Splitting the Control Flow with Boolean Flags

Axel Simon

DI, École Normale Supérieure, rue d'Ulm, 75230 Paris cedex 05, France**
Axel.Simon@ens.fr

Abstract. Tools for proving the absence of run-time errors often deploy a numeric domain that approximates the possible values of a variable using linear inequalities. These abstractions are adequate since the correct program state is often convex. For instance, if the upper and lower bound of an index lie within the bounds of an array, then so do all the indices inbetween. In certain cases, for example when analysing a division operation, the correct program state is not convex. In this case correctness can be shown by splitting the control flow path, that is, by partitioning the set of execution traces which is normally implemented by analysing a path several times. We show that adding a Boolean flag to the numeric domain has the same effect. The paper discusses prerequisites, limitations and presents an improved points-to analysis using Boolean flags.

1 Introduction

With better expressiveness and scalability, static analysis is increasingly used to prove the absence of run-time errors of certain types of C software [3, 7]. A classic approach of approximating the state at a particular point in the program are convex spaces described by a finite conjunction of linear inequalities (a polyhedron [8]). For example, an array access `a[i]` in C is within bounds if the inferred set of inequalities at that program point imply a space where $i \geq 0$ and $i \leq s-1$ where s is the size of the array. In certain cases, a single convex approximation to the possible state space is not precise enough. A notorious example is a guard against a division by zero, as implemented by the following C fragment:

```
int r=MAX_INT;
if (d!=0) r=v/d;
```

The test $d \neq 0$ must be implemented by intersecting the current state, say P, with $d > 0$ and $d < 0$, resulting in $P^+ = P \cap [\![d > 0]\!]$ and $P^- = P \cap [\![d < 0]\!]$ where $[\![c]\!]$ denotes the sub-space of the Euclidean space that satisfies the constraint c. The state with which the division operation is analysed is approximated by $P' = P^+ \sqcup P^-$, where $\sqcup$ is the join of the lattice, implemented by calculating the closure of the convex hull of the two polyhedra. However, the convex hull re-introduces the state $P \sqcap \{d = 0\}$ for which the division `r=v/d` is erroneous. In fact, it turns out that $P' = P$ whenever P^+ and P^- are non-empty.

** This work was supported by the INRIA project "Abstraction" of CNRS and ENS.

The Astrée analyser verifies [7] the above example by partitioning the set of execution traces, that is, by executing the division instructions twice, first with P^-, then with P^+. The question when to partition the traces into two sets and when to rejoin the separate trace sets is guided by heuristics and by manual annotations [14]. Thus, the Astrée analyser is designed with the ambition to choose the correct split and joint points in order to prove the program correct.[1] The Static Driver Verifier (SDV) [3] verifies the API usage of device drivers by converting their C source into a program with only Boolean variables [5] whose correctness is then checked against a pre-defined set of rules using a model checker [4]. In case a rule cannot be proved or disproved, a new predicate is synthesised, thereby splitting the set of traces in the concrete program with the aim to improve the precision of the abstraction with respect to the property of interest. Thus, SDV iteratively splits the control flow path and re-analyses the source until the rules can be proved. Both approaches explicitly partition the set of traces or, semantically equivalent, perform a program transformation that splits the control flow path into two. The cost of the analysis thus depends on the choice of the split and join points. In this work we propose to use polyhedral analysis but implement the split of a control flow path by adding a Boolean flag to the domain. Although the cost of adding such a flag might be higher than analysing the code twice, an additional flag is very cheap if the two states are identical. This allows our method to be used opportunistically, that is, Boolean flags may be introduced by default when it is not know if two states need to be separated later. Furthermore, our approach is appealing due to its simplicity.

The motivation of our work stems from the need to improve points-to information during a polyhedral value-range analysis. By using Boolean flags to indicate if an l-value is part of a points-to set, it is possible to infer that e.g. the error flag returned from a function is set whenever a pointer is `NULL`. Points-to information presents a good example where the re-analysis of code with every possible set of points-to information can be more expensive than adding flags.

Adding a Boolean flag may result in a loss of precision, especially when used with weakly relational domain such as Octagons [15] or TVPI polyhedra [22]. To summarise, this paper makes the following contributions:

- We show that, under certain prerequisites, adding a Boolean flag is equivalent to analyzing a control flow path twice.
- We discuss the way Boolean flags can be used in the context of general polyhedra and weakly relational domains such as the TVPI domain.
- We show how points-to analysis and numeric analysis can be combined using Boolean flags, thereby improving precision without explicit propagation between the two domains.

The paper is organised as follows. The next section gives a definition of the polyhedral domain before Sec. 3 discusses the separation of states using Boolean flags. Section 4 demonstrates practical applications and Section 5 concludes.

[1] Astrée can also partition the set of traces by the value of variables. In this case, joining the partitions is automatic, e.g. when the variable goes out of scope. Our proposal can be used for both applications but is more amenable to the latter task.

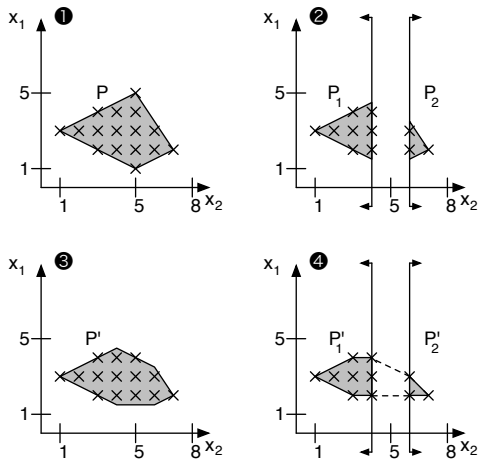


Fig. 1. Example to show that $\mathbb{Z}$-polyhedra are not closed under intersection. Integral points are shown as crosses and the half-spaces $[\![x \leq 4]\!]$ and $[\![x \geq 6]\!]$ are indicated by vertical lines with arrows pointing towards the feasible space.

2 An Introduction to Convex Polyhedra

This section introduces some basic notation and properties of polyhedra, putting particular emphasis on the set of integral points contained in a polyhedron.

