# Improving Static Analyses of C Programs with Conditional Predicates

Sandrine Blazy[1], David Bühler[2], and Boris Yakobowski[2]

[1] IRISA - University of Rennes, France
sandrine.blazy@irisa.fr
[2] CEA, LIST, Software Safety Lab, 91191 Gif-sur-Yvette, France
{david.buhler,boris.yakobowski}@cea.fr

**Abstract.** Static code analysis is increasingly used to guarantee the absence of undesirable behaviors in industrial programs. Designing sound analyses is a continuing trade-off between precision and complexity. Notably, dataflow analyses often perform overly wide approximations when two control-flow paths meet, by merging states from each path. This paper presents a generic abstract interpretation based framework to enhance the precision of such analyses on join points. It relies on *predicated* domains, that preserve and reuse information valid only inside some branches of the code. Our predicates are derived from conditionals statements, and postpone the loss of information. The work has been integrated into Frama-C, a C source code analysis platform. Experiments on real code show that our approach scales, and improves significantly the precision of the existing analyses of Frama-C.

## 1 Introduction

Formal program verification is an increasingly sought-after approach to guarantee the absence of undesirable behaviors in software. Static code analysis has already shown its industrial applicability to prove safety properties on critical or embedded code. Still, so as to remain tractable, these analyses involve sound but incomplete approximations of a program behavior. This may lead to false alarms, when some required properties cannot be proved statically even though they always hold at runtime. Abstract interpretation [5,6] is a well-known framework to over-approximate program executions through *abstractions* of the most precise mathematical characterization of the program. Designing such abstractions is a continuing trade-off between precision and efficiency.

*Flow-sensitivity*, which allows to infer static properties that depend on program points, is often considered as a prerequisite to obtain a precise program analysis. More agressive analyses are *path-sensitive*: the analysis of a program statement depends on the control-flow path followed to reach this statement. Nevertheless, most analyses sacrifice full path-sensitivity and perform approximations when two control-flow paths meet. Those approximations may lead to a significant loss of precision, and may preclude inferring some interesting properties of the program.

F. Lang and F. Flammini (Eds.): FMICS 2014, LNCS 8718, pp. 140–154, 2014.

```
1    if (flag1)
2      { fd1 = open(path1);
3        if (fd1==-1) exit(); }
4    [...] // code 1
```

```
if (flag2)                                5
  { fd2 = open(path2);                    6
    if (fd2==-1) {                        7
      if (flag1) close(fd1);              8
      exit(); } }                         9
[...] // code 2                           10
if (flag1) close(fd1);                    11
if (flag2) close(fd2);                    12
```

**Fig. 1.** Example of interleaved conditionals

Consider as an example the code fragment of Fig. 1. Proving that the three calls to the `close` function are correct, i.e. that the corresponding `fd` variable has been properly created, heavily relies on the possible values for the `flag1` and `flag2` variables. An analysis that does not keep track of the relation between `flag1` and `fd1` on the one hand, and `flag2` and `fd2` on the other hand, will not be able to prove that the program is correct.

In this paper, we define an analysis in which information about the conditionals that have been encountered so far is retained using boolean predicates. These predicates guard the values inferred about the program. Our analysis is parameterized by a pre-existing analysis domain, which we use to derive a new *predicated* analysis. More precisely, we propagate two kinds of information that are not present in the original domain: a context and an implication map.

1. A *context* is a boolean predicate synthesized from the guards of the conditionals that have been reached so far, and that is guaranteed to hold at the current program point. In our example, at the beginning of line 8, the context would be $\texttt{flag2} \wedge (\texttt{fd2} = -1)$.
2. An *implication map* is a set of facts from the original analysis domain, guarded by boolean predicates. Each fact is guaranteed to hold when its guard holds. Implication maps postpone the loss of precision usually present at join points. In our example, assuming the existence of an analysis that verifies the validity of file descriptors, the implication map after line 6 would consist of the two following implications:

$$\texttt{flag1} \mapsto \mathsf{valid\_fd}\,(\texttt{fd1}) \qquad \texttt{true} \mapsto \mathsf{valid\_fd}\,(\texttt{fd2}) \vee (\texttt{fd2} = -1)$$

   The first implication results from the analysis of the conditionals at lines 1-3; it precisely models the information we need between `flag1` and `fd1`. The second implication is simply the postcondition of the `open` function, which holds unconditionnaly: either `open` succeeds, or it fails with a return code of $-1$.

Our framework, based on abstract interpretation, is generic. We also integrated it into Frama-C, a modular platform dedicated to the analysis of C code [8]. Frama-C provides various sound analyses based on abstract interpretation, deductive verification or testing, implemented by a collection of plugins built around a common kernel. These plugins collaborate through logical properties expressed in ACSL, a C specification language [1,4]. Among them, the value analysis plugin [9,3] performs a forward dataflow analysis over intricate low-level

abstract domains to compute an over-approximation of the possible values of variables at each program point. It aims at ensuring the absence of run-time errors in a given program. Our experiments show that predicated analyses over much simpler domains may significantly enhance and complement the results of the value analysis.

**Related Work.** Different approaches have been proposed in the litterature to solve instances of the problem we are addressing. *Trace-partitioning* [14] and *boolean partitioning* [7] would keep separate the different execution traces coming from the conditionals on `flag1` and `flag2` in Fig. 1. One downside is that `code 1` may need to be analyzed twice, and `code 2` up to four times. To avoid a blow-up in analysis time, the analyzer would need to reuse some parts of previous analyses. However, this requires a modular analysis and significant implementation efforts. Also, traces should be merged when it is no longer useful to keep them seperate. Syntactic criteria need to be used to detect such merge points. Conversely, trace partitioning can be used to unroll loop symbolically, something our approach does not handle. *Predicate abstraction* [11] would propagate a single fact along all execution paths, but this fact may be arbitrarily complex. In particular, the predicate is found incrementally, and refined until it is sufficient to guarantee the property under consideration. In our example, the predicate would likely link `flag1`, `fd1`, `flag2` and `fd2`. Using this approach, [10] shows how to transform any existing dataflow analysis into a predicated one, the predicates being found by successive refinement iterations. Still, finding the proper predicate may be abitrarily complex, resulting in hard to predict analysis times. Also, the refinement phase requires decidable theories and powerful decision procedures to find the counter-examples from which the predicate is deduced. We instead chose to limit ourselves to first-order predicates relating the conditionals present in the program.

