Abstract Dependences for Alarm Diagnosis

Xavier Rival

École Normale Supérieure
45, rue d'Ulm,
75230, Paris cedex 5, France

Abstract. We propose a framework for dependence analyses, adapted –among others– to the understanding of static analyzers outputs. Static analyzers like ASTRÉE are sound but not complete; hence, they may yield false alarms, that is report not being able to prove part of the properties of interest. Helping the user in the alarm inspection task is a major challenge for current static analyzers. Semantic slicing, i.e. the computation of precise abstract invariants for a set of erroneous traces, provides a useful characterization of a possible error context. We propose to enhance semantic slicing with information about abstract dependences. Abstract dependences should be more informative than mere dependences: first, we propose to restrict to the dependences that can be observed in a slice; second, we define dependences among abstract properties, so as to isolate abnormal behaviors as source of errors. Last, stronger notions of slicing should allow to restrict slices to such dependences.

1 Introduction

In the last few years, many static analyzers were developed so as to answer the need for certification methods and to check that critical programs satisfy certain correctness properties, such as memory properties [21], the safety of pointer operations [25], the absence of buffer overruns [15], or the absence of runtime errors [11, 5]. These tools should produce sound results (they should not claim any false property to hold) and be automatic (they infer program invariants for the certification instead of asking the user to provide the invariants and just check them). Due to the undecidability of the properties they intend to prove, these tools are necessarily incomplete: they may report *false alarms*, i.e. critical operations they are not able to prove safe. From the user point of view, an alarm could be either a true error, or a false alarm (which may be non-trivial to check manually); hence, alarm inspection is a major issue in static analysis.

In the case of ASTRÉE, a static analyzer for proving the absence of runtime-errors in large C programs, a lot of work was done in order to make the analyzer precise, i.e. reduce the number of false alarms [5]; this approach allowed us to reduce the number of false alarms to 0 in some families of programs. Then, our previous work [23] proposed *semantic slicing* as a way to approximate precisely a set of executions satisfying some conditions; it may help to prove an alarm false or to make the alarm diagnosis process easier. Among possible *criteria*

```
X, X0, X1, X2, X3
    floating point variables
t, floating point array of length 2
initializations:
t[2] = {0, 0}; X = 0;

l0  while(true){
l1    input(X0 ∈ [−100., 100.]);
l2    input(X1 ∈ [−50 000, 50 000]);
l3    X2 = −0.5 * t[0] + 5 * t[1] + X;
l4    t[0] = t[1];
l5    t[1] = X1;
l6    X3 = X0 + X1;
l7    X = X3 + X2;
l8  }
l9  ...
```
Fig. 1: An unstable retroaction

for defining semantic slices, we can cite the data of a (set of) final state(s) (e.g. states which may lead to an error), of conditions on the inputs, and of "execution patterns" which specify sets of control flow paths (described, e.g. by automata). Semantic slices are helpful in the alarm inspection process. Yet, the amount of data to investigate may still be fairly important. Moreover, [23] requires the user to provide the semantic slicing criteria, so we wish to help the user with a more automatized process, even though these criteria are usually rather simple.

We propose to reinforce the basic dependence analysis implemented in [23] with more restrictive analyses, producing fewer dependences supposed to be more "related" to the alarm under investigation. More precisely, we intend to restrict to dependences that can be *observed* on a set of program executions corresponding to an alarm and to compute abstract dependences, i.e. chains of dependences among *abstract properties* likely to capture the cause for an alarm. Such dependences should help making the alarm inspection process more automatic, by providing good candidate slicing criteria. For instance, in the example of Fig. 1, an unstable retroaction causes $X, X2$ to diverge: in case the input $X1$ is large for all iterations, X will grow, and will eventually overflow –whatever $X0$. Astrée discovers two alarms at l_3 and l_7. Semantic slicing allows to inspect sets of diverging traces but does not lead to the causes for the divergence. We expect some dependence analysis to provide some hint about what part of the program to look at; for instance, the input $X0$ plays little role in the alarm compared to $X1$, so we would expect to rule it out, which is not achieved by classical slicing [26], conditioned slicing [6], or semantic slicing [23] methods. Last, the cyclic dependence among "diverging" variables $(X, X2)$ should suggest to unroll the loop in order to study the divergence; this information may be used to inferring semantic slicing criteria automatically, thus enhancing [23].

The contribution of this paper is both theoretical and practical:

– we introduce alternative, more selective notions of dependences and propose algorithms for computing them; we also propose new notions of non-executable, but analyzable program slices;
– we illustrate these concepts with examples and case studies, together with early implementation results; moreover, we show how these dependences help for better semantic slicing and more efficient alarm inspection.

Sect. 2 defines observable and abstract dependences and shows the relevance of these notions. Sect. 3 provides an ordering among abstract dependences. Sect. 4 focuses on the approximation of observable dependences. Sect. 5 tackles the case of abstract dependences. Sect. 6 presents a few case studies. Sect. 7 concludes and reviews related work.

2 Dependences framework

2.1 Basic notations

We let $\mathbb{X}$ (resp. $\mathbb{V}$) denote the set of variables (resp. of values); we write $\mathbb{e}$ (resp. $\mathbb{s}$) for the set of expressions (resp. statements, aka programs). We assume that $\mathbb{X}$ is finite. A variable (resp. value) has scalar or boolean type. An expression is either a constant $v \in \mathbb{V}$, a variable $x \in \mathbb{X}$, or the application $e_0 \oplus e_1$ of a binary operator $\oplus$ to a pair of expressions $e_0, e_1 \in \mathbb{e}$. A statement is either an assignment $x = e$ (where $x \in \mathbb{X}$, $e \in \mathbb{e}$), a conditional $\mathbf{if}(e)\{s_0\}\mathbf{else}\{s_1\}$ (where $e \in \mathbb{e}, s_0, s_1 \in \mathbb{s}$), a loop $\mathbf{while}(e)\{s_0\}$, a sequence of statements $s_0; \ldots; s_1$, or an $\mathbf{input}(x \in V)$ statement which writes a random value chosen in $V \subseteq \mathbb{V}$ into variable x. We do not consider more involved C data and control structures (pointers, unions, functions, recursion) so as to make the presentation less technical. The control point before each statement and at the end of each block is associated to a unique label $l \in \mathbb{L}$.

We let $\mathbb{S}$ denote the set of states; a state is defined by a control state $l \in \mathbb{L}$ and a memory state $\rho \in \mathbb{M}$, so that $\mathbb{S} = \mathbb{L} \times \mathbb{M}$. An execution (or *trace*) σ of a program is a finite sequence of states $\langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle$ such that $\forall i,\ (l_i, \rho_i) \rightarrow (l_{i+1}, \rho_{i+1})$ where $(\rightarrow) \subseteq \mathbb{S}^2$ is the transition relation of the program; Σ is the set of all traces. For instance, in the case of the assignment $l_0 : x := e; l_1$, there is a transition $(l_0, \rho) \rightarrow (l_1, \rho[x \leftarrow [\![e]\!](\rho)])$, where $[\![e]\!] \in \mathbb{M} \rightarrow \mathbb{V}$; in the case of the input statement $l_0 : \mathbf{input}(x \in V); l_1$, then $(l_0, \rho) \rightarrow (l_1, \rho[x \leftarrow v])$, where $v \in V$. The semantics $[\![s]\!]$ of program s collects all such traces. If P is a set of stores and $x \in \mathbb{X}$, we write $P(x)$ for $\{\rho(x) \mid \rho \in P\}$.