A polyhedral analysis expresses numeric constraints over the set of abstract variables $\mathcal{X}$. For the sake of this section, let $\boldsymbol{x}$ denote the vector of all variables in $\mathcal{X}$, thereby imposing an order on $\mathcal{X}$. Let *Lin* denote the set of linear expressions of the form $\boldsymbol{a} \cdot \boldsymbol{x}$ where $\boldsymbol{a} \in \mathbb{Z}^{|\mathcal{X}|}$ and let *Ineq* denote the set of linear inequalities $\boldsymbol{a} \cdot \boldsymbol{x} \leq c$ where $c \in \mathbb{Z}$. Let e.g. $6x_3 \leq x_1 + 5$ abbreviate $\langle -1, 0, 6, 0, \ldots 0 \rangle \cdot \boldsymbol{x} \leq 5$ and let e.g. $x_2 = 7$ abbreviate the two opposing inequalities $x_2 \leq 7$ and $x_2 \geq 7$, the latter being an abbreviation of $-x_2 \leq -7$. For simplicity, we assume that the analysis only infers integral properties and we use the notation $e_1 < e_2$ to abbreviate $e_1 \leq e_2 - 1$. Each inequality $\boldsymbol{a} \cdot \boldsymbol{x} \leq c \in \mathit{Ineq}$ induces a half-space $[\![\boldsymbol{a} \cdot \boldsymbol{x} \leq c]\!] = \{\boldsymbol{x} \in \mathbb{Q}^{|\mathcal{X}|} \mid \boldsymbol{a} \cdot \boldsymbol{x} \leq c\}$. A set of inequalities $I \subseteq \mathit{Ineq}$ induces a closed, convex space $[\![I]\!] = \bigcap_{\iota \in I} [\![\iota]\!]$. Define $\mathit{Poly} = \{[\![I]\!] \mid I \in \mathit{Ineq} \wedge |I| \in \mathbb{N}\}$ to be the set of all closed, convex polyhedra. Let $P_1, P_2 \in \mathit{Poly}$, then $P_1 \sqsubseteq P_2$ iff $P_1 \subseteq P_2$ and define $P_1 \sqcap P_2$ as $P_1 \cap P_2$ which can be implemented by joining the two sets of inequalities that describe P_1 and P_2. Furthermore define $P_1 \sqcup P_2$ to be the smallest polyhedron P such that $P_1 \sqsubseteq P$ and $P_2 \sqsubseteq P$, that is $P_1 \sqcup P_2 = \bigcap \{P \in \mathit{Poly} \mid P_1 \sqsubseteq P \wedge P_2 \sqsubseteq P\}$. This operation corresponds to the topological closure of the convex hull of P_1 and P_2. Since the number of inequalities $|I|$ defining a $P \in \mathit{Poly}$ is finite by definition, the lattice of convex polyhedra $\langle \mathit{Poly}, \sqsubseteq, \sqcup, \sqcap \rangle$ is incomplete as neither the join nor meet of an arbitrary number of polyhedra is necessarily a polyhedron. Thus, in order to ensure that a fixpoint calculation always terminates on elements of *Poly* a widening operator is required [8].

In the context of storing Boolean flags in a polyhedral domain, it is interesting to equate polyhedra that contain the same set of integral points. These equivalence classes define the lattice of $\mathbb{Z}$-polyhedra $\langle Poly_{\equiv\mathbb{Z}}, \sqsubseteq^{\mathbb{Z}}, \sqcup^{\mathbb{Z}}, \sqcap^{\mathbb{Z}}\rangle$. Each equivalence class can be represented by its smallest polyhedron, namely a $\mathbb{Z}$-polyhedron which is characterised by the fact that all its vertices, that is, all points that are not a convex combination of other points, have integral coordinates. In this case it is possible to set $\sqsubseteq^{\mathbb{Z}} = \sqsubseteq$ and $\sqcup^{\mathbb{Z}} = \sqcup$. However, the meet operation $\sqcap$ is not closed for $\mathbb{Z}$-polyhedra. In order to illustrate this, consider Fig. 1. The state space $P \in Poly_{\equiv\mathbb{Z}}$ over x_1, x_2 in the first graph is transformed by evaluating the conditional $x_2 \neq 5$ which is implemented by calculating $P' = (P \sqcap [\![x \leq 4]\!]) \sqcup (P \sqcap [\![x \geq 6]\!])$. Observe that the input P as well as the two half-spaces $[\![x \leq 4]\!]$ and $[\![x \geq 6]\!]$ are $\mathbb{Z}$-polyhedra. The second graph shows the two intermediate results $P_1 = P \sqcap [\![x \leq 4]\!]$ and $P_2 = P \sqcap [\![x \geq 6]\!]$ which both have two non-integral vertices. As a consequence, the join of P_1 and P_2, shown as third graph, has non-integral vertices as well and is therefore not a $\mathbb{Z}$-polyhedron. However, if the intermediate results were shrunk around the contained integral point sets, as done in the fourth graph, all vertices of the intermediate results would be integral and the join would be a $\mathbb{Z}$-polyhedron, too. However, for general, n-dimensional polyhedra, the number of inequalities necessary to represent a $\mathbb{Z}$-polyhedron can grow exponentially with respect to a polyhedron over $\mathbb{Q}$ that contains the same integral points [17, Chap. 23]. Thus, no efficient algorithm exists to implement the $\sqcap$-operation on $\mathbb{Z}$-polyhedra. For the weakly relational domain of TVPI polyhedra where each inequality takes on the form $ax + by \leq c$ where $a, b, c \in \mathbb{Z}$, reducing the polyhedron to a $\mathbb{Z}$-polyhedron is still NP-complete [13], however, a polynominal algorithm exists for calculating the $\mathbb{Z}$-polyhedron for any given planar polyhedron [11] on which an efficient approximation of $\mathbb{Z}$-TVPI polyhedra can be built. Interestingly, for the Octagon domain [15], an efficient algorithm exists [2].

The next section re-examines the introductory example and illustrates its analysis in the context of using a single convex polyhedron.

3 Principles of Boolean Flags in Polyhedra

Reconsider the evaluation of the following code fragment, given the state P:

```
int r=MAX_INT;
if (d!=0) r=v/d;
```

Suppose this block of code is executed with a value of $[-9, 9]$ for `d`. Rather than analysing the division twice, once with positive values of d, once with negative values of d, Fig. 2 shows how the states P^+ and P^- can be stored in a single state without introducing an integral point where $d = 0$. Specifically, the figure shows the state $(P^- \sqcap [\![f = 0]\!]) \sqcup (P^+ \sqcap [\![f = 1]\!])$ which collapses to the empty $\mathbb{Z}$-polyhedron when intersected with $d = 0$, where $[\![d = 0]\!]$ is the state for which the division is erroneous. The prerequisites for merging two

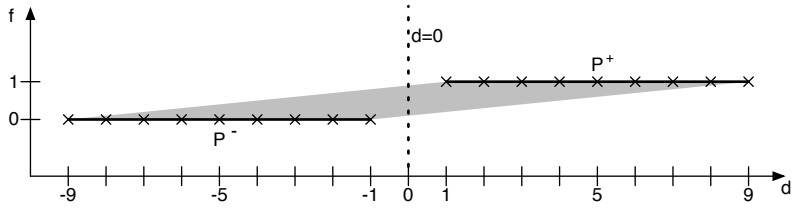


Fig. 2. Using a Boolean flag to perform control flow splitting. Feasible integral points are indicated by crosses, the dashed line indicating the polyhedron $[\![d = 0]\!]$.

