Understanding the Origin of Alarms in Astrée

Xavier Rival

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

Abstract. Static analyzers like Astrée are incomplete, hence, may produce false alarms. We propose a framework for the investigation of the alarms produced by Astrée, so as to help classifying them as *true errors* or *false alarms* that are due to the approximation inherent in the static analysis. Our approach is based on the computation of an approximation of a set of traces specified by an initial and a (set of) final state(s). Moreover, we allow for finer analyses to focus on some execution patterns or on some possible inputs. The underlying algorithms were implemented inside Astrée and used successfully to track alarms in large, critical embedded applications.

1 Introduction

The risk of failure due to software bugs is no longer considered acceptable in the case of critical applications (as in aerospace, energy, automotive systems). Therefore, sound program analyzers have been developed in the last few years, that aim at proving safety properties of critical, embedded software such as memory properties [24], absence of runtime errors [5], absence of buffer overruns [13], correctness of pointer operations [31]. These tools attempt to prove automatically the correctness of programs, even though this is not decidable; they are sound (they never claim the property of interest to hold even though it does not) and always terminate; hence, they are *incomplete*: they may produce false alarms, i.e. report not being able to prove the correctness of some critical operation even though no concrete execution of the program fails at this point.

Alarms are a major issue for end-users. Indeed, in case the analyzer reports an alarm, it could either be a false alarm or a real bug that should be fixed. Ideally, a report for a true error should come with an error scenario. Currently, the alarm investigation process in Astrée [5] mainly relies on the manual inspection of invariants, partly with a graphical interface [12]; this process turns out to be cumbersome, since even simple alarms may take days to classify.

A false alarm might either be due to an imprecision of some abstract operations involved in the analysis (e.g. the abstract join operator usually loses precision) as in Fig. 1(a) (simplified version of an alarm formerly reported by Astrée) or to a lack of expressiveness of the domain (checking the example of Fig. 1(c) requires proving a very deep arithmetic theorem). In the former case, we may expect a (semi)-automatic refinement process to prove the alarm to be

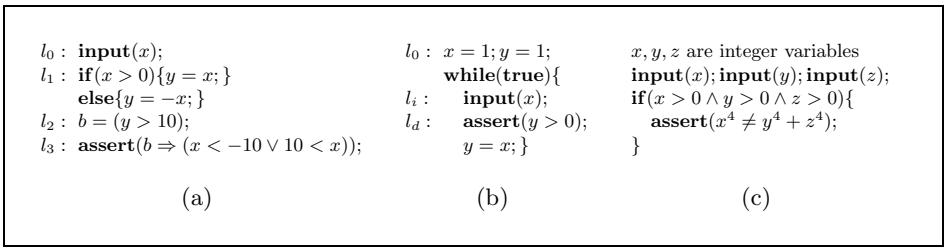


Fig. 1: three example programs

false; in the latter, the design of a refined domain can hardly be automated, so we can only hope for a refined alarm description.

Our goal is to provide some support in the alarm investigation process. We propose to resort to automatic, sound static analysis techniques so as to refine an initial static analysis into an approximation of a subset of traces that actually lead to an error. If a combination of forward and backward refining analyses allows to prove that this set is empty, we can conclude the alarm is false (as in Fig. 1(a)); otherwise, we get a refined characterization of the (possibly fictitious) erroneous traces. We propose to refine this kind of *semantic slicing* (i.e. extraction of part of the program traces) by selecting some alarm contexts (e.g. traces leading to an error after two iterations in a loop or traces constrained by some set of inputs). A similar process allows to check an error scenario, by slicing the traces reaching the alarm point in some context specified by an execution pattern and a set of inputs that are supposed to be valid: in case the analysis reveals that such conditions *always* lead to an error, the error scenario is validated (this can be achieved in the example of Fig. 1(b)); it is a counter-example.

The contribution of the paper is both theoretical and practical:

- we propose a framework for alarm inspection, based on backward analysis [8, 9], trace partitioning [25], and slicing techniques [32, 21];
- we provide encouraging practical results obtained by an implementation inside the Astrée static analyzer [4, 5, 12].

Sect. 2 introduces forward-backward analysis-based approximation (semantic slicing) of a set of traces resulting in an error. Semantic slicing refinements are introduced in Sect. 3: restriction to some execution patterns in Sect. 3.1 and to some inputs in Sect. 3.2. Sect. 4 applies syntactic slicing techniques to the reduction of the amount of code to analyze. Sect. 5 presents some practical examples. Sect. 6 concludes and reviews related work.

2 Backward Analysis

2.1 Standard notations

We restrict to a subset of C for the sake of concision. 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). Variables and values have an integer, floating-point or boolean type. We consider assignments, conditionals, loops, assertions,

and an additional $\mathbf{input}(x)$ statement, that emulates the reading of a volatile variable x: this statement replaces the value of x with a random value of the corresponding type. The grammar is given below. The control point before each statement and at the end of each block is associated to a unique label $l \in \mathbb{L}$.

$$\begin{array}{l} e\ (e \in \mathfrak{e}) ::= v\ \ (\text{where } v \in \mathbb{V})\ \mid\ x\ \ (\text{where } x \in \mathbb{X})\ \mid\ e \oplus e \\ s\ (s \in \mathfrak{s}) ::= x := e\ \ (\text{where } x \in \mathbb{X}, e \in \mathfrak{e})\ \mid\ s; s\ \mid\ \mathbf{skip} \\ \qquad\qquad \mid\ \mathbf{input}(x)\ \mid\ \mathbf{assert}(e)\ \mid\ \mathbf{if}(e)\{s\}\mathbf{else}\{s\}\ \mid\ \mathbf{while}(e)\{s\} \end{array}$$

In practice, the subset of C we analyze also includes functions, pointers, composite data-structures, all kinds of definitions, and all control structures. It excludes recursive functions, dynamic memory allocation, and destructive updates.

A state $s \in \mathbb{S}$ is either the error state Ω or a pair (l, ρ) where l is a label and $\rho \in \mathbb{M} = \mathbb{X} \to \mathbb{V}$ is a *memory state* (aka *store*). Note that we assume there are no errors in expressions; hence, the error state Ω can only be reached after a failed assertion. The semantics $[\![s]\!] = \{\langle \sigma_0, \ldots, \sigma_n \rangle \mid \forall i,\ \sigma_i \to \sigma_{i+1}\}$ of a program s is a set of sequences of states (aka traces), such that any two successive states are related by the transition relation $\to$ of the program. The relation $\to$ is defined by local rules, such as the following (the full definition is given in appendix A):

- assert statement $l_0 : \mathbf{assert}(e); l_1$: if $[\![e]\!](\rho) = \mathbf{true}$ ($[\![e]\!](\rho)$ is the result of the evaluation of e in ρ), then $(l_0, \rho) \to (l_1, \rho)$; if $[\![e]\!](\rho) = \mathbf{false}$, then $(l_0, \rho) \to \Omega$.
- assignment $l_0 : x := e; l_1$: $(l_0, \rho) \to (l_1, \rho[x \leftarrow [\![e]\!](\rho)])$ (where $[\![e]\!] \in \mathbb{M} \to \mathbb{V}$);
- input statement $l_0 : \mathbf{input}(x); l_1$: if $v \in \mathbb{V}$ has the same type as x, then $(l_0, \rho) \to (l_1, \rho[x \leftarrow v])$;

2.2 Approximation of Dangerous Traces

Dangerous states: We consider a program $P \in \mathfrak{s}$. A state σ is *dangerous* if it is not Ω and may lead to Ω in one transition step: $\sigma \to \Omega$. A *dangerous label* is a label l followed by an assertion statement ($l : \mathbf{assert}(e);$). Astrée over-approximates the set of *reachable* dangerous states; hence, our goal is to start with such an over-approximation and to make a diagnosis whether a set of concrete, dangerous states is actually reachable.

