Trace Partitioning in Abstract Interpretation Based Static Analyzers

Laurent Mauborgne and Xavier Rival

DI, École Normale Supérieure, 45 rue d'Ulm, 75 230 Paris cedex 05, France
Emails: `Laurent.Mauborgne@ens.fr` and `Xavier.Rival@ens.fr`

Abstract. When designing a tractable static analysis, one usually needs to approximate the trace semantics. This paper proposes a systematic way of regaining some knowledge about the traces by performing the abstraction over a partition of the set of traces instead of the set itself. This systematic refinement is not only theoretical but tractable: we give automatic procedures to build pertinent partitions of the traces and show the efficiency on an implementation integrated in the ASTRÉE static analyzer, a tool capable of dealing with industrial-size software.

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. For instance, let us consider the following program:

$$\begin{array}{l}\textbf{if}(x < 0)\{\ sgn = -1;\ \}\\ \textbf{else}\{\ sgn = 1;\ \}\end{array}$$

Clearly *sgn* is either equal to 1 or -1 at the end of this piece of code; in particular *sgn* cannot be equal to 0. As a consequence, dividing by *sgn* is safe. However, a simple interval analysis [7] would not discover it, since the lub (least upper bound) of the intervals $[-1,-1]$ and $[1,1]$ is the interval $[-1,1]$ and $0 \in [-1,1]$. A simple fix would be to use a more expressive abstract domain. For instance, the disjunctive completion [8] of the interval domain would allow the property to be proved: 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. Yet, the cost of disjunctive completion is prohibitive. Other domains could be considered as an alternative to disjunctive completion; yet, they may also be costly in practice and their design may be involved. For instance, common relational domains like octagons [15] or polyhedra [10] 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 [12] 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 [16]; 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 [13] where the history of computation is traced using an automaton chosen before the analysis.

Choosing of 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 [1] is to use profiling to determine the partitions, but this approach is relevant in optimization problems only.

The contribution of the paper is both theoretical and practical:

– We introduce a *theoretical framework* for trace partitioning, that can be instantiated in a broad series of cases. More partitioning configurations are supported than in [16] and the framework also supports *dynamic partitioning* (choice of the partitions during the abstract computation);
– We provide detailed practical information about the use of the trace partitioning domain. First, we describe the implementation of the domain; second, we review some strategies for partition creation during the analysis.

All the results presented in the paper are supported by the experience of the design, implementation and practical use of the Astrée static analyzer [2, 14]. This analyzer aims at certifying the absence of run-time errors (and user-defined non-desirable behaviors) in very large synchronous embedded applications such as avionics software. Trace partitioning turned out to be a very important tool to reach that goal; yet, this technique is not specific to the families of software addressed here and can be applied to almost any kind of software.

In Sect. 2, we set up a general theoretical framework for trace partitioning. The main choices for the implementation of the partitioning domain are evoked in Sect. 3; we discuss strategies for partitioning together with some practical examples in Sect. 4. Finally, we conclude in Sect. 5.

2 Theoretical Framework

This section supposes basic knowledge of the abstract interpretation framework [5]. For an introduction, the reader is referred to [9].

2.1 Definitions

Programs: We define a program P as a transition system $(\mathcal{S}, \rightarrow, \mathcal{S}_\iota)$ where $\mathcal{S}$ is the set of states of the program; $\rightarrow$ is the transition relation describing the possible execution elementary steps and $\mathcal{S}_\iota$ denotes the set of *initial states*.

Traces: We write $\mathcal{S}^\star$ for the set of all finite non-empty sequences of states. If σ is a finite sequence of states, σ_i will denote the $(i+1)^{\text{th}}$ state of the sequence, σ_0 the first state and $\sigma_\dashv$ the last state. We define $\varsigma(\sigma)$ as the set of all the states in σ. We extend this notation to sets of sequences: $\varsigma(\Sigma) \stackrel{\text{def}}{=} \bigcup_{\sigma \in \Sigma} \varsigma(\sigma)$.

If τ is a prefix of σ, we write $\tau \preceq \sigma$. A *trace* of the program P is defined as an element of $[\![P]\!] \stackrel{\text{def}}{=} \{\sigma \in \mathcal{S}^\star \mid \sigma_0 \in \mathcal{S}_\iota \wedge \forall i,\, \sigma_i \rightarrow \sigma_{i+1}\}$. Note that the set $[\![P]\!]$ is prefix-closed. An execution of the program is a possibly infinite sequence starting from an initial state and such that there is no possible transition from the final state, if any. Executions are represented by the set of their prefixes, thus avoiding the need to deal with infinite sequences.

