A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework

SE-WON KIM, S-Core
XAVIER RIVAL, CNRS, ENS, INRIA Paris-Rocquencourt, PSL* University
SUKYOUNG RYU, KAIST

Program analyses often utilize various forms of *sensitivity* such as context sensitivity, call-site sensitivity, and object sensitivity. These techniques all allow for more precise program analyses, that are able to compute more precise program invariants, and to verify stronger properties. Despite the fact that sensitivity techniques are now part of the standard toolkit of static analyses designers and implementers, no comprehensive frameworks allow the description of all common forms of sensitivity. As a consequence, the soundness proofs of static analysis tools involving sensitivity often rely on ad hoc formalization, which are not always carried out in an abstract interpretation framework. Moreover, this also means that opportunities to identify similarities between analysis techniques to better improve abstractions or to tune static analysis tools can easily be missed.

In this article, we present and formalize a framework for the description of *sensitivity in static analysis*. Our framework is based on a powerful abstract domain construction, and utilizes reduced cardinal power to tie basic abstract predicates to the properties analyses are sensitive to. We formalize this abstraction, and the main abstract operations that are needed to turn it into a generic abstract domain construction. We demonstrate that our approach can allow for a more precise description of program states, and that it can also describe a large set of sensitivity techniques, both when sensitivity criteria are static (known before the analysis) or dynamic (inferred as part of the analysis), and sensitive analysis tuning parameters. Last, we show that sensitivity techniques used in state-of-the-art static analysis tools can be described in our framework.

CCS Concepts: • **Software and its engineering → Automated static analysis**;

Additional Key Words and Phrases: Abstract interpretation, program analysis, analysis sensitivity, analysis framework

13

ACM Reference format:
Se-Won Kim, Xavier Rival, and Sukyoung Ryu. 2018. A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework. *ACM Trans. Program. Lang. Syst.* 40, 3, Article 13 (August 2018), 44 pages.
https://doi.org/10.1145/3230624

This work was done while Se-won Kim was affiliated at KAIST.
This work has received funding from National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177) and the European Research Council under the EU's seventh framework programme (FP7/2007-2013), grant agreement 278673, Project MemCAD, and from the ARTEMIS Joint Undertaking no. 269335 (see Article II.9 of the JU Grant Agreement).
Authors' addresses: S.-W. Kim, Hwangsaeul-ro 258, Bundang-gu, Seongnam-si, Gyeonggi-do; email: kim.sewon@gmail.com; X. Rival, DI - Ecole Normale Supérieure, 45, rue d'Ulm, 75230 Paris Cedex 05, France; email: Xavier.Rival@ens.fr; S. Ryu, KAIST, 291 Daehak-ro, Yuseong-gu, Daejeon 34141, Republic of Korea; email: sryu.cs@kaist.ac.kr.

https://doi.org/10.1145/3230624

1 INTRODUCTION

In this article, we consider program static analyses that aim at computing nontrivial semantic properties of programs, so as to establish safety properties such as absence of certain classes of errors, or partial correctness. The results of static analyses should be *sound*, that is, they should hold for *all program executions*. Furthermore, the analyses should be *automatic*. Since the properties of interest are by essence not computable, sound and automatic static analyses are necessarily conservative: they may fail to compute the most precise semantic properties, and they may fail to establish semantic properties that hold true. Of course, static analysis designers strive to limit such failures, and pay great attention to the *precision* of their analysis algorithms: a precise static analysis should fail to establish the property of interest as rarely as possible. Program static analyses often explicitly rely on the notion of abstraction (Cousot and Cousot 1977), where each abstract property describes a set of concrete states. The goal of a precise static analysis is then to compute a tight abstraction of the reachable concrete states.

Sensitivity is one of the most commonly encountered techniques to improve the precision of static analysis tools. It consists in considering invariants where distinct (but not necessarily disjoint) sets of program behaviors are described separately. More precisely, sensitivity relies on a splitting of the state space into chunks that will always be discriminated by the analysis. For instance, *call-site string sensitivity* (Sharir and Pnueli 1981) was introduced to discriminate states according to an abstraction of their calling contexts defined by their call strings. The advantage of this approach is that the analysis does not have to produce a precise abstraction consisting of a single abstract element, for all the states that reach a given procedure f; instead, it may simply produce several abstractions for subsets that reach f, and that correspond to sets of contexts. In essence, this amounts to using *symbolic disjunctions* of abstract properties during the course of the analysis, hereby improving the precision of abstract operations such as control flow join. On the other hand, such disjunctions may incur a higher cost, both in terms of space and of analysis runtime.

We can observe that call-site string sensitivity is a form of sensitivity among many, and that most static analysis tools use one or more forms of it. For instance, *flow sensitivity* distinguishes states based on the control point at which they can be observed. It has been formalized by Cousot (Cousot 1981) as an abstract interpretation (Cousot and Cousot 1977, 1979). The notion of call-site string sensitivity (Sharir and Pnueli 1981) was also applied to functional languages (Shivers 1991). *Trace partitioning* abstractions (Handjieva and Tzolovski 1998; Rival and Mauborgne 2007) rely on control flow paths (such as loop iteration counts or condition branches) or on properties observed at a certain point of program executions (such as the value of a variable at a specific program point) in order to distinguish states, and improve precision, by effectively delaying control flow joins that would otherwise deteriorate precision. The trace partitioning framework has been applied to significantly reduce the number of false alarms produced when analyzing safety-critical embedded programs (Rival and Mauborgne 2007). Moreover, other forms of sensitivities have been proposed to tie abstract properties to (an abstraction of) the value of some variables. This includes *object sensitivity* (Milanova et al. 2005; Smaragdakis et al. 2011), the *Cartesian product algorithm* (Agesen 1995), and the *functional approach* (Sharir and Pnueli 1981). The Reps-Horwitz-Sagiv (RHS) algorithm (Reps et al. 1995) is a special case of the functional approach where we can use an efficient graph reachability algorithm.

We can observe that a salient common feature of all these forms of sensitivity is that they introduce disjunctions into abstract states in a very structured way: each disjunct corresponds to a set of executions characterized by a specific property, which we call a *view*, and attaches some abstract property to it. In this view, each piece of the disjunction actually reads as a logical implication, which states that, whenever a concrete execution is an element of this view, then it also meets the

attached abstract property. Therefore, we can see an abstract value as a *conjunction of implications* rather than an unstructured disjunction. As an example, call string sensitivity describes sets of states by tying each state to an abstraction of its call stack. This construction was initially formalized by Cousot and Cousot as the *reduced cardinal power* (Cousot and Cousot 1979), and it was further developed and generalized by Giacobazzi, Ranzato and Scozzari in Giacobazzi and Scozzari (1998) and Giacobazzi et al. (2005).

As we noted, sensitivity aims at improving precision of static analysis, but may degrade its performance significantly, as abstract states become more complex and thus, harder to manipulate. To avoid performance issues, static analysis designers need to pay very careful attention to the amount of sensitivity they build into the analyses they design. A common pattern is to start with a basic flow-sensitive analysis, and to add in additional forms of sensitivities to improve precision, while making sure that (1) soundness of the analysis is preserved, (2) an adequate level of precision is reached, and (3) performance is not impacted to the point where the analysis would not be scalable anymore. Unfortunately, there exists, as of now, no general framework to guide this process, even though common forms of sensitivity obviously share some characteristics. Worse still, the forms of sensitivity mentioned above are formalized in different frameworks, which does not help their combination, although it is often necessary to integrate several forms of sensitivity in a single static analysis tool. Therefore, the task of designing and implementing static analyses relying on sensitivity remains tedious.

In this article, we present a *unifying framework for sensitive abstractions* in program analysis. Essentially, a *sensitive abstraction* is an abstraction which is able to bind different pieces of abstract information to different subsets of program behaviors. In logical terms, this binding boils down to a conjunction of implications. Our study is carried out in the abstract interpretation framework (Cousot and Cousot 1977, 1979), and provides a formal foundation to design, combine, and engineer sensitivities. We make the following contributions:

- —We set up a general framework for sensitivity in static analysis by abstract interpretation, which relies on a powerful abstract domain combination operation, based on the reduced cardinal power abstract domain construction (Cousot and Cousot 1979). The power of our framework largely comes from the way it represents abstract properties as conjunctions of implications rather than unstructured disjunctions. We provide generic abstract operations for sensitive abstractions, including the post-condition operations, the widening operator used to enforce convergence of abstract iterates.
- —We formalize the most common forms of sensitivity as instances of our framework, which demonstrates its generality. Moreover, we demonstrate that our framework allows one to reason on the combination of existing forms of sensitivity and can be applied to the creation of new forms of sensitivity. Therefore, our framework provides a powerful setup to reason about sensitivity in static analysis.
- —We apply our framework to the description of the forms of sensitivity used in real-world static analyzers, Astrée (Blanchet et al. 2003), Sparrow (Oh et al. 2009), SAFE (Lee et al. 2012b), and TAJS (Møller et al. 2014). This study demonstrates that our framework is effective in practice.

Our framework generalizes the trace partitioning framework (Rival and Mauborgne 2007) since trace partitions can be viewed as specific instances of views. It also allows one to describe forms of sensitivities that would better be described at other levels than traces. An important motivation for such a framework is to better identify which sensitivity is required to cope with a given static analysis problem, at a lower cost (i.e., with fewer, simpler views), and with a sufficient level of precision.

This article is organized as follows. In Section 2, we informally study the form of the predicates expressed in sensitive analyses. Section 3 presents the analysis domains and the Galois connection

```
          int readpos()                    function returning random, positive values

          bool b_0, b_1, ..., b_{N-1};     boolean variables storing random values
          int x_0, x_1, ..., x_{N-1}, y;   uninitialized integer variables
ℓ_0 :     if(b_0){
              x_0 = readpos();             reads a strictly positive value into x_0
          }else{
              x_0 = 0;                     zeroes out x_0
          }
ℓ_1 :     if(b_1){
              x_1 = readpos();             reads a strictly positive value into x_1
          }else{
              x_1 = 0;                     zeroes out x_1
          }
          ...
ℓ_{N-1} : if(b_{N-1}){
              x_{N-1} = readpos();         reads a strictly positive value into x_{N-1}
          }else{
              x_{N-1} = 0;                 zeroes out x_{N-1}
          }
ℓ_y :     y = x_0 * x_1 * ... * x_{N-1};   combination
ℓ_exit :  ...
```

Fig. 1. Pseudo-code of a simple program.

between them for sensitive analyses, and discusses their basic properties. Section 4 defines a precise and sound abstract execution on the analysis domains defined in Section 3. Section 5 describes how our framework can safely adjust sensitivity dynamically. In Section 6, we present techniques to construct sensitive analyses, including basic instances of sensitivities and operations to compose several sensitivities together. Section 7 describes four powerful static analysis tools, Astrée, Sparrow, SAFE, and TAJS, as instances of our framework and discusses related works and we conclude in Section 8.

2 OVERVIEW

Throughout the article, we consider programs that can be represented as control flow graphs. In the following, we discuss a simple example of a program, and show how reasoning about it can take advantage of some simple form of sensitivity. While the program is simplified to better show the impact of sensitivity on the analysis, it does derive from existing programming patterns that can be encountered in error handling routines, or in user interaction code. For example, a function in the generic keyboard shortcut handler provided by the Google Closure library[1] computes the value of `strokes` by using the values of the variables `modifiers` and `keyCode`, which are set depending on whether the value of `key` is one of the modifier keys `shift`, `ctrl`, `alt`, and `meta`.

This program shown in Figure 1 declares N Boolean variables, which may read arbitrary values (e.g., in the user actions or in previous computations) and N integer variables known to store positive values. For the sake of simplicity, we also assume integer arithmetic operators perform ideal mathematical operations (instead of modular arithmetic). The program computes an integer value $\mathtt{y}$ from the random Boolean inputs $\mathtt{b}_0, \ldots, \mathtt{b}_{N-1}$, by first computing a series of integer variables $\mathtt{x}_0, \ldots, \mathtt{x}_{N-1}$ (either as a strictly positive value returned by function `readpos`, or as 0) and

[1] https://closure-library.googlecode.com/git-history/docs/local_closure_goog_ui_keyboardshortcuthandler.js.source.html.

combines them. One can consider y in this program as `strokes` in the above-mentioned function, the Boolean inputs as whether `key`'s value is one of the modifier keys, and the integer variables as `modifiers` and `keyCode`.

Obviously, in the concrete executions, the final value of y depends on all the control flow branches that occur during the execution of the program. In fact, y is strictly positive if and only if all Booleans are true and it is 0 if and only if any of the Booleans is false. Thus, the following two properties hold at the end of the execution of the program:

$$\begin{aligned} &(\mathcal{P}_0)\ \ (\forall k \in \{0,\ldots,N-1\},\ \mathsf{b}_k = \texttt{TRUE}) \Longrightarrow \mathsf{y} \geq 1,\\ &(\mathcal{P}_1)\ \ (\exists k \in \{0,\ldots,N-1\},\ \mathsf{b}_k = \texttt{FALSE}) \Longrightarrow \mathsf{y} = 0. \end{aligned} \tag{1}$$

Such properties can be inferred or verified in a number of ways. A common technique consists in performing a forward analysis, starting from a pre-condition, and trying to establish the aforementioned properties as post-condition. In this case, this analysis should start from the trivial pre-condition that holds on any state (all variables initially hold random values). Since we focus on the properties the analyses can establish and do not consider their representation and algorithms to compute them, we describe properties that can be obtained using logical formulas. For presentation brevity, we assume that the analysis uses a *non-relational* abstraction, which means a set of states will typically be described by a conjunction of constraints where each variable is described in a separate element of the conjunction. Therefore, base domain abstract properties are expected to be of the form $P_{\mathsf{b}_0} \wedge \ldots \wedge \ldots P_{\mathsf{b}_{N-1}} \wedge P_{\mathsf{x}_0} \wedge \ldots \wedge P_{\mathsf{x}_{N-1}} \wedge P_{\mathsf{y}}$ where P_{b_i} and P_{x_j} denote constraints on b_i and x_j, respectively.

Non-Sensitive Analysis. As a first attempt, we consider a fully non-sensitive analysis. Then, it will compute as a post-condition a single conjunction of properties of the variables. Since this abstract post-condition should account for states where b_i is true and for states where it is false, it cannot capture any precise information about b_i. The same holds for x_i. Therefore, it will also account for cases where y is strictly positive as well as for cases where it is 0. Therefore, a non-sensitive analysis will compute at best a property of the form

$$(\mathsf{b}_0 \in \{\textbf{true}, \textbf{false}\}) \wedge \ldots \wedge (\mathsf{b}_{N-1} \in \{\textbf{true}, \textbf{false}\}) \wedge (\mathsf{x}_0 \geq 0) \wedge \ldots \wedge (\mathsf{x}_{N-1} \geq 0) \wedge (\mathsf{y} \geq 0).$$

Therefore, a non-relational and non-sensitive analysis will prove neither $\mathcal{P}_0$ nor $\mathcal{P}_1$.

Fully Sensitive Analysis. In order to alleviate the limitations of the previous non-sensitive analysis, the most obvious solution consists in letting it manipulate and produce *disjunctions* of abstract states of the previous form. A *fully disjunctive* analysis can introduce as many disjuncts as it needs in order to achieve a precise result (this approach raises termination issues that we do not consider in this article). The fully disjunctive analysis of the example of Figure 1 would produce the following disjunction of states:

$$\begin{array}{ll} & (\mathsf{b}_0 = \textbf{true} \wedge \ldots \wedge \mathsf{b}_{N-2} = \textbf{true} \wedge \mathsf{b}_{N-1} = \textbf{true} \\ & \quad \wedge \mathsf{x}_0 \geq 1 \wedge \ldots \wedge \mathsf{x}_{N-2} \geq 1 \wedge \mathsf{x}_{N-1} \geq 1 \wedge \mathsf{y} \geq 1) \\ \vee & (\mathsf{b}_0 = \textbf{true} \wedge \ldots \wedge \mathsf{b}_{N-2} = \textbf{true} \wedge \mathsf{b}_{N-1} = \textbf{false} \\ & \quad \wedge \mathsf{x}_0 \geq 1 \wedge \ldots \wedge \mathsf{x}_{N-2} \geq 1 \wedge \mathsf{x}_{N-1} = 0 \wedge \mathsf{y} = 0) \\ \vdots & \qquad\qquad\qquad \vdots \\ \vee & (\mathsf{b}_0 = \textbf{false} \wedge \ldots \wedge \mathsf{b}_{N-2} = \textbf{false} \wedge \mathsf{b}_{N-1} = \textbf{false} \\ & \quad \wedge \mathsf{x}_0 = 0 \wedge \ldots \wedge \mathsf{x}_{N-2} = 0 \wedge \mathsf{x}_{N-1} = 0 \wedge \mathsf{y} = 0). \end{array}$$

However, this *disjunction* does not convey much structure, since it does not allow one to clearly see where each disjunct comes from. In fact, there are at least two ways to relate each disjunct to a set of behaviors of the program, which will be observed during the analysis:

— each disjunct is characterized by an initial mapping for the Boolean variables;
— equivalently, each disjunct corresponds to a unique control flow path in the program, which is determined by the **true** and **false** branches that are traversed.

Both of these approaches actually suggest very similar formulas, based on a *conjunction of implications*, where each implication characterizes a given configuration of the Boolean variables or a given control flow path. These correspond exactly to *value-sensitive* and *path-sensitive* analyses. The formula corresponding to the value-sensitive analysis boils down to

$$\begin{array}{ll} & (\mathsf{b}_0 = \textbf{true} \wedge \ldots \wedge \mathsf{b}_{N-2} = \textbf{true} \wedge \mathsf{b}_{N-1} = \textbf{true}) \\ & \quad \Longrightarrow (\mathsf{x}_0 \geq 1 \wedge \ldots \wedge \mathsf{x}_{N-2} \geq 1 \wedge \mathsf{x}_{N-1} \geq 1 \wedge \mathsf{y} \geq 1) \\ \wedge & (\mathsf{b}_0 = \textbf{true} \wedge \ldots \wedge \mathsf{b}_{N-2} = \textbf{true} \wedge \mathsf{b}_{N-1} = \textbf{false}) \\ & \quad \Longrightarrow (\mathsf{x}_0 \geq 1 \wedge \ldots \wedge \mathsf{x}_{N-2} \geq 1 \wedge \mathsf{x}_{N-1} = 0 \wedge \mathsf{y} = 0) \\ \vdots & \qquad\qquad\qquad \vdots \\ \wedge & (\mathsf{b}_0 = \textbf{false} \wedge \ldots \wedge \mathsf{b}_{N-2} = \textbf{false} \wedge \mathsf{b}_{N-1} = \textbf{false}) \\ & \quad \Longrightarrow (\mathsf{x}_0 = 0 \wedge \ldots \wedge \mathsf{x}_{N-2} = 0 \wedge \mathsf{x}_{N-1} = 0 \wedge \mathsf{y} = 0). \end{array} \tag{2}$$

Path sensitivity would result in a similar looking formula (essentially, the left-hand side of each implication would turn into a conjunction of branch choices of the form $l_i : b$), thus we focus on value sensitivity in this section. Obviously, this formula describes in a very precise manner the set of states that can be observed at the end of the execution of the code fragment, and it allows one to verify both property $\mathcal{P}_0$ and property $\mathcal{P}_1$. However, we also note the very high cost of this approach, since this analysis computes a conjunction of 2^N formulas. Therefore, we cannot expect fully disjunctive, fully value-sensitive or fully path-sensitive analyses to scale.

More Compact Choices of a Set of Cases. The main issue from the fully sensitive analyses discussed in the previous paragraph stems from the fact that they generate too many case splits which are not all absolutely necessary to establish the properties of interest. In particular, whenever any of the Boolean variables contains **false**, the resulting value for y is necessarily 0. Therefore, we can use a smaller set of implications to achieve the same result, even if we stick with conjunctions of variable predicates in the left-hand side of the implications:

$$\begin{array}{lccc} & (\mathsf{b}_0 = \textbf{true} \wedge \ldots \wedge \mathsf{b}_{N-1} = \textbf{true}) & \Longrightarrow & (\mathsf{x}_0 \geq 1 \wedge \ldots \wedge \mathsf{x}_{N-1} \geq 1 \wedge \mathsf{y} \geq 1) \\ \wedge & \mathsf{b}_0 = \textbf{false} & \Longrightarrow & (\mathsf{x}_0 = 0 \wedge \mathsf{y} = 0) \\ \vdots & \vdots & \vdots & \vdots \\ \wedge & \mathsf{b}_{N-1} = \textbf{false} & \Longrightarrow & (\mathsf{x}_{N-1} = 0 \wedge \mathsf{y} = 0). \end{array} \tag{3}$$

This case split also corresponds to a value-sensitive analysis although it does not distinguish all vectors of Boolean values, thus we can call this approach a form of *partially value-sensitive analysis*. It also allows one to verify both property $\mathcal{P}_0$ and property $\mathcal{P}_1$ though the number of cases it uses is linear $(N + 1)$ instead exponential (2^N) with the fully sensitive analysis.

