ПРИЛОЗИ, Одделение за природно-математички и биотехнички науки, МАНУ, том , бр. , стр. 00–00 (2019)
CONTRIBUTIONS, Section of Natural, Mathematical and Biotechnical Sciences, MASA, Vol. **,** No. , pp. 00–00 (2019)

Received:
Accepted:

ISSN
e-ISSN
UDC: 000.000.4$97.7)
DOI: 10.20903/csnmbs.masa.2019.00

Original scientific paper

ON CALCULATING ASSERTION PROBABILITIES FOR PROGRAM FAMILIES

Aleksandar S. Dimovski*

Faculty for Informatics Sciences, “Mother Teresa” University, Skopje, Macedonia

*e-mail: aleksandar.dimovski@unt.edu.mk

Highly configurable software systems (program families) appear in many application areas and for many reasons. They can produce a potentially large variety of related programs (variants) by selecting suitable configuration options (features) at compile time. Many of those configurable software systems can input and manipulate uncertain data.

In this paper, we present an approach that calculates the assertion probabilities of program families with uncertain input data. First, we use a combination of forward and backward family-based (lifted) analyses based on abstract interpretation in order to infer necessary preconditions for a given assertion to be satisfied/violated in all variants of a program family. We use lifted analyses based on binary decision diagrams (BDDs) and numerical abstract domains (e.g., Polyhedral) that infer numerical invariants in every program location. Second, model counting techniques are exploited to count the number of solutions to the discovered necessary preconditions (given in the form of linear constraints) on input stores. We use those counts to estimate the probability that the target assertion is satisfied/violated in all variants individually. We implement our approach in a prototype tool, and we evaluate it on several interesting C program families.

Key words: Static analysis by abstract interpretation, Model counting, Software Product Lines (program families)

INTRODUCTION

Customization becomes increasingly popular in today’s software systems. Many software systems adopt Software Product Line (SPL) methodology [1], where *features* (statically configured options) are used to control presence and absence of software functionality in a program family. Different family members, called *variants* or *valid products*, are derived by switching features on and off, while reuse of the common code is maximized. In fact, the main benefits from using Software Product Lines are: productivity gains, shorter time to market, greater market coverage, etc. Software Product Lines (program families) are commonly seen in development of commercial embedded software, such as in cars, phones, avionics, medicine, robotics, etc. In this case, variation points are used to either support different application scenarios for embedded components, to provide portability across different hardware platforms and configurations, or to produce variations of products for different market segments or different customers. We consider here SPLs implemented using #ifdef directives from the C preprocessor [2]. Many of the above configurable software systems use and manipulate uncertain data. Uncertainty is a common aspect especially for systems that manipulate error-prone data coming from sensors and other external environments. In this case, it is essential to learn how the presence of uncertainty in the input affect the behavior of all valid variants in the family individually.

Probabilistic program analysis [3,4,5] aims to quantify the probability that a given program satisfies a required property. It has many potential applications, from program understanding and debugging to computing program reliability, compiler optimizations and quantitative information flow analysis for security. In these situations, it is usually more relevant to quantify the probability of satisfying/violating a given property than to just assess the possibility of such events to occur. In this work, we describe an efficient method for probabilistic static analysis of program families. In particular, we show how to calculate the assertion probability, and so estimate the program reliability of all variants, by using static analysis based on abstract interpretation and model counting.

Abstract interpretation [6,7] is a unifying theory of sound approximation of structures. It represents a well-established general framework, which provides safe and efficient static analyses of real programs. Still, 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). Therefore, we use so-called family-based (lifted) static analysis [8,9,10] which works on the family level, analyzing all variants of the family simultaneously at once, without generating any of them explicitly. In particular, we consider here a lifted analysis based on a binary decision diagram (BDD) lifted domain [10], where (Boolean) features are organized in decision nodes and leaf nodes contain a particular analysis property. Elements from the BDD domain are used to map the values of Boolean features (represented in decision nodes) to an analysis property (represented in leaf nodes) for the variant specified by the values of features along the path leading to the leave. The efficiency of BDDs comes from the opportunity to share equal subtrees, in case some properties are independent from the value of some features.

The practical success of abstract interpretation is mainly enabled by the design of numerical abstract domains, which reason on numerical properties of program variables. For example, the polyhedral abstract domain [11] infers linear constraints between all program variables in the form: $\alpha_1 x_1 + \cdots + \alpha_k x_k \geq \beta$, where $\alpha_1, \ldots, \alpha_k, \beta \in \mathbb{R}$ (reals), and $x_1, \ldots, x_k$ are program variables. In our case, we use the BDD-based lifted analysis domain, in which the polyhedral domain is used for the leaf nodes. We design two types of static lifted analyses of C program families: a forward analysis to automatically infer invariants in all program locations, and a backward analysis to automatically infer necessary preconditions in all program locations. We combine these two analyses to automatically generate the necessary preconditions on input variables that lead to the satisfaction/violation of a given assertion in a program family. If obtained preconditions are satisfied by some concrete values for input variables, then they represent input values that will allow the given assertion to be satisfied/violated. In fact, we run two backward analyses: the first one determines necessary preconditions for the given assertion to be satisfied, while the second one determines necessary preconditions for the assertion to be violated.

Model counting is the problem of determining the number of solutions of a given constraint (formula). The LATTE tool [12] implements state-of-the-art algorithms for computing volumes, both real and integral, of convex polytopes as well as integrating functions over those polytopes. More specifically, we use the LATTE tool and model counting techniques to estimate algorithmically the exact number of points of a bounded (possibly very large) discrete domain that satisfy given linear constraints.

In this work, we describe a method which uses abstract interpretation-based lifted static analysis and model counting to perform a specific type of quantitative analysis of program families, which is the calculation of *assertion probabilities*. Calculating the probability of a given target assertion involves counting the number of solutions to necessary preconditions that ensure satisfaction/violation of the given assertion for any variant by using model counting, and dividing it by the total space of values of the inputs. We assume that the input values are all *uniformly distributed* within their finite discrete domain. As the set of obtained necessary preconditions represents an over-approximation of the set of exact input values which guarantee that all executions starting from them lead to satisfaction/violation of the given assertion, we calculate upper and lower bounds of exact probabilities that a given assertion is satisfied or violated. The reported uncertainty is due to the approximation inherent in abstract interpretation, which is introduced in order to obtain a scalable and fully automatic analysis.

We have developed a prototype probabilistic lifted analyzer which uses the BDDAPRON library [13] to implement the BDD lifted domain and the LATTE tool [11] to implement model counting algorithms. BDDAPRON uses the polyhedral numerical domain [11] from the APRON library [14] for the leaf nodes. APRON provides a common high-level API to the most common numerical property domains, such as intervals, octagons, and polyhedral. We have implemented a combination of forward and backward lifted analyses of #ifdef-enriched C programs for the automatic inference of invariants and necessary preconditions in all program locations. In this way, we can use the probabilistic lifted analyzer to calculate assertion probabilities of C program families, which represent majority of industrial embedded code. We restrict our attention on program families that have finite input domains and on polyhedral numerical elements expressed as *linear integer arithmetic* (LIA) constraints over variables whose values are *uniformly distributed* over their assigned interval domains.