2.2 Dependences on functions

Our purpose is to track the following kind of dependences: we would like to know what observation of the program (that is, which variable, and at which point) may affect the value (or some abstraction of it) of variable x at point l. We give a definition for "classical" dependences in the case of functions first and extend this definition to the dependences in control flow graphs afterwards; extended definitions are provided in the next subsections. We write $\mathfrak{Den}$ for $\mathbb{M} \rightarrow \mathcal{P}(\mathbb{M})$.

Definition 1 (Classical dependences). *Let $\phi \in \mathfrak{Den}$, $x_0, x_1 \in \mathbb{X}$. We say that ϕ induces a* dependence *of x_1 on x_0 if and only if there exist $\rho_0 \in \mathbb{M}$, $v_a, v_b \in \mathbb{V}$ such that $\phi(\rho_a)(x_1) \neq \phi(\rho_b)(x_1)$ where $\rho_i = \rho_0[x_0 \leftarrow v_i]$. Such a dependence is written $x_1 \overset{\phi}{\rightsquigarrow} x_0$ (or $x_1 \rightsquigarrow x_0$ when there is no ambiguity about the function ϕ). Last, we let $\mathfrak{D}_{\mathrm{f}}[\phi]$ denote the set of dependences induced by ϕ: $\mathfrak{D}_{\mathrm{f}}[\phi] = \{(x_0, x_1) \mid x_1 \overset{\phi}{\rightsquigarrow} x_0\} \in \mathfrak{Dep}_{\mathrm{f}}$ (we write $\mathfrak{Dep}_{\mathrm{f}} = \mathcal{P}(\mathbb{X}^2)$).*

Intuitively, there is a dependence of x_1 on x_0 if a single modification of the input value of x_0 may result in a different result for x_1. This definition is comparable to the notion of non-interference [18]; in fact the occurrence of a dependence corresponds to the opposite of non-interference. It can also be related to the notions of secure information flow [13]. Our motivation is to investigate the origin of some (abnormal) results, which is clearly related to the information flows to the alarm location. Usual notions of slicing require more dependences to be taken into account, since they aim at collecting all parts of the program that play a role in the computation of the result (constant parts are required even if they do not affect the result; their case will be considered in Sect. 6).

Example 1 (Dependences of functions). Let $x, y \in \mathbb{X}$. Let us consider the function $\phi \in \mathfrak{Den}$ defined by $\phi(\rho) = \{\rho[y \leftarrow \rho(x)]\}$ if $\rho(b) = \mathbf{true}$ and $\phi(\rho) = \emptyset$ if $\rho(b) = \mathbf{false}$. Then, if $\rho_0 \in \mathbb{M}$, and $z \in \mathbb{X}$, $\phi(\rho_0[b \leftarrow \mathbf{false}])(z) = \emptyset \neq \phi(\rho_0[b \leftarrow \mathbf{true}])(z)$; hence, $z \overset{\phi}{\rightsquigarrow} b$. Similarly, we would show that $y \rightsquigarrow x$. Last, if $z \in \mathbb{X} \backslash \{y\}$, we could prove that $z \rightsquigarrow z$, and that ϕ has no other dependence.

The definition of dependences among variables in a control flow graph derives from the above definition and the classical abstraction of sets of traces into functions [9]. Indeed, let $l_0, l_1 \in \mathbb{L}$, $x_0, x_1 \in \mathbb{X}$. Then, we shall approximate the set of traces starting at l_0 and ending at l_1 with a function in $\mathfrak{f}_{l_0}^{l_1} \in \mathfrak{Den}$ defined by $\mathfrak{f}_{l_0}^{l_1}(\rho_0) = \{\rho_1 \mid \exists\langle(l_0, \rho_0), \ldots, (l_1, \rho_1)\rangle \in [\![s]\!]\}$. We say that s induces a dependence of (l_1, x_1) on (l_0, x_0) if and only if $(x_0, x_1) \in \mathfrak{D}_{\mathrm{f}}[\mathfrak{f}_{l_0}^{l_1}]$ (such a dependence will be denoted by $(l_1, x_1) \rightsquigarrow (l_0, x_0)$). Last, we note $\mathfrak{D}_{\mathrm{t}}[s]$ for $\{((l_0, x_0), (l_1, x_1)) \mid l_0, l_1 \in \mathbb{L},\ (x_0, x_1) \in \mathfrak{D}_{\mathrm{f}}[\mathfrak{f}_{l_0}^{l_1}]\}$ (we write $\mathfrak{Dep}_{\mathrm{t}}$ for $\mathcal{P}((\mathbb{L} \times \mathbb{X})^2)$).

Example 2 (Ex. 1 continued). Let us consider the program fragment $l_0 : \mathbf{if}(b)\{l_1 : y = x; l_2 : \ldots\}\ldots$. Then, the function $\mathfrak{f}_{l_0}^{l_2}$ corresponds to the function ϕ introduced in Ex. 1. Therefore, the set of dependences between l_0 and l_2 is $\mathfrak{D}_{\mathrm{f}}[\mathfrak{f}_{l_0}^{l_2}] = \{(b, y); (x, y)\} \cup \{(v, z) \mid z \in \mathbb{X} \setminus \{y\},\ v = z \vee v = b\}$.

In the following, we rely on this straightforward extension of the definition of dependences induced by functions into dependences induced by programs (so that we do not have to state it again). It is important to note that $\mathfrak{D}_{\mathrm{f}}[]$ is *not* monotone; in particular the greatest element of $\mathfrak{Den}$ is $(\lambda\rho.\mathbb{M})$ and induces *no* dependence. The purpose of the following subsections is to select *some* dependences that should be relevant to the problem under consideration.

2.3 Observable dependences

We consider now the problem of restricting the dependences that can be observed on a subset of traces (aka a *semantic slice*). The semantic slice is usually defined by a criterion c chosen in some domain $\mathbb{C}$; moreover, we assume a concretization function $\gamma_{\mathbb{C}} : \mathbb{C} \to \mathcal{P}(\Sigma)$ describes the meaning of semantic slicing criteria in terms of sets of traces. For instance, we may fix sets of initial and final states; hence, $\mathbb{C} = \mathcal{P}(\mathbb{S}) \times \mathcal{P}(\mathbb{S})$ and $\gamma_{\mathbb{C}}(\mathcal{I}, \mathcal{F}) = \{\langle s_0, \ldots, s_n\rangle \in \Sigma \mid s_0 \in \mathcal{I} \wedge s_n \in \mathcal{F}\}$; in the end the semantic slice (traces starting in $\mathcal{I}$ and ending in $\mathcal{F}$) boils down to $[\![s]\!] \cap \gamma_{\mathbb{C}}(\mathcal{I}, \mathcal{F})$. Other useful examples of semantic slices were introduced in [23] (input constraints and restriction to some execution patterns).

We let $\mathcal{E} \subseteq \Sigma$ denote a semantic slice and $\mathcal{E}^\sharp : \mathbb{L} \to \mathcal{P}(\mathbb{M})$ denote an "abstraction" [10] for the semantic slice $\mathcal{E}$: if $\langle \ldots, (l, \rho), \ldots\rangle \in \mathcal{E}$, then $\rho \in \mathcal{E}^\sharp(l)$. In practice, $\mathcal{E}^\sharp$ is computed by a static analyzer like ASTRÉE [5] ($\mathcal{E}^\sharp(l)$ is the

```
                               l0  if(b){
                               l1      y = x;
Initial condition (l0):        l2  }else{
   b = true                    l3      ...
                               l4  }
                               l5  ...

             (a)
```

```
Initial condition (l0):    l0  if(x > 5){
   x ∈ [0, 10]             l1      y = 1 000 ⋆ x;
   y ∈ [0, 5]              l2  }else{
   z ∈ [−4, 15]            l3      y = y + z;
Final condition (l5):      l4  }
   y ≥ 1 000               l5  ...

             (b)
```

Fig. 2: Observable dependences

concretization of the local, abstract numerical invariant at point l). Moreover, we assume that $\mathcal{E}$ satisfies a *closeness assumption*: $\langle s_0, \ldots, s_n, \ldots, s_m \rangle \in \mathcal{E} \iff \langle s_0, \ldots, s_n \rangle \in \mathcal{E} \wedge \langle s_n, \ldots, s_m \rangle \in \mathcal{E}$. This assumption is required for the derivation of computable approximations of dependences. It is satisfied for all semantic slices proposed in [23] (a little complication arises in the case of the "pattern"-based semantic slicing: in this case, the dependence analysis should use the same partitioning criteria as the static analysis that computes $\mathcal{E}^\sharp$; this case is evoked in Sect. 4).