However, one may argue that the choice of case splits that was used for this analysis is somewhat arbitrary, since it gives a very special role to a specific vector of inputs. While this choice is very suitable for some program/properties, it is much less so for others. For instance, the following linear case split is based on a similar structure, and only permutes **true** and **false** in the left-hand side formulas, yet it would fail to verify both property $\mathcal{P}_0$ and property $\mathcal{P}_1$:

$$\begin{array}{lcl} \mathsf{b}_0 = \textbf{true} & \Longrightarrow & (\mathsf{x}_0 > 0 \wedge \mathsf{y} \geq 0) \\ \vdots \quad\quad \vdots & \vdots & \vdots \quad\quad\quad \vdots \\ \wedge \quad \mathsf{b}_{N-1} = \textbf{true} & \Longrightarrow & (\mathsf{x}_{N-1} > 0 \wedge \mathsf{y} \geq 0) \\ \wedge\ (\mathsf{b}_0 = \textbf{false} \wedge \ldots \wedge \mathsf{b}_{N-1} = \textbf{false}) & \Longrightarrow & (\mathsf{x}_0 = 0 \wedge \ldots \wedge \mathsf{x}_{N-1} = 0 \wedge \mathsf{y} = 0). \end{array} \tag{4}$$

Another Linear Case Split. The previous paragraphs have shown a very powerful but expensive analysis as well as a much less costly yet very powerful one, though the latter requires an ad hoc choice of partitions. However, we can observe that there exists a very natural way of preserving information about each condition as a conjunction of implication, that is also quite precise. The fundamental idea is to use another conjunction of implications, where the left-hand side of each implication accounts for the executions that take a specific (true or false) branch of a specific conditional statement. The resulting condition characterizing the state at point ℓ_y writes as follows:

$$\begin{array}{lcl} \mathsf{b}_0 = \textbf{true} & \Longrightarrow & \mathsf{x}_0 > 1 \\ \wedge \quad \mathsf{b}_0 = \textbf{false} & \Longrightarrow & \mathsf{x}_0 = 0 \\ \vdots \quad\quad \vdots & \vdots & \vdots \quad\quad \vdots \\ \wedge\ \mathsf{b}_{N-1} = \textbf{true} & \Longrightarrow & \mathsf{x}_{N-1} > 1 \\ \wedge\ \mathsf{b}_{N-1} = \textbf{false} & \Longrightarrow & \mathsf{x}_{N-1} = 0. \end{array} \tag{5}$$

This post-condition results from an analysis that maintains value sensitivity over all conditions, but that does not perform full sensitivity during the analysis, since it does not attempt to precisely characterize the results obtained from all combinations of inputs. Analyzing the final assignment under this pre-condition clearly allows one to prove $\mathcal{P}_1$: indeed, if $\mathsf{b}_i = \textbf{false}$, then it implies that $\mathsf{x}_i = 0$, thus $\mathsf{y} = 0$. More interestingly, it also allows one to prove $\mathcal{P}_0$. Indeed, if $\forall i,\ \mathsf{b}_i = \textbf{true}$, then the above formula entails $\mathsf{x}_0 \geq 1 \wedge \ldots \wedge \mathsf{x}_{N-1} \geq 1$, therefore $\mathsf{y} \geq 1$ after the assignment. Such a logical step is possible since the cases in the above conjunction of implication *are not mutually exclusive*, thus the analysis of the final assignment may combine them. Thus, those two properties get verified in quite different ways: in the case of $\mathcal{P}_1$ only one implication needs to be used, whereas in the case of $\mathcal{P}_0$, several (N) implications are combined together.

Moreover, this conjunction of implications has linear size since it is made of $2N$ implications, and it is also less specific than the cases encountered in the previous paragraph, in Equation (3) and in Equation (4). The property can still be proved, but at the cost of a more careful combination of cases in the case of $\mathcal{P}_0$.

Sensitive Analyses. In this section, we have observed that several instances of value- and path-sensitive analyses can be described using logical predicates represented by conjunctions of implications, and that this presentation gives a more structured view to the properties manipulated by the analysis than basic disjunctions. This representation based on conjunctions of implications also allows one to better understand the various ratios between performance and precision that each analysis achieves, as it precisely characterizes what amount of information is used to define each case. Last, conjunctions of implications may split sets of states into sets of cases that are not mutually exclusive, while retaining a high level of precision. The next sections will formalize these observations.

3 ABSTRACTION

In this section, we define abstract elements and their concretization, and compare the resulting abstraction with existing approaches.

3.1 Views

Our abstract domain construction utilizes *views* to describe the properties analyses are sensitive to, and tie them using properties that can be expressed as conjunctions of implications. Therefore, we first formalize the meaning of views. Intuitively, a *view* stands for a set of behaviors, thus it is very natural to let them be described using an abstract domain specifically designed for this purpose. As an example, if we consider a transitional semantics, a view v denotes a set of states E, which means that those states are *observed* by v. For instance, in the context of call-site string sensitivity, a view describes all executions that are matching with a given call-string.

In the following, we let $\mathbb{C}$ denote the *concrete domain*, where program behaviors can be described. In most cases, this domain is a powerset over states or execution traces, and unless stated otherwise, we assume that $\mathbb{C}$ is a powerset domain. As each view describes a set of behaviors, views can be defined as an abstraction of the concrete domain.

Definition 3.1 (Views). *Views* are an abstraction defined by

— set of views $\mathbb{V}^\sharp$;
— concretization function $\gamma_\mathbb{V} : \mathbb{V}^\sharp \to \mathbb{C}$.

We can apply this notion of views to the example shown in Section 2 in several manners.

Example 3.2 (Views over Boolean values). We consider the program of Figure 1 and discuss the views that correspond to the abstractions described in Section 2. We write $\mathcal{V}_{\text{bool}}$ for the set of Boolean values $\{\texttt{TRUE}, \texttt{FALSE}\}$, and $\mathbb{M}$ for the set of memory states. We let $\mathbb{V}^\sharp = \mathcal{V}_{\text{bool}}^N$ and let $\gamma_\mathbb{V}$ be defined by

$$\gamma_\mathbb{V}(b_0, \ldots, b_{N-1}) = \{\mu \in \mathbb{M} \mid \forall i,\ \mu(\mathsf{b}_i) = b_i\}.$$

Essentially, these views describe all the possible settings of all the Boolean variables, thus this set of views defines the elements of the partition encountered in the fully sensitive analysis, where sensitivity is parameterized by *values*. Each view corresponds to the left-hand side of an implication in the conjunction of implications describing the abstract states.

We can also describe the cases that arise in the second linear set of views where we defined a conjunction of implications, the left-hand side of which are of the form $\mathsf{b}_i = \texttt{TRUE}$ ($\mathsf{b}_i = \texttt{FALSE}$, respectively):

$$\forall k,\ \gamma_\mathbb{V}(v_{2k}) = \{\mu \in \mathbb{M} \mid \mu(\mathsf{b}_k) = \texttt{TRUE}\},$$
$$\forall k,\ \gamma_\mathbb{V}(v_{2k+1}) = \{\mu \in \mathbb{M} \mid \mu(\mathsf{b}_k) = \texttt{FALSE}\}.$$

Again, we observe that the left-hand side of each implication of the conjunction shown in Section 2 describes exactly one view.

Additionally, cases where *all* Boolean variables are equal to TRUE and where *at least one* Boolean variable is equal to FALSE can be described by views $v_{\forall\texttt{TRUE}}$ and $v_{\exists\texttt{FALSE}}$, the concretizations of which are defined by

$$\gamma_\mathbb{V}(v_{\forall\texttt{TRUE}}) = \{\mu \in \mathbb{M} \mid \forall k,\ \mu(\mathsf{b}_k) = \texttt{TRUE}\},$$
$$\gamma_\mathbb{V}(v_{\exists\texttt{FALSE}}) = \{\mu \in \mathbb{M} \mid \exists k,\ \mu(\mathsf{b}_k) = \texttt{FALSE}\}.$$

Thus, the set of views required for this analysis is

$$\mathbb{V}^\sharp = \{v_0, \ldots, v_{2N-1}, v_{\forall\texttt{TRUE}}, v_{\exists\texttt{FALSE}}\}.$$

Example 3.3 (Views including control states). Example 3.2 defines views that split memory states. Flow sensitivity (Cousot 1981) is a common technique that abstracts separately memory states encountered at different program points. We can combine this sensitivity with the sensitivity shown in Example 3.2 by defining a new set of views $\mathbb{V}^\sharp$ as a product of the set of control states (that can be represented by statement line numbers) with the set of views defined in the previous example.

```
1 :  x = v;
2 :  y = 0;
3 :  if(x ≥ 0){
4 :        y = x + 1;
5 :  } else {
6 :        y = x − 1;
7 :  }
8 :  z = 60/abs(y + 1);
9 :  ...
```

Fig. 2. A partial example code.

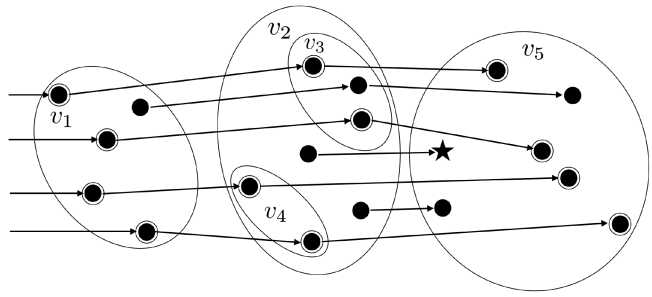

Fig. 3. Transitional semantics of the code in Figure 2.

Example 3.4 (Views over sets of executions). In Section 2, we observed that the case splits needed to perform the analysis could also be expressed in terms of control flow paths, as a trace partitioning. We noted that the structure of the views describing the cases would then be similar, and that the only difference would be that the left-hand sides of the implications manipulated in the analysis would describe properties of execution traces instead of sets. Therefore, as in Example 3.2, we can describe the sets of behaviors that correspond to the left-hand sides of implications using views. For instance, if we consider the full path sensitivity:

— $\mathbb{V}^\sharp$ should denote the set of all control flow paths through the N conditions;
— $\gamma_{\mathbb{V}}$ should map each control flow path to the set of all executions that follow it.

This set of views is isomorphic to the set of views shown in Example 3.2.

We present a last example, which is simpler than the two previous ones, and that we will refer to further in this section:

Example 3.5 (Views over a basic transitional semantics). We consider the piece of code shown in Figure 2. Figure 3 illustrates the transitional semantics of this program, and displays a few program executions, starting from various values of v, as well as a few other transitions that are not part of complete traces. Each node denotes a state, and circled nodes denote states that are reachable from some initial state. Transitions are depicted by arrows. A star denotes a failure state. The four complete traces (starting from a double circle state) correspond to cases where v is initially equal to -4, -3, 1, or 3.

Moreover, we define the following views:

— v_1 describes all states observed at line 3 (after executing the assignments at lines 1 and 2);
— v_2 describes all states observed after executing the whole **if**-statement (i.e., at the beginning of line 8);
— v_3 denotes all the states observed after executing the code at line 4 (at the exit of the true branch);
— v_4 denotes all the states observed after executing the code at line 6 (at the exit of the false branch);
— v_5 denotes all the states observed at the end of the program (after the assignment of line 8 is executed).

This informal description defines a set of views $\mathbb{V}^\sharp$ together with a $\gamma_{\mathbb{V}}$ function.

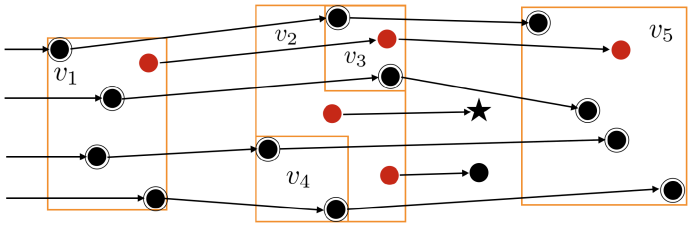

Fig. 4. Abstraction of reachable states in the views in Figure 3.

3.2 Sensitive Abstraction Based on a Loose Cardinal Power Domain

We can now define our general sensitive abstract domain. In the following paragraphs, we provide several definitions that are essentially equivalent, but are stated under distinct sets of assumptions and provide additional insight.

Abstract States and Concretization Relation, Based on a Cardinal Power. In Section 2, we have described program invariants as conjunctions of implications. Since we have now proposed views as an abstraction for the behaviors used as a left-hand side of these implications, we can now describe such a conjunction of implications as follows: *whenever a concrete behavior (say, a program state) satisfies the property expressed by a view v, then it should also satisfy the right-hand side attached to that view.* This suggests that conjunctions of implications can be described as a reduced cardinal power (Cousot and Cousot 1979), where views abstract the properties in the left-hand sides of implications, whereas another abstract domain is used to abstract the formulas in the right-hand sides.

To do this, we assume for now that the concrete semantics of a program can be described as a set of "behaviors," where behaviors are typically program states or execution traces. We let $\mathbb{E}$ denote the set of all behaviors and write $\mathbb{C} = \mathcal{P}(\mathbb{E})$. Then, a view abstracts a set of behaviors. Furthermore, we let $\mathbb{A}^\sharp$ denote an abstract domain, and $\gamma_{\mathbb{A}} : \mathbb{A}^\sharp \to \mathbb{C}$ be a concretization into sets of behaviors. We can now define the sensitive abstract domain.

Definition 3.6 (Sensitive abstract domain and concretization). The *sensitive abstract domain* $\mathbb{S}^\sharp$ and its concretization $\gamma_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{C}$ are defined by

$$\begin{aligned} \mathbb{S}^\sharp &= \mathbb{V}^\sharp \longrightarrow \mathbb{A}^\sharp, \\ \gamma_{\mathbb{S}}(s^\sharp) &= \left\{ e \in \mathbb{E} \mid \forall v \in \mathbb{V}^\sharp,\ e \in \gamma_{\mathbb{V}}(v) \Longrightarrow e \in \gamma_{\mathbb{A}}(s^\sharp(v)) \right\}. \end{aligned}$$

We note that we can write this definition of the concretization in other, equivalent forms:

$$\begin{aligned} \gamma_{\mathbb{S}}(s^\sharp) &= \textstyle\bigcap_{v \in \mathbb{V}^\sharp} \left\{ e \in \mathbb{E} \mid e \in \gamma_{\mathbb{V}}(v) \Longrightarrow e \in \gamma_{\mathbb{A}}(s^\sharp(v)) \right\} \\ &= \textstyle\bigcap_{v \in \mathbb{V}^\sharp} \left((\mathbb{E} \setminus \gamma_{\mathbb{V}}(v)) \cup \gamma_{\mathbb{A}}(s^\sharp(v)) \right). \end{aligned}$$

Example 3.7 (Abstraction based on a cardinal power). Figure 4 displays an abstraction for the views that were defined in Example 3.5. Each rectangle maps a view into an abstraction of the states that satisfy it. To set up an instance of the sensitive abstraction, which is based on this set of views, we simply need to fix $\mathbb{A}^\sharp$. In this example, we let it be the interval abstract domain: $\gamma_{\mathbb{A}}$ maps an interval $[a, b]$ into the set of states such that the value of variable y lies in that interval. Then, v_3 is mapped to $[1, +\infty[$, v_4 is mapped to $]-\infty, -2]$, and v_2 is mapped to $]-\infty, +\infty[$. We note that the states corresponding to views v_3 and v_4 are included into those corresponding to v_2, which clearly shows that this abstraction differs from an abstraction based on disjunctions.

The other examples of Section 3.1 lead to similar abstractions.

Representation of sensitive abstract states. When designing an abstract domain, the choice of the computer representation of abstract elements is paramount to the efficiency of static analysis algorithms, therefore we briefly discuss the possible representations for sensitive abstract states. Abstract operations on abstract states will be presented in Section 4 and typically need to access to or iterate over some views of the abstract states they operate on. As a consequence, and when the set of views $\mathbb{V}^\sharp$ is finite, dictionary structures provide efficient algorithms and thus are a good fit. An important remark is that sparse representations can be used, so as to avoid enumerating all views in all abstract states: in particular, when an abstract state $s^\sharp$ maps a view v to the $\top$ element of $\mathbb{A}^\sharp$, this mapping does not need be represented. Similarly, other redundant mappings may be omitted resulting in more compact representations. When the set of views $\mathbb{V}^\sharp$ is infinite, such sparse representations are required to ensure that dictionary structures can be used.

A Definition Based on Galois-Connection. It is also common to define abstraction relations using *abstraction* functions, which map concrete behaviors into the most precise abstract property that describes them, thus we propose to give such a definition for sensitive abstractions. As in the previous paragraph, concrete domain $\mathbb{C}$ is the powerset of behaviors ($\mathbb{C} = \mathcal{P}(\mathbb{E})$) and is ordered by inclusion $\subseteq$. We now assume that the abstract domain $\mathbb{A}^\sharp$ also features an order relation $\sqsubseteq_\mathbb{A}$ and an abstraction function $\alpha_\mathbb{A} : \mathbb{C} \to \mathbb{A}^\sharp$ that forms a *Galois-connection* together with $\gamma_\mathbb{A}$, which means that

$$\forall c \in \mathbb{C},\ \forall a \in \mathbb{A}^\sharp,\ \alpha_\mathbb{A}(c) \sqsubseteq_\mathbb{A} a \iff c \subseteq \gamma_\mathbb{A}(a).$$

We note a pair of adjoint functions satisfying this property as follows:

$$(\mathbb{C}, \subseteq) \underset{\alpha_\mathbb{A}}{\overset{\gamma_\mathbb{A}}{\leftrightarrows}} (\mathbb{A}^\sharp, \sqsubseteq_\mathbb{A}).$$

Then, we can define the sensitive abstraction as a Galois connection as well, where $\mathbb{S}^\sharp$ and $\gamma_\mathbb{S}$ are defined as in Definition 3.6.

Lemma 3.8 (Sensitive abstraction Galois connection). *We let $\sqsubseteq_\mathbb{S}$ be the pointwise extension of $\sqsubseteq_\mathbb{A}$ over $\mathbb{S}^\sharp = \mathbb{V}^\sharp \longrightarrow \mathbb{A}^\sharp$ and $\alpha_\mathbb{S}$ be defined by*

$$\alpha_\mathbb{S}(E) = \lambda v \in \mathbb{V}^\sharp \cdot \alpha_\mathbb{A}(E \cap \gamma_\mathbb{V}(v)).$$

Then, $\alpha_\mathbb{S}$ and $\gamma_\mathbb{S}$ define a Galois connection:

$$(\mathbb{C}, \subseteq) \underset{\alpha_\mathbb{S}}{\overset{\gamma_\mathbb{S}}{\leftrightarrows}} (\mathbb{S}^\sharp, \sqsubseteq_\mathbb{S}).$$

The proof is provided in Appendix A.

Example 3.9 (Abstraction based on a Galois-connection). We keep the same notations as in Example 3.7. Then, the base abstract domain $\mathbb{A}^\sharp$ clearly defines a Galois connection, and so does $\mathbb{S}^\sharp$, following Definition 3.8.

Example 3.10 (Pure sensitive abstraction). A particular case of Galois-connection is the identity abstraction, which is defined by $\mathbb{A}^\sharp = \mathbb{C}$, $\alpha_\mathbb{A} = \gamma_\mathbb{A} = \lambda(C \in \mathbb{C}) \cdot C$. The resulting sensitive abstraction simply associates to each view the behaviors that are associated to it. As it composes no other abstraction, we call it the *pure sensitive abstraction*. This sensitive abstraction is especially interesting as the sensitive abstraction defined by any concretization $\gamma_\mathbb{A}$ (unnecessarily identity) can be derived from the pure sensitive abstraction by composing $\gamma_\mathbb{A}$ pointwisely.

Notes on Generality. While our abstract domain does not put any restrictions on $\mathbb{V}^\sharp$, most of the existing sensitive analyses require two conditions on $\mathbb{V}^\sharp$ and $\gamma_\mathbb{V}$ in order to achieve soundness and reasonable precision. They require them to define

(1) a *covering*

$$\bigcup_{v \in \mathbb{V}^\sharp} \gamma_{\mathbb{V}}(v) = \mathbb{E}, \text{ and}$$

(2) a *disjoint* family of sets

$$\forall v_1, v_2 \in \mathbb{V}^\sharp,\ v_1 \neq v_2 \Rightarrow \gamma_{\mathbb{V}}(v_1) \cap \gamma_{\mathbb{V}}(v_2) = \emptyset.$$

Therefore, they rely on a *partitioning* of concrete domain $(\mathcal{P}(\mathbb{E}), \subseteq)$. On the contrary, our sensitive abstraction framework does not require either of these conditions be satisfied but it also ensures soundness and can often provide higher precision than the traditional analyses. For example, in Figure 3, $\mathbb{V}^\sharp$ is a covering but is not a disjoint family of sets, as observed in Example 3.7.

Concretization by Implicit Partitions Filtering. To provide additional insight into the expressiveness of our construction, we propose another definition of the concretization. In this paragraph, we assume that $\mathbb{A}^\sharp$ features a $\sqcap_{\mathbb{A}}$ for the greatest lower bound operator. First, we remark that we can define an equivalence relation over $\mathbb{E}$ as follows.

Definition 3.11 (Equivalent behaviors $\equiv_{\mathbb{V}}$). For all elements $x_1, x_2 \in \mathbb{E}$, we note $x_1 \equiv_{\mathbb{V}} x_2$ if and only if

$$\forall v \in \mathbb{V}^\sharp,\ x_1 \in \gamma_{\mathbb{V}}(v) \iff x_2 \in \gamma_{\mathbb{V}}(v).$$

The relation $\equiv_{\mathbb{V}}$ explicitly represents the implicit partitions induced by $\mathbb{V}^\sharp$ on $\mathbb{E}$. Then, the elements of $\mathbb{E}/_{\equiv_{\mathbb{V}}}$ serve as atoms of $\mathbb{V}^\sharp$ in the sense that, for each $v \in \mathbb{V}^\sharp$, we can compute $\gamma_{\mathbb{V}}(v)$ as the disjoint union of a family of equivalence classes from $\mathbb{E}/_{\equiv_{\mathbb{V}}}$. We can now prove the following lemma.