The following contributions are made in this work:

- We employ the lifted analysis based on BDDs introduced in [10] and model counting [12] to calculate assertion probabilities in all variants of a program family.
- We provide step-by-step example-driven demonstration of how our approach works.
- We implement our approach using the polyhedral numerical domain from the BDDAPRON library [13] and the LATTE model counting tool [12].
- We evaluate our approach for calculating assertion probabilities on a several interesting #ifdef-enriched C programs.

MOTIVATING EXAMPLE

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

```
void main() {
1:       int j := [0,9];
l_input:  int i := 0;
3:       while (i<100) {
4:               i := i+1;
5:               #ifdef (A) j := j+1; #endif
6:               #ifdef (B) j := j+1; #endif
7:       }
l_final:  assert(j<=105); }
```

The set of Boolean features in the above program family P is $\mathbb{F} = \{A, B\}$ and the set of configurations is $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$. The family P contains two #ifdef directives, which increase the variable j by 1, depending on which features from $\mathbb{F}$ are enabled. For each configuration from $\mathbb{K}$ a different variant (single program) can be generated by appropriately resolving #ifdef-s. For example, the variant corresponding to the configuration $A \wedge B$ will have both features A and B enabled (set to true), so that both assignments j := j+1 in program locations 5 and 6 will be included in this variant. On the other hand, the variant for configuration $\neg A \wedge \neg B$ will have both features A and B disabled (set to false), so the above assignments in program locations 5 and 6 will not be included in it. There are $|\mathbb{K}| = 4$ variants in total.

We assume that the initial value of variable j ranges over the integer domain [0, 9], and the chosen initial random value is independently and uniformly distributed across this range. We perform two lifted analyses: a forward invariant and a backward necessary condition, using the BDD lifted domain and the polyhedral numerical domain for the leaf nodes. The forward invariant lifted analysis will find that at the location l_{final} the following invariants hold (see also the result in Fig. 1a): the invariant (200 ≤ j ≤ 209) for the variant $A \wedge B$, the invariant (100 ≤ j ≤ 109) for variants $A \wedge \neg B$ and $\neg A \wedge B$, and the invariant (0 ≤ j ≤ 9) for the variant $\neg A \wedge \neg B$. Therefore, the target assertion (j ≤ 105) is always violated for the variant $A \wedge B$ and it is always satisfied for the variant $\neg A \wedge \neg B$. However, for variants $A \wedge \neg B$ and $\neg A \wedge B$, the assertion can be both satisfied and violated. In those cases, we perform a backward necessary condition lifted analysis for assertion satisfaction/violation, which computes the preconditions on input states such that all executions starting from those states will satisfy/violate the given assertion. The backward necessary condition analysis for variants $A \wedge \neg B$ and $\neg A \wedge B$ will infer the precondition (0 ≤ j ≤ 5) at location l_{input} for the assertion to be satisfied, while the precondition (6 ≤ j ≤ 9) at location l_{input} will be inferred for the assertion to be violated (see the results in Fig. 1b and 1c). The size of the input domain is 10, since j belongs to [0, 9]. By calling the LATTE tool to count the number of solutions to the above preconditions, we can calculate that the probability for the assertion to be satisfied (*success probability*) is less or equal to: 0=0% for the variant $A \wedge B$, 6/10=60% for variants $A \wedge \neg B$ and $\neg A \wedge B$, and 1=100% for the variant $\neg A \wedge \neg B$. On the other hand, the probability for the assertion to be violated (*failure probability*) is less or equal to: 1=100% for the variant $A \wedge B$, 4/10=40% for variants $A \wedge \neg B$ and $\neg A \wedge B$, and 0=0% for the variant $\neg A \wedge \neg B$.

Fig. 1 shows the analysis results given as BDDs obtained using the forward invariant analysis and two backward necessary condition analysis for assertion satisfaction and violation. Note that the inner nodes of BDDs in Fig. 1 are labelled with features from $\mathbb{F}$, the leaves are labelled with the elements from the polyhedral domain, and the edges are labeled with the truth value of the decision on the parent node, true or false (we use solid edges for true, and dashed edges for false). We can see that BDDs offer possibilities for sharing and interaction between analysis properties corresponding to different variants. Thus, they provide symbolic and compact representation of lifted analysis elements. For example, the cases when (A is true and B is false) and (A is false and B is true) are identical, so they share the same leaf nodes in all three BDDs in Fig. 1. Notice that, in the worst case, BDDs still need $|\mathbb{K}|$ different leaf nodes, but experimental evidence shows that sharing often occurs in practice.

PROGRAMMING LANGUAGE

Let $\mathbb{F} = \{A1, \ldots, An\}$ be a finite and totally ordered set of Boolean variables representing the *features* available in a program family. The total ordering of features is: $A1 < \cdots < An$. A specific subset of features, $k \subseteq \mathbb{F}$, known as *configuration*, specifies a variant (valid product) of a program family. We assume that only a subset $\mathbb{K} \subseteq 2^{\mathbb{F}}$ of all possible configurations are *valid*. An alternative representation of configurations is based upon propositional formulae. Each configuration $k \in \mathbb{K}$ can be represented by a formula: $k(A1) \wedge \ldots \wedge k(An)$, where $k(Ai) = Ai$ if $Ai \in k$, and $k(Ai) = \neg Ai$ if $Ai \notin k$ for $1 \leq i \leq n$. We will use both representations interchangeably.

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

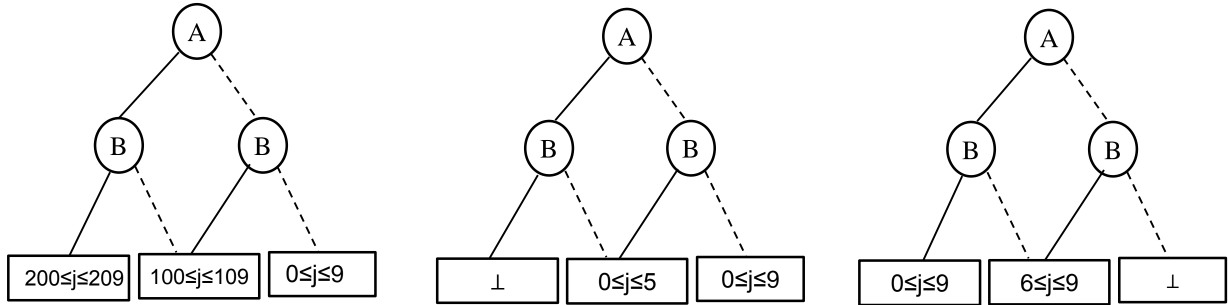
$$\theta ::= true \mid A \in \mathbb{F} \mid \neg\theta \mid \theta 1 \wedge \theta 2$$

We will use $\theta \in$ FeatExp($\mathbb{F}$) to define presence conditions in program families.