Example 3 (Semantic slicing). Fig. 2 presents some cases of semantic slices defined by a set of initial and final states. In the case of Fig. 2(a) (similar to Ex. 2), the condition on the input b entails that only the true branch may be executed; moreover, the value of b may not change (it is equal to **true**), so we expect all dependences on (l_0, b) be removed.

Similarly, in the case of Fig. 2(b), the output condition on y may only be achieved by executions flowing through the true branch of the conditional; therefore, we expect the dependences of (l_5, y) on $(l_0, z), (l_0, y)$ not to be considered.

In the same way as in Sect. 2.2, we propose a definition for dependences induced by functions (the definition for dependences induced by a program follows straightforwardly). More precisely, we consider in the following definition the case of a function ϕ constrained by input and output conditions; the case of functions abstracting a program semantic slice is more technical but similar.

Definition 2 (Observable dependences). *Let $\phi \in \mathfrak{Den}$, $\mathcal{M}_i, \mathcal{M}_o \subseteq \mathbb{M}$, $x_0, x_1 \in \mathbb{X}$. We say that ϕ induces an* observable dependence *of x_1 on x_0 in the semantic slice $(\mathcal{M}_i, \mathcal{M}_o)$ if and only if $\exists \rho \in \mathcal{M}_i,\ v_a, v_b \in \mathcal{M}_i(x_0)$ and such that $\phi(\rho[x_0 \leftarrow v_a])(x_1) \cap \mathcal{M}_o(x_1) \neq \phi(\rho[x_0 \leftarrow v_b])(x_1) \cap \mathcal{M}_o(x_1)$. We write $x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \mapsto \mathcal{M}_o} x_0$ for such a dependence.*

Intuitively, an observable dependence is a dependence of *the original function*, which can be observed even when considering executions and values in the slice only. This notion generalizes the classical dependences presented in Def. 1: indeed, if we let $\mathcal{M}_i = \mathcal{M}_o = \mathbb{M}$, we find the same notion as in Def. 1.

Other possible definitions for observable dependences could have been chosen; however, most of them are flawed. For instance, considering the dependences of the restriction $\widetilde{\phi} : \rho \mapsto \phi(\{\rho\} \cap \mathcal{M}_i) \cap \mathcal{M}_o$ would have caused many additional dependences, with no intuitive interpretation: slicing $\phi = \lambda\rho.\{\rho\}$ with the input condition $\rho(x) = 0$ would include dependences of the form $y \overset{\widetilde{\phi}}{\rightsquigarrow} x$ for *any* variable y, which would not be meaningful. Therefore, we consider dependences of ϕ, observed on the restriction.

Example 4 (Ex. 3 continued). Let us consider the program in Fig. 2(a). There is only one possible value for b at l_0, so Def. 2 defines no dependence on (l_0, b), as expected in Ex. 3.

In the program of Fig. 2(b), the condition on the output rules out all traces going through the false branch. As a consequence, the set of dependences in the semantic slice between l_0 and l_2 is $\{(y, x)\} \cup \{(z, z) \mid z \in \mathbb{X},\ z \neq y\}$.

2.4 Abstract dependences

A second restriction consists in defining dependences among *abstractions* of the values the variables may take in the semantic slice. We consider simple abstractions only. For instance, we may wish to find out what may cause some variable to take large values (e.g., to investigate an overflow alarm), or very small values (e.g., to investigate a division by 0), or out-of-spec values (in case a user-provided specification maps variables to ranges they are supposed to live in). We let such a property be represented by an abstraction of sets of values, defined by a Galois-connection $\mathcal{P}(\mathbb{V}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} D$. The formalism of Galois-connections is powerful enough for our needs here, since we express dependences among simple abstractions only (i.e. α is always defined). We let $\mathbb{A}$ denote the set of such abstractions. An abstraction will be identified to its abstraction function, since there is no ambiguity. For instance, if k is a large scalar value, we may define $\gamma^{[k]}(\mathcal{P}_{\forall}^{[k]}) = \{v \mid |v| < k\}$, $\gamma^{[k]}(\mathcal{P}_{\exists}^{[k]}) = \{v \mid |v| \geq k\}$ and $\mathbb{D}^{[k]} = \{\bot, \mathcal{P}_{\forall}^{[k]}, \mathcal{P}_{\exists}^{[k]}, \top\}$; this abstraction allows to select which variable may take a large value. If the analyzer reports a possible overflow of x, then, we may wish to check what x depends on, and more precisely what variables may take abnormal or special (e.g., large) values, causing x to overflow; indeed, most arithmetic operators (like $+$, $-$, $\star$) tend to propagate large values in concrete executions and abstract analyses. For instance, we may want to learn what may cause y to grow above $1\,000$ in the program in Fig. 2(b) (this was the purpose of the output condition on y when defining the semantic slice) and more precisely to track other abnormal values in the computation of y. This is the goal of the following definition:

Definition 3 (Abstract dependences). *Let $\phi \in \mathfrak{Den}$, $\mathcal{M}_i, \mathcal{M}_o \subseteq \mathbb{M}$, $x_0, x_1 \in \mathbb{X}$, $\alpha_0, \alpha_1 \in \mathbb{A}$ (we write D_0, D_1 for the abstract domains corresponding to α_0, α_1). We say that ϕ induces an* abstract dependence *of (x_1, α_1) on (x_0, α_0) in the semantic slice $(\mathcal{M}_i, \mathcal{M}_o)$ if and only if $\exists \rho \in \mathcal{M}_i,\ d_a, d_b \in D_0$ and such that:*
- $\forall j \in \{a, b\}, \gamma_0(d_j) \cap \mathcal{M}_i(x_0) \neq \emptyset$;
- $\alpha_1(\phi(\rho[x_0 \leftarrow \gamma_0(d_a)])(x_1) \cap \mathcal{M}_o(x_1)) \neq \alpha_1(\phi(\rho[x_0 \leftarrow \gamma_0(d_b)])(x_1) \cap \mathcal{M}_o(x_1))$.

We write $(x_1, \alpha_1) \rightsquigarrow_{\mathcal{M}_i \Rightarrow \mathcal{M}_o} (x_0, \alpha_0)$ for such a dependence.

Intuitively, an abstract dependence is a dependence that can be observed by looking at abstractions of the values of the variables only. In particular, we can remark that the notion of abstract dependences generalizes the notion of observable dependences: if we let $\alpha_0 = \alpha_1 = \mathbf{id}$ where $\forall P \subseteq \mathbb{M}$, $\mathbf{id}(P) = P$, we define the same notion as in Def. 2. As usual, this definition is implicitly extended to dependences in programs.