Lemma 3.12. *Given abstract domain $\mathbb{A}^\sharp$, and concretization function $\gamma_{\mathbb{A}} : \mathbb{A}^\sharp \to \mathbb{C}$, we can write the concretization function $\gamma_{\mathbb{S}}$ of Definition 3.6 as follows:*

$$\gamma_{\mathbb{S}}(s^\sharp) = \bigcup \left\{ C \cap \gamma_{\mathbb{A}} \left(\bigsqcap_{\mathbb{A}} \{ s^\sharp(v) | C \subseteq \gamma_{\mathbb{V}}(v) \} \right) \middle| C \in \mathbb{E}/_{\equiv_{\mathbb{V}}} \right\}. \tag{6}$$

Essentially, this lemma shows that the concretization can be viewed as that of a disjunction, after refining the view information into the partition it induces, and that is defined by $\equiv_{\mathbb{V}}$. Moreover, if $\mathbb{V}^\sharp$ is partitioning (i.e., if $\equiv_{\mathbb{V}}$ boils down to the equality relation), the above definition of $\gamma_{\mathbb{S}}$ can be simplified into

$$\gamma_{\mathbb{S}}(s^\sharp) = \bigcup \{ \gamma_{\mathbb{V}}(v) \cap \gamma_{\mathbb{A}} \circ s^\sharp(v) | v \in \mathbb{V}^\sharp \}. \tag{7}$$

General Definition. In the previous paragraphs, we have defined concretization and abstraction functions for the sensitive abstract domain under the assumption that the concrete domain is a powerset, as required in the original definition of the cardinal power (Cousot and Cousot 1979). While this is the most common case, we can also define the abstraction even when that is not the case using more general assumptions. To achieve this, we use in the following paragraphs more general implicational structures than the conventional cardinal power construction. We observe that Giacobazzi and Scozzari have achieved such results using the notion of Heyting completion in Giacobazzi and Scozzari (1998). More recently, Giacobazzi, Ranzato and Scozzari have proposed a completion based on linear implication in Giacobazzi et al. (2005). These two works have been applied to the static analysis of logic programs, and aim for a notion of sensitivity comparable to the one we consider in this article in the context of logic languages.

The traditional cardinal power domain and its generalized variant semi-quantale (Giacobazzi and Ranzato 1999) require the exponents of the domains be complete lattices. By contrast, our abstract domain allows $\mathbb{V}^\sharp$ to be an arbitrary set. We call our abstract domain a *loose cardinal*

power domain. In this paragraph, and as in these works, we require the concrete domain $\mathbb{C}$ be a *complete Heyting algebra*. A Heyting algebra is a complete lattice (we note $\sqsubseteq_{\mathbb{C}}$ for the order relation, $\bot_{\mathbb{C}}$ for the infimum, $\top_{\mathbb{C}}$ for the supremum, $\sqcap_{\mathbb{C}}$ for the greatest lower bound operator, and $\sqcup_{\mathbb{C}}$ for the least upper bound operator), which also provides an operator $\hookrightarrow_{\mathbb{C}}$ such that

$$\forall c_0, c_1, c_2 \in \mathbb{C},\ c_0 \sqcap_{\mathbb{C}} c_1 \sqsubseteq_{\mathbb{C}} c_2 \iff c_0 \sqsubseteq_{\mathbb{C}} c_1 \hookrightarrow_{\mathbb{C}} c_2.$$

Last, we let $\mathbb{A}^\sharp$ denote an abstract domain with a monotone concretization function $\gamma_{\mathbb{A}} : \mathbb{A}^\sharp \to \mathbb{C}$.

Definition 3.13 (Loose cardinal power abstract domain). The *loose cardinal power abstract domain* $\mathbb{S}^\sharp$ and its concretization $\gamma_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathbb{C}$ are defined by

$$\mathbb{S}^\sharp = \mathbb{V}^\sharp \longrightarrow \mathbb{A}^\sharp,$$
$$\gamma_{\mathbb{S}}(s^\sharp) = \bigsqcap\nolimits_{\mathbb{C}} \left\{ \gamma_{\mathbb{V}}(v) \hookrightarrow_{\mathbb{C}} \gamma_{\mathbb{A}} \circ s^\sharp(v) \mid v \in \mathbb{V}^\sharp \right\}.$$

Furthermore, if $\mathbb{A}^\sharp$ also has an abstraction function $\alpha_{\mathbb{A}}$ that defines a Galois connection together with $\gamma_{\mathbb{A}}$, then $\mathbb{S}^\sharp$ also defines a Galois connection using the pointwise extension of the order relation over $\mathbb{A}^\sharp$ and with the abstraction function $\alpha_{\mathbb{S}}$ defined by

$$\alpha_{\mathbb{S}}(E) = \lambda v \in \mathbb{V}^\sharp . \alpha_{\mathbb{A}}(E \sqcap_{\mathbb{C}} \gamma_{\mathbb{V}}(v)).$$

Additionally, Definition 3.13 generalizes Definition 3.6: when concrete domain $\mathbb{C}$ is a powerset domain (as in Lemma 3.8), it also satisfies the assumptions of Definition 3.13, which is then equivalent to Lemma 3.8. Indeed, if $\mathbb{C} = \mathcal{P}(\mathbb{E})$, then

$$\gamma_{\mathbb{V}}(v) \hookrightarrow_{\mathbb{C}} \gamma_{\mathbb{A}} \circ s^\sharp(v) = (\mathbb{E} \setminus \gamma_{\mathbb{V}}(v)) \cup \gamma_{\mathbb{A}}(s^\sharp(v)).$$

3.3 Internal reduction

As observed earlier, an element $s^\sharp$ of the abstract domain $\mathbb{S}^\sharp$ intuitively denotes a conjunction of implications. This logical structure actually allows one to *strengthen* abstract elements by combining the constraints they express so as to produce stronger constraints. In general, if we know $X_0 \implies Y_0$ and $X_1 \implies Y_1$, then we can derive $(X_0 \wedge X_1) \implies (Y_0 \wedge Y_1)$. In our abstract domain, this basic reasoning principle also holds, and means that the property of "view" $(X_0 \wedge X_1)$ can be refined using the properties of "views" X_0 and X_1.

This process is implemented by a *reduction operation* (Cousot and Cousot 1979). In the following, a reduction operator is a function $\rho_{\mathbb{S}} : \mathbb{S}^\sharp \longrightarrow \mathbb{S}^\sharp$, which is sound (i.e., such that $\forall s^\sharp,\ \gamma_{\mathbb{S}}(\rho_{\mathbb{S}}(s^\sharp)) = \gamma_{\mathbb{S}}(s^\sharp)$) and reductive (i.e., such that $\forall s^\sharp,\ \rho_{\mathbb{S}}(s^\sharp) \sqsubseteq_{\mathbb{S}} s^\sharp$).

In the following two paragraphs, we require $\mathbb{A}^\sharp$ to have a complete lattice structure and lift this assumption afterwards. We write $\sqcup_{\mathbb{A}}$ ($\sqcap_{\mathbb{A}}$, respectively) for the least upper bound (greatest lower bound, respectively) operator.

Reduction in the Galois-Connection Setup. In the following, we define such an operator, assuming that $\mathbb{C}$ is a powerset over $\mathbb{E}$ and that $\mathbb{A}^\sharp$ defines a Galois connection over $\mathbb{C}$. Thus, we use the notation of Lemma 3.8. Under these assumptions, there exists an optimal reduction operator, which boils down to the lower closure operator $\alpha_{\mathbb{S}} \circ \gamma_{\mathbb{S}}$. As this is often too costly to compute, we will simply search for an over-approximation of this ideal operator. The reduction principle we expose below generalizes to the more general sensitive abstract domain definitions, yet at the cost of more complex formulas, albeit based on the same principle. We recall the definition of $\alpha_{\mathbb{S}} \circ \gamma_{\mathbb{S}}$:

$$\alpha_{\mathbb{S}} \circ \gamma_{\mathbb{S}}(s^\sharp) = \lambda(v_0 \in \mathbb{V}^\sharp) \cdot \alpha_{\mathbb{A}}(\gamma_{\mathbb{S}}(s^\sharp) \cap \gamma_{\mathbb{V}}(v_0)).$$

First, note the following result:

Lemma 3.14. *For $s^\sharp \in \mathbb{S}^\sharp$ and $v \in \mathbb{V}^\sharp$, we have $\alpha_{\mathbb{A}}(\gamma_{\mathbb{S}}(s^\sharp) \cap \gamma_{\mathbb{V}}(v)) \sqsubseteq_{\mathbb{A}} s^\sharp(v)$.*

The proof of Lemma 3.14 is given in Appendix A. Informally speaking, this lemma states that $s^\sharp(v)$ is a sound approximation of the set of all the concrete behaviors that satisfy both $s^\sharp$ and v. The intuition underlying this result is that of the logical modus ponens rule where we deduce Y from $(X \Longrightarrow Y) \wedge X$, and where view v corresponds to X.

While $s^\sharp(v)$ conservatively approximates the set of all behaviors satisfying v, we can actually produce a more precise abstraction using the contribution of other views. Indeed, if $E \subseteq \gamma_{\mathbb{V}}(v)$, and since $\alpha_{\mathbb{A}}$ is monotone, we get

$$\alpha_{\mathbb{A}}(\gamma_{\mathbb{S}}(s^\sharp) \cap E) \sqsubseteq_{\mathbb{A}} s^\sharp(v) \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}} \circ \gamma_{\mathbb{V}}(v). \tag{8}$$

This formula is equivalent to Lemma 3.14 in the context of a set of behaviors E that is subsumed by a single view v. We can actually also generalize this formula to the case where *several* views subsume the set E. To express this, we define a notion of covering that collects all the sets of views that over-approximate a given concrete behavior.

Definition 3.15 (Covering of a set of concrete behaviors). For all $E \subseteq \mathbb{E}$, we let the *covering* of E be defined by

$$\mathbf{Covers}_{\mathbb{V}^\sharp}(E) = \left\{ V \subseteq \mathbb{V}^\sharp \;\middle|\; E \subseteq \bigcup_{v \in V} \gamma_{\mathbb{V}}(v) \right\}.$$

We can now generalize Equation (8) to the concrete behaviors that are captured by many views.

Lemma 3.16. *Let $s^\sharp \in \mathbb{S}^\sharp$, $E \subseteq \mathbb{E}$ and $V \in \mathbf{Covers}_{\mathbb{V}^\sharp}(E)$. Then*

$$\alpha_{\mathbb{A}}(\gamma_{\mathbb{S}}(s^\sharp) \cap E) \sqsubseteq_{\mathbb{A}} \bigsqcup\nolimits_{\mathbb{A}} \{ s^\sharp(v) \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}} \circ \gamma_{\mathbb{V}}(v) | v \in V \}.$$

Moreover, we can generalize this to all coverings of a given $E \subseteq \mathbb{E}$:

$$\alpha_{\mathbb{A}}(\gamma_{\mathbb{S}}(s^\sharp) \cap E) \sqsubseteq_{\mathbb{A}} \bigsqcap\nolimits_{\mathbb{A}} \left\{ \bigsqcup\nolimits_{\mathbb{A}} \left\{ s^\sharp(v) \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}} \circ \gamma_{\mathbb{V}}(v) \middle| v \in V \right\} \middle| V \in \mathbf{Covers}_{\mathbb{V}^\sharp}(E) \right\}.$$

The proof of Lemma 3.16 is provided in Appendix A. If we select $E = \gamma_{\mathbb{V}}(v_0)$, we obtain the following reduction operator.

Lemma 3.17 (Reduction Operator). *We let $\rho_{\mathbb{S}}$ be defined by*

$$\rho_{\mathbb{S}}(s^\sharp) = \lambda(v_0 \in \mathbb{V}^\sharp) \cdot \bigsqcap\nolimits_{\mathbb{A}} \left\{ \bigsqcup\nolimits_{\mathbb{A}} \left\{ s^\sharp(v) \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}} \circ \gamma_{\mathbb{V}}(v) \middle| v \in V \right\} \middle| V \in \mathbf{Covers}_{\mathbb{V}^\sharp}(\gamma_{\mathbb{V}}(v_0)) \right\}.$$

Then

$$\forall s^\sharp,\ \alpha_{\mathbb{S}} \circ \gamma_{\mathbb{S}}(s^\sharp) \sqsubseteq_{\mathbb{S}} \rho_{\mathbb{S}}(s^\sharp).$$

Moreover, $\rho_{\mathbb{S}}(s^\sharp) \sqsubseteq_{\mathbb{S}} s^\sharp$, thus $\rho_{\mathbb{S}}$ defines a reduction operator.

The proof of Lemma 3.17 is provided in Appendix A. The inclusion $\rho_{\mathbb{S}}(s^\sharp) \sqsubseteq_{\mathbb{S}} s^\sharp$ follows from the fact that $\{v_0\}$ is a covering of $\gamma_{\mathbb{V}}(v_0)$, thus $\rho_{\mathbb{S}}(s^\sharp)(v_0) \sqsubseteq_{\mathbb{A}} s^\sharp(v_0)$.

Note that when there are multiple coverings for E, it performs the meet operation on the values from all the coverings. Since we have multiple sound approximations using each covering, we can get a more precise value by performing the meet operation on all of them.

General Definition of Reduction. We now propose to extend the previous definition to the case where the abstraction defined by $\mathbb{A}^\sharp$ does not feature an abstraction function, and we assume only a concretization $\gamma_{\mathbb{A}} : \mathbb{A}^\sharp \longrightarrow \mathbb{C}$. In the previous definition, we used $\alpha_{\mathbb{A}}$ in order to let an element of $\mathbb{V}^\sharp$ refine one of $\mathbb{A}^\sharp$, thus we have to define an operator with the same effect, and that does not require $\alpha_{\mathbb{A}}$. To achieve this, we require domain $\mathbb{A}^\sharp$ to feature a sound *view restriction* operator $\mathbf{restrict}^\sharp$.

Definition 3.18 (Restriction to a view). Function $\mathbf{restrict}^\sharp : \mathbb{V}^\sharp \times \mathbb{A}^\sharp \to \mathbb{A}^\sharp$ is a sound *view restriction* operator if it satisfies the following condition:

$$\forall a^\sharp \in \mathbb{A}^\sharp,\ \forall v \in \mathbb{V}^\sharp,\ \gamma_{\mathbb{A}}(a^\sharp) \cap \gamma_{\mathbb{V}}(v) \subseteq \gamma_{\mathbb{A}}(\mathbf{restrict}^\sharp(v, a^\sharp)).$$

Following the same reasoning as in the previous paragraph, we can derive the reduction formula below that maps a view v to an abstract element derived from all the covering of v.

Lemma 3.19 (Reduction Operator). *We let $\rho_{\mathbb{S}}$ be defined by*

$$\rho_{\mathbb{S}}(s^\sharp) = \lambda(v_0 \in \mathbb{V}^\sharp) \cdot \bigsqcap \mathbb{A} \left\{ \bigsqcup\nolimits_{\mathbb{A}} \left\{ \mathbf{restrict}^\sharp(v, s^\sharp(v)) \middle| v \in V \right\} \middle| V \in \mathbf{Covers}_{\mathbb{V}^\sharp}(\gamma_{\mathbb{V}}(v_0)) \right\}.$$

Then

$$\forall s^\sharp,\ \gamma_{\mathbb{S}}(s^\sharp) \sqsubseteq_{\mathbb{S}} \gamma_{\mathbb{S}}(\mathbf{reduce}^\sharp_{\mathbb{S}}(s^\sharp)).$$

Lemma 3.19 provides a more general definition of reduction operator than Lemma 3.17. Indeed, in the setup of the latter, $\lambda(v, a^\sharp) \cdot a^\sharp \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}} \circ \gamma_{\mathbb{V}}(v)$ defines a sound $\mathbf{restrict}^\sharp$ operator. Moreover, the proof of Lemma 3.19 is very similar to that of Lemma 3.17 (it relies on the properties of concretization functions instead of those of Galois connections).

Computable Reduction Operator. The reduction operators defined in Lemma 3.19 and Lemma 3.17 are not necessarily computable, or even defined if $\mathbb{A}^\sharp$ is not a complete lattice (which is common). To define a sound reduction operator, we simply need to require sound over-approximations for concrete unions and intersections. We assume that $\mathbf{join}^\sharp$ defines a sound approximation of the concrete state set union ($\forall a_0^\sharp, a_1^\sharp \in \mathbb{A}^\sharp,\ \gamma_{\mathbb{A}}(a_0^\sharp) \cup \gamma_{\mathbb{A}}(a_1^\sharp) \subseteq \gamma_{\mathbb{A}}(\mathbf{join}^\sharp(a_0^\sharp, a_1^\sharp))$), and that $\mathbf{meet}^\sharp$ defines a sound approximation of the concrete state set intersection ($\forall a_0^\sharp, a_1^\sharp \in \mathbb{A}^\sharp,\ \gamma_{\mathbb{A}}(a_0^\sharp) \cap \gamma_{\mathbb{A}}(a_1^\sharp) \subseteq \gamma_{\mathbb{A}}(\mathbf{meet}^\sharp(a_0^\sharp, a_1^\sharp))$). Then, we derive the following reduction operator.

Theorem 3.20 (Computable, Sound Reduction Operator). *We let $\mathbf{reduce}^\sharp_{\mathbb{S}}$ be defined by*

$$\begin{array}{l} \forall (v_0 \in \mathbb{V}^\sharp), \\ \quad \mathbf{reduce}^\sharp_{\mathbb{S}}(s^\sharp)(v_0) = \\ \qquad \mathbf{meet}^\sharp \left(\left\{ \mathbf{join}^\sharp \left(\left\{ \mathbf{restrict}^\sharp(v, s^\sharp(v)) \middle| v \in V \right\} \right) \middle| V \in \mathbf{Covers}_{\mathbb{V}^\sharp}(\gamma_{\mathbb{V}}(v_0)) \right\} \right). \end{array}$$

Then

$$\forall s^\sharp,\ \gamma_{\mathbb{S}}(s^\sharp) \sqsubseteq_{\mathbb{S}} \gamma_{\mathbb{S}}(\mathbf{reduce}^\sharp_{\mathbb{S}}(s^\sharp)).$$

This result follows directly from Lemma 3.19, and from the use of abstract operations that over-approximate the concrete ones.

Discussion and Examples. In logical terms, this operation amounts to applying the following logical principle:

$$\left(\left[\bigwedge_{i \in I} (X_i \to Y_i) \right] \wedge \left[\bigvee_{i \in I} X_i \right] \right) \Longrightarrow \left[\bigvee_{i \in I} Y_i \right].$$

The following example shows how this reduction works on a basic example.

Example 3.21 (Internal reduction basic strengthening). In this example, we let $\mathbb{A}^\sharp = \mathbb{C} = (\mathcal{P}(\{1, 2\}), \subseteq)$, and assume that $\gamma_{\mathbb{A}}$ is the identity function. In addition, we let $\mathbb{V}^\sharp = \{v_1, v_2\}$, and

$$\gamma_{\mathbb{V}}(v_1) = \{1\}, \qquad \gamma_{\mathbb{V}}(v_2) = \{2\}.$$

We consider an abstract state $s^\sharp = \{v_1 \mapsto \{2\}, v_2 \mapsto \emptyset\}$. Obviously, this abstract state is not tight, as v_1 is mapped into an element that is not compatible with it. The above reduction operator

strengthens it into the desired abstract state, namely, $\{v_1 \mapsto \emptyset, v_2 \mapsto \emptyset\}$. Indeed, if we consider the covering $V = \{v_1\}$ of itself and apply Lemma 3.17, we obtain $\rho_{\mathbb{S}}(s^\sharp)(v_1) = \emptyset$.

In practice, using the exact definition of the coverings is often too expensive to compute, or even impossible when the set of views is infinite. However, a static analysis tool may perform a partial reduction using a safe approximation of coverings.

Definition 3.22 (Sound approximate covering). A *sound approximate covering* is a function $\mathbf{Covers}^\sharp_{\mathbb{V}^\sharp} : \mathbb{C} \to \mathcal{P}(\mathcal{P}(\mathbb{V}^\sharp))$ such that

$$\forall E \in \mathbb{C},\ \mathbf{Covers}^\sharp_{\mathbb{V}^\sharp}(E) \subseteq \mathbf{Covers}_{\mathbb{V}^\sharp}(E).$$

Lemma 3.17 still holds when using $\mathbf{Covers}^\sharp_{\mathbb{V}^\sharp}$ instead of $\mathbf{Covers}_{\mathbb{V}^\sharp}$, and the original $\mathbf{Covers}_{\mathbb{V}^\sharp}$ is also a sound $\mathbf{Covers}^\sharp_{\mathbb{V}^\sharp}$.

3.4 Disjunctions and Conjunctions of Implications

We have observed that the concretization function of the sensitive abstraction naturally writes down as a conjunction of implications (Definition 3.6 and Definition 3.13), which is consistent with the intuition drawn from the example of Section 2. We now propose to compare our approach which relies on conjunctions of implications with the approaches that rely on disjunctions. We first consider when a family of views is disjoint and then when views have non-empty intersections.

Several *partitioning abstractions* such as those proposed in Rival and Mauborgne (2007) and Jeannet (2003) perform a two step abstraction: the first phase splits a set of concrete behaviors into a partition and the second phase abstracts each of those subsets separately. These works let a concretization function be a union of terms, which effectively boils down to a disjunction:

$$\gamma_{\text{part}}(s^\sharp) = \bigcup_{v \in \mathbb{V}^\sharp} \gamma_{\mathbb{A}} \circ s^\sharp(v). \tag{9}$$

We note that we have also introduced in Lemma 3.12 a definition of the sensitive abstract domain concretization function as a least upper bound. Assuming $\mathbb{V}^\sharp$ is partitioning (and that $\mathbb{C}$ is a powerset domain in order to simplify notations), this formula boils down to

$$\gamma_{\mathbb{S}}(s^\sharp) = \bigcup_{v \in \mathbb{V}^\sharp} \gamma_{\mathbb{V}}(v) \cap \gamma_{\mathbb{A}} \circ s^\sharp(v). \tag{10}$$

Although both definitions include a set union, this definition of $\gamma_{\mathbb{S}}$ is radically different from that of γ_{part} though: each of the "disjuncts" includes an intersection with the concretization $\gamma_{\mathbb{V}}(v)$ of the view under consideration, which makes the sensitive abstract domain concretization function tighter. We illustrate this issue in the following example.