2.2 Reachability Analysis

In order to prove safety properties about programs, one needs to approximate the set of reachable states of the programs. This is usually done in one step by the design of an abstract domain $D^\sharp$ representing sets of states and a concretization function that maps a representation of a set of states to the set of all traces containing these states only. In order to be able to refine that abstraction, we decompose it in two steps. The first step is the *reachability abstraction*, the second one the *set of states abstraction*.

We start from the most precise description of the behaviors of program P, given by the concrete semantics $[\![P]\!]$ of P, i.e the set of finite traces of P, so the concrete domain is defined as $\mathcal{P}_\preceq(\mathcal{S}^\star) \stackrel{\text{def}}{=} \{\Sigma \subseteq \mathcal{S}^\star \mid \Sigma \text{ is prefix-closed}\}$.

Reachability Abstraction: The set of reachable states of Σ can be defined by the abstraction $\alpha_R(\Sigma) \stackrel{\text{def}}{=} \{\sigma_\dashv \mid \sigma \in \Sigma\}$. Considering the concretization $\gamma_R(T) \stackrel{\text{def}}{=} \{\sigma \in \mathcal{S}^\star \mid \forall i, \sigma_i \in T\}$, we get a Galois connection $\mathcal{P}_\preceq(\mathcal{S}^\star) \underset{\alpha_R}{\overset{\gamma_R}{\leftrightarrows}} \mathcal{P}(\mathcal{S})$. This Galois connection will allow us to describe the relative precision of the refinements defined in the sequel of this section.

Set of States Abstraction: In the rest of the section, we will assume an abstract domain $D^\sharp$ representing sets of states and a concretization function[1] $\gamma : D^\sharp \rightarrow \mathcal{P}(\mathcal{S})$. Basically, $\gamma(I)$ represents the biggest set of states safely approximated by the (local) abstract invariant I. The goal of this abstraction is to compute an approximation of the set of states effectively.

[1] Abstract domains don't necessarily come with an abstraction function.

2.3 Trace Discrimination

Definition 1 (Covering). *A function $\delta : E \rightarrow \mathcal{P}(F)$ is said to be a* covering *of F if and only if $\bigcup_{x \in E}(\delta(x)) = F$.*

Definition 2 (Partition). *A function $\delta : E \rightarrow \mathcal{P}(F)$ is said to be a* partition *of F if and only if δ is a covering of F and $\forall x, y \in E$, $x \neq y \Rightarrow \delta(x) \cap \delta(y) = \emptyset$.*

Trace Discriminating Reachability Domain: Using a well-chosen function δ of $E \rightarrow \mathcal{P}(\mathcal{S}^\star)$, one can keep more information about the traces. We define the trace discriminating reachability domain D_R^δ as the set of functions from E to $\mathcal{P}(\mathcal{S})$, ordered pointwise. The trace discriminating reachability abstraction is $\alpha_R^\delta : \mathcal{P}_\preceq(\mathcal{S}^\star) \rightarrow D_R^\delta$, $\alpha_R^\delta(\Sigma)(x) \stackrel{\text{def}}{=} \{\sigma_\dashv \mid \sigma \in \Sigma \cap \delta(x)\}$. The concretization is then $\gamma_R^\delta(f) = \{\sigma \mid \forall \tau \preceq \sigma, \forall x,\ \tau \in \delta(x) \Rightarrow \tau_\dashv \in f(x)\}$ ($(\alpha_R^\delta, \gamma_R^\delta)$ form a Galois connection).

Comparing Trace Discriminating and Standard Reachability: Following [8], we compare the abstractions using the associated upper closure operators (the closure operator associated to an abstraction α, γ is $\gamma \circ \alpha$). The simple reachability upper closure maps any set of traces Σ to the set $\{\sigma \mid \forall i, \exists \tau \in \Sigma, \sigma_i = \tau_\dashv\}$ of traces composed of states in Σ. Thus, in order to give a better approximation, the new upper closure must not map any Σ to a set containing a state which was not in Σ. If δ is not a covering, then there is a sequence which is not in $\bigcup_{x \in E} \delta(x)$, and by definition of γ_R^δ, that sequence can be in any $\gamma_R^\delta(f)$, so it is very likely that D_R^δ is not as precise as the simple reachability domain. On the other hand, if $\bigcup_{x \in E} \delta(x) = \mathcal{S}^\star$, $\gamma_R^\delta \circ \alpha_R^\delta$ is always at least as precise as $\gamma_R \circ \alpha_R$.