states without loss is that both states are represented by polytopes (polyhedra in which all variables are bounded) and that techniques for integral tightening are present. This is formalised in the following proposition:

Proposition 1. *Let* $P_0, P_1 \in \mathit{Poly}$*, let* $P = (P_0 \sqcap [\![f = 0]\!]) \sqcup (P_1 \sqcap [\![f = 1]\!])$ *and* $P'_i = P \sqcap [\![f = i]\!]$ *for* $i = 0, 1$*. Then* $P_i \sqcap [\![f = i]\!] \cap \mathbb{Z}^n = P'_i \cap \mathbb{Z}^n$ *for all* $i = 0, 1$*.*

Proof. Without loss of generality, assume $i = 0$. Suppose that $\mathcal{X}$ is arranged as $\langle x_1, \ldots x_{n-1}, f\rangle = \boldsymbol{x}$ which we abbreviate as $\langle \bar{\boldsymbol{x}}|f\rangle = \boldsymbol{x}$. We consider two cases:

"soundness": Let $\langle \bar{\boldsymbol{a}}|f\rangle \in P_0 \sqcap [\![f = 0]\!] \cap \mathbb{Z}^n$. Then $\langle \bar{\boldsymbol{a}}|f\rangle \in P$ by the definition of $\sqcup$. Since $f = 0$ in $\langle \bar{\boldsymbol{a}}|f\rangle$ it follows that $\langle \bar{\boldsymbol{a}}|f\rangle \in P \sqcap [\![f = 0]\!]$ and thus $\langle \bar{\boldsymbol{a}}|f\rangle \in P'_0$. We chose the vector such that $\langle \bar{\boldsymbol{a}}|f\rangle \in \mathbb{Z}^n$ and hence $\langle \bar{\boldsymbol{a}}|f\rangle \in P'_0 \cap \mathbb{Z}^n$.

"completeness": Let $\langle \bar{\boldsymbol{a}}|f\rangle \in P'_0 \cap \mathbb{Z}^n$. Then $\langle \bar{\boldsymbol{a}}|f\rangle \in P \sqcap [\![f = 0]\!]$ and hence $f = 0$. For the sake of a contradiction, suppose that $\langle \bar{\boldsymbol{a}}|0\rangle \notin P_0$. Since $P \in \mathit{Poly}$ is convex, $\langle \bar{\boldsymbol{a}}|0\rangle = \lambda\langle \bar{\boldsymbol{a}}_{\boldsymbol{1}}|f_1\rangle + (1-\lambda)\langle \bar{\boldsymbol{a}}_{\boldsymbol{2}}|f_2\rangle$ for some $0 \leq \lambda \leq 1$. Observe that the join $P = (P_0 \sqcap [\![f = 0]\!]) \sqcup (P_1 \sqcap [\![f = 1]\!])$ is defined as an intersection of all states $\hat{P}$ such that $(P_i \sqcap [\![f = i]\!]) \sqsubseteq \hat{P}$ for $i = 0, 1$; this holds in particular for $\hat{P} = [\![f \geq 0]\!]$ and for $\hat{P} = [\![f \leq 1]\!]$. Thus, $0 \leq f \leq 1$ for all $\langle \bar{\boldsymbol{a}}|f\rangle \in P$. Hence, $0 \leq f_1 \leq 1$ and $0 \leq f_2 \leq 1$ must hold. Given the constraints $0 = \lambda f_1 + (1-\lambda) f_2$ and $\bar{\boldsymbol{a}} = \lambda\bar{\boldsymbol{a}}_{\boldsymbol{1}} + (1-\lambda)\bar{\boldsymbol{a}}_{\boldsymbol{2}}$, either $f_1 = 0$ or $f_2 = 0$ so that one vector, say $\langle \bar{\boldsymbol{a}}_{\boldsymbol{1}}|f_1\rangle$, must lie in P_0, which implies $\lambda = 1$. In particular with $\bar{\boldsymbol{a}} = \lambda\bar{\boldsymbol{a}}_{\boldsymbol{1}} + (1-\lambda)\bar{\boldsymbol{a}}_{\boldsymbol{2}}$ and $\bar{\boldsymbol{a}}_{\boldsymbol{2}} \in \mathbb{Z}^{n-1}$ having only finite coefficients, $\bar{\boldsymbol{a}} = \bar{\boldsymbol{a}}_{\boldsymbol{1}}$ follows, which contradicts our assumption of $\langle \bar{\boldsymbol{a}}|0\rangle \notin P_0$.

Equivalently, *Poly* over $\mathcal{X} \cup \{f\}$ is isomorphic to $\mathit{Poly}^{\{0,1\}}$ over $\mathcal{X}$, a reduced cardinal power domain [6, Sect. 10.2] and thus equivalent to a partitioning. We briefly comment on the requirements that P_0, P_1 must be polytopes and integral.

3.1 Boolean Flags and Unbounded Polyhedra

With respect to the first requirement, namely that the state space described by the polyhedron must be bounded, Fig. 3 shows that a precision loss occurs for unbounded polyhedra. Specifically, taking convex combinations of points in the state $P^+ = [\![\{d \geq 1, f = 1\}]\!]$ and those in $P^- = [\![\{-9 \leq d \leq -1, f = 0\}]\!]$ leads to the grey state. Even though the line $[\![\{f = 0, d > -1\}]\!]$ is not part of the grey

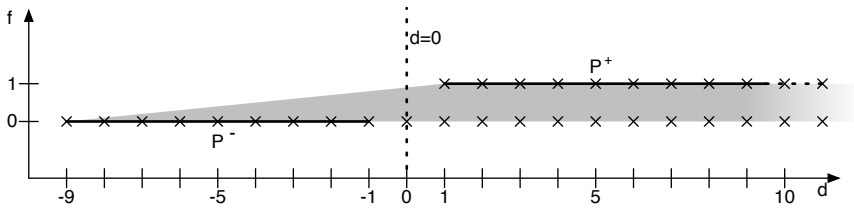


Fig. 3. Adding a Boolean flag to an unbounded polyhedra results in a loss of precision.

state, the polyhedral join $P^+ \sqcup P^-$ approximates this state with sets of non-strict inequalities, thereby including the line. Hence, the definition of $\sqcup$ automatically closes the resulting space and thereby introduces points $\langle \bar{\boldsymbol{a}}, f \rangle \in P^+ \sqcup P^-$ where $f = 0$, even though $\langle \bar{\boldsymbol{a}}, 0 \rangle \notin P^+$. As a result the intersection with $[\![d = 0]\!]$ contains $\langle 0, 0 \rangle \in \mathbb{Z}^2$ and the verification of the division operation fails. Note that allowing strict inequalities $\boldsymbol{a} \cdot \boldsymbol{x} < c$ to describe open facets [1] is not sufficient to define the convex space $\bigcap\{S \mid P^+ \subseteq S \wedge P^- \subseteq S\}$ as the lower bound on f is closed for $-9 \leq d \leq -1$ and open for $d > -1$.