Dangerous traces: First, we are interested in real executions only, that is traces starting from an *initial* state. We let $\mathcal{I} \subseteq \mathbb{S}$ denote the set of initial states. Then, the set of real executions is $\overrightarrow{T} = \{\langle \sigma_0, \ldots, \sigma_n \rangle \in [\![P]\!] \mid \sigma_0 \in \mathcal{I}\} = \mathbf{lfp}_\emptyset \overrightarrow{F}$ where $\overrightarrow{F} : E \mapsto \{\langle \sigma \rangle \mid \sigma \in \mathcal{I}\} \cup \{\langle \sigma_0, \ldots, \sigma_n, \sigma_{n+1} \rangle \mid \langle \sigma_0, \ldots, \sigma_n \rangle \in E \wedge \sigma_n \to \sigma_{n+1}\}$ constructs execution traces forward, and $\mathbf{lfp}_X F$ is the least fixpoint of F greater than X. Second, we restrict to executions ending in a dangerous state (or at a dangerous label). We let $\mathcal{F} \subseteq \mathbb{S}$ denote the set of final states of interest. The set of executions ending in $\mathcal{F}$ is $\overleftarrow{T} = \{\langle \sigma_0, \ldots, \sigma_n \rangle \in [\![P]\!] \mid \sigma_n \in \mathcal{F}\} = \mathbf{lfp}_\emptyset \overleftarrow{F}$ where $\overleftarrow{F}$ is defined in a similar way as $\overrightarrow{F}$: $\overleftarrow{F} : E \mapsto \{\langle \sigma \rangle \mid \sigma \in \mathcal{F}\} \cup \{\langle \sigma_{-1}, \sigma_0, \ldots, \sigma_n \rangle \mid \langle \sigma_0, \ldots, \sigma_n \rangle \in E \wedge \sigma_{-1} \to \sigma_0\}$.

The set of traces of interest is $T = \overrightarrow{T} \cap \overleftarrow{T} = \mathbf{lfp}_\emptyset \overrightarrow{F} \cap \mathbf{lfp}_\emptyset \overleftarrow{F}$. It is a subset of all program behaviors $[\![P]\!]$; in this respect, we call T a *semantic slice*.

In the following, $\mathcal{F}$ may represent either $\mathcal{F}_l = \{(l,\rho) \mid \rho \in \mathbb{M}\}$ or $\mathcal{F}_{\mathfrak{D},l} = \{(l,\rho) \mid \rho \in \mathbb{M} \wedge (l,\rho) \rightarrow \Omega\}$, unless we specify explicitly; the slice T_l (resp. $T_{\mathfrak{D},l}$) defined by $\mathcal{F}_l$ (resp. $\mathcal{F}_{\mathfrak{D},l}$) collects the executions ending at label l (resp. the executions causing an error at label l). T shall represent either T_l or $T_{\mathfrak{D},l}$.

Example 1. In the code of Fig. 1(a), label l_3 is a dangerous label; the set of dangerous states for the corresponding **assert** is $\mathcal{F}_{\mathfrak{D},l_3} = \{(l_3,\rho) \mid \rho \in \mathbb{M} \wedge \rho(b) = \mathbf{true} \wedge -10 \leq \rho(x) \leq 10\}$. The set of initial states is $\mathcal{I} = \{(l_0,\rho) \mid \rho \in \mathbb{M}\}$. Clearly, this program does not cause any error: if $y > 10$ at l_2, then, either $x > 0$ and $x = y > 10$ or $x \leq 0$ and $x = -y < -10$. Hence, we wish to prove that $T_{\mathfrak{D},l_3} = \emptyset$.

Alarm inspection: Our goal is to determine whether an alarm is true or not. We may fall in either of the following cases:

Case a) **alarm proved false:** if the static analysis proves that $T_{\mathfrak{D},l} = \emptyset$, then the dangerous states in $\mathcal{F}_{\mathfrak{D},l}$ are not reachable and *the alarm is false;*

Case b) **alarm proved true:** if the static analysis proves that any trace in T_l violates the assert (i.e. all traces reaching l cause an error at this point), then *the alarm is a true error;*

Case c) **undecided case:** obviously, we may not be able to conclude to either of the previous cases; then, either *an error would occur in some cases only* (this case is considered in Sect. 3) or *the lack of conclusion* is due to a lack of expressivity of the abstract domain (the refined analysis of the alarm context should help designing a domain refinement).

Trace approximation: The approximation of sets of traces is based on an abstraction of sets of stores defined by a domain $(D_n^\sharp, \sqsubseteq)$ and a concretization function $\gamma_n : D_n^\sharp \rightarrow \mathcal{P}(\mathbb{M})$ [10]. We assume that $D_n^\sharp$ provides a widening operator ∇, approximate binary lub ($\sqcup$) and glb ($\sqcap$) operators, and a least element $\bot$, with the usual soundness assumptions. The domain for approximating sets of executions is defined by $D^\sharp = \mathbb{L} \rightarrow D_n^\sharp$ and $\gamma : (I \in D^\sharp) \mapsto \{\langle (l_0,\rho_0), \ldots, (l_n,\rho_n)\rangle \mid \forall i,\ \rho_i \in \gamma_n(I(l_i))\}$: a set of traces is approximated by local invariants, approximating the sets of stores that can be encountered at any label. We let $\mathbf{lfp}^\sharp$ denote an abstract post-fixpoint operator, derived from ∇: if $F : D \rightarrow D$, $F \circ \gamma \subseteq \gamma \circ F^\sharp$ and $X \subseteq \gamma(X^\sharp)$, then $\mathbf{lfp}_X F \subseteq \gamma(\mathbf{lfp}^\sharp_{X^\sharp} F^\sharp)$. The domain $D_n^\sharp$ is supposed to feature sound abstract operations $\overrightarrow{\mathit{assign}} : \mathbb{X} \times \mathfrak{e} \times D_n^\sharp \rightarrow D_n^\sharp$, $\mathit{guard} : \mathfrak{e} \times D_n^\sharp \rightarrow D_n^\sharp$, $\mathit{forget} : \mathbb{X} \times D_n^\sharp \rightarrow D_n^\sharp$ that soundly mimic the concrete assignment, testing of conditions, and reading of inputs (by forgetting the value of the modified variable). For instance, the soundness of guard boils down to $\forall d \in D_n^\sharp,\ \forall \rho \in \gamma_n(d),\ [\![e]\!](\rho) = \mathbf{true} \Rightarrow \rho \in \gamma_n(\mathit{guard}\,(e,d))$.

We let $\mathcal{I}^\sharp \in D^\sharp$ be a sound approximation of the traces made of just one initial state, i.e. $\{\langle\sigma\rangle \mid \sigma \in \mathcal{I}\} \subseteq \gamma(\mathcal{I}^\sharp)$; we let $\mathcal{F}^\sharp \in D^\sharp$ be a sound approximation of $\mathcal{F}$ in the same way, where $\mathcal{F}$ may be either $\mathcal{F}_l$ or $\mathcal{F}_{\mathfrak{D},l}$. The purpose of the following subsections is to approximate the semantic slice T.

2.3 Forward Analysis

We consider here the approximation of $\overrightarrow{T}$. It is well-known that a sound abstract interpreter in $D^\sharp$ can be derived from $\overrightarrow{F}$. More precisely, we can define a family

of functions $\overrightarrow{\delta}_{l,l'} : D_n^\sharp \to D_n^\sharp$ that compute the effect of any transition at the abstract level. The soundness of $\overrightarrow{\delta}_{l,l'}$ writes $\forall \rho, \rho' \in \mathbb{M},\ \forall d \in D_n^\sharp,\ \rho \in \gamma_n(d) \wedge (l, \rho) \to (l', \rho') \Rightarrow \rho' \in \gamma_n(\overrightarrow{\delta}_{l,l'}(d))$ and is a direct consequence of the soundness of the basic abstract operations. The (classical) complete definition of $\overrightarrow{\delta}_{l,l'}$ is postponed to appendix B. The forward abstract interpreter is: $\overrightarrow{F}^\sharp : D^\sharp \to D^\sharp;\ \ I \mapsto \lambda(l \in \mathbb{L}).\ \sqcup \{ \overrightarrow{\delta}_{l,l'}(I(l)) \mid l' \in \mathbb{L} \}$ (this presentation leaves the choice for an iteration strategy; Astrée uses a denotational iteration scheme, so as to keep the need for local invariant storage down). The soundness of the forward abstract interpreter is proved by standard abstract interpretation methods [10].

Theorem 1 (Soundness of the forward abstract interpreter). $T \subseteq \overrightarrow{T} \subseteq \gamma(\mathbb{I}_0)$ *where* $\mathbb{I}_0 = \mathbf{lfp}^\sharp_{\mathcal{I}^\sharp} \overrightarrow{F}^\sharp$.

Example 2 (Ex. 1 continued). A simple non relational analysis yields the invariant $b \in \{\mathbf{true}, \mathbf{false}\},\ y \geq 0$ at point l_3; the assertion is not proved safe.

2.4 The backward interpreter

We consider the refinement of the approximation $\mathbb{I}_0$ of $\overrightarrow{T}$ (hence, of T) into a better approximation, by taking into account the second fixpoint $\overleftarrow{T}$.