We consider a programming language that is a subset of C for writing program families, which will be used to exemplify our work. The language is extended with a compile-time conditional statement for encoding multiple variants of a program. The new statement ``#ifdef (θ) s'' contains a feature expression $\theta \in$ FeatExp($\mathbb{F}$) as a presence condition, such that only if θ is satisfied by a configuration $k \in \mathbb{K}$ then the statement s will be included in the variant

Fig. 1. BDD-based lifted analyses results obtained using a forward invariant analysis and two backward necessary condition analyses for assertion satisfaction and violation. We use solid edges for true, and dashed edges for false.

a) Invariant at point l_{final}. **b)** Precondition for assert satisf. at l_{input}. **c)** Precondition for assert viol. at l_{input}.



corresponding to k. The syntax of the language is:

s ::=skip| x:=e| x:=[n,n'] |s1; s2|if (e) then s1 else s2
| while (e) do s | #ifdef (e) s | assert (e)
e ::= n | x | e1 ⊕ e2

where n ranges over integers, [n, n'] ranges over integer intervals, x ranges over variable names Var, and ⊕ over binary arithmetic-logic operators. Non-deterministic interval assignment x := [n, n'] represents an input statement which assigns to the input variable x an uniformly distributed random integer from the interval [n, n']. The interval assignment can occur only in the *input section* of the program. The set of all generated statements s is denoted by Stm, whereas the set of all expressions e is denoted by Exp. We assume l_{input} is the location after the input statements (i.e. it denotes the end of input section) and l_{final} is the location at the end of the program, where an assertion assert(e_f) is posed. Without loss of generality, a program is a sequence of statements followed by a single assertion.

The program families are evaluated in two stages. First, a preprocessor takes as input a program family and a configuration $k \in \mathbb{K}$, and outputs a variant, i.e. a single program without #ifdef-s, corresponding to k. Second, the obtained variant is evaluated using the standard single-program semantics [14]. The first stage is specified by the projection function $\mathcal{P}_k$, which is an identity for all basic statements of the program family and recursively pre-processes all sub-statements of compound statements. Hence, $\mathcal{P}_k$(skip) = skip and $\mathcal{P}_k$(s; s') = $\mathcal{P}_k$(s) ; $\mathcal{P}_k$(s'). The interesting case is ``#ifdef (θ) s'' statement, where the statement s is included in the resulting variant iff k satisfies θ, otherwise the statement s is removed (i.e. replaced with skip). Thus,

$\mathcal{P}_k$(#ifdef (θ) s) = $\mathcal{P}_k$(s), if k satisfies θ, and

$\mathcal{P}_k$(#ifdef (θ) s) = skip, if k does not satisfy θ

SINGLE-PROGRAM STATIC ANALYSES

In this section, we introduce the combination of forward and backward single-program analyses for inferring necessary preconditions that a given assertion is satisfied/violated. Those preconditions are used for computing the probabilities that the given assertion is satisfied (called *success probability*) or violated (called *failure probability*). For easy presentation, we focus on single programs in this section, and then show how to extend our technique to program families in the next section. We introduce basic theoretical concepts only as required.

Recall the running example program family P from "Motivating example" section. The variant corresponding to the configuration $A \wedge \neg B$ is:

```
1:       int j := [0,9];
l_input: int i := 0;
3:       while (i<100) {
4:               i := i+1;
5:               j := j+1;
6:               skip;
7:       }
l_final: assert(j<=105);
```

We denote it as $\mathcal{P}_{A\wedge\neg B}$(P).

Concrete semantics. The "*state*" of an imperative program is a program control location, together with the values of all variables in that location. The semantics of a single program is given by defining a state transition function $trans$, as follows:

$$State = Program\ location\ \times Store$$
$$Store = Variables\ \rightarrow Value$$
$$trans : State\ \rightarrow State$$

We first introduce a *collecting (concrete) semantics* which associates with each program location the set of all memory stores that can ever occur when program control reaches that location. We assume the program is run on data from a set $\mathbb{E} \in 2^{Store}$ of possible initial input stores. Let l_{input} be the program's input location and let $\boldsymbol{l}$ be another program point. The set of stores that can be reached at location $\boldsymbol{l}$ are defined:

$coll_l = \{s \mid (l,s) = trans^n(l_{input}, s_0), s_0 \in \mathbb{E}, n \geq 0\}$

where $trans^n$ is the n-th iterate of $trans$ (i.e. $trans^n = trans \circ trans^{n-1}$). The collecting semantics thus associates with each program location the set of stores $coll_l \in 2^{Store}$.

For the running example $\mathcal{P}_{A\wedge\neg B}$(P), there are two variables i and j. Thus, a set of stores has the form:

$\{[\mathrm{i} \mapsto v_1, \mathrm{j} \mapsto n_1], [\mathrm{i} \mapsto v_2, \mathrm{j} \mapsto n_2] \ \dots\}$

where $[\mathrm{i} \mapsto v_1, \mathrm{j} \mapsto n_1], [\mathrm{i} \mapsto v_2, \mathrm{j} \mapsto n_2] \in Store$. For notational simplicity, we can identify this set with the set:

$\{[\mathrm{i} \mapsto \{v_1, v_2, \dots\}, \mathrm{j} \mapsto \{n_1, n_2, \dots\}]\}$

Given an initial input set $\mathbb{E} = \{[\mathrm{j} \mapsto \{0, \dots, 9\}]\}$, the set of stores reachable at all program locations are:

$coll_3 = \{[\mathrm{i} \mapsto \{0\}, \mathrm{j} \mapsto \{0, \dots, 9\}]\}$
$coll_4 = \{[\mathrm{i} \mapsto \{0, \dots, 99\}, \mathrm{j} \mapsto \{0, \dots, 108\}]\}$
$coll_5 = \{[\mathrm{i} \mapsto \{1, \dots, 100\}, \mathrm{j} \mapsto \{0, \dots, 108\}]\}$
$coll_6 = \{[\mathrm{i} \mapsto \{1, \dots, 100\}, \mathrm{j} \mapsto \{0, \dots, 109\}]\}$
$coll_7 = \{[\mathrm{i} \mapsto \{1, \dots, 100\}, \mathrm{j} \mapsto \{0, \dots, 109\}]\}$
$coll_{l_{final}} = \{[\mathrm{i} \mapsto \{100\}, \mathrm{j} \mapsto \{100, \dots, 109\}]\}$

The set 2^{Store} of all sets of stores forms a lattice with set inclusion $\subseteq$ as its partial order, so any two store sets A, B have least upper bound $A \cup B$ and greatest lower bound $A \cap B$. The lattice 2^{Store} is complete, meaning that any collection of sets of stores has a least upper bound in 2^{Store}, namely its union.

The above sets of reachable stores $coll_l$ can be obtained as *solution* to the following *forward data-flow equations*, which have a unique least fixpoint solution by completeness of 2^{Store}.

