The Trace Partitioning Abstract Domain

XAVIER RIVAL and LAURENT MAUBORGNE
École Normale Supérieure

In order to achieve better precision of abstract interpretation based static analysis, we introduce a new generic abstract domain, the trace partitioning abstract domain. We develop a theoretical framework allowing a wide range of instantiations of the domain, proving that all these instantiations give correct results. From this theoretical framework, we go into implementation details of a particular instance developed in the ASTRÉE static analyzer. We show how the domain is automatically configured in ASTRÉE and the gain and cost in terms of performance and precision.

Categories and Subject Descriptors: F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Verification, Experimentation, Theory

1. INTRODUCTION

Usually, concrete program executions can be described with traces; yet, most static analyses abstract them and focus on proving properties of the set of reachable states. For instance, checking the absence of runtime errors in C programs can be done by computing an over-approximation of the reachable states of the program and then checking that none of these states is erroneous. When computing a set of reachable states, any information about the execution order and the concrete flow paths is lost.

However, this *reachable states abstraction* might lead to too harsh an approximation of the program behavior, resulting in a failure of the analyzer to prove the desired property. This is easily illustrated with the following example.

1.1 A Simple Motivating Example

Let us consider the program:

```
     int $x, sgn$;
$l_0$ if$(x < 0)\{$
$l_1$     $sgn = -1;$
$l_2$ $\}$else$\{$
$l_3$     $sgn = 1;$
$l_4$ $\}$
$l_5$ $y = x / sgn;$
$l_6$ ...
```

Clearly *sgn* is either equal to 1 or -1 at point l_4; in particular, *sgn* cannot be equal to 0. As a consequence, dividing by *sgn* at point l_5 is safe. However, a simple interval analysis [Cousot and Cousot 1977] would not discover it, since the lub (least upper bound) of the intervals $[-1, -1]$ and $[1, 1]$ is the interval $[-1, 1]$ and $0 \in [-1, 1]$. Indeed, it is well-known that the lub operator of domains expressing convex constraints may induce a loss of precision, since elements in the convex hull may be added to the result, which are not in either arguments. A simple fix would be to use a more expressive abstract domain.

A first possible refinement relies on disjunctive completion [Cousot and Cousot 1979], i.e., the possible values for a variable are abstracted into the union of a set of intervals. An abstract value would be a finite union of intervals; hence, the analysis would report x to be in $[-1,-1] \cup [1,1]$ at the end of the above program. An important drawback of disjunctive completion is its cost: when applied to a finite domain of cardinal n, it produces a domain of 2^n elements, with chains of length $n+1$. Moreover, the design of a widening for the domains obtained by disjunctive completion is a non-trivial issue; in particular, a good widening operator should decide which elements of a partition to merge or to widen.

A second solution to these issues is to refine the abstract domain, so as to express a *relation* between x and sgn. For instance, we would get the following constraint, at point ℓ_5:

$$\begin{cases} x < 0 \Rightarrow sgn = -1 \\ x \geq 0 \Rightarrow sgn = 1 \end{cases}$$

Such an abstraction would be very costly if applied exhaustively, to any variable (especially if the program to analyze contains thousands of variables), therefore a strategy should be used in order to determine which relations may be useful to improve the precision of the result. However, the choice of the predicate which should guide the partitioning (i.e., $x < 0$ in the above example) may not always be obvious.

For instance, common relational domains like octagons [Miné 2001] or polyhedra [Cousot and Halbwachs 1978] would not help here, since they describe convex sets of values, so the abstract union operator is an imprecise over-approximation of the concrete union. A reduced product of the domain of intervals with a congruence domain [Granger 1989] succeeds in proving the property, since -1 and 1 are both in $\{1 + 2 \times k \mid k \in \mathbb{N}\}$.

However, a more intuitive way to solve the difficulty would be to relate the value of sgn to the way it is computed. Indeed, if the *true* branch of the conditional was executed, then $sgn = -1$; otherwise, $sgn = 1$. This amounts to keeping *some* disjunctions based on control criteria. Each element of the disjunction is related to some property about the history of concrete computations, such as “which branch of the conditional was taken”. This approach was first suggested by [Handjieva and Tzolovski 1998]; yet, it was presented in a rather limited framework and no implementation result was provided. The same idea was already present in the context of data-flow analysis in [Holley and Rosen 1980] where the history of computation is traced using an automaton chosen before the analysis.

Choosing the relevant partitioning (which explicit disjunctions to keep during the static analysis) is a rather difficult and crucial point. In practice, it can be necessary to make this choice at analysis time. Another possibility presented in [Ammons and Larus 1998] is to use profiling to determine the partitions, but this approach is relevant in optimization problems only.

1.2 The Astrée Analyzer

The previous example —and many others— where encountered during the development of the Astrée analyzer [Blanchet et al. 2002; 2003]. The Astrée analyzer is an abstract interpretation-based static program analyzer aiming at proving automatically the absence of run time errors in programs written in the C programming language. The errors detected by Astrée include out-of-bound array access, illegal arithmetic operations, overflows and simple user-defined assertions [Mauborgne 2004]. Astrée has been applied with success to large safety critical real-time software, producing a correctness proof for complex software without any false alarm in a few hours of computation [Cousot et al. 2005]. The challenge of analyzing such large codes without any false alarms is very demanding, especially in a context of large number of global variables and intensive floating point computation. In order to achieve that result, we needed techniques which are

— well founded theoretically to provide a good confidence in the correctness proofs;

— efficient to scale up to programs with thousands of global variables and hundred of thousands of lines of code;

— flexible enough to provide a solution to many precision issues, in order to be reactive to end-users of the analyzer.

The trace partitioning abstract domain presented in this paper seems to fulfill those requirements. We provide a *theoretical framework* for trace partitioning, that can be instantiated in a broad series of cases. More partitioning configurations are supported than in [Handjieva and Tzolovski 1998] and the framework also supports *dynamic partitioning* (choice of the partitions during the abstract computation).

Also, we provide detailed practical information about the use of the trace partitioning domain. These details are supported by the experience of the design, implementation and practical use of the Astrée analyzer. We describe the implementation of the domain and we review some strategies for partition creation during the analysis.

Finally, we assess the efficiency of the technique by presenting experimental results with Astrée and more examples where Astrée uses trace partitioning to solve precision issues.

1.3 Outline of the Paper

Section 2 introduces the most basic notions and notations used throughout the paper.

The next two sections are devoted to the theoretical trace partitioning framework. The control-based partitioning of transition system is formalized in Section 3; it is the basis for the definition of trace partitioning abstract domains in Section 4.

Then, we describe the structure of the trace partitioning domain used in the Astrée analyzer in Section 5, and the implementation of this domain, together with partitioning strategies and detailed experimental evaluations in Section 6.

Last, Section 7 concludes and reviews related works.

2. ABSTRACT INTERPRETATION-BASED STATIC ANALYSIS

This section introduces basic notions and notations used in the paper. It also collects the most important technical facts about the ASTRÉE analyzer, which are required in order to understand the future sections about trace partitioning in ASTRÉE.

2.1 Notations

In this paper, all example programs are written in the C language (which is the language analyzed by ASTRÉE). However, we provide here a formal model of programs, as transition systems, so that all techniques and algorithms can be generalized to other languages as well.

2.1.1 *Syntax.* We describe an imperative (C) program with a transition system.

More precisely, we let $\mathbb{V}$ denote a set of *values*; $\mathbb{X}$ denote a finite set of *memory locations* (aka variables). A *memory state* (or *store*) describes the values stored in the memory at a precise time in the execution of the program; it is a mapping of program variables into values. A store is a function $\sigma \in \mathbb{M}$, where $\mathbb{M} = \mathbb{X} \to \mathbb{V}$.

A *control state* (or *program point*) roughly corresponds to the program counter at a precise time in the execution of the program; we usually write $\mathbb{L}$ for the set of control states.

A *state* s is a pair made of a control state $\ell \in \mathbb{L}$ and a memory state $\sigma \in \mathbb{M}$. We write $\mathbb{S}$ for the set of states, so $\mathbb{S} = \mathbb{L} \times \mathbb{M}$.

A program is defined by a set $\mathbb{L}$ of control states, a set of *initial states* $\mathbb{S}^{\mathrm{i}}$, and a transition relation $(\to) \subseteq \mathbb{S} \times \mathbb{S}$, which describes how the execution of the program may step from one state to the next one.

In practice, $\mathbb{S}^{\mathrm{i}} = \{\ell^{\mathrm{i}}\} \times \mathbb{M}$, where $\ell^{\mathrm{i}} \in \mathbb{L}$ is the *entry control state*, i.e. the first point in the program.

Real world programs may crash, e.g., due to a memory or arithmetic error. Therefore, we should also add a special *error state*; though, we do not need to deal formally with errors in this paper, so we omit it.

Last, we will also consider *interprocedural* programs. Then, a control state is defined by a pair (κ, ℓ), where κ is a calling stack (stack of function names) and ℓ a syntactic control point. We write $\mathbb{k}$ for the set of stacks, and keep the notation $\mathbb{L}$ for the syntactic control points, so that a state in an interprocedural program is a tuple in $(\mathbb{k} \times \mathbb{L}) \times \mathbb{M}$.

2.1.2 *Semantics.* We assume here that a program P is defined by the data of a tuple $(\mathbb{L}, \mathbb{X}, \to, \mathbb{S}^{\mathrm{i}})$. The most common semantics for describing the behavior of transition systems is the *operational semantics*, which we sketch here. It was introduced, e.g. in [Plotkin 1981].

An execution of a program is represented with a sequence of states, called a *trace*; the semantics of the program collects all such executions:

Definition 2.1.1. (TRACE, SEMANTICS) A *trace* σ is a *finite* sequence $\langle s_0, \ldots, s_n \rangle$ where $s_0, \ldots, s_n \in \mathbb{S}$. We write $\mathbb{S}^\star$ for the set of such traces, and $\mathbf{length}(\sigma)$ for the length of σ.

A trace of P is a trace such that any two successive states are bound by the transition relation: $\forall i,\ s_i \to s_{i+1}$. The *semantics* $[\![P]\!]$ of P is the set of traces of P, i.e. $[\![P]\!] = \{\langle s_0, \ldots, s_n\rangle \in \mathbb{S}^\star \mid s_0 \in \mathbb{S}^{\mathrm{i}} \wedge \forall i,\ s_i \rightarrow s_{i+1}\}$.

Note that we restrict to finite traces.

We can remark that the semantics $[\![P]\!]$ of P writes down as a least fixpoint:

$$[\![P]\!] = \mathbf{lfp}^{\subseteq}_{\mathcal{S}^{\mathrm{i}}} F_{\overrightarrow{P}}$$

where $F_{\overrightarrow{P}}$ is the *semantic function*, defined by:

$$\begin{array}{rcl} F_{\overrightarrow{P}} : \mathbb{S}^\star & \rightarrow & \mathbb{S}^\star \\ \mathcal{E} & \mapsto & \mathcal{E} \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid \langle s_0, \ldots, s_n\rangle \in \mathcal{E} \wedge s_n \rightarrow s_{n+1}\} \end{array}$$

2.2 Abstraction

The semantics introduced in Section 2.1.2 is not decidable; therefore, proving safety properties about programs usually requires computing an approximation of the reachable states. We describe such an abstraction here.

2.2.1 *Set of Traces of Interest.* In this section, we consider a program P defined by the data of a tuple $(\mathbb{X}, \mathbb{L}, \mathbb{S}^{\mathrm{i}}, \rightarrow)$. We focus on the approximation of the *executions* of P, i.e. on the states which appear in a trace of P. As a consequence, we wish to approximate the set of traces $\mathcal{T} = \{\langle s_0, \ldots, s_n\rangle \mid \exists \rho_0, s_0 = (\ell^{\mathrm{i}}, \rho_0)\}$. We recall that that $\mathcal{T} = \mathbf{lfp}_{\mathbb{S}^{\mathrm{i}}} F$.

We proceed to the abstraction of traces into *reachable states*: we wish to abstract the traces into an approximation for the set of states $\mathcal{S}$ which appear in at least one tract in $\mathcal{T}$. In the following, we approximate all the states distinct from Ω: deciding whether Ω is reachable from the set of all reachable, non-error states is usually straightforward (it amounts to checking whether there exists a state s such that $s \rightarrow \Omega$ in the set $\mathcal{S}$).

2.2.2 *Abstraction of Traces.* We assume that an abstract domain $(D^\sharp_{\mathbb{M}}, \sqsubseteq)$ for representing sets of stores is defined, together with a concretization function $\gamma_{\mathbb{M}} : D^\sharp_{\mathbb{M}} \rightarrow \mathcal{P}(\mathbb{M})$.

The abstract values in such a domain usually express constraints among the variables of the program. The interval domain [Cousot and Cousot 1977] allows to express ranges the variables should live in. Relational abstractions allow to express constraints involving several variables; we can cite the polyhedra [Cousot and Halbwachs 1978], octagons [Miné 2001] as examples of relational abstractions.

We let the abstraction for approximating the concrete semantics be defined by:

— the abstract domain $D^\sharp = \mathbb{L} \rightarrow D^\sharp_{\mathbb{M}}$, with the pointwise ordering induced by $\sqsubseteq$ (which we also write $\sqsubseteq$);

— the concretization function $\gamma : \imath \in D^\sharp \mapsto \{\langle(\ell_0, \rho_0), \ldots, (\ell_n, \rho_n)\rangle \mid \forall i,\ \rho_i \in \gamma_{\mathbb{M}}(\imath(\ell_i))$.

Intuitively, this very simple abstraction collects the memory states corresponding to each control state and applies the store abstraction to the resulting sets of stores.

2.2.3 *Abstract Operations.* Moreover, we assume that the domain $D^\sharp_{\mathbb{M}}$ provides some *sound abstract operations* (in the following, we write $\mathbb{e}$ for the set of expressions, $\mathbb{l}$ for the set of l-values, and $\mathbb{B}$ for the set of booleans):

— a *least* element $\bot$, such that $\gamma_{\mathbb{M}}(\bot) = \emptyset$;

— a *greatest* element $\top$, such that $\gamma_{\mathbb{M}}(\top) = \mathbb{M}$;

— an abstract join operator $\sqcup$, approximating the concrete operator ($\forall x, y \in \mathcal{P}(\mathbb{M}),\ x^\sharp, y^\sharp \in D^\sharp_{\mathbb{M}},\ x \subseteq \gamma_{\mathbb{M}}(x^\sharp) \wedge y \subseteq \gamma_{\mathbb{M}}(y^\sharp) \Longrightarrow x \cup y \subseteq \gamma_{\mathbb{M}}(x^\sharp \sqcup y^\sharp)$).

— a sound counterpart $\mathit{guard} : \mathbb{e} \times \mathbb{B} \times D^\sharp_{\mathbb{M}} \to D^\sharp_{\mathbb{M}}$ for the concrete testing of conditions:

$$\begin{array}{l}\forall \rho \in \mathbb{M},\ e \in \mathbb{e},\ b \in \mathbb{B},\ d \in D^\sharp_{\mathbb{M}},\\ \left.\begin{array}{r}\rho \in \gamma_{\mathbb{M}}(d)\\ \wedge\ \llbracket e \rrbracket(\rho) = b\end{array}\right\} \Longrightarrow \rho \in \gamma_{\mathbb{M}}(\mathit{guard}(e, b, d))\end{array}$$

Since the operator $\mathit{guard} : (e, b, d) \mapsto d$ trivially satisfies the above assumption, we assume that the guard operator is reductive:

$$\forall \rho \in \mathbb{M},\ e \in \mathbb{e},\ b \in \mathbb{B},\ \gamma_{\mathbb{M}}(\mathit{guard}(e, b, d)) \subseteq \gamma_{\mathbb{M}}(d)$$

— a sound counterpart $\mathit{assign} : \mathbb{l} \times \mathbb{e} \times D^\sharp_{\mathbb{M}} \to D^\sharp_{\mathbb{M}}$ for the concrete assignment:

$$\begin{array}{l}\forall \rho \in \mathbb{M},\ \forall l \in \mathbb{l},\ e \in \mathbb{e},\ d \in D^\sharp_{\mathbb{M}},\\ \left.\begin{array}{r}\rho \in \gamma_{\mathbb{M}}(d)\\ \wedge\ \llbracket l \rrbracket(\rho) = x\\ \wedge\ \llbracket e \rrbracket(\rho) = v\end{array}\right\} \Longrightarrow \rho[x \leftarrow v] \in \gamma_{\mathbb{M}}(\mathit{assign}(l, e, d))\end{array}$$

— a sound counterpart $\mathit{forget} : \mathbb{l} \times D^\sharp_{\mathbb{M}} \to D^\sharp_{\mathbb{M}}$ for the "variable-forget" operation, which writes a random value into a variable:

$$\begin{array}{l}\forall \rho \in \mathbb{M},\ \forall l \in \mathbb{l},\ \forall v \in \mathbb{V},\ \forall d \in D^\sharp_{\mathbb{M}},\\ \left.\begin{array}{r}\rho \in \gamma_{\mathbb{M}}(d)\\ \wedge\ \llbracket l \rrbracket(\rho) = x\end{array}\right\} \Longrightarrow \rho[x \leftarrow v] \in \gamma_{\mathbb{M}}(\mathit{forget}(l, d))\end{array}$$

The purpose of these abstract operations is to ensure that we can use them in the design of a static analyzer, which over-approximates each computation step. As a result, the soundness of such an analyzer follows from a simple fixpoint transfer theorem, like:

THEOREM 2.2.1. (FIXPOINT TRANSFER) *We let $x \in \mathcal{P}(\mathbb{M})$, $d \in D^\sharp_{\mathbb{M}}$. Let $F : \mathcal{P}(\mathbb{M}) \to \mathcal{P}(\mathbb{M})$ and $F^\sharp : D^\sharp_{\mathbb{M}} \to D^\sharp_{\mathbb{M}}$. Then, if $x \subseteq \gamma_{\mathbb{M}}(d)$, and $F \circ \gamma_{\mathbb{M}} \subseteq \gamma_{\mathbb{M}} \circ F^\sharp$, then $\mathbf{lfp}_x F \subseteq \gamma_{\mathbb{M}}(\mathbf{lfp}_d F^\sharp)$.*

2.2.4 *Widening Iteration and Convergence.* The fixpoint-transfer scheme presented above leaves one major issue to be addressed: the sequences of abstract iterates might be infinite, in case the abstract domain has infinite increasing chains (this is the case for most domains, including intervals, polyhedra, octagons...). Therefore, we replace the abstract join operator with a *widening operator* [Cousot and Cousot 1977], which is an approximate join [Cousot and Cousot 1992b], with additional termination properties:

Definition 2.2.2. (WIDENING OPERATOR) A *widening* is a binary operator ∇ on $D^\sharp$, which satisfies the two following properties:

(1) $\forall x^\sharp, y^\sharp \in D^\sharp,\ x^\sharp \sqsubseteq x^\sharp \nabla y^\sharp \wedge y^\sharp \sqsubseteq x^\sharp \nabla y^\sharp$

(2) For any sequence $(x_n)_{n\in\mathbb{N}}$, the sequence $(y_n)_{n\in\mathbb{N}}$ defined below is not strictly increasing:

$$\begin{cases} y_0 & = x_0 \\ \forall n \in \mathbb{N},\ y_{n+1} & = y_n \nabla x_{n+1} \end{cases}$$

The following theorem [Cousot and Cousot 1977] shows how widening operators makes it possible to compute in a finite number of iterations a sound over-approximation for the concrete properties:

Theorem 2.2.3. (Abstract iteration with widening) *We assume a concretization $\gamma : D^\sharp \rightarrow D$ is defined and that $F^\sharp$ is such that $F \circ \gamma \subseteq \gamma \circ F^\sharp$. Let $x \in D$, $x^\sharp \in D^\sharp$, such that $x \subseteq \gamma(x^\sharp)$. We define the sequence $(x_n)_{n\in\mathbb{N}}$ as follows:*

$$\begin{cases} x_0 & = x^\sharp \\ \forall n \in \mathbb{N},\ x_{n+1} & = x_n \nabla F^\sharp(x_n) \end{cases}$$

Then, the sequence $(x_n)_{n\in\mathbb{N}}$ is ultimately stationary and its limit $\lim(x_n)_{n\in\mathbb{N}}$ is a sound approximation of $\mathbf{lfp}_x F$:

$$\mathbf{lfp}_x F \subseteq \gamma(\lim(x_n)_{n\in\mathbb{N}})$$

Proof. See [Cousot and Cousot 1977]. □

2.3 Static Analysis

Last, we briefly review the design of a *denotational style abstract interpreter* for a fragment of C. In particular, the iterator of the Astrée analyzer follows this scheme. In this subsection, we assume that a domain $D^\sharp_{\mathbb{M}}$ is given. Indeed, the iterator of the Astrée analyzer accepts an abstract domain for representing sets of stores as a parameter.