The remainder of this paper is organized as follows. First, Sect. 2 introduces our language, simplified for the sake of illustration. Section 3 defines predicated domains and explains how to build a predicated analysis over a standard dataflow analysis, which we further improve in Sect. 4. Sect. 5 describes two domains that we used to validate our framework. Then, Sect. 6 presents the experimental evaluation of our practical implementation. Finally, Sect. 7 draws some conclusions.

## 2 A Generic Abstract Interpretation Based Framework

Our static analysis is based on abstract interpretation [5,6], and handles the whole C language. However, for the sake of brevity, we only present here a toy language. Abstract interpretation links a very precise, but generally undecidable, *concrete* semantics, to an *abstract* decidable one – the abstract semantics being a sound approximation of the concrete one. This section first defines the syntax of our toy language, then its concrete and abstract semantics.

*Syntax.* Figure 2 presents the syntax of our language. Programs operate over a fixed, finite set of variables $\mathcal{V}$ whose values belong to an unspecified set $\mathbb{V}$.

$$
\begin{array}{lll}
e \in \mathbf{exp} ::= x & x \in \mathcal{V} & \mathtt{i} \in \mathbf{stmt} ::= x := e \\
\quad\mid v & v \in \mathbb{V} & \quad\mid c \triangleleft \\
\quad\mid e \star e & & \\
c, p \in \mathbb{C} ::= e \mid \neg c \mid c \wedge c \mid c \vee c & & P \in \mathbf{prog} \triangleq \mathcal{P}(\mathbb{N} \times \mathbf{stmt} \times \mathbb{N})
\end{array}
$$

**Fig. 2.** Syntax of our language

Expressions are either variables, constants, or the application of a binary operator $\star$ to expressions. We stratify expressions and conditionals, the truth value of an element of $\mathbb{V}$ being given by a mapping $\mathsf{T}$ from $\mathbb{V}$ to booleans. Statements are either assignments, or assume filters that halt execution when the condition does not hold. A program is represented by its control-flow graph where nodes are integer-numbered program points and edges are labelled by statements. By convention, the program starts at node 0. Encoding standard program constructs such as `if` or `for` in such graphs is immediate and not detailed in this paper. For clarity, we write our examples using a C-like syntax.

*Concrete Semantics.* A concrete state of the program at a node $n$ of its control-flow graph is described by an environment $\rho \in \mathbb{V}^{\mathcal{V}}$ assigning a value to each variable. The semantics $[\![e]\!]_\rho$ (resp. $[\![c]\!]_\rho$) of an expression $e$ (resp. a condition $c$) is its evaluation in the environment $\rho$, and implicitely depends on the evaluation of the operators $\star$.

Our concrete semantics maps each program node $n$ to the set $\mathbb{S}\,(n)$ of all possible environments at this point; hence our semantics is a function in $\mathcal{P}\left(\mathbb{V}^{\mathcal{V}}\right)^{\mathbb{N}}$. The semantics $[\![\mathtt{i}]\!]$ of a statement $\mathtt{i}$ is a transfer function over a set of states, described in the first equalities of Fig. 3a. After an assignment on $x$, $x$ is bound in the new states to the evaluation of the expression. Assume filters block evaluation, only allowing states in which the condition holds. The concrete semantics of the entire program $P$ is then the smallest solution of the rightmost equations of Fig. 3a.

*Abstract Semantics* Abstract interpretation based analyses rely on an abstract domain $\mathcal{L}$, whose computable elements model a set of concrete states at a given program point. Such abstract domains must provide:

- a partial order $\sqsubseteq_{\mathcal{L}}$ according to the precision of abstract states,
- a monotone *concretization* function $\gamma_{\mathcal{L}}$ from $\mathcal{L}$ to $\mathcal{P}\left(\mathbb{V}^{\mathcal{V}}\right)$, linking the abstract states to the concrete ones,
- greatest and smallest elements $\top_{\mathcal{L}}$ and $\bot_{\mathcal{L}}$, such that $\gamma_{\mathcal{L}}(\top_{\mathcal{L}}) = \mathbb{V}^{\mathcal{V}}$ and $\gamma_{\mathcal{L}}(\bot_{\mathcal{L}}) = \emptyset$,
- sound approximations $\sqcup_{\mathcal{L}}$ and $\sqcap_{\mathcal{L}}$ of union and intersection of concrete states,
- sound abstract transfer functions $[\![\mathtt{i}]\!]^\sharp_{\mathcal{L}}$ from $\mathcal{L}$ to $\mathcal{L}$ that approximate the concrete semantics.

The correction theorems for the soundness of the abstract semantics are stated in the leftmost column of Fig. 3b. The abstract semantics is the least solution of the system of equations in the rightmost column. The soundness properties

**(a) Concrete semantics**

$$\begin{array}{l|l} [\![x := e]\!]\,(S) \triangleq \{\rho\,[x \mapsto [\![e]\!]_\rho] \mid \rho \in S\} & \mathbb{S}\,(0) \triangleq \mathbb{V}^{\mathcal{V}} \\ \quad [\![c \triangleleft]\!]\,(S) \triangleq \{\rho \mid \rho \in S \wedge \mathsf{T}\,([\![c]\!]_\rho) = \mathtt{true}\} & \mathbb{S}\,(n) \triangleq \bigcup_{(m,\mathtt{i},n)\in P} [\![\mathtt{i}]\!]\,(\mathbb{S}\,(m)) \end{array}$$

**(b) Abstract semantics**

$$\begin{array}{l|l} \gamma_{\mathcal{L}}(\top_{\mathcal{L}}) = \mathbb{V}^{\mathcal{V}} & \mathbb{S}^\sharp_{\mathcal{L}}\,(0) \triangleq \top_{\mathcal{L}} \\ \gamma_{\mathcal{L}}(l_1) \cup \gamma_{\mathcal{L}}(l_2) \subseteq \gamma_{\mathcal{L}}(l_1 \sqcup_{\mathcal{L}} l_2) & \mathbb{S}^\sharp_{\mathcal{L}}\,(n) \triangleq \bigsqcup_{\mathcal{L}} \left\{ [\![i]\!]^\sharp_{\mathcal{L}} \left(\mathbb{S}^\sharp_{\mathcal{L}}\,(m)\right) \mid (m,\mathtt{i},n) \in P \right\} \\ \quad [\![\mathtt{i}]\!]\,(\gamma_{\mathcal{L}}(l)) \subseteq \gamma_{\mathcal{L}}([\![\mathtt{i}]\!]^\sharp_{\mathcal{L}}\,(l)) & \end{array}$$

**Fig. 3.** Concrete and abstract semantics

ensure that any solution is a correct approximation of the concrete semantics. In practice, such systems are solved by iterative data-flow analysis [13,2].