The imprecise handling of unbounded polyhedra is generally not a problem in verifiers that perform a forward reachability analysis as program variables are usually finite and wrap when they exceed their limit. Flagging wrapping as erroneous [7] or making wrapping explicit [21] effectively restricts the range of a variable. Thus, states in a forward analysis are usually bounded. Polyhedra have also been used to infer an input-output relationship of a function, thereby achieving a context-sensitive analysis by instantiating this input-output behaviour at various call sites [10]. In this application the inputs are generally unbounded and a Boolean flag does not distinguish any differences between states. However, even in this application it might be possible to restrict the range of input variables to the maximum range that the concrete program variable may take on, thereby ensuring that input-output relationships are inferred using polytopes.

3.2 Integrality of the Solution Space

A second prerequisite for distinguishing two states within a single polyhedron is that the polyhedron is reduced to the contained $\mathbb{Z}$-polyhedron upon each intersection. Tightening a polyhedron to a $\mathbb{Z}$-polyhedron is an exponential operation which can be observed by translating a Boolean function f over n variables to a $\mathbb{Z}$-polyhedron over $\mathbb{Z}^n$ by calculating the convex hull of all Boolean vectors (using 0,1 for false and true) for which f is true. For example, Fig. 4 shows how common Boolean functions over two variables are represented as planar polyhedra. An argument similar to Prop. 1 is possible to show that joining all n-ary vectors for which f is true leads to a polyhedron that expresses f exactly. The integral meet operation $\sqcap^{\mathbb{Z}}$ therefore becomes a decision procedure for satisfiability of n-ary Boolean formulae. Hence, Octagons [15] together with the complete algorithm for $\sqcap^{\mathbb{Z}}$ presented in [2] provides an efficient decision procedure for 2-SAT.

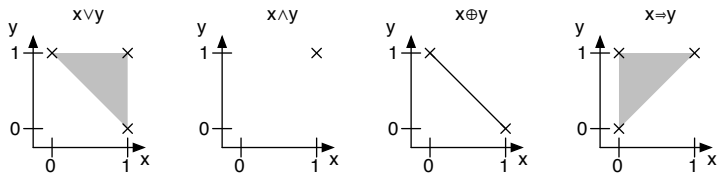


Fig. 4. Boolean functions can be expressed exactly in the polyhedral domain when satisfiable assignments of variables are modelled as vertices in the polyhedron.

While calculating a full $\mathbb{Z}$-polyhedron from a given polyhedron with rational intersection points is expensive, a cheap approximation often suffices in practice. For instance, the abstract transfer function of the division operation will add the constraint $d = 0$ to the state space shown in Fig. 2 with the result that possible values of f lie in $[0.1, 0.9]$. Rounding the bounds to the nearest feasible integral value yields the empty interval $[1, 0]$ which indicates an unreachable state, thereby proving that a division by zero cannot happen.

3.3 Using Boolean Flags in Common Polyhedral Domains

In this section we briefly discuss the applicability of our approach to three polyhedral domains: general convex polyhedra, TVPI polyhedra and Octagons.

General convex polyhedra [8] over $\mathbb{Q}^n$ can be defined by sets of inequalities of the form $a_1x_1 + \ldots a_nx_n \leq c$ where $a_1, \ldots a_n, c \in \mathbb{Z}$. The major hindrance for a widespread use is the complexity of the join operation $P_1 \sqcup P_2$ which is commonly implemented by converting the set of inequalities describing P_1 and P_2 into the vertex, ray and line representation which is usually exponential in the number of inequalities, even for simple polyhedra. This intermediate representation can be avoided by using projection to calculate the convex hull [20]. However, we are not aware of any efficient methods to approximate the calculation of a $\mathbb{Z}$-polyhedron from a rational polyhedron. One simple step is to calculate the common divisor of the coefficients $d = \gcd(a_1, \ldots a_n)$ and to tighten each inequality to $\frac{a_1}{d}x_1 + \ldots \frac{a_n}{d}x_n \leq \lfloor \frac{c}{d} \rfloor$. Rounding the constant down looses no integral solutions and thus constitutes a simple integral tightening [16].

The TVPI domain [22] is a weakly relational domain in that it only tracks relationships of the form $ax_i + bx_j \leq c$ where $a, b, c \in \mathbb{Z}$ and $x_i, x_j \in \mathcal{X}$. The idea is to repeatedly calculate resultants, that is, to combine any two inequalities $ax + by \leq c_{xy}$ and $dy + ez \leq c_{yz}$ where $sgn(b) = -sgn(d)$ to a new inequality $fx + gz \leq c_{xz}$ and to add this inequality to the set unless it is redundant. Once no more non-redundant inequalities can be generated, the system is called *closed*. On a closed system, the join $\sqcup$ and inclusion test $\sqsubseteq$ on the n-dimensional TVPI polyhedron can be implemented by performing a much simpler operation on $O(n^2)$ planar polyhedra. Specifically, the planar algorithms are run on the inequalities that only contain x_i, x_j, for each pair of variables $x_i, x_j \in \mathcal{X}$. Each planar operation runs in at most $O(m \log m)$ where m is the number of inequalities, thus providing an efficient domain. Furthermore, integral tightening

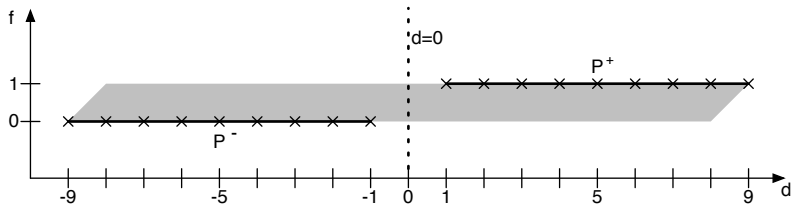


Fig. 5. The shown state is the octahedral approximation of the state in Fig. 2.

is possible by calculating the $\mathbb{Z}$-polyhedron for each projection in $O(m \log |A|)$ where A is the largest coefficients in any of the inequalities [11]. However, after applying this algorithm to the projection x_i, x_j, it is possible to calculate new resultants by combining the new inequalities with other inequalities containing x_i or x_j which may create rational intersection points in other, already tightened, projections. In fact, calculating a $\mathbb{Z}$-TVPI polyhedron cannot be done efficiently, as the problem is NP-complete [13]. However, performing integral tightening once after each closure is an efficient approximation to a $\mathbb{Z}$-TVPI polyhedron.

Since the TVPI domain has to approximate every inequality with more than two variables, the ability to separate states within the same TVPI polyhedron using a Boolean flag is somewhat limited. In fact, a Boolean flag f in the TVPI domain can only state that the range of a program variable is different. For instance, the TVPI domain is able to express the polyhedron in Fig. 2 and thereby prove the introductory example correct. It is beyond the TVPI domain to state how the linear relationship between two variables change between two states. This is discussed in Sect. 5 where we also suggest a partial solution.