2.3.1 *The Core of the Interpreter.*

Let s be a statement; we write $\ell_\vdash$ (resp. $\ell_\dashv$) for the control point before (resp. after) s. We let the denotational semantics of s be the function $[\![s]\!]_\delta = \alpha_{\iota\mathcal{F}[\ell_\vdash,\ell_\dashv]}([\![s]\!])$. The abstract semantics of s is the function $[\![s]\!]^\sharp : D^\sharp_{\mathbb{M}} \rightarrow D^\sharp_{\mathbb{M}}$, which inputs an abstract pre-condition and returns a strongest post-condition. It should be sound in the sense that the output of the abstract semantics should over-approximate the set of output states of the underlying, concrete denotational semantics.

We propose on Figure 1 the definition of a very simple denotational semantics-based interpreter; we provide for each common language construction (assignment, conditional, loop, reading of an input, assertion) a simple abstract transfer function. The abstract semantics displayed in Figure 1 is sound:

Theorem 2.3.1. (Soundness of the analysis) *The abstract semantics soundly approximates the denotational semantics:*

$$\forall \rho, \rho' \in \mathbb{M},\ d \in D^\sharp_{\mathbb{M}},\ \rho \in \gamma_{\mathbb{M}}(d) \wedge \rho' \in [\![s]\!]_\delta(\rho) \Longrightarrow \rho' \in \gamma_{\mathbb{M}}([\![s]\!]^\sharp(d))$$

Proof. By induction on the structure of the code.

The case of the loop is based on the soundness of the $\mathbf{lfp}^\sharp$ operator; in practice, it is derived from a widening operator $\nabla_{\mathbb{M}}$ over $D^\sharp_{\mathbb{M}}$, so the soundness and termination of $\mathbf{lfp}^\sharp$ follow from Theorem 2.2.3. □

statement s	abstract semantics
$x := e;$	$[\![s]\!]^\sharp : d \mapsto \mathit{assign}(x, e, d)$
$\mathbf{if}(e)\{s_0\}\mathbf{else}\{s_1\}$	$[\![s]\!]^\sharp : d \mapsto [\![s_0]\!]^\sharp(\mathit{guard}(e, \mathbf{true}, d)) \sqcup [\![s_1]\!]^\sharp(\mathit{guard}(e, \mathbf{false}, d))$
$\mathbf{while}(e)\{s\}$	$[\![s]\!]^\sharp : d \mapsto \mathit{guard}(e, \mathbf{false}, \mathbf{lfp}^\sharp F^\sharp)$ where $F^\sharp : \begin{array}{lcl} D^\sharp_{\mathbb{M}} & \to & D^\sharp_{\mathbb{M}} \\ d_0 & \mapsto & d_0 \sqcup [\![s]\!]^\sharp(\mathit{guard}(e, \mathbf{true}, d)) \end{array}$ and $\mathbf{lfp}^\sharp$ computes an abstract post-fixpoint
$\mathbf{input}(x \in V);$	$[\![s]\!]^\sharp : d \mapsto \mathit{guard}(x \in V^\sharp, \mathit{forget}(x, d))$
$\mathbf{assert}(e);$	$[\![s]\!]^\sharp : d \mapsto \mathit{guard}(e, \mathbf{true}, d)$

Fig. 1: A simple abstract interpreter

As a corollary, the abstract semantics is sound with respect to the standard, operational semantics. Indeed, if $\ell_\vdash : s; \ell_\dashv$ is a program, then:

$$\forall \langle (\ell_\vdash, \rho_\vdash), \ldots, (\ell_\dashv, \rho_\dashv) \rangle \in [\![s]\!],\ \forall d \in D^\sharp_{\mathbb{M}},\ \rho_\vdash \in \gamma_{\mathbb{M}}(d) \Longrightarrow \rho_\dashv \in \gamma_{\mathbb{M}}([\![s]\!]^\sharp(d))$$

2.3.2 *Output of the Analysis.* We showed how to compute a sound invariant after a statement from a pre-condition; however, the ASTRÉE analyzer not only outputs an invariant in the end of the analyzed program, but also:

— Alarm reports, if the invariants do not ensure that all critical operations (including arithmetic operations, memory operations, user assertions) are safe;

— Local invariants, if the invariant export option is enabled: then, the analyzer saves local invariants for all control states in the program, so that the user can scan the results of the analysis (of course, this option requires a lot of memory, when a large program is being analyzed, which is the reason why we insisted on the choice of a strategy, where the export of all local invariants is not mandatory).

In particular, the analyzer is equivalent to an analyzer computing an invariant in $D^\sharp = \mathbb{L} \to D^\sharp_{\mathbb{M}}$, even though it explicitly outputs only an invariant in $D^\sharp_{\mathbb{M}}$.

2.3.3 *Running the Analyzer.* The previous paragraphs summarize the principle of the iterator of the ASTRÉE analyzer; however, during the analysis of a program, ASTRÉE performs many other operations, including:

— several pre-processing steps, such as constant propagation in the code to analyze;

— the choice of analysis parameters, which affect the abstract domain and the iteration strategy, such as the choice of packs of variables relations should be computed for; similar choices will be made in the case of the trace partitioning domain (the strategy defining which partitions should be created will be exposed in Section 6.2)

The implementation of the abstract domain was guided by the kind of predicates required for tight invariants to be inferred. By default, the analyzer uses a reduced product of a series of abstract domains including intervals [Cousot and Cousot 1977], octagons [Miné 2001], boolean relations (i.e., a generalization of BDDs [Bryant 1986]), arithmetic-geometric progressions [Feret 2005], and a domain dedicated to the analysis of digital filters [Feret 2004].

3. CONTROL-BASED PARTITIONING OF TRANSITION SYSTEMS

In this section, we formalize the notion of *partition of a transition system.* Such partitions should describe the same set of traces as the the semantics of the initial program (or an approximation of it) and should allow to distinguish traces depending on the history of control flow. As a result, we can address the imprecision mentioned in the introduction by analyzing a partitioned system.

3.1 Partitioning Control States

First of all, we underline that partitioning the reachable states with the control states is a rather common approach in static analysis. Later, we generalize drastically this technique.

3.1.1 *Non-Procedural Case.* Indeed, the analysis proposed in Section 2 relies on this kind of partitioning. The abstraction of sets of traces can be seen as a two steps abstraction:

(1) abstraction of **traces into states**, with *partitioning*:

$$(\mathcal{P}(\mathbb{S}^{\star}), \subseteq) \underset{\alpha_{\mathcal{P}(\mathbb{S})}}{\overset{\gamma_{\mathcal{P}(\mathbb{S})}}{\leftrightarrows}} (\mathbb{L} \to \mathcal{P}(\mathbb{M}), \subseteq)$$

$$\begin{array}{rcl} \alpha_{\mathcal{P}(\mathbb{S})} : \mathcal{P}(\mathbb{S}^{\star}) & \to & (\mathbb{L} \to \mathcal{P}(\mathbb{S})) \\ \mathcal{E} & \mapsto & \lambda(\ell \in \mathbb{L}) \cdot \{\rho \mid \langle \ldots, (\ell, \rho), \ldots \rangle \in \mathcal{E}\} \end{array}$$

Whenever the concretization function is defined straightforwardly from the abstraction function, we provide the abstraction function only: in a complete lattice, any monotone abstraction function defines a unique concretization [Cousot and Cousot 1977].

(2) abstraction of **sets of states**, defined by the concretization function $\gamma_{\mathbb{M}} : D^{\sharp}_{\mathbb{M}} \to \mathcal{P}(\mathbb{M})$.

The first step includes a partitioning in the sense of [Cousot and Cousot 1992a, §4.2.3.2]. Indeed, it amounts to partitioning the set of sets of states using the partition $\{\{(\ell, \rho) \mid \rho \in \mathbb{M}\} \mid \ell \in \mathbb{L}\}$; the resulting domain is in bijection with $\mathbb{L} \to \mathcal{P}(\mathbb{M})$.

3.1.2 *Procedural Case.* In case the language features procedures, similar abstractions are usually implemented.

When designing an analysis for such a procedural language, one faces the problem of deciding how to replace the abstraction mentioned in step 1 above. Among the possible choices, we can cite [Sharir and Pnuelli 1981]:

— the **full abstraction of the stack:** we may abstract away the stack and keep only the control states (analysis insensitive to the calling context):

$$\begin{array}{rcl} \alpha_{\mathcal{P}(\mathbb{S})} : \mathcal{P}(\mathbb{S}^{\star}) & \to & (\mathbb{L} \to \mathcal{P}(\mathbb{M})) \\ \mathcal{E} & \mapsto & \lambda(l \in \mathbb{L}) \cdot \{\rho \mid \exists \kappa \in \mathbb{k},\ \exists \langle \ldots, (\kappa, \ell, \rho), \ldots \rangle \in \mathcal{E}\} \end{array}$$

— the **partitioning with the stack:** we may keep the stack, i.e. abstract traces into functions mapping pairs made of a stack and a control state into a set of memory states (analysis completely sensitive to the calling context):

$$\begin{array}{rcl} \alpha_{\mathcal{P}(\mathbb{S})} : \mathcal{P}(\mathbb{S}^\star) & \rightarrow & ((\mathbb{k} \times \mathbb{L}) \rightarrow \mathcal{P}(\mathbb{S})) \\ \mathcal{E} & \mapsto & \lambda((\kappa, \ell) \in (\mathbb{k} \times \mathbb{L})) \cdot \{\rho \mid \langle \ldots, (\ell, \rho), \ldots \rangle \in \mathcal{E}\} \end{array}$$

This approach amounts to inlining functions; it works only in the case of non-recursive function calls (the stack may grow infinite in the case of recursive calls). At the time this thesis is written, this is the technique implemented in ASTRÉE.

Many intermediate abstractions exist, which allow to retain a good level of precision in some cases and abstract long sequences of calls (the main such technique is k-limiting).

Another approach to the analysis of procedural programs is to modelize the effect of each function (intra-procedural phase) and then, to perform a global iteration [Reps et al. 1995]. This technique relies on the resolution of the reachability along "interprocedural realizable paths", which is also based on some abstraction of the stack (this method was also used in slicing [Horwitz et al. 1988]).

3.2 Partitions and Coverings

We now set up the notions of *partitioned set* and *partitioned system*. Our goal is to design sets of finer partitions at the control structure level, which will be used in the following sections as a basis for building trace partitioning domains.

3.2.1 *Partitioning Function.* A *covering* of a set F is a family of subsets of F, such that any element of F belongs to some element of the family. A *partition* is a covering such that any two distinct elements of the family are disjoint; in particular, for any element $x \in F$, there exists a unique element A of the partition such $x \in A$. In the following, we need to index the elements of coverings (resp. partitions); hence, the following definition resorts to functions, defined on a set of *indexes*.

Definition 3.2.1. (PARTITIONED SET) Let E, F be two sets, and $\delta : E \rightarrow \mathcal{P}(F)$. Then:

— δ is a *covering* of F if and only if:

$$\forall x \in E,\ \delta(x) \neq \emptyset$$

and,

$$F = \bigcup_{x \in E} \delta(x)$$

— δ is a *partition* of F if and only if it is a covering and:

$$\forall x, y \in E,\ x \neq y \Longrightarrow \delta(x) \cap \delta(y) = \emptyset$$

We note that a covering (resp. partitioning) δ of F defines an abstraction of $(\mathcal{P}(F), \subseteq)$:

LEMMA 3.2.1. (PARTITIONING ABSTRACTION) *Let $\alpha_{\mathfrak{P}(\delta)}$ and $\gamma_{\mathfrak{P}(\delta)}$ be defined by:*

$$\begin{array}{rlcl} \alpha_{\mathfrak{P}(\delta)}: & \mathcal{P}(F) & \rightarrow & (E \rightarrow \mathcal{P}(F)) \\ & \mathcal{E} & \mapsto & \lambda(x \in E) \cdot \mathcal{E} \cap \delta(x) \end{array}$$

$$\begin{array}{rlcl} \gamma_{\mathfrak{P}(\delta)}: & (E \rightarrow \mathcal{P}(F)) & \rightarrow & \mathcal{P}(F) \\ & \phi & \mapsto & \bigcup_{x \in E} \phi(x) \end{array}$$

Then, if δ is a covering, we have a Galois-connection $(\mathcal{P}(F), \subseteq) \underset{\alpha_{\mathfrak{P}(\delta)}}{\overset{\gamma_{\mathfrak{P}(\delta)}}{\leftrightarrows}} (E \rightarrow \mathcal{P}(F), \subseteq)$, and $\alpha_{\mathfrak{P}(\delta)}$ is into (Galois injection).

Moreover, if δ is a partition, then $\alpha_{\mathfrak{P}(\delta)}$ is one-to-one (Galois bijection).

PROOF. Straightforward application of the definition of coverings. □

Definition 3.2.1 would allow to set up very general notions of trace partitioning. In particular, the partitioning of traces using the control state of the traces (Section 3.1) of the last state fits in this framework (with $E = \mathbb{L}$); the case of calling stacks is similar (with either $E \equiv \mathbb{L}$, or $E = \Bbbk \times \mathbb{L}$, or other partitions). We may even design some weaker partitions: for instance, we may decide to merge together the state corresponding to several distinct control states (with E a partition of $\mathbb{L}$). However, we wish to derive the partitions from the history of executions; therefore the following paragraph introduces the notion of *partitioned system*.

3.2.2 *Partitioning Transitions.* In the following, we assume that a program P is given, and defined by $(\mathbb{L}, \mathbb{S}^{\mathrm{i}}, \rightarrow)$. We consider partitions finer than the partition defined by $E = \mathbb{L}$ only. More precisely, we let $\mathbb{T}$ be a set of tokens, and $\mathfrak{T} = \mathcal{P}(\mathbb{T})$.

We define *extended transition systems* as transition systems over the sets of labels extended with a set of tokens $T \subseteq \mathbb{T}$; it is basically defined by T and by extensions of the set of initial states and of the transition relation. Such a system P_0 is a covering of P_1 if and only if it simulates the transitions of P_1; moreover, P_0 is a partition if and only if any transition in P_1 is simulated by exactly *one* transition in P_0 (and the same for the initial states). System P_0 is complete in case it does not add any fictitious transition, when compared to P_1. Intuitively, a complete partition or covering P_0 shall describe the same set of traces as P_1, up-to some information added in the control states. The main difference between a covering and a partition is that the covering may not ensure the unicity of the counterpart of the traces of the initial program.

The extra information embedded in the control structure of the extended system will be the basis of the partitioning abstraction. The notions of covering, partitioning and complete systems are formalized in the following definition.

Definition 3.2.2. (PARTITIONED SYSTEM) Let $T \in \mathfrak{T}$. We write $\mathbb{L}_T$ for the set of *partitioned control states* $\mathbb{L} \times T$, $\mathbb{S}_T$ for the set of *partitioned states* $\mathbb{L}_T \times \mathbb{M}$, and $\mathbb{S}^{\mathrm{i}}_T \subseteq \mathbb{S}_T$ for a set of *partitioned initial states*, and $\rightarrow_T$ for a transition relation among partitioned states. An *extended system* is defined by the data of a tuple $(T, \mathbb{S}^{\mathrm{i}}_T, \rightarrow_T)$. Last, $\mathbb{S}^{\star}_T$ denotes the set of traces made of states in $\mathbb{S}_T$.

$$\begin{tikzcd}
 & (\ell_0, t) \arrow[dl] \arrow[dr] & \\
(\ell_1, t) \arrow[dr] & & (\ell_2, t) \arrow[dl] \\
 & (\ell_3, t) \arrow[d] & \\
 & (\ell_4, t) &
\end{tikzcd}$$

(a) System P_0

$$\begin{tikzcd}
 & (\ell_0, t_0) \arrow[dl] \arrow[dr] & \\
(\ell_1, t_1) \arrow[d] & & (\ell_2, t_2) \arrow[d] \\
(\ell_3, t_1) \arrow[dr] & & (\ell_3, t_2) \arrow[dl] \\
 & (\ell_4, t_0) &
\end{tikzcd}$$

(b) System P_1

$$\begin{tikzcd}
 & (\ell_0, t_0) \arrow[dl] \arrow[dr] & \\
(\ell_1, t_1) \arrow[d] & & (\ell_2, t_2) \arrow[d] \\
(\ell_3, t_1) \arrow[d] & & (\ell_3, t_2) \arrow[d] \\
(\ell_4, t_1) & & (\ell_4, t_2)
\end{tikzcd}$$

(c) System P_2

Fig. 2: Partitioned systems

For all $T, T' \in \mathfrak{T}$ and $\tau : T \to T'$, we define the *forget functions* for control states, for states and for traces as follows:

$$\begin{array}{rlcl}
\pi^{\mathbb{L}}_{\tau} : & \mathbb{L}_T & \to & \mathbb{L}_{T'} \\
 & (\ell, t) & \mapsto & (\ell, \tau(t)) \\
\pi^{\$}_{\tau} : & \$_T & \to & \$_{T'} \\
 & ((\ell, t), \rho) & \mapsto & (\pi^{\mathbb{L}}_{\tau}(\ell, t), \rho) \\
\pi^{\$^\star}_{\tau} : & \$^\star_T & \to & \$^\star_{T'} \\
 & \langle s_0, \ldots, s_n \rangle & \mapsto & \langle \pi^{\$}_{\tau}(s_0), \ldots, \pi^{\$}_{\tau}(s_n) \rangle
\end{array}$$

We consider the extended systems $P_T = (\mathbb{L}_T, \mathbb{L}^{\mathrm{i}}_T, \to_T)$ and $P_{T'} = (\mathbb{L}_{T'}, \mathbb{L}^{\mathrm{i}}_{T'}, \to_{T'})$, and the function $\tau : T \to T'$.