$coll_{l_{input}} = \mathbb{E}$
$coll_3 = coll_{l_{input}} \cap \{[\mathrm{i} \mapsto \{0\}, \mathrm{j} \mapsto \mathbb{N}]\}$
$coll_4 = (coll_3 \cup coll_7) \cap \{\mathrm{i} \mapsto \{0, \dots, 99\}, \mathrm{j} \mapsto \mathbb{N}\}$
$coll_5 = \{[\mathrm{i} \mapsto v+1, \mathrm{j} \mapsto coll_4(j)] \mid v \in coll_4(i)\}$
$coll_6 = \{[\mathrm{i} \mapsto coll_5(i), \mathrm{j} \mapsto n+1)] \mid n \in coll_5(j)\}$
$coll_7 = coll_6$
$coll_{l_{final}} = (coll_3 \cup coll_7) \cap \{[\mathrm{i} \mapsto \{100, \dots\}, \mathrm{j} \mapsto \mathbb{N}]\}$

where $coll_l$(j) denotes the set of values assigned to the variable j at location *l*. From the solution of the above equations for $coll_{l_{final}}(j) = \{100, \dots, 109\}$, we can see that the target assertion (j ≤ 105) can be both satisfied and violated for the variant $A \wedge \neg B$.

In order to determine the success and failure probabilities for the variant $A \wedge \neg B$, we need to calculate a backward collecting semantics, which given a set of final stores finds at each program location the set of stores necessary to imply the given set of final stores at termination. For the running example program, the appropriate backward data-flow equations are:

$b_coll_{l_{input}} = \{[\mathrm{j} \mapsto b_coll_3(\mathrm{j})] \mid 0 \in b_coll_3(\mathrm{i})\}$
$b_coll_3 = (b_coll_{l_{final}} \cap \{[\mathrm{i} \mapsto \{100, \dots\}, \mathrm{j} \mapsto \mathbb{N}]\}) \cup (b_coll_4 \cap \{[\mathrm{i} \mapsto \{0, \dots, 99\}, \mathrm{j} \mapsto \mathbb{N}]\})$
$b_coll_4 = \{[\mathrm{i} \mapsto v, \mathrm{j} \mapsto b_coll_5(\mathrm{j})] \mid v+1 \in b_coll_5(\mathrm{i})\}$
$b_coll_5 = \{[\mathrm{i} \mapsto b_coll_6(\mathrm{i}), \mathrm{j} \mapsto n] \mid n+1 \in b_{coll_6}(\mathrm{j})\}$
$b_coll_6 = b_coll_7$
$b_coll_7 = (b_coll_{l_{final}} \cap \{[\mathrm{i} \mapsto \{100, \dots\}, \mathrm{j} \mapsto \mathbb{N}]\}) \cup (b_coll_4 \cap \{[\mathrm{i} \mapsto \{100, \dots\}, \mathrm{j} \mapsto \mathbb{N}]\})$

where b_coll_l is the set of all stores at location *l* that cause control to reach point l_{final} with a final store $b_coll_{l_{final}}$. Note that, $b_coll_{l_{final}}$ is a parameter for the above equation system. Now, we find solutions of two backward data-flow equations: the first one when $b_coll_{l_{final}} = \{[i \mapsto \mathbb{N}, j \mapsto \{100, \dots, 105\}]\}$ which causes the target assertion to be satisfied; and the second one when $b_coll_{l_{final}} = \{[\mathrm{i} \mapsto \mathbb{N}, \mathrm{j} \mapsto \{106, \dots, 109\}]\}$ which causes the target assertion to be violated. From the solution of the first backward equations, we obtain $b_coll_{l_{input}}(j) = \{0,1,2,3,4,5\}$, that is $\mathbb{E}_{sat} = [\mathrm{j} \mapsto \{0,1,2,3,4,5\}]$. From the solution of the second backward equations, we obtain $b_coll_{l_{input}}(j) = \{6,7,8,9\}$, that is $\mathbb{E}_{viol} = [\mathrm{j} \mapsto \{6,7,8,9\}]$. Therefore, we conclude that the success probability is $Pr^s(\mathcal{P}_{A\wedge\neg B}(\mathrm{P})) = \frac{|\mathbb{E}_{sat}|}{|\mathbb{E}|} = \frac{6}{10} = 60\%$ and the failure probability is $Pr^f(\mathcal{P}_{A\wedge\neg B}(\mathrm{P})) = \frac{|\mathbb{E}_{viol}|}{|\mathbb{E}|} = \frac{4}{10} = 40\%$.

Abstract semantics. The collecting concrete semantics is obviously *incomputable*, due to the insolvability of the halting problem and nearly any other question concerning program behavior. Therefore, we seek for sound approximations. We demonstrate how to derive approximate, but computable analyses, which statically determine dynamic properties of programs. The effect of thus obtained abstract analyses is that the price paid for finite computability is loss of precision. The abstract analyses will infer necessary preconditions on input stores so that all executions starting from those input stores lead to satisfaction (resp., violation) of the final assertion. In this way, we will compute the over-approximations of sets $\mathbb{E}_{sat}$ and $\mathbb{E}_{viol}$, and thus find the lower and upper bounds for Pr^s and Pr^f.

Recall that the collecting semantics works on sets of stores, 2^{Store}. Various approximations can be expressed by simpler domains (lattices) Abs, connected to 2^{Store} by an abstraction function $\alpha: 2^{Store} \longrightarrow Abs$, and a dual concretization function $\gamma: Abs \longrightarrow 2^{Store}$. Here, Abs represents a lattice of approximate descriptions of sets of stores 2^{Store}. The pair of functions α and γ, which capture information loss between two domains (lattices) 2^{Store} and Abs, are required to form a Galois connection [6, 15]. Abstract analysis may thus be thought of as executing the program over a lattice of imprecise but computable abstract store descriptions instead of the precise and incomputable collecting semantics lattice.

There exist various abstract domains, which can be used for automatic discovery of program properties. They differ in expressive power and computational complexity. The most suitable abstract domain for computing necessary preconditions on input stores is the polyhedral numerical domain [11]. The polyhedral domain is a fully relational numerical property domain, which allows manipulating conjunctions of linear inequalities of the form $\alpha_1 x_1 + \cdots + \alpha_k x_k \geq \beta$, where $\alpha_1, \ldots, \alpha_k, \beta \in \mathbb{R}$ (reals), and $x_1, \ldots, x_k$ are program variables. Polyhedral analysis is computationally expensive but very precise.

The polyhedral abstract domain $\mathbb{P}$ is equipped with (over-approximating) sound operators for abstraction $\alpha_{\mathbb{P}}: 2^{Store} \longrightarrow \mathbb{P}$, concretization $\gamma_{\mathbb{P}}: \mathbb{P} \longrightarrow 2^{Store}$, partial ordering $\sqsubseteq_{\mathbb{P}}$, least upper bound (join) $\sqcup_{\mathbb{P}}$, greatest lower bound (meet) $\sqcap_{\mathbb{P}}$, the least element (bottom) $\bot_{\mathbb{P}}$, the greatest element (top) $\top_{\mathbb{P}}$, as well as sound transfer functions for assignments $assign_{\mathbb{P}}: Stm \times \mathbb{P} \rightarrow \mathbb{P}$, tests (which occur in while-s and if-s) $filter_{\mathbb{P}}: Exp \times \mathbb{P} \rightarrow \mathbb{P}$, backward assignments $b_assign_{\mathbb{P}}: Stm \times \mathbb{P} \rightarrow \mathbb{P}$, and backward tests $b_filter_{\mathbb{P}}: Exp \times \mathbb{P} \rightarrow \mathbb{P}$. For example, the transfer function $filter_{\mathbb{P}}(e, p)$ returns an abstract store p' which is restriction of the abstract store p, so that it satisfies the given test (expression) e. The domain $\mathbb{P}$ also contains widening $\triangle_{\mathbb{P}}$ and narrowing $\odot_{\mathbb{P}}$ operators in order to compute an over-approximation of least fixpoints.

