10.1109/ASE.2011.6100075`, `doi:10.1109/ASE.2011.6100075`.
3. Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. Strategies for product-line verification: case studies and experiments. In *35th International Conference on Software Engineering, ICSE '13*, pages 482–491, 2013.
4. Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spl$^{\text{lift}}$: statically analyzing software product lines in minutes instead of years. In *ACM SIGPLAN Conference on PLDI '13*, pages 355–364, 2013.

5 Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. *T. Aspect-Oriented Software Development*, 10:73–108, 2013.

6 Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. The polyranking principle. In *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, Proceedings*, volume 3580 of *LNCS*, pages 1349–1361. Springer, 2005. doi:10.1007/11523468_109.

7 Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz Cortés, and Mike Hinchey. An overview of dynamic software product line architectures and techniques: Observations from research and industry. *J. Syst. Softw.*, 91:3–23, 2014. doi:10.1016/j.jss.2013.12.038.

8 Carlos Cetina, Øystein Haugen, Xiaorui Zhang, Franck Fleurey, and Vicente Pelechano. Strategies for variability transformation at run-time. In *Software Product Lines, 13th International Conference, SPLC 2009, Proceedings*, volume 446 of *ACM International Conference Proceeding Series*, pages 61–70. ACM, 2009. URL: https://dl.acm.org/citation.cfm?id=1753245.

9 Junjie Chen and Patrick Cousot. A binary decision tree abstract domain functor. In *Static Analysis - 22nd International Symposium, SAS 2015, Proceedings*, volume 9291 of *LNCS*, pages 36–53. Springer, 2015. doi:10.1007/978-3-662-48288-9_3.

10 Andreas Classen, Arnaud Hubaux, and Patrick Heymans. A formal semantics for multi-level staged configuration. In *Third International Workshop on Variability Modelling of Software-Intensive Systems, Seville, Spain, January 28-30, 2009. Proceedings*, volume 29 of *ICB Research Report*, pages 51–60. Universität Duisburg-Essen, 2009. URL: http://www.vamos-workshop.net/proceedings/VaMoS_2009_Proceedings.pdf.

11 Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

12 Patrick Cousot and Radhia 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*, pages 238–252. ACM, 1977. URL: http://doi.acm.org/10.1145/512950.512973, doi:10.1145/512950.512973.

13 Patrick Cousot and Radhia 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*, volume 631 of *LNCS*, pages 269–295. Springer, 1992. doi:10.1007/3-540-55844-6_142.

14 Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005, Proceedings*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005. doi:10.1007/978-3-540-31987-0_3.

15 Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. A scalable segmented decision tree abstract domain. In *Time for Verification, Essays in Memory of Amir Pnueli*, volume 6200 of *LNCS*, pages 72–95. Springer, 2010. doi:10.1007/978-3-642-13754-9_5.

16 Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96. ACM Press, 1978. doi:10.1145/512760.512770.

17 Ferruccio Damiani, Luca Padovani, and Ina Schaefer. A formal foundation for dynamic delta-oriented software product lines. In *Generative Programming and Component Engineering, GPCE'12*, pages 1–10. ACM, 2012. doi:10.1145/2371401.2371403.

18 Aleksandar 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*, pages 102–114. ACM, 2019. doi:10.1145/3357765.3359518.

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

20 Aleksandar S. Dimovski and Sven Apel. Tool artifact for "lifted static analysis of dynamic program families by abstract interpretation". *Zenodo*, 2021. URL: `https://zenodo.org/record/4718697#.YJrDzagzbIU`, `doi:10.5281/zenodo.4718697`.

21 Aleksandar S. Dimovski, Sven Apel, and Axel 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*, volume 12649 of *LNCS*, pages 67–86. Springer, 2021. `doi:10.1007/978-3-030-71500-7\_4`.

22 Aleksandar S. Dimovski, Sven Apel, and Axel Legay. Program sketching using lifted analysis for numerical program families. In *NASA Formal Methods - 13th International Symposium, NFM 2021, Proceedings*, volume 12673 of *LNCS*. Springer, 2021.

23 Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Variability abstractions: Trading precision for speed in family-based analyses. In *29th European Conference on Object-Oriented Programming, ECOOP 2015*, volume 37 of *LIPIcs*, pages 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. URL: `http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.247`, `doi:10.4230/LIPIcs.ECOOP.2015.247`.