(1) P_T is a τ-*covering* of $P_{T'}$ if and only if:

— $\$^{\mathrm{i}}_{T'} \subseteq \pi^{\$}_{\tau}(\$^{\mathrm{i}}_T)$

— $\forall s_0 \in \$_T,\ s'_1 \in \$_{T'},\ \pi^{\$}_{\tau}(s_0) \to_{T'} s'_1 \Longrightarrow \exists s_1 \in \$_T,\ \begin{cases} s'_1 = \pi^{\$}_{\tau}(s_1) \\ s_0 \to_T s_1 \end{cases}$

(2) P_T is a τ-*partition* of $P_{T'}$ if and only if:

— $\forall s' \in \$^{\mathrm{i}}_{T'},\ \exists! s \in \$^{\mathrm{i}}_T,\ s' = \pi^{\$}_{\tau}(s)$

— $\forall s_0 \in \$_T,\ s'_1 \in \$_{T'},\ \pi^{\$}_{\tau}(s_0) \to_{T'} s'_1 \Longrightarrow \exists! s_1 \in \$_T,\ \begin{cases} s'_1 = \pi^{\$}_{\tau}(s_1) \\ s_0 \to_T s_1 \end{cases}$

(3) P_T is τ-*complete* with respect to $P_{T'}$ if and only if:

— $\forall s \in \$^{\mathrm{i}}_T,\ \pi^{\$}_{\tau}(s) \in \$^{\mathrm{i}}_{T'}$

— $\forall s_0, s_1 \in \$_T,\ s_0 \to_T s_1 \Longrightarrow \pi^{\$}_{\tau}(s_0) \to_{T'} \pi^{\$}_{\tau}(s_1)$

The notions of "complete covering" or "complete partition" are derived from the above definition as well.

Example 3.2.3. (Partitioned systems) We make the assumption that $\mathbb{M}$ is a singleton here, so that transitions relations are mere relations among control states. Let us consider the two extended systems P_0 and P_1, displayed respectively in Figure 2(a) and in Figure 2(b).

— the original system represents a program with a conditional statement followed by one statement (each branch of the conditional contains exactly one statement);

— P_0 is isomorphic to the original system; it corresponds to $T_0 = \{t\}$

— P_1 is an extended system defined by $T_1 = \{t_0, t_1, t_2\}$.

We consider the following forget function $\tau : \lambda(t_i \in T_1) \cdot t$.

Then, any execution of P_0 corresponds to exactly one execution of P_1: for instance, $\langle(\ell_0, t), (\ell_1, t), (\ell_3, t), (\ell_4, t)\rangle$ corresponds to $\langle(\ell_0, t_0), (\ell_1, t_1), (\ell_2, t_1), (\ell_4, t_0)\rangle$. In particular, any transition step in P_0 is mimicked by a transition step in P_1 as mentioned in Definition 3.2.2, 2. Therefore, P_1 is a τ-partition of P_0.

Similarly, we can check that any execution, including one-step transitions of P_1 corresponds to some execution of P_1. Hence, P_1 is τ-complete with respect to P_0.

These two properties make P_1 a very useful extended system, in the analysis of P_0.

Intuitively, the extended system P_1 corresponds to a partition of P_0 obtained by delaying the merge in the exit of the conditional statement after the statement following the conditional, i.e. at point ℓ_4; this amounts to doing the following rewriting:

$$
\begin{array}{ll}
\ell_0 : & \mathbf{if}(e)\{ \\
\ell_1 : & \quad s_1 \\
 & \}\mathbf{else}\{ \\
\ell_2 : & \quad s_2 \\
 & \} \\
\ell_3 : & s_3 \\
\ell_4 : & \ldots
\end{array}
\quad \longrightarrow \quad
\begin{array}{ll}
(\ell_0, t_0) : & \mathbf{if}(e)\{ \\
(\ell_1, t_1) : & \quad s_1; \\
(\ell_3, t_1) : & \quad s_3 \\
 & \}\mathbf{else}\{ \\
(\ell_2, t_2) : & \quad s_2; \\
(\ell_3, t_2) : & \quad s_3 \\
 & \} \\
(\ell_4, t_0) : & \ldots
\end{array}
$$

In particular, applying this partitioning to the example presented in the introduction (Section 1.1) would solve the imprecision. Indeed, it would allow proving that *sgn* cannot be equal to 0 at ℓ_5, so that the division by *sgn* is safe; moreover, it allows proving that the absolute value of x computed in y is always positive.

The System P_2 displayed in Figure 2(c) is also a complete partition of P_0. It amounts do performing a similar partitioning of the conditional structure without merging the traces at point ℓ_4. Such a partitioning would be more costly if applied to many **if**-statements in a large program.

In fact, we can also note that P_2 is a complete partition of P_1.

Remark 3.2.4. (Extending the notion of covering) We may extend the definition of covering, by replacing the τ function with a relation $(\Rightarrow_\tau) \subseteq T \times T'$. Then, the function $\pi_\tau^{\mathbb{L}}$ becomes a relation $(\Rightarrow_\tau^{\mathbb{L}}) \subseteq \mathbb{L}_T \times \mathbb{L}_{T'}$.

Intuitively, $t \Rightarrow_\tau^{\mathbb{L}} t'$ means that the token t is "simulated" by t' in $P_{T'}$. Clearly, this definition is weaker, since a token t may be simulated by several tokens in $P_{T'}$.

The results in the following would extend to this weaker definition of covering system.

Note that we do not require the set of partitions to be finite. This assumption is not required in order to prove the partitioning correct.

3.2.3 *Trivial Extension.* We let $t_\epsilon \in \mathbb{T}$ and write $T_\epsilon = \{t_\epsilon\}$. The *trivial extension* of P is the extended system $P_\epsilon = (\mathbb{L}_\epsilon, \mathbb{S}_\epsilon^{\mathrm{i}}, \rightarrow_\epsilon)$, where:

— $\mathbb{L}_\epsilon = \mathbb{L} \times T_\epsilon$;
— $\mathbb{S}_\epsilon^{\mathrm{i}} = \{((\ell, t_\epsilon), \rho) \mid (\ell, \rho) \in \mathbb{S}^{\mathrm{i}}\}$;
— $((\ell_0, t_\epsilon), \rho) \rightarrow_\epsilon ((\ell_1, t_\epsilon), \rho) \iff (\ell_0, \rho) \rightarrow (\ell_1, \rho)$.

This extended system is isomorphic to P (the traces of both programs are equal up to isomorphism); it is the "simplest" extension of P. We write $\pi_{\epsilon}^{\$^\star}$ for the trivial mapping of traces of P_ϵ into traces of P.

3.3 Soundness of Control Partitioning

The goal of Section 4 is to define an abstraction as the data of a partition (or covering) and an abstraction of the semantics of the corresponding extended system. Therefore, in the two following subsections, we set up an ordering, so as to compare the semantics of partitioned systems and build an ordering among partitioned systems (these are two crucial requirements, before we can set up the trace partitioning domains in Section 4).

The semantics of extended systems is defined in the usual way, as in Section 2.1.2. Furthermore, we propose to partition the semantics with the partitioned control states *including* the token (i.e., we choose $E = \mathbb{L}_T = \mathbb{L} \times T$), of the last state in the traces, which amounts to applying the same abstraction as $\alpha_{\mathcal{P}(\$)}$ (Section 3.1) in the case of the extended system:

Definition 3.3.1. (PARTITIONED SEMANTICS) If P_T is the extended system defined by $(T, \$^{\mathrm{i}}_T, \rightarrow_T)$, we let $[\![P_T]\!]^{\mathrm{p}}$ be the partitioned semantics defined by:

$$[\![P_T]\!]^{\mathrm{p}} = \alpha_{\mathfrak{P}(\delta_{\mathbb{L}_T})}([\![P_T]\!])$$

where $\delta_{\mathbb{L}_T}$ is defined by:

$$\begin{array}{llll} \delta_{\mathbb{L}_T} : & \mathcal{P}(\$^\star) & \rightarrow & (\mathbb{L}_T \rightarrow \mathcal{P}(\$^\star)) \\ & \mathcal{E} & \mapsto & \lambda((\ell, t) \in \mathbb{L}_T) \cdot \{\sigma \in \mathcal{E} \mid \exists \rho \in \mathbb{M},\ \sigma = \langle \ldots, ((\ell, t), \rho)\rangle\} \end{array}$$

The properties of covering (resp. partitioning, complete) systems extend to their semantics, as pointed out in the following lemma (the definitions for covering, partitioning and complete extended systems were designed so as to achieve these properties): for instance, a complete partition P_T of $P_{T'}$ provides a unique counterpart σ for any trace σ' of $P_{T'}$. In the following, we consider the programs $P_T = (T, \$^{\mathrm{i}}_T, \rightarrow_T)$ and $P_{T'} = (T', \$^{\mathrm{i}}_{T'}, \rightarrow_{T'})$, and $\tau : T \rightarrow T'$.

LEMMA 3.3.1. (SEMANTIC ADEQUACY – TRACES) *Then:*

— *If P_T is a τ-covering of $P_{T'}$, then:*

$$\forall \ell' \in \mathbb{L}_{T'},\ \forall \sigma' \in [\![P_{T'}]\!]^{\mathrm{p}}(\ell'),\ \exists \ell \in \mathbb{L}_T,\ \begin{cases} \ell' = \pi_\tau^{\mathbb{L}}(\ell) \\ \exists \sigma \in [\![P_T]\!]^{\mathrm{p}}(\ell),\ \sigma' = \pi_\tau^{\$^\star}(\sigma) \end{cases}$$

— *If P_T is a τ-partition of $P_{T'}$, then:*

$$\forall \ell' \in \mathbb{L}_{T'},\ \forall \sigma' \in [\![P_{T'}]\!]^{\mathrm{p}}(\ell'),\ \exists!(\ell, \sigma) \in \mathbb{L}_T \times \$^\star_T,\ \begin{cases} \ell' = \pi_\tau^{\mathbb{L}}(\ell) \\ \sigma \in [\![P_T]\!]^{\mathrm{p}}(\ell), \\ \sigma' = \pi_\tau^{\$^\star}(\sigma) \end{cases}$$

— *If P_T is τ-complete with respect to $P_{T'}$, then:*

$$\forall \ell \in \mathbb{L}_T,\ \forall \sigma \in [\![P_T]\!]^{\mathrm{p}}(\ell),\ \pi_\tau^{\$^\star}(\sigma) \in [\![P_{T'}]\!]^{\mathrm{p}}(\pi_\tau^{\mathbb{L}}(\ell))$$

PROOF. The proofs for these properties are similar, so we consider the last one only.

Therefore, we assume that P_T is τ-complete with respect to $P_{T'}$, and that $\ell \in \mathbb{L}_T, \sigma \in [\![P_T]\!]^{\mathrm{p}}(\ell)$, and we attempt to prove that $\pi_\tau^{\$^\star}(\sigma) \in [\![P_{T'}]\!]^{\mathrm{p}}(\pi_\tau^{\mathbb{L}}(\ell))$.

We write $\sigma = \langle s_0, \ldots, s_n \rangle$ and $\forall i,\ s_i' = \pi_\tau^{\$}(s_i)$ (so that $\sigma' = \langle s_0', \ldots, s_n' \rangle = \pi_\tau^{\$^\star}(\sigma)$).

— First, we prove by induction on the length of σ that $\sigma' \in [\![P_{T'}]\!]$:
— $s_0 \in \$^{\mathrm{i}}_T$; since P_T is τ-complete with respect to $P_{T'}$, $s_0' = \pi_\tau^{\$}(s_0) \in \$^{\mathrm{i}}_{T'}$;
— Let $i \in \mathbb{N},\ 0 \leq i < n$. Since $\sigma \in [\![P_T]\!]$, $s_i \rightarrow_T s_{i+1}$; hence, $s_i' \rightarrow_{T'} s_{i+1}'$, because P_T is τ-complete with respect to $P_{T'}$.

— Second, we prove that $\pi_\tau^{\$^\star}(\sigma) \in [\![P_{T'}]\!]^{\mathrm{p}}(\pi_\tau^{\mathbb{L}}(\ell))$: since $\sigma \in [\![P_T]\!]^{\mathrm{p}}(\ell)$, $\sigma \in [\![P_T]\!]$; hence, $\pi_\tau^{\$^\star}(\sigma) \in [\![P_{T'}]\!]$ (as proved in the first point). Moreover, σ' ends at point $\pi_\tau^{\mathbb{L}}(\ell)$, since $s_n' = \pi_\tau^{\$}(s_n)$. Hence, $\pi_\tau^{\$^\star}(\sigma) = \sigma' \in [\![P_{T'}]\!]^{\mathrm{p}}(\pi_\tau^{\mathbb{L}}(\ell))$

The cases of partitioning and covering systems are similar. $\square$

Let Γ_τ be the function defined as:

$$\begin{array}{llll} \Gamma_\tau : & (\mathbb{L}_T \rightarrow \mathcal{P}(\$^\star_T)) & \rightarrow & (\mathbb{L}_{T'} \rightarrow \mathcal{P}(\$^\star_{T'})) \\ & \Phi & \mapsto & \lambda(\ell' \in \mathbb{L}_{T'}) \cdot \bigcup\{\pi_\tau^{\$^\star}(\Phi(\ell)) \mid \ell \in \mathbb{L}_T,\ \tau(\ell) = \ell'\} \end{array}$$

Here are a few trivial properties of the Γ_τ functions:

Lemma 3.3.2. (Properties of Γ_τ) *For all τ, Γ_τ is monotone.*
If $\tau_0 : T_0 \rightarrow T_1,\ \tau_1 : T_1 \rightarrow T_2$, then $\Gamma_{\tau_1 \circ \tau_0} = \Gamma_{\tau_1} \circ \Gamma_{\tau_0}$.

The following theorem comes as a straightforward consequence of Lemma 3.3.1; it is an important step in proving the soundness of the partitioning abstractions.

Theorem 3.3.2. (Semantic adequacy) *With the above notations:*

— *If P_T is a τ-partition or a τ-covering of $P_{T'}$, then* $[\![P_{T'}]\!]^{\mathrm{p}} \subseteq \Gamma_\tau([\![P_T]\!]^{\mathrm{p}})$ (soundness).

— *If P_T is τ-complete with respect to $P_{T'}$, then* $\Gamma_\tau([\![P_T]\!]^{\mathrm{p}}) \subseteq [\![P_{T'}]\!]^{\mathrm{p}}$ (completeness).

— *Hence, if P_T is a τ-complete partition of $P_{T'}$, or a τ-complete covering of $P_{T'}$, then* $[\![P_{T'}]\!]^{\mathrm{p}} = \Gamma_\tau([\![P_T]\!]^{\mathrm{p}})$ (adequacy).

— *If P_T is a partitioning system of $P_{T'}$, then:*

$$\forall \ell, \ell' \in \mathbb{L}_T,\ \ell \neq \ell' \Longrightarrow \Gamma_\tau([\![P_T]\!]^{\mathrm{p}})(\ell) \cap \Gamma_\tau([\![P_T]\!]^{\mathrm{p}})(\ell) = \emptyset$$

3.4 Pre-Ordering Properties of Partitions

In the following, we use an ordering among partitions. Therefore, we study the pre-ordering properties of the following relations, among extended transition systems:

— "is a covering of" (for some forget function τ);
— "is a partition of" (for some forget function τ);
— "is complete with respect to" (for some forget function τ).

Then, we can prove that, any such ordering $\preccurlyeq$ is transitive:

Lemma 3.4.1. (Transitivity) *Let P_T be $(T, \$^{\mathrm{i}}_T, \rightarrow_T)$, $P_{T'}$ be $(T', \$^{\mathrm{i}}_{T'}, \rightarrow_{T'})$, and $P_{T''}$ be $(T'', \$^{\mathrm{i}}_{T''}, \rightarrow_{T''})$. Furthermore, we consider the forget functions $\tau : T \rightarrow T'$, and $\tau' : T' \rightarrow T''$. Then:*

— *if P_T is a τ-covering (resp. τ-partition) of $P_{T'}$ and $P_{T'}$ is a τ'-covering (resp. τ-partition) of $P_{T''}$, then P_T is a $(\tau' \circ \tau)$-covering (resp. $(\tau' \circ \tau)$-partition) of $P_{T''}$.*

— *if P_T is τ-complete with respect to $P_{T'}$ and $P_{T'}$ is τ'-complete with respect to $P_{T''}$, then P_T is $(\tau' \circ \tau)$-complete with respect to $P_{T''}$.*

Proof. We can first remark that $\pi^{\mathbb{L}}_{\tau' \circ \tau} = \pi^{\mathbb{L}}_{\tau'} \circ \pi^{\mathbb{L}}_{\tau}$ (and similarly for the other forget functions).

Let us prove the second point (transitivity of completeness).

— Let $s \in \mathbb{S}^{\mathrm{i}}_T$. Then, $\pi^{\mathbb{S}}_{\tau}(s) \in \mathbb{S}^{\mathrm{i}}_{T'}$, since P_T is τ-complete with respect to $P_{T'}$. Moreover, $\pi^{\mathbb{S}}_{\tau' \circ \tau}(s) = \pi^{\mathbb{S}}_{\tau'} \circ \pi^{\mathbb{S}}_{\tau}(s) \in \mathbb{L}^{\mathrm{i}}_{T''}$, since $P_{T'}$ is τ'-complete with respect to $P_{T''}$.

— Let $s_0, s_1 \in \mathbb{S}_T$, such that $s_0 \rightarrow_T s_1$. Again, we apply successively the two assumptions of completeness and derive $\pi^{\mathbb{S}}_{\tau}(s_0) \rightarrow_{T'} \pi^{\mathbb{S}}_{\tau}(s_1)$ (since P_T is τ-complete with respect to $P_{T'}$ and then $\pi^{\mathbb{S}}_{\tau' \circ \tau}(s_0) \rightarrow_{T''} \pi^{\mathbb{S}}_{\tau' \circ \tau}(s_1)$, since $P_{T'}$ is τ'-complete with respect to $P_{T''}$.

The proof of the first point is similar. □

Moreover, the relations mentioned above are clearly reflexive

Lemma 3.4.2. (Reflexivity) *Let $P_T = (T, \mathbb{S}^{\mathrm{i}}_T, \rightarrow_T)$ and $\tau : T \rightarrow T;\ t \mapsto t$. Then, clearly P_T is a τ-covering (resp. partition) of P_T and P_T is τ-complete with respect to itself.*

Such an ordering should allow to compare the *precision* of partitions (yet, note that the more precise partition is the greater element, instead of the smaller, as is usually the case in static analysis) and to define valid *computational orderings* [Cousot and Cousot 1992b], which we will illustrate in the next section.