We define a system of *approximate (abstract) forward data-flow equations*, which describe the program's behavior on $Abs = \mathbb{P}$. We model concrete data-flow equations by applying the abstraction function $\alpha_{\mathbb{P}}: 2^{Store} \longrightarrow \mathbb{P}$ to the sets involved in equations. Set inclusion $\subseteq$, union $\cup$, and intersection $\cap$ in the world of actual, concrete computations are modeled by $\sqsubseteq_{\mathbb{P}}$, $\sqcup_{\mathbb{P}}$, and $\sqcap_{\mathbb{P}}$ in the world of abstract computations over $\mathbb{P}$. Thus, we obtain the following system of abstract forward data-flow equations:

$inv_{l_{input}} = \alpha_{\mathbb{P}}(\mathbb{E})$
$inv_3 = assign_{\mathbb{P}}(\mathrm{i} := 0, inv_{l_{input}})$
$inv_4 = (inv_3 \sqcup_{\mathbb{P}} inv_7) \sqcap_{\mathbb{P}} filter_{\mathbb{P}}(\mathrm{i} < 100, \top_{\mathbb{P}})$
$inv_5 = assign_{\mathbb{P}}(\mathrm{i} := \mathrm{i} + 1, inv_4)$
$inv_6 = assign_{\mathbb{P}}(\mathrm{j} := \mathrm{j} + 1, inv_5)$
$inv_7 = inv_6$
$inv_{l_{final}} = (inv_3 \sqcup_{\mathbb{P}} inv_7) \sqcap_{\mathbb{P}} filter_{\mathbb{P}}(\neg(\mathrm{i} < 100), \top_{\mathbb{P}})$

The lattice $\mathbb{P}$ is also complete, so the above equation system has a unique least fixpoint solution. The two forward data-flow equation systems, $coll$ and inv, are related by: $inv_l \sqsupseteq_{\mathbb{P}} \alpha_{\mathbb{P}}(coll_l)$ for any program location l. This is the *soundness* relation showing that stores computed by abstract equations inv_l over-approximate the stores computed by concrete equations $coll_l$, for any location l. For the running example $\mathcal{P}_{A \wedge \neg B}(\mathrm{P})$, we obtain the following solution for $inv_{l_{final}} = (100 \leq \mathrm{j} \leq 109)$. Hence, we conclude that the assertion at l_{final} can be satisfied for $p_{final}^{sat} = filter_{\mathbb{P}}\left(\mathrm{j} \leq 105, inv_{l_{final}}\right) = (100 \leq \mathrm{j} \leq 105)$, and the assertion at l_{final} can be violated for $p_{final}^{viol} = filter_{\mathbb{P}}\left(\neg(\mathrm{j} \leq 105), inv_{l_{final}}\right) = (106 \leq \mathrm{j} \leq 109)$.

We now design two abstract backward interpreters that propagate backwards the invariants ensuring that the final assertion is satisfied p_{final}^{sat} and violated p_{final}^{viol}, respectively. The abstract backward interpreters refine the invariants found by inv. We have the following system of abstract backward data-flow equations:

$cond_{l_{input}} = b_assign_{\mathbb{P}}(\mathrm{i} := 0, cond_3)$
$cond_3 = (cond_{l_{final}} \sqcap_{\mathbb{P}} b_filter_{\mathbb{P}}(\neg(\mathrm{i} < 100), \top_{\mathbb{P}})) \sqcup_{\mathbb{P}} (cond_4 \sqcap_{\mathbb{P}} b_filter_{\mathbb{P}}(\mathrm{i} < 100, \top_{\mathbb{P}}))$
$cond_4 = b_assign_{\mathbb{P}}(\mathrm{i} := \mathrm{i} + 1, cond_5)$
$cond_5 = b_assign_{\mathbb{P}}(\mathrm{j} := \mathrm{j} + 1, cond_6)$
$cond_6 = cond_7$
$cond_7 = (cond_{l_{final}} \sqcap_{\mathbb{P}} b_filter_{\mathbb{P}}(\neg(\mathrm{i} < 100), \top_{\mathbb{P}})) \sqcup_{\mathbb{P}} (cond_4 \sqcap_{\mathbb{P}} b_filter_{\mathbb{P}}(\mathrm{i} < 100, \top_{\mathbb{P}}))$

The solution of the above system when $cond_{l_{final}} = p_{final}^{sat}$ is $cond_{l_{input}}^{sat} = (0 \leq \mathrm{j} \leq 5)$, whereas when $cond_{l_{final}} = p_{final}^{viol}$ is $cond_{l_{input}}^{viol} = (6 \leq \mathrm{j} \leq 9)$. Next, we call the LATTE tool to count the number of solutions from the input domain $\mathrm{j} \in [0,9]$ to the above preconditions $cond_{l_{input}}^{sat}$ and $cond_{l_{input}}^{viol}$. Finally, we obtain that the success probability is $Pr^s(\mathcal{P}_{A \wedge \neg B}(\mathrm{P})) = 60\%$ and $Pr^f(\mathcal{P}_{A \wedge \neg B}(\mathrm{P})) = 40\%$. Note that, for this running example the abstract analyses do not lose any precision, and they compute the same results for the success and failure probabilities as the concrete semantics. However, in general it is possible to lose some precision by abstract analyses, so the computed results represent lower and upper bounds of the exact ones. We now show one example, where we obtain approximate results. Consider the following program P':

```
void main() {
1:       int x := [0,9], y:=[0,9];
l_input: int s := x - y;
3:       if (s>=2) y:=y+2;
l_final: assert (y>3);
}
```

The forward analysis will infer that the program can both satisfy and violate the assertion. The backward necessary condition analysis for assertion satisfaction will discover the constraint: $\mathrm{x} + 2\mathrm{y} \geq 8 \wedge 0 \leq \mathrm{x} \leq 9 \wedge 2 \leq \mathrm{y} \leq 9$, thus we find that the upper bound probability for assertion satisfaction is 74%. Moreover, the input stores that do not satisfy the above precondition definitely lead to the assertion violation. Thus, the lower bound probability for assertion violation is 100%-74%=26%. The backward

necessary precondition analysis for assertion violation will discover the constraint: $x + 5y \leq 23 \wedge 0 \leq x \leq 9 \wedge 0 \leq y \leq 3$, thus we find that the upper bound probability for assertion violation is 32%. By similar reasoning as above, the lower bound probability for assertion satisfaction is 68%. On the other hand, we can calculate by hand that the success probability is exactly 71%, while the failure probability is exactly 29%.