**Lemma 1.** *All behaviors of the concrete semantics are captured by the abstract one. That is,* $\forall n \in P, \mathbb{S}\,(n) \subseteq \gamma_{\mathcal{L}}\left(\mathbb{S}^\sharp_{\mathcal{L}}\,(n)\right)$

We also define an operator called $\mathsf{deps}$ from expressions to sets of variables $\mathcal{P}(\mathcal{V})$, that will be useful when computing memory footprints. $\mathsf{deps}\,(e)$ is the set of variables on which the evaluation of $e$ depends. On our toy language, this is the set of variables syntactically present in $e$. However, in a language with pointers, $\mathsf{deps}\,(e)$ usually depends on the current program point.

## 3 Predicated Analyses

This section presents our predicated analysis. We first define the domain that will represent its abstract states, then the transfer function on statements.

### 3.1 Predicated Domains

Our analysis derives a *predicated* analysis on top of an abstract domain $\mathcal{L}$. The additional information is two-fold. First, we add a boolean predicate $c \in \mathbb{C}$, called the *context*, standing for a set of facts that we know to hold at the current program point. Second, we add a mapping $I$ from predicates in $\mathbb{C}$ to elements of $\mathcal{L}$, called a *map*. Maps stand for implications from guards to values; hence they contain information that are *conditional*: $I(p) = l$ implies that $l$ is a correct approximation of the state as soon as $p$ is verified. We use the syntax $\lambda p.l$ to denote maps, and write $\langle p \to l \rangle$ for a value $l$ guarded by a predicate $p$.

We say that $\langle p \to l \rangle$ is *trivial* when $l = \top_{\mathcal{L}}$, as the value $\top_{\mathcal{L}}$ brings no information whatsoever. In order to have a decidable semantics, we restrict ourselves to maps in which all but a finite number of implications are trivial. To guarantee that our inclusion operator is antisymmetric, we only consider pairs of a context and a map in which implications that contradict the context or are redundant with another (stronger) implication are trivial. Formally, for any

$$
\begin{aligned}
\top_{\texttt{pred}} &\triangleq (\texttt{true}, \lambda p.\top_{\mathcal{L}}) \\
\bot_{\texttt{pred}} &\triangleq (\texttt{false}, \lambda p.\top_{\mathcal{L}}) \\
(c_1, I_1) \sqsubseteq_{\texttt{pred}} (c_2, I_2) &\triangleq c_1 \Rightarrow c_2 \wedge \forall p \in \mathbb{C},\ (c_1, I_1) \blacktriangleright \langle p \rightarrow I_2\,(p_2)\rangle \\
(c, I) \blacktriangleright \langle p \rightarrow l\rangle &\triangleq \neg\,(c \wedge p) \vee \left(\exists p', p \Rightarrow p' \wedge I(p') \sqsubseteq_{\mathcal{L}} l\right) \\
(c_1, I_1) \sqcup_{\texttt{pred}} (c_2, I_2) &\triangleq \mathsf{canonize}\,(c_1 \vee c_2\ ,\ \lambda p.\ (l_\cup\,(p) \sqcap_{\mathcal{L}} l_1\,(p) \sqcap_{\mathcal{L}} l_2\,(p))) \\
&\text{where } \begin{cases} l_\cup\,(p) = \bigsqcap_{\mathcal{L}} \{I_1(p_1) \sqcup_{\mathcal{L}} I_2(p_2) \mid p \equiv p_1 \wedge p_2\} \\ l_1\,(p) = \bigsqcap_{\mathcal{L}} \{I_1(p_1) \mid p \equiv \neg c_2 \wedge p_1\} \\ l_2\,(p) = \bigsqcap_{\mathcal{L}} \{I_2(p_2) \mid p \equiv \neg c_1 \wedge p_2\} \end{cases} \\
\gamma_{\texttt{pred}}\,(c, I) &\triangleq \{\rho \mid [\![c]\!]_\rho \wedge \forall p \in \mathbb{C},\ [\![p]\!]_\rho \Rightarrow \rho \in \gamma_{\mathcal{L}}\,(I(p))\}
\end{aligned}
$$

**Fig. 4.** Definition of $\mathcal{L}^{\texttt{pred}}$, the predicated domain over $\mathcal{L}$

$p \in \mathbb{C}$, a pair $(c, I)$ must verify respectively $\neg(p \wedge c) \Rightarrow (I(p) = \top_{\mathcal{L}})$ and $\forall p' \not\equiv p, (p \Rightarrow p' \wedge I(p') \sqsubseteq_{\mathcal{L}} I(p)) \Rightarrow (I(p) = \top_{\mathcal{L}})$. A context and a map that do not verify these last two properties can always be canonized into a pair that does, by mapping the contradictory or redundant implications of $I$ to $\top_{\mathcal{L}}$. We write $\mathsf{canonize}$ this operation. We call *context-implication-map pair,* ranged over by $\Phi$ and abbreviated as *CI-pair*, a context and a map that verify all these properties. CI-pairs will represent the abstract state of our predicated analysis.