A straightforward way to do this would be to design a backward interpreter in the same way as we did for $\overrightarrow{F}^\sharp$ and to compute the intersection of both analyses. Yet, this approach would not be effective, mainly because in most cases, the greatest pre-conditions are not very precise, so that we would face a major loss of precision. For instance, in the case of a function call through a pointer dereference $(\star \mathtt{f})()$, the flow depends on the value of `f` *before* the call; hence, the called function cannot be determined from the state after the call and the backward analysis of such a statement with no data about the pre-condition would be very imprecise ($\star \mathtt{f}$ could be any function in the program). Examples of similar issues when analyzing assignments are given in Sect. 2.5.

Hence, the refining backward interpreter $\overleftarrow{F}^\sharp_r$ takes two elements of $D^\sharp$ as inputs: an invariant to refine and an invariant to propagate backwards. It is based on a family of backward transfer functions $\overleftarrow{\delta}_{l,l'} : D^\sharp \times D^\sharp \to D^\sharp$ maps a pre-condition to refine and a post-condition into a refined pre-condition, as stated by the soundness condition: $\forall \rho, \rho' \in \mathbb{M}, \forall d, d' \in D_n^\sharp,\ \rho \in \gamma_n(d) \wedge \rho' \in \gamma_n(d') \wedge (l, \rho) \to (l', \rho') \Rightarrow \rho \in \gamma_n(\overleftarrow{\delta}_{l,l'}(d, d'))$ (i.e. d is refined into a stronger pre-condition, by taking into account the post-condition d'). The definition for a very simple $\overleftarrow{\delta}_{l,l'}$ operator is given and discussed below. It is based on a backward abstract assignment operator $\overleftarrow{assign} : \mathbb{X} \times \mathbb{e} \times D_n^\sharp \times D_n^\sharp \to D_n^\sharp$, satisfying a similar soundness condition. The design of this operator is detailed in Sect. 2.5.

- assignment $l_0 : x := e;\ l_1$: $\overleftarrow{\delta}_{l_0,l_1}(d_0, d_1) = \overleftarrow{assign}(x, e, d_0, d_1)$
- conditional $l_0 : \mathbf{if}(e)\{l_1^t : s^t;\ l_2^t\}\mathbf{else}\{l_1^f : s^f;\ l_2^f\};\ l_3$:
  $\overleftarrow{\delta}_{l_0,l_1^t}(d_0, d_1) = d_0 \sqcap d_1$ $\overleftarrow{\delta}_{l_2^t,l_3}(d_2, d_3) = d_2 \sqcap d_3$
  $\overleftarrow{\delta}_{l_0,l_1^f}(d_0, d_1) = d_0 \sqcap d_1$ $\overleftarrow{\delta}_{l_2^f,l_3}(d_2, d_3) = d_2 \sqcap d_3$

– loop $l_0 : \mathbf{while}(e)\{l_1 : s;\ l_2\};\ l_3$:
$\overleftarrow{\delta}_{l_0,l_1}(d_0, d_1) = d_0 \sqcap d_1 \quad \overleftarrow{\delta}_{l_2,l_0}(d_2, d_0) = d_2 \sqcap d_0 \quad \overleftarrow{\delta}_{l_0,l_3}(d_0, d_3) = d_0 \sqcap d_3$
– assertion $l_0 : \mathbf{assert}(e);\ l_1$: $\overleftarrow{\delta}_{l_0,l_1}(d_0, d_1) = d_0 \sqcap d_1$
– input $l_0 : \mathbf{input}(x);\ l_1$: $\overleftarrow{\delta}_{l_0,l_1}(d_0, d_1) = d_0 \sqcap \mathit{forget}(x, d_1)$

It might be desirable to improve the precision by locally refining the computation of $\overleftarrow{\delta}_{l,l'}$. Indeed, if $\overrightarrow{\delta}_{l,l'}$ and $\overleftarrow{\delta}_{l,l'}$ are sound, then so is $\overleftarrow{\delta}^{(n)}_{l,l'} : (d, d') \mapsto d^{(n)}$, where: $d^{(0)} = \overleftarrow{\delta}_{l,l'}(d, d')$ and $\forall n \in \mathbb{N},\ d^{(n+1)} = \overleftarrow{\delta}_{l,l'}(d^{(n)}, \overrightarrow{\delta}_{l,l'}(d^{(n)}))$. This process is known as local iterations [19] and usually allows to improve the precision of backward abstract operations and condition testings. For instance, in the case of the **if** statement, we may replace $\overleftarrow{\delta}_{l_0,l_1^t}$ with $\overleftarrow{\delta}_{l_0,l_1^t}(d_0, d_1) = \mathit{guard}(e, d_0 \sqcap d_1)$. Our experience proved local iterations not extremely useful, in the presence of a refined abstract domain, able to carry out rather expressive constraints.

The backward analyzer is defined by a function $\overleftarrow{F}^\sharp_r : D^\sharp \times D^\sharp \to D^\sharp;\ \ (I, I') \mapsto \lambda(l \in \mathbb{L}).\sqcup\{\overleftarrow{\delta}_{l,l'}(I(l), I'(l')) \mid l' \in \mathbb{L}\}$ and satisfies the soundness result:

Theorem 2 (Soundness of the backward abstract interpreter). *$T \subseteq \gamma(\mathbb{I}_1)$ where $\mathbb{I}_1 = \mathbf{lfp}^\sharp_{\mathcal{F}^\sharp \sqcap \mathbb{I}_0}[\lambda(I \in D^\sharp).\overleftarrow{F}^\sharp_r(\mathbb{I}_0, I)]$ and $\mathbb{I}_0$ is the result of the forward analysis (Th. 1).*

2.5 Backward Assignment

The domain: Astrée uses a reduced product of several domains, including an interval domain, constraints among boolean variables or between boolean and scalar variables. Among the numerical relational domains, we can cite octagons [26] that express relations of the form $\pm x \pm y \leq c$ and specific domains like [15], adapted to the analysis of control command software components. Complex expressions can be abstracted prior to evaluation inside the abstract domain into interval linear forms [27]: given an abstract value $d \in D^\sharp_n$, e is abstracted into $e' = \mathbf{lin}(e, d) = \sum_k I_k \cdot x_k$ ($\cdot$ is a product operator, $\forall k$, I_k is a real interval, x_k a variable), such that $\forall \rho \in \gamma_n(d),\ [\![e]\!](\rho) \in [\![e']\!](\rho)$. Linearization allows complex (e.g. non linear) expressions to be analyzed precisely inside relational domains.

We consider now the definition of $\overleftarrow{\mathit{assign}}(x, e, d_{\text{pre}}, d_{\text{post}})$. Note that we assume that the l-value x is resolved exactly; this is always the case in the subset of C introduced in Sect. 2.1. In practice, may-assign (e.g. in the case of arrays) and assignment of pointer values are also taken into account. In the proofs below, we let $\rho \in \gamma_n(d_{\text{pre}})$; we write $v = [\![e]\!](\rho)$ and we also assume $\rho[x \leftarrow v] \in \gamma_n(d_{\text{post}})$.

Boolean transfer function: If x is a boolean variable, we let:

$$\overleftarrow{\mathit{assign}}(x, e, d_{\text{pre}}, d_{\text{post}}) = \begin{cases} \mathit{guard}(e, \mathit{forget}(x, \mathit{guard}(x, d_{\text{post}})) \sqcap d_{\text{pre}}) \\ \sqcup\ \mathit{guard}(\neg e, \mathit{forget}(x, \mathit{guard}(\neg x, d_{\text{post}})) \sqcap d_{\text{pre}}) \end{cases}$$

Indeed, let us assume $v = \mathbf{true}$. Then $\rho \in \gamma_n(\mathit{forget}(x, \mathit{guard}(x, d_{\text{post}})))$, due to the hypothesis on $\rho[x \leftarrow \mathbf{true}]$. Moreover, $[\![e]\!](\rho) = \mathbf{true}$, so $\rho \in \gamma_n(\mathit{guard}(e, \mathit{forget}(x, \mathit{guard}(x, d_{\text{post}}))))$, which shows the soundness of the above definition.