A function $\delta : E \rightarrow \mathcal{P}(\mathcal{S}^\star)$ can distinguish a set of traces Σ_1 from a set Σ_2 if there exists x in E such that $\Sigma_1 \subseteq \delta(x)$ and $\Sigma_2 \cap \delta(x) = \emptyset$. The following theorem states that, if the covering δ can distinguish at least two executions with a state in common, then the abstraction based on δ is more precise than standard reachability. Moreover, the abstraction based on δ is always at least as precise as the standard reachability abstraction.

Theorem 1. *Let δ be a covering of $\mathcal{S}^\star$. Then, $(D_R^\delta, \gamma_R^\delta)$ is a more precise abstraction of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ than $(\mathcal{S}, \gamma_R)$. Moreover, if there are two elements of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ which share a state and are distinguished by δ, then the abstraction $(D_R^\delta, \gamma_R^\delta)$ of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ is strictly more precise than $(\mathcal{S}, \gamma_R)$.*

Proof. By definition, $\gamma_R^\delta \circ \alpha_R^\delta(\Sigma)$ is the set of traces σ such that $\forall \tau \preceq \sigma, \forall x$, $(\tau \in \delta(x) \Rightarrow \exists \upsilon \in \Sigma \cap \delta(x),\ \tau_\dashv = \upsilon_\dashv)$. $\exists \upsilon \in \Sigma \cap \delta(x), \tau_\dashv = \upsilon_\dashv$ implies $\exists \upsilon \in \Sigma, \sigma_i = \upsilon_\dashv$. If δ is a covering, then for all τ, there is at least one x such that $\tau \in \delta(x)$. So $\gamma_R^\delta \circ \alpha_R^\delta \subseteq \gamma_R \circ \alpha_R$, meaning that the abstraction $(D_R^\delta, \gamma_R^\delta)$ of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ is more precise than $(\mathcal{S}, \gamma_R)$.

To prove that we have a strictly more precise abstraction, we exhibit a set of traces Σ such that $\gamma_R^\delta \circ \alpha_R^\delta(\Sigma)$ is strictly smaller than $\gamma_R \circ \alpha_R(\Sigma)$. Following the hypothesis, let Σ_1, Σ_2, s and x be such that s is a state in $\varsigma(\Sigma_1) \cap \varsigma(\Sigma_2)$, and

$\Sigma_1 \subseteq \delta(x)$ and $\Sigma_2 \cap \delta(x) = \emptyset$. Let σ be a sequence of Σ_1 such that $\sigma_\dashv = s$ (this is always possible because Σ_1 is an element of $\mathcal{P}_\preceq(\mathcal{S}^\star)$, and as such prefix-closed). Let $\Sigma = (\varsigma(\delta(x)) - \{s\})^\star \cup \Sigma_2$. Then $\varsigma(\sigma) \subseteq \varsigma(\Sigma)$, so σ is in $\gamma_R \circ \alpha_R(\Sigma)$. But whatever $\upsilon \in \Sigma \cap \delta(x)$, υ does not contain s, so it cannot end with s, hence $\sigma \notin \gamma_R^\delta \circ \alpha_R^\delta(\Sigma)$. $\square$

Corollary 1. *If δ is a non trivial partition of $\mathcal{S}^\star$ (no $\delta(x)$ is $\mathcal{S}^\star$), then the abstraction $(D_R^\delta, \gamma_R^\delta)$ of $\mathcal{P}_\preceq(\mathcal{S}^\star)$ is strictly more precise than $(\mathcal{S}, \gamma_R)$.*

Proof. Suppose that for an x, $\forall s \in \varsigma(\delta(x))$, $\forall y \neq x, s \notin \varsigma(\delta(y))$. Then, because δ is a covering, all sequences containing a state of $\delta(x)$ is in $\delta(x)$, which means $\delta(x) = (\varsigma(\delta(x)))^\star$. Since δ is a non trivial partition of $\mathcal{S}^\star$ not all $\delta(x)$ can be of this form. So there is an x and a y such that $\delta(x)$ distinguishes between $\delta(x)$ and $\delta(y)$ having a state in common. $\square$

In practice so far, only partitions will be considered, so the results of Theorem 1 apply.