Example 3.23 (Comparison of concretizations). In this example, we reuse the same notations as in Example 3.21: we let $\mathbb{A}^\sharp = \mathbb{C} = (\mathcal{P}(\{1,2\}), \subseteq)$, assume that γ_0 is the identity function, let $\mathbb{V}^\sharp = \{v_1, v_2\}$, and let $\gamma_{\mathbb{V}}(v_1) = \{1\}, \gamma_{\mathbb{V}}(v_2) = \{2\}$. We consider an abstract value $s^\sharp = \{v_1 \mapsto \{2\}, v_2 \mapsto \emptyset\}$. Then, if we concretize it using γ_{part}, we obtain

$$\gamma_{\text{part}}(s^\sharp) = \{2\}.$$

However, this concretization is not tight, as element 2 is actually incompatible with view v_1, since $\gamma_{\mathbb{V}}(v_1) = \{1\}$. By contrast, the sensitive abstract domain concretization returns the more precise result that was expected:

$$\gamma_{\mathbb{S}}(s^\sharp) = \emptyset.$$

This means that the sensitive abstract domain actually retains *more* information on views; this information can actually be retrieved thanks to the reduction operator of Lemma 3.17:

$$\rho_{\mathbb{S}}(s^\sharp) = \{v_1 \mapsto \emptyset, v_2 \mapsto \emptyset\}.$$

This element is actually the "optimal representation" of $\gamma_{\mathbb{S}}(s^\sharp)$. This reduction cannot be performed using the disjunction-based concretization γ_{part}.

In this instance, the concretization of the partitioning abstract domain is clearly imprecise as it admits the concrete value 2 even though it is bound to a view that is incompatible with it. A way to address this issue consists in making sure that the analysis constructs only abstract values such that

$$\forall v \in \mathbb{V}^\sharp,\ s^\sharp(v) \subseteq \gamma_{\mathbb{V}}(v).$$

To summarize, this observation entails that the disjunction-based abstraction is less tight, and does not allow a precise reduction as shown in Section 3.3, as it cannot take advantage of the meanings of views to constrain concrete behaviors.

Note that this does not imply that the partitioning abstract domain will lead to inherently less precise analyses in general. Indeed, the restriction on abstract states that is mentioned above allows one to mitigate the loss in precision. Furthermore, analyses such as Rival and Mauborgne (2007) utilize refined transition systems that are consistent with the partition to prevent the effect illustrated in Example 3.23 from occurring. However, it means that the sensitive abstract domain concretization $\gamma_{\mathbb{S}}$ naturally allows for tight reduction opportunities in cases where γ_{part} does not.

Now, let us consider the case where views have non-empty intersections, that is, where several views subsume a set of behaviors E. Our approach relying on conjunctions of implications produces a precise reduction by Lemma 3.17 using a covering of E. When there are multiple coverings for E, we can get a more precise value by performing the meet operation on the values from all the coverings. On the contrary, because γ_{part} defined in Equation (9) simply joins all the abstract values in $s^\sharp$, it is less precise than our approach. While $\gamma_{\mathbb{S}}$ defined in Equation (10) produces more precise values than γ_{part}, it produces less precise values than our approach, because it joins abstract values of *all* the views that have non-empty intersections with E. Note that our conjunctions of implications can join *a subset V of* the views if V covers E.

4 ABSTRACT INTERPRETATION: ABSTRACT TRANSFER FUNCTIONS AND ITERATION

In this section, we demonstrate how to design a sound approximation of the standard concrete semantics of transition systems, using a *static* sensitive abstraction. The assumption that the sensitivity is static simply asserts that the set of views to be used in the analysis should be fixed once and for all, before the analysis (the design of an analysis using a *dynamic* sensitive abstraction will be discussed in Section 5), and that it is finite. We base the design of our analysis on a classical standard, forward transitional semantics.

4.1 Concrete Semantics

The analyses presented in this article derive as abstract interpretations of a standard *transitional semantics*. We let a program P be defined by the following components:

— the set of all possible states $\mathbb{S}$;
— the transition relation $\rightsquigarrow \subseteq \mathbb{S} \times \mathbb{S}$;
— the set of initial states $\mathbb{S}_I$.

Forward Semantics. The *single step forward execution* function maps a set of states S into its set of immediate successors (for the sake of clarity, we add primes over the predecessor states in the following subsections):

$$\mathbf{post}(S) ::= \{\sigma \in \mathbb{S} \mid \exists \sigma' \in S,\ \sigma' \rightsquigarrow \sigma\}.$$

This function allows one to define in turn the *forward execution semantics* (or semantics of *reachable states*), which collects all the states reachable by zero, one, or many steps of program execution, starting from any initial state in $\mathbb{S}_I$:

$$[\![P]\!] = \mathbf{lfp}\,(\lambda S \in \mathcal{P}(\mathbb{S}) \cdot \mathbb{S}_I \cup \mathbf{post}(S)) = \bigcup_{n \in \mathbb{N}} \mathbf{post}^n(\mathbb{S}_I).$$

As this function is continuous, and defined over a complete lattice, the existence of the fixpoint and the definition as the join over all iterates follows from standard fixpoint iteration techniques.

In the case of the program of Figure 2, the forward execution semantics iterates the transition relation from the initial states at line 1.

Toward an Abstract Semantics. As in Section 3, we let $\gamma_{\mathbb{V}} : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{S})$ denote the view abstraction, $\gamma_{\mathbb{A}} : \mathbb{A}^\sharp \to \mathcal{P}(\mathbb{S})$ an abstraction of sets of states, and we let $\gamma_{\mathbb{S}} : \mathbb{S}^\sharp \to \mathcal{P}(\mathbb{S})$ represent the sensitive abstraction, where $\mathbb{S}^\sharp = \mathbb{V}^\sharp \to \mathbb{A}^\sharp$.

Given such an abstraction, and concrete forward single step execution **post**, a sound over-approximation of the concrete semantics $[\![P]\!]$ can be computed from

- an *abstract single step forward execution* $[\![P]\!]^\sharp : \mathbb{S}^\sharp \to \mathbb{S}^\sharp$ that is sound in the sense that
$$[\![P]\!] \subseteq \gamma_{\mathbb{S}} \circ [\![P]\!]^\sharp$$
- an *abstract iterator*, using widening if necessary, in order to over-approximate concrete least-fixpoints.

We address the design of an abstract single step forward execution function in Section 4.2, and discuss the construction of an abstract iterator in Section 4.3.

4.2 Abstract Single Step Forward Execution

The traditional way to derive a sound abstract semantic function from a concretization function $\gamma_{\mathbb{S}}$ and a concrete semantic function $[\![P]\!]$ consists in rewriting step by step $[\![P]\!] \circ \gamma_{\mathbb{S}}(s^\sharp)$ into some expression of the form $\gamma_{\mathbb{S}}(X)$. We follow this approach in this section, and apply it step by step, starting with the pure sensitive abstraction which was defined in Example 3.10, and derive analysis functions for general sensitive abstractions from that case.

Derivation of the Pure Sensitive Abstraction. To simplify notations, we assume that the concrete domain is a powerset lattice $\mathbb{C} = \mathcal{P}(\mathbb{E})$, where $\mathbb{E}$ is the set of states $\mathbb{S}$ and let $\mathbb{E} = \mathbb{S}$ (although similar results would hold if we let program behaviors be traces, the formalization would be slightly heavier). As we assume that $\mathbb{A}^\sharp$ defines the pure sensitive abstraction, $\mathbb{A}^\sharp = \mathbb{C}$, $\alpha_{\mathbb{A}}$ and $\gamma_{\mathbb{A}}$ are the identity, and $\mathbb{S}^\sharp = \mathbb{V}^\sharp \to \mathbb{C} = \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{S})$.

Thus, we look for a function $\overline{\mathbf{post}} : (\mathbb{V}^\sharp \to \mathcal{P}(\mathbb{S})) \longrightarrow (\mathbb{V}^\sharp \to \mathcal{P}(\mathbb{S}))$ such that

$$\forall \overline{s} : \mathbb{V}^\sharp \to \mathcal{P}(\mathbb{S}),\ \mathbf{post} \circ \gamma_{\mathbb{S}}(\overline{s}) \subseteq \gamma_{\mathbb{S}} \circ \overline{\mathbf{post}}(\overline{s}).$$

Intuitively, when applied to a set of states S, the semantic function **post** performs one step of computation of a transition system P from any state in S. Therefore, we expect $\overline{\mathbf{post}}$ to perform the same action, albeit maintaining the correspondence between states and views.

To this end, we first define a couple of utility functions. First, for any pair of views v' and v, we define the transition function from v' to v.

Definition 4.1 (Transition from view to view). We let $\mathbf{post}_{v' \to v}$ be defined by

$$\begin{aligned}\mathbf{post}_{v' \to v} : (\mathbb{V}^\sharp \to \mathcal{P}(\mathbb{S})) &\longrightarrow \mathcal{P}(\mathbb{S}) \\ \overline{s} &\longmapsto \gamma_{\mathbb{V}}(v) \cap \mathbf{post}(\overline{s}(v') \cap \gamma_{\mathbb{V}}(v')).\end{aligned}$$

In essence, this function restricts the transition relation to the steps from a state described by view v' into a state described by view v. Moreover, given views v' and v, we say that v' covers a transition $\sigma' \rightsquigarrow \sigma$ if $\sigma \in \mathbf{post}_{v' \to v}(\overline{s})$ for all $\overline{s}$ such that $\sigma' \in \gamma_{\mathbb{S}}(\overline{s})$.

Additionally, given a view $v \in \mathbb{V}^\sharp$, in order to compute the set of states that can be described by v after a transition, we need to account for all the transitions that can reach a state described by this view. On the other hand, if we look for an over-approximation of the successor of an abstract state $\overline{s}$ and of the image of view v by that successor, we are not required to look for all views v' that cover the same transition $\sigma' \rightsquigarrow \sigma$: indeed, covering that transition once is enough to guarantee it will be taken into account by the analysis. Therefore, we define a notion of *predecessor covering*.

Definition 4.2 (Predecessor covering). We call a *predecessor covering* of a view v a set of views V' such that

$$\forall \sigma \in \gamma_{\mathbb{V}}(v),\ \forall \sigma' \in \mathbb{S},\ \sigma' \rightsquigarrow \sigma \Longrightarrow \exists v' \in V',\ \sigma' \in \gamma_{\mathbb{V}}(v').$$

Moreover, we let $P_{\to v}$ denote the set of all predecessor coverings of v.

We also let $\rightsquigarrow$ be the relation defined over views by

$$\forall v', v \in \mathbb{V}^\sharp,\ v' \rightsquigarrow v \iff (\exists \sigma' \in \gamma_{\mathbb{V}}(v'),\ \exists \sigma \in \gamma_{\mathbb{V}}(v),\ \sigma' \rightsquigarrow \sigma).$$

Example 4.3 (Predecessor coverings). In this example and the subsequent one, we discuss the forward execution of the last statement of the program shown in Figure 1 to establish the implications of (1). We use the set of views of Example 3.3.

— Let us consider the predecessors of states that satisfy the view $(\ell_{exit}, v_{\forall \texttt{TRUE}})$:
As the variable b_k is not modified by the assignment at line ℓ_y, such a predecessor state should map b_k to TRUE. Therefore, each singleton of the form below forms a valid predecessor covering:

$$\{(\ell_y, v_{2k})\}.$$

— Let us consider the predecessors of states that satisfy the view $(\ell_{exit}, v_{\exists \texttt{FALSE}})$:
Since the assignment at ℓ_y does not modify any of the Boolean variables b_k, a predecessor covering of $(\ell_{exit}, v_{\exists \texttt{FALSE}})$ should comprise cases where any of those variables is equal to FALSE. Therefore, the following set of views forms a valid predecessor covering:

$$\{(\ell_y, v_{2k+1}) \mid 0 \leq k < N\}.$$

We now discuss step by step the construction of an over-approximation of the transitions in the pure sensitive abstraction. We consider a view v, a predecessor covering V' of v, an element $\overline{s} \in \mathbb{S}^\sharp$, and a state $\sigma \in \gamma_{\mathbb{V}}(v)$. Then

$$\begin{aligned}&\sigma \in \mathbf{post} \circ \gamma_{\mathbb{S}}(\overline{s}) \\ &\iff \exists \sigma' \in \gamma_{\mathbb{S}}(\overline{s}),\ \sigma' \rightsquigarrow \sigma \\ &\iff \exists \sigma' \in \mathbb{S},\ (\forall v' \in \mathbb{V}^\sharp,\ \sigma' \in \gamma_{\mathbb{V}}(v') \Longrightarrow \sigma' \in \overline{s}(v')) \wedge \sigma' \rightsquigarrow \sigma\ .\end{aligned}$$

Since V' is a predecessor covering of v and $\sigma' \rightsquigarrow \sigma$, the property $\sigma' \rightsquigarrow \sigma$ implies that there exists a view $v' \in V'$ such that $\sigma' \in \gamma_{\mathbb{V}}(v')$. Then, in turn, the property $\forall v' \in \mathbb{V}^\sharp,\ \sigma' \in \gamma_{\mathbb{V}}(v') \Longrightarrow \sigma' \in \overline{s}(v')$

implies that $\sigma' \in \overline{s}(v')$. Therefore, we can derive the following:

$$\begin{array}{l}\sigma \in \mathbf{post} \circ \gamma_{\mathbb{S}}(\overline{s}) \\ \Longrightarrow \exists v' \in V',\ \exists \sigma' \in \gamma_{\mathbb{V}}(v'),\ \sigma' \in \overline{s}(v') \wedge \sigma' \rightsquigarrow \sigma \\ \Longrightarrow \exists v' \in V',\ \exists \sigma' \in \gamma_{\mathbb{V}}(v') \cap \overline{s}(v'),\ \sigma' \rightsquigarrow \sigma \\ \Longrightarrow \exists v' \in V',\ \sigma \in \mathbf{post}_{v' \to v}(\overline{s}) \\ \Longrightarrow \sigma \in \bigcup_{v' \in V'} \mathbf{post}_{v' \to v}(\overline{s})\end{array}.$$

The above property holds for all predecessor coverings of v, thus we can even generalize it more:

$$\begin{array}{l}\sigma \in \mathbf{post} \circ \gamma_{\mathbb{S}}(\overline{s}) \\ \Longrightarrow \sigma \in \bigcap_{V' \in P_{\to v}} \left(\bigcup_{v' \in V'} \mathbf{post}_{v' \to v}(\overline{s})\right)\end{array}.$$

This implication was proved under the assumption that $\sigma \in \gamma_{\mathbb{V}}(v)$. Thus, we have proved

$$\forall v \in \mathbb{V}^\sharp,\ \forall \sigma \in \mathbb{S},\ \sigma \in \mathbf{post} \circ \gamma_{\mathbb{S}}(\overline{s}) \wedge \sigma \in \gamma_{\mathbb{V}}(v) \Longrightarrow \sigma \in \bigcap_{V' \in P_{\to v}} \left(\bigcup_{v' \in V'} \mathbf{post}_{v' \to v}(\overline{s})\right).$$

We derive from this the following abstract single step forward execution function.

Theorem 4.4 (Abstract single step forward execution function). *We let* $\overline{\mathbf{post}} : (\mathbb{V}^\sharp \to \mathcal{P}(\mathbb{S})) \to (\mathbb{V}^\sharp \to \mathcal{P}(\mathbb{S}))$ *be defined by*

$$\overline{\mathbf{post}} : \overline{s} \longmapsto \lambda(v \in \mathbb{V}^\sharp) \cdot \bigcap_{V' \in P_{\to v}} \left(\bigcup_{v' \in V'} \mathbf{post}_{v' \to v}(\overline{s})\right).$$

Then,

$$\mathbf{post} \circ \gamma_{\mathbb{S}}(\overline{s}) \subseteq \gamma_{\mathbb{S}} \circ \overline{\mathbf{post}}(\overline{s}).$$

The above formula gives a general solution to over-approximate the concrete computation steps in the abstract, but it does not require the analysis to follow this structure exactly. In particular, the intersection over *all* predecessor coverings is likely to be too costly to compute, and to not always bring a significant payoff in precision: this is not an issue, as a sound alternate solution consists in considering only *some* predecessor coverings; then, the intersection ranges over fewer elements, and thus returns a conservative set of states (note that the empty intersection would return the set of all states, and is always a possible solution). In fact, if we consider two predecessor coverings V_0 and V_1 of a same view such that $V_0 \subseteq V_1$, then the information computed for V_1 can only be worse than that computed for V_0, thus we should discard V_1 for the analysis.

An interesting over-approximation of the abstract function of Theorem 4.4 can be derived using $V' = \mathbb{V}^\sharp$ as a single predecessor covering:

$$\lambda(v \in \mathbb{V}^\sharp) \cdot \bigcup_{v' \in \mathbb{V}^\sharp} \mathbf{post}_{v' \to v}(\overline{s}). \tag{11}$$

As noted above, this formula will generally cause redundant information be taken into account, resulting in a possible loss in precision, depending on the state abstraction and on the program.

Example 4.5 (Forward Execution). We now discuss the forward execution of the last statement of the program shown in Figure 1 using the set of views of Example 3.3. We reuse the predecessor coverings of Example 4.3, which will result in an over-approximation of Theorem 4.4. Before we detail the forward execution computation, we can note that the states at point ℓ_y correspond to each view:

—view (ℓ_y, v_{2k}) corresponds to states where variable b_k is equal to TRUE and $\mathsf{x}_k \geq 1$;
—view (ℓ_y, v_{2k+1}) corresponds to states where variable b_k is equal to FALSE and $\mathsf{x}_k = 0$.

Thus, the abstract pre-condition before this statement is characterized by

$$\overline{s}(\ell_y, v_{2k}) \cap \gamma_{\mathbb{V}}(\ell_y, v_{2k}) = \{\mu \in \mathbb{M} \mid \mu(\mathsf{b}_k) = \texttt{TRUE} \wedge \mu(\mathsf{x}_k) \geq 1\},$$
$$\overline{s}(\ell_y, v_{2k+1}) \cap \gamma_{\mathbb{V}}(\ell_y, v_{2k+1}) = \{\mu \in \mathbb{M} \mid \mu(\mathsf{b}_k) = \texttt{FALSE} \wedge \mu(\mathsf{x}_k) = 0\}.$$

We now consider the views corresponding to the left-hand sides of the implications of Equation (1):

—Let us consider the predecessors of states that satisfy the view $v = (\ell_{exit}, v_{\forall \texttt{TRUE}})$:
We focus on the set of predecessor coverings $\mathcal{P} = \{\{(\ell_y, v_{2k})\} \mid 0 \leq k < N\} \subset P_{\rightarrow \ell_{exit}, v_{\forall \texttt{TRUE}}}$ (many other predecessor coverings could be formed by adding elements to these):

$$\begin{aligned}
&\textstyle\bigcap_{V' \in \mathcal{P}} \left(\bigcup_{v' \in V'} \mathbf{post}_{v' \rightarrow v}(\overline{s})\right)\\
&= \textstyle\bigcap_{k=0}^{N-1} \mathbf{post}_{\ell_y, v_{2k} \rightarrow v}(\overline{s})\\
&= \textstyle\bigcap_{k=0}^{N-1} \gamma_{\mathbb{V}}(v) \cap \mathbf{post}(\overline{s}(\ell_y, v_{2k}) \cap \gamma_{\mathbb{V}}(\ell_y, v_{2k}))\\
&= \textstyle\bigcap_{k=0}^{N-1} \{\mu \in \mathbb{M} \mid \mu(\mathsf{b}_k) = \texttt{TRUE} \wedge \mu(\mathsf{x}_k) \geq 1 \wedge \mu(\mathsf{y}) = \mu(\mathsf{x}_0) \cdot \ldots \cdot \mu(\mathsf{x}_{N-1})\}\\
&= \{\mu \in \mathbb{M} \mid \forall k,\ \mu(\mathsf{b}_k) = \texttt{TRUE} \wedge \mu(\mathsf{x}_k) \geq 1 \wedge \mu(\mathsf{y}) = \mu(\mathsf{x}_0) \cdot \ldots \cdot \mu(\mathsf{x}_{N-1})\}\\
&\subseteq \{\mu \in \mathbb{M} \mid \forall k,\ \mu(\mathsf{b}_k) = \texttt{TRUE} \wedge \mu(\mathsf{x}_k) \geq 1 \wedge \mu(\mathsf{y}) \geq 1\}.
\end{aligned}$$

Thus, the forward execution using the sensitive abstraction shows the first implication of Equation (1).

—Let us consider the predecessors of states that satisfy the view $v = (\ell_{exit}, v_{\exists \texttt{FALSE}})$:
We reuse the predecessor covering displayed in Example 4.3$\{(\ell_y, v_{2k+1}) \mid 0 \leq k < N\}$, and denote it with V'. The union of the transitions over this predecessor covering is

$$\begin{aligned}
&\textstyle\bigcup_{v' \in V'} \mathbf{post}_{v' \rightarrow v}(\overline{s})\\
&= \textstyle\bigcup_{k \in \{0, \ldots, N-1\}} \gamma_{\mathbb{V}}(v) \cap \mathbf{post}(\overline{s}(\ell_y, v_{2k+1}) \cap \gamma_{\mathbb{V}}(\ell_y, v_{2k+1}))\\
&= \textstyle\bigcup_{k=0}^{N-1} \gamma_{\mathbb{V}}(v) \cap \{\mu \in \mathbb{M} \mid \mu(\mathsf{b}_k) = \texttt{FALSE} \wedge \mu(\mathsf{x}_k) = 0 \wedge \mu(\mathsf{y}) = \mu(\mathsf{x}_0) \cdot \ldots \cdot \mu(\mathsf{x}_{N-1})\}\\
&= \textstyle\bigcup_{k=0}^{N-1} \gamma_{\mathbb{V}}(v) \cap \{\mu \in \mathbb{M} \mid \mu(\mathsf{b}_k) = \texttt{FALSE} \wedge \mu(\mathsf{x}_k) = 0 \wedge \mu(\mathsf{y}) = 0\}.
\end{aligned}$$

Thus, the forward execution using the sensitive abstraction shows the second implication of Equation (1).