Example 5 (Ex. 3 continued). We consider the dependences of (l_5, y) on the example of Fig. 2(b) again, but we wish to consider dependences involving "large" values only, i.e. we consider abstractions of the form $\alpha^{[k]}$ where $k > 1\,000$, with the above notations. Since x does not take any large value, the dependence of $(l_5, y, \alpha^{[1\,000]})$ is restricted to $(l_2, y, \alpha^{[1\,000]})$. Furthermore, in this case, the first occurrence of a large value in the program coincides with the assignment right before l_2; in this sense, following the abstract dependence allows to get an insight about where the abnormal value for y comes from.

Clearly, the approach proposed here may not lead to the actual error behind an alarm (e.g., an overflow). First, some large values may be caused by a division by small values (this case requires considering abstract dependences involving various kind of abstractions). Second an overflow may be due to a slow divergence; in this case, only the "cycle" of dependences corresponding to the diverging values will be discovered. Ideally, we would look for dependences of the form $(x_1, \alpha_1) \rightsquigarrow (x_0, \mathbf{id})$ in order to collect all dependences of a variable x_1 causing an overflow; yet this would tend to yield too many dependences. Our approach mainly aims at characterizing which variables are more likely to cause an error, by looking at the dependences that may carry abnormal (e.g. large) values first.

3 Comparing dependences

In this section, we show in what extent abstract and observable dependences are stronger forms of dependences than the standard notion presented in Def. 1.

Theorem 1 (Hierarchy of dependences). *We let $\phi \in \mathfrak{Den}$, $\mathcal{M}_i, \mathcal{M}_o \subseteq \mathbb{M}$, $x_0, x_1 \in \mathbb{X}$, $\alpha_0, \alpha_1 \in \mathbb{A}$. Then:*

- *if ϕ induces a dependence $x_1 \rightsquigarrow_{\mathcal{M}_i \mapsto \mathcal{M}_o} x_0$, and $\mathcal{M}_i', \mathcal{M}_o'$ are such that $\mathcal{M}_i \subseteq \mathcal{M}_i'$ and $\mathcal{M}_o \subseteq \mathcal{M}_o'$, then ϕ induces a dependence $x_1 \rightsquigarrow_{\mathcal{M}_i' \mapsto \mathcal{M}_o'} x_0$;*
- *if ϕ induces a dependence $(x_1, \alpha_1) \rightsquigarrow_{\mathcal{M}_i \mapsto \mathcal{M}_o} (x_0, \alpha_0)$, and α_0' is less abstract than α_0 (i.e., there exists an abstraction α_0'' such that $\alpha_0 = \alpha_0'' \circ \alpha_0'$) and α_1' is less abstract than α_1, then ϕ induces a dependence $(x_1, \alpha_1') \rightsquigarrow_{\mathcal{M}_i \mapsto \mathcal{M}_o} (x_0, \alpha_0')$ (in particular, if we let $\alpha_0 = \alpha_1 = \mathbf{id}$, then we conclude that there exists a dependence $x_1 \rightsquigarrow_{\mathcal{M}_i \mapsto \mathcal{M}_o} x_0$);*
- *in particular, if ϕ induces a dependence $(x_1, \alpha_1) \rightsquigarrow_{\mathcal{M}_i \mapsto \mathcal{M}_o} (x_0, \alpha_0)$, and if we let $\mathcal{M}_i' = \mathcal{M}_o' = \mathbb{M}$ and $\alpha_0 = \alpha_1 = \mathbf{id}$, then we conclude that there exists a dependence $x_1 \rightsquigarrow x_0$ in the sense of Def. 1.*

These properties are very intuitive: the smaller a semantic slice, the less dependences one can observe on it; similarly, the more "abstract" the abstractions we

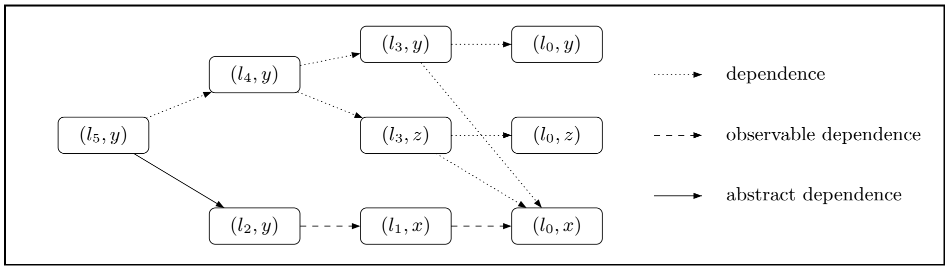

Fig. 3: Dependences from (l_5, y) in the program of Fig. 2(b)

consider, the less dependences they let observe (the abstractions may hide dependences). In particular, for any semantic slice and any abstraction, observable and abstract dependences are a subset of the "usual" dependences introduced in Def. 1. As a consequence, the abstract dependences allow to select *some* dependences, that are more likely to be useful when trying to understand the origin of (true or false) alarms; in this sense, they provide more precise information than mere dependences. We now apply these principles to the program of Fig. 2(b):

Example 6 (Ex. 5 continued). We present in Fig. 3 all possible kinds of *local* dependences (i.e.dependences on one-step transitions) collected recursively from (l_5, y). Next section discusses how to approximate dependences with such a graph. As shown in the figure, the restriction to observable dependences in a semantic slice defined by the conditions in Fig 2(a) allows to throw away the dependences induced by the **false** branch; the abstract dependences are even more restrictive, with only one abstract dependence. This dependence points to the assignment in the **true** branch where a large value is assigned to y.

4 Fixpoint-based approximation for observable dependences

At this point, we have introduced some relevant notions of dependences; yet, we need algorithms to compute them (exactly or with some approximation); the goal of this section is to provide a computable approximation for observable dependences, to compare it with existing methods and propose refinements. We generalize this techinique to the approximation of abstract dependences in the next section.

We start with a semantic-based fixpoint algorithm for approximating dependences and benefit from the semantic foundation to implement various refinements. The principle of this algorithm is comparable to existing dependence analyses [19]; yet, the advantage of our presentation is to allow for a wide variety of refinements inherited from static analysis to be formulated and proved; these refinements are described in the end of the section.

The approximation of composition: First, we propose to define an approximation for $\circ$ in $\mathfrak{Dep}_{\mathrm{f}}$. We let $\phi_0, \phi_1 \in \mathfrak{Den}$ and consider the function

$\phi = \phi_1 \circ \phi_0$. We let $\mathcal{D}_0, \mathcal{D}_1$ be over-approximations of the dependences induced by ϕ_0, ϕ_1; we try to approximate the set $\mathcal{D}$ of dependences of ϕ. Let $(x_0, x_2) \in \mathbb{X}^2$, such that $\forall x_1 \in \mathbb{X},\ (x_0, x_1) \notin \mathfrak{D}_0 \vee (x_1, x_2) \notin \mathfrak{D}_1$. We let $\rho \in \mathbb{M},\ v_a, v_b \in \mathbb{V}$, and $W = \{x_1 \in \mathbb{X} \mid \phi_0(\rho_a)(x_1) \neq \phi_0(\rho_b)(x_1)\}$ where $\forall i,\ \rho_i = \rho[x_0 \leftarrow v_i]$. We can prove by induction on the number of elements of W that $\phi_1 \circ \phi_0(\rho_a)(x_2) = \phi_1 \circ \phi_0(\rho_b)(x_2)$ (W is finite since $\mathbb{X}$ is finite). As a consequence:

Lemma 1 (Approximation of $\circ$). *With the above notations, $\mathcal{D} \subseteq \mathcal{D}_0 \boxcircle \mathcal{D}_1$, where $\boxcircle$ is the binary operator defined over $\mathfrak{Dep}_{\mathrm{f}}$ by $\mathcal{D}_0 \boxcircle \mathcal{D}_1 = \{(x_0, x_2) \in \mathbb{X}^2 \mid \exists x_1 \in \mathbb{X},\ (x_0, x_1) \in \mathcal{D}_0 \wedge (x_1, x_2) \in \mathcal{D}_1\}$. As a consequence, if $\mathfrak{D}_{\mathrm{f}}[\phi_0] \subseteq \mathcal{D}_0$ and $\mathfrak{D}_{\mathrm{f}}[\phi_1] \subseteq \mathcal{D}_1$, then $\mathfrak{D}_{\mathrm{f}}[\phi_1 \circ \phi_0] \subseteq \mathcal{D}_0 \boxcircle \mathcal{D}_1$.*

This approximation is clearly strict in general. Intuitively, the operator $\boxcircle$ provides a sound approximation for $\circ$ in $\mathfrak{Dep}_{\mathrm{f}}$. An approximation for the dependences of semantic slices can be computed in a similar way. Let $\phi_0, \phi_1 \in \mathfrak{Den}$, and $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2 \subseteq \mathbb{M}$, and $\phi = \phi_1 \circ \phi_0$. We consider the semantic slices of ϕ_0 and ϕ_1 defined respectively by $(\mathcal{M}_0, \mathcal{M}_1)$ and $(\mathcal{M}_1, \mathcal{M}_2)$; the semantic slice of the composition is $\widetilde{\phi} : \rho \mapsto \phi_1(\phi_0(\{\rho\} \cap \mathcal{M}_0) \cap \mathcal{M}_1) \cap \mathcal{M}_2$. If $\mathcal{D}_0, \mathcal{D}_1$ over-approximate the dependences of the semantic slices of ϕ_0 and of ϕ_1 respectively, then we can prove that $\mathcal{D}_0 \boxcircle \mathcal{D}_1$ over-approximates the dependences of the slice $\widetilde{\phi}$.

Fixpoint-based over-approximation of dependences: The restriction $[\![s]\!]_{[p]}$ of $[\![s]\!]$ to a path $p = l_0 \cdot l_1 \cdot \ldots \cdot l_n$ is the set of traces that follow that path (i.e. of the form $\langle (l_0, \rho_0), (l_1, \rho_1), \ldots, (l_n, \rho_n) \rangle$). We can abstract $[\![s]\!]_{[p]}$ into a function $\mathfrak{f}_{[p]} \in \mathfrak{Den}$ defined by $\mathfrak{f}_{[p]} : \rho_0 \mapsto \{\rho_n \in \mathbb{M} \mid \langle (l_0, \rho_0), (l_1, \rho_1), \ldots, (l_n, \rho_n) \rangle \in [\![s]\!]\}$; furthermore, $\mathfrak{f}_{[p]} = \delta_{l_n}^{l_{n+1}} \circ \ldots \circ \delta_{l_0}^{l_1}$ where $\forall l, l' \in \mathbb{L},\ \delta_l^{l'}(\rho) = \{\rho' \in \mathbb{M} \mid (l, \rho) \rightarrow (l', \rho')\}$ is a local semantic transformer. At this point, we can make two remarks:

- Lemma 1 provides an approximation for the dependences induced by $\mathfrak{f}_{[p]}$: $\mathfrak{D}_{\mathrm{f}}[\mathfrak{f}_{[p]}] \subseteq \mathfrak{D}_{\mathrm{f}}[\delta_{l_0}^{l_1}] \boxcircle \ldots \boxcircle \mathfrak{D}_{\mathrm{f}}[\delta_{l_n}^{l_{n+1}}]$;
- the abstraction $\mathfrak{f}_{l_0}^{l_n}$ of the traces from l_0 to l_n can be decomposed along all paths from l_0 to l_n: $\forall \rho \in \mathbb{M},\ \mathfrak{f}_{l_0}^{l_n}(\rho) = \cup\{\mathfrak{f}_{[p]}(\rho) \mid p \text{ path from } l_0 \text{ to } l_n\}$; this allows to prove that a dependence between l_0 and l_n should be observable on at least one path from l_0 to l_n.

We let $\mathcal{D}_{\mathrm{loc}} \in \mathfrak{Dep}_{\mathrm{t}}$ be an approximation of all local dependences in s: $(x, x') \in \mathfrak{D}_{\mathrm{f}}[\delta_l^{l'}] \Rightarrow ((l, x), (l', x')) \in \mathcal{D}_{\mathrm{loc}}$. An example of a rough definition for $\mathcal{D}_{\mathrm{loc}}$ is shown on Fig. 4. We deduce from the two points above the following theorem (if F is montone, we write $\mathbf{lfp}_{x_0} F$ for the least fixpoint of F, greater than x_0):

Theorem 2 (Dependences approximation). *Let $\circledast$ be the operator defined on $\mathfrak{Dep}_{\mathrm{t}}$ by $\mathcal{D}_0 \boxast \mathcal{D}_1 = \{(\nu_0, \nu_2) \mid \exists \nu_1 \in (\mathbb{L} \times \mathbb{X}),\ (\nu_0, \nu_1) \in \mathcal{D}_0 \wedge (\nu_1, \nu_2) \in \mathcal{D}_1\}$, and $\Delta = \{(\nu, \nu) \mid \nu \in (\mathbb{L} \times \mathbb{X})\}$. Then, the dependences of s are approximated by:*

$$\mathfrak{D}_{\mathrm{t}}[s] \subseteq \mathbf{lfp}_{\Delta} \mathfrak{F}_{\mathrm{dep}} \qquad \textit{where } \mathfrak{F}_{\mathrm{dep}} : \mathfrak{Dep}_{\mathrm{t}} \rightarrow \mathfrak{Dep}_{\mathrm{t}};\ \mathcal{D} \mapsto \mathcal{D} \cup \mathcal{D}_{\mathrm{loc}} \boxast \mathcal{D}$$

This theorem provides a fixpoint-based algorithm for the over-approximation of dependences. Comparable algorithms can be obtained via typing approaches [1,

$\mathit{use} : \mathbb{e} \rightarrow \mathcal{P}(\mathbb{X})$

$\mathit{use}(c) = \emptyset$

$\mathit{use}(v) = \{v\}$

$\mathit{use}(e_0 \oplus e_1) = \mathit{use}(e_0) \cup \mathit{use}(e_1)$

$\forall e \in \mathbb{e},\ \rho \in \mathbb{M},\ x \in \mathbb{X},\ v_a, v_b \in \mathbb{V},$

$[\![e]\!](\rho_a) \neq [\![e]\!](\rho_b) \Rightarrow x \in \mathit{use}(e)$

(where $\rho_i = \rho[x \leftarrow v_i]$)

(a) Deps. in expressions

assignment $l_0 : x = e; l_1 :$

$\mathfrak{D}_{\mathrm{f}}[\delta_{l_0}^{l_1}] \subseteq \mathit{use}(e) \times \{x\} \cup \{(y, y) \mid y \in \mathbb{X} \setminus \{x\}\}$

loop $l_0 : \mathbf{while}(e)\{l_1 : \ldots\}$

$\mathfrak{D}_{\mathrm{f}}[\delta_{l_0}^{l_1}] \subseteq \mathit{use}(e) \times \mathbb{X} \cup \{(y, y) \mid y \in \mathbb{X}\}$

conditional $l_0 : \mathbf{if}(e)\{l_1 : \ldots; l_2\}\mathbf{else}\{\ldots\}l_3$

$\mathfrak{D}_{\mathrm{f}}[\delta_{l_0}^{l_1}] \subseteq \{(x, y) \in \mathbb{X}^2 \mid x = y \vee x \in \mathit{use}(e)\}$

$\mathfrak{D}_{\mathrm{f}}[\delta_{l_2}^{l_3}] \subseteq \{(x, x) \mid x \in \mathbb{X}\}$

(b) Approximation for local dependences

Fig. 4: Local dependences for a simple language

2]; we prefer providing a fixpoint based definition in order to design various kinds of refinements (see the end of this section). Again this theorem also holds true in the case of observable dependences; however, the proof relies on the closeness assumption mentioned in Sect. 2.3. This hypothesis is necessary in order to prove the first point (decomposition of the dependences along a path).