The third domain we consider is the Octagon domain. This domain pre-dates the TVPI domain and is based on similar ideas [15]. The Octagon domain can express inequalities of the form $\pm x_i \pm x_j \leq c$ where $c \in \mathbb{Q}$ or $c \in \mathbb{Z}$. A variant of the Floyd-Warshall shortest paths algorithm is used to calculate a closed system. As above, calculating the join and the inclusion check is implemented for each x_i, x_j variable pair and is as simple as calculating the maximum and performing a comparison on the constants, respectively. Recently, it was shown how the closure algorithm can be modified to obtain a $\mathbb{Z}$-Octagon [2]. However, due to the restriction on the coefficients of the Octagon domain to $+1$ and -1, adding a Boolean flag cannot generally separate two ranges within a single Octagon. Specifically, a Boolean flag can only express the change of another variable by one or minus one, any other change is approximated. Figure 5 shows the state space $P^+ \sqcup P^-$ of the introductory example when approximated with the Octagon domain. Given that the state space can only be delimited by inequalities with unit coefficients, the approximation includes the points $\langle d, f \rangle = \langle 0, 0 \rangle$ and $\langle 0, 1 \rangle$ and hence cannot show that $d \neq 0$. Thus, in the context of the Octagon domain, splitting the set of traces along a control flow path can only be attained by re-evaluating the path several times [14], an approach that is abundant in the Astrée analyser [7] whose principal relational domain is the Octagon domain.

4 Applications of Control Flow Path Splitting

In the remainder of the paper we present various applications where Boolean flags are a valuable alternative to a repeated analysis. We commence with an example on pointer analysis before discussing Boolean flags in string buffer analysis [19].

4.1 Refining Points-To Analysis

The verification of C code often hinges on the ability to analyse pointer operations precisely. A cheap approach to dealing with pointers is to run a flow-insensitive points-to analysis [12] up front and assume that in each pointer access all memory regions in the points-to set of that pointer are accessed. A more accurate approach is to perform a flow-sensitive points-to analysis alongside the fixpoint computation on the numeric domain. To this end, define the points-to domain $Pts = \mathcal{X} \to \mathcal{P}(\mathcal{A})$ which maps each abstract variable $x \in \mathcal{X}$ to a set of abstract addresses $\{a_1, \ldots a_k\} \in \mathcal{P}(\mathcal{A})$ where each abstract address a_i represents the address of a global, automatic or one or more heap-allocated memory regions. The analysis now calculates a fixpoint over the product domain $\langle P, A\rangle \in Poly \times Pts$. This flow-sensitive analysis allows points-to sets to be refined through conditionals and, in particular, it can track if a pointer is `NULL`.

In order to define the semantics of points-to sets, we define $\rho : \mathcal{A} \to \mathcal{P}(\mathbb{N})$ to map an abstract address to concrete addresses. Using this map makes it possible to model variables in functions that are currently not executed as well as summarised heap regions that are represented with a single abstract variable. The possible values of a concrete variable `p` when modelled by x_p are defined as

$$\gamma_{\texttt{p}}(\langle P, A\rangle) := \{v + p \mid v \in P(x_p) \wedge p \in \rho(a) \wedge a \in A(x_p)\},$$

where $P(x_p)$ denotes the set of integral values that the variable x_p can take on in $P \in Poly$. In order to state that a pointer may be `NULL` or is simply a pure value, we introduce the special abstract address NULL with $\rho(\text{NULL}) = \{0\}$. As an example, consider the variable `p` that is represented by $x_p \in \mathcal{X}$ with $P(x_p) = [0, 1]$ and $A(x_p) = \{\text{NULL}, a_1, a_2\}$. Assuming that the abstract addresses map to the addresses $\rho(a_1) = \{0x4000\}$ and $\rho(a_2) = \{0x4004\}$, the program variable `p` can take on the values $0, 1, 0x4000, 0x4001, 0x4004$ and $0x4005$.

Operations on the product domain $\langle P, A\rangle \in Poly \times Pts$ have to evaluate the information in both domains and update them accordingly. For instance, the expression `p+q-r`, where `p` is represented by $x_p \in \mathcal{X}$ and so on for `q` and `r`, has different values, depending on the points-to sets $A(x_p)$, $A(x_q)$ and $A(x_r)$. For instance, if $\{a\} = A(x_q)$ and $\{a\} = A(x_r)$ then `q-r` represents a pointer difference, its value is $x_p + x_q - x_r$ and its points-to set is $A(x_p)$. However, if $|A(x_r)| \geq 2$ a severe loss of precision occurs: Suppose $A(x_r) = A(x_q) = \{a_1, a_2\}$ then it is not clear that the pointers contain the same abstract address and it has to be assumed that $a_1 \in A(x_r)$, $a_2 \in A(x_q)$ is possible. A sound approximation is to return the points-to set $\{\text{NULL}\}$ and $[0, 2^{32} - 1]$ for the value of the expression.

Applying the Abstraction In order to illustrate the weakness of the above abstraction, consider the following call to the function `f` that may assign to `p`:

```
char *p;
int r;
r = f(&p);
/* other statments here */
if (r) printf("value: %s", p);
```

The return value of `f` indicates whether setting `p` was successful; if it was, the value is printed. The function `f` itself is implemented as follows:

```
int f(char** pp) {
  if (rand()) return 0; /* error */
  *pp = "Success."; return 1; /* success */
}
```

Since the value of `p` is not initialised, the corresponding variable x_p has the points-to set $A(x_p) = \{\text{NULL}\}$ and takes on the range $[0, 2^{32} - 1]$, assuming `p` is a 32-bit variable. In case the random number generator returns zero, the pointer is set to the address of the string buffer `"Success."` with an offset of zero. Hence, at the end of `f`, the value of x_p is zero and its points-to set is $A(x_p) = \{a_s\}$ where a_s denotes the address of the string. By returning a different value for each case, the range of the variable x_p is $[0, 2^{32} - 1]$ whenever $r = 0$ and $[0, 0]$ whenever $r = 1$. Thus, testing the return flag in the caller will restrict the offset x_p of the pointer to zero before `printf` is called. However, the points-to set remains unchanged and thus, the analysis will warn that `p` might be `NULL`.

This example can readily be verified by partitioning the set of traces at the `if` statement in the function `f` such that $P(x_p) = [0, 2^{32} - 1]$ and $A(x_p) = \{\text{NULL}\}$ in one set of traces and $P(x_p) = [0, 0]$ and $A(x_p) = \{a_s\}$ in the other. However, it is also possible to use a Boolean flag f_p^s to indicate if x_p contains the address a_s. In particular, the analysis will infer that f_p^s is equal to `r`. Tracking a single equality constraint in addition to the polyhedron of the above analysis is likely to be cheaper than analysing a potentially long sequence of statements twice.