Arithmetic backward transfer function: Let us assume now that x has scalar type, e.g. floating point. We let $\mathbf{lin}(e,d) = \sum_k I_k \cdot x_k$ be an interval linear form of e. We consider the derivation of new octagon and interval constraints:

- the octagon domain provides backward assignment and guard abstract transfer functions for interval linear form expressions [28];
- the interval information in d_{pre} is refined as follows: we let $\mathfrak{I}_x^{\text{pre}}$ (resp. $\mathfrak{I}_x^{\text{post}}$) denote the interval information about x in d_{pre} (resp. d_{post}) and we compute a refined interval information $\mathfrak{I}_{x_j}^{\text{ref}}$ for x_j. The soundness of linearization implies that $v \in (\sum_{k\neq j} I_k \cdot \mathfrak{I}_{x_k}^{\text{pre}}) + I_j \cdot \rho(x_j)$; hence, if $0 \notin I_j$, $\rho(x_j) \in (v - (\sum_{k\neq j} I_k \cdot \mathfrak{I}_{x_k}^{\text{pre}}))/I_j \subseteq (\mathfrak{I}_x^{\text{post}} - (\sum_{k\neq j} I_k \cdot \mathfrak{I}_{x_k}^{\text{pre}}))/I_j$, so that the definition $\mathfrak{I}_{x_j}^{\text{ref}} = (\mathfrak{I}_x^{\text{post}} - (\sum_{k\neq j} I_k \cdot \mathfrak{I}_{x_k}^{\text{pre}}))/I_j$ is correct. These refined intervals are computed with a floating point-based approximation of the semantics of linear interval forms defined in terms of real numbers [27].

Note that the linearization should be computed using d_{pre}: using d_{post} would be unsound, since the value of the assigned variable changed; d_{pre} is also most useful to get the interval information *before* the assignment; hence, the first argument of $\overleftarrow{assign}$ is crucial to compute a precise and sound d_{pre}.

Example 3 (Backward assignment for intervals). We consider the assignment $x := y \cdot x + z$, $d_{\text{pre}} = \{x \geq 0,\ y \in [1,2],\ z \in [1,2], \ldots\}$, $d_{\text{post}} = \{x \in [3,4], \ldots\}$. Linearization converts it into $x := [1,2] \cdot x + z$; the backward assignment refines the range for x into $[0.5, 3]$. Local iteration would not improve the precision.

2.6 Iteration strategies

Iterative refining process: At the concrete level, T could be defined as the intersection of two independent fixpoints. However, at the abstract level, the invariant $\mathbb{I}_1$ obtained after one forward analysis and one backward analysis might be refined by further analyses. For instance, in case the backward analysis reveals that no trace is going through the true branch of a conditional $l : \mathbf{if}(e)\{s_t\}\mathbf{else}\{s_f\}; l' : s'$, a refining forward analysis from $\mathbb{I}_1$ may refine the local invariants inside s', since the possible imprecision due to the least upper bound at l' no longer occurs. Note that a further backward analysis would likely improve the results inside s_f also.

Therefore, we propose to define a refining forward analysis and to iterate the refining forward-backward process [8, 11]. The *refining forward analyzer* $\overrightarrow{F}_r^\sharp : D^\sharp \times D^\sharp \to D^\sharp$ is based on the forward analyzer and refines its first argument as the backward analyzer: $\overrightarrow{F}_r^\sharp : (I, I') \mapsto \lambda(l' \in \mathbb{L}).\ \sqcup \{\ \overrightarrow{\delta}_{l,l}(I(l)) \sqcap I(l') \mid l \in \mathbb{L}\}$.

The *refining sequence* $(\mathbb{I}_n)_{n\in\mathbb{N}}$ is defined by:

- $\mathbb{I}_0$ has been defined in Th. 1 by $\mathbb{I}_0 = \mathbf{lfp}^\sharp_{\mathcal{I}^\sharp} \overrightarrow{F}^\sharp$;
- if $n \geq 0$, $\mathbb{I}_{2n+1} = \mathbf{lfp}^\sharp_{\mathcal{F}^\sharp \sqcap \mathbb{I}_{2n}}[\lambda(I \in D^\sharp).\overleftarrow{F}_r^\sharp(\mathbb{I}_{2n}, I)]$ (akin to $\mathbb{I}_1$, see Th. 2);
- if $n \geq 0$, $\mathbb{I}_{2n+2} = \mathbf{lfp}^\sharp_{\mathcal{I}^\sharp \sqcap \mathbb{I}_{2n+1}}[\lambda(I \in D^\sharp).\overrightarrow{F}_r^\sharp(\mathbb{I}_{2n+1}, I)]$.

Theorem 3 (Soundness of the forward-backward refinement). *The above process is sound:* $\forall n \in \mathbb{N},\ T \subseteq \gamma(\mathbb{I}_n)$.

The proof is done by induction; it is similar to [11, Chap. 6]. Note that, given $\mathcal{I}^\sharp$ and $\mathcal{F}^\sharp$, the sequence of refined invariants is obtained *fully automatically*.

Example 4 (Ex. 1 continued: refined analysis). We let $\mathcal{F}^\sharp$ be $x \in [-10, 10] \wedge y \geq 0 \wedge b = \mathbf{true}$; clearly $\mathcal{F}_{\mathfrak{D},l_3} \subseteq \gamma_n(\mathcal{F}^\sharp)$. The table below shows the result of the first two refining iterations, using a non relational abstraction:

label	$\mathbb{I}_0$	$\mathbb{I}_1$	$\mathbb{I}_2$
l_1	$\top$	$\bot$	$\bot$
l_2	$y \geq 0$	$-10 < x < 10 \wedge y \geq 10$	$\bot$
l_3	$y \geq 0 \wedge b \in \{\mathbf{true}, \mathbf{false}\}$	$-10 < x < 10 \wedge y \geq 10 \wedge b = \mathbf{true}$	$\bot$

$T_{\mathfrak{D},l_3} \subseteq \gamma(\mathbb{I}_2)$; hence, $T_{\mathfrak{D},l_3} = \emptyset$: the second refining iteration proves the correctness of the program, i.e. the alarm was false (Sect. 2.2, Case a).

Local iterations: The above refinement process is not optimal from the efficiency point of view. In the case of the **if** statement considered above, it amounts to completing the backward analysis of the *whole* program before doing a new forward analysis so as to refine the invariant at label l. We might want to do *local iterations*, that is carrying out forward and backward local analysis steps in a single iteration phase. In practice, we found that the refinement process done with an expressive abstract domain (like the domain present in Astrée) does not require much local iterations. Carrying out iterative refinements on large blocks of code (e.g. functions) was a more efficient strategy.

Implementation of the interpreters: The forward analyzer Astrée is written as a function that inputs a statement and an abstract pre-condition and returns an abstract post-condition; it is defined in denotational style, recursively on the syntax. The export of invariants is optional and one may choose the labels local invariants are exported at. The refining forward analyzer is based on the latter; a parameter just forces it to compute greater lower bounds with a previously computed invariant. The backward analyzer is very similar (same layout, same iteration strategy).

3 Specifying Alarm Contexts

Sect. 2 described the forward-backward analysis involved in the approximation of the set of "dangerous traces". Yet, it does not solve the following issues:

– if we analyze backwards a statement $\mathbf{while}(e)\{l : \mathbf{assert}(e); \ldots\}$, the backward interpreter computes a least-fixpoint on the loop; at the end of the process the invariant at l approximates not only the states right before an error occurs but also the states encountered one, two, or many iterations before, which results in a massive loss of precision at l;

– after refinement of the invariants, we may have the intuition that $T_{\mathfrak{D},l} \neq \emptyset$; should that case arise, we would like to envisage and check an "error scenario", which needs to be defined.

Example 5. We consider the example shown on Fig. 1(b) along this section. Clearly, this program may fail: it may read a negative input; at the next iteration,

y is negative, which causes the assertion to fail. However, if the input is always positive, it does not fail. Last, note that it will not fail in the first iteration. The attempt to determine the alarm raised after computing $\mathbb{I}_0$ using a simple interval analysis leaves us in the undetermined case (Sect. 2.2, Case c).

3.1 Restriction to an execution pattern

We propose to extend the semantic slicing introduced in Sect. 2, by specifying some *execution patterns* in addition to the initial and final states: for instance, we may restrict a slice to the executions that cause an error after at least two iterations of the main loop and distinguish the states encountered in the last two iterations when approximating this slice. In practice, we resort to some kind of trace partitioning technique, that fits in the framework of [25].

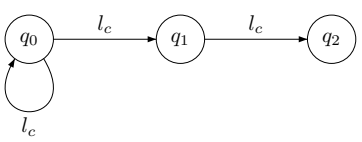
Restriction to a pattern: We extend the syntax of the language presented in Sect. 2.1 with a statement $l_0 : \mathbf{cnt}; l_1$. Tne new semantics should keep track of the order such statements are executed in. We propose to abstract this order.