We define $\mathcal{L}^{\texttt{pred}}$, the *predicated domain over* $\mathcal{L}$, as the set of CI-pairs equipped with the operations of Fig. 4. $\top_{\texttt{pred}}$ (resp. $\bot_{\texttt{pred}}$) denotes the most general (resp. most restrictive) context. Both $\top_{\texttt{pred}}$ and $\bot_{\texttt{pred}}$ are made up of trivial implications only. CI-pairs are ordered by the relation $\sqsubseteq_{\texttt{pred}}$: $(c_1, I_1)$ is more precise than $(c_2, I_2)$ when $c_1$ is stronger than $c_2$, and when $(c_1, I_1)$ implies all the implications of $I_2$. This last property is defined using an auxiliary relation $\blacktriangleright$ stating that a CI-pair *verifies* an implication. The relation $(c, I) \blacktriangleright \langle p \rightarrow l\rangle$ holds when either $p$ contradicts $c$, or there exists an implication of $I$ stronger than $\langle p \rightarrow l\rangle$.

*Example 1.* Consider the example of Fig. 5, where $\mathcal{L}$ is a basic interval domain. The notation $[i]$ stands for the singleton interval $[i; i]$. We write $\Phi_i$ for the state at the end of line $i$, its context and non-trivial implications being shown in the two rightmost columns. For instance, $\Phi_3$ maps the three variables assigned at lines 1-3 to their respective values, under the `true` guard. The relation $\Phi_7 \sqsubseteq_{\texttt{pred}} \Phi_8$, that relates the state at the end of the else branch and after the first conditional, holds. First, the implications guarded by `true` and $\neg c$ in $\Phi_8$ are implied by the `true`-guarded implication of $\Phi_7$. Then, the implication guarded by $c$ contradicts the context of $\Phi_7$. Finally, all trivial implications of $\Phi_8$ are implied by the corresponding one in $\Phi_7$.

Let $\Phi_1 = (c_1, I_1)$ and $\Phi_2 = (c_2, I_2)$ be two CI-pairs. The join $\sqcup_{\texttt{pred}}$ between them is the smallest CI-pair whose context is implied by $c_1$ and $c_2$, and whose implications are verified by both $\Phi_1$ and $\Phi_2$. Its context is simply $c_1 \vee c_2$. Within the implication map, the operator $l_\cup$ combines implications of the two previous

```
1   x = 0;
2   y = 0;
3   v = 1;
4   if (c) {
5     x = v;
6   } else {
7     y = v;
8   }
9   w = 0;
10  if (c) {
11    c = 2;
12  }
```

| line | $\Phi_{\text{line}}$: state after the statement | |
|---|---|---|
| | context | implications |
| 3 | `true` | $\texttt{true} \mapsto v \in [1], x \in [0], y \in [0]$ |
| 5 | $c$ | $\texttt{true} \mapsto v \in [1], x \in [1], y \in [0]$ |
| 7 | $\neg c$ | $\texttt{true} \mapsto v \in [1], x \in [0], y \in [1]$ |
| 8 | $c \vee \neg c \equiv \texttt{true}$ | $\texttt{true} \mapsto v \in [1], x \in [0,1], y \in [0,1]$<br>$c \mapsto v \in [1], x \in [1], y \in [0]$<br>$\neg c \mapsto v \in [1], x \in [0], y \in [1]$ |
| 9 | `true` | $\texttt{true} \mapsto v \in [1], w \in [0], x \in [0,1], y \in [0,1]$<br>$c \mapsto v \in [1], w \in [0], x \in [1], y \in [0]$<br>$\neg c \mapsto v \in [1], w \in [0], x \in [0], y \in [1]$ |
| 10 | $c$ | $\texttt{true} \mapsto v \in [1], w \in [0], x \in [1], y \in [0]$ |
| 11 | `true` | $\texttt{true} \mapsto v \in [1], w \in [0], x \in [1], y \in [0], c \in [2]$ |
| 12 | `true` | $\texttt{true} \mapsto v \in [1], w \in [0], x \in [0,1], y \in [0,1]$<br>$c \mapsto v \in [1], w \in [0], x \in [1], y \in [0], c \in [2]$ |

**Fig. 5.** Example of an analysis using a predicated interval analysis

maps: the $\mathcal{L}$-join of values present under guards $p_1$ and $p_2$ respectively of $\Phi_1$ and $\Phi_2$ is kept under the new guard $p_1 \wedge p_2$. Conversely, the operators $l_1$ and $l_2$ preserve the values only present in $\Phi_1$ or $\Phi_2$ respectively. A value valid in $\Phi_1$ under a guard $p_1$ may be present in the join provided that the new guard negates $c_2$ (so that $\Phi_2$ also verifies the implication) – resulting in the guard $\neg c_2 \wedge p_1$. Values present in $\Phi_2$ are likewise present under guards that negate $c_1$. Note that this additional information from $\Phi_i$ is thrown away if all guards $p \wedge \neg c_j$ contradict the new context, i.e. whenever $c_i \Rightarrow c_j$.

*Example 2.* Consider again Fig. 5. We have $\Phi_8 = \Phi_5 \sqcup_{\texttt{pred}} \Phi_7$. The value implied by `true` in $\Phi_8$ comes from the operator $l_\cup$, and is equal to $I_5(\texttt{true}) \sqcup_{\mathcal{L}} I_7(\texttt{true})$. Conversely, the value implied by $c$ comes from the operator $l_1$, which negates the context of $\Phi_7$; furthemore, the value is exactly $\Phi_5(\texttt{true})$. Note that the intervals inferred in $\Phi_5$ and $\Phi_7$ are entirely retained, guarded by the negations of the converse contexts; no information is actually lost.

Finally, the concretization $\gamma_{\texttt{pred}}(c, I)$ of an element of the predicated domain is the set of states wherein $c$ is true and all implications of $I$ are valid.