A Revised Abstraction In order to track the contents of points-to sets in the numeric part of the domain, we use a revised points-to domain $A : \mathcal{X} \to \mathcal{P}(\mathcal{A} \times \mathcal{X})$ that is fixed such that $A(x_p) = \{\langle a_1, f_1^p \rangle, \ldots \langle a_k, f_k^p \rangle\}$ where the set $\{a_1, \ldots a_k\}$ contains all abstract addresses that may be stored in x_p and where x_p itself holds the offset or value of `p`. Suppose the variable x_p corresponds to the ith dimension of the polyhedron $P \in \mathit{Poly}$ and $\langle f_1^p, \ldots f_k^p \rangle$ correspond to the next k dimensions, then the value of `p` is defined as follows:

$$\begin{aligned}\gamma_{\mathtt{p}}(P) := \{v + f_1^p p_1 + \ldots f_k^p p_k \mid\ & \langle \ldots x_{i-1}, v, f_1^p, \ldots f_k^p, x_{i+k+1}, \ldots \rangle \in P \cap \mathbb{Z}^n \\ & \wedge \{\langle a_1, f_1^p \rangle, \ldots \langle a_k, f_k^p \rangle\} = A(x_p) \\ & \wedge p_i \in \rho(a_i), i = 1 \ldots k\}\end{aligned}$$

The above concretisation interprets the Boolean variables in the domain as multiplier for the addresses with which they are associated with. Thus, a `NULL` pointer is characterised by all flags being zero in the polyhedron and an explicit NULL tag in $\mathcal{A}$ is not necessary anymore. The above interpretation simplifies the evaluation of linear expressions: Consider calculating `e=p+q-r` and suppose that $A(x_p) = \{\langle a_1, f_1^p\rangle, \ldots \langle a_k, f_k^p\rangle\}$ and similarly for x_q, x_r and x_e. Then the result is calculated by updating the polyhedral variables $x_e, f_1^e, \ldots f_k^e$ of `e` as follows:

$$\begin{array}{cc} x_e = x_p + x_q - x_r \\ f_1^e = f_1^p + f_1^q - f_1^r \\ \vdots \qquad\qquad \vdots \\ f_k^e = f_k^p + f_k^q - f_k^r \end{array}$$

Thus, rather than matching certain common patterns for pointer subtraction and pointer offset expressions, the Boolean flags can be added component-wise. Since this makes it possible to add several Boolean flags together, the value of a particular f_i^e might not be in $[0, 1]$. As a consequence, an access through the pointer `e` with offset x_e in state P has to be checked as follows:

- $P \sqsubseteq [\![\{0 \leq f_1^e \leq 1, \ldots 0 \leq f_k^e \leq 1\}]\!]$: `e` contains each l-value at most once.
- $P \sqsubseteq [\![f_1^e + \ldots + f_k^e \leq 1]\!]$: `e` contains at most one l-value.
- $P \sqsubseteq [\![f_1^e + \ldots + f_k^e \geq 1]\!]$: `e` is not `NULL`.

The semantic function for the pointer access is then executed k times, once for each target a_i and state $P \sqcap [\![\{f_i^e = 1\}]\!]$ where $i = 1, \ldots k$.

Evaluating a conditional $a\ op\ b$ where $op \in \{<, \leq, =, \neq, \geq, >\}$ has to take the points-to sets of the expressions a and b into account. Suppose that the points-to sets of a and b are given by $A(x_a) = \{\langle a_1, f_1^a\rangle, \ldots, \langle a_k, f_k^a\rangle\}$ and $A(x_b) = \{\langle a_1, f_1^b\rangle, \ldots, \langle a_k, f_k^b\rangle\}$ and that no more than one flag is set in each expression. The semantics of $a\ op\ b$ in the state P can then be calculated by considering different combinations of flags $f_1^a, \ldots f_k^a$ and flags $f_1^b, \ldots f_k^b$. For instance, for all states $P' = P \sqcap [\![\{f_1^a = f_1^b = 0, \ldots f_{i-1}^a = f_{i-1}^b = 0, f_i^a = f_i^b = 1, f_{i+1}^a = f_{i+1}^b = 0, \ldots f_k^a = f_k^b = 0\}]\!]$, the comparison can be evaluated as $P' \sqcap [\![a\ op\ b]\!]$ since the addresses that are implicitly present in each side of the condition are the same (unless $|\rho(a_i)| > 1$, which indicates that a_i summarises several addresses). All combinations in which the flag set for a is different to that from b indicate that different pointers are compared which can be flagged as erroneous. In case a pointer is compared with a value, the address contained in the pointer has to be made explicit by adding a range of possible addresses to the expression. Possible addresses of variables are typically $[4096, 2^{32} - 1]$ on a 32-bit machine, since no variable can be stored in the first page of memory. Thus, given the test `p==NULL` and a points-to set of `p` of $A(x_p) = \{\langle a, f\rangle\}$, the input state P can be transformed to $P_1 \sqcup P_2$ where $P_1 = P \sqcap [\![\{f = 1, x_p + [4096, 2^{32} - 1] = 0\}]\!]$ and $P_2 = P \sqcap [\![\{f = 0, x_p = 0\}]\!]$. Here, P_1 is empty if the pointer `p` has a zero offset, in which case $P_1 \sqcup P_2 \sqsubseteq [\![\{f = 0\}]\!]$, that is, `p` has an empty points-to set.

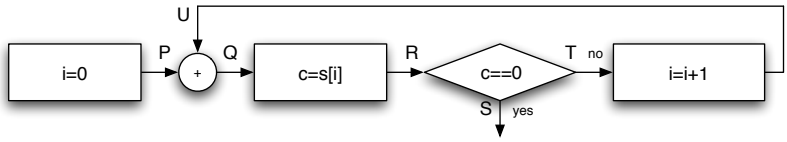
This concludes the presentation of the abstract semantics in the context of points-to sets that are guarded by Boolean flags. We now present a practical problem in which refined points-to information is key to a successful verification.