FAMILY-BASED (LIFTED) STATIC ANALYSES

Lifted analyses are designed by *lifting* existing single-program analyses to work on program families, rather than on individual programs. They directly analyze the code base of program families, without preprocessing them by taking the variability introduced by #ifdef-s into account.

Concrete semantics. Since, we work with program families, we lift all definitions for single-program analyses configuration-wise. Thus, we work with *lifted stores* $\overline{Store} = Store^{\mathbb{K}} = \prod_{k\in\mathbb{K}} Store$ and *lifted states* $\overline{State} = State^{\mathbb{K}} = \prod_{k\in\mathbb{K}} State$, which represent a tuple of $|\mathbb{K}|$ copies of $Store$ and $State$, one for each valid configuration. Given a lifted store $\bar{s} \in \overline{State}$, $\pi_k(\bar{s})$ selects the k-th component of the tuple $\bar{s}$. We also work with a *lifted transition function* $\overline{trans}: (State \to State)^{\mathbb{K}}$, which represents a tuple of $|\mathbb{K}|$ independent simple functions, for which the k-th component of the function value only depends on the k-th component of the argument.

The collecting lifted semantics works on lifted stores and defines the set of lifted stores that can be reached at some program location *l*:

$$\overline{coll}_l = \{\bar{s} \mid (l, \bar{s}) = \overline{trans}^n(l_{input}, \bar{s_0}), \bar{s_0} \in \overline{\mathbb{E}}, n \geq 0\}$$

where $\overline{\mathbb{E}} \in 2^{\overline{Store}}$ is the set of input lifted stores.

We now show how our approach works for the running example family P from “Motivating example” section. The program family P has two variables i and j, and four configurations $\mathbb{K} = \{A \wedge B, A \wedge \neg B, \neg A \wedge B, \neg A \wedge \neg B\}$. The set of input lifted stores is $\overline{\mathbb{E}} = \{([j \mapsto \{0,\dots,9\}], [j \mapsto \{0,\dots,9\}], [j \mapsto \{0,\dots,9\}], [j \mapsto \{0,\dots,9\}])\}$ or $\overline{\mathbb{E}} = \{\prod_{k\in\mathbb{K}}[j \mapsto \{0,\dots,9\}]\}$ for short. The first component of a lifted store, i.e. a tuple, $([j \mapsto 0], [j \mapsto 0], [j \mapsto 0], [j \mapsto 0])$ corresponds to config. $A \wedge B$, the second to $A \wedge \neg B$, the third to $\neg A \wedge B$, and the fourth to $\neg A \wedge \neg B$. Forward and backward concrete lifted data-flow equations are the same as for single programs, except that they now work on lifted stores instead of stores. For example, we have:

$$\overline{coll}_{l_{input}} = \overline{\mathbb{E}}$$
$$\overline{coll}_3 = \overline{coll}_{l_{input}} \mathbin{\overline{\cap}} \{\prod_{k\in\mathbb{K}}[i \mapsto \{0\}, j \mapsto \mathbb{N}]\}$$
$$\overline{b_coll}_{l_{input}} = \{\prod_{k\in\mathbb{K}}[j \mapsto \pi_k(\overline{b_{coll_3}}(j))] \mid 0 \in \pi_k(b_{coll_3}(i))\}$$

where $\overline{\cap}$, $\overline{\cup}$ are lifted versions of intersection and union that work on tuples. We obtain the following solution for the above forward lifted equations $\overline{coll}_{final} = \{([i \mapsto \{100\}, j \mapsto \{200,\dots,209\}], [i \mapsto \{100\}, j \mapsto \{100,\dots,109\}], [i \mapsto \{100\}, j \mapsto \{100,\dots,109\}], [i \mapsto \{100\}, j \mapsto \{0,\dots,9\}])\}$. We can see that the target assertion ($j \leq 105$) is definitely violated for $A \wedge B$, and satisfied for $\neg A \wedge \neg B$. Therefore, the success probability Pr^s that a variant satisfies the target assertion is: $Pr^s(\pi_{A\wedge B}(P)) = 0\%$ and $Pr^s(\pi_{\neg A\wedge\neg B}(P)) = 100\%$, whereas the failure probability Pr^f that a variant violates a target assertion is: $Pr^f(\pi_{A\wedge B}(P)) = 100\%$ and $Pr^f(\pi_{\neg A\wedge\neg B}(P)) = 0\%$. In order to determine the success and failure probabilities for variants $A \wedge \neg B$ and $\neg A \wedge B$, we run two backward collecting lifted semantics: one for assertion satisfaction and one for assertion violation. Similarly as in the previous section, we can establish that $Pr^s(\mathcal{P}_{A\wedge\neg B}(\mathrm{P})) = Pr^s(\mathcal{P}_{\neg A\wedge B}(\mathrm{P})) = 60\%$ and $Pr^f(\mathcal{P}_{A\wedge\neg B}(\mathrm{P})) = Pr^f(\mathcal{P}_{\neg A\wedge B}(\mathrm{P})) = 40\%$.

Abstract semantics. Polyhedral abstract lifted analyses work on lifted domain $\overline{\mathbb{P}} = \mathbb{P}^{\mathbb{K}} = \prod_{k\in\mathbb{K}} \mathbb{P}$, which contains one separate copy for each configuration of $\mathbb{K}$. All abstract operations that work for $\mathbb{P}$ are lifted configuration-wise to work for $\mathbb{P}^{\mathbb{K}}$. Thus, we have lifted versions of partial ordering $\dot{\sqsubseteq}$, join $\dot{\sqcup}$, meet $\dot{\sqcap}$, bottom $\dot{\bot} = (\bot_{\mathbb{P}},\dots,\bot_{\mathbb{P}})$, top $\dot{\top} = (\top_{\mathbb{P}},\dots,\top_{\mathbb{P}})$, widening $\dot{\nabla}$, and narrowing $\dot{\triangle}$. There are also lifted versions of transfer functions for assignments $\overline{assign_{\mathbb{P}}}: Stm \times \overline{\mathbb{P}} \to \overline{\mathbb{P}}$, tests $\overline{filter_{\mathbb{P}}}: Exp \times \overline{\mathbb{P}} \to \overline{\mathbb{P}}$, backward assignments $\overline{b_assign_{\mathbb{P}}}: Stm \times \overline{\mathbb{P}} \to \overline{\mathbb{P}}$, and backward tests $\overline{b_filter_{\mathbb{P}}}: Exp \times \overline{\mathbb{P}} \to \overline{\mathbb{P}}$. Finally, we define two transfer functions to handle variability introduced by ``#ifdef (θ) s'' statements:

$$\overline{f_filter}(\theta, \bar{p}) = \prod_{k\in\mathbb{K}} \begin{cases} \pi_k(\bar{p}), \text{if } k \text{ satisfies } \theta \\ \bot_{\mathbb{P}}, \text{if } k \text{ does not satisfy } \theta \end{cases}$$