### 3.2 Abstract Transfer Functions

We now define in Fig. 6 our abstract semantics for statements in $\mathcal{L}^{\texttt{pred}}$. The gist of the analysis is to apply the transfer functions of $\mathcal{L}$ to each of its elements in the map, which is carried out by the lift function, while new implications will be created by $\sqcup_{\texttt{pred}}$ for values that are present in only one branch at a junction point. However, to remain sound, we also need to invalidate predicates (either in the context or in a guard) whose truth values are possibly modified by a statement. Following standard dataflow terminology, we define a kill operator,

$$
\begin{aligned}
\mathsf{lift}\,(\mathtt{i},(c,I)) &\triangleq \left(c, \lambda p.\, [\![\mathtt{i}]\!]^\sharp_{\mathcal{L}}\,(I\,(p))\right)\\
\mathsf{kill}\,(x,(c,I)) &\triangleq \begin{pmatrix} \begin{cases} c & \text{if } x \notin \mathsf{deps}\,(c)\\ \mathtt{true} & \text{otherwise}\end{cases}\\ \lambda p.\, \begin{cases} I\,(p) & \text{if } x \notin \mathsf{deps}\,(p)\\ \top_{\mathcal{L}} & \text{otherwise}\end{cases}\end{pmatrix}\\
\mathsf{refine}\,(h,(c,I)) &\triangleq \mathsf{canonize}\left(c \wedge h, \lambda p. \bigsqcap\nolimits_{\mathcal{L}} \left\{I\,(p') \mid h \wedge p \Rightarrow p'\right\}\right)\\
[\![x := e]\!]^\sharp_{\mathtt{pred}}\,(\Phi) &\triangleq \mathsf{lift}\,(x := e, \mathsf{kill}\,(x,\Phi))\\
[\![c \triangleleft]\!]^\sharp_{\mathtt{pred}}\,(\Phi) &\triangleq \mathsf{lift}\,(c \triangleleft, \mathsf{refine}\,(c,\Phi))
\end{aligned}
$$

**Fig. 6.** Definition of the abstract semantics $[\![.]\!]^\sharp_{\mathtt{pred}}$

that removes contexts and implications depending on a certain variable $x$. This operator is used after an assignment $x := e$, as it modifies the value of $x$.

While kill and lift used in conjunction are sufficient to define a sound abstract semantics for $\mathcal{L}^{\mathtt{pred}}$, they never use the existing implications or enrich the context. Yet, the join operation retains specific information of each branch only when they have different non-`true` contexts. Thus, we define an operator refine that enriches the context by a new predicate $h \in \mathbb{C}$, supposed to be verified. This operator also learns information by simplifying the map according to $h$. More precisely, the valid value under a guard $p$ is the $\mathcal{L}$-meet of the elements implied by any guard weaker than $p \wedge h$.

Within our abstract semantics $[\![.]\!]^\sharp_{\mathtt{pred}}$, there are two natural places where refine may be used. First, after an assume statement $c\triangleleft$, the predicate $c$ holds by definition. Second, after a statement $x := e$, the equality $x = e$ holds (provided the value of $e$ does not depend on $x$). In practice, this second rule rapidly leads to the creation of intractable contexts. Hence, we only enrich our states on assume statements. Let us stress that any application of $\mathsf{refine}(h, \cdot)$ is sound, provided $h$ actually holds. Refining more or less aggressively results in a trade-off between precision and complexity.

*Example 3.* At line 4 in Fig. 5, in the branches of the conditional, the operator refine enriches the context according to the condition. After the conditional, the context reverts to `true` due to the join between $\Phi_5$ and $\Phi_7$. Note that despite the canonization, the join and the lift function duplicate the value of variables $v$ and $w$ at line 8 and 9 respectively. At line 10, on a conditional with the same condition $c$, the refine operator maps the `true` guard to $I_9(\mathtt{true}) \sqcap_{\mathcal{L}} I_9(c)$, as both `true` and $c$ are implied by the new context $c$. We have re-learnt the information known about $x$ and $y$ at line 5. Meanwhile, the $c$ guard becomes redundant with the `true` one, while $\neg c$ contradicts the context. Both implications are changed to trivial ones by the canonize operator. On line 11, $c$ is overwritten. Hence, the context $c$ is reset to `true` by the kill operator. Finally, upon exiting the conditional, we lose the information coming from the else branch, as negating the context `true` would result in a trivial implication, that would never hold.

But the information coming from the then branch is preserved under the guard $\neg\neg c \equiv c$.

**Lemma 2.** *Our predicated analysis over $\mathcal{L}^{\mathtt{pred}}$ is sound.*

$$\gamma_{\mathtt{pred}}(\Phi_1) \cup \gamma_{\mathtt{pred}}(\Phi_2) \subseteq \gamma_{\mathtt{pred}}\left(\Phi_1 \sqcup_{\mathtt{pred}} \Phi_2\right)$$
$$[\![\mathtt{i}]\!]\left(\gamma_{\mathtt{pred}}(\Phi)\right) \subseteq \gamma_{\mathtt{pred}}\left([\![\mathtt{i}]\!]^\sharp_{\mathtt{pred}}(\Phi)\right)$$

Moreover, we can state a stronger result, that links, at a program point $n$, the abstract semantics of $\mathbb{S}^\sharp_{\mathcal{L}}$ with its equivalent $\mathbb{S}^\sharp_{\mathtt{pred}}$ for $\mathcal{L}_{\mathtt{pred}}$.

**Theorem 1.** *Our predicated analysis is as precise as the non-predicated one.*

$$\forall n \in P, \textit{ given } (c_n, I_n) = \mathbb{S}^\sharp_{\mathtt{pred}}(n)\textit{ , then } I_n(\mathtt{true}) \sqsubseteq_{\mathcal{L}} \mathbb{S}^\sharp_{\mathcal{L}}(n)$$

Of course, the predicated analysis can be more precise. As an example, on line 10 of the program of Fig. 5, the non-predicated analysis would have inferred the value $I_9(\mathtt{true})$. Our own result – namely $I_{10}(\mathtt{true})$ – is much more precise.

## 4 Improving the Analysis

This section explains how to avoid computing guarded values that are needlessly redundant, and details some strategies to decrease the complexity of our analysis.

### 4.1 Avoiding Redundant Values

As previously remarked in example 3, our analysis keeps within implications more information than needed. Even though we avoid redundant implications in the map, some values of $\mathcal{L}$ may encode information partially present under weaker guards. Furthermore, the transfer function of the underlying domain may be costly and it is applied to every element of $\mathcal{L}$ in the map. In order to decrease the practical complexity of the predicated analysis, we require two additional features from the underlying domain $\mathcal{L}$.