4.2 Boolean Flags and String Buffer Analysis

Consider the task of advancing a pointer `s` by executing the loop `while(*s) s++`. Suppose `s` points to a string buffer whose length is given by the first element with value 0. For ease of presentation, we augment and expand the loop as follows:

```
char s[11] = "the␣string";
int i = 0;
while (true) {
  c = s[i];
  if (c==0) break;
  i = i+1;
};
```

The task is to check that the string buffer `s` is only accessed within bounds. In order to simplify the presentation, we define the polyhedral operations and discuss the stability of the fixpoint. To this end, let $n \in \mathcal{X}$ represent the index of the first zero in `s`, i.e. the position of the ASCII character NUL. We decorate the control flow graph of the above loop with polyhedra P, Q, R, S, T, U as follows:



The initial values of the program variables is described by $P = [\![\{i = 0, n = 10\}]\!]$. The join of P and the back edge that is decorated by $U \in \mathit{Poly}$ is defined as $Q = P \sqcup U$. To verify that the array access `s[i]` is within bounds, we compute $Q' = Q \sqcap [\![\{0 \leq i \leq 10\}]\!]$ and issue a warning if $Q \not\sqsubseteq Q'$. We use the projection operator $\exists_{x_i}(P) = \{\langle x_1, \ldots x_{i-1}, v, x_{i+1}, \ldots x_n\rangle \mid \langle x_1, \ldots x_n\rangle \in P, v \in \mathbb{Q}\}$ to remove all information on x_i. The following definition of R is explained below:

$$\begin{aligned} R = &(\exists_c(Q) \sqcap \{i < n, 1 \leq c \leq 255\}) \\ &\sqcup (\exists_c(Q) \sqcap \{i = n, c = 0\}) \\ &\sqcup (\exists_c(Q) \sqcap \{i > n, 0 \leq c \leq 255\}) \end{aligned}$$

We assume that no information is tracked about the contents of the array, except for the NUL character at position n. The above definition of R therefore defines the read character c in terms of the value of the pointer offset i in Q and restricts c to the desired range. Specifically, the value of c is restricted to $[1, 255]$ if $i < n$, it is set to 0 if $i = n$ and to $[0, 255]$ if $i > n$. The last three equations that comprise the system are given by the following definitions:

$$\begin{aligned} S &= R \sqcap [\![c = 0]\!] \\ T &= (R \sqcap [\![c < 0]\!]) \sqcup (R \sqcap [\![c > 0]\!]) \\ U &= \{\langle x_1, \ldots i + 1, \ldots x_n\rangle \mid \langle x_1, \ldots i, \ldots x_n\rangle \in T\} \end{aligned}$$

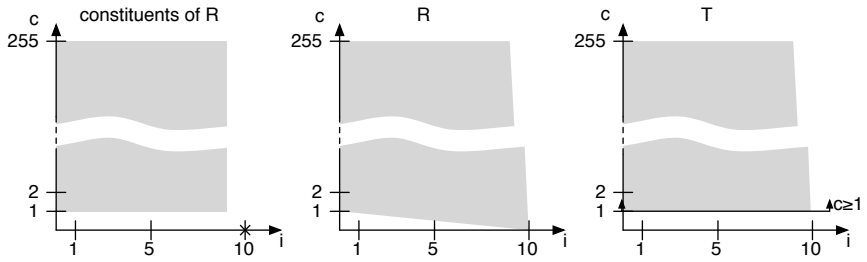


Fig. 6. The fixpoint of the state spaces after accessing the buffer.

The test `c!=0` is implemented by joining the two state spaces $R \sqcap [\![c < 0]\!]$ and $R \sqcap [\![c > 0]\!]$. Note that $R \sqcap [\![c < 0]\!] = \emptyset$ as the definition of R confines c to the range $[0, 255]$. Figure 6 shows the state space R at the fixpoint. Specifically, the first graph shows the contribution of the first and second equation, the second graph shows the join of both states. The third graph shows the calculation of T in that R is intersected with $c > 0 \equiv c \geq 1$. The intersection reduces the upper bound of i to less than 10 and integral tightening ensures that $i \leq 9$ before i is incremented by one to define U. Since $U \sqsubseteq Q$, a fixpoint has been reached.

Observe that the character c acts as a Boolean flag, even though in the non-zero state c takes on many values. Specifically, c distinguishes the state where $i < n$ and $i = n$. Furthermore, all equations that act as guards in the definition of R have coefficients in $\{1, -1\}$. Thus, this Boolean behaviour can be exploited by splitting the control flow according to the value of c which makes it possible to prove that the access `s[i]` is within bounds using the weaker Octagon domain.

4.3 Accessing Several String Buffers

The last section demonstrated that, within the domain of polyhedra, an invariant such as $p < n$ (the access position lies in front of NUL position) can be recovered through the relational information in the domain merely by intersection with the loop invariant $c \neq 0$ (the read character is not NUL) which acts like a Boolean flag. This section demonstrates how string buffer analysis suffers from precision loss when no relational information exists between the polyhedral domain and the points-to domain. In particular, the following example demonstrates that the fixpoint is missed if the pointer may point to two different strings.

```
char s[10] = "Spain";
char v[10] = "Valencia";
char* u = v;
if (rand()) u = s;
char *p = u;
while (*u) u++;
printf("length␣is␣%i\n", (u-p));
```

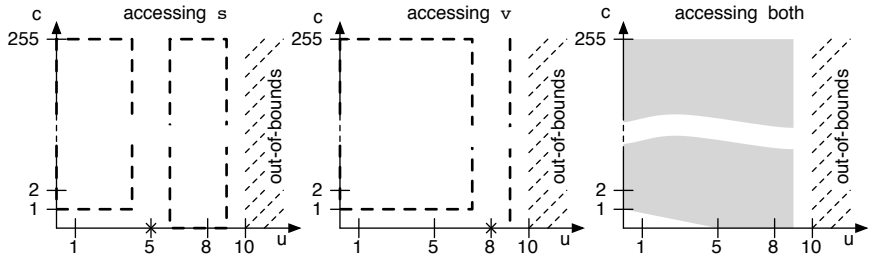


Fig. 7. Finding the NUL position in more than one buffer at a time is impossible without stating that a points-to set changes from a certain iteration onwards.

The shown code sets the pointer `u` to either point to the string `"Spain"` or to `"Valencia"`, depending on a random number. A backup of `u` is stored in `p` before `u` is advanced to the NUL position in the string buffer. The last statement prints the length of the accessed string by calculating the pointer difference `u-p`.

Let $u \in \mathcal{X}$ denote the offset of `u` and let a_v and a_s denote the abstract addresses of the variables `v` and `s`, respectively. Thus, when using a normal flow-sensitive points-to analysis, the points-to set of `u` can be described as $A(u) = \{a_v, a_s\}$. In this case, analysing the read access `*u` in the loop condition has to assume that both string buffers might be accessed which can be implemented by evaluating the semantic equation of R in Sect. 4.2 twice, once with the NUL position of `s` and once with that of `v` and joining the result. Let $c \in \mathcal{X}$ denotes the value of the read character. The value of c with respect to the access position u is shown in Fig. 7. While the analysis can infer that c may be zero as soon as $u \geq 5$ (from the first graph), the analysis correctly infers that the loop may continue to increment u since the NUL position in `v` at index 8 has not been reached yet. However, once this index is reached, the value of c resulting from accessing `s` can either be zero or non-zero, as the access lies past the first known NUL position. Hence, $c > 0$ and the analysis assumes that the loop can continue to iterate, leading to the state in the third graph. In the next iteration the analysis will emit a waring that the loop will access both buffers past their bound.