Remark 1 (Control dependences). To simplify the presentation, the fixpoint algorithm of Theorem 2 does not distinguish control and data dependences like most dependence analyses do [19]. This would result in a loss of precision: for instance, in the case of a conditional $l_0 : \mathbf{if}(b)\{l_1\}l_2$, a dependence $(l_2, x) \rightsquigarrow (l_0, b)$ would be inferred for any variable x, since there is a dependence $(l_1, x) \rightsquigarrow (l_0, b)$. Yet, we can prove that, if $l, l' \in \mathbb{L}$ are such that $\forall \rho \in \mathbb{M},\ \mathfrak{f}_l^{l'}(\rho) \neq \emptyset$, if there exists a dependence $(l', x') \rightsquigarrow (l, x)$, then there exists a path from l to l' where the value of x is modified. In the above example, x is not modified between l_0 and l_1. Hence, our algorithm does not suffer the loss of precision mentioned above (our implementation does not include the fictitious dependence $(l_2, x) \rightsquigarrow (l_0, b)$).

Example 7 (Ex. 3 continued). We consider the program in Fig. 2(a) (the input condition is ignored here). Then, $\mathcal{D}_{\mathrm{loc}}$ contains the local dependences $(l_5, y) \rightsquigarrow (l_2, y)$, $(l_2, y) \rightsquigarrow (l_1, x), (l_1, x) \rightsquigarrow (l_0, b)$; the fixpoint algorithm of Theorem 2 composes these dependences together so, the dependence $(l_5, y) \rightsquigarrow (l_0, b)$ is discovered. The dependence $(l_5, y) \rightsquigarrow (l_0, x)$ is inferred in the same way. Obviously, the dependence on (l_0, b) does not hold in the semantic slice; so we show in the following how to get rid of it, by taking the properties of the semantic slice into account. Similarly, in the case of the program in Fig. 2(b), dependences through the false branch yield the dependences $(l_5, y) \rightsquigarrow (l_0, y), (l_0, z)$, which are not observable in the semantic slice.

Remark 2. Note that a sound dependence analysis for a real language (like C) requires sound aliasing information to be known: indeed, if x and y are aliased, an assignment to x creates an implicit dependence on y. Many alias analyses exist in the literature, e.g. [8, 14], so we do not develop this issue here.

Dependence graphs: In general, we are not interested in *all* the dependences of s; we only wish to track the dependences of a *criterion* c, i.e. a set of pairs (control state,variable) of interest: the set of dependences of interest is $\mathit{dep}[c] = \mathfrak{D}_{\mathrm{t}}[s] \cap ((\mathbb{L} \times \mathbb{X}) \times c)$. For instance, in the case of Fig. 2(b), we considered the dependence of $\{(l_5, y)\}$. We can approximate $\mathit{dep}[c]$ by a least-fixpoint form:

Theorem 3 (Dependences of a criterion). $\mathit{dep}[c] \subseteq \mathbf{lfp}_{(\mathbb{L}\times\mathbb{X})\times c}\mathfrak{F}_{\mathrm{dep}}$

In practice, a superset of the dependences (i.e. of $\mathcal{D}_{\mathrm{loc}}$) is collected during a linear pass; then the computation of an over-approximation of the dependence of a criterion $c \subseteq \mathbb{L} \times \mathbb{X}$ follows from Theorem 3.

Refinements: We propose now a series of refinements, in order to restrict the local dependences and their global composition so as to carry out more precise fixpoint computations. These refinements can be expressed and proved formally on the basis of Theorem 2. We consider a semantic slice $\mathcal{E}$, approximated by $\mathcal{E}^\sharp : \mathbb{L} \to \mathcal{P}(\mathbb{M})$. Among these refinements, we can cite:

- **Removal of unreachable control states:** some control state $l \in \mathbb{L}$ may be unreachable in the semantic slice. In this case, it is obvious there can be no observable dependence from or to that point. In practice, the invariant $\mathcal{E}^\sharp$ computed in the semantic slicing phase [23] provides an over-approximation of the reachable control states in the semantic slice (if l reachable, then $\mathcal{E}^\sharp(l) \neq \emptyset$); any other control state should be removed from the dependences at this point.
- **Removal of constant variables:** similarly, a variable x may be proved constant at point l in the semantic slice by the analyzer (this amounts to proving $\exists v \in \mathbb{V},\ \mathcal{E}^\sharp(l)(x) \subseteq \{v\}$); in this case, there can be no dependence to (l, x): indeed, we cannot pick up two distinct values for x at l; as a consequence any two stores ρ_a, ρ_b differing at most in the value for x generate the same transitions from this point. For instance, in case the semantic slice specifies a constant value for some input variable, any variable computed from this input only is constant, hence should be removed from the dependences.
  Note that the same simplification on the other side of the dependence does not hold: indeed, proving $\rho(x) \subseteq \{v\}$ does not rule out that $\rho(x)$ may be $\emptyset$.
- **Simplification of constant expressions:** The above principle also applies to sub-expressions, which may help reducing the local dependences induced by assignments or conditions. For instance, if we consider the assignment $x = x_0 \star x_1 + x_1 \star x_2$, where $x, x_0, x_1, x_2 \in \mathbb{X}$ and x_0, x_1 are proved constant in the semantic slice, then only the dependence on x_2 should be considered.
- **Control partitioning:** the analysis carried out in the semantic slicing may resort to some kind of trace partitioning (either control-based [22] or to distinguish execution patterns [23]); then, the same principle could be applied to the dependence analysis. In particular, this approach allows to benefit from precise abstract invariants, so it may increase the number of contexts the above refinements can be applied to (for instance, some statements may be unreachable in *some* partitions, as shown in Ex. 9).

Example 8 (Ex. 7 continued). In Fig. 2(a), the value of b at l_0 is **true** (constant value) in the semantic slice; as a result, any dependence $(l_1, v) \rightsquigarrow (l_0, b)$ is removed, so that the dependence $(l_5, y) \rightsquigarrow (l_0, b)$ does not appear in the fixpoint computation anymore.

Similarly, in the semantic slice of the program in Fig. 2(b), the false branch of the conditional is unreachable; as a result any local dependence involving l_3 or l_4 is removed from $\mathcal{D}_{\text{loc}}$; as a result, the dependences $(l_5, y) \rightsquigarrow (l_0, y), (l_0, z)$ are no longer computed.

Example 9 (Partitioning analysis). Let us consider the program $l_0 : \mathbf{if}(b)\{x_0 = y\}\mathbf{else}\{x_1 = y\}; \mathbf{if}(b')\{z = x_0\}\mathbf{else}\{z = x_1\}; l_1$ and the semantic slice collecting all executions going through the *same* branch in both **if** statements. Then, the partitioning dependence analysis infers only one dependence from (l_1, z), namely (l_0, y). The non-partitioning analysis would also include dependences on $(l_0, b), (l_0, b'), (l_0, x_1), (l_0, x_0)$. We can see that this refinement allows for global precision improvements.