2.4 Some Trace Partitioning Abstractions

In this paragraph, we instantiate the framework to various kinds of partitions. In this instantiation we suppose a state can be decomposed into a control state in $\mathcal{L}$ and a memory state in $\mathcal{M}$. Thus $\mathcal{S} = \mathcal{L} \times \mathcal{M}$. We also assume that the abstract domain $D^\sharp$ forgets about the control state, just keeping an approximation of the memory states.

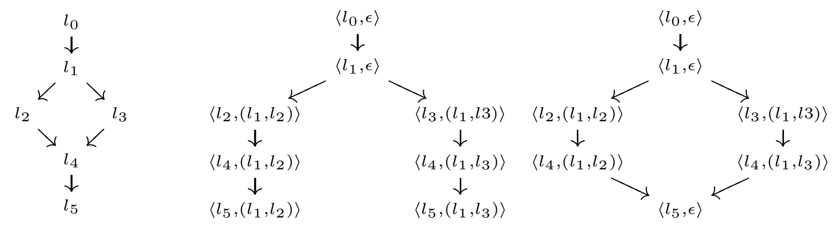
We illustrate some partitions with a simple abstract program containing a conditional on Fig 1.

Final Control State Partition: Let $\delta_\mathcal{L} : \mathcal{L} \to \mathcal{P}_\preceq(\mathcal{S}^\star)$ be the partition of $\mathcal{S}^\star$ based on the final control state: $\delta_\mathcal{L}(l) \overset{\text{def}}{=} \{\sigma \in \mathcal{S}^\star \mid \exists \rho,\ \sigma_\dashv = (l, \rho)\}$. This partition is very common and usually done silently when designing the abstract semantics. It leads to the abstraction $(D_l^\sharp, \gamma)$ of D, where $D_l^\sharp \overset{\text{def}}{=} \mathcal{L} \to D^\sharp$ and $\gamma(\mathcal{I}) \overset{\text{def}}{=} \{\sigma \in \mathcal{P}_\preceq(\mathcal{S}^\star) \mid \forall i,\ \sigma_i = (l_i, \rho_i) \wedge \rho_i \in \gamma(\mathcal{I}(l_i))\}$.

Control Flow Based Partition: In [16], Tzolovski and Handjieva introduced trace-based partitioning using control flow. To simplify, they proposed to extend the control states with an history of the control flow in the form of lists of tags t_i or f_i (meaning that the test number i was true or false). Then, they perform a final control state partition on this new set of control states. In order to keep the set of control states finite, they associate with each while loop an integer limiting the number of t_i to be considered.

Formally, let $\mathcal{B} \subseteq \mathcal{L}$ be the set of control points introducing a branching (e.g. conditionals, while loops...). We define $\mathbb{C} \overset{\text{def}}{=} \{(b, l) \in \mathcal{B} \times \mathcal{L} \mid \exists \rho, \rho' \in \mathcal{M},\ (b, \rho) \to (l, \rho')\}$ as the set of possible branch choices in the program. Note that in a branch choice (b, l), l is necessarily directly accessible from b. In order to define the trace partition used in [16], we define the control flow abstraction of a trace as the sequence $\text{cf}(\sigma) \subseteq \mathbb{C}^\star$ made of the maximal sequence of branch choices taken in the trace. Then, the control flow based partition is defined as the partition $\delta_{\text{cf}} : \mathcal{L} \times \mathbb{C}^\star \rightarrow \mathcal{P}(\mathcal{S}^\star)$, $\delta_{\text{cf}}(l, \beta) \overset{\text{def}}{=} \{\sigma \in \delta_{\mathcal{L}}(l) \mid \text{cf}(\sigma) = \beta\}$.