Derivation of a Sensitive Abstraction in the General Case. We now look beyond the pure sensitive abstraction case and consider situations where $\gamma_{\mathbb{A}}$ is not the identity function (i.e., we are not considering the pure sensitive abstraction anymore). Then, we can derive $\gamma_{\mathbb{S}}$ by composing $\gamma_{\mathbb{A}}$ with the pure sensitive abstraction as remarked in Example 3.10. To achieve this, we simply need to over-approximate one by one all the operators and functions used in the definition of $\overline{\mathbf{post}}$ in Theorem 4.4:

—a sound approximation $\mathbf{join}^\sharp$ of the concrete $\cup$ state set union operator $(\forall a_0^\sharp, a_1^\sharp \in \mathbb{A}^\sharp,\ \gamma_{\mathbb{A}}(a_0^\sharp) \cup \gamma_{\mathbb{A}}(a_1^\sharp) \subseteq \gamma_{\mathbb{A}}(\mathbf{join}^\sharp(a_0^\sharp, a_1^\sharp)))$;
—a sound approximation $\mathbf{meet}^\sharp$ of the concrete $\cap$ state set intersection operator $(\forall a_0^\sharp, a_1^\sharp \in \mathbb{A}^\sharp,\ \gamma_{\mathbb{A}}(a_0^\sharp) \cap \gamma_{\mathbb{A}}(a_1^\sharp) \subseteq \gamma_{\mathbb{A}}(\mathbf{meet}^\sharp(a_0^\sharp, a_1^\sharp)))$;
—a sound approximation $\mathbf{post}^\sharp_{v \rightarrow v'}$ of each $\mathbf{post}_{v \rightarrow v'}$ function (for all $v, v' \in \mathbb{V}^\sharp$), such that $\mathbf{post}_{v \rightarrow v'} \circ \gamma_{\mathbb{A}} \subseteq \gamma_{\mathbb{A}} \circ \mathbf{post}^\sharp_{v \rightarrow v'}$;
—a sound *under*-approximation of predecessor coverings $P^\sharp_{\rightarrow v} \subseteq P_{\rightarrow v}$ for each view.

Using these elements, and following the construction of Theorem 4.4, we can derive a sound abstract single step forward execution function.

Theorem 4.6 (Abstract Single Step Forward Execution). *If we let*

$$\mathbf{post}^\sharp : s^\sharp \longmapsto \lambda(v \in \mathbb{V}^\sharp) \cdot \mathbf{meet}^\sharp \left(\left\{ \mathbf{join}^\sharp \left(\left\{ \mathbf{post}^\sharp_{v' \to v}(s^\sharp(v') \mid v' \in V' \right\} \right) \middle| V' \in P^\sharp_{\to v} \right\} \right),$$

then

$$\mathbf{post} \circ \gamma_{\mathbb{S}} \subseteq \gamma_{\mathbb{S}} \circ \mathbf{post}^\sharp.$$

Note that this function is computable since the set of views was assumed to be finite. Indeed, if $\mathbb{V}^\sharp$ was infinite, the join over all predecessor views of v would not be computable.

We can also observe a similarity between the abstract post-condition operator shown in Theorem 4.4 and the reduction operator of Lemma 3.17: both operators use an intersection over coverings in order to strengthen the abstract values they produce.

The remarks that follow Theorem 4.4 fully apply here as well. In particular, it is in general too costly to compute the intersection over all predecessor coverings, but this is also not always necessary for a sufficient level of precision, thus a strict under-approximation of predecessor coverings should be used.

Possible imprecisions may come from the operators used (approximations of join, meet, and concrete post-conditions), and from the selection of a small set of predecessor coverings. The reduction operator of Lemma 3.17 may be used to enhance the precision of the result, and to mitigate these losses in precision.

Partitioning Case. In Section 3.2, we have observed that the concretization function $\gamma_{\mathbb{S}}$ can be significantly simplified when views define a partitioning. We can actually also significantly simplify the definition of the abstract single step forward execution function. We consider the derivation of the best single step forward abstract execution function, and let $\overline{s}$ be an abstract element of $\mathbb{V}^\sharp \to \mathcal{P}(\mathbb{S})$, σ be a state, and v be a view:

$$\begin{aligned} \mathbf{post} \circ \gamma_{\mathbb{S}}(\overline{s}) &= \{\sigma \in \mathbb{S} \mid \exists \sigma' \in \gamma_{\mathbb{S}}(\overline{s}),\ \sigma' \rightsquigarrow \sigma\} \\ &= \{\sigma \in \mathbb{S} \mid \exists v' \in \mathbb{V}^\sharp,\ \exists \sigma' \in \gamma_{\mathbb{V}}(v') \cap \overline{s}(v'),\ \sigma' \rightsquigarrow \sigma\}. \end{aligned}$$

Since the views abstraction is partitioning, for all state σ, there exists a unique view v such that $\sigma \in \gamma_{\mathbb{V}}(v)$. Therefore,

$$\begin{aligned} \mathbf{post} \circ \gamma_{\mathbb{S}}(\overline{s}) &= \{\sigma \in \mathbb{S} \mid \exists \sigma' \in \gamma_{\mathbb{S}}(\overline{s}),\ \sigma' \rightsquigarrow \sigma\} \\ &= \{\sigma \in \mathbb{S} \mid \exists v', v \in \mathbb{V}^\sharp,\ \exists \sigma' \in \gamma_{\mathbb{V}}(v') \cap \overline{s}(v'),\ \sigma' \rightsquigarrow \sigma \wedge \sigma \in \gamma_{\mathbb{V}}(v)\} \\ &= \{\sigma \in \mathbb{S} \mid \exists v', v \in \mathbb{V}^\sharp,\ \sigma \in \gamma_{\mathbb{V}}(v) \cap \mathbf{post}_{v' \to v}(\overline{s})\} \\ &= \bigcup_{v, v' \in \mathbb{V}^\sharp} (\gamma_{\mathbb{V}}(v) \cap \mathbf{post}_{v' \to v}(\overline{s})) \\ &= \gamma_{\mathbb{S}} \circ \overline{\mathbf{post}}(\overline{s}), \end{aligned}$$

where

$$\overline{\mathbf{post}}(\overline{s}) = \lambda(v \in \mathbb{V}^\sharp) \cdot \bigcup_{v, v' \in \mathbb{V}^\sharp} \mathbf{post}_{v' \to v}(\overline{s}). \tag{12}$$

This result shows that, in the partitioning case, an analysis derived from Equation (11) loses no precision related to the view coverings (although we have seen above that, in the non-partitioning case, it may lose some precision).

Approaches Based on Disjunctions. Analyses based on disjunctive formulas, and that do not have a notion of views to constrain each disjunct, rely on pointwise application of abstract transformers, whereas analyses that use a notion of views but the less tight concretization of Equation (9) also do

not compute meet on abstract transitions like Theorem 4.6 does. As an example, we consider the latter case, and assume $\mathbb{V}^\sharp$ is a covering. Then, the abstract single step forward execution operator is sound:

$$\mathbf{post}^\sharp_\vee : s^\sharp \longmapsto \lambda(v \in \mathbb{V}^\sharp) \cdot \mathbf{join}^\sharp \left(\left\{ \mathbf{post}^\sharp(s^\sharp)(v') \middle| v' \in \mathbb{V}^\sharp, v' \rightsquigarrow v \right\} \right). \tag{13}$$

The first important difference between Equation (13) and the operator of Theorem 4.6 is that the latter constrains both its input and output abstract states with the views, since it uses $\mathbf{post}^\sharp_{v' \to v}$ instead of $\mathbf{post}^\sharp$ ($\mathbf{post}^\sharp_{v' \to v}$ over-approximates operator $\mathbf{post}_{v' \to v} : \overline{s} \longmapsto \gamma_{\mathbb{V}}(v) \cap \mathbf{post}(\overline{s}(v') \cap \gamma_{\mathbb{V}}(v'))$): this is in general more precise, as it ensures the information attached to a view is not "polluted" by states that should be associated to another view. Furthermore, it refines the information attached to view v with several predecessor coverings, which can strengthen the abstract result: this amounts to doing an incremental reduction step.

In the partitioning case, we have shown in Equation (12) another tighter abstract post-condition, which looks closer to Equation (13), since it does not resort to a meet over predecessor coverings. However, it retains the first advantage of Theorem 4.6 mentioned in the previous paragraph, since it also uses $\mathbf{post}^\sharp_{v' \to v}$ instead of $\mathbf{post}^\sharp$.

Another advantage of Theorem 4.6 compared to analyses based on Equation (13) is that it can also be applied when the views abstraction does not define a covering of concrete states, whereas Equation (13) can be used only under the assumption that we are using a covering. Indeed, if $\mathbb{V}^\sharp$ does not define a predecessor covering of v, then there exists no predecessor covering for v, and Theorem 4.6 will return an abstract element that maps v into $\top$.

While this discussion shows that in general Theorem 4.6 provides a more precise analysis algorithm, it does not mean that analyses based on the concretization of Equation (9) are necessarily imprecise. For instance, Rival and Mauborgne (2007) utilizes a notion of "partitioned system," which implies that (1) the views form a covering and (2) that the transitions of the system being analyzed correspond to the transitions defined by $\mathbf{post}_{\_\to\_}$. Therefore, this analysis effectively relies on Equation (13), but achieves the same precision as Equation (12) and in Theorem 4.6, at a lower cost, since it does not consider multiple predecessor coverings. However, compared to this analysis, the operator of Theorem 4.6 is more general.

4.3 Abstract Iterator

As we have observed in Section 4.1, the concrete semantics writes as the least-fixpoint of a function of the single step forward execution, thus we now show how to derive an abstract interpretation of this semantics. To this end, we need to (1) propose a widening operator (i.e., an operator over-approximating concrete join, and ensuring termination of sequences of abstract iterates) to enforce convergence and (2) lift the concrete least-fixpoint into an abstract iterator.

Widening. In this section, we assumed the set of views is finite. As a consequence, the abstract domain $\mathbb{S}^\sharp$ is of finite height if and only if $\mathbb{A}^\sharp$ is, and if so, a widening operator for $\mathbb{S}^\sharp$ can be defined by

$$\forall s^\sharp_0, s^\sharp_1,\ \mathbf{widen}^\sharp_{\mathbb{S}}(s^\sharp_0, s^\sharp_1) = \lambda(v \in \mathbb{V}^\sharp) \cdot \mathbf{join}^\sharp(s^\sharp_0(v), s^\sharp_1(v)).$$

On the other hand, if $\mathbb{S}^\sharp$ does not have a finite height, the same holds for $\mathbb{A}^\sharp$, and a proper widening operator is needed. The pointwise extension of a widening operator $\mathbf{widen}^\sharp : \mathbb{A}^\sharp \times \mathbb{A}^\sharp$ over $\mathbb{A}^\sharp$ then clearly provides an over-approximation of concrete unions and enforces termination.

$$\forall s^\sharp_0, s^\sharp_1,\ \mathbf{widen}^\sharp_{\mathbb{S}}(s^\sharp_0, s^\sharp_1) = \lambda(v \in \mathbb{V}^\sharp) \cdot \mathbf{widen}^\sharp(s^\sharp_0(v), s^\sharp_1(v)).$$

Analysis. Given an over-approximation $s_I^\sharp$ of the set of initial states $\mathbb{S}_I$ (i.e., such that $\mathbb{S}_I \subseteq \gamma_{\mathbb{S}}(s_I^\sharp)$), an over-approximation of $[\![P]\!]$ can be computed as the limit of the following (converging) sequence of abstract iterates (convergence follows directly from the widening property):

$$\begin{aligned} a_0^\sharp &= s_I^\sharp, \\ a_{n+1}^\sharp &= \mathbf{widen}_{\mathbb{S}}^\sharp(a_n^\sharp, \mathbf{post}^\sharp(a_n^\sharp)). \end{aligned} \tag{14}$$

The limit $a_N^\sharp$ of this sequence provides an over-approximation for the concrete semantics:

$$[\![P]\!] \subseteq \gamma_{\mathbb{S}}(a_N^\sharp).$$

Improved iteration strategies (e.g., with unrolling of the first iterations (Blanchet et al. 2003)) can be applied as well.

This analysis can be implemented using a straightforward worklist algorithm in a straightforward manner, where the worklist stores the set of views that have to be re-computed as they have not stabilized yet: whenever the image of a view v' is updated, all the views that accept v' as a predecessor should in turn be added to the worklist (namely, if there exists a view v such that there may exist $\sigma' \in \gamma_{\mathbb{V}}(v'), \sigma \in \gamma_{\mathbb{V}}(v)$, such that $\sigma' \rightsquigarrow \sigma$, then the view-to-view transition $v' \rightsquigarrow v$ should be added to the worklist).

One additional caveat is that Equation (14) turns all views into a widening point, which is not optimal in general. Instead, it is sufficient to perform widening only on one view per cycle in the "predecessor" relation over views defined in the previous paragraph. In the case of flow sensitivity, this boils down to the (very usual) selection of at least one program location per cycle in the dependence graph.

5 DYNAMIC SENSITIVITY

The previous section described a sensitive analysis, where views are *static*, which means that the views are fixed beforehand and do not change over the course of the analysis. We now study *dynamic* sensitivity, which is the case where views may change over the course of the analysis.

5.1 Dynamicity in Program Analysis

Need for a Dynamic Approach. Given a program and a property of interest the inference of which requires some sensitive analysis be performed, it is generally preferable not to introduce too many views, as this is likely to make the analysis slower and more memory intensive. Conversely, not using enough views would result in a failed attempt at proving the property of interest. Therefore, selecting an appropriate set of views for the analysis is not a trivial task. In general, there exists no unique appropriate set of views as shown in Section 2. However, that example also demonstrates that some sets of views result in more expensive analyses than others. Therefore, it is quite reasonable to expect that a standard, pre-defined set of views will not let the analysis compute invariants that are sufficiently precise for the verification of an arbitrary program.

While a solution could be to manually refine the set of views and re-launch the analysis (as many automatic static analysis tools can also be manually tuned for increased precision), this approach is often not satisfactory since (1) it requires several runs of the analysis, which is more time consuming, and (2) it puts an additional and tedious burden on the user.

An alternative approach lets the analysis compute the set of views at the same time as the invariants. This technique is called *dynamic sensitive analysis*. It was initially proposed in the context of partitioning analyses by Bourdoncle (Bourdoncle 1992) and it has also been developed for trace partitioning by Rival and Mauborgne (Rival and Mauborgne 2007). Other applications include dynamic partitioning of array areas in array analyses (Cousot et al. 2011) and analysis using predicate

domains (Henzinger et al. 2002). Furthermore, such techniques have been implemented in several state-of-the-art static analysis tools, as discussed in Section 7. In essence, this approach aims at inferring complex invariants without user supplied annotations and repeated static analysis runs.

This dynamic approach is actually useful not only in the case where the set of possible views is infinite, but also when that set is possibly large, and when representing all views at all times could hinder efficiency. While a static approach would carry all views in $\mathbb{V}^\sharp$ from the beginning of the analysis, a dynamic analysis may start with only a subset of $\mathbb{V}^\sharp$ and extend this set when the needs arise.

Example 5.1 (Dynamic Sensitivity). We consider the example program of Figure 1 in Section 2. Example 3.2 describes the set of views $\mathbb{V}^\sharp = \{v_0, \ldots, v_{2N-1}, v_{\forall\texttt{TRUE}}, v_{\exists\texttt{FALSE}}\}$, which defines a sensitive analysis that carries all these views. A *dynamic* sensitive analysis would add a special view $v_\top$ that describes any possible behavior:

$$\begin{aligned}\gamma_{\mathbb{V}}(v_\top) &= \mathbb{E},\\ \mathbb{V}^\sharp &= \{v_\top, v_0, \ldots, v_{2N-1}, v_{\forall\texttt{TRUE}}, v_{\exists\texttt{FALSE}}\}.\end{aligned}$$

Then, a forward abstract interpretation of the program would start with the abstract state that contains a single view:

$$s^\sharp : v_\top \longmapsto \top.$$

Furthermore, as the dynamic analysis progresses and discovers information about the Boolean variables, it introduces views $v_0, \ldots, v_{2N-1}$.

Design of a Dynamic Sensitive Analysis. Effectively adjusting the set of views during the analysis is a nontrivial operation that requires a sound (possibly conservative) *conversion* of abstract values defined over a set of views into abstract values defined over a different set of views. This conversion operation should transform an abstract element $s_0^\sharp$ of the sensitive abstract domain with a set of views V_0 into another abstract element $s_1^\sharp$ with a set of views V_1 (which may contain views not present in V_0, and which may not contain all views in V_0), which should over-approximate all the states described by $s_0^\sharp$. If this conversion would lose too much precision or incur a prohibitive computational cost, the dynamic sensitive analysis would not be worthwhile. Thus, we study this operation in Section 5.2.

Moreover, the dynamic sensitive analysis should also preserve termination. This implies that the views modifying process should be terminating for any analysis run. Therefore, the dynamic sensitive analysis requires the definition of a *widening* operator over abstract elements that comprise the sets of views, which we discuss in Section 5.3.

5.2 Conversion of Sets of Views

In this section, we work in the framework set up in Definition 3.6. We assume sets of views $\mathbb{V}_0^\sharp$ and $\mathbb{V}_1^\sharp$, where $\mathbb{V}_0^\sharp$ denotes the "initial set of views," whereas $\mathbb{V}_1^\sharp$ denotes the "target set of views." As there is no real ambiguity, we note $\gamma_{\mathbb{V}}(v_0)$ ($\gamma_{\mathbb{V}}(v_1)$, respectively) for the concretization of a view $v_0 \in \mathbb{V}_0^\sharp$ ($v_1 \in \mathbb{V}_1^\sharp$, respectively). We consider an abstract element $s_0^\sharp \in \mathbb{V}_0^\sharp \to \mathbb{A}^\sharp$ and seek for an abstract element $s_1^\sharp \in \mathbb{V}_1^\sharp \to \mathbb{A}^\sharp$ such that $s_1^\sharp$ over-approximates $s_0^\sharp$, i.e., $\gamma_{\mathbb{S},0}(s_0^\sharp) \subseteq \gamma_{\mathbb{S},1}(s_1^\sharp)$ where $\gamma_{\mathbb{S},0}$ ($\gamma_{\mathbb{S},1}$, respectively) denotes the concretization function associated to $\mathbb{V}_0^\sharp \to \mathbb{A}^\sharp$ ($\mathbb{V}_1^\sharp \to \mathbb{A}^\sharp$, respectively). Intuitively, this conversion amounts to the computation of a post-condition for the concrete operation "identity," and replacing the set of views $\mathbb{V}_0^\sharp$ with the set of views $\mathbb{V}_1^\sharp$. Therefore, intuition can be drawn from the abstract post-condition function shown in Theorem 4.4: given a view v_1, $s_1^\sharp(v_1)$ should over-approximate the images by $s_0^\sharp$ of any set of views in $\mathbb{V}_0^\sharp$

that cover v_1. Besides, in this process, the images $s_i^\sharp(v_i)$ can be further refined so as to account only for the states that do satisfy v_i, following the internal reduction principle set up in Section 3.3.

To this end, we define the following notion of *covering across views*.

Definition 5.2 (Covering across views). Let $v_1 \in \mathbb{V}_1^\sharp$ be a view in the target set of views. A $\mathbb{V}_0^\sharp$-*covering* of v_1 is a set of views $V_0 \subseteq \mathbb{V}_0^\sharp$ such that

$$\gamma_{\mathbb{V}}(v_1) \subseteq \bigcup_{v_0 \in V_0} \gamma_{\mathbb{V}}(v_0).$$

We write $\mathbf{Covers}_{\mathbb{V}_0^\sharp}(v_1)$ for the set of $\mathbb{V}_0^\sharp$-coverings of v_1.

In general, the analysis may use an approximation of the set of $\mathbb{V}_0^\sharp$-coverings, therefore we let $\mathbf{Covers}^\sharp_{\mathbb{V}_0^\sharp}(v_1)$ denote a subset of $\mathbf{Covers}_{\mathbb{V}_0^\sharp}(v_1)$.

Using the above notations, we can now provide a general operation to convert across views, using the view restriction function introduced in Definition 3.18.

Theorem 5.3 (Views Conversion Operator). *We define* $\mathbf{Convert}_{\mathbb{V}_1^\sharp \to \mathbb{V}_0^\sharp}$ *by*

$$\begin{array}{l}\forall v_1 \in \mathbb{V}_1^\sharp,\\ \quad \mathbf{Convert}_{\mathbb{V}_1^\sharp \to \mathbb{V}_0^\sharp}(s_0^\sharp)(v_1) =\\ \quad\quad \mathbf{meet}^\sharp\left(\left\{\mathbf{join}^\sharp\left(\left\{\mathbf{restrict}^\sharp(v_1, \mathbf{restrict}^\sharp(v_0, s_0^\sharp(v_0)))\middle| v_0 \in V_0\right\}\right)\middle| V_0 \in \mathbf{Covers}^\sharp_{\mathbb{V}_0^\sharp}(v_1)\right\}\right).\end{array}$$

Then,

$$\gamma_{\mathbb{S},0}(s_0^\sharp) \subseteq \gamma_{\mathbb{S},1}(\mathbf{Convert}_{\mathbb{V}_1^\sharp \to \mathbb{V}_0^\sharp}(s_1^\sharp)).$$

Let $\sigma \in \gamma_{\mathbb{S},0}(s_0^\sharp)$, and let us prove that $\sigma \in \gamma_{\mathbb{S},1}(\mathbf{Convert}_{\mathbb{V}_1^\sharp \to \mathbb{V}_0^\sharp}(s_1^\sharp))$. To do that, we let $v_1 \in \mathbb{V}_1^\sharp$, assume that $\sigma \in \gamma_{\mathbb{V}}(v_1)$, and prove that

$$\sigma \in \gamma_{\mathbb{A}}(\mathbf{Convert}_{\mathbb{V}_1^\sharp \to \mathbb{V}_0^\sharp}(s_1^\sharp)(v_1)).$$

Given the definition of the conversion operator, we assume that $V_0 \in \mathbf{Covers}^\sharp_{\mathbb{V}_0^\sharp}(v_1)$, and show that

$$\sigma \in \gamma_{\mathbb{A}}\left(\mathbf{join}^\sharp\left(\left\{\mathbf{restrict}^\sharp(v_1, \mathbf{restrict}^\sharp(v_0, s_0^\sharp(v_0)))\middle| v_0 \in V_0\right\}\right)\right). \tag{15}$$

By the definition of $\mathbf{Covers}^\sharp_{\mathbb{V}_0^\sharp}(v_1)$ and since $\sigma \in \gamma_{\mathbb{V}}(v_1)$, there exists $v_0 \in V_0$ such that $\sigma \in \gamma_{\mathbb{V}}(v_0)$. Therefore, we have $\sigma \in \gamma_{\mathbb{V}}(v_0)$, $\sigma \in \gamma_{\mathbb{V}}(v_1)$, and $\sigma \in \gamma_{\mathbb{A}}(s_0^\sharp(v_0))$. As a consequence, Equation (15) holds, and this concludes the proof of the soundness of $\mathbf{Convert}_{\mathbb{V}_1^\sharp \to \mathbb{V}_0^\sharp}$.