5 Approximating abstract dependences

Chains of abstract dependences: All results of Sect. 4 can be generalized straightforwardly to the case of abstract dependences. In particular, Lemma 1 and Theorem 2 can be generalized, by taking the abstractions into account in the definition of $\boxdot$ and $\boxast$. Indeed, we could prove as for Lemma 1 that $((x_0, \alpha_0), (x_2, \alpha_2)) \in \mathfrak{D}_{\mathrm{f}}[\phi_1 \circ \phi_0]$ entails that there exists $(x_1, \alpha_1) \in \mathbb{X} \times \mathbb{A}$ such that $((x_0, \alpha_0), (x_1, \alpha_1)) \in \mathfrak{D}_{\mathrm{f}}[\phi_0]$ and $((x_1, \alpha_1), (x_2, \alpha_2)) \in \mathfrak{D}_{\mathrm{f}}[\phi_1]$.

However, this solution is not completely satisfactory for several reasons:

- the lattice of all abstractions of $\mathcal{P}(\mathbb{V})$ is not representable.
- the fixpoint-based expressions would lead to a rough approximation. In particular if $(l_2, x_2, \alpha_2) \overset{\phi_1}{\rightsquigarrow} (l_1, x_1, \alpha_1)$ and $(l_1, x_1, \alpha_1) \overset{\phi_0}{\rightsquigarrow} (l_0, x_0, \alpha_0)$, then a dependence $(l_2, x_2, \alpha_2) \overset{\phi_1 \circ \phi_0}{\rightsquigarrow} (l_0, x_0, \alpha_0)$ will always be added, which is overly conservative in the case of abstract dependences.
- we wish to compute sets of abstract dependences that are *immediately* relevant to the criterion; indeed, given a criterion $c = (l, x, \alpha)$, we would like to track the observable abstract dependences following immediately from c first; more complex dependences should be considered only after the simpler ones did not reveal relevant causes for the alarm under investigation. In this sense, an *under*-approximation of the abstract dependences from the criterion makes sense.

As a consequence, we introduce a notion of abstract dependence chain, which collects local abstract dependences, involving "interesting" abstractions only:

Definition 4 ($\mathbb{a}$-abstract dependence chain). *We let $\mathbb{a} \subseteq \mathbb{A}$ be a set of abstractions of interest and c be the criterion (l, x, α). An $\mathbb{a}$-*abstract dependence chain *from c is a finite sequence $(l_0, x_0, \alpha_0), \ldots, (l_n, x_n, \alpha_n)$, such that:*

1. *$\forall i,\ \alpha_i \in \mathbb{a}$,*

2. $\forall i,\ ((l_i, x_i, \alpha_i), (l_{i+1}, x_{i+1}, \alpha_{i+1})) \in \mathcal{D}_{\mathrm{loc}}$.

For instance, we may choose a family of abstractions composed of the abstractions mentioned in Sect. 2.4; e.g., we may let $\mathfrak{a} = \{\alpha^{[10^n]} \mid n \in \mathbb{N},\ n \geq 3\}$, so as to track large values.

Computation of abstract dependence chains: We need an *abstract dependence graph*, that is, an over-approximation for all abstract dependences involving abstractions in $\mathfrak{a}$ only, that occur on one-step transitions (that is, on edges of the control flow graph). The rules defined in Sect. 2.4 apply for the over-approximation of such dependences; refinements of these local dependences are considered below. In practice, the representation of the abstract dependence graph consists in a dependence graph, with labels on the edges, that approximate the abstractions the dependences they correspond to are valid for.

Once the abstract dependence graph is computed, an over-approximation of the $\mathfrak{a}$-abstract dependence chains from any criterion $c \in \mathbb{L} \times \mathbb{X} \times \mathbb{A}$ can be computed as suggested by Theorem 3, by a fixpoint-based algorithm.

Refinements: All refinements introduced in the case of (concrete) observable dependences, in Sect. 2.4 are also sound in the case of abstract dependences.

We propose a refinement that generalizes the "removal of constant variables" (Sect. 4) to abstract dependences. Let us consider $(l_0, x_0, \alpha_0), (l_1, x_1, \alpha_1) \in \mathbb{L} \times \mathbb{X} \times \mathbb{A}$. If there exists a minimal element d_0 of $D_0 \setminus \{\bot\}$ (where $\bot$ is the least element of D_0) such that $\mathcal{E}^\sharp(l_0)(x_0) \subseteq \gamma_0(d_0)$, then the abstract domain D_0 is not able to distinguish the values observed for x_0 at l_0 in the semantic slice. An obvious application of Def. 3 shows that there is no dependence $(l_1, x_1, \alpha_1) \overset{\mathcal{E}}{\rightsquigarrow} (l_0, x_0, \alpha_0)$. For instance, this refinement applies if α_0 abstracts together all "normal" (i.e., not too large) values and if all values for x_0 at point l_0 are "normal".

Example 10 (Ex. 5 continued). For instance, in the case of the program in Fig. 2(b), $x \in [0, 10]$ at point l_0; hence, if we consider $\mathfrak{a}$ as defined above, there exist no abstractions $\alpha_x, \alpha_y \in \mathfrak{a}$ such that $(l_2, y, \alpha_y) \overset{\mathcal{E}}{\rightsquigarrow} (l_1, x, \alpha_x)$. As a consequence, the only remaining abstract dependence from (l_5, y) in the semantic slice and involving abstractions in $\mathfrak{a}$ is a dependence of (l_5, y) on (l_2, y); this $\mathfrak{a}$-abstract dependence chain leads to the point where an "abnormal" value appears for the first time in the sequence of computations leading to y (see Fig. 3).

6 Slicing and case study

Slicing: Slicing [26] aims at selecting a subset of the statements of a program that may play a role in the computation of some variable x at some point l. The principle is to include in the slice any statement at point l' that may modify a variable x' such that (l, x) depends on (l', x').

The semantics of program slicing is rather subtle for several reasons:

- The notion of dependence involved in slicing is quite different to the one we considered in Sect. 2. For instance the slice of $l_0 : x = 3; l_1 : y = x; l_2$ for the criterion (l_2, y) should include the statement $l_0 : x = 3; l_1$ as well, even though (l_2, y) does not depend on (l_1, x) according to Def. 2, since x is constant at l_1.
- The usual expression of slicing correctness resorts to some kind of projection of the program semantics, which is preserved by slicing. However, the removal of non-terminating loops (or of possible sources for errors) may cause the slice to present *more* behaviors than the projection of the semantics of the source program. This issue can be solved by considering a non-standard, more concrete semantics [7], which is preserved by the transformation, yet this approach is not natural for static analysis.

As a consequence, we propose a transformation that should be more adapted to static analysis.

Smaller, non-executable slices: In [23], semantic slices approximate program executions with abstract invariants. Such an invariant together with a (subset of a) syntactic slice allow to describe even more precisely a set of program executions:

Definition 5 (Abstract slice). *An abstract slice $\mathcal{E}$ of a program s is defined by a sound invariant $\mathcal{E}^\sharp : \mathbb{L} \to \mathcal{P}(\mathbb{M})$ for $\mathcal{E}$ and a subset s' of the program statements, which is defined by the set of corresponding control states $\mathcal{L}'$.*

The semantics of a semantic slice is defined both by the program transitions (for the statements which are included in the slice) and by the abstract invariants:

Definition 6 (Abstract slice semantics). *The semantics $(\!| s' |\!)$ of the abstract slice collects all the traces $\langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle$ such that:*
- $\forall i,\ \rho_i \in \mathcal{E}^\sharp(l_i)$;
- $\forall i,\ (l_i \in \mathcal{L}' \wedge l_{i+1} \in \mathcal{L}') \Longrightarrow (l_i, \rho_i) \to (l_{i+1}, \rho_{i+1})$.