$$\overline{ifdef}(\#ifdef\ (\theta)\ s, \bar{p}) = [[\bar{s}]](\overline{f_filter}(\theta, \bar{p})) \mathbin{\dot{\sqcup}} \overline{f_filter}(\neg\theta, \bar{p})$$

where $[[\bar{s}]]$ represents the lifted transfer function for the statement s. The function $\overline{f_filter}$ keeps those components k of the input tuple $\bar{p}$ that satisfy θ, and replaces the other components of $\bar{p}$ with $\bot_{\mathbb{P}}$. The function $\overline{ifdef}$ captures the effect of analyzing the statement s in those components k of the input tuple $\bar{p}$ that satisfy θ, otherwise it is an identity for the other components of $\bar{p}$. Now, if we perform abstract lifted forward and backward analyses for the running example P, we can calculate the exact values for the success and failure probabilities for all four variants. For example, some abstract forward lifted data-flow equations are:

$$\overline{inv}_6 = \overline{ifdef}(\#if\ (A)\ j := j + 1, inv_5)$$
$$\overline{inv}_7 = \overline{ifdef}(\#if\ (B)\ j := j + 1, inv_6)$$

Optimization. We can speed up the abstract lifted analyses by using *shared representation* to represent sets of configurations with equivalent analysis

Table 1. Experimental evaluation of probabilistic lifted analysis based on BDDs vs. probabilistic lifted analyses based on tuples . All times are in seconds.

Bench.	source	$\lvert\mathbb{F}\rvert$	LOC	BDD^{10}		BDD^{1000}		EXACT
				$TIME^{10}$	$IMPROVE^{10}$	$TIME^{1000}$	$IMPROVE^{1000}$	
count_up_down*.c	loops	4	25	0.018	5×	0.019	5×	√
hhk2008.c	loop-lit	5	25	0.037	8.5×	0.040	9×	√
gsv2008.c	loop-lit	2	25	0.006	2×	0.007	2×	√
bwd_loop2.c	[12]	2	15	0.007	2.1×	0.007	2×	√
example7.c	[13]	4	20	0.013	6×	0.014	6.5×	≈

information. For this aim, we use *binary decision diagrams* (BDDs) as lifted analysis domains. We exploit the well-known efficiency of BDDs [16,17] for representing formulae that combine Boolean variables and analysis properties. The elements of the BDD domain are disjunctions of the leaf nodes that belong to an existing (single-program) analysis domain (e.g. the polyhedral domain), which are separated by the values of Boolean features organized in the decision nodes. Therefore, we encapsulate the set $\mathbb{K}$ into the decision nodes of a BDD where each top-down path represents one or several configurations from $\mathbb{K}$, and we store in each leaf node the property generated from the variants derived by the corresponding configurations.

We now formally define the lifted domain of BDDs. A *binary decision tree* (BDT) $t \in \mathbb{T}(\mathbb{F},\mathbb{P})$ over the set of features $\mathbb{F}$ and the leaf Polyhedral domain $\mathbb{P}$ is either a leaf <*p*> with $p \in \mathbb{P}$ and $\mathbb{F} = \varnothing$, or [*A*:*tl*,*tr*] where *A* is the *smallest element* of $\mathbb{F}$ with respect to its ordering, *tl* is the left subtree of *t* representing its true branch, and *tr* is the right subtree of *t* representing its false branch, such that $tl, tr \in \mathbb{T}(\mathbb{F}\backslash\{A\},\mathbb{P})$. Recall that $\mathbb{F} = \{A1, \dots, An\}$ is a totally ordered set with ordering $A1 < \cdots < An$. There are several reductions that can be applied to BDTs in order to remove the redundancy from their representation [16,17]: Removal of duplicate leaves; Removal of redundant tests; and Removal of duplicate non-leaves. If we apply the above reductions to a BDT t, we obtain a reduced *binary decision diagram* (BDD) $d \in \mathbb{D}(\mathbb{F},\mathbb{P})$. Thanks to the sharing of information enabled by the above reductions, BDDs are quite compact representation of tuples from $\mathbb{P}^{\mathbb{K}}$. Moreover, if the ordering on the features from $\mathbb{F}$, occurring on any path is fixed, then the resulting BDDs have a *canonical form*. This means that any property from the lifted domain $\mathbb{P}^{\mathbb{K}}$ can be represented in a unique way by a BDD.

For example, consider the running example program family P. If we use the lifted analysis domain $\mathbb{P}^{\mathbb{K}}$, we obtain the following solutions of abstract lifted analyses.

$$\overline{inv}_{l_{final}} = \{(200 \leq j \leq 209, 100 \leq j \leq 109, 100 \leq j \leq 109, 0 \leq j \leq 9)\}$$
$$\overline{cond}^{sat}{}_{l_{input}} = \{(\bot_{\mathbb{P}}, 0 \leq j \leq 5, 0 \leq j \leq 5, 0 \leq j \leq 9)\}$$
$$\overline{cond}^{viol}{}_{l_{input}} = \{(0 \leq j \leq 9, 6 \leq j \leq 9, 6 \leq j \leq 9, \bot_{\mathbb{P}})\}$$

If we use the BDD-based lifted domain instead, the BDDs representing the above locations are given in Fig. 1. They use three polyhedral properties instead of four as above in case of tuples. Moreover, for the input lifted store, the tuple-representation is $\overline{inv}_{l_{input}} = \{(0 \leq j \leq 9, 0 \leq j \leq 9, 0 \leq j \leq 9, 0 \leq j \leq 9)\}$, while the BDD representation uses only one polyhedral property, thus maximizing the effects of sharing. This ability for sharing is the key motivation behind the introduction of the BDD-representation.

EXPERIMENTS

In this section, we evaluate our approach for computing assertion probabilities of program families. We have implemented a prototype lifted static analyzer for analyzing programs written in C with `#ifdef` directives. The only basic data types are mathematical integers. The tool reports as output the upper and lower bounds of probabilities that the target assertion is satisfied or violated in all variants of the given family. The prototype tool is written in OCAML. All abstract operators and sound transfer functions for the polyhedral domain are provided by the APRON library [13]. The BDD domain which combines Boolean formulae and APRON domains is provided by BDDAPRON library [12]. The tool also calls the LATTE model counter [11] to determine the number of solutions to preconditions discovered by abstract lifted analyses.

All experiments are executed on a 64-bit IntelCoreTM i5 CPU, Lubuntu VM, with 8 GB memory. The reported times represent the average runtime of five independent executions. We have implemented two versions of lifted analysis: one based on BDDs $\mathbb{D}(\mathbb{F},\mathbb{P})$, and one based on tuples $\mathbb{P}^{\mathbb{K}}$. Thus, we evaluate the performances of our approach when using BDDs vs. tuples. In general, the chosen feature ordering makes a significant difference to the

size of the obtained BDD. In this work, the ordering of features is syntactically directed, that is the features occurring earlier in the syntax of a program are smaller in the ordering. It is an interesting topic for future research to consider other heuristics for finding good orderings [17].