In the next paragraphs, we instantiate Theorem 5.3 to a series of common ways to modify the set of views, by adding/removing or splitting/merging existing views. While Theorem 5.3 and those instances all provide sound ways to convert views, they may not be the most precise or efficient ones in general, and other ways to dynamically change the set of views may be implemented.

Addition of New Views. The *addition* of a set of views happens when $\mathbb{V}_1^\sharp \supset \mathbb{V}_0^\sharp$. It should be performed whenever some views not present in the initial set of views appear to be required so as

to strengthen the precision of abstract states. In this case, an appropriate abstract covering across views can be defined by

$$\forall v_1 \in \mathbb{V}_1^\sharp,\ \mathbf{Covers}^\sharp_{\mathbb{V}_0^\sharp}(v_1) = \begin{cases} \{\{v_1\}\} & \text{if } v_1 \in \mathbb{V}_0^\sharp \\ V & \text{where } V \subseteq \mathcal{P}(\mathbb{V}_0^\sharp) \text{ covers } v_1, \text{ otherwise.} \end{cases}$$

The conversion operator obtained by applying the construction of Theorem 5.3 will synthesize information for the new views.

Example 5.4 (Addition of views). We follow up on Example 5.1. When the forward analysis of the program of Figure 1 encounters an **if**-statement, it infers information about a Boolean variable b_i, which means it is a good point to add views v_{2i} and v_{2i+1} (which, respectively, describe states where b_i is true or false). The addition of these views follows the above algorithm, and the analysis synthesizes precise information for v_{2i} and v_{2i+1} using the condition tests.

Removal of Views. The *removal* of a set of views corresponds to the case where $\mathbb{V}_1^\sharp \subset \mathbb{V}_0^\sharp$. It should be used when some views become irrelevant. Typically, they have been useful for the beginning of the analysis and will not be used anymore onwards, so that they can be discarded so as to make the representation of abstract states more compact and cheap to compute on. In this case, an appropriate abstract covering across views can be defined by

$$\forall v_1 \in \mathbb{V}_1^\sharp,\ \mathbf{Covers}^\sharp_{\mathbb{V}_0^\sharp}(v_1) = \{\{v_1\}\}.$$

This essentially amounts to throwing away the information about views that belong to $\mathbb{V}_0^\sharp \setminus \mathbb{V}_1^\sharp$. Before that information is removed, the reduction operator introduced in Lemma 3.17 may be applied so as to preserve the information enclosed in the abstract state.

Splitting of Views. While the addition operation preserves all existing views and inserts novel views, it is often more intuitive to partition existing views into more precise ones. This is especially natural when the analysis should postpone the computation of some joins in the abstract, e.g., when analyzing loops or other conditional branching control structures. An extreme case is when the analysis starts with a single view (that abstracts any state), and refines this view each time it infers that an abstract join is better avoided. We call this operation a *splitting of views*. It is defined by a surjective function $h : \mathbb{V}_1^\sharp \to \mathbb{V}_0^\sharp$ such that

$$\forall v_0 \in \mathbb{V}_0^\sharp,\ \gamma_{\mathbb{V}}(v_0) = \bigcup \left\{ \gamma_{\mathbb{V}}(v_1) \middle| v_1 \in \mathbb{V}_1^\sharp \wedge h(v_1) = v_0 \right\}.$$

For more insight, we describe fully the resulting conversion operation when a given view v_s is split into two views v_s' and v_s'', that is, $\mathbb{V}_1^\sharp = \mathbb{V}_0^\sharp \setminus \{v_s\} \uplus \{v_s', v_s''\}$ where $h(v_s') = h(v_s'') = v_s$. Then, the operation $\mathbf{Convert}_{\mathbb{V}_1^\sharp \to \mathbb{V}_0^\sharp}$ maps $s_0^\sharp$ into $s_1^\sharp$ where

$$\begin{aligned} s_1^\sharp(v_s') &= \mathbf{restrict}^\sharp(v_s', \mathbf{restrict}^\sharp(v_s, s_0^\sharp(v_s))), \\ s_1^\sharp(v_s'') &= \mathbf{restrict}^\sharp(v_s'', \mathbf{restrict}^\sharp(v_s, s_0^\sharp(v_s))), \\ s_1^\sharp(v_1) &= s_0^\sharp(v_1) \qquad\qquad \text{if } v_1 \notin \{v_s', v_s''\}. \end{aligned}$$

This formula generalizes to more complex splittings while retaining the same basic principle: the abstract information attached to the views that are split is refined to the view partitions.

While some existing analyses always split a view into a set of disjoint views, the splitting defined by function h is more general, and it also allows one view to be replaced with a set of views that cover it.

Merging of Views. The dual operation corresponding to the splitting is the *merging of views*, and aims at collapsing together views that were split for a local improvement in precision. Intuitively, when the analysis benefits from more views in a *local* manner only, the merging should occur as soon as the views generated by splitting are not useful anymore. For instance, in the case of a conditional structure where views have been split, the analysis should merge them at the point where the effect of the condition is not immediately relevant anymore, and before the additional partitions become an unnecessary burden to the analysis. The merging is defined by a function $h : \mathbb{V}_0^\sharp \to \mathbb{V}_1^\sharp$ such that

$$\forall v_0 \in \mathbb{V}_0^\sharp,\ \gamma_{\mathbb{V}}(v_0) \subseteq \gamma_{\mathbb{V}}(h(v_0)).$$

For more insight, we describe fully the resulting conversion operation when a given pair of views v_s' and v_s'' are merged into a view v_s, that is, $\mathbb{V}_1^\sharp = \mathbb{V}_0^\sharp \uplus \{v_s\} \setminus \{v_s', v_s''\}$ where $h(v_s') = h(v_s'') = v_s$. Then, the operation $\mathbf{Convert}_{\mathbb{V}_1^\sharp \to \mathbb{V}_0^\sharp}$ maps $s_0^\sharp$ into $s_1^\sharp$ where

$$s_1^\sharp(v_s) = \mathbf{restrict}^\sharp(v_s, \mathbf{join}^\sharp(\mathbf{restrict}^\sharp(v_s', s_0^\sharp(v_s')), \mathbf{restrict}^\sharp(v_s'', s_0^\sharp(v_s'')))),$$
$$s_1^\sharp(v_1) = s_0^\sharp(v_1) \quad \text{if } v_1 \neq v_s.$$

5.3 Widening Over Sets of Views

As noted earlier in this section, the second issue raised by dynamic sensitive analyses is termination. When views are static, the widening introduced in Section 4.3 consists of a direct pointwise application of the widening of underlying abstract domain $\mathbb{A}^\sharp$ to the image of each view. However, this solution does not provide a widening when the set of views is dynamic and evolves during the analysis: indeed, the dynamic approach also needs to enforce the convergence of the underlying set of views, otherwise the view conversion/underlying domain widening process will not terminate.

Intuitively, and to enforce convergence, the dynamic sensitive analysis should first enforce convergence of the set of views, and then enforce convergence in the underlying domain, as in Section 4.3. A general way to achieve this is to resort to a *cofibered abstract domain* structure (Venet 1996), where

— sets of views are elements of a lattice called the *basis*; and
— an abstract state is given by an element $\mathbb{V}^\sharp$ of the basis and an abstract element $s^\sharp$ defined over $\mathbb{V}^\sharp$.

Furthermore, the basis is required to provide a widening, which guarantees the termination of the process of converting sets of views. This approach has been implemented in Rival and Mauborgne (2007) by bounding the numbers of split views introduced for each loop, which guarantees the number of views introduced by the dynamic sensitive analysis of a given program is bounded, even if views are selected among an infinite set.

6 CONSTRUCTING SENSITIVITIES FOR PROGRAM ANALYSIS

In this section, we define several ways to construct instances of sensitive abstract domains for the static analysis of programs. This includes basic instances of sensitive abstract domains (including some that are not usually viewed as sensitive abstractions), and techniques to compose several forms of sensitive abstraction. Therefore, this section demonstrates the generality of the sensitive

abstraction framework to describe not only families of properties that are used in static analysis, but also ways to combine sensitivities so as to enhance analysis precision and efficiency. For each construction, we simply need to define

— the kind of behaviors that are described by views (e.g., states or execution traces), the set of views, and their concretization (Definition 3.1);
— the view-to-view transition function (Definition 4.1) and the corresponding predecessor covering (Definition 4.2).

Indeed, in general, these fully define the abstract interpretation of programs. Throughout this section, we use the notations introduced in Section 3 and Section 4. Moreover, we let $\mathbb{T}$ denote the set of finite execution traces over a set of states $\mathbb{S}$; we write $\langle \sigma_0, \ldots, \sigma_n \rangle$ for the execution trace composed of states $\sigma_0, \ldots, \sigma_n$ in that order; given two traces $\tau, \tau' \in \mathbb{T}$, we write $\tau \cdot \tau'$ for their concatenation.

6.1 Flow Sensitivity

We first consider *flow sensitivity*, which is a standard static analysis technique that partitions states depending on the control states they correspond to. For instance, we employed it throughout Section 2, as we were describing local invariants for each control state, and we also discussed it in Example 3.3.

We write $\mathbb{L}$ for the set of *control states*. Moreover, for each program state $\sigma \in \mathbb{S}$, we let $\mathcal{L}(\sigma)$ denote the *control state* of σ. A state is often defined as a tuple, one of the components of which is the control state, so that the definition of the function $\mathcal{L} : \mathbb{S} \longrightarrow \mathbb{L}$ is usually trivial.

We can now define the components of the *flow-sensitive abstraction* that partitions states using their control states before applying another abstraction (usually on sets of stores):

— Each view describes a set of states, the set of views is the set of control states, and the concretization of a view ℓ comprises all the states at the control state ℓ:

$$\begin{aligned} \mathbb{C} &= \mathcal{P}(\mathbb{S}), \\ \mathbb{V}^\sharp &= \mathbb{L}, \\ \gamma_{\mathbb{V}}(\ell) &= \{\sigma \in \mathbb{S} \mid \mathcal{L}(\sigma) = \ell\}. \end{aligned}$$

  Thus, abstract states map control states into the elements of the abstract domain $\mathbb{A}^\sharp$.
— Given views ℓ and ℓ', the view to view transition $\mathbf{post}_{\ell' \to \ell}$ captures exactly the program steps that move from the control state ℓ' to the control state ℓ, and thus, the predecessor covering of a view ℓ comprises the set of predecessors of ℓ in the control flow graph.

The corresponding abstract functions simply implement the well-known flow-sensitive analysis over a control flow graph (Cousot 1981).

Furthermore, coarser abstractions of control states that group several control states into a single abstract label can be used. The flow-insensitive abstraction is derived when using a single abstract label that describes all states. These can be formalized into our framework like the flow-sensitive abstraction.

6.2 Call String Sensitivity

Another common form of sensitivity is *context sensitivity*, where the analysis groups states by *calling contexts* before abstracting them (usually using some store abstraction). For instance, Sharir and Pnueli (1981) uses *call strings* in order to characterize program states. Context-sensitive abstractions may use both function names and call sites to describe a chain of function calls, or only function names. Moreover, we can distinguish *fully context-sensitive* analyses that use the full call

chain, and *partially context-sensitive* analyses that use only a part of the calling context to achieve that (typically, at most k inner calls, where k is a fixed integer). In general, our framework can describe all these forms of sensitivity, thus we simply show the description of two of them.

We let $\mathbb{K}$ denote the set of call strings. A call string $\mathcal{k} \in \mathbb{K}$ describes a full calling context including the full list of ongoing function calls (the outermost call is in the end of the call string, whereas the current call is in the beginning); each call is represented by its name and its call site. Thus, such an element $\mathcal{k}$ consists of a sequence $(\ell_0, f_0) \cdot \ldots \cdot (\ell_n, f_n)$. Moreover, we let $\mathcal{K}(\sigma)$ denote the calling context of the state σ.

Full context sensitivity. Fully context-sensitive abstractions use full call strings, and rely on the following sets of views and view coverings:

—Views are call strings, and also abstract sets of states, thus,

$$\begin{aligned} \mathbb{C} &= \mathcal{P}(\mathbb{S}), \\ \mathbb{V}^\sharp &= \mathbb{K}, \\ \gamma_{\mathbb{V}}(\mathcal{k}) &= \{\sigma \in \mathbb{S} \mid \mathcal{K}(\sigma) = \mathcal{k}\}. \end{aligned}$$

Thus, abstract states are functions from calling contexts into $\mathbb{A}^\sharp$.

—The view to view transition relation describes three kinds of transitions:

(1) intra-procedural execution steps are accounted by transitions of the form $\mathbf{post}_{\mathcal{k} \to \mathcal{k}}$ for some $\mathcal{k} \in \mathbb{K}$ (such transitions are accounted by the intra-procedural semantics);
(2) function call execution steps are accounted by transitions of the form $\mathbf{post}_{\mathcal{k} \to ((\ell, f) \cdot \mathcal{k})}$ for some $\mathcal{k} \in \mathbb{K}$, $\ell \in \mathbb{L}$ and the function name f;
(3) function return execution steps are accounted by transitions of the form $\mathbf{post}_{((\ell, f) \cdot \mathcal{k}) \to \mathcal{k}}$ for some $\mathcal{k} \in \mathbb{K}$, $\ell \in \mathbb{L}$ and the function name f.

Partial Context Sensitivity. As fully context-sensitive analyses are often too expensive, or not even practical in the presence of recursive functions (due to unbounded size call strings), *partially context-sensitive abstractions* use all call strings of length at most k (where k is a fixed integer value, chosen before the analysis is run) and use call string prefixes. We let $\mathbb{K}_k$ denote this set of call strings, and we write $\mathcal{k}_{|k}$ for the prefix of length at most k of the call string $\mathcal{k}$. Then, partially context-sensitive abstractions are defined by the following:

—Views are call strings, thus,

$$\begin{aligned} \mathbb{C} &= \mathcal{P}(\mathbb{S}), \\ \mathbb{V}^\sharp &= \mathbb{K}_k, \\ \gamma_{\mathbb{V}}(\mathcal{k}) &= \{\sigma \in \mathbb{S} \mid (\mathcal{K}(\sigma))_{|k} = \mathcal{k}\}. \end{aligned}$$

—The view to view transition relation should be defined for the same three cases as above, yet the transitions related to function calls and function returns differ slightly, due to the restriction on prefixes of length at most k. This restriction is the source of possible imprecision because the call at depth k cannot be precisely restored after a call at depth $k + 1$ returns.

Our framework also describes abstractions based on call strings without call site information. Moreover, it can also model cases where calling context information is abstracted using a different lattice that does not necessarily define a partition of $\mathbb{K}$.

6.3 Product of Sensitivities

The analysis of interprocedural programs often resorts both to flow sensitivity (Section 6.1) and to some kind of context sensitivity (Section 6.2), which corresponds to another notion of sensitivity that extends these two previous forms. Fortunately, our framework also allows one to define *generic* methods to combine several sensitivities, and to produce new forms of sensitivity.

We assume that two forms of sensitivities describing the same kind of behaviors $\mathbb{C}$ are defined by sets of views $\mathbb{V}_0^\sharp$ and $\mathbb{V}_1^\sharp$, and their concretization functions $\gamma_{\mathbb{V}_i} : \mathbb{V}_i^\sharp \longrightarrow \mathbb{C}$ (for $i \in \{0, 1\}$). The *product* sensitivity is defined straightforwardly by

$$\mathbb{V}_\times^\sharp = \mathbb{V}_0^\sharp \times \mathbb{V}_1^\sharp,$$
$$\forall (v_0, v_1) \in \mathbb{V}_0^\sharp \times \mathbb{V}_1^\sharp,\ \gamma_{\mathbb{V}_\times}(v_0, v_1) = \gamma_{\mathbb{V}_0}(v_0) \cap \gamma_{\mathbb{V}_1}(v_1).$$

Essentially, a *product view* (v_0, v_1) characterizes exactly the program behaviors that are characterized by both v_0 in $\mathbb{V}_0^\sharp$ and v_1 in $\mathbb{V}_1^\sharp$, thus it amounts to a conjunction of views. Abstract states can be represented either as functions from $\mathbb{V}_0^\sharp \times \mathbb{V}_1^\sharp$ into $\mathbb{A}^\sharp$ or as functions from $\mathbb{V}_0^\sharp$ into $\mathbb{V}_1^\sharp \longrightarrow \mathbb{A}^\sharp$.

This form of composed sensitivities adequately describes the composition of context sensitivity and flow sensitivity, where the analysis is sensitive to both the calling context and the program point. Indeed, to achieve this, we let $\mathbb{V}_0^\sharp$ be the context sensitivity abstraction and $\mathbb{V}_1^\sharp$ be the flow-sensitivity abstraction and apply the product construction. Then abstract states are functions mapping calling contexts into functions from control states into $\mathbb{A}^\sharp$. This product instance also demonstrates that in general, not all pairs of views are useful in the product covering: indeed, when a view v_0 describes a calling context where the current function is f and a view v_1 describes a control state that is not located inside the function f, then the view (v_0, v_1) describes no states at all. Therefore, such irrelevant product views should not be included in abstract states.

The definition of abstract single step forward execution functions for the product of two sensitivities requires the choice of a set of predecessor coverings for each relevant product view (v_0, v_1). As observed in Theorem 4.6 (Section 4.2), any under-approximation of the set of predecessor coverings may be used, so if $P^\sharp_{\to v_0}$ ($P^\sharp_{\to v_1}$, respectively) is an under-approximation of the predecessor covering of v_0 (of v_1, respectively), then we can define an under-approximation of the predecessor covering for (v_0, v_1) by

$$P^\sharp_{\to (v_0, v_1)} = \{V_0 \times V_1 \mid V_0 \in P^\sharp_{\to v_0} \wedge V_1 \in P^\sharp_{\to v_1}\}.$$

For the same reason as not all product views are relevant, some of these predecessor coverings are not relevant and should be pruned out. Intuitively, context- and flow-sensitive analyses will restrict to calling transitions where call sites, caller contexts, and callee contexts are coherent.

To summarize, products of sensitive analyses yield increased precision both due to the finer grained views (which lead to applying the states abstraction defined by $\mathbb{A}^\sharp$ and $\gamma_{\mathbb{A}}$ to smaller sets of states) and to the finer grained treatment of program transitions in the computation of abstract single step forward executions.

6.4 Value Sensitivity

In Section 6.1 and Section 6.2, we have discussed sensitive abstractions based on program control states (flow or context). Other abstractions based on *values* may be used as well, so as to group states.

The most obvious case is when the value of a variable is used. For instance, if a variable b has type **bool**, the concrete states can be divided into two sets of states, where b has value `TRUE` (`FALSE`, respectively). This case induces the following sensitive analysis:

—Each view describes a set of states, the set of views is the set of Boolean values, and the concretization of a view b comprises all the states where b has value b (we let $\sigma(\mathsf{b})$ denote the value of the variable b in the state σ):

$$\begin{aligned}\mathbb{C} &= \mathcal{P}(\mathbb{S}),\\ \mathbb{V}^\sharp &= \{\mathtt{TRUE}, \mathtt{FALSE}\},\\ \gamma_{\mathbb{V}}(b) &= \{\sigma \in \mathbb{S} \mid \sigma(\mathsf{b}) = b\}.\end{aligned}$$

—Given views b and b', the view to view transition $\mathbf{post}_{b' \to b}$ captures exactly the program steps that move from a state where b stores the value b' into a state where b stores the value b.

This form of value sensitivity is usually combined by the sensitivity product (Section 6.3) with flow or context sensitivity. Moreover, it generalizes easily to the case where views account for the values of several Boolean variables. In particular, this sensitivity was used in Section 2 in order to analyze precisely the example code of Figure 1 as shown in Equation (2).

Value sensitivity generalizes in many ways. First, it may be applied to scalar values, when these range over a finite (small) set of values, or using the dynamic approach, with widening, that was introduced in Section 5. Second, the set of views could be defined using a coarser division of the values of the variables that the sensitivity is based on: in Section 2, the results presented in Equation (3), Equation (4), and Equation (5) select a coarse partition or covering of the values of the Boolean variables used in the program. Note that it is not necessary to use a view concretization $\gamma_{\mathbb{V}}$ that partitions concrete states, and that a $\gamma_{\mathbb{V}}$ that induces a covering is also doable as is the case for Equation (5). Last, other families of scalar predicates such as inequalities could be used instead as in Jeannet (2003).