1. A more lightweight transfer function $[\![\mathtt{i}, p]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}$ over statements $\mathtt{i}$, parameterized by the predicate $p$ that guards the processed value. This way, the analysis can be more precise on the `true` guard only and avoids the duplication of new information. Thus, $[\![\mathtt{i}, \mathtt{true}]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}$ may be defined as $[\![\mathtt{i}]\!]^\sharp_{\mathcal{L}}$, while $[\![\mathtt{i}, \cdot]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}$ applied to a non-`true` guard should be defined as a very imprecise operation, that only guarantees the soundness of the analysis on $\mathcal{L}$. Formally, we only require $[\![\mathtt{i}, \cdot]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}$ to be an over-approximation of $[\![\mathtt{i}]\!]^\sharp_{\mathcal{L}}$. The $\mathsf{lift}$ operator is then redefined as

$$\mathsf{lift}(\mathtt{i}, (c, I)) \triangleq \left(c, \lambda p.\ [\![\mathtt{i}, p]\!]^\sharp_{\mathcal{L}\times\mathbb{C}}(I(p))\right)$$

```
3     ...
4     if (c) {
5       x = v;
6     } else {
7       y = v;
8     }
9     w = 0;
10    ...
```

| line | context | implications after the statement |
|---|---|---|
| 8 | $c \vee \neg c \equiv$ `true` | $\texttt{true} \mapsto v \in [1]\,; x \in [0,1]\,; y \in [0,1]$ |
| | | $c \mapsto x \in [1]\,; y \in [0]$ |
| | | $\neg c \mapsto x \in [0]\,; y \in [1]$ |
| 9 | `true` | $\texttt{true} \mapsto v \in [1]\,; w \in [0]\,; x \in [0,1]\,; y \in [0,1]$ |
| | | $c \mapsto x \in [1]\,; y \in [0]$ |
| | | $\neg c \mapsto x \in [0]\,; y \in [1]$ |

**Fig. 7.** Analysis of Fig. 5 with factorization

2. A difference operation $\setminus_{\mathcal{L}}$ that discards information already contained in another element of $\mathcal{L}$, that we use to simplify implication maps. Ideally, $a \setminus_{\mathcal{L}} b$ should be as large as possible, while retaining all the information of $a$ not already present in $b$. To be sound, we require $a \sqsubseteq_{\mathcal{L}} a \setminus_{\mathcal{L}} b$. We define an operator $\mathsf{reduce}$, that simplifies each implication by all the values mapped to weaker guards, and we use it whenever we need to canonize a map (i.e. after a join or a refinement).

$$\mathsf{reduce}\,(I) \triangleq \lambda p.\, I\,(p) \setminus_{\mathcal{L}} \left(\bigsqcap\nolimits_{\mathcal{L}} \{I\,(q) \mid p \Rightarrow q, p \not\equiv q\}\right)$$

$$\mathsf{canonize}'\,(\Phi) \triangleq \mathsf{reduce}\,(\mathsf{canonize}\,(\Phi))$$

These two operators may lose a lot of information; ideally, they would just keep the values that the non-predicated analysis fails to compute.

*Example 4.* Let us come back to the example of Fig. 5, improved in Fig. 7. When joining the values coming from lines 5 and 7, the $\mathsf{reduce}$ operator removes under the guards $c$ and $\neg c$ the information about $v$, which is already present under the weaker guard `true`. In parallel, after line 9, the modified $\mathsf{lift}$ operator does not apply the full interval analysis to the values guarded by $c$ and $\neg c$. Instead, we use a simpler abstraction, that only removes information about variables that are overwritten. This way, the information about $w$ is no longer duplicated.

### 4.2 Convergence of the Analysis and Practical Complexity

Throughout the analysis of a given program, all guards of non trivial implications present in a map are derived from the conditionals of the program, so their number remains finite. In practice, this number can be high; we discuss a possible way of limiting it in Sect. 6. The predicated analysis essentially amounts to performing the underlying analysis over the values under each guard (except for the $\mathsf{refine}$ operations, which allow us to be more precise). Thus, if the underlying domain provides (or requires) a widening operator to effectively compute the fixpoint, then it can (and should) be lifted as well. Finally, if the underlying transfer functions are monotonic, so are the predicated ones, which ensures the termination of our analysis.

Some operators of the abstract semantics may seem costly to compute, but efficient implementations or simpler operators can mitigate this. For instance, at a junction point of the control-flow graph, $\Phi_1 \sqcup_{\texttt{pred}} \Phi_2$ creates fresh implications through the operators $l_1$, $l_2$ and $l_\cup$ (Fig. 4). Both $l_1$ and $l_2$ only traverse one map once. On the other hand, $l_\cup$ requires $|\Phi_1| \times |\Phi_2|$ operations, where $|\Phi|$ is the number of non trivial implications in $\Phi$. However, any implication $\langle p \to l \rangle$ that held before the control-flow split (and that has not been invalidated since) still exists in $\Phi_1$ and $\Phi_2$, and will exist in the join. Then, any implication of the form $\langle p \wedge p' \to l \sqcup_{\mathcal{L}} l' \rangle$ is redundant with $\langle p \to l \rangle$ and does not need to be considered. An optimized implementation should thus consider only the subparts of the maps that are distinct. In order to further speed up the analysis, we can also use a more approximate join, that keeps only implications $\langle p \to l_1 \sqcup_{\mathcal{L}} l_2 \rangle$ such that $\langle p \to l_1 \rangle \in \Phi_1$ and $\langle p \to l_2 \rangle \in \Phi_2$.

The $\mathsf{refine}$ and $\mathsf{reduce}$ operators alter the values guarded in the implications, w.r.t. the context (for $\mathsf{refine}$) and weaker guards (for $\mathsf{reduce}$). Nevertheless, the value under the $\texttt{true}$ guard is quite special, as it is the broadest one. We can define easier to compute versions of these operators, at the expense of precision. They only refine the value under $\texttt{true}$, and reduce other values accordingly:

$$\mathsf{reduce}\,(I) \triangleq \lambda p.\, I\,(p) \setminus_{\mathcal{L}} I\,(\texttt{true})$$

$$\mathsf{refine}\,(h,(c,I)) \triangleq \mathsf{canonize}'\left(c \wedge h, I\left[\texttt{true} \mapsto \bigsqcap\nolimits_{\mathcal{L}} \{I\,(p) \mid h \Rightarrow p\}\right]\right)$$

## 5 Applications

This section describes the two abstract domains on which we have instanciated a predicated analysis in the Frama-C platform. Note that our framework could also be applied to other domains, e.g. intervals or the "valid file descriptors" domain used for Fig. 1.