4. TRACE PARTITIONING DOMAINS

In this section, we address the design of trace partitioning domains. Basically, an element of such a domain defines a partition of the initial system, together with a semantic denotation, relative to *this* partition. In the concrete level, this denotation associates sets of traces to each control state; in the concrete level it maps control states into local invariants.

After we define the domains, we also discuss the design of widening operators and static analyses using trace partitioning domains.

4.1 The Trace Partitioning Domain

First, we consider the definition of a *concrete* trace partitioning domain.

4.1.1 *Definition of the Basis.* In this section, we assume that a transition system $P = (\mathbb{L}, \mathbb{S}^{\mathrm{i}}, \rightarrow)$ is given, and we consider the complete coverings of P; we write $\mathfrak{B}$ for the set of the extended systems which satisfy these properties.

First, we let $\preccurlyeq$ be the order among extended systems defined by:

$$P_{T_0} \preccurlyeq P_{T_1} \iff \exists \tau : T_1 \rightarrow T_0,\ P_{T_1} \text{ is a } \tau\text{-covering}$$

As remarked in Section 3.4, we may choose other definitions for $\preccurlyeq$, such as:

$$P_{T_1} \preccurlyeq P_{T_1} \iff \exists \tau : T_1 \rightarrow T_0, \begin{cases} P_{T_1} \text{ is a } \tau\text{-partition of } P_{T_0} \\ P_{T_1} \text{ is } \tau\text{-complete with respect to } P_{T_0} \end{cases}$$

In case the property on the right side is satisfied, we also write $P_{T_0} \preccurlyeq_\tau P_{T_1}$ for τ, so as to make τ explicit.

The trivial extension of P is clearly the least element of $\mathfrak{B}$ for $\preccurlyeq$.

Note that other choices for $\mathfrak{B}$ and $\preccurlyeq$ could have been made and would have allowed to prove the same results in the following.

Example 4.1.1. (The ordering over the basis) We showed in Example 3.2.3 that the systems P_0, P_1 and P_2 are such that:

$$P_0 \preccurlyeq P_1 \preccurlyeq P_2$$

4.1.2 *The Domain.* At this point we can define the trace partitioning domain. An element of this domain should denote:

— a covering P_T of the original transition system;

— and a semantic denotation for each control state ℓ of the covering P_T:
— in the basic domain, this denotation shall be a set of traces ending at point ℓ);
— in the abstract domain, this denotation shall be an invariant in $D^\sharp_{\mathbb{M}}$.

More formally:

Definition 4.1.2. (Trace partitioning domain) An element of the trace partitioning domain is a tuple (T, P_T, Φ), where:

— $T \in \mathfrak{T}$;

— P_T denotes a complete covering $(T, \mathbb{S}^{\mathrm{i}}_T, \rightarrow_T)$ of P;

— Φ is a function $\Phi : \mathbb{L}_T \rightarrow \mathcal{P}(\mathbb{S}^\star_T)$.

We write $\mathbb{D}$ for the set of such tuples.

Let $(T_0, P_{T_0}, \Phi_0), (T_1, P_{T_1}, \Phi_1) \in \mathbb{D}$. Then, we write $(T_0, P_{T_0}, \Phi_0) \leqslant_\tau (T_1, P_{T_1}, \Phi_1)$ –or, for short $(T_0, P_{T_0}, \Phi_0) \leqslant (T_1, P_{T_1}, \Phi_1)$– if and only if:

— $P_{T_0} \preccurlyeq_\tau P_{T_1}$ for τ;

— $\Phi_0 \subseteq \Gamma_\tau(\Phi_1)$.

It follows from the results presented in Section 3.4 that $\leqslant$ defines a pre-ordering on $\mathbb{D}$.

4.1.3 *The Concretization Function.* The concretization of an element (T, P_T, Φ) of $\mathbb{D}$ is a set of traces of the initial system, which is computed by:

(1) merging all the partitions together, by projecting Φ onto the trivial extension P_ϵ of P (i.e., applying function Γ_{τ_ϵ}) and then collapsing the partitions with $\gamma_{\mathfrak{P}(\mathbb{L}_T)}$;

(2) applying the isomorphism $\pi^{\mathbb{S}^\star}_\epsilon$ between traces of P_ϵ and P.

It is defined formally in the following definition:

Definition 4.1.3. (Concretization function) We let $\gamma_{\mathbb{P}}$ be the concretization function defined by

$$\gamma_{\mathbb{P}} = \pi^{\mathbb{S}^\star}_\epsilon \circ \gamma_{\mathfrak{P}(\mathbb{L}_T)} \circ \Gamma_{\tau_\epsilon}$$

Or equivalently, by:

$$\begin{array}{rlcl} \gamma_{\mathbb{P}} : & \mathbb{D} & \rightarrow & \mathbb{S}^\star \\ & (T, P_T, \Phi) & \mapsto & \{\pi^{\mathbb{S}^\star}_\epsilon(\sigma) \mid \exists \ell \in \mathbb{L}_T,\ \sigma \in \Phi(\ell)\} \end{array}$$

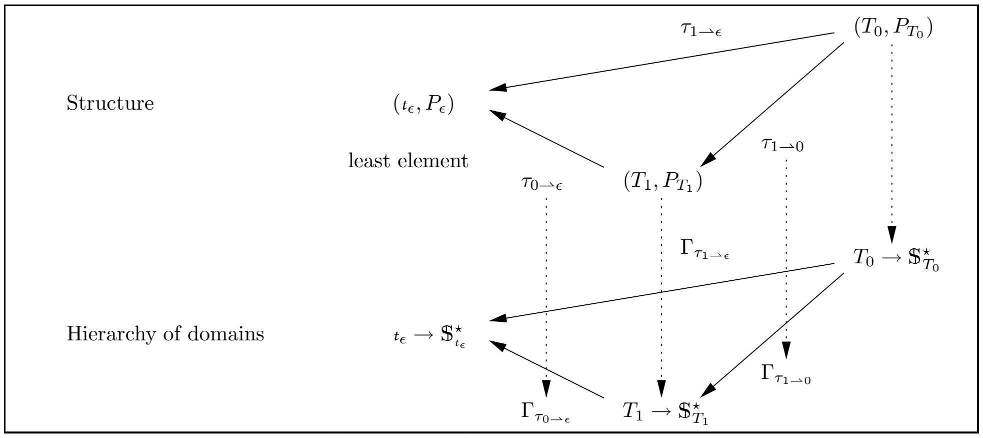

$\tau_{1\to\epsilon}$
(T_0, P_{T_0})
Structure
(t_ϵ, P_ϵ)
$\tau_{1\to 0}$
least element
$\tau_{0\to\epsilon}$
(T_1, P_{T_1})
$\Gamma_{\tau_{1\to\epsilon}}$
$T_0 \to \mathbb{S}^\star_{T_0}$
Hierarchy of domains
$t_\epsilon \to \mathbb{S}^\star_{t_\epsilon}$
$\Gamma_{\tau_{1\to 0}}$
$\Gamma_{\tau_{0\to\epsilon}}$
$T_1 \to \mathbb{S}^\star_{T_1}$

Fig. 3: Structure of the partitioning domain

Clearly, this function is monotone.

Example 4.1.4. (TRACE PARTITIONING DOMAIN) Let us consider the case of the systems introduced in Example 3.2.3. An element of the domain is characterized by the data of a system and denotation corresponding to this element. For instance, if the element of the basis is P_1, then the last element of the tuple should be a function mapping each element of the set $\{(\ell_0, t_0), (\ell_1, t_1), (\ell_3, t_1), (\ell_2, t_2), (\ell_3, t_2), (\ell_4, t_0)\}$ into a set of traces.

The concretization removes all the tokens; hence, it generates sets of traces of the initial system (which is isomorphic to P_0).

4.1.4 *Soundness of the Partitioned Systems.* A last, trivial yet very important remark is that the partitioning of the initial system is sound:

THEOREM 4.1.5. (SOUNDNESS OF CONTROL PARTITIONING) *Let* (T_0, P_{T_0}) *and* $(T_1, P_{T_1}) \in \mathfrak{T} \times \mathfrak{B}$*, such that* $P_{T_0} \preccurlyeq_\tau P_{T_1}$*. Then,* $(T_0, P_{T_0}, [\![P_{T_0}]\!]^{\mathrm{P}}) \leqslant_\tau (T_1, P_{T_1}, [\![P_{T_1}]\!]^{\mathrm{P}})$.

In particular, in case $(T_0, P_{T_0}) = (T_\epsilon, P_\epsilon)$*, then we get the soundness with respect to the* original *transition system:* $[\![P]\!] \subseteq \gamma_{\mathbb{P}}(T_1, P_{T_1}, [\![P_{T_1}]\!]^{\mathrm{P}})$.

PROOF. The first point follows from Theorem 3.3.2; the second is a corollary of the first point. □

This domain structure can be related to the cofibered domain structure defined in [Venet 1996]. More precisely, the element of the basis fixes a partition of the original system, and the last argument of the tuple corresponding to an element of the domain $\mathbb{D}$ provides a semantic denotation defined in a domain relative to the basis element. Figure 3 gives an overall intuition about the structure of the partitioning domain $\mathbb{D}$.

The presentation in [Venet 1996] relies on categories; we use orderings instead, but the principle is similar: the structure of the basis provides the frame for a hierarchy of domains. The comparison of elements across different domains can be done thanks to the projection functions Γ_τ provided by the ordering on the basis.

4.1.5 *Gain in Precision.* Let $(T, P_T, \Phi) \in \mathbb{D}$ be an element of the domain. This element describes the *same* set of traces as the initial program P. However, it allows for a more precise description of sets of traces ending at each control state than the usual abstractions (i.e., the $\alpha_{\mathcal{P}(\mathbb{S})}$ abstraction defined in Section 3.1), if there exists a control state $\ell \in \mathbb{L}$, $\sigma, \sigma' \in [\![P]\!]$, such that σ and σ' both end at ℓ but are not in the same partitions, when mapped into the extended system P_T. This gain in precision really pays off, when a further abstraction (such as the abstraction defined by $\gamma_{\mathbb{M}}$) is composed, as done in the next subsection.

4.1.6 *Comparison with Other Approaches to Partitioned Systems.* Our approach considerably generalizes the trace partitioning technique of [Handjieva and Tzolovski 1998], since we leave the choice of partitions as a *parameter*: various partitioning strategies can be implemented (for instance, we allow the merge of partitions).

The path sensitive techniques [Holley and Rosen 1980] proposed in data flow analysis context do not allow for *abstractions of sets of paths* to be considered. In our settings, a token stands for an approximation for a set of paths, which renders the design of analyses more flexible.

Other authors proposed to perform a partitioning of memory states or to convert part of the data into control structures, as can be done for booleans [Jeannet et al. 1999]. However, this solution presents several drawbacks in our opinion. In particular, the relations partitions are based on may not be found straightforwardly in the memory states; in the other hand, a partitioning guided by the conditions is rather intuitive. Another drawback comes from the fact that the method exposed in [Jeannet et al. 1999] is based on a refinement process, which would not be so effective in the case of the ASTRÉE analyzer. By contrast this approach seem to be more effective for the analysis of synchronous programs.

The following subsections express fundamental properties of $\mathbb{D}$:

— composition of further abstractions (such as the abstraction of sets of stores into collections of predicates), in Section 4.2;

— application to static analysis and definition of widening operators on such domains, in Section 4.3;

— implementation of efficient analyzers in Section 4.4.

4.2 Composing Store Abstraction

We now propose to design an *abstract* trace partitioning domain.

We derive from Definition 4.1.2 the definition of a new partitioning abstraction, by abstracting sets of stores into collections of constraints in the same way as in Section 2.2. Therefore, we assume that an abstraction $(D^\sharp_{\mathbb{M}}, \sqsubseteq)$ is defined for representing sets of stores is defined, together with a concretization function $\gamma_{\mathbb{M}} : D^\sharp_{\mathbb{M}} \to \mathcal{P}(\mathbb{M})$, which defines the meaning of a set of abstract constraint as the set of stores which satisfy them.

The partitioning abstract domain is derived from $\mathbb{D}$ by replacing functions mapping extended labels into sets of traces with functions mapping extended labels into elements of $D^\sharp_{\mathbb{M}}$:

Definition 4.2.1. (PARTITIONING ABSTRACT DOMAIN) An element of the partitioning abstract domain is a tuple $(T, P_T, \Phi^\sharp)$, where:

— $T \in \mathfrak{T}$;

— P_T is a complete covering of P $(T, \mathbb{S}^{\mathrm{i}}_T, \to_T)$;

— $\Phi^\sharp$ is a function $\Phi^\sharp : \mathbb{L}_T \to D^\sharp_{\mathbb{M}}$.

We write $\mathbb{D}^\sharp$ for the set of such tuples.

Remark 4.2.2. (Representation of abstract values) An abstract value is a value in $\mathbb{L}_T \to D^\sharp_{\mathbb{M}} = (\mathbb{L} \times T) \to D^\sharp_{\mathbb{M}}$. By curryfication; it is isomorphic to a value in $\mathbb{L} \to (T \to D^\sharp_{\mathbb{M}})$. This latter representation turns out to be very natural in practice: each control state corresponds to an abstract value in the partitioning domain $D^\sharp_{\mathbb{P},\mathbb{M}} = T \to D^\sharp_{\mathbb{M}}$, mapping partitioning tokens into sets of stores; hence, it allows to describe precisely the partitions associated to each program point.

The ordering is also inherited from Definition 4.1.2. Indeed, we let:

$$\begin{array}{llll}\Gamma^\sharp_\tau : & (\mathbb{L}_T \to D^\sharp_{\mathbb{M}}) & \to & (\mathbb{L}_{T'} \to D^\sharp_{\mathbb{M}}) \\ & \Phi^\sharp & \mapsto & \lambda(\ell' \in \mathbb{L}_{T'}) \cdot \bigsqcup\{\Phi(\ell) \mid \ell \in \mathbb{L}_T,\ \tau(\ell) = \ell'\}\end{array}$$

If the join operator $\sqcup$ of $D^\sharp_{\mathbb{M}}$ is not associative, commutative, the definition of $\Gamma^\sharp_\tau$ would not be unique, which would cause various technical complications; therefore, we assume that $\sqcup$ is associative and commutative in our presentation. Then:

Definition 4.2.3. (Ordering) Let $(T_0, P_{T_0}, \Phi^\sharp_0), (T_1, P_{T_1}, \Phi^\sharp_1) \in \mathbb{D}$, and a function $\tau : T_1 \to T_0$. Then, we write $(T_0, P_{T_0}, \Phi^\sharp_0) \leqslant^\sharp_\tau (T_1, P_{T_1}, \Phi^\sharp_1)$ (or, for short $(T_0, P_{T_0}, \Phi^\sharp_0) \leqslant^\sharp (T_1, P_{T_1}, \Phi^\sharp_1)$) if and only if:

— $P_{T_0} \preccurlyeq_\tau P_{T_1}$ for τ;

— $\Phi^\sharp_0 \sqsubseteq \Gamma^\sharp_\tau(\Phi^\sharp_1)$.

It follows from the results presented in Section 3.4 that $\leqslant$ defines a pre-ordering on $\mathbb{D}$.

The concretization of an element of $\mathbb{D}^\sharp$ into an element of $\mathbb{D}$ applies the concretization function $\gamma_{\mathbb{M}}$ pointwise, i.e. by applying it to $\Phi^\sharp$.

Definition 4.2.4. (Concretization)

$$\begin{array}{llll}\gamma^\sharp_{\mathbb{P}} : & \mathbb{D}^\sharp & \to & \mathbb{D} \\ & (T, P_T, \Phi^\sharp) & \mapsto & (T, P_T, \lambda(\ell \in \mathbb{L}_T) \cdot \gamma_{\mathbb{M}} \circ \Phi^\sharp(\ell))\end{array}$$

We remark, that $(T, P_T, \Phi^\sharp)$ may provide a better approximation of $[\![P]\!]$ than an element in $D^\sharp = \mathbb{L} \to D^\sharp_{\mathbb{M}}$ whenever the extended systems distinguishes traces of P, i.e., if there exists a control state ℓ, and $\sigma, \sigma' \in [\![P]\!]$ such that σ and σ' both end at ℓ and are in different partitions, when mapped into traces of P_T.

In the other hand, any approximation for $[\![P]\!]$ in $D^\sharp$ can be translated in an *equivalent* abstraction in $(T, P_T, \Phi^\sharp)$, for *any* choice of (T, P_T). As a consequence, we expect the partitioning domain to provide results at least as good as the non partitioning domain, and strictly better results when the (T, P_T) allows to distinguish real traces of P.

At this point, we can state a few remarks, which should give a better understanding of the structure of the partitioning domain.

Remark 4.2.5. ($\preccurlyeq$ is a computational ordering) The ordering introduced in Definition 4.2.3 is essentially a computational ordering [Cousot and Cousot 1992b].

Indeed, an analysis starts with a coarse partition, defined by the program control structure and then may perform some refinements of the system. When a refinement is performed, the basis element is replaced with a greater element, and so is the current abstract invariant. Therefore, the abstract computation should produce monotone sequences of elements for the ordering of Definition 4.2.3.

Next subsection proposes the definition of an extrapolation operator based on the same computational order.

Remark 4.2.6. (Direction of the ordering on the basis) We pointed out that the ordering among elements of the basis is an *inverse* for the "refinement" ordering in the end of Section 3.4: the greater for $\preccurlyeq$, the more refined the partition.

Therefore, one may suggest using the opposite ordering, so that smaller elements represent finer partitions, but:

— The inverse of $\preccurlyeq$ would not capture the precision ordering better than $\preccurlyeq$. Indeed, the precision ordering is usually defined as the ordering of the concretizations; and, we may have $(T_0, P_{T_0}, \Phi_0^\sharp) \lessdot^\sharp (T_1, P_{T_1}, \Phi_1^\sharp)$ even though P_{T_0} and P_{T_1} are not comparable for $\preccurlyeq$; opposing the ordering on the basis would not change anything here. In fact, the definition of the precision ordering in $\mathbb{D}$ would be much more complicated (and not interesting when designing static analyses)·

— It would be possible to write the analysis so that it starts with a *completely partitioned system* (which may not be easy to define, depending on the instantiation of the partitioning framework) and use the opposite ordering as a computational ordering also (the analysis should merge partitions so as to ensure termination): however, we found this idea less intuitive; in particular, it is easier to reason about *creating* partitions instead of not deleting partitions.

4.3 Static Analysis with Partitioning and Widening Operator

The domain introduced in Section 4.2 allows to carry out a static analysis of P, with a partitioning domain. However, several approaches to such analyses are feasible:

— **static partitioning** relies on the choice of a fixed partition;

— **dynamic partitioning** allows for the partition to be changed during the static analysis.

The latter approach is more powerful but may also result in a more involved implementation. In particular, in case infinitely many partitions might be chosen and different partitions can be used for successive iterations in an abstract fixpoint computation, the termination of the analysis shall be enforced by the use of a *widening* operator. For instance, it may start analyzing a loop by unrolling the first iterates and decide to give up the unrolling at some point, so as to guarantee termination of the analysis.