Our approach involves the choice of an *automaton* $(\mathbb{Q}, \rightsquigarrow)$, where $\mathbb{Q}$ is a finite set of states and $(\rightsquigarrow) \subseteq \mathbb{Q} \times \mathbb{L} \times \mathbb{Q}$ is a transition relation (we write $q \overset{l}{\rightsquigarrow} q'$ for $(q, l, q') \in (\rightsquigarrow)$). The labels are replaced with pairs $(l, q) \in \mathbb{L} \times \mathbb{Q}$ in the definition of the concrete semantics: we replace $\mathbb{L}$ with $\mathbb{L}_p = \mathbb{L} \times \mathbb{Q}$; $\mathbb{S}$ with $\mathbb{S}_p = \mathbb{L}_p \times \mathbb{M} \cup \{\Omega\}$. The new, partitioned semantics $[\![P]\!]_p$ is defined similarly to $[\![P]\!]$, using the new transition relation $(\rightarrow_p) \subseteq \mathbb{S}_p \times \mathbb{S}_p$ instead of $(\rightarrow)$:

- case of $l_0 : \mathbf{cnt}; l_1$: if $q_0 \overset{l_0}{\rightsquigarrow} q_1$, then $\forall \rho \in \mathbb{M},\ ((l_0, q_0), \rho) \rightarrow_p ((l_1, q_1), \rho)$;
- case of any other transition (defined in appendix A):
  if $(l_0, \rho_0) \rightarrow (l_1, \rho_1)$, then $((l_0, q), \rho_0) \rightarrow_p ((l_1, q), \rho_1)$;
  if $(l_0, \rho_0) \rightarrow \Omega$, then $((l_0, q), \rho_0) \rightarrow_p \Omega$.

The *execution pattern* defined by a pair of states (q, q') is $\gamma_{\mathbb{Q}}(q, q') = \{\langle((l_0, q_0), \rho_0), \ldots, ((l_n, q_n), \rho_n)\rangle \mid q_0 = q \wedge q_n = q'\}$.

Example 6 (Ex. 5 continued). We insert a **cnt** statement in the loop and consider the automaton $\mathcal{Q}$ below. Then, $\gamma_{\mathbb{Q}}(q_0, q_2)$ specifies all the traces reaching the dangerous label at the iteration n where $n \geq 2$ and distinguishes the last two iterations (q_1 stands for iteration $n-1$; q_2 stands for iteration n). The automaton allows us to restrict to the executions that spend more than one iteration in the loop (hence, that may cause an error).




```
l0 :  x = 1; y = 1;
      while(true){
lc :    cnt;
li :    input(x);
ld :    assert(y > 0);
        y = x; }
```

We write $\pi_{\mathbb{L}} : \mathbb{L}_p \rightarrow \mathbb{L}$ (resp. $\pi_{\mathbb{S}} : \mathbb{S}_p \rightarrow \mathbb{M}, \pi : \mathbb{S}_p^\star \rightarrow \mathbb{S}^\star$) for the erasure function that removes the elements of $\mathbb{Q}$ in labels (resp. stores, traces); we let π also be defined for sets of traces. If $\tau \in [\![P]\!]_p$, then $\pi(\tau) \in [\![P]\!]$ (proof obvious).

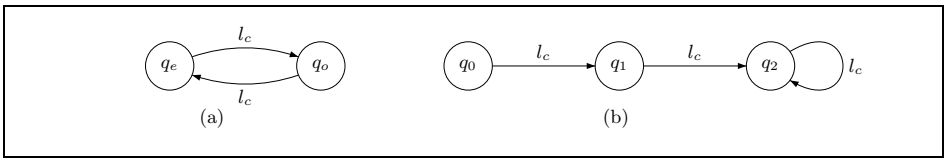


Fig. 2: Automata specifying trace patterns

Refining the semantic slice: We also need to extend $\mathcal{I}$ and $\mathcal{F}$. Let $q_i, q_f \in \mathbb{Q}$. We define $\mathcal{I}_p = \{((l, q_i), \rho) \mid (l, \rho) \in \mathcal{I}\}$ and $\mathcal{F}_p = \{((l, q_f), \rho) \mid (l, \rho) \in \mathcal{F}\}$. The automaton $(\mathbb{Q}, \rightsquigarrow)$ and the states q_i, q_f are currently chosen by the user so as to specify some set of execution paths and to specialize even more the semantic slice T; the automatic selection of refinements is left as future work (see discussion in Sect. 5). Other choices for $\mathcal{I}$ or $\mathcal{F}$, involving several states in the automaton are possible (the extension is easy). The definition of T_p from $\mathcal{I}_p, \mathcal{F}_p$ is similar to the definition of T from $\mathcal{I}, \mathcal{F}$ (Sect. 2.2). It satisfies the following property, which clearly shows that it is restricted to the execution pattern defined by q_i, q_f:

$$\pi(T_p) = T \cap \pi(\gamma_{\mathbb{Q}}(q_i, q_f))$$

Approximation of the semantic slice: We replace $D^\sharp = \mathbb{L} \to D^\sharp_n$ with the partitioning domain $D^\sharp_p = \mathbb{L} \times \mathbb{Q} \to D^\sharp_n$; we let $\gamma_p : D^\sharp_p \to \mathcal{P}(\mathbb{S}_p), I \mapsto \{\langle((l_0, q_0), \rho_0), \ldots, ((l_n, q_n), \rho_n)\rangle \mid \forall i,\ \rho_i \in \gamma_n(I(l_i, q_i))\}$. The definition of forward and backward abstract interpreters and of the sequence of refined invariants $\mathbb{I}^p_n$ follows the steps of Sect. 2, with extended definitions for $\overrightarrow{\delta}_{l,l'}$, $\overleftarrow{\delta}_{l,l'}$:

- Case of $l_0 : \mathbf{cnt}; l_1$:
  - forward analysis: if $q_0 \overset{l_0}{\rightsquigarrow} q_1$, $\overrightarrow{\delta}_{(l_0,q_0),(l_1,q_1)}(d) = d$;
  - backward analysis: if $q_0 \overset{l_0}{\rightsquigarrow} q_1$, $\overleftarrow{\delta}_{(l_0,q_0),(l_1,q_1)}(d, d') = d \sqcap d'$;
- Case of other statements: $\overrightarrow{\delta}_{(l_0,q),(l_1,q)}$ (resp. $\overleftarrow{\delta}_{(l_0,q),(l_1,q)}$) is defined like $\overrightarrow{\delta}_{l_0,l_1}$ (resp. $\overleftarrow{\delta}_{l_0,l_1}$) in Sect. 2.

Theorem 4 (Soundness). *The static analysis of the partitioned system leads to a sequence of sound invariants:* $\forall n \in \mathbb{N},\ T_p \subseteq \gamma_p(\mathbb{I}^p_n)$.

Proof: with respect to $[\![P]\!]_p$.

Example 7 (Execution patterns). Many patterns can be encoded easily (we assume that the program is of the form $\mathbf{while}(e)\{l_c : \mathbf{cnt}; \ldots\}$):

– On Fig. 2(a), q_e and q_o correspond to even and odd iteration numbers: $\gamma_{\mathbb{Q}}(q_e, q_o)$ slices the traces iterating the loop an even number of times; it also helps distinguishing states reached after an odd (resp. even) number of iterations;

– On Fig. 2(b), $\gamma_{\mathbb{Q}}(q_0, q_1)$ corresponds to the first iteration, $\gamma_{\mathbb{Q}}(q_0, q_2)$ to the n-th iteration ($n \geq 2$).

Example 8 (Ex. 5 continued).

– The refinement of $T_{\mathfrak{D},(l_d,q_1)}$ with the automaton of Fig. 2(b) and the pattern $q_i = q_0, q_f = q_1$ shows that no error may happen in the first iteration;

– Similarly, the refinement of $T_{\mathfrak{D},(l_d,q_2)}$ with the automaton $\mathcal{Q}$ (Ex. 6) gives some intuition about the traces that cause an error: at (l_d, q_2), we get $y \leq 0$; at (l_d, q_1), we get $x \leq 0$, which suggests the input of x should be negative one iteration before the error. We wish now to verify this error scenario.

Remarks: Note that the choice of an automaton with only one state q and such that for any statement l : **cnt**, $q \overset{l}{\rightsquigarrow} q$ results in the same analyses as in Sect. 2.

The trace partitioning presented in this section runs on the top of the one described in [25]; the latter aims at computing more precise invariants thanks to delayed merges of flows (e.g. out of **while** or **if** statements). Our goal here is to extract some execution patterns and to refine the corresponding invariants.

3.2 Restriction to a set of inputs

We now consider the slices defined by constraining the inputs; for instance, this may allow to show that this input always leads to an error.