The example can readily be verified using the revised points-to abstraction presented in Sect. 4.1 such that $A(u) = \{\langle a_s, f_s\rangle, \langle a_v, f_v\rangle\}$. The statements before the loop ensure that $0 \leq f_v \leq 1, f_s = -f_v$. The semantics of the access `*u` is calculated by evaluating the equation for R firstly with $Q \sqcap [\![\{f_s = 1\}]\!]$ and `s` and secondly with $Q \sqcap [\![\{f_v = 1\}]\!]$ and `v`. Analysing the loop twice, once with $f_s = 1$ and once with $f_v = 1$ will implicitly restrict the access to one of the string buffers, thereby inferring that the loop terminates with $u = 5$ and $u = 8$, respectively. Alternatively, the loop may be analysed once. In this case the analysis infers a linear relationship between the pointer offset and f_v in that $u \geq 6$ implies $f_v = 1$ and $f_s = 0$. Thus, in the next evaluations of the loop, calculating $Q \sqcap [\![f_s = 1]\!]$ before evaluating the access to the buffer a_s will restrict u to values smaller than 5. Hence, termination of the loop is inferred when $u = 5, f_s = 1$ or $u = 8, f_v = 1$: The states are $S = [\![\{c = 0, 3f - u = 5, u \geq 5\}]\!]$ and $R = [\![\{c+255u-765f_v \leq 1275, c \leq 255, 5c+u \geq 5, 8c+u-3f_v \geq 5, u \geq 0\}]\!]$.

5 Discussion and Conclusion

The last section demonstrated how Boolean flags help to refine the abstract semantics of pointer accesses, yielding a more precise abstraction. From Sect. 3 it follows that adding a Boolean flag is equivalent to analysing those parts of a program twice for which the Boolean flag can take on both values. Thus, the natural question to ask is which approach is cheaper. In order to get an initial answer to this question, we implemented the semantics of the above loop using both techniques. As numeric domain, we used general polyhedra [20] as the loop invariant requires three variables and is therefore beyond the reach of the TVPI domain [22]. Interestingly, inferring the given invariant does not require any integral tightening techniques as the three guards $i < n$, $i = n$ and $i > n$ in the semantic equation defining R discard rational solutions since $i < n$ merely abbreviates $i \leq n - 1$. For comparable results, neither widening nor any approximation of the convex hull was applied. Since the analysis of the loop does not take very long, we calculated the fixpoint one million times in order to get more accurate results. Nevertheless, any variation in the implementation can have an immense effect on the given numbers, such that they have to be taken with a grain of salt. Analysing the loop twice, once for `s`, once for `v`, requires $7.7\mu s$ per two fixpoint calculations. Analysing the loop once using a Boolean flag requires $5.6\mu s$ for one fixpoint calculation. Interestingly, if both strings are of length 8, the analysis speeds up to $4.8\mu s$ as the state contains more equalities that can be factored out. While our experiment suggests that using flags is preferable, their exponential cost may grind an analyser to a halt. For example, using the character c to distinguish states quickly leads to inequalities with very large coefficients. Integral tightening as implementable in the TVPI domain could help to reduce the size of coefficients. However, the TVPI domain is too weak to verify the example above. One way of making TVPI polyhedra more expressive is to track linear inequalities over more than two variables symbolically. Such inequalities would take on the role of propagation rules in the context of Constraint Handling Rules [9]. Reconsider the first example on string buffers where $c = 0$ implies that $p < n$. In case n is not constant, but may take on a range of values, the termination of the loop in Sect. 4.2 cannot be proved since three variables are necessary to express the invariant [18]. Instead of re-analyzing the code for $1 \leq c \leq 255$ and $c = 0$ it is possible to track the implication $c = 0 \Rightarrow p < n$ and automatically insert the inequality $p < n$ whenever the interval of c is restricted to $[0, 0]$. Future work will asses if this trick can also be applied to Boolean flags.

In conclusion, we proposed to use Boolean flags to separate states in a single convex polyhedron as an alternative to partitioning the set of traces by reanalysing code. We demonstrated their use in applications such as points-to analysis where Boolean flags can be superior to analysing code twice.

References

1. R. Bagnara, P. M. Hill, and E. Zaffanella. Not Necessarily Closed Convex Polyhedra and the Double Description Method. *FAC*, 17(2):222–257, 2005. Springer.

2. R. Bagnara, P. M. Hill, and E. Zaffanella. An Improved Tight Closure Algorithm for Integer Octagonal Constraints, 2008.
3. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough Static Analysis of Device Drivers. In *European Systems Conference*, pages 73–85. ACM, 2006.
4. T. Ball and S. K. Rajamani. Bebop: A Symbolic Model Checker for Boolean Programs. In *SPIN Workshop on Model Checking and Software Verification*, pages 113–130, London, UK, 2000. Springer.
5. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN Workshop on Model Checking of Software*, pages 103–122, New York, NY, USA, 2001. Springer.
6. P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.
7. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyzer. In *European Symposium on Programming*, pages 21–30, Edinburgh, Scotland, 2005. Springer.
8. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Constraints among Variables of a Program. In *Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press.
9. Thom Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.
10. D. Gopan and T. W. Reps. Low-Level Library Analysis and Summarization. In *Computer Aided Verification*, volume 4590 of *LNCS*, pages 68–81. Springer, 2007.
11. W. Harvey. Computing Two-Dimensional Integer Hulls. *SIAM Journal on Computing*, 28(6):2285–2299, 1999.
12. N. Heintze and O. Tardieu. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *Programming Language Design and Implementation*, pages 254–263, 2001.
13. J. C. Lagarias. The Computational Complexity of Simultaneous Diophantine Approximation Problems. *SIAM Journal on Computing*, 14(1):196–209, 1985.
14. L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In M. Sagiv, editor, *European Symposium on Programming*, volume 3444 of *LNCS*, pages 5–20. Springer-Verlag, 2005.
15. A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.
16. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.
17. A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley&Sons, 1998.
18. A. Simon. *Value-Range Analysis of C Programs*. Springer, 2008. To Appear.
19. A. Simon and A. King. Analyzing String Buffers in C. In *Algebraic Methodology and Software Technology*, volume 2422 of *LNCS*, pages 365–379. Springer, 2002.
20. A. Simon and A. King. Exploiting Sparsity in Polyhedral Analysis. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium*, volume 3672 of *LNCS*, pages 336–351. Springer, September 2005.
21. A. Simon and A. King. Taming the Wrapping of Integer Arithmetic. In G. File and H. R. Nielson, editors, *Static Analysis Symposium*, volume 4634 of *LNCS*, pages 121–136. Springer, August 2007.
22. A. Simon, A. King, and J. M. Howe. Two Variables per Linear Inequality as an Abstract Domain. In M. Leuschel, editor, *Logic-based Program Synthesis and Transformation*, volume 2664 of *LNCS*, pages 71–89. Springer, September 2003.