The definition of a widening operator on $\mathbb{D}^\sharp$ is necessary when infinite or very large sets of partitions shall be used, and when (quick) termination is required, e.g. for static analysis. This issue would not occur in case the set of partitions was chosen once for all.

We propose to define a widening operator for $\mathbb{D}^\sharp$ by:

— choosing a widening $\nabla_{\mathbb{M}}$ over $D_{\mathbb{M}}^\sharp$;

— choosing a widening $\nabla_{\mathfrak{B}}$ over the basis (in the sense of Definition 2.2.2);

— defining a pairwise widening over $\mathbb{D}^\sharp$.

Formally, the widening operator for the partitioning domain is defined by:

Definition 4.3.1. (Widening for the partitioning domain) If $(T_0, P_{T_0}, \Phi_0^\sharp)$, $(T_1, P_{T_1}, \Phi_1^\sharp) \in \mathbb{D}$, then, we let:

$$(T_0, P_{T_0}, \Phi_0^\sharp)\nabla_{\mathrm{P}}(T_1, P_{T_1}, \Phi_1^\sharp) = (T_2, P_{T_2}, \Phi_2^\sharp)$$

where:

— $P_{T_2} = P_{T_0}\nabla_{\mathfrak{B}}P_{T_1}$, so that $P_{T_0} \preccurlyeq_{\tau_0} P_{T_2}$ and $P_{T_1} \preccurlyeq_{\tau_1} P_{T_2}$;

— $\Phi_2^\sharp = (\Phi_0^\sharp \circ \tau_0)\nabla_{\mathbb{M}}(\Phi_1^\sharp \circ \tau_1)$ (pointwise application of $\nabla_{\mathbb{M}}$ to elements of $\mathbb{L}_{T_2} \rightarrow D_{\mathbb{M}}^\sharp$).

Indeed, this approach leads to a widening over the partitioning abstract domain, as shown in the following theorem:

Theorem 4.3.2. (Widening for partitioning domains) *The operator ∇_{P} is a widening operator on $\mathbb{D}$, in the sense of Definition 2.2.2.*

Proof. Proving point 1 in Definition 2.2.2 is straightforward, so we consider point 2.

Let $(T_n, P_{T_n}, \Phi_n^\sharp)_{n\in\mathbb{N}}$ be a sequence elements of $\mathbb{D}$, and $(T_n', P_{T_n'}, \Phi_n'^\sharp)_{n\in\mathbb{N}}$ be defined as:

$$\begin{aligned}(T_0', P_{T_0'}, \Phi_0'^\sharp) &= (T_0, P_{T_0}, \Phi_0^\sharp)\\ (T_{n+1}', P_{T_{n+1}'}, \Phi_{n+1}'^\sharp) &= (T_n', P_{T_n'}, \Phi_n'^\sharp)\nabla_{\mathrm{P}}(T_n, P_{T_n}, \Phi_n^\sharp)\end{aligned}$$

Then:

— by definition of the widening over the basis $\nabla_{\mathfrak{B}}$, the element of the basis stabilizes after a finite number of iterations: $\exists n \in \mathbb{N},\ \forall m \in \mathbb{N},\ m \geq n \Longrightarrow P_{T_m} = P_{T_n}$.

— if we consider the subsequence $(T_m', P_{T_m'}, \Phi_m'^\sharp)_{m\in\mathbb{N}, m\geq n}$, then $\forall m \geq n,\ T_m' = T_n' \wedge P_{T_m'} = P_{T_n'}$ and the sequence of the last arguments form a widening sequence in $\mathbb{L}_{T_n'} \rightarrow D_{\mathbb{M}}^\sharp$; $\mathbb{L}_{T_n'}$ is finite and $\nabla_{\mathbb{M}}$ is a widening over $D_{\mathbb{M}}^\sharp$, therefore this sequence is ultimately stationary.

This proves that the sequence $(T_n', P_{T_n'}, \Phi_n'^\sharp)_{n\in\mathbb{N}}$ is ultimately stationary; hence, ∇_{P} is a widening operator over $\mathbb{D}$. □

Again, the proof of the widening operator can be compared with the definition of a widening on cofibered domains [Venet 1996]. Basically, a widening operator for $\mathbb{D}$ should stabilize the basis first (i.e., enforce the termination of the refinement of the partition), and then stabilize the image in the abstract domain $D_{\mathbb{M}}^\sharp$; therefore, an alternate definition for ∇_{P} would delay the widening in $D_{\mathbb{M}}^\sharp$ until the element of the basis reaches a limit.

4.4 Denotational Style Partitioning Static Analysis

The design of static analyzers as abstractions of the denotational semantics of statements was proposed in Section 2.3. In particular, we showed that this design allows for natural and efficient iteration strategies. Therefore, we propose to adapt this scheme to partitioning analyses.

4.4.1 *Partitioning Denotational Semantics.* First, we apply the "from point to point" denotational abstraction $\alpha_{t\mathcal{F}[\ell_\vdash,\ell_\dashv]}$.

More precisely, we consider in this subsection an extended system P_T, such that $P \leqslant_\tau P_T$, and let $\ell_\vdash, \ell_\dashv \in \mathbb{L}$. The concrete denotational semantics from $\ell_\vdash$ to $\ell_\dashv$ maps an "input" state at $\ell_\vdash$ to the set of possible "output" states at $\ell_\dashv$. Hence, the denotational semantics in the extended system should map tuples made of a partitioning token and a store into similar tuples:

Definition 4.4.1. (Partitioned denotational semantics) We define the abstraction function $\alpha_{t\mathcal{F}\,\mathbb{P}[\ell_\vdash,\ell_\dashv]} : \mathcal{P}(\mathbb{S}^\star) \to ((\mathbb{T}\times\mathbb{M}) \to \mathcal{P}(\mathbb{T}\times\mathbb{M}))$, where $\alpha_{t\mathcal{F}\,\mathbb{P}[\ell_\vdash,\ell_\dashv]}(\mathcal{E})$ is defined by:

$$\begin{array}{rlll} \alpha_{t\mathcal{F}\,\mathbb{P}[\ell_\vdash,\ell_\dashv]}(\mathcal{E}) : & (\mathbb{T}\times\mathbb{M}) & \to & \mathcal{P}(\mathbb{T}\times\mathbb{M}) \\ & (\iota_\vdash, \rho_\vdash) & \mapsto & \{(\iota_\dashv, \rho_\dashv) \mid \exists \sigma \in \mathcal{E},\ \sigma = \langle ((\ell_\vdash, \iota_\vdash), \rho_\vdash), \ldots, ((\ell_\dashv, \iota_\dashv), \rho_\dashv)\rangle\} \end{array}$$

We write $\gamma_{t\mathcal{F}\,\mathbb{P}[\ell_\vdash,\ell_\dashv]}$ for the corresponding concretization function.

Last, the *partitioned denotational semantics* is $\alpha_{t\mathcal{F}\,\mathbb{P}[\ell_\vdash,\ell_\dashv]}(\llbracket P_T \rrbracket^{\mathrm{P}})$.

4.4.2 *Abstract Partitioning Denotational Semantics.* The denotational-style static analyzer of Section 2.3 was derived as an abstraction of the denotational semantics; therefore, we propose to derive a static analyzer for the partitioned system in the same way. However, we should note a slight difference: in Definition 4.4.1, an initial state consists in a pair made of a partitioning token and a store. Hence, the abstract semantics follows the same scheme:

Definition 4.4.2. (Partitioned abstract denotational semantics) We write $D^\sharp_{\mathbb{P},\mathbb{M}}$ for $T \to D^\sharp_{\mathbb{M}}$. A function $\llbracket P_T \rrbracket^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]} : D^\sharp_{\mathbb{P},\mathbb{M}} \to D^\sharp_{\mathbb{P},\mathbb{M}}$ is a *sound abstract semantics* of P_T, between $\ell_\vdash$ and $\ell_\dashv$ if and only if:

$$\begin{array}{l} \forall (\iota, \rho), (\iota', \rho') \in \mathbb{T}\times\mathbb{M},\ \forall d_p \in D^\sharp_{\mathbb{P},\mathbb{M}} \\ \left.\begin{array}{l} (\iota', \rho') \in \alpha_{t\mathcal{F}\,\mathbb{P}[\ell_\vdash,\ell_\dashv]}(\llbracket P_T \rrbracket)(\iota, \rho) \\ \rho \in d_p(\iota) \end{array}\right\} \Longrightarrow \rho' \in \llbracket P_T \rrbracket^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]}(d_p)(\iota') \end{array}$$

In this sense, $\llbracket P_T \rrbracket^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]}$ should be an approximation of the denotational semantics introduced in Definition 4.4.1.

The partitioned denotational abstract semantics is sound with respect to the standard semantics of the initial system:

Theorem 4.4.3. (Soundness of the static partitioning analysis) *Let* $(T, P_T) \in \mathfrak{B}$ *such that* $(T_\epsilon, P_\epsilon) \preccurlyeq_\tau (T, P_T)$.

Let $d_t \in D^\sharp_{\mathbb{P},\mathbb{M}}$, $(\iota, \rho) \in \mathbb{T}\times\mathbb{M}$ *such that* $\rho \in d_t(\iota)$. *Moreover, we let* $\rho' \in \alpha_{t\mathcal{F}[\ell_\vdash,\ell_\dashv]}(\llbracket P \rrbracket)(\rho)$. *Then, there exists* ι' *such that:*

$$\rho' \in \llbracket P_T \rrbracket^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]}(d_p)(\iota')$$

Proof. The above result follows from the soundness of the control partitioning (Theorem 4.1.5) and the soundness of the abstract semantics $[\![P_T]\!]^{\sharp}_{\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}$ (Definition 4.4.2). □

In practice, an abstract semantics $[\![P_T]\!]^{\sharp}_{\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}$ is defined in a similar way as the abstract semantics of statements described in Section 2.3, and in Figure 1.

Moreover, we can remark that the abstract semantics $[\![P_T]\!]^{\sharp}_{\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}$ may postpone the computation of abstract joins so as to approximate flows in distinct partitions. This ability allows in many cases for a greater precision (even if a local improvement in precision does not always guarantee a global improvement, since several abstract operators including widening usually are not monotone).

Example 4.4.4. (Denotational style abstraction of a **if**-statement) We consider the program introduced in Example 3.2.3. In particular, this program is equivalent to the transition system P_0, displayed in Figure 2(a). We consider the partition defined by the system P_1 (Figure 2(b)): the analysis partitions the traces depending on the branch of the **if**-statement they visited until point ℓ_4 (the partitions are merged at this point).

We present the static analysis of various statements in this piece of code (the analysis is carried out on P_1):

— statement s_1 (true branch of the conditional): the only partitions before and after this statement is t_1, to $[\![s_1]\!]^{\sharp}_{\mathbb{P}[\ell_1,\ell_3]}$ is a function:

$$[\![s_1]\!]^{\sharp}_{\mathbb{P}[\ell_1,\ell_3]} : (\{t_1\} \to D^{\sharp}_{\mathbb{M}}) \longrightarrow (\{t_1\} \to D^{\sharp}_{\mathbb{M}})$$

(the analysis propagates the partition t_1);

— conditional structure (statement $s = \mathbf{if}(e)\, s_1\, \mathbf{else}\, s_2$): it splits the partition t_0 into two sets of traces corresponding to t_1 and t_2; hence, $[\![s]\!]^{\sharp}_{\mathbb{P}[\ell_1,\ell_3]}$ is a function:

$$[\![s]\!]^{\sharp}_{\mathbb{P}[\ell_1,\ell_3]} : (\{t_0\} \to D^{\sharp}_{\mathbb{M}}) \longrightarrow (\{t_1, t_2\} \to D^{\sharp}_{\mathbb{M}})$$

— statement s_3 (statement right after the conditional): it inputs two partitions corresponding to t_1 and t_2 and outputs similar partitions; however, the partitions are merged right after the analysis of the statement (at point ℓ_4), so we can write down $[\![s_3]\!]^{\sharp}_{\mathbb{P}[\ell_3,\ell_4]}$ as a function:

$$[\![s_3]\!]^{\sharp}_{\mathbb{P}[\ell_3,\ell_4]} : (\{t_1, t_2\} \to D^{\sharp}_{\mathbb{M}}) \longrightarrow (\{t_0\} \to D^{\sharp}_{\mathbb{M}})$$

— the whole program inputs and outputs only one partition, corresponding to t_0, so its abstract semantics is a function:

$$[\![P_1]\!]^{\sharp}_{\mathbb{P}[\ell_1,\ell_3]} : (\{t_0\} \to D^{\sharp}_{\mathbb{M}}) \longrightarrow (\{t_0\} \to D^{\sharp}_{\mathbb{M}})$$

4.4.3 *Making the Partitioning Dynamic.* The above definition introduces a static form of partitioning: the analysis of the statement may not change the partitions, e.g. by refining the system. Therefore, we propose a new definition for an abstract semantics for statements, which may refine the partitions.

First, we define a new partitioning abstract domain for approximating sets of stores *and* partitions:

Definition 4.4.5. (DOMAIN FOR DYNAMIC PARTITIONING) An element of the domain is a tuple (T, P_T, d_T), where:

— $T \in \mathbb{T}$;

— P_T is a complete covering $(T, \$^{\mathrm{i}}_T, \rightarrow_T)$ of the initial system P;

— $d_p \in D^\sharp_{\mathbb{P},\mathbb{M}}$ is such that $\forall \iota \in \mathbb{T} \setminus T,\ d_p(\iota) = \bot$.

We write $D^\sharp_{\delta\mathbb{P},\mathbb{M}}$ for this domain; the ordering is the pointwise extension of the orderings on the basis and on $D^\sharp_{\mathbb{M}}$.

The latter condition ensures that d_p assigns invariants to "relevant" tokens only: the invariant corresponding to a token not in T (i.e., not in the current extended system) should be $\bot$.

A partitioning abstract semantics can be defined as follows:

Definition 4.4.6. (DYNAMIC PARTITIONING ANALYSIS) The *abstract semantics* of P_T between $\ell_\vdash$ and $\ell_\dashv$ is a function $[\![P_T]\!]^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]} : D^\sharp_{\delta\mathbb{P},\mathbb{M}} \rightarrow D^\sharp_{\delta\mathbb{P},\mathbb{M}}$ such that, if $(T, P_T, d_T), (T', P_{T'}, d_{T'}) \in D^\sharp_{\delta\mathbb{P},\mathbb{M}}$ are such that $(T', P_{T'}, d'_{T'}) = [\![P_T]\!]^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]}(T, P_T, d_T)$, then, there exists $\tau : T' \rightarrow T$ satisfying the following conditions:

— $P_{T'}$ refines P_T, i.e. $P_T \preccurlyeq_\tau P_{T'}$;

— $d'_{T'}$ approximates the output of $P_{T'}$ at $\ell_\dashv$ when the input at $\ell_\vdash$ is described by d_T in the previous system in a sound manner, which is expressed by the following condition, where $d_{T'} = d_T \circ \tau$:

$$\left.\begin{array}{l} \forall(\iota,\rho),(\iota',\rho') \in \mathbb{T}\times\mathbb{M}, \\ (\iota',\rho') \in \alpha_{\iota\mathcal{F}\,\mathbb{P}[\ell_\vdash,\ell_\dashv]}([\![P_{T'}]\!])(\iota,\rho) \\ \rho \in d_{T'}(\iota) \end{array}\right\} \Longrightarrow \rho' \in d'_{T'}(\iota')$$

Note that the soundness of the "abstract transfer function" in the second of point of Definition 4.4.6 is expressed in the refined system: the input invariant d_T is refined into $d_{T'}$ first, and then the abstract transition is performed in T'.

This abstract semantics is sound as well:

THEOREM 4.4.7. (SOUNDNESS OF THE DYNAMIC PARTITIONING ANALYSIS) *Let* $(T, P_T) \in \mathfrak{B}$ *such that* $(T_\epsilon, P_\epsilon) \preccurlyeq_\tau (T, P_T)$. *Let* $d_t \in D^\sharp_{\mathbb{P},\mathbb{M}}$, $(\iota,\rho) \in \mathbb{T}\times\mathbb{M}$ *such that* $\rho \in d_t(\iota)$. *We write* $(T', P_{T'}, d'_{T'})$ *for the result of the analysis* $[\![P_T]\!]^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]}(T, P_T, d_T)$. *Moreover, we let* $\rho' \in \alpha_{\iota\mathcal{F}[\ell_\vdash,\ell_\dashv]}([\![P]\!])(\rho)$. *Then, there exists* ι' *such that*

$$\rho' \in d'_{T'}(\iota')$$

PROOF. Similar to the proof of 4.4.3. □

Again, the core of the soundness of the analysis lies in the definition of the abstract transformer in the refined transition system, which should soundly approximate the partitioning of the transitions of the original system.

Example 4.4.8. (DENOTATIONAL STYLE ABSTRACTION OF A **if**-STATEMENT) Example 4.4.4 demonstrates the analysis of a conditional statement, based on a static partitioning of P_0 into P_1.

In the case of dynamic partitioning, the main difference is that, before the analysis of the conditional, the system under consideration is P_0 and that the analysis refines P_0 into P_1 at point ℓ_1 (beginning of the conditional). After this refinement, the book-keeping of the partitions is the same as in Example 4.4.4.

5. STATIC ANALYSIS WITH A TRACE PARTITIONING DOMAIN

We now introduce the trace partitioning domain integrated in the ASTRÉE analyzer, together with some examples showing how it contributes to improving precision.

Basically, this domain is an instantiation of the framework for defining trace partitioning domains, which we set up in Section 3 and in Section 4. This domain is tailored in order to cope with imprecisions observed when analyzing real programs.

5.1 Partitioning Criteria

First, we list the criteria for trace partitioning in ASTRÉE:

(1) **Partitioning of conditional structures,** by delaying the merge of flows in the end of the conditional;

(2) **Partitioning of loop structures,** by distinguishing the first iterations in the analysis of the loop body and delaying the merge of flows after the end of the loop. This criteria allow for:
— more precise invariants to be derived in the first iterations, thanks to unrolling;
— relations between numbers of iterations and values to be inferred and used after the loop, thanks to the delayed abstract join;

(3) **Partitioning guided by the value of a variable** x at some point ℓ (the partitions are computed at point ℓ and not modified by an assignment to x): this partitioning is similar to a case analysis based on the value of a variable (this partitioning scheme is most useful when dealing with weak updates, and array accesses);

(4) **Inlining of functions** (as suggested in Section 3.1);

(5) **Merge of partitions:** the cost of successive creations of partitions would be prohibitive in practice. For instance, the partitioning of a conditional structure multiplies by 2 the number of partitions in the current flow, so a series of n conditional structures would lead to a 2^n blow-up, which is not acceptable (no scalable analysis can afford an exponential cost). Therefore, we avail ourselves the possibility of merging together unnecessary partitions (i.e. partitions which are not expected to lead to further improvements in precision), in any order.