Specification of inputs: We let $\mathbb{L}_{\text{in}}$ denote the set of **input** statements labels: $\mathbb{L}_{\text{in}} = \{l \in \mathbb{L} \mid l : \textbf{input}(x_l)\}$. An *input specification* is a function $\nu : \mathbb{L}_p \to \mathcal{P}(\mathbb{V})$, mapping a label to the set of values that may be read at this point. The definition of ν over the partitioned labels allows to select different inputs for different execution contexts (corresponding to different states in the automaton introduced in Sect. 3.1) at the same label. The denotation of the input function ν is the set of traces $\gamma_{\mathbb{V}}(\nu) = \{\langle((l_0, q_0), \rho_0), \ldots, ((l_n, q_n), \rho_n)\rangle \mid \forall i, l_i \in \mathbb{L}_{\text{in}} \Rightarrow \rho_{i+1}(x_{l_i}) \in \nu(l_i, q_i)\}$: such traces satisfy the property that reading an input at label (l, q) yields a value in $\nu(l, q)$.

Refining the semantic slice: The semantic slice constrained to ν is:

$$T_v = T_p \cap \gamma_{\mathbb{V}}(\nu) = T \cap \gamma_{\mathbb{Q}}(q_i, q_f) \cap \gamma_{\mathbb{V}}(\nu) \ .$$

Approximation of the semantic slice: The only modification required to take into account the input specification concerns the rule for the l_0 : **input**$(x); l_1$ statement. In this case, we let $\overrightarrow{\delta}_{(l_0,q),(l_1,q)}(d) = \mathit{forget}(x, d) \sqcap \nu^\sharp(l_0, q)$ where $\nu^\sharp(l_0, q)$ is a sound approximation of $\nu(l_0, q)$: $\{\rho \in \mathbb{M} \mid \rho(x) \in \nu(l_0, q)\} \subseteq \gamma_n(\nu^\sharp(l_0))\}$. The case of the backward analysis requires no modification.

Theorem 5 (Soundness). *The resulting abstract interpreters are sound and compute a sequence of sound refined invariants:* $\forall n \in \mathbb{N}$, $T_v \subseteq \gamma_p(\mathbb{I}_n^p)$.

The proof follows the definition of a variation $[\![P]\!]_v$ of the concrete semantics $[\![P]\!]_p$: $[\![P]\!]_v$ is obtained from $\rightarrow_{p,\nu}$ just as $[\![P]\!]_p$ from $\rightarrow_p$, where $\rightarrow_{p,\nu}$ is the transition relation constrained by the input function ν. The only modification in the definition of $\rightarrow_{p,\nu}$ comes from the case of the l : **input**(x_l); l' statement: $((l, q), \rho) \rightarrow ((l', q), \rho[x \leftarrow v])$ where $v \in \nu(l, q)$.

Example 9 (Ex. 5 continued). Let us consider the input specification $\nu(l_i, q_1) = -1$. Then, $\mathbb{I}_0$ shows that $y = -1$ at point (l_d, q_2) (the interval analysis proves this property). Hence, the automaton $\mathcal{Q}$ and the input specification ν define a

valid error scenario: any execution iterating the loop n times and such that the value read for x during the $(n-1)$th iteration is -1 will result in an error. Such situations are feasible, so the static analysis showed a real bug in the program (Sect. 2.2, Case b).

Example 10 (Ex. 7 continued). The automaton of Fig. 2(a) allows to specify a cyclic input; the automaton of Fig. 2(b) allows to isolate initialization inputs read during the first iteration and inputs read at iteration n for $n \geq 2$.

Currently, the function $\nu^\sharp$ should be provided by the user; further work should allow to synthesize an input specification $\nu^\sharp$ exhibiting an error.

4 Slicing

The approach followed in the definition of the backward abstract interpreter in Sect. 2.4 and 2.6 is not completely satisfactory for several reasons:

- it requires each analysis to save a local invariant at each label, i.e. at each statement, which would result in a dramatic memory cost;
- it leads to the forward-backward analysis of the whole program, which would result in rather long execution times due to the analysis of the *full* program, even if only part of the program is relevant to the alarm to investigate.

Therefore, we propose to use regular, syntactic slicing techniques [32, 21] so as to restrict the amount of code to apply the refining analyses to. We use the notations of Sect. 2 for the sake of simplicity (even though mixing slicing techniques and the methods introduced in Sect. 3 does not incur any major issue).

We assume a program s is given, that contains a statement l_d : **assert**(e). We write **use**(e) for the set of variables that appear in expression e.

Slicing: A slicing criterion is a set $\mathcal{C} \subseteq \mathbb{L} \times \mathbb{X}$ of pairs made of a label and a variable ; it specifies a set of variables we wish to observe the value of, at some point. A typical choice is $\mathcal{C} = \{(l_d, x) \mid x \in \textbf{use}(e)\}$. A simple way of computing the slice $s' = \mathcal{S}\langle\mathcal{C}, s\rangle$ is sketched here:

- a dependence analysis (e.g. based on the pre-computation of a dependence graph [21]) closes the criterion $\mathcal{C}$ into $\overline{\mathcal{C}}$ by taking into account any induced dependence. For instance:
  - $l_0 : x := e; l_1$ defines x from the value of the variables in e at l_0; hence, for any $y \in \textbf{use}(e)$, (x, l_1) depends on (y, l_0);
  - l_0 : **input**$(x); l_1$ defines x from no other variable; hence (x, l_1) depends on nothing;
  - **if** and **while** statements carry dependences through branches and add dependences due to conditions; for instance if $y \in \textbf{use}(e)$ and s_0 defines x, then l : **if**$(e)\{\ldots l_0 : s_0 \ldots\}$ induces a dependence of (x, l_0) on (y, l);
- the slicing transformation extracts from s any statement $l_0 : s_0$ defining a variable x and such that $(x, l_0) \in \overline{\mathcal{C}}$, the original assertion statement and any **if** or **while** statement containing them.

Analysis of slices: We assume that $\mathcal{I}, \mathcal{F}, T$ are defined as in Sect. 2.2; we let $\mathcal{I}', \mathcal{F}', T'$ be defined similarly for the slice s'. The slicing transformation is sound:

Theorem 6 (Soundness of slicing). *The observation of $\overline{\mathcal{C}}$ on s' approximates those on s, i.e., for any state $(l, \rho) \in \mathbb{S}$ encountered in T, there exists a store ρ' such that (l, ρ') is encountered in T' and $\forall x \in \mathbb{K}$, $(l, x) \in \overline{\mathcal{C}} \Longrightarrow \rho(x) = \rho'(x)$.*

The approximation stated in Th. 6 is strict in general, because slicing may remove causes for non-termination or errors, hence, introduce strictly more behaviors in the statements after **while** or **assert** statements. The main consequence of this result is that analyzing the slice s' instead of the original program s results in an over-approximation of the behavior of s when restricting to $\overline{\mathcal{C}}$.

Beyond the restriction of the size of the code to analyze, an advantage of considering the slice defined by the l_d : **assert**(e) statement is that most of the remaining statements and variables are relevant to the observation at label l_d, i.e. to the alarm under investigation. Forms of slicing to cut down the complexity of further static analyses were proposed, e.g. in [1].

Reducing the size of slices: If $\mathbb{I}_n(l) = \bot$, then $\mathbb{I}_n$ proves that the statement at label l is not relevant to the semantic slice of interest T; hence, this statement can be safely removed from the slice and its dependences thrown away, which allows to reduce even more the size of the syntactic slice. Such a transformation preserves the soundness and should speed up the computation of $\mathbb{I}_{n+k}$ $(k \geq 1)$.

Approximation of slices: Slicing should reduce the programs to analyze to a reasonable size; however, even the slices extracted from some **assert** statements may have prohibitive sizes, when extracted from very large programs, with long, cyclic dependence chains. Thus, we propose to do "aggressive slicing" and to approximate any removed statement in a sound manner during the analysis.

For instance, let us consider the forward analysis of a statement $l_0 : x := e; l_1$ (that should be extracted in the slice). As seen in Sect. 2, the forward abstract transfer function for this statement is $\overrightarrow{\delta}_{l_0,l_1} : d \mapsto \overrightarrow{assign}(x, e, d)$. The aggressive slicing of this statement consists in replacing the previous definition of $\overrightarrow{\delta}_{l_0,l_1}$ with the following: $\overrightarrow{\delta}_{l_0,l_1} : d \mapsto forget(x, d)$. Observe that this is sound: $d \mapsto forget(x, d)$ approximates all the concrete transitions defined by the assignment; hence, this new definition for $\overrightarrow{\delta}_{l_0,l_1}$ leads to a sound forward abstract interpreter (the soundness result of Th. 1 is preserved); similar results are achieved for the forward and backward refining analyzers (Th. 2 and Th. 3).