For our experiment, we use several C programs taken from the 8th International Competition on Software Verification (SV-COMP 2019) (https://sv-comp.sosy-lab.org/2019) as well as from the abstract interpretation community [17, 18]. We have selected some numerical programs with integers that our tool can handle. We have manually added input sections and variability, and in some of the programs we have also defined target assertions. Then, we have analyzed those programs using our prototype static analyzer. Table 1 summarizes relevant characteristics for each benchmark: the source where it is taken from, the number of lines of code (LOC), and the number of features.

Table 1 shows the performance of our technique on a selected set of benchmarks. Regarding the preciseness of the results, we can see in the EXACT column that our tool gives exact results without any approximation very often (√ means that the result is exact, ≈ means the result is approximate). Note that the exact results are reported when the found lower and upper bounds for success and failure probabilities are the same. We obtain exact results in most of the cases due to the fact that we use the expressive and very precise polyhedral abstract domain. For BDD-based analysis, there are two columns. In the first column, TIME, we report the running time in seconds to analyze the given program family using BDDs. In the second column, IMPROVE, we report how many times a BDD-based analysis is faster than the corresponding analysis based on tuples. This way, IMPROVE represents a measure showing how much sharing occurs in BDDs for each benchmark. We can see that all BDD-based analyses achieve significant speed-ups compared to the tuple-based analyses, which range from 5 to 15 times. We have also experimented with different domain sizes n of input variables (for n=10 and n=1000). Thus, n denotes the number of possible values per input variable. We observe that we obtain similar time performance results for n=10 and n=1000, mostly due to the fact that LATTE and APRON are largely insensitive to those values in terms of time.

RELATED WORK

A formal methodology for deriving tuple-based lifted analysis from existing single-program analysis phrased in the abstract interpretation framework has been introduced in [8]. Subsequently, a lifted analysis with improved representation via BDDs has been proposed [10]. This paper extends the previous works on lifted analysis [8,10] by applying them in performing a specific type of probabilistic lifted analysis, that is the calculation of assertion probabilities. In particular, we combine the results obtained from lifted analyses and the model counting techniques to count the number of values for input variables that will lead to assertion satisfaction/violation. Hence, we put in practice the lifted analyses [8,10] to solve a practical problem.

Probabilistic analysis of single programs has been used before [3,4,5]. The work [3] uses symbolic execution to calculate path probabilities by counting the number of solutions to a path condition. However, in presence of loops this approach loses precision, since it cannot enumerate all program paths but considers only a finite number of feasible paths. The work [4] performs a probabilistic analysis of open programs with undefined identifiers (e.g. calls to library functions) using symbolic game semantics and model counting. In the presence of loops and undefined functions, bounded exploration is also used to obtain a feasible analysis. In this work, we use abstract interpretation to analyze programs, thus we provide a complete treatment of loops. Moreover, while all above analysis [3,4,5] work on single program, here we consider program families.

CONCLUSION

In this work, we have presented a combination of forward and backward abstract lifted analyses for computing reliability of program families. In particular, we calculate the lower and upper bounds of probabilities that a given assertion is satisfied or violated for all variants of a program family. The BDD-based lifted domain provides a symbolic and very compact representation of lifted properties of program families, where the sharing of information is maximized. We evaluate the proposed lifted domains on several C product lines. We experimentally demonstrate the effectiveness of BDD-based lifted analyses vs. tuple-based lifted analysis.

We currently support only uniform distribution of input values within their finite discrete domains. In future, we plan to model imprecision in the input by different non-uniform distributions, such as Binomial, Poisson, etc [20]. Our focus here is on estimating probability for safety properties. An interesting direction for future work would also be to consider liveness properties (termination) and expectation queries [19]. Another way to speedup lifted analyses is via so-called variability abstractions [21,22], which reduce the configuration space to something more tractable. Combining variability abstractions and BDD-representation would be interesting to consider in future.

REFERENCES

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

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

[3] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In Mats Per Erik Heimdahl and Zhendong Su, editors, International Symposium on Software Testing and Analysis, ISSTA 2012, pages 166--176. {ACM}, 2012.

[4] A. S. Dimovski. Probabilistic analysis based on symbolic game semantics and model counting. In Proceedings Eighth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2017, Roma, Italy, 20-22 September 2017., volume 256 of EPTCS, pages 1--15, 2017.

[5] A. S. Dimovski, A. Legay. Computing Program Reliability Using Forward-Backward Precondition Analysis and Model Counting. In Fundamental Approaches to Software Engineering - 23rd International Conference, FASE 2020. Proceedings, volume 12076 of LNCS, pages 182--202. Springer, 2020.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL'77), pages 238--252. ACM, 1977.

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

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

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

[10] A. S. Dimovski. Lifted static analysis using a binary decision diagram abstract domain. In Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, pages 147--160. ACM, 2019.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages (POPL'78)}, pages 84--96. ACM Press, 1978.

[12] Latte integrale. UC Davis, Mathematics.

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

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

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

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

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

[18] A. Mine. Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions. Sci. Comput. Program., 93:154--182, 2014.

[19] C. Urban and A. Mine. 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.

[20] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. In ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, pages 447--458. ACM, 2013.

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

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

ЗА ПРЕСМЕТУВАЊЕ НА ВЕРОЈАТНОСТИТЕ НА ТВРДЕЊАТА КАЈ ФАМИЛИИ ОД ПРОГРАМИ

Александар С. Димовски

Факултет за Информатички науки, “Мајка Тереза” Универзитет, Скопје, Македонија

Високо-кофигурабилни софтверски системи (фамилии од програми) се појавуваат во многу апликациски подрачја и поради многу причини. Тие може да продуцираат потенцијално многу слични програми (варијанти) преку селектирање на соодветни конфигурациски опции (особини) во компајлирачко време. Многу од овие конфигурабилни софтверски системи можат да примаат на влез и да манипулираат несигурни податоци.

Во овој труд, ние претставуваме еден метод за пресметка на веројатности на тврдења (assertions) во фамилии од програми со несигурни влезни податоци. Прво, ние користиме комбинација од на-напред и на-назад фамилијарни анализи кои се базирани на апстрактна интерпретација за да се пронајдат неопходните пред-услови за да дадено тврдење биде задоволено/незадоволено во сите варијанти од една фамилија на програми. Ние користиме фамилијарни анализи базирани на бинарни одлучувачки дијаграми (БДД-ја) и нумерички апстрактни домени (на пример, Полихедралниот домен) со помош на кои наоѓаме нумерички инваријанти во секоја програмска локација. Второ, техниките за броење на модели се искористени за да се најдат бројот на решенија на пронајдените неопходни пред-услови (кои се дадени во форма на линеарни ограничувања, т.е. линеарни неравенки). Ние ги користиме овие броеви за да ја процениме веројатноста дека целното тврдење е задоволено/незадоволено. Ние го имплементиравме овој метод во една прототип алатка, и направиме нејзина евалуација на неколку интересни фамилии од Ц програми.

Клучни зборови: Статичка анализа преку апстрактна интерпретација, Броење на модели, Софтверски Продуктни Линии (фамилии од програми)