6.5 Trace History Sensitivity

The sensitive analyses described in the previous subsections all rely on views that abstract sets of *states*. Using a sensitivity based on *traces* allows improved expressiveness because any view abstracting states can also be viewed as an abstraction of the traces; however, the opposite is not true, in general, since traces contain strictly more information than their last states. As an example, trace partitioning based on the history of execution traces (Handjieva and Tzolovski 1998; Rival and Mauborgne 2007) groups concrete states depending on an abstraction of the execution traces that lead to them.

A classical form of history-based sensitivity discriminates execution traces based on the branch taken for a given **if** statement. Thus, we formalize this case here. We assume the program contains an **if** statement at a control state ℓ, and we write ℓ_t (ℓ_f, respectively) for the first point of the true branch (the false branch, respectively) of that condition:

—Each view describes a set of execution traces; moreover, views characterize three sets of traces, which respectively, (1) went through the true branch of the condition, (2) went through the false branch of the condition, and (3) did not enter the condition; therefore, the view concretization formalizes as follows:

$$\begin{aligned}\mathbb{C} &= \mathcal{P}(\mathbb{T}),\\ \mathbb{V}^\sharp &= \{\ell_t, \ell_f, \ell_\odot\},\\ \gamma_{\mathbb{V}}(\ell_t) &= \{\langle\sigma_0, \ldots, \sigma_n\rangle \in \mathbb{T} \mid \exists i,\ \mathcal{L}(\sigma_i) = \ell \wedge \mathcal{L}(\sigma_{i+1}) = \ell_t\},\\ \gamma_{\mathbb{V}}(\ell_f) &= \{\langle\sigma_0, \ldots, \sigma_n\rangle \in \mathbb{T} \mid \exists i,\ \mathcal{L}(\sigma_i) = \ell \wedge \mathcal{L}(\sigma_{i+1}) = \ell_f\},\\ \gamma_{\mathbb{V}}(\ell_\odot) &= \{\langle\sigma_0, \ldots, \sigma_n\rangle \in \mathbb{T} \mid \forall i,\ \mathcal{L}(\sigma_i) \notin \{\ell_t, \ell_f\}\}.\end{aligned}$$

—The view to view transitions derive straightforwardly from the definition of $\gamma_{\mathbb{V}}$; in particular, a transition from $\ell_{\odot}$ to ℓ_t corresponds to the entry into the true branch of the condition for the first time.

As an example, we have observed in Section 2 that the abstract state described by Equation (5) can also be derived using views that denote execution paths, which corresponds exactly to this form of trace history sensitivity.

We note that, if the program contains loops or gotos, a given execution trace may belong to both $\gamma_{\mathbb{V}}(\ell_t)$ and $\gamma_{\mathbb{V}}(\ell_f)$, thus this set of views induces a covering (and not a partitioning) of the program executions. An alternative approach would characterize the *last* condition statement entry only; it would induce a partitioning. This set of views never merges sets of execution traces as in Handjieva and Tzolovski (1998). By contrast, Rival and Mauborgne (2007) selects *merging points* to avoid accumulating too many unnecessary views.

Other kinds of trace history sensitivity use views that describe traces depending on the number of iterations spent in a given loop or on an abstraction of that number (e.g., considering iteration parity), the value of a variable at a given point in the execution, and others. In particular, flow sensitivity (Section 6.1) and context sensitivity (Section 6.2) can also be viewed as trace history sensitivity. In the case where the criterion is based on the value of a variable x, the major difference with value sensitivity (Section 6.4) lies in the fact that trace history adjusts the view to state relation only at given program points, whereas value sensitivity modifies it whenever the value of x changes. Thus, trace history sensitivity captures a different property and may be more economical.

6.6 Call-Site Sensitivity

In Section 6.2, we have described a few techniques that use an abstraction of the call stack as views in order to tackle function calls. We now show that other techniques to deal with function calls, such as analyses computing function summaries, and analyses that track relations with function parameter values, can be described as sensitive analyses as well.

Abstract Call State Sensitivity and Application to Functional (Summary-Based) Analyses. Among the variants of sensitivities used for the analysis of programs with procedures, we can cite the *functional approach* (Sharir and Pnueli 1981). While the call-string sensitivity uses call-site strings for call contexts, the functional approach uses input abstract call states as contexts (ultimately, this allows one to abstract a function using pairs of abstract states that respectively describe input and output states). As such, it also fits into our framework, and can be described using a view abstraction based on abstract call states: indeed, in this approach, a view $v \in \mathbb{V}^\sharp$ stands for an abstraction of a set of *input* states into a function. Thus, to account for this form of sensitivity, we should let concrete behaviors be pairs (τ, σ) where τ is the stack of entry states for all the ongoing calls (from the outer call to the inner call).

We assume a state abstraction defined by a domain $\mathbb{A}_{\mathrm{e}}^\sharp$ and a concretization function $\gamma_{\mathbb{A}_{\mathrm{e}}} : \mathbb{A}_{\mathrm{e}}^\sharp \to \mathcal{P}(\mathbb{S})$. Moreover, we let $\mathcal{T}(\tau)$ be the topmost element of τ. Then, the functional approach corresponds to the following sensitive abstraction:

—Views are abstractions of call states, thus

$$\begin{aligned} \mathbb{C} &= \mathcal{P}(\mathbb{T}), \\ \mathbb{V}^\sharp &= \mathbb{A}_{\mathrm{e}}^\sharp, \\ \gamma_{\mathbb{V}}(a_{\mathrm{e}}^\sharp) &= \{(\tau, \sigma) \mid \mathcal{T}(\tau) \in \gamma_{\mathbb{A}_{\mathrm{e}}}(a_{\mathrm{e}}^\sharp) \wedge \sigma \in \mathbb{S}\}. \end{aligned}$$

— As in the trace history sensitivity approach, the predecessor coverings and abstract execution functions adjust the view to state relations only at function call and function return sites.

Parameter Value Sensitivity and Object Sensitivity. Another common approach utilizes abstractions of parameter values so as to enhance the precision of the analysis of a function call. This approach is very common when analyzing object-oriented programs, since the dependency on the object supporting the method calls is often critical to derive precise invariants; in this case, we call the sensitivity to method parameters *object sensitivity* (Milanova et al. 2005). This clearly defines an instance of the trace history sensitivity shown in Section 6.5, as the abstract states group concrete states based on the value of some parameter at a specific point (i.e., the call sites).

In fact, parameter value sensitivity is an instance of the abstract call state sensitivity explained earlier in this subsection. Indeed, the view abstraction that underlies parameter value sensitivity simply assumes that $\mathbb{A}_e^\sharp$ and $\gamma_{\mathbb{A}_e}$ describe the possible values of procedure parameters. As an example, let us formalize object sensitivity, where the views describe the allocation site of the object a method is called on. We assume that objects are abstracted by their allocation sites, and that $\mathbb{O}$ describes the set of possible allocation sites. We write $\mathcal{A}(o)$ for the allocation site of object o. Then, object sensitivity is derived from the following call state abstraction:

$$\begin{aligned}\mathbb{A}_e^\sharp &= \mathbb{O} \uplus \{\bot, \top\},\\ \gamma_{\mathbb{A}_e}(\ell) &= \{\sigma \in \mathbb{S} \mid \mathcal{A}(\sigma(\mathbf{this})) = \ell\},\\ \gamma_{\mathbb{A}_e}(\bot) &= \emptyset,\\ \gamma_{\mathbb{A}_e}(\top) &= \mathbb{S}.\end{aligned}$$

6.7 Sum of Sensitivities

Our framework can also combine several sensitivities by summing them. Assuming two forms of sensitivities describing the same kind of behaviors $\mathbb{C}$ are defined by their sets of views $\mathbb{V}_0^\sharp$ and $\mathbb{V}_1^\sharp$ (that we assume disjoint), and their concretization functions $\gamma_{\mathbb{V}_i} : \mathbb{V}_i^\sharp \longrightarrow \mathbb{C}$ (for $i \in \{0, 1\}$), the *summed sensitivity analysis* defines $\mathbb{V}_+^\sharp$ and $\gamma_{\mathbb{V}_+^\sharp}$ as follows:

$$\begin{aligned}\mathbb{V}_+^\sharp &= \mathbb{V}_0^\sharp \uplus \mathbb{V}_1^\sharp,\\ \gamma_{\mathbb{V}_+^\sharp}(v) &= \begin{cases}\gamma_{\mathbb{V}_0}(v) & \text{if } v \in \mathbb{V}_0^\sharp\\ \gamma_{\mathbb{V}_1}(v) & \text{if } v \in \mathbb{V}_1^\sharp\end{cases}.\end{aligned}$$

Intuitively, this allows the analysis to choose to maintain two kinds of sensitivities together, without combining the views themselves: both views of $\mathbb{V}_0^\sharp$ and of $\mathbb{V}_1^\sharp$ are maintained in the same analysis. A definition of predicate coverings for $\mathbb{V}_+^\sharp$, using directly those in terms of $\mathbb{V}_0^\sharp$ and of $\mathbb{V}_1^\sharp$, would not yield a very precise result in general, as it would ignore relations between views of $\mathbb{V}_0^\sharp$ and views of $\mathbb{V}_1^\sharp$.

As we have observed that dropping the information attached to some views is always sound, a common approach consists in selecting *some* views of $\mathbb{V}_0^\sharp$ and *some* views of $\mathbb{V}_1^\sharp$. An example use of this scheme into a large analysis tool will be given in Section 7.2.

6.8 Partial Product of Sensitivities

Sensitivity tends to add cost to static analyses, as it replaces an abstract state $a^\sharp$ with *several* (possibly many) abstract states $a_0^\sharp, \ldots, a_n^\sharp$ of the same form (one per view). Products of sensitivities multiply (Section 6.3) the number of views, which makes the cost increase even worse. Therefore, when

using a product of two sensitivities $\mathbb{V}_0^\sharp$ and $\mathbb{V}_1^\sharp$ defined by their concretizations $\gamma_{\mathbb{V}_i} : \mathbb{V}_i^\sharp \longrightarrow \mathbb{C}$, it is common to apply the second sensitivity in a *local* manner with respect to the first one. This means that for some views of $\mathbb{V}_0^\sharp$, the sensitivity defined by $\mathbb{V}_1^\sharp$ is not used. For instance, when combining call sensitivity (Section 6.2) and trace history sensitivity (Section 6.5), the analysis may determine that applying the latter form of sensitivity is useful for precision only in some contexts.

To formalize this technique, we assume a partition $\mathbb{V}_{0,a}^\sharp \uplus \mathbb{V}_{0,b}^\sharp$ of $\mathbb{V}_0^\sharp$, and let $\mathbb{V}_{0,a}^\sharp$ (delresp, $\mathbb{V}_{0,b}^\sharp$, respectively) denote the views where the sensitivity $\mathbb{V}_1^\sharp$ is applied (not applied, respectively). This defines the following set of views and view concretization:

$$\begin{aligned} \mathbb{V}_+^\sharp &= \mathbb{V}_{0,a}^\sharp \times \mathbb{V}_1^\sharp \uplus \mathbb{V}_{0,b}^\sharp, \\ \forall (v_0, v_1) \in \mathbb{V}_{0,a}^\sharp \times \mathbb{V}_1^\sharp,\ \gamma_{\mathbb{V}_+^\sharp}(v_0, v_1) &= \gamma_{\mathbb{V}_0}(v_0) \cap \gamma_{\mathbb{V}_1}(v_1), \\ \forall v_0 \in \mathbb{V}_{0,b}^\sharp,\ \gamma_{\mathbb{V}_+^\sharp}(v_0) &= \gamma_{\mathbb{V}_0}(v_0). \end{aligned}$$

This sensitivity can be viewed as the combination of a sum (Section 6.7) and of a product (Section 6.3). An example use of this technique into a large analysis tool will be given in Section 7.1.

6.9 Choosing an Adequate Sensitivity

The previous paragraphs have presented a number of instances of our sensitivity framework and techniques to create new forms of sensitivities from existing ones, thus the selection of the right sensitivity for a given static analysis problem may appear challenging. In the remainder of this section, we discuss this issue, and focus on the criteria to select views to represent, on the inherent cost of sensitive analysis and how to mitigate it, and on the sensitivity selection process.

Criteria for the Selection of Views. As we observed in Section 2, the basic motivation for sensitivity is to ensure that static analysis can compute sufficiently precise results. Let us assume that an abstraction is described by a concretization function γ and a concrete operation f over the concrete domain is over-approximated by the abstract operation $f^\sharp$ over the abstract domain. We thus have $f \circ \gamma \subseteq \gamma \circ f^\sharp$. When the abstract $f^\sharp$ returns abstract states that are too coarse over-approximation of the result of $f \circ \gamma$, subsequent analysis steps or final analysis results may become unacceptable. In that setup, the switch to a sensitive abstraction allows one to apply $f^\sharp$ to different abstract elements, where hopefully less imprecision will occur.

Very often, the operation f that requires such a treatment is concrete unions. Indeed, the approximation of concrete unions tends to incur a significant precision loss when using abstract domains that describe conjunctive logical properties. As an example, if the abstract domain is based on interval constraints, then joining $[0, 5]$ with $[10, 15]$ causes to include all values between 5 and 10 in the abstraction. This case was also observed in Section 2.

However, the same issue may occur with different kinds of operations. As an example, shape analyses for inductive data structures such as dynamically linked lists need to handle precisely memory cell reads and destructive updates over summarized regions (Sagiv et al. 2002; Chang and Rival 2008). To analyze precisely such operations, these works perform a *materialization* operation that substitutes an abstract state with a disjunctive approximation of it, e.g., with two cases covering the case of an empty list and the case of a non-empty list. Although they formalize such case splits as a disjunctive analysis, we can also view these as a form of sensitivity (where the views describe the possible states of the list).

Cost Inherent in Sensitive Analyses. Added cost is an important concern when adding sensitivity to an existing analysis. Indeed, sensitivity makes the representation of abstract elements larger and

the analysis algorithms more complex. In particular, we can list the following sources of increase in analysis cost:

— the *number of views* causes an increase in the memory required for the representation of abstract elements (as seen in Section 3.2) and in the number of operations in the underlying domain, for example, for the computation of a post-condition (as shown in Section 4.2);
— the *representation and computation of views* add up to the analysis time (Section 4.2);
— the *operations that modify the set of views* cause the recomputation of elements of the base domain (Section 3.3).

Therefore, an efficient sensitive analysis should utilize enough views to reach a desired level of precision, but should avoid carrying too many unnecessary views, and should avoid the most costly operations such as global views reorganization (in the case of a dynamic sensitivity) and reduction. Several techniques presented in the previous sections allow one to limit the number of views that are effectively represented. For instance, dynamic sensitivity (Section 5) allows not to represent all views at all times. Similarly, the partial product (Section 6.8) generates smaller sets of views than a direct product.

The trace partitioning framework of Rival and Mauborgne (2007) provides a representation of views and a set of operations which minimize the costly reductions and modifications over partitions in the case of trace history sensitivity. Other specific sets of views may lead to different implementations.

Sensitivity Selection Process. There is no general and systematic way to characterize the operation that is limiting the precision of a static analysis, thus the choice of views to solve such a precision issue is generally not systematic and typically requires human intervention.

Dynamic sensitivity (Section 5) can be considered as a partially automated view selection process, where a range of possible views is fixed first, and the analysis selects in this range set of useful views at each step. The design of this view selection algorithm is generally a complex task, which cannot be automated.

7 FRAMEWORK INSTANCES

We now show how a few selected analysis tools (for C and JavaScript) implement various forms of sensitivities that can be formalized inside our framework.

7.1 Astrée

Astrée (Blanchet et al. 2003) is a static analyzer that was designed specifically for the verification of absence of runtime errors in synchronous safety critical embedded softwares, as sound in avionics, aerospace, automotive, and energy production systems. It is aimed at computing very precise invariants for such softwares, and its design takes advantage of the characteristics of embedded programs.

Sensitivity in Astrée. The analysis makes a very careful use of several forms of sensitivity, in order to achieve this precision/efficiency ratio: the analysis is *fully flow* sensitive and *fully calling-context* sensitive, it performs *trace partitioning* (Rival and Mauborgne 2007), and it also utilizes *partial state partitioning guided* by the values of Boolean variables (Blanchet et al. 2003). These forms of sensitivity can all be expressed in our framework.

Indeed, flow sensitivity works as described in Section 6.1, and calling-context partitioning follows the full context sensitivity approach presented in Section 6.2. Since safety critical embedded softwares are supposed *not* to use recursion, the analysis is fully sensitive to the calling-context,

which allows a very high level of precision in softwares where some functions can be called in very different contexts.

The trace partitioning abstract domain (Rival and Mauborgne 2007) implements the trace history sensitivity described in Section 6.5. It is defined by an alphabet of directives $\mathcal{D}$ containing the following elements:

— $\mathbf{Dir}_\ell^T, \mathbf{Dir}_\ell^F$, when the analyzed code contains an if-statement at program location ℓ (these directives respectively denote the traces that went to the true and false branches of the if-statement the last time it was traversed);
— $\mathbf{Dir}_\ell^0, \mathbf{Dir}_\ell^1, \ldots, \mathbf{Dir}_\ell^n, \mathbf{Dir}_\ell^{>n}$ (where n is a fixed integer), when the analyzed code contains a loop statement at program location ℓ (these directives respectively denote the traces that went through 0, 1, n, and more than n iterations of this loop the last time it was reached);
— $\mathbf{Dir}_\ell^{\mathtt{x}=k}$ (where k is an integer), when the analyzed code contains an integer variable x and a location ℓ (this directive denotes the traces such that variable x was containing value k the last time it reached location ℓ).

Trace partitioning allows one to tie the abstraction of scalar values to the control flow paths, and is most useful in order to derive precise invariants of programs containing multiple if-statements with related conditions, and mathematical functions such as interpolations. Interpolations are often computed using loop-up tables searches through loops (array dereferences, respectively) and thus can be better analyzed using loop partitioning (trace partitioning guided by the value of a variable at the beginning of the interpolation, respectively). While the above directives can be defined for all matching code locations, the analysis will in practice use only a small set of directives, determined by analysis strategies (thus, Astrée uses *dynamic* sensitivity).

Value sensitivity follows the principles of Section 6.4, and is applied only on Boolean variables, as synchronous softwares often encode part of the control flow in Booleans, and keeping relations between Booleans and scalars is crucial to achieve precise invariants. This form of sensitivity is never applied to the global state, and usually relates only a small group of scalar variables to the values of a few Boolean variables, determined by static analyzer strategies (Blanchet et al. 2003).

Combination of Sensitivities in Astrée. Intuitively, the basic technique for combining sensitivities is the multiplication composition described in Section 6.3. In fact, Rival and Mauborgne (2007) formalizes the calling context sensitivity as part of the trace partitioning abstraction, effectively defining the multiplication of the two.

However, the analysis strategies make this combination more complex, and some forms of trace partitioning are applied in a parsimonious and local manner: a pre-analysis phase computes candidate partitioning points, and the analysis assesses whether partitioning should be done when it reaches these points, thus it may decide to actually do the partitioning or not. For instance, when the pre-analysis phase marks a piece of code as a candidate for trace partitioning guided by the values of a variable, and when the analysis computes a large range (over a few hundreds), so that the partitioning would be costly, then no partitioning is done for that variable. This combination of sensitivities implements a dynamic (Section 5) partial product of sensitivities (Section 6.8).

Impact of Sensitivity in Astrée. Sensitivity plays a great role in the performance of the analyzer (both in terms of precision and of runtime). For instance, Rival and Mauborgne (2007) performs a comparison between path history sensitive and path history insensitive analyses: on the largest industrial code considered (400 kLOC), enabling this form of sensitivity increases the analysis time by 6% but reduces the number of alarms from 7,524 to 0. In fact, the added cost inherent in the sensitivity is made up by the fact the analysis does not need to explore so many unreachable

paths and to compute as long iteration sequences (converging slowly toward imprecise invariants), as a less precise analysis would. By contrast, a brutal product of sensitivities would generate an exponential tower of sensitivities, and hence would not scale.

7.2 Sparrow

Sparrow (Oh et al.) is a static analyzer that aims to verify the absence of fatal bugs in C programs. While Astrée is specialized for analyzing synchronous safety critical embedded softwares, Sparrow is designed for supporting the full set of the C programming language. It uses various analysis techniques such as a general sparse analysis framework (Oh et al. 2012) for scalability and alarm clustering (Lee et al. 2012a) for convenience. In addition to traditional sensitivities, Sparrow supports a way to use context sensitivity selectively (Oh et al. 2014). After estimating the impact of context-sensitivity on the analysis's precision, it turns on and off context-sensitivity depending on whether it improves the analysis precision.

Selective Context Sensitivity in Sparrow. The selective context sensitivity can be also expressed in our framework. Sparrow uses a context selector K that maps procedures to sets of selected calling contexts that are of interest, and it uses ϵ to denote all the other contexts not included in K. In our framework, we can have two views $\mathbb{V}_0^\sharp$ for full contexts and $\mathbb{V}_1^\sharp$ for partial contexts. Accordingly, we can denote the full context sensitivity as $\mathbb{V}_0^\sharp \to \mathbb{A}^\sharp$ and a partial context sensitivity as $\mathbb{V}_1^\sharp \to \mathbb{A}^\sharp$. Then, we can represent the Sparrow's selective context sensitivity using the sum composition described in Section 6.7. It is a summed sensitivity of $\mathbb{V}_0^\sharp$ and $\mathbb{V}_1^\sharp$ like $m \in \mathbb{V}_0^\sharp + \mathbb{V}_1^\sharp \to \mathbb{A}^\sharp$ where we forget some views of m by implicitly mapping them to $\top$ losing some analysis precision. Indeed, it is very close to what Astrée does with trace partitioning. To make the analysis scalable while keeping the precision, the Sparrow analyzer uses context sensitivity for the parts of the code where it will have more impact.