Some of these cases could be handled by rewriting the code. This approach is depicted in Figure 4(a), in the case of the partitioning of a conditional structure (case 1), as suggested in Example 3.2.3: the statements following the conditional are duplicated in the end of both branches. Case 2 (loops) and case 4 (function inlining) could be handled in a similar manner. For instance, Figure 4(b) displays the rewriting equivalent to the unrolling of the first iteration of a loop.

However, we show in Section 5.3 that the design of a trace partitioning domain was preferable, so that finer partitions can be handled.

```
$\ell_0$ : if($e$){                 ($\ell_0$) : if($e$){
$\ell_1$ :     $s_1$                ($\ell_1$) :     $s_1$;
      }else{                        ($\ell_3$) :     $s_3$
$\ell_2$ :     $s_2$      ⟶                      }else{
      }                             ($\ell_2$) :     $s_2$;
$\ell_3$ : $s_3$                    ($\ell_3$) :     $s_3$
$\ell_4$ : ...                                   }
                                    ($\ell_4$) : ...
```

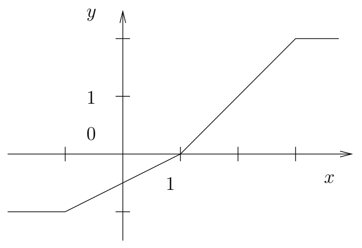
(a) Partitioning of a conditional

```
                                    ($\ell_0$) if($e$){
                                    ($\ell_1$)     $s$;
$\ell_0$ : while($e$){              ($\ell_0$)     while($e$){
$\ell_1$ :     $s$;        ⟶        ($\ell_1$)         $s$;
$\ell_2$ : }                        ($\ell_2$)     }
$\ell_3$ : ...                      ($\ell_2$) }
                                    ($\ell_3$) ...
```

(b) Loop unrolling

Control states in parentheses denote partitioned control states.

Fig. 4: Code rewriting



$$y = \begin{cases} -1 & \text{if } x \leq -1 \\ -0.5 + 0.5 \times x & \text{if } -1 \leq x \leq 1 \\ -1 + x & \text{if } 1 \leq x \leq 3 \\ 2 & \text{if } 3 \leq x \end{cases}$$

(a) Function

```
$\ell_0$ : int $i = 0$;
$\ell_1$ : while($i < n$ && $x > tx[i+1]$)      $tc = \{0; 0.5; 1; 0\}$
$\ell_2$ :         $i++$;                       $tx = \{0; -1; 1; 3\}$
$\ell_3$ : $y = tc[i] \times (x - tx[i]) + ty[i]$      $ty = \{-1; -0.5; -1; 2\}$
$\ell_4$ : ...
```

(b) Implementation

Fig. 5: Linear interpolation, via indirection arrays

5.2 Applications of Trace Partitioning to the Computation of More Precise Invariants

Before we set up the partitioning domain, we provide a few examples, so as to show how the main criteria for partitioning introduced in Section 5.1 are useful, in Astrée.

5.2.1 *Linear Interpolation Function, via Indirection Arrays.* We consider the case of the interpolation function $\mathrm{f_{lin}}$ described in Figure 5 first.

The piece of code for this function determines what formula should be used by localizing in what range x can be found, using a loop and an array of input values. Then, two arrays contain the coefficients which should be used in order to compute the value of $\mathrm{f_{lin}}(x)$. Clearly, the output of this function is bounded: $\forall x,\ \mathrm{f_{lin}}(x) \in [-1, 2]$.

However, inferring this most precise range is not feasible with a standard interval analysis, even if we partition the traces depending on the values of i at point ℓ_3. Let us try with $-100 \leq x \leq 0$: then, we get $i \in \{0, 1\}$ at point ℓ_3. The range for y at point ℓ_4 is $[-0.5 + 0.5 \times (-100.), -0.5] \equiv [-50.5, -0.5]$ (this range is obtained in the case $i = 1$; the case $i = 0$ yields $y = -1$). Accumulating such huge imprecision during the analysis may cause the properties of interest (e.g. the absence of runtime errors or the range of output values) not to be proved. We clearly see that some relations between the value of x and the value of i are required here.

Our approach is to partition the traces according to the number of iterations in the loop. Indeed, if the loop is not iterated, then $i = 0$ at point ℓ_3 and $x < -1$; if it is iterated exactly once, then $i = 1$ at point ℓ_3 and $-1 \leq x \leq 1$ and so forth. This approach yields the most precise range. Let us resume the analysis, with the initial constraint $-100 \leq x \leq 0$. The loop is iterated at most once and the partitions at point ℓ_3 give:

— 0 iteration: $i = 0$; $x < -1$; $y = -1$
— 1 iteration: $i = 1$; $-1 \leq x \leq 0$; $-1 \leq y \leq -0.5$.

Therefore, the resulting range is $y \in [-1, -0.5]$, which is the optimal range (i.e. exactly the range of all output values that can be observed in concrete executions).

This optimal result is obtained thanks to a partitioning of the traces by the number of iterations in the loop. The partitions can be merged after the output of the function, since they should not result in any further gain in precision.

5.2.2 *Linear Interpolation Function, via Discretization.* The second example consists in another kind of interpolation function: the input value is disctretized, and then a formula depending on the discretized value is applied to it. More precisely, if $\lfloor x \rfloor = n$, and f is the function to approximate, then the interpolation $\mathrm{f}_{\mathrm{lin}}$ returns $\mathrm{f}(n) + (x - n) \times (\mathrm{f}(n+1) - \mathrm{f}(n))$. From the mathematical point of view, it is a particular case of the interpolation function considered in the previous paragraph, where the values of the array tx are successive integer values. In the example presented in Figure 6, the array ty is such that $ty[n] = \mathrm{f}(n)$. Any interpolation based on a regular partition of a bounded range could be implemented in a similar way, by applying a linear function to the argument so as to recover a partition of the form $0, 1, \ldots, n$.

We found that this kind of interpolations were rather common, e.g. for approximating trigonometric functions. For the same reason as in the case of the previous interpolation function, the computation of a precise range for the output of $\mathrm{f}_{\mathrm{lin}}$ requires some precise relation between n and x.

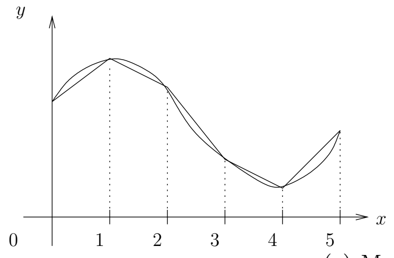
However, the possible values for n cannot be related to distinct control flow paths; therefore, we propose to perform a partitioning guided by the *value of* n *computed at* ℓ_1. Doing the same partitioning at point ℓ_2 would not allow for relations between x and i to be obtained.

5.3 The Domain

We now introduce the trace partitioning domain used in ASTRÉE formally.

5.3.1 *Need for a Trace Partitioning Domain.* As we pointed out in Section 5.1, some of the partitioning configurations could have been carried out by rewriting the code. However, we enumerate a number of reasons in favor of the design of a real domain.

First, the **“syntactic transformation” approach is limiting**. In particular, it would not allow to represent and handle large sets of partitions in the same way as a dedicate domain would:



if $n \leq x < n+1$, where $n \in \mathbb{Z}$
$y \ = \ \mathrm{f}(n) + (x-n) \times (\mathrm{f}(n+1) - \mathrm{f}(n))$

(a) Mathematical definition

```
ℓ0 : int n = 0;
ℓ1 : n = cast_{float→int} x;
ℓ2 : y = ty[n] + (x − cast_{int→float} n)∗
         (ty[n + 1] − ty[n])
ℓ3 : ...
```

(b) Implementation

Fig. 6: Linear interpolation function, via discretization

— a domain allows to represent *more* partitions than mere syntactic rewriting, since not all possible partitions need to be generated during the analysis despite the syntactic approach would require to generate them all prior to the analysis;

— a syntactic rewriting of the code would be inherently *static*, which is not practically compatible with very large sets of partitions. For instance, a partitioning guided by the values of a variable may generate a huge number of partitions if the variable may take a large number of values (e.g., thousands of values); in this case, a built-in strategy would not perform the partitioning (by not sending the partitioning order to the domain), whereas the decision whether to partition or not would need to be made prior to the analysis in the case of syntactic partitioning. In this case, the implementation of a partitioning domain allows to tune the partitioning strategy during the analysis, so that better decisions can be taken about whether or not some partitions should be generated.

Secondly, as we pointed out above, **the partitions** sometimes **need to be merged** together. Currently, where and which partitions are merged is the result of some strategies (Section 6.2). However, the last partition created may not be merged first, which implies that the structure of partitions should be found in abstract elements (as a consequence, the code rewriting approach would fail to offer the same flexibility).

Thirdly, in some cases, partitions could be created **in a lazy way only** not only for cost reasons, as in the following cases:

— in the case of a function call, where the function is the result of the dereference of a pointer, the control flow can only be known at analysis time;

— some strategies may determine that a loop should be unrolled n times and the analysis may prove that after $m < n$ iterations the execution of the loop terminates; then a syntactic unrolling would not make sense.

Last, the **inspection of analysis results** is easier, when the invariants can be related to the original program, with accurate partition names (i.e., tokens in the

$$\begin{array}{rll} d ::= & \mathfrak{part}\langle \mathbf{If}, \ell, b\rangle & \text{traces in the } b \text{ branch of the conditional at point } \ell \\ \mid & \mathfrak{part}\langle \mathbf{While}, \ell, n\rangle & \text{traces with exactly } n \text{ iterations in the loop at point } \ell \\ \mid & \mathfrak{part}\langle \mathbf{While}, \ell, > n\rangle & \text{traces with more than } n \text{ iterations in the loop at point } \ell \\ \mid & \mathfrak{part}\langle \mathbf{Val}, \ell, x = n\rangle & \text{traces such that } x = n \text{ at point } \ell \\ \mid & \mathfrak{part}\langle \mathbf{Fun}, \ell, f\rangle & \text{traces calling } f \text{ at point } \ell \\ \mid & \mathfrak{part}\langle \mathbf{None}\rangle & \text{void directive} \end{array}$$

(a) Directives (notation for directives: $d \in \mathcal{D}$)

$$\begin{array}{rll} \iota ::= & \epsilon & \text{empty stack, initial partition} \\ \mid & d :: \iota' & \text{addition of a directive on top of } \iota' \end{array}$$

(b) Tokens ($\iota \in \mathbb{T}$)

Fig. 7: Naming partitions

scheme of Section 3). Rewriting large pieces of code as suggested in Figure 4 would make the understanding of the result of static analyses more difficult, since the user would have to relate the invariants computed for the transformed program to the original program. By contrast, the values of the partitioning domain should tell what partitions numerical constraints correspond to, thanks to the partitioning tokens.

5.3.2 *Elements.* We now define formally the instantiation of the framework presented in Section 4 corresponding to the criteria listed in Section 5.1.

Intuitively, the creation of a partition corresponds to a partitioning directive, as defined in Section 5.1. We provide the formal definition of directives in Figure 7(a). The name of each directive corresponds very intuitively to a criterion listed in Section 5.1, except for the last one: the directive $\mathfrak{part}\langle \mathbf{None}\rangle$ is included here for the sake of implementation only, and stands for a void directive (we explain the use of this directive in Section 6.1).

The name of a partition (i.e., token corresponding to it, in the sense of Section 3.2) consists in the series of the partitioning directives encountered before creating this partition. We give the formal definition for tokens in Figure 7(b). We note that each partitioning directive encloses a control state, which stands for the point the partition was created at. The directive $\mathfrak{part}\langle \mathbf{None}\rangle$ stands for a void directive, and as such, it can be removed from tokens without changing their meaning: in other words, the equality on tokens is defined modulo removal of void directives (i.e., $\mathfrak{part}\langle \mathbf{None}\rangle :: \mathfrak{part}\langle \mathbf{If}, \ell, b\rangle = \mathfrak{part}\langle \mathbf{If}, \ell, b\rangle$).

For instance, in the case of a conditional at point ℓ, two partitions are created right after the testing of the condition, corresponding to the directives "true branch of the conditional at point ℓ" and "false branch of the conditional at point ℓ". When these partitions are merged, these directives are removed from the names of the partitions.

As usual, we write $D^\sharp_{\mathbb{M}}$ for the domain for representing sets of stores (Section 2.2). In the same way as in Section 4.4, the domain $D^\sharp_{\mathbb{P},\mathbb{M}}$ is defined as $\mathbb{T} \rightarrow D^\sharp_{\mathbb{M}}$.

5.3.3 *Hints (or Directives) in the Code.* A pre-processing phase inserts directives as special commands in the source code. We do not introduce them formally here (the directives are represented as text between braces in programs). Intuitively, directives in the code cause directives to be added in tokens (partition creation) or be deleted from tokens (partition merge).

5.3.4 *Widening.* The set of tokens is clearly infinite, since the length of tokens as sequences of directives is not bounded. Even in case we limit the length of tokens the number of tokens is very large: indeed, if we fix $\ell \in \mathbb{L}$ and $x \in \mathbb{X}$, the number of directives of the form $\mathfrak{part}\langle \mathbf{Val}, \ell, x = n\rangle$ is equal to the number of integer values in the language (i.e., in practice 2^{32}). Therefore, the termination of the analysis should rely on a widening operator, designed as in Section 4.3.

In practice,

— the widening operator on the basis forbids the synthesis of arbitrary long tokens, by preventing the generation of tokens containing two directives corresponding to the same control point: basically, this operator interrupts the generation of partitions;

— the generation of partitions after a directive recommending the partitioning guided by the values of a variable x is performed only if the size of the set of possible values for x determined by the analysis is small enough (e.g., below 1000);

— the current partitioning strategy is designed so as not to keep partitions beyond the scope they should improve the precision in; this strategy allows to merge partitions soon enough, so that the widening operator does not need to collapse partitions down (widening is applied at loop heads only [Bourdoncle 1993]).

5.4 Structure of the Abstract Interpreter

As stated in Section 2.3, the iterator consists in a function mapping statements into abstractions of their denotational semantics, as defined in Section 2.3. As a consequence, the design of the abstract interpreter follows the principle described in Section 4.4: the abstract interpretation $[\![s]\!]^\sharp$ of a statement s should map a pair $(P_T, d_T) \in \mathfrak{B} \times D^\sharp_{\mathbb{P},\mathbb{M}}$, where $\forall \iota \notin T,\ d_T(\iota) = \bot$ into a pair $(P_{T'}, d'_{T'}) \in \mathfrak{B} \times D^\sharp_{\mathbb{P},\mathbb{M}}$, where $P_{T'}$ is a refinement of P_T and $d'_{T'}$ is an over-approximation of the output of s when applied to the input d_T (Definition 4.4.6).

The iterator of ASTRÉE does not keep track of the whole refined program P_T. Instead, it keeps track of the *current partitions*, i.e. of the tokens corresponding to a set of partitions covering the ongoing flows:

Definition 5.4.1. (ONGOING TOKEN SET) The *ongoing token set* corresponding to the abstract flow $d_T \in D^\sharp_{\mathbb{P},\mathbb{M}}$ is $\mathbf{tokens}_T\langle d_T\rangle = \{\iota \in \mathbb{T} \mid d_T \neq \bot\}$.

This notion was implicitly illustrated in Example 4.4.4 (we described the partitioning abstract interpretation of an **if**-statement).

If (T, P_T, d_T) is the result of the static analysis of a statement, then, the property $\mathbf{tokens}_T\langle d_T\rangle \subseteq T$ is straightforward.

The abstract interpretation $[\![s]\!]^\sharp$ of a statement s simply maps an element $d_T \in D^\sharp_{\mathbb{P},\mathbb{M}}$ into a second element $d'_{T'} \in D^\sharp_{\mathbb{P},\mathbb{M}}$: all the information about the partitioning carried out by the analysis are enclosed in the d_T element.

This is a common advantage of denotational style abstract interpreters: this iteration scheme keeps only the information which are useful for the end of the analysis and discards the values which were useful only in the past and will not be required anymore. For instance, we remarked that the analyzer presented in

Section 2.3 does not need to store invariants at every control point. The restriction to the set of tokens corresponding to the ongoing flows is similar.

This approach is feasible, since the partitioning tokens contain all the information about the transitions associated to them.

Last, we note that the pre-processing phase inserts hints in the code and selects this way a family of extended systems which may be used during the analysis. As a consequence, most of the partitioning decisions are made statically; the only decisions taken at analysis time are whether or not to obey to some directives. In this sense, the partitioning implemented in ASTRÉE is dynamic, but mostly determined statically; reducing the number of choices made at analysis time simplifies the implementation.

5.5 Transfer Functions

We consider three kinds of transfer functions:

— the "partition creation" transfer function generate new partitions;

— the "partition merge" folds partitions together;

— the "standard" transfer functions (i.e., which are not specific to partitioning analyses) stand for e.g., abstract assignments, condition testing...

5.5.1 *"Usual" transfer function, e.g. assignment.* we extend pointwisely the usual transfer functions presented in Section 2.2 to $D^\sharp_{\mathbb{P},\mathbb{M}}$.

5.5.2 *Partition creation.* we let $\mathit{generate} : \mathcal{D} \times D^\sharp_{\mathbb{P},\mathbb{M}} \rightarrow D^\sharp_{\mathbb{P},\mathbb{M}}$ be the partition creation abstract transfer function. It inputs a directive ∂ and an abstract element $d \in D^\sharp_{\mathbb{P},\mathbb{M}}$ and adds the directive ∂ to all ongoing tokens in d. Formally, it outputs an element d', defined by:

$$\begin{cases} \mathbf{tokens}_T\langle d'\rangle = \{(\partial :: t) \mid t \in \mathbf{tokens}_T\langle d\rangle\} \\ \forall t \in \mathbf{tokens}_T\langle d\rangle,\ d'(\partial :: t) = d(t) \end{cases}$$

5.5.3 *Partition merge.* we let $\mathit{merge} : \mathcal{P}(\mathcal{D}) \times D^\sharp_{\mathbb{P},\mathbb{M}} \rightarrow D^\sharp_{\mathbb{P},\mathbb{M}}$ be the transfer function for merging partitions. It folds partitions by removing any directive in $\mathcal{D}$ for the partition names (tokens). Therefore merge inputs a set of directives D and an abstract element d and returns a new abstract element d', where any reference to the directives in D are removed. Formally, if $D = \{\partial\}$, then d' is defined by:

$$\begin{cases} (\partial_{i_0} :: \ldots :: \partial_{i_m}) \in \mathbf{tokens}_T\langle d'\rangle \iff \begin{cases} (\partial_0 :: \ldots :: \partial_n) \in \mathbf{tokens}_T\langle d\rangle \\ \{i_k \mid k \in [\![0, m]\!]\} = \{i \in [\![0, n]\!] \mid \partial_i \neq \partial\} \\ i_0 < \ldots < i_m \end{cases} \\ \text{With the above notations, } d'(\partial_{i_0} :: \ldots :: \partial_{i_m}) = d(\partial_0 :: \ldots :: \partial_n) \end{cases}$$

The above definition extends straightforwardly to the general case (D not necessarily a singleton).