The advantage is that this form of slicing actually does not require to modify the program to analyze; indeed the analysis can be carried out on the original program; the only difference with the analyses proposed in Sect. 2, is that the transfer functions for some statements are replaced with $forget(x, .)$ operators.

Among the possible strategies to reduce the size of "aggressive slices", we can cite the limiting of dependency chains, the restriction to a given subset of variables or the elimination of loop-carried dependences: these approaches lead to an under-approximation $\widehat{\mathcal{C}}$ of the dependences induced by $\mathcal{C}$ $(\mathcal{C} \subseteq \widehat{\mathcal{C}} \subseteq \overline{\mathcal{C}})$.

Semantic criteria are also being investigated (we may wish to extract only the parts of the program backward analysis is able to refine the invariants of).

5 Case Studies

A typical alarm investigation session proceeds as follows:

1. do a forward analysis, determine a superset of the possible errors (Th. 1);
2. choose an alarm to investigate; restrict to a slice including the alarm point;
3. define $\mathcal{I}^\sharp, \mathcal{F}^\sharp$, attempt to prove the alarm wrong with forward-backward refinement (Th. 3), otherwise a more precise alarm context slice is found;
4. in case of failure, specialize the alarm context (Sect. 3.1);
5. in case no attempt to get the analyzer to prove $T_{\mathfrak{D},l} = \emptyset$ succeeds, then attempt to prove the alarm true by choosing a set of inputs (Sect. 3.2).

Application to a family of programs: We applied our methodology to the alarms raised by ASTRÉE on a series of 3 early development versions of some critical embedded programs (bugs were not unlikely in the development versions).

Size of the C code (lines)	67 500	233 000	412 000
Number of functions	650	1900	2900
Analysis time ($\mathbb{I}_0$) in sec.	1 300	16 200	37 500
Number of alarms	4	1	0
Alarm names	a_1, a_2, a_3, a_4	a_5	-

Slicing (Sect. 4) showed that a_2 (resp. a_4) is a direct consequence of a_1 (resp. a_3); hence, we restricted to the investigation of a_1, a_3 and a_5. The computation of a semantic slice for the corresponding dangerous states on the slices revealed rather informative conditions on the inputs. Specializing some inputs and carrying out a new, forward analysis *allowed to prove the alarms true*, thanks to an input specification as in Ex. 10. The table below provides some data about the process: the number of input constraints is the number of points an input constraint had to be specified for (Sect. 3.2); the number of execution patterns corresponds to the number of automata we considered (Sect. 3.1). The size of the slices (number of lines, functions and variables) involved in the alarms show that a_1, a_3 were rather subtile; a_5 was much simpler. The number of additional constraints generated during the forward-backward refinement is rather difficult to express simply due to the trace partitioning, and to the use of sophisticated numerical domains; we can only mention that it is much higher than the number of variables or of program points. One forward-backward iteration necessitates a reasonable amount of resources for these slices (up to 1 min., 80 Mb).

Alarm	a_1	a_3	a_5
Size of the slice (lines)	1280	4096	244
Number of functions in the slice	29	115	8
Number of variables in the slice	215	883	30
including: int, bool, float variables	$15, 60, 146$	$122, 553, 208$	$7, 11, 23$
Execution patterns	2	2	2
Input constraints	4	4	2

The only manual step is the choice of adequate execution patterns and of constraints on inputs, so as to get an error scenario; in all the above cases, these numbers are very low, which shows the amount of work for the user is very reasonable: only 4 inputs had to be chosen in the most complicated case (a_3). However each of these choices had to be made carefully, with respect to complex conditions on bit-fields and arithmetic values. The choices for the execution patterns to examine only required considering very few simple automata (similar to unrolling of loops, akin to Fig. 2(b) and Ex. 6), so that the selection of execution patterns should be easy to automatize.

All alarms found involve intricate floating point computations. For instance, a_5 is due to a mis-use of (interpolated) trigonometric functions, leading to a possibly negative result, causing a square root computation to fail.

Resolution of a former alarm: We disabled some ASTRÉE relational domain packing options aimed at solving a previously reported alarm on the second development version and could successfully prove it to be false (as in Ex. 4).

Early experience conclusions: The use of the system reduced the alarm investigation time to a few hours in the worst case we faced; the refining analyses are fully automatic and default parameters (fixed number of global forward-backward steps, no local iterations) did not have to be twicked too much to give good results. Fully manual inspection of such alarms would have required days of work and would have made the definition of an error scenario much more involved. Moreover, we could successfully classify all alarms, which means that *no false alarm remains*.

6 Conclusion and Related Work

We proposed a framework adapted to alarm inspection. Early experiments are positive about the ability of the system to reduce the burden of tracking the source of alarms in ASTRÉE: overall, *all considered alarms could be classified* (no case similar to Fig. 1(c) left), which is a very positive result.

Some forms of conditioned slicing [23, 6] attack a similar problem. However, these methods are essentially based on a purely syntactic process, not only for the extraction but also for the shape of the result (a slice is a subset of the program statements [32]). Slicing has been employed for debugging tasks. Recent advances in this area led to the implementation of conditioned slicing tools like [16], that could be applied to testing and software debugging [20]. However, our system is able to produce *semantic* slices, i.e. to provide global information about a set of executions instead of a mere syntactic subset of the program; this is a major advantage when investigating complex errors. The downside is the use of more sophisticated algorithms; however, syntactic slicing alone would not help significantly the alarm inspection process in ASTRÉE.

The search for counter-examples and automatic refinement has long been a motivation in the model-checking-based systems, such as [7, 3, 29, 30, 18]. In particular, the automatic refinement process plays a great role in the determination

of the set of predicates (i.e. abstract domain) needed for a precise analysis [2]. Our goal is to bring such methods in static analyzers like ASTRÉE for a different purpose, i.e. to solve the few, subtile alarms, after an already very precise analysis [5] (the construction of the domain requires no internal refinement process).

Forward-backward analysis schemes have been applied, e.g. in [22], to the inference of safety properties. Some static analysis systems have been extended with counter-examples search facilities: [17] relies on random test generation; [14] uses a symbolic under-approximation of erroneous traces and theorem proving. The main difference is that we chose to start with an over-approximation of erroneous traces until conditions on inputs are precise enough so that a counter-example could be found since the search space for counter-examples was huge in our case, due to the size of the programs. For instance, the systematic exploration of paths as in [14] over length above 1000, with hundreds of variables would not work. Moreover, we allow abstract error scenario to be tested unlike [17, 14]: this reduces the amount of input constraints to fix to a minimum. On the other hand, we leave the automatic generation of counter-examples as a future work.

Future work should make the process more automatic for attempting to discover an error scenario, by proposing input sequences and restricting to adapted alarm contexts (which are user provided in Sect. 3.1 and Sect. 3.2). We also plan to make the choice of slices to analyze (Sect. 4) more sensible, by using the result of the initial forward analysis, to choose which part of the invariant to refine.

Acknowledgments We wish to thank deeply B. Blanchet, P. Cousot, R. Cousot, J. Feret, C. Hymans, L. Mauborgne, A. Miné, and D. Monniaux for comments on early version of this paper and discussions. We are also grateful to M. Sagiv for interesting discussions about related work.

References

1. S. Adams, T. Ball, M. Das, S. Lerner, S. K. Rajamani, M. Seigle, and W. Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *SAS*, 2002.
2. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, 2001.
3. T. Ball, M. Naik, and S. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL*, 2003.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, LNCS 2566. 2002.
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. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
8. P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, 1978.