Obviously, the definition of abstract slices leaves the choice of the syntactic slice undetermined. However, the purpose of the abstract slices is to restrict to the most interesting parts the program; hence, we propose to compute abstract dependence chains and include any assignment which affect a variable in a dependence chain: this way, the slice preserves only the α-abstract dependence chains and abstract any other statement of the program into the invariants in $\mathcal{E}^\sharp$. Let us note that this notion allows to solve the two points mentioned above:
- parts of the program that are not immediately relevant to the criterion under investigation (in the sense that they do not appear in the dependences introduced in Def. 1, Def. 2 and Def. 3) do *not* need to be included into the slice anymore; instead, they can be replaced with program invariants (in the semantic slice). For instance, the assignment $l_0 : x = 3; l_1$ can be replaced with the invariant $x = 3$ at point l_1. Obviously, applying this principle to larger programs may result in huge gain in slice sizes.
- the intersection with program invariants limits the loss of precision induced by, e.g. the removal of a loop.

Example 11 (Abstract slice). Let us consider the program of Fig. 2(b), together with its input/output conditions. Fig. 3 displays the local, observable and abstract dependences that can be recursively composed when starting from (l_5, y). In case we compute an abstract slice for this program, starting from (l_5, y), we find only one $\mathfrak{a}$-abstract dependence chain (Ex. 10). As a consequence, we get the abstract slice defined by the set of control states $\mathcal{L}' = \{l_1, l_2, l_5\}$. In particular, the abstract slice contains the assignment $l_1 : y = 1000 \star x; l_2$, with the invariant $(x \in [5, 10])$, which gives a likely cause for the error.

Early implementation results and case studies: A simple abstract dependence analysis was implemented inside ASTRÉE (for tracking large values and overflows), together with an abstract slice extraction algorithm. We could run these algorithms on some 70 kLOC real world program, which we modified so as to make some computations unstable (ASTRÉE proves the absence of overflow in the original version). The static analysis by ASTRÉE takes roughly 20 minutes and uses 500 Mb on a Bi-opteron 2.2 Ghz with 8 Gb of RAM. The computation of the dependence graph (by collecting all local dependences and applying local refinements) takes 72 seconds and requires 300 Mb, on the same machine; this phase provides all data required to extract a slice from any criterion. The slice extraction computes a least fixpoint from the criterion (Theorem 3) and applies recursively local dependences; in the case of abstract dependences, this amounts to collecting $\mathfrak{a}$-abstract dependence chains. The typical slice extraction time is about 5 seconds, with low memory requirements (around 110 Mb).

The table below displays the gain in size obtained by computing abstract slices for a series of alarms (size of slices are in LOCs):

Slicing point	a_1	a_2	a_3
Classical slice	543	368	1572
Abstract slice	39	160	96

The resulting slices proved helpful for finding the direct consequences of errors like overflows; moreover, it seemed promising for deriving automatically semantic slicing criteria, which was one of the motivations for our present work. We remarked that the refinements presented in Section 4 played a great role in keeping the size of dependences down. Cyclic abstract dependence chains suggest some kind of partitioning could be done in order to isolate certain execution patterns; they also allow to restrict the part of the program to look at in order to define an adequate input for defining an error scenario, so that we envisage synthesizing input constraints in the future. Another possible use for abstract slices is to cut down the size of programs to analyze during alarm inspection sessions, by abstracting into invariants parts of the code to analyze.

7 Conclusion and Related Work

We proposed a framework for defining and computing valuable dependence information, for the understanding and refinement of static analysis results. Early experiments back-up favorably the usefulness of this approach, especially for giving good hints for the choice of semantic slicing criteria [23].

Our definition for dependences are rather related to the definition of non-interference [18] commonly used in language-based security [24]. This approach is rather different to the more traditional ways of defining dependences in program slicing, which rely on program dependence graphs [19], yet these two problems are related [2, 1]. We found that the main benefit of the "dependences as interference" definition is to allow for wide varieties of refinements for dependence analyses and extension for the definition of dependences to be stated.

In particular, our definition of abstract dependences is closely related to the notion of abstract non interference introduced in [17] in the security area, which aims at classifying program attackers as abstract-interpretations. The authors propose to compute the strongest safe attacker of a program by resolving an equation on domains by fixpoint. In our settings, the abstraction on the output is fixed by the kind of alarm being investigated; moreover, the dependence analysis should discover the variables the criterion depends on and not only for what observation. Therefore, the algorithms proposed in [17] do not apply to our goal.

Program slicing [26] is another area related to our work. Many alternative notions of slices have been proposed since the first, syntactic versions of slicing. In particular, conditioned slicing [6] (applied, e.g. in [12]) aim at extracting slices preserving *some* executions of programs, specified by, e.g. a relation on inputs. Our approach goes beyond these methods: indeed, a set of program executions defined by a semantic property (e.g. leading to an error) is characterized precisely by semantic slicing [23]; these invariants allow to refine precisely the dependences. Dynamic slicing [3, 20, 16] records states during *concrete* executions and inserts a dependence among the corresponding nodes according to a standard, rough dependence analysis, in order to produce "dynamic", non-executable slices. This approach is adapted to debugging; yet it does not allow to characterize precisely a set of executions defined by semantic constraints either.

There exist a wide variety of methods applied to error cause localization. For instance, [4] proposes to characterize transitions that *always* lead to an error in abstract models; however, this kind of approach requires enumerating the predicates and/or transitions; hence, it does not apply to ASTRÉE, due to the number of predicates in the abstract invariants (domains nearly infinite).

Debugging methods start with a *concrete* trace, which we precisely do not have, since alarms arise from abstract analyzes.

A first possible direction for future work would be to express abstract dependences involving more complicated, e.g. relational abstractions. Indeed, tracking the origin of an alarm raised in the analysis of $z = \sqrt{x + y}$ requires looking at dependences involving the property $x + y < 0$. A second challenge is to let the dependence analysis interact more closely with the forward-backward analyses carried out by the semantic slicer [23]; in particular the dependence information could give some hints about what part of the invariants to refine (after specializing the semantic slicing criteria).

Acknowledgments We deeply thank B. Blanchet and J. Feret for their comments on a preliminary version of this article.

References

1. M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 1999.
2. M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *POPL*, 1999.
3. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *PLDI*, 1990.
4. T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL*, 2003.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety Critical Software. In *PLDI*, 2003.
6. G. Canfora, A. Cimitille, and A. D. Lucia. Condition program slicing. *Information and Software Technology; Special issue on Program Slicing*, 1998.
7. R. Cartwright and M. Felleisen. The semantics of program dependence. In *PLDI*, 1989.
8. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *PLDI*, 1993.
9. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *ENTCS*, 6, 1997.
10. P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
11. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, 2005.
12. S. Danicic, D. Daoudi, C. Fox, R. Hierons, M. Harman, J. Howroyd, L. Ouarbya, and M. Ward. ConSUS: A Light-Weight Program Conditioner. *Journal of Systems and Software*, 2004.
13. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.
14. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI*, 1994.
15. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.
16. C. Fox, S. Danicic, M. Harman, and R. Hierons. ConSIT: A Conditioned Program Slicing System. *Software - Practice and Experience*, 2004.
17. R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL*, 2004.
18. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*, 1982.
19. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, 1988.
20. B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 1988.
21. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.
22. L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP*, 2005.
23. X. Rival. Understanding the origin of alarms in ASTRÉE. In *SAS*, 2005.
24. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.
25. A. Venet and G. Brat. Precise and efficient array bound checking for large embeded c programs. In *PLDI*, 2004.
26. M. Weiser. Program slicing. In *Proceeding of the Fifth International Conference on Software Engineering*, pages 439–449, 1981.