Example 5.5.1. (TRANSFER FUNCTIONS IN A PARTITIONING ANALYSIS) Figure 8 displays a simple piece of code, containing an **if**-statement (Figure 8(a)). The pre-processing phase of ASTRÉE includes some directives in the code, which specify what partitions should be created. We assume that the strategies recommend to partition the traces in the beginning of the **if**-statement and to merge the partitions at point ℓ_5, as shown in Figure 8(b)).

```
$\ell_0$ : $s_0$;
$\ell_1$ : if($c$){
$\ell_2$ :       $s_1$
      }else{
$\ell_3$ :       $s_2$
      }
$\ell_4$ : $s_3$;
$\ell_5$ : $s_4$;
```

(a) Initial program

```
$\ell_0$ : $s_0$;
      ≀Partition the traces
         in the following if statement∫
$\ell_1$ : if($c$){
$\ell_2$ :       $s_1$
      }else{
$\ell_3$ :       $s_2$
      }
$\ell_4$ : $s_3$;
      ≀Merge the partitions of
         the if statement at this point∫
$\ell_5$ : $s_4$;
```

(b) Program with directives added

Fig. 8: Partitioning analysis of a **if**-statement: directives

Here are the main steps of the analysis:

— at point ℓ_0, only one partition exist; it corresponds to the void token ϵ;

— when entering the **if**-statement, the analyzer creates two partitions corresponding to the directives $\mathfrak{part}\langle\mathbf{If}, \ell_0, \mathbf{true}\rangle$ (true branch) and $\mathfrak{part}\langle\mathbf{If}, \ell_0, \mathbf{false}\rangle$ (false branch): at this step it applies the transfer function which associates the abstract element $\mathit{generate}(\mathfrak{part}\langle\mathbf{If}, \ell_0, \mathbf{true}\rangle, d)$ to d and $d \mapsto \mathit{generate}(\mathfrak{part}\langle\mathbf{If}, \ell_0, \mathbf{false}\rangle, d)$;

— the analysis of the body of both branches involve usual transfer functions;

— at point ℓ_4 the join of the invariants corresponding to both branches should be computed, so that we get an invariant d_4, such that $\mathbf{tokens}_T\langle d_4\rangle = \{\mathfrak{part}\langle\mathbf{If}, \ell_0, \mathbf{true}\rangle :: \epsilon, \mathfrak{part}\langle\mathbf{If}, \ell_0, \mathbf{false}\rangle :: \epsilon\}$;

— at point ℓ_5 the analyzer merges the partitions together, by applying the transfer functions $d \mapsto \mathit{merge}(\{\mathfrak{part}\langle\mathbf{If}, \ell_0, \mathbf{true}\rangle, \mathfrak{part}\langle\mathbf{If}, \ell_0, \mathbf{false}\rangle\}, d)$.

6. TRACE PARTITIONING IN THE ASTRÉE ANALYZER

In this section, we deal with practical issues related to the implementation of the trace partitioning domain presented in Section 5 in Astrée. We describe the data structures and the partitioning strategies. Last, we provide extensive implementation results.

6.1 Implementation of the Domain

6.1.1 *The Data-Structure.* In practice, $\mathbf{tokens}_T\langle d_T\rangle$ can be considered the set of paths into the leaves of a tree, where each branch in the tree is labeled with a directive. Therefore, trees are a natural representation for the elements of $D^\sharp_{\mathbb{P},\mathbb{M}}$, with elements of $D^\sharp_{\mathbb{M}}$ at the leaves and with directives as labels for the branches:

Definition 6.1.1. (Representation of the elements of $D^\sharp_{\mathbb{P},\mathbb{M}}$) The physical representation of the elements of $D^\sharp_{\mathbb{P},\mathbb{M}}$ is defined by induction by:

$$
\begin{array}{rcll}
d_T & ::= & \mathrm{leaf}[d] \text{ where } d \in D^\sharp_{\mathbb{M}} & (\text{leaf } D^\sharp_{\mathbb{M}} \text{ element}) \\
 & | & \mathrm{node}[\phi] \text{ where } \phi \in \mathcal{D} \to D^\sharp_{\mathbb{P},\mathbb{M}} & (\text{function mapping directives into } D^\sharp_{\mathbb{P},\mathbb{M}})
\end{array}
$$

The use of this representation is exemplified in Example 6.1.3, after we define the transfer functions.

Remark 6.1.2. (Use of the $\mathfrak{part}\langle\mathbf{None}\rangle$ directive) In some cases, we may have to represent an invariant d_T, such that $t \in \mathbf{tokens}_T\langle d_T\rangle$ and $(\partial :: t) \in \mathbf{tokens}_T\langle d_T\rangle$ (for some token t and some directive ∂). Then, the above definition does not provide a way to represent the invariant corresponding to t since t is a prefix of $\partial :: t$ and Definition 6.1.1 does not allow for numerical invariants to be assigned to nodes of the trees (numerical invariants correspond to leaves only).

The $\mathfrak{part}\langle\mathbf{None}\rangle$ directive solves this problem: indeed, $\mathfrak{part}\langle\mathbf{None}\rangle :: t$ is equivalent to t, and a numerical invariant can be assigned to the leaf corresponding to $\mathfrak{part}\langle\mathbf{None}\rangle :: t$.

Such configurations do not occur in the analysis; they may arise in the invariant export (Section 2.3), when all local invariants corresponding to a control state ℓ_0 (possibly in different contexts, e.g., for different function calls) should be represented together. In particular the abstract join operator may generate $\mathfrak{part}\langle\mathbf{None}\rangle$ directives.

6.1.2 *The Transfer Functions.* The implementation of the transfer functions proceeds by induction on the structure of the trees. Indeed, let us consider the three kinds of transfer functions, which we introduced in Section 5.5 (in the following, we augment the names of the transfer functions for the partitioning domain with the index $_{\mathbb{P}}$):

— **Abstract binary operators, e.g. join** are defined by induction on the structure of trees.
If the join of the set of paths in both trees contains two tokens t_0, t_1 such that t_0 is a strict prefix of t_1, then t_0 is replaced with $\mathfrak{part}\langle\mathbf{None}\rangle :: t_0$ so that the result can be represented, as explained in Remark 6.1.2.

— **"Usual" transfer functions:** we consider the case of the $\mathit{guard}_{\mathbb{P}} : \mathbb{e} \times \mathbb{B} \times D^{\sharp}_{\mathbb{P},\mathbb{M}} \rightarrow D^{\sharp}_{\mathbb{P},\mathbb{M}}$ transfer function, which inputs a condition $e \in \mathbb{e}$, a boolean $b \in \mathbb{B}$, and an abstract element d and outputs an over-approximation of the stores in d which evaluate e into b (in the case of assignments, variable forget... are similar). The definition of $\mathit{guard}_{\mathbb{P}}$ is based on the function guard defined over $D^{\sharp}_{\mathbb{M}}$:

$$\forall e \in \mathbb{e},\ \forall b \in \mathbb{B},\ \begin{cases} \mathit{guard}_{\mathbb{P}}(e, b, \mathrm{leaf}[d]) & = \mathrm{leaf}[\mathit{guard}(e, b, d)] \\ \mathit{guard}_{\mathbb{P}}(e, b, \mathrm{node}[\phi]) & = \mathrm{node}[\partial_p \mapsto \mathit{guard}(e, b, \phi(\partial_p))] \end{cases}$$

— **Partition creation:** the partition creation abstract transfer function $\mathit{generate} : \mathcal{D} \times D^{\sharp}_{\mathbb{P},\mathbb{M}} \rightarrow D^{\sharp}_{\mathbb{P},\mathbb{M}}$ inputs a partitioning directive ∂ and an abstract element d and pushes the token ∂ on top of the tokens. Basically, it mimics the creation of a partition triggered by the directive ∂, which amounts to adding a node on top of each leaf in d, with a branch indexed by ∂ in between:

$$\forall \partial \in \mathcal{D},\ \begin{cases} \mathit{generate}(\partial, \mathrm{leaf}[d]) & = \mathrm{node}[\partial \mapsto \mathrm{leaf}[d]] \\ \mathit{generate}(\partial, \mathrm{node}[\phi]) & = \mathrm{node}[\partial_p \mapsto \mathit{generate}(\partial, \phi(\partial_p))] \end{cases}$$

In practice, the partition generation function takes into account the names of the partitions, so as to create only *some* partitions.

— **Partition merge:** the transfer function $\mathit{merge} : \mathcal{P}(\mathcal{D}) \times D^{\sharp}_{\mathbb{P},\mathbb{M}} \rightarrow D^{\sharp}_{\mathbb{P},\mathbb{M}}$ inputs $D \subseteq \mathcal{D}, d \in D^{\sharp}_{\mathbb{P},\mathbb{M}}$; it goes recursively through the tree representing d and removes all occurrences of a directive in D. The implementation follows the following algorithm:

$$\forall D \in \mathcal{P}(\mathcal{D}), \begin{cases} \mathit{merge}(D, \mathrm{leaf}[d]) = \mathrm{leaf}[d] \\ \mathit{merge}(D, \mathrm{node}[\phi]) = \mathrm{node}[\phi'] \\ \quad \text{where } \phi' : \begin{cases} \partial \notin D & \mapsto \mathit{merge}(D, \phi(\partial)) \\ \mathfrak{part}\langle \mathbf{None} \rangle & \mapsto \bigsqcup\{d \text{ at a leaf of } \phi(\partial) \mid \partial \in D\} \end{cases} \end{cases}$$

The directive $\mathfrak{part}\langle \mathbf{None} \rangle$ allows to fold together *some* branches leaving from a node. In case all branches can be folded, then these directives can be safely removed from trees:

$$\mathrm{node}[\{\mathfrak{part}\langle \mathbf{None} \rangle \mapsto d_0\}] \rightarrow d_0$$

Example 6.1.3. (Application to the partitioning of an **if**-statement) We consider the program considered in Example 5.5.1, with the partitioning strategy displayed in Figure 8(b). We assume that the analysis starts with a single partition (i.e., only one ongoing token at point ℓ_0).

Figure 9 displays the partitions obtained when the analysis reaches each control state in this program:

— statement s_0 does not generate any new partition, so the layout of the abstract element for ℓ_1 (Figure 9(a)) is the same as for ℓ_0 (Figure 9(b));

— the conditional causes a partitioning of the traces at ℓ_1, so two trees are created after this point (yet, the partition corresponding to **false** is not created explicitly in the true branch, since it would be empty), which are depicted in Figure 9(c) and Figure 9(d);

— the abstract join outputs a new abstract element, with two partitions corresponding to both sides of the conditional at point ℓ_4 (Figure 9(e));

— the merge of partitions is performed after the analysis of s_3, so that the tree in ℓ_5 consists in only a leaf (Figure 9(f)) at in ℓ_0.

As a shortcut, we write ∂_{t} for $\mathfrak{part}\langle \mathbf{If}, \mathbf{false}, \ell_1 \rangle$ and ∂_{f} for $\mathfrak{part}\langle \mathbf{If}, \mathbf{true}, \ell_1 \rangle$, and d for any invariant in $D^{\sharp}_{\mathbb{M}}$. Dotted lines denote the partitions which are not generated, since the analysis proves them empty.

6.2 Strategies for Trace Partitioning

6.2.1 *Implementation of a Partitioning Strategy.* As mentioned in Section 2.3, a pre-processing phase generates hints for the abstract domains, including the partitioning domain. Such hints specify the cases where partitions might be helpful in order to compute tighter invariants. In the analysis phase, partitioning may or may not be performed at these points, depending on the choice of the interpreter. Indeed, in case the pre-processing phase recommends a partitioning guided by the values of a variable v and the analyzer infers too large a range for v (i.e., the number of generated partitions would be prohibitive), the analyzer will not perform the partitioning. Similarly, it will not create empty partitions: for instance, in the case of a conditional statements which should be partitioned, if the analysis proves the

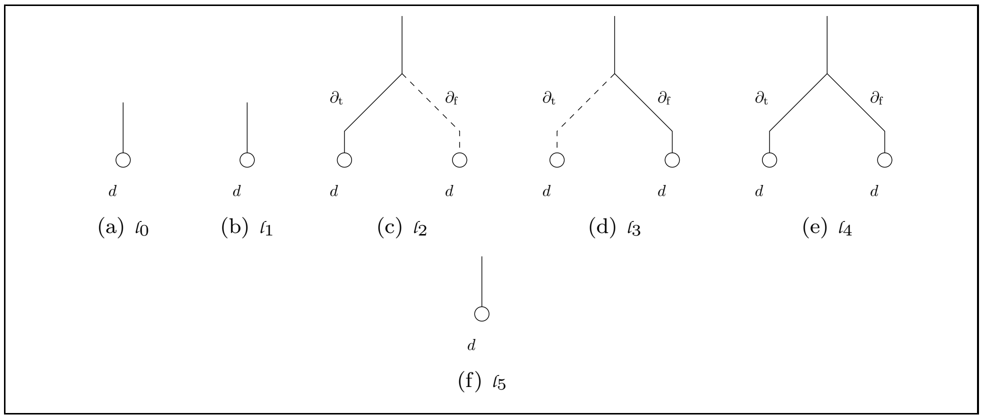


Fig. 9: Application to the partitioning of an **if**-statement

condition always evaluates to **true**, then, the partition corresponding to the **false** branch will not be generated.

6.2.2 *Strategies for Generating "Good" Partitions.* We enumerate a few cases where the current pre-processing phase suggests partitions to be generated:

— *sequences of conditional statements:* partitioning the traces in the first **if**-statement may greatly improve the precision in the following conditional statements, if the condition of the second **if**-statement depends on the content of the branches of the first one, or if its value depends on the value of the condition of the first **if**-statement.

— *assignment to an integer variable i* used *as an array index:* the partitioning guided by the value of i generates some relations with the variables in the right hand side of the assignment and may improve the precision of the subsequent array operation, since distinct array cells are treated separately, in a refined environment. This criterion causes the right partitions to be generated in the case of the interpolation function with regular discretization of the input, which we presented in Section 5.2, and Figure 6.

— *small loops assigning an integer variable i* used *e.g., as an array index:* the unrolling of the loop allows for the same kind of relations to be computed as in the previous point; hence, it results in the same opportunities for gains in precision. This criterion triggers the generation of the right partitions in the case of the interpolation function with indirection arrays, which we described in Section 5.2 and Figure 5.

6.3 Experimental Evaluation

6.3.1 *Methodology for the Benchmarks.* The results below were obtained on 2 GHz Bi-opteron machines, with 8 Gb of RAM (total) and 1 Mb of cache memory (per processor), running Linux. All the analyses reported in the sequel used only one processor, despite ASTRÉE also features the ability of being ran in "*parallel*" mode.

Table I. Benchmarks

Program	Size (LOCs)	Functions	Variables			
			Global and static		Local	
			int	float	int	float
$\mathcal{P}_1^1$	370	20	23	87	2	0
$\mathcal{P}_2^1$	9 500	236	35 100	835	4	8
$\mathcal{P}_3^1$	70 000	2 010	11 700	27 400	22	516
$\mathcal{P}_1^2$	70 000	1 150	71 400	8 670	11 700	5 700
$\mathcal{P}_2^2$	226 000	3 410	35 700	24 900	44 300	21 900
$\mathcal{P}_3^2$	400 000	5 680	58 700	35 500	83 400	35 100

Table II. Partitioning Strategy

Program	Size (LOCs)	Conditional		Loops		Value-based
		partitioned	total	partitioned	total	partitioning
$\mathcal{P}_1^1$	370	4	28	1	1	0
$\mathcal{P}_2^1$	9 500	18	283	1	3	0
$\mathcal{P}_3^1$	70 000	498	4 617	3	5	112
$\mathcal{P}_1^2$	70 000	300	2 624	106	106	0
$\mathcal{P}_2^2$	226 000	1 805	9 381	591	591	19
$\mathcal{P}_3^2$	400 000	2 802	17 562	906	916	32

The analyzer was ran on a series of programs, chosen among two families of embedded codes, which we detail in table I. Programs in family 1 (denoted with $\mathcal{P}_i^1$) are older, and of smaller size than programs in family 2 (denoted with $\mathcal{P}_i^2$).

6.3.2 *Partitioning Strategy.* Table II displays the results of the partitioning strategy. We give the total number of conditional structures, and the number of *partitioned* conditional structures. We provide similar information about the partitioning of loop structures; however, only the internal loops are taken into account here (we recall that a program in either families consists in a main loop, which contains most of the code). Last, we mention the number of directives recommending a partitioning guided by the values of a variable. Overall, partitioning directives are inserted in the case of 10 % to 20 % of the conditional structures and for almost all internal loops. The partitioning guided by the values of variables tend to have less importance (much fewer directives inserted, and only in the larger applications).

6.3.3 *Analysis with Partitioning Enabled.* In the following T.p.I. stands for "Time per iteration"; it corresponds to the average time spent in *one* iteration of the main loop of the program being analyzed. This time is roughly representative of the efficiency of the transfer functions and of the precision of the abstract control flow. The number of iterations assesses the efficiency of the convergence. The global time of the analysis depends both on the efficiency of transfer functions and the speed of the convergence.

Times are written in seconds (s); amounts of memory in megabytes (Mb).

The first benchmark displays the result of the analysis with the default settings: *trace partitioning is enabled* and the directives are inserted by the *automatic strategy*, evoked in Section 6.2.

Table III. Results of the Analysis with Trace Partitioning

	Size	Memory peak	Analysis time	Iterations		Alarms
	(LOCs)	(Mb)	(s)	$\sharp$	T.p.I. (s)	
$\mathcal{P}_1^1$	370	45	1.96	9	0.21	0
$\mathcal{P}_2^1$	9 500	175	104	17	6.1	8
$\mathcal{P}_3^1$	70 000	636	2 818	35	80.5	0
$\mathcal{P}_1^2$	70 000	434	1 064	20	53.2	0
$\mathcal{P}_2^2$	226 000	1 533	17 035	51	334	0
$\mathcal{P}_3^2$	400 000	2 423	36 480	72	507	0

Table IV. Analysis without Partitioning

	Size	Memory peak	Analysis time	Iterations		Alarms
	(LOCs)	(Mb)	(s)	$\sharp$	T.p.I.	
$\mathcal{P}_1^1$	370	45 (-)	1.55 (-21 %)	9	0.17s	0 (0)
$\mathcal{P}_2^1$	9 500	170 (- 3 %)	87 (- 17 %)	17	5.1s	8 (8)
$\mathcal{P}_3^1$	70 000	660 (+ 3 %)	1 614 (- 43 %)	35	46.1s	750 (0)
$\mathcal{P}_1^2$	70 000	376 (-13 %)	921 (- 13 %)	20	46s	443 (0)
$\mathcal{P}_2^2$	226 000	1 341 (- 12 %)	37 274 (+ 112 %)	282	134s	5 402 (0)
$\mathcal{P}_3^2$	400 000	2 040 (- 16 %)	34 147 (- 6 %)	127	269s	7 524 (0)

6.3.4 *Global Impact of Partitioning.* First, we compare the results of the analyses with or without trace partitioning enabled: table IV displays the results *without* trace partitioning. Note that the partitioning inherent in the function calls (function inlining) is not affected by the disabling of trace partitioning: turning off partitioning removes the partitioning relative to loop iterations, conditional and variables values only.