### 5.1 A First Abstract Domain: Initialized Variables

A first simple domain retains at each progam point the set of variables that were properly initialized. Our experiments on a generated C code, where initialization of variables happens far before their uses, showed that this domain is very useful. In the abstract semantics of this domain, we introduce a new default value $\varnothing$ in $\mathbb{V}$, to which all variables are equal at program entry (i.e. $\mathbb{S}\,(0) \triangleq \{\lambda x.\varnothing\}$).

$$\gamma_{\mathsf{init}}\,(V) = \{\rho \mid \forall x \in V, \rho\,(x) \neq \varnothing\}$$

$$[\![x := e]\!]^\sharp_{\mathsf{init}}\,(V) = \begin{cases} V \cup \{x\} & \text{if } \mathsf{deps}\,(e) \subseteq V \\ V \setminus \{x\} & \text{otherwise} \end{cases}$$

$$[\![c \triangleleft]\!]^\sharp_{\mathsf{init}}\,(V) = V$$

$$\mathsf{deps}\,(x := e) \triangleq \mathsf{deps}\,(e)$$

$$\mathsf{deps}\,(c \triangleleft) \triangleq \mathsf{deps}\,(c)$$

The execution of a statement is correct when all the involved variables are initialized. We extend $\mathsf{deps}$ to instuctions: $\mathsf{deps}\,(\texttt{i})$ denotes the set of variables the statement $\texttt{i}$ depends on. Then, a program $P$ is correct according to this initialized semantics when $\forall\,(n,\texttt{i},m) \in P, \mathsf{deps}\,(\texttt{i}) \subseteq \mathbb{S}^\sharp_{\mathsf{init}}\,(n)$.

$$[\![c \lhd, p]\!]^\sharp_{\mathsf{eq}\times\mathbb{C}}(E) \triangleq \begin{cases} E \cup \{e_1 = e_2\} & \text{if } p \equiv \texttt{true} \text{ and } c = (e_1 = e_2) \\ E & \text{otherwise} \end{cases}$$

$$[\![x := e, p]\!]^\sharp_{\mathsf{eq}\times\mathbb{C}}(E) \triangleq \begin{cases} \mathsf{kill}_{\mathsf{eq}}(x, E) \cup \{x = e\} & \text{if } p \equiv \texttt{true} \text{ and } x \notin \mathsf{deps}(e) \\ \mathsf{kill}_{\mathsf{eq}}(x, E) & \text{otherwise} \end{cases}$$

$$\mathsf{kill}_{\mathsf{eq}}(v, E) \triangleq \{(a = b) \in E \mid v \notin \mathsf{deps}(a) \wedge v \notin \mathsf{deps}(b)\}$$

$$E \setminus_{\mathsf{eq}} F \triangleq \{(a = b) \in E \mid (a = b) \notin F\}$$

$$\gamma_{\mathsf{eq}}(E) \triangleq \{\rho \mid (a = b) \in E \Rightarrow [\![a]\!]_\rho = [\![b]\!]_\rho\}$$

**Fig. 8.** Abstract semantics for the equality domain

### 5.2 A Second Abstract Domain: Equalities

Our experiments also relied on a symbolic domain tracking equalities between C expressions. It aims at enhancing the precision of Frama-C's existing value analysis plugin, whose abstract domains are non-relational. Our intents are also somewhat similar to those of Miné [12], in particular abstracting over temporary variables resulting from code normalization. Our equality domain boils down to retaining equalities stemming from assignments or equality conditions. Its formal definition is presented in Fig. 8, where the set $E$ of equalities increases on equality assume statements, and on assignments that do not refer to the variable being modified. To be sound, the transfer function on assignments must also remove equalities that involve the overwritten variable, through the $\mathsf{kill}_{\mathsf{eq}}$ operator. Following Sect. 4.1, we present simplified transfer functions, for which only the `true` guard is enriched, and in which the operator $\setminus_{\mathsf{eq}}$ can be used to remove redundant equalities.

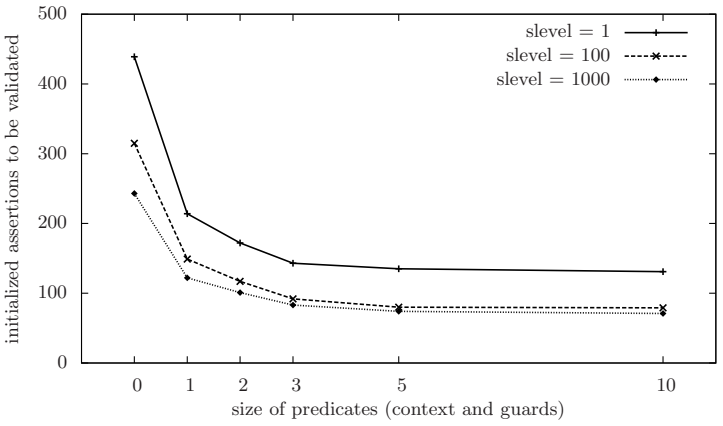
This domain lends itself to a natural extension of our analysis, namely the strengthening of the context by backward-propagating information from $\mathcal{L}$ when modifying the context. For example, the equalities can be used to quotient the context by equal expressions. Furthermore, during the weakening of a context – when the truth value of one of its litterals is modified – we may substitute the litteral by an expression equal to it, instead of resetting the context.

Moreover, we were faced with code patterns similar to the one presented in Fig. 9, in which the condition of a branch is defined within a previous one – resulting in an implicit dependency between the two conditions `h` and `p`. To handle this pattern, we extend $\mathsf{refine}$ so that, when computing $\mathsf{refine}(h, (c, I))$ with $I$ containing $\langle p \to (h = 0)\rangle$, then we also add $\neg p$ to the new context.

```
if(p) {
    ...
    h = e;
} else {
    h = 0;
}
if (h)
    ...
```

**Fig. 9.** Code pattern

Thus, imprecision in the refinement of contexts can be reduced by crossing information between underlying and predicated domains.




| slevel | assertions to be validated | initialization assertions | validated assertions/clevel | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 5 | 10 | |
| 1 | 632 | 439 | 225 | 267 | 296 | 305 | 308 | 6.4s |
| 10 | 600 | 409 | 199 | 241 | 270 | 279 | 282 | 9.8s |
| 100 | 504 | 315 | 166 | 198 | 223 | 235 | 236 | 38s |
| 1000 | 430 | 243 | 121 | 142 | 160 | 169 | 172 | 502s |
| | | | 6s | 9s | 15s | 24s | 116s | time |

**Fig. 10.** Experimental results

## 6 Experimental Results

We have integrated our predicated analyses framework as a new plugin of the Frama-C platform. This plugin runs above the value analysis plugin (abbreviated as VA), which we mainly use to get aliasing information on pointers. This information is needed to ensure the soundness of the deps operator.