Impact of the Selective Context Sensitivity in Sparrow. Selective context sensitivity serves an important role in the analysis precision with modest overhead. For example, Oh et al. (2014) performs a comparison between the baseline context-insensitive interval analysis and the selective context-sensitive analysis: the selective sensitivity reduces the number of (false) alarms by 24.4%, while increasing the analysis cost by 27.8% on average. As the selective sensitivity method also improves the precision of a relational analysis, it may be applicable to other sensitive analyses like flow-sensitive analysis and loop-sensitive analysis (Park and Ryu 2015).

7.3 SAFE

SAFE (Scalable Analysis Framework for ECMAScript) (Lee et al. 2012b; KAIST PLRG 2014) is a general analysis framework for JavaScript web applications. It is designed to be an open-source, pluggable framework to support various kinds of JavaScript analyses including static and dynamic analyses. It provides a baseline static analyzer, a clone detector (Cheung et al. 2016), and a bug detector that can report type-related bugs and misuses of Web APIs (Bae et al. 2014).

Sensitivity in SAFE. The baseline analysis supports numerous forms of sensitivity: the analysis can be flow sensitive, k call-string sensitive, object sensitive (Milanova et al. 2005), and loop sensitive (Park and Ryu 2015). These forms of sensitivity can all be expressed in our framework.

The k call-string sensitivity follows the partial context sensitivity principle described in Section 6.2. Unlike C program analysis where highly sensitive analysis with high k produces precise analysis results but requires an expensive cost, analysis of JavaScript programs with high k may provide better precision at a much cheaper cost than its counterpart with low k (Kashyap et al.

2014). SAFE provides an infrastructure to experiment with different contexts to understand the peculiarities of JavaScript analysis.

The object sensitivity follows the principles of Section 6.6, and is applied only on the receiver object. While object sensitivity performs better than k call-string sensitivity for Java program analysis in general (Milanova et al. 2005), it does not apply to JavaScript program analysis (Kashyap et al. 2014) due to the extremely dynamic and functional features of JavaScript.

The loop sensitivity (Park and Ryu 2015) works as an instance of the trace history sensitivity described in Section 6.5. It is similar to the trace partitioning abstract domain (Rival and Mauborgne 2007) used in Astrée. While trace partitioning uses default values for loop unrolling counts that can be modified by partitioning strategies or annotations, the loop sensitivity distinguishes loop iterations as many as needed by automatically choosing loop unrolling numbers during analysis. The loop sensitivity distinguishes different loop contexts precisely, which is critical in the analysis performance (in terms of both precision and scalability) of JavaScript programs where object properties are dynamically constructed and initialized their values via loops.

Combination of Sensitivities in SAFE. The basic technique for combining sensitivities is the multiplication composition described in Section 6.3. In fact, Park and Ryu (2015) uses a combination of flow sensitivity, k call-string sensitivity, and loop sensitivity as the multiplication of all of them. To maintain the analysis scalable even with the multiplication of sensitivities, loop sensitivity controls the number of distinct contexts while preserving its soundness.

Impact of Sensitivity in SAFE. Sensitivity is critical in the performance of the SAFE analyzer. For instance, Park and Ryu (2015) performs a comparison between the analysis results of SAFE with those of state-of-the-art static analyzers, TAJS (Møller et al. 2014) and WALA (IBM Research 2003). For programs that simply load several versions of the top 5 JavaScript libraries according to W3Techs,[2] WALA could not analyze most of them. Even though TAJS is also equipped with various sensitivities as we discuss in Section 7.4, it could not analyze programs that load BootStrap,[3] Mootools,[4] and Prototype.[5] On the contrary, SAFE can analyze all but BootStrap.

7.4 TAJS

TAJS (Andreasen and Møller 2014; Møller et al. 2014) is a static analyzer for JavaScript programs. In addition to conventional sensitivities like flow sensitivity, context sensitivity, and object sensitivity, it extends its context sensitivity to distinguish more functions and loops using the values of function parameters and loop variables selected in heuristic ways (Andreasen and Møller 2014).

On-the-Fly Heuristics for Sensitivities in TAJS. In order to improve the analysis precision, TAJS introduces heuristics to several sensitivity techniques. It selectively applies value sensitivity described in Section 6.4 by choosing values to distinguish during analysis, dubbed *selective parameter sensitivity*. It takes advantage of its constant propagation analysis to select parameters with constant values in each context to identify them as determinacy information. It also introduces two kinds of *loop specialization*, one for `for` loops and the other for `for-in` loops. Indeed, such sensitivities are selective with simple heuristics, thus, TAJS uses dynamic sensitivity.

Impact of the On-the-Fly Heuristics for Sensitivities in TAJS. The combination of different on-the-fly heuristics for sensitivities improves analysis precision, while it increases theoretical worst-case

[2] http://w3techs.com/technologies/overview/javascript_library/all.
[3] http://getbootstrap.com/javascript/.
[4] http://mootools.net.
[5] http://prototypejs.org.

complexity. Even though the multiplication of various sensitivities may be too costly in theory, analysis of programs using the most widely used jQuery[6] library did not show the overhead in practice.

8 CONCLUSION

We proposed a general abstract interpretation framework to formalize sensitive analyses, which can describe a wide set of sensitivities. The main foundation of our framework is a precise tracking of the link between views that define what the analysis is sensitive to, and sets of program behaviors (usually states or traces) that can be represented using other abstractions. It is based on the reduced cardinal power abstract domain construction technique (Cousot and Cousot 1979).

Our framework provides general definitions for the sensitive abstract domains, their concretization functions (and abstraction functions when they exist), abstract execution functions, and analysis algorithms. It relies on a notion of view that generalizes the trace partitions of Rival and Mauborgne (2007). It can be used not only when views are statically known but also when they are computed during the analysis itself. We have demonstrated that many kinds of sensitive analyses can be viewed as instances of our framework. Furthermore, our approach allows one to compose several forms of sensitivities together, so as to get increased precision.

The main advantage of our formalization is that it identifies precisely the relation between views and abstractions of sets of program behaviors, and thus it provides great control over analysis precision and cost by pointing out how the information over views is tracked during the analysis (for instance, in the definition of reduction operators). Combination techniques such as sum of sensitivities or partial product of sensitivities allow one to further trim the cost of a static analysis that still provides a sufficient level of precision. We have also shown the generality of our approach by showing how it accounts for design choices that underlie various static analysis tools, even when they were not formalized explicitly in such a general way.

APPENDIX

A PROOFS

Proof of Lemma 3.8 (Sensitive abstraction Galois connection). Let $E \subseteq \mathbb{E}$ be a set of behaviors and $s^\sharp \in \mathbb{S}^\sharp$ be an abstract state in the sensitive abstract domain. Then,

$$
\begin{aligned}
& E \subseteq \gamma_{\mathbb{S}}(s^\sharp) \\
& \iff \forall v \in \mathbb{V}^\sharp,\ E \subseteq \{e \in \mathbb{E} \mid e \in \gamma_{\mathbb{V}}(v) \implies e \in \gamma_{\mathbb{A}}(s^\sharp(v))\} \\
& \iff \forall v \in \mathbb{V}^\sharp,\ \forall e \in \mathbb{E},\ e \in \gamma_{\mathbb{V}}(v) \implies e \in \gamma_{\mathbb{A}}(s^\sharp(v)) \\
& \iff \forall v \in \mathbb{V}^\sharp,\ E \cap \gamma_{\mathbb{V}}(v) \subseteq \gamma_{\mathbb{A}}(s^\sharp(v)) \\
& \iff \forall v \in \mathbb{V}^\sharp,\ \alpha_{\mathbb{A}}(E \cap \gamma_{\mathbb{V}}(v)) \sqsubseteq_{\mathbb{A}} s^\sharp(v) \\
& \iff \forall v \in \mathbb{V}^\sharp,\ \alpha_{\mathbb{S}}(E)(v) \sqsubseteq_{\mathbb{A}} s^\sharp(v) \qquad \text{where } \alpha_{\mathbb{S}} \text{ is defined as in Lemma 3.8.} \\
& \iff \alpha_{\mathbb{S}}(E) \sqsubseteq_{\mathbb{S}} s^\sharp
\end{aligned}
$$

This concludes the proof.

Proof of Lemma 3.12. We recall that

$$\gamma_{\mathbb{S}}(s^\sharp) = \left\{e \in \mathbb{E} \mid \forall v \in \mathbb{V}^\sharp,\ e \in \gamma_{\mathbb{V}}(v) \implies e \in \gamma_{\mathbb{A}}(s^\sharp(v))\right\}.$$

We let Γ be defined by

$$
\begin{aligned}
\Gamma : \mathbb{S}^\sharp &\longrightarrow \mathcal{P}(\mathbb{E}) \\
s^\sharp &\longmapsto \bigcup \left\{ C \cap \gamma_{\mathbb{A}} \left(\bigsqcap\nolimits_{\mathbb{A}} \left\{ s^\sharp(v) \middle| C \subseteq \gamma_{\mathbb{V}}(v) \right\} \right) \middle| C \in \mathbb{E}/\equiv_{\mathbb{V}} \right\}.
\end{aligned}
$$

[6] http://jquery.com.

To prove the lemma, we simply let $s^\sharp$ be any element of $\mathbb{S}^\sharp$ and show that $\Gamma(s^\sharp) = \gamma_{\mathbb{S}}(s^\sharp)$. We prove this equality by double inclusion.

—Let us first prove that $\gamma_{\mathbb{S}}(s^\sharp) \subseteq \Gamma(s^\sharp)$. We let $e \in \gamma_{\mathbb{S}}(s^\sharp)$, and show that $e \in \Gamma(s^\sharp)$. Let C_0 be the equivalence class of e for relation $\equiv_{\mathbb{V}}$. By definition of $\equiv_{\mathbb{V}}$, the relation $e \in C_0$ holds. If v is a view such that $e \in \gamma_{\mathbb{V}}(v)$, then, by definition of $\gamma_{\mathbb{S}}$, $e \in \gamma_{\mathbb{A}}(s^\sharp(v))$. In particular, this holds for any view v such that $C_0 \subseteq \gamma_{\mathbb{V}}(v)$. Therefore,

$$e \in \bigcap \left\{ \gamma_{\mathbb{A}}(s^\sharp(v)) \middle| C_0 \subseteq \gamma_{\mathbb{V}}(v) \right\}.$$

We recall that concretization functions and greatest lower bounds commute. Moreover, $e \in C_0$. Thus the above rewrites into

$$e \in C_0 \cap \gamma_{\mathbb{A}} \left(\bigsqcap_{\mathbb{A}} \left\{ s^\sharp(v) \middle| C_0 \subseteq \gamma_{\mathbb{V}}(v) \right\} \right).$$

To sum up, we have proved that

$$e \in \bigcup \left\{ C \cap \gamma_{\mathbb{A}} \left(\bigsqcap_{\mathbb{A}} \left\{ s^\sharp(v) \middle| C \subseteq \gamma_{\mathbb{V}}(v) \right\} \right) \middle| C \in \mathbb{E}/\equiv_{\mathbb{V}} \right\} = \Gamma(s^\sharp).$$

This terminates the proof of the first inclusion.

—Secondly, we prove that $\Gamma(s^\sharp) \subseteq \gamma_{\mathbb{S}}(s^\sharp)$. We let $e \in \Gamma(s^\sharp)$, and show that $e \in \gamma_{\mathbb{S}}(s^\sharp)$. By definition of Γ, there exists an equivalence class C_0 or relation $\equiv_{\mathbb{V}}$ such that $e \in C_0$ and

$$e \in \gamma_{\mathbb{A}} \left(\bigsqcap_{\mathbb{A}} \left\{ s^\sharp(v) \middle| C_0 \subseteq \gamma_{\mathbb{V}}(v) \right\} \right).$$

We let v_0 be a view such that $e \in \gamma_{\mathbb{V}}(v_0)$ and we show that $e \in \gamma_{\mathbb{A}}(s^\sharp(v_0))$. By definition of $\equiv_{\mathbb{V}}$ and since C_0 is an equivalence class for this relation, $e \in \gamma_{\mathbb{V}}(v_0)$ implies that $C_0 \subseteq \gamma_{\mathbb{V}}(v_0)$. Therefore,

$$\bigsqcap_{\mathbb{A}} \left\{ s^\sharp(v) \middle| C_0 \subseteq \gamma_{\mathbb{V}}(v) \right\} \sqsubseteq_{\mathbb{A}} s^\sharp(v_0)$$

and since $\gamma_{\mathbb{A}}$ is monotone,

$$\gamma_{\mathbb{A}} \left(\bigsqcap_{\mathbb{A}} \left\{ s^\sharp(v) \middle| C_0 \subseteq \gamma_{\mathbb{V}}(v) \right\} \right) \subseteq \gamma_{\mathbb{A}}(s^\sharp(v_0)).$$

We have seen above that e belongs to the left-hand side, thus $e \in \gamma_{\mathbb{A}}(s^\sharp(v_0))$. This finishes the proof of the second inclusion.

Proof of Lemma 3.14. Let $s^\sharp \in \mathbb{S}^\sharp$ be an abstract state in the sensitive abstract domain and $v \in \mathbb{V}^\sharp$ be a view. Let e be a behavior such that $e \in \gamma_{\mathbb{S}}(s^\sharp) \cap \gamma_{\mathbb{V}}(v)$. Then, by definition of $\gamma_{\mathbb{S}}$, and since $e \in \gamma_{\mathbb{V}}(v)$, we deduce that $e \in \gamma_{\mathbb{A}}(s^\sharp(v))$. As this holds for all elements of $\gamma_{\mathbb{S}}(s^\sharp) \cap \gamma_{\mathbb{V}}(v)$, we derive that $\gamma_{\mathbb{S}}(s^\sharp) \cap \gamma_{\mathbb{V}}(v) \subseteq \gamma_{\mathbb{A}}(s^\sharp(v))$. Thus,

$$\alpha_{\mathbb{A}}(\gamma_{\mathbb{S}}(s^\sharp) \cap \gamma_{\mathbb{V}}(v)) \sqsubseteq_{\mathbb{A}} s^\sharp(v).$$

Proof of Lemma 3.16. We let $E \subseteq \mathbb{E}$ be a set of program behaviors and $s^\sharp \in \mathbb{S}^\sharp$ be an abstract state. Moreover, we assume $V \in \mathbf{Covers}_{\mathbb{V}^\sharp}(E)$ is a covering of E:

$$E \subseteq \bigcup_{v \in V} \gamma_{\mathbb{V}}(v).$$

Then

$$
\begin{aligned}
&\alpha_{\mathbb{A}}(\gamma_{\mathbb{S}}(s^\sharp) \cap E) \\
&\quad \sqsubseteq_{\mathbb{A}} \alpha_{\mathbb{A}}\left(\gamma_{\mathbb{S}}(s^\sharp) \cap \left(\bigcup\nolimits_{v \in V} \gamma_{\mathbb{V}}(v)\right)\right) \\
&\qquad \text{since } \alpha_{\mathbb{A}} \text{ is monotone} \\
&\quad = \alpha_{\mathbb{A}}\left(\bigcup\nolimits_{v \in V} (\gamma_{\mathbb{S}}(s^\sharp) \cap \gamma_{\mathbb{V}}(v))\right) \\
&\qquad \text{by distributivity} \\
&\quad = \alpha_{\mathbb{A}}\left(\bigcup\nolimits_{v \in V} (\gamma_{\mathbb{A}}(s^\sharp(v)) \cap \gamma_{\mathbb{V}}(v))\right) \\
&\qquad \text{since } \gamma_{\mathbb{S}}(s^\sharp) \cap \gamma_{\mathbb{V}}(v) \subseteq \gamma_{\mathbb{A}}(s^\sharp(v)) \cap \gamma_{\mathbb{V}}(v) \\
&\quad = \bigsqcup\nolimits_{\mathbb{A}} \left\{\alpha_{\mathbb{A}}(\gamma_{\mathbb{A}}(s^\sharp(v)) \cap \gamma_{\mathbb{V}}(v)) \middle| v \in V\right\} \\
&\qquad \text{as in general, } \alpha_{\mathbb{A}}(E_0 \cup E_1) = \alpha_{\mathbb{A}}(E_0) \sqcup_{\mathbb{A}} \alpha_{\mathbb{A}}(E_1) \\
&\quad \sqsubseteq_{\mathbb{A}} \bigsqcup\nolimits_{\mathbb{A}} \left\{\alpha_{\mathbb{A}}(\gamma_{\mathbb{A}}(s^\sharp(v))) \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}}(\gamma_{\mathbb{V}}(v)) \middle| v \in V\right\} \\
&\qquad \text{since } \alpha_{\mathbb{A}}(E_0 \cap E_1) \sqsubseteq_{\mathbb{A}} \alpha_{\mathbb{A}}(E_0) \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}}(E_1) \\
&\quad \sqsubseteq_{\mathbb{A}} \bigsqcup\nolimits_{\mathbb{A}} \left\{s^\sharp(v) \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}}(\gamma_{\mathbb{V}}(v)) \middle| v \in V\right\} \\
&\qquad \text{since } \alpha_{\mathbb{A}} \circ \gamma_{\mathbb{A}}(s^\sharp) \sqsubseteq_{\mathbb{A}} s^\sharp.
\end{aligned}
$$

Proof of Lemma 3.17 (Reduction operator). Let $s^\sharp \in \mathbb{S}^\sharp$ be an abstract state. We show that $\alpha_{\mathbb{S}} \circ \gamma_{\mathbb{S}}(s^\sharp)(v_0) \sqsubseteq_{\mathbb{S}} \rho_{\mathbb{S}}(s^\sharp)(v_0)$ for any view $v_0 \in \mathbb{V}^\sharp$:

$$
\begin{aligned}
&\alpha_{\mathbb{S}} \circ \gamma_{\mathbb{S}}(s^\sharp)(v_0) \\
&\quad = \alpha_{\mathbb{A}}(\gamma_{\mathbb{S}}(s^\sharp) \cap \gamma_{\mathbb{V}}(v_0)) \\
&\quad \sqsubseteq_{\mathbb{A}} \bigsqcap\nolimits_{\mathbb{A}} \left\{\bigsqcup\nolimits_{\mathbb{A}} \left\{s^\sharp(v) \sqcap_{\mathbb{A}} \alpha_{\mathbb{A}}(\gamma_{\mathbb{V}}(v)) \middle| v \in V\right\} \middle| V \in \mathbf{Covers}_{\mathbb{V}^\sharp}(\gamma_{\mathbb{V}}(v_0))\right\} \\
&\qquad \text{by applying Lemma 3.16 to } E = \gamma_{\mathbb{V}}(v_0) \\
&\qquad \text{and to any of its coverings } V \in \mathbf{Covers}_{\mathbb{V}^\sharp}(\gamma_{\mathbb{V}}(v_0)) \\
&\quad = \rho_{\mathbb{S}}(v_0).
\end{aligned}
$$

REFERENCES

Ole Agesen. 1995. The Cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*. Springer-Verlag, 2–26.

Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'14)*. ACM, 17–31.

SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFE$_{\text{WAPI}}$: Web API misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE'14)*. ACM, 507–517.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jéróme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, 196–207.

François Bourdoncle. 1992. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming* 2, 4 (1992), 407–423.

Bor-Yuh Evan Chang and Xavier Rival. 2008. Relational inductive shape analysis. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*. 247–260.

Wai Ting Cheung, Sukyoung Ryu, and Sunghun Kim. 2016. *Empirical Software Engineering* 21, 2 (2016), 517–564.

Patrick Cousot. 1981. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, Chapter 10, 303–342.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*. ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'79)*. ACM, 269–282.

Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, 105–118.

Roberto Giacobazzi and Francesco Ranzato. 1999. The reduced relative power operation on abstract domains. *Theoretical Computer Science* 216, 1–2 (1999), 159–211.

Roberto Giacobazzi, Francesco Ranzato, and Francesca Scozzari. 2005. Making abstract domains condensing. *ACM Transactions on Computational Logic (TOCL)* 6, 1 (2005), 33–60.

Roberto Giacobazzi and Francesca Scozzari. 1998. A logical model for relational abstract domains. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20, 5 (1998), 1067–1109.

Maria Handjieva and Stanislav Tzolovski. 1998. Refining static analyses by trace-based partitioning using control flow. In *Proceedings of the 5th International Symposium on Static Analysis (SAS'98)*. Springer-Verlag, 200–214.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*. ACM, 58–70.

IBM Research. 2003. T. J. Watson Libraries for Analysis (WALA). Retrieved August 4, 2018 from http://wala.sf.net.

Bertrand Jeannet. 2003. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design* 23, 1 (2003), 5–37.

KAIST PLRG. 2014. SAFE: Scalable Analysis Framework for ECMAScript. Retrieved from http://safe.kaist.ac.kr.

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM.

Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012b. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *Proceedings of the International Workshop on Foundations of Object Oriented Languages (FOOL'12)*.

Woosuk Lee, Wonchan Lee, and Kwangkeun Yi. 2012a. Sound non-statistical clustering of static analysis alarms. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12)*. Springer-Verlag, 299–314.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (ToSEM)* 14, 1 (2005), 1–41.

Anders Møller, Simon Holm Jensen, Peter Thiemann, Magnus Madsen, Matthias Diehn Ingesman, Peter Jonsson, and Esben Andreasen. 2014. TAJS: Type Analyzer for JavaScript. Retrieved from https://github.com/cs-au-dk/TAJS.

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2009. Sparrow. Retrieved from http://ropas.snu.ac.kr/sparrow.

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, 229–238.

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective context-sensitivity guided by impact pre-analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, 475–484.

Changhee Park and Sukyoung Ryu. 2015. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'15)*. Dagstuhl Publishing, 735–756.

Thomas W. Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. ACM, 49–61.

Xavier Rival and Laurent Mauborgne. 2007. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 5 (2007), 26.

Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 2002. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24, 3 (2002), 217–298.

Micha Sharir and Amir Pnueli. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, Chapter 7.

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. dissertation. Carnegie Mellon University.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. 2011. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. ACM, 17–30.

Arnaud Venet. 1996. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Proceedings of the 3rd International Symposium on Static Analysis (SAS'96)*. Springer-Verlag, 366–382.

Received April 2016; revised April 2017; accepted May 2018