24 Aleksandar S. Dimovski and Axel Legay. Computing program reliability using forward-backward precondition analysis and model counting. In *Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020, Proceedings*, volume 12076 of *LNCS*, pages 182–202. Springer, 2020. `doi:10.1007/978-3-030-45234-6\_9`.

25 Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In *Static Analysis - 17th International Symposium, SAS 2010. Proceedings*, volume 6337 of *LNCS*, pages 287–303. Springer, 2010. `doi:10.1007/978-3-642-15769-1\_18`.

26 Robert J. Hall. Fundamental nonmodularity in electronic mail. *Automated Software Engineering*, 12(1):41–79, 2005. `doi:10.1023/B:AUSE.0000049208.84702.84`.

27 Svein O. Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. 2nd international workshop on dynamic software product lines DSPL 2008. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, page 381. IEEE Computer Society, 2008. `doi:10.1109/SPLC.2008.69`.

28 Svein O. Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008. `doi:10.1109/MC.2008.123`.

29 Svein O. Hallsteinsen, Erlend Stav, Arnor Solberg, and Jacqueline Floch. Using product line techniques to build adaptive systems. In *Software Product Lines, 10th International Conference, SPLC 2006, Baltimore, Maryland, USA, August 21-24, 2006, Proceedings*, pages 141–150. IEEE Computer Society, 2006. `doi:10.1109/SPLINE.2006.1691586`.

30 Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *Software Product Lines, 12th International Conference, SPLC 2008, Proceedings*, pages 139–148. IEEE Computer Society, 2008. `doi:10.1109/SPLC.2008.25`.

31 Alexander Helleboogh, Danny Weyns, Klaus Schmid, Tom Holvoet, Kurt Schelfthout, and Wim Van Betsbrugge. Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines. In *Synamic Software Product Lines, 3rd International Workshop, SSPL 2009, Proceedings*, 2009.

32 Alexandru F. Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Effective analysis of c programs by rewriting variability. *Programming Journal*, 1(1):1, 2017. `doi:10.22152/programming-journal.org/2017/1/1`.

33 Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, 21st International Conference, CAV*

2009. Proceedings, volume 5643 of *LNCS*, pages 661–667. Springer, 2009. doi:10.1007/978-3-642-02658-4_52.

34 Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0.* PhD thesis, University of Magdeburg, Germany, May 2010.

35 Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, (POPL'73)*, pages 194–206, 1973. doi:10.1145/512927.512945.

36 Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015. URL: http://dx.doi.org/10.1016/j.scico.2015.04.005, doi:10.1016/j.scico.2015.04.005.

37 Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. doi:10.1007/s10990-006-8609-1.

38 Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017. doi:10.1561/2500000034.

39 David Lorge Parnas. On the design and development of program families. *IEEE Trans. Software Eng.*, 2(1):1–9, 1976. doi:10.1109/TSE.1976.233797.

40 Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, 1997, Proceedings*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997. doi:10.1007/BFb0053389.

41 Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. Flexible feature binding in software product lines. *Autom. Softw. Eng.*, 18(2):163–197, 2011. doi:10.1007/s10515-011-0080-5.

42 Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring dynamic software product lines. In *Generative Programming And Component Engineering, Proceedings of the 10th International Conference on Generative Programming and Component Engineering, GPCE 2011*, pages 3–12. ACM, 2011. doi:10.1145/2047862.2047866.

43 Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010. Proceedings*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010. doi:10.1007/978-3-642-15579-6_6.

44 Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2015*, pages 303–313. ACM, 2015. doi:10.1145/2737924.2738000.

45 Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs.* PhD thesis, École Normale Supérieure, Paris, France, 2015. URL: https://tel.archives-ouvertes.fr/tel-01176641.

46 Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In *Static Analysis - 21st International Symposium, SAS 2014. Proceedings*, volume 8723 of *LNCS*, pages 302–318. Springer, 2014. doi:10.1007/978-3-319-10936-7_19.

47 Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. Variability-aware static analysis at scale: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 27(4):18:1–18:33, 2018. doi:10.1145/3280986.

48 Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. Variability encoding: From compile-time to load-time variability. *J. Log. Algebraic Methods Program.*, 85(1):125–145, 2016. doi:10.1016/j.jlamp.2015.06.007.