*Perimeter of our Analysis.* At each program point where it cannot guarantee the absence of run-time error, VA emits as an alarm an ACSL assertion that excludes the failure case. These alarms may correspond to real bugs, if the statement can give rise to an error at execution time, or may be due to a lack of precision. To limit imprecisions caused by junctions in the control-flow graph, VA implements an instance of trace partitionning, and propagates separately multiple abstract states coming from different branches. As dissociating every feasible execution path leads to untractable analyses, the number of parallel states maintained by VA is limited but configurable by the slevel parameter. Still, high slevel values may lead to high analysis time.

Using a predicated analysis over a simple domain to prove some of the ACSL assertions emitted by VA can avoid this blow up. By construction, we mainly

improve VA's results on successive assume statements with identical conditions[1]. Although such pattern is relatively unusual in idiomatic C code, it is much more frequent in generated programs, for which our method is well adapted.

Some generated programs can include a very large number of nested conditional branchs and loops, leading to overly wide contexts in our own analysis. To avoid a complexity explosion, we limit the number of litterals in the predicates used in contexts and guards (thereby decreasing the precision of our results), according to a parameter clevel. Conversely, our prototype implements a precise version of the abstract semantics operators presented in Sect. 3, without the relaxations proposed in Sect. 4.2.

*Results.* We tested our plugin on a C program of 5000 lines generated by the industrial environment SCADE, devoted to real-time software. As often with such codes, multiple conditionals are heavily used – typically to test automata states or clocks. Our results are presented in Fig. 10. We first applied VA, which emitted various assertions to further validate (column 2). As expected, a higher slevel results in fewer alarms. Between 55% and 70% of those are assertions requiring variables to be properly initialized (column 3), which are those our underlying domain understands. We then ran our predicated analysis, instantiated by the domain presented in Sect. 5.1, with different limits for the size of predicates (columns "validated assertions"). The last column indicates the analysis time of VA, while that of the predicated analysis is given in the last line.

While VA produces significantly less alarms with a higher slevel, its analysis time also increases drastically. This is unsurprising, as fully partitioning for $k$ successive conditionals may require as much as $2^k$ distinct states. On the other hand, our plugin is effective to quickly validate numerous assertions left unproven by VA, even with strongly limited predicates. The precision of our analysis increases rapidly with the clevel parameter, while the analysis times remains reasonable. More generally, it turns out that small contexts are sufficient to retain most of the relevant information: less assertions remain to be validated with $\mathsf{clevel} = 1$ and $\mathsf{slevel} = 1$ than with $\mathsf{clevel} = 0$ and $\mathsf{slevel} = 1000$. Intuitively, even inside deeply nested conditionals (which generate complex contexts), only the more recent guards are useful. In general, our results show that it is much more cost efficient to increase the clevel parameter than the slevel parameter.

## 7 Conclusion

This work provides a generic framework to enhance the precision of standard dataflow analyses. This framework constructs a derived predicated analysis able to mitigate information loss at junction points of the control-flow graph, by retaining the conditional values about each branch. Our analysis strives to minimize redundant information processing due to these disjunctions. Experimental tests led through the static analysis platform Frama-C on generated C code showed that a predicated analysis over simple domains can significanlty improve the results of prior analyses.

[1] Modulo conjunction, disjunction and negation, but only over uninterpreted expressions; in particular, $x < y$ and $y > x$ are not considered as being equivalent guards.

The litterals of our predicates are expressions that we currently consider as opaque. In order to improve our analysis, we intend to give some meaning to the operators in these expressions and to extend the logical implication between guards accordingly. In particular, we will handle successive conditions on distinct but related expressions, such as $(x \geq 0)\,\triangleleft$ and $(x \geq 2)\,\triangleleft$. Moreover, prior syntactic analyses or heuristics could help to select relevant predicates for the contexts, which would no longer be extended at each assume statement. This would avoid maintaining implication guards that will never be useful again later in the program. Finally, it would be worthwhile to apply our predicated analysis over more complex abstract domains.

## References


1. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, Version 1.8 (2014), `http://frama-c.com/download/acsl-implementation-Neon-20140301.pdf`
2. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Pottosin, I.V., Bjorner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
3. Chebaro, O., Cuoq, P., Kosmatov, N., Marre, B., Pacalet, A., Williams, N., Yakobowski, B.: Behind the scenes in sante: a combination of static and dynamic analyses. Autom. Softw. Eng. 21(1), 107–143 (2014)
4. Correnson, L., Signoles, J.: Combining analyses for C program verification. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 108–130. Springer, Heidelberg (2012)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)
6. Cousot, P., Cousot, R.: Abstract interpretation frameworks. J. Log. Comput. 2(4), 511–547 (1992)
7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 272–300. Springer, Heidelberg (2007)
8. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - A software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
9. Cuoq, P., Prevosto, V., Yakobowski, B.: Frama-C's value analysis plug-in, `http://frama-c.com/download/value-analysis-Neon-20140301.pdf`
10. Fischer, J., Jhala, R., Majumdar, R.: Joining dataflow with predicates. In: Wermelinger, M., Gall, H. (eds.) ESEC/SIGSOFT FSE, pp. 227–236. ACM (2005)
11. Graf, S., Saïdi, H.: Verifying invariants using theorem proving. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 196–207. Springer, Heidelberg (1996)
12. Miné, A.: Symbolic methods to enhance the precision of numerical abstract domains. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 348–363. Springer, Heidelberg (2006)
13. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (2005)
14. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. 29(5) (2007)