The number in parentheses allow to compare with the default, partitioning analyses. This first comparison shows the great impact of partitioning in most cases, and especially in the case of the large applications, i.e., the programs which compare most closely with real applications due to their size and structure. The first two programs are experimental programs, which do not comprise all the features of the largest applications and involve smaller chains of computations, so the trace partitioning does not impact the number of alarms. Yet, the invariants are noticeably less precise, even in the case of the first example. The analyses of larger, real-world applications generate dramatic number of alarms: trace partitioning proves a crucial technique in ASTRÉE.

Secondly, we remark that the execution time is not necessarily better when trace partitioning is disabled. In particular, the analysis of the two largest programs require a *much larger* number of iterations when trace partitioning is turned off: this effect was most noticeable in the case of the second program in the second family (282 iterations instead of 52!). In fact, a lower precision *may* result in a longer analysis time for many reasons related to the exploration of a larger state space:

— the widening of the analyzer attempts to stabilize variables, with a widening threshold scale [Blanchet et al. 2003]; therefore, if some variable cannot be stabilized to a small range (for instance, because some property cannot be proved due to the

Table V. Impact of the Partitioning of Conditional Structures

	Size	Memory peak	Analysis time	Iterations		Alarms
	(LOCs)	(Mb)	(s)	$\sharp$	T.p.I.	
$\mathcal{P}_1^1$	370	45 (-)	1.96 (-)	9	0.17s	0 (0)
$\mathcal{P}_2^1$	9 500	173 (- 1 %)	88 (- 15 %)	17	5.2s	8 (8)
$\mathcal{P}_3^1$	70 000	616 (- 3 %)	5 004 (+ 76 %)	32	156s	398 (0)
$\mathcal{P}_1^2$	70 000	467 (+ 8 %)	1 466 (+ 38 %)	20	73.2s	389 (0)
$\mathcal{P}_2^2$	226 000	1 680 (+ 10 %)	199 500 (+ 1 071 %)	290	688s	5 190 (0)
$\mathcal{P}_3^2$	400 000	2 735 (+ 12 %)	187 773 (+ 415 %)	125	1 502s	5 542 (0)

trace partitioning being turned off), it goes through a longer sequence of widened ranges (the analyzer attempts to find a larger, stable range), before it eventually reaches the "top" value (i.e., range containing all concrete values). This is an explanation for larger numbers of iterations in the case of less precise analyses.

— the control flow of the static analysis need to be more exhaustive when the precision is worse: for instance, in the case of a conditional, a less precise input invariant may require the analysis of *both* branches of the conditional whereas a more precise invariant may require analyzing only one branch, hence, require less time to complete.

Overall, we remark that the time per iteration is lower in the case of non-partitioning analyses and the partitioning analyses tend to require a lower number of iterations However, it is difficult to say for sure what is the most important factor: we may guess that only the first factor plays a significant role here (longer analyses due to longer widening chains), however, we should remark that the non-partitioning transfer functions handle much simpler data-structures; the latter factor may explain the shorter iterations.

Moreover, it is rather intuitive that one iteration of a partitioning analysis should take longer than one iteration of a non-partitioning analysis; however, the cost in time of trace partitioning (whether global analysis time or time per iteration) never turns out prohibitive.

Last, we remark that partitioning analyses require more memory in most cases; this result is to be expected, since partitioning analyses generate more data-structures and handle more numerical invariants. Yet, this cost is rather reasonable, since it never goes above 20 % (10 % average). This is mostly due to the fact that most partitioning criteria are *local*: they do not yield to huge sets of global partitions, thanks to the insertion of merge directives (Section 5.1).

In the following, we focus on several kinds of partitioning criteria and measure their impact on the results of the analysis.

6.3.5 *Impact of the Partitioning of Conditional Structures.* Second, we compare the default, partitioning analysis with analyses carried out without *some* partitions. Table V reports the result of the analysis without partitioning of conditional structures. The results in precision fall between the results of the partitioning analysis and the results of the non-partitioning analysis. In the case of the largest applications, the number of alarms is still dramatic.

In the resource usage point of view, these results are much worse than those of the non-partitioning analysis and of the partitioning analysis. Not only the number

Table VI. Impact of Inner Loops Partitioning

	Size	Memory peak	Analysis time	Iterations		Alarms
	(LOCs)	(Mb)	(s)	$\sharp$	T.p.I.	
$\mathcal{P}_1^1$	370	45	1.96 (-)	9	0.21s	0 (0)
$\mathcal{P}_2^1$	9 500	173 (-1 %)	85 (-18 %)	17	5s	8 (8)
$\mathcal{P}_3^1$	70 000	596 (- 6 %)	3 928 (+ 39 %)	63	62.3s	529 (0)
$\mathcal{P}_1^2$	70 000	391 (- 10 %)	12 319 (+1 058 %)	292	42.2s	208 (0)
$\mathcal{P}_2^2$	226 000	1 400 (- 9 %)	14 277 (- 16 %)	75	190s	2 954 (0)
$\mathcal{P}_3^2$	400 000	2 204 (- 9 %)	41 932 (+ 15 %)	115	364s	4 017 (0)

Table VII. Impact of Value-Guided Partitioning

	Size	Memory peak	Analysis time	Iterations		Alarms
	(LOCs)	(Mb)	(s)	$\sharp$	T.p.I.	
$\mathcal{P}_1^1$	370	45 (-)	1.58 (- 27 %)	9	0.18s	0 (0)
$\mathcal{P}_2^1$	9 500	173 (-)	82 (- 20 %)	17	4.8s	8 (8)
$\mathcal{P}_3^1$	70 000	682 (+ 7 %)	2 236 (+ 26 %)	33	67.8s	563 (0)
$\mathcal{P}_1^2$	70 000	438 (+ 1 %)	1 335 (+ 25 %)	20	66.7s	4 (0)
$\mathcal{P}_2^2$	226 000	1 550 (+ 1 %)	16 589 (- 3 %)	66	251s	3 (0)
$\mathcal{P}_3^2$	400 000	2 434 (-)	26 165 (- 28 %)	64	409s	8 (0)

of iterations but also the time per iteration tend to be worse than those of the partitioning analysis (despite simpler structures being used). At this point, we can imagine that not only the disabling of the partitioning of **if**-statements caused the analyzer to go through longer widening chains but also that it resulted in a coarser approximation of control flow. Another possibility is that the imprecision due to the absence of partitioning after **if**-statement may cause more imprecise partitions based on other criteria (loops, values of variables) to be generated, resulting in worse performances.

6.3.6 *Inner Loops Partitioning.* Table VI reports the result of the analysis without partitioning of loops. Again, we remark that loop partitioning is crucial for the precision of the analyses in the case of large applications, since the analysis of the four larger applications generate hundreds or thousands of false alarms. The invariants generated for the other programs are also significantly less precise (even though, the imprecision does not cause a larger number of alarms).

In the analysis time point of view, the same comments as above apply: in general the number of iterations is bigger, the time per iteration is smaller. In some cases ($\mathcal{P}_2^2$), the analysis is faster; in other cases ($\mathcal{P}_3^1, \mathcal{P}_1^2, \mathcal{P}_3^2$) it is slower. We note that $\mathcal{P}_1^2$ requires a very large number of iterations.

6.3.7 *Impact of Value-Guided Partitioning.* The impact of partitioning guided by values is less significant than the impact of the previous partitioning criteria, except in the case of the program $\mathcal{P}_3^1$ (dramatic number of alarms).

We report no very important difference in execution time. Yet, we note that the more precise analysis of $\mathcal{P}_2^2$ requires *more* iterations.

Overall, it turns out extremely difficult to explain all variations in resources required by static analyses: no rule allows to predict the speed of an analysis; and, in practice, too many factors play a role, even though one may be able to tell in some cases what the most important ones are.

7. CONCLUSION

7.1 Contribution

We proposed a generic framework for defining trace partitioning domains, which allow the partitioning of traces to be based on the history of the control flow. In particular, we allow for static partitioning analyses (the partition is chosen before the abstract iteration is carried out) and dynamic partitioning analyses (the analysis may refine the choice of partitions) to be designed in this framework.

We described an instantiation of this framework in the context of the ASTRÉE project and presented the main data-structures and the strategies used in order to choose the partitions which should be created. In particular, the structure of the domain does not require any specific assumption to be made about the numerical domain; moreover, the iterator does not need much adaptation either.

Last, we provided experimental evidence of the role played by trace partitioning in the success of the analyzer, which can analyzer very large industrial applications, and prove their safety (or at least, produce a very low number of false alarms). In fact, we noticed that trace partitioning not only improved the precision (which was to be expected) but also significantly reduces the execution time (which was not so intuitive), mainly due to faster convergence to stable ranges.

7.2 Related Work

The partitioning of control systems was introduced early in the static analysis field, e.g., in [Cousot 1981].

Among the closest related works, we can cite the trace partitioning static analysis framework proposed in [Handjieva and Tzolovski 1998]; however, this framework does not allow for the *merge* of partitions. Therefore, it incurs an exponential cost (in the number of **if** and **while** statements). Moreover, it does not allow for the dynamic partitioning guided by the values of a variable.

Another approach to partitioning in static analysis can be found in [Jeannet et al. 1999; Jeannet 2003]. It is based on a partitioning defined by predicates about the memory state, such as the value of boolean variables. The main difference is that we focus on the history of control flow, instead of the values of some variables. We believe our approach is more adapted to our case, since the predicate partitioning should be based on might not be expressed with a single variable, whereas distinct partitions correspond intuitively to control flow paths (overall, the control flow properties we use in order to guide the partitioning are rather simple).

We can also find several occurrences of refinements of the control structure in the literature about data-flow analysis. For instance, [Sharir and Pnuelli 1981] studied the most common approaches to interprocedural analyses. A finer handling of paths in control flow graphs was proposed in [Holley and Rosen 1980]: it proceeds by integrating some information about the paths in the edges of the control flow graph, so as to allow for a finer approximation of the control flow to be computed. In particular, this technique was used in order to infer sets of *feasible paths*, so as to allow for more precise data-flow analyses. Similarly [Bodík et al. 1997] determines branch correlations so as to detect incompatible branchings and cut down the approximation set of feasible paths. Our approach not only performs intuitive abstractions of the paths, but also takes the path into account dynamically during the analysis.

The qualified flow analysis technique was extended with path profiles [Ball and Larus 1996] in [Ammons and Larus 1998]: profiling data should determine a set of *hot* paths (i.e., more frequently taken); then, these paths can be analyzed separately, with a higher precision (no path joins). Similarly, the express lane transformation [Melski and Reps 2003] aims at duplicating hot paths, so as to improve precision. However, this approach does not apply in our case. First, profiling very large applications with very large numbers of variables does not seem a realistic solution (at least in the time point of view). Secondly, this approach analyzes all "non-hot paths" together (i.e., with no partitioning), which would result in a low precision, with possibly many alarms. Indeed, the precision required in the analysis of a path for proving it safe is not related to how frequently it is used; therefore, our approach ignoring the frequency of paths is more adapted to program certification.

Recently, a large number of path sensitive analyses were proposed and implemented in various frameworks, such as [Ball and Rajamani 2001; Flanagan et al. 2002] and contributed to the verification of complex properties. However, path sensitivity is very costly in practice: we could not apply this technique to a single iteration of the main loop of either of the programs considered in Section 6.3. An interesting solution to the cost of path sensitivity (yet, not applicable in our case) proposed in [Das et al. 2002] relies on the encoding of the property of interest into an automaton (*finite state machine*): the transitions in the automaton can be used as criteria for partitioning the paths, and a heuristic is introduced so as to merge paths as well.

7.3 Exploring Other Partitioning Criteria

The definition of other kinds of partitioning criteria is a nice area for future work.

In particular, we are working on another instantiation of the framework described in Section 4, so as to partition traces according to an abstraction of the history of program executions defined by a collection of "events".

This approach should allow to discriminate traces which satisfy some conditions defined from the history of program executions (such as: condition P was satisfied at point ℓ_0 at the previous iteration in a loop and is violated at the current iteration) prove some functional properties of programs. We can relate this technique with the notion of synchronous product of the program to analyze with an adapted control structure: this method has been proposed and widely used for the verification of synchronous programs [Halbwachs et al. 1993].

We already applied this technique to the semantic slicing involved in the investigation of alarms raised by Astrée in [Rival 2005]. Though, we only considered rather simple abstractions of control flow history (based on automata), and we plan to consider more ambitious families of abstractions.

ACKNOWLEDGMENTS

We would like to thank the other members of the Astrée team, Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné and David Monniaux for their help, suggestions and comments during the early developments of this work. We also acknowledge Bruno Blanchet for his contribution to the ASTRÉE project.

REFERENCES

AMMONS, G. AND LARUS, J. R. 1998. Improving data-flow analysis with path profiles. In *Proc. of the Conference on Programming Languages, Design and Implementation (PLDI'98)*. ACM Press, New York, NY, Montréal (Canada), 72–84.

BALL, T. AND LARUS, J. R. 1996. Efficient path profiling. In *ACM International Symposium on Microarchitecture (MICRO 96)*. IEEE Computer Society, Washington DC, USA, 46–57.

BALL, T. AND RAJAMANI, S. K. 2001. Automatically validating temporal safety properties of interfaces. In *8th International SPIN Workshop*. Lecture Notes in Computer Science. Springer-Verlag, Toronto (Canada), 103–122.

BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2002. 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*, T. Mogensen, D. Schmidt, and I. Sudborough, Eds. Lecture Notes in Computer Science 2566. Springer-Verlag, Berlin, Germany, 85–108.

BLANCHET, B., COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2003. A Static Analyzer for Large Safety Critical Software. In *Conference on Programming Languages, Design and Implementation (PLDI'03)*. ACM Press, New York, San Diego (USA), 196–207.

BODÍK, R., GUPTA, R., AND SOFFA, M. L. 1997. Refining data flow information using infeasible paths. In *6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*. Springer-Verlag, Zurich (Switzerland), 361–377.

BOURDONCLE, F. 1993. Efficient chaotic iteration strategies with widenings. *Lecture Notes in Computer Science 735*, 128–142.

BRYANT, R. 1986. Graph based algorithms for boolean function manipulation. *IEEE Trans. Comput. C-35*, 677–691.

COUSOT, P. 1981. Semantic foundations of program analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, Chapter 10, 303–342.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Symposium on Principles of Programming Languages (POPL'77)*. ACM Press, New York, NY, Los Angeles, California, 238–252.

COUSOT, P. AND COUSOT, R. 1979. Systematic design of program analysis frameworks. In *Conference Record of the 6th Symposium on Principles of Programming Languages (POPL'79)*. ACM Press, New York, NY, San Antonio, Texas, 269–282.

COUSOT, P. AND COUSOT, R. 1992a. Abstract interpretation and application to logic programs. *Journal of Logic Programming 13,* 2–3, 103–179.

COUSOT, P. AND COUSOT, R. 1992b. Abstract interpretation frameworks. *Journal of Logic and Computation 2,* 4 (Aug.), 511–547.

COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2005. The ASTRÉE analyzer. In *European Symposium On Programming (ESOP'05)*. Lecture Notes in Computer Science, vol. 3444. Springer-Verlag, Edimburgh (Scotland).

COUSOT, P. AND HALBWACHS, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th Symposium on Principles of Programming Languages (POPL'78)*. ACM Press, New York, NY, Tucson, Arizona, 84–97.

DAS, M., LERNER, S., AND SEIGLE, M. 2002. Esp: Path-sensitive program verification in polynomial time. In *Proc. of the Conference on Programming Languages, Design and Implementation (PLDI'02)*. ACM Press, New York, NY, Berlin (Germany), 57–68.

FERET, J. 2004. Static analysis of digital filters. In *European Symposium On Programming (ESOP'04)*. Number 2986 in Lecture Notes in Computer Science. Springer-Verlag, Barcelona, Spain, 33–48.

FERET, J. 2005. The arithmetic-geometric progression abstract domain. In *6th conference on Verification, Model-Cecking and Abstract Interpretation (VMCAI'05)*, R. Cousot, Ed. Lecture Notes in Computer Science 3385. Springer-Verlag, Paris, France, 2–18.

FLANAGAN, C., LEINO, K. R., LILLIBRIDGE, M., NELSON, G., SAXE, J. B., AND STATA, R. 2002. Extended static checking for java. In *Proc. of the Conference on Programming Languages, Design and Implementation (PLDI'02)*. ACM Press, New York, NY, 234–245.

GRANGER, P. 1989. Static analysis of arithmetical congruences. In *International Journal of Computer Mathematics*. Vol. 30. 165–190.

HALBWACHS, N., LAGNIER, F., AND RAYMOND, P. 1993. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST '93)*. Workshops in Computing. Springer, Twente (Netherlands), 83–96.

HANDJIEVA, M. AND TZOLOVSKI, S. 1998. Refining static analyses by trace-based partitioning using control flow. In *5th International Static Analysis Symposium (SAS'98)*. Lecture Notes in Computer Science. Springer Verlag, 200–214.

HOLLEY, L. H. AND ROSEN, B. K. 1980. Qualified data flow problems. In *Proc. of the 7th ACM Symposium on Principles of Programming Languages (POPL'80)*. ACM Press, New York, NY, Las Vegas (Nevada), 68 – 82.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1988. Interprocedural slicing using dependence graphs. In *Conference on Programming Languages, Design and Implementation (PLDI'03)*. ACM Press, New York, Atlanta (USA), 35–46.

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

JEANNET, B., HALBWACHS, N., AND RAYMOND, P. 1999. Dynamic partitioning in analyses of numerical properties. In *6th Static Analysis Symposium SAS*. Lecture Notes in Computer Science, vol. 1694. Springer-Verlag, Venice (Italy), 39–50.

MAUBORGNE, L. 2004. ASTRÉE: Verification of absence of run-time error. In *Building the Information Society*. Kluwer Academic Publishers, Toulouse, France, Chapter 4, 384–392.

MELSKI, D. AND REPS, T. W. 2003. The interprocedural express-lane transformation. In *12th International Conference on Compiler Construction (CC'03)*. Lecture Notes in Computer Science. Springer-Verlag, Varsaw (Poland), 200–216.

MINÉ, A. 2001. The Octagon Abstract Domain. In *Analysis, Slicing and Transformation (in WCRE)*. IEEE. IEEE CS Press, Stuttgart, Germany, 310–319.

PLOTKIN, G. D. 1981. A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, Aarhus University, Denmark. Sept.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *22nd Symposium on Principles of Programming Languages (POPL'95)*. ACM Press, New York, San Francisco (USA), 49–61.

RIVAL, X. 2005. Understanding the origin of alarms in ASTRÉE. In *12th Static Analysis Symposium (SAS'05)*. LNCS, vol. 3672. Springer-Verlag, London (UK), 303–319.

SHARIR, M. AND PNUELLI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, Chapter 7, 189–233.

VENET, A. 1996. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *Static Analysis Symposium (SAS'96)*. Lecture Notes in Computer Science, vol. 1145. Springer-Verlag, Aachen, Germany.

Received September 2005.