9. P. Cousot. Semantic foundations of program analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.
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 and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.
12. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP*, 2005.
13. N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, 2003.
14. G. Erez. Generating counter examples for sound abstract interpretation. Master's thesis, 2004.
15. J. Feret. Static analysis of digital filters. In *ESOP*, 2004.
16. C. Fox, S. Danicic, M. Harman, and R. Hierons. ConSIT: A Conditioned Program Slicing System. *Software - Practice and Experience*, 2004.
17. F. Gaucher, E. Jahier, B. Jeannet, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *AADEBUG*, 2003.
18. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, pages 361–416, 2000.
19. P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *FSTTCS*, 1992.
20. R. Hierons, M. Harman, C. Fox, , L. Ouarbya, and D. Daoudi. Conditioned slicing supports partition testing. *Journal of Software Testing, Verification and Reliability*, 2002.
21. S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing using Program Dependence Graphs. *Programming Languages and Systems*, 1990.
22. B. Jeannet. Dynamic partitioning in linear relation analysis. *Formal Methods in System Design*, 2003.
23. B. Korel and J. Laski. Dynamic Program Slicing. *Information Processing Letters*, 1988.
24. T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, 2000.
25. L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *ESOP*, 2005.
26. A. Miné. The Octagon Abstract Domain. In *Analysis, Slicing and Transformation (in WCRE)*, 2001.
27. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, 2004.
28. A. Miné. *Weakly relational numerical abstract domains*. PhD thesis, 2004.
29. G. Pace, N. Halbwachs, and P. Raymond. Counter-example generation in symbolic abstract model-checking. In *6th International Workshop on Formal Methods for Industrial Critical Systems, FMICS*, 2001.
30. A. Podelski. Software model checking with abstraction refinement. In *VMCAI*, 2003.
31. A. Venet and G. Brat. Precise and efficient array bound checking for large embedded c programs. In *PLDI*, 2004.
32. M. Weiser. Program slicing. In *Proceeding of the Fifth International Conference on Software Engineering*, pages 439–449, 1981.

A Concrete Semantics

Expressions: We assume that all variables have a type (boolean, integer, or floating point) and that all programs are well typed: the type of the right hand side and the type of the left hand side of an assignment are the same; the type of a condition is boolean.

The semantics of expressions maps an expression and a memory state to the value of the expression in this state: $\forall e \in \mathtt{e},\ [\![e]\!] : \mathtt{M} \to \mathtt{V}$. It is defined by induction on the syntax of expressions and by using the concrete definition for the arithmetic and boolean operators.

Transition rules: If $\rho \in \mathtt{M}, x \in \mathtt{X}, v \in \mathtt{V}$, we let $\rho[x \leftarrow v]$ denote the update of x with the value v: $\rho[x \leftarrow v] : \mathtt{X} \to \mathtt{V}; x \mapsto v; x' \neq x \mapsto \rho(x')$.

We list below the transition rules associated to all the (labeled) statements:

- assignment $l_0 : x := e; l_1$ $(x \in \mathtt{X}, e \in \mathtt{e})$: if $v = [\![e]\!](\rho)$,
$$(l_0, \rho) \to (l_1, \rho[x \leftarrow v])$$
- input statement $l_0 : \mathbf{input}(x)$ $(x \in \mathtt{X}, v \in \mathtt{V})$: if v has the same type as x,
$$(l_0, \rho) \to (l_1, \rho[x \leftarrow v])$$
- assertion statement $l_0 : \mathbf{assert}(e); l_1$ $(e \in \mathtt{e})$:
$$[\![e]\!](\rho) = \mathbf{true} \Longrightarrow (l_0, \rho) \to (l_1, \rho)$$
$$[\![e]\!](\rho) = \mathbf{false} \Longrightarrow (l_0, \rho) \to \Omega$$
- if statement $l_0 : \mathbf{if}(e)\{l_0^t : s_0; l_1^t\}\mathbf{else}\{l_0^f : s_1; l_1^f\}; l_1$ $(e \in \mathtt{e}, s_0, s_1 \in \mathtt{s})$:
$$[\![e]\!](\rho) = \mathbf{true} \Longrightarrow (l_0, \rho) \to (l_0^t, \rho)$$
$$[\![e]\!](\rho) = \mathbf{false} \Longrightarrow (l_0, \rho) \to (l_0^f, \rho)$$
$$(l_1^t, \rho) \to (l_1, \rho)$$
$$(l_1^f, \rho) \to (l_1, \rho)$$
- loop statement $l_0 : \mathbf{while}(e)\{l_b : s; l_b'\}; l_1$ $(e \in \mathtt{e}, s \in \mathtt{s})$:
$$[\![e]\!](\rho) = \mathbf{true} \Longrightarrow (l_0, \rho) \to (l_b, \rho)$$
$$[\![e]\!](\rho) = \mathbf{false} \Longrightarrow (l_0, \rho) \to (l_1, \rho)$$
$$(l_b', \rho) \to (l_0, \rho)$$

B Definition of the Forward Abstract Interpreter

Soundness of the abstract operators: We assume the abstract operators provided by $D_n^\sharp$ to satisfy the following:

- forward assignment $\overrightarrow{assign} : \mathtt{X} \times \mathtt{e} \times D_n^\sharp \to D_n^\sharp$:
$$\forall \rho, x, e, d,\ \rho \in \gamma_n(d) \Longrightarrow \rho[x \leftarrow [\![e]\!](\rho)] \in \gamma_n(\overrightarrow{assign}(x, e, d))$$
- condition operator $guard : \mathtt{e} \times D_n^\sharp \to D_n^\sharp$:
$$\forall \rho, e, d,\ \rho \in \gamma_n(d) \wedge [\![e]\!](\rho) = \mathbf{true} \Longrightarrow \rho \in \gamma_n(guard(e, x))$$

– forget operator $\mathit{forget} : \mathbb{X} \times D_n^\sharp \to D_n^\sharp$:

$$\forall \rho, x, d, v,\ \rho \in \gamma_n(d) \Longrightarrow \rho[x \leftarrow v] \in \gamma_n(\mathit{forget}(x, d))$$

Forward abstract transfer functions: The forward transfer functions associated to each statement are defined by:

– assignment $l_0 : x := e; l_1$ $(x \in \mathbb{X}, e \in \mathbb{e})$:

$$\overrightarrow{\delta}_{l_0,l_1} : d \mapsto \overrightarrow{\mathit{assign}}(x, e, d)$$

– input statement $l_0 : \mathbf{input}(x)$ $(x \in \mathbb{X})$:

$$\overrightarrow{\delta}_{l_0,l_1} : d \mapsto \mathit{forget}(x)$$

– assertion statement $l_0 : \mathbf{assert}(e); l_1$ $(e \in \mathbb{e})$:

$$\overrightarrow{\delta}_{l_0,l_1} : d \mapsto \mathit{guard}(e, d)$$

– if statement $l_0 : \mathbf{if}(e)\{l_0^t : s_0; l_1^t\}\mathbf{else}\{l_0^f : s_1; l_1^f\}; l_1$ $(e \in \mathbb{e}, s_0, s_1 \in \mathbb{s})$:

$$\overrightarrow{\delta}_{l_0,l_0^t} : d \mapsto \mathit{guard}(e, d)$$
$$\overrightarrow{\delta}_{l_0,l_0^f} : d \mapsto \mathit{guard}(\neg e, d)$$
$$\overrightarrow{\delta}_{l_1^t,l_1} : d \mapsto d$$
$$\overrightarrow{\delta}_{l_1^f,l_1} : d \mapsto d$$

– loop statement $l_0 : \mathbf{while}(e)\{l_b : s; l_b'\}; l_1$ $(e \in \mathbb{e}, s \in \mathbb{s})$:

$$\overrightarrow{\delta}_{l_0,l_b} : d \mapsto \mathit{guard}(e, d)$$
$$\overrightarrow{\delta}_{l_0,l_1} : d \mapsto \mathit{guard}(\neg e, d)$$
$$\overrightarrow{\delta}_{l_b,l_0} : d \mapsto d$$

– for any $l_0, l_1 \in \mathbb{L}$ such that no $\overrightarrow{\delta}_{l_0,l_1}$ has been defined by the above rules, we let $\overrightarrow{\delta}_{l_0,l_1} : d \mapsto \bot$.

Soundness of $\{\overrightarrow{\delta}_{l,l'} \mid l, l' \in \mathbb{L}\}$ writes down as follows:

$$\forall d \in D_n^\sharp, (l, \rho), (l', \rho') \in \mathbb{S},\ \rho \in \gamma_n(d) \wedge (l, \rho) \to (l', \rho') \Longrightarrow \rho' \in \gamma_n(\overrightarrow{\delta}_{l,l'})$$

Forward abstract interpreter: The forward abstract interpreter is defined by:

$$\begin{aligned} \overrightarrow{F}^\sharp : D^\sharp &\to D^\sharp \\ I &\mapsto \lambda(l \in \mathbb{L}). \sqcup \{\overrightarrow{\delta}_{l,l'}(I(l)) \mid l' \in \mathbb{L}\} \end{aligned}$$

Soundness of $\overrightarrow{F}^\sharp$ boils down to:

$$\forall I \in D^\sharp,\ \forall (l, \rho), (l', \rho') \in \mathbb{S},\ \rho \in \gamma_n(I(l)) \wedge (l, \rho) \to (l', \rho') \Longrightarrow \rho' \in \gamma_n(\overrightarrow{F}^\sharp(I)(l'))$$