In order to keep the partition finite, [16] limits the number of partitions per branching control points. They use a parameter $\kappa : \mathcal{B} \rightarrow \mathbb{N}$ in the abstraction function. The κ-limiting abstraction is defined as $\lambda_\kappa(\beta)$ which is the subsequence of β obtained by deleting the branching choices $\beta_i = (b, l)$ such that if b is the conditional of a loop, the loop have been taken more than $\kappa(b)$ consecutive times (if b is a simple branching, it is deleted if $\kappa(b)$ is 0). Then, if we use $\lambda_\kappa \circ \text{cf}$ instead of cf, the effect will be to merge partitions distinguishing the simple branchings for which κ is 0 and the iterations of the loops after some point. So the partition finally used is $\delta_{\text{cf}} : \mathcal{L} \times \lambda_\kappa(\mathbb{C}^\star) \rightarrow \mathcal{P}(\mathcal{S}^\star)$.



l_0
l_1
l_2
l_3
l_4
l_5
$\langle l_0, \epsilon \rangle$
$\langle l_1, \epsilon \rangle$
$\langle l_2, (l_1, l_2) \rangle$
$\langle l_3, (l_1, l3) \rangle$
$\langle l_4, (l_1, l_2) \rangle$
$\langle l_4, (l_1, l_3) \rangle$
$\langle l_5, (l_1, l_2) \rangle$
$\langle l_5, (l_1, l_3) \rangle$
$\langle l_0, \epsilon \rangle$
$\langle l_1, \epsilon \rangle$
$\langle l_2, (l_1, l_2) \rangle$
$\langle l_3, (l_1, l3) \rangle$
$\langle l_4, (l_1, l_2) \rangle$
$\langle l_4, (l_1, l_3) \rangle$
$\langle l_5, \epsilon \rangle$

a. Control State Partition **b.** Control Flow Partition **c.** Merging Control Flow Partition

Fig. 1. Some partitions for the program $l_0 : s_0$; $l_1 :$ **if**$(c)\{\, l_2 : s_1; \}$ **else** $\{\, l_3 : s_2; \}\, l_4 : s_3$; $l_5 : s_4$;

2.5 Designing Pertinent Partitions

The control flow based partition can give very precise results but is also very expensive. Even if the parameter κ is very restrictive (keeping the useful partitions only) the setting will keep the partitions after they are needed. This is very costly because each partitioning point multiplies the cost by 2 at least. Using the more general setting we describe, it is possible to define more pertinent and less expensive partitions. We describe here two examples.

Merging Control Flow: In order to reduce the cost it is possible to include in the same partition, not only the traces before a branching point, but also the traces exceeding a certain following control point. We do that if we guess that the partition based on the branching is not useful after this point. In conjunction with the final state control partition, it means that at each control point, we keep only some presumably useful partitions. Formally, we introduce a new parameter, $\mathbb{M} \subseteq \mathcal{L}$ and modify cf such that we forget everything that is before a control point in $\mathbb{M}$. To be more precise, it is even possible to use a function $\mathbb{M} \rightarrow \mathcal{B}$ and forget only the branching points corresponding to the merging point. On the example of Fig. 1, if $\mathbb{M} = \{l_5\}$, we get the partition in Fig. 1-c.

Value Based Trace Partition: The most adapted information for the definition of the partitions might not lie in the control flow. For instance, to regain some complex relationship between a variable with a few possible values and the other variables, we can add a partition according to the values of the variable at a given control point. The advantage of this approach is the very low implementation cost compared to the design of a new relational domain.

2.6 The Trace Partitioning Abstract Domain

The partitions of $\mathcal{S}^\star$ are naturally ordered by the notion of being finer (they even form a complete lattice).

Definition 3. *A partition δ_1 is finer than δ_2 if $\forall x, \exists y,\ \delta_1(x) \subseteq \delta_2(y)$. We write $\delta_1 \precsim \delta_2$.*

This lattice can be the basis of an abstract domain, using techniques inspired of the cofibered domains of [17]. The interest of such a domain is twofold. First, it allows *dynamic partitioning* by changing the partitions during the analysis (depending on the properties of the program inferred during the analysis). Second, it gives the possibility of using infinite partitions (or very big partitions) which can be abstracted away during the computation of the invariants by widening.

Basis: We introduce the notion of equivalence between partitions: δ_1 is *equivalent* to δ_2 if $\forall x, \exists y,\ \delta_1(x) = \delta_2(y)$. The *basis* of the trace partitioning abstract domain is the set $\mathfrak{T}$ of all partitions of $\mathcal{S}^\star$ up to equivalence.

Let δ_1 and δ_2 in $\mathfrak{T}$. If $\delta_1 \precsim \delta_2$, then the abstraction $D_R^{\delta_1}$ is more precise than $D_R^{\delta_2}$ if compared as closure operators, as seen in Sect. 2.3. The most precise (finest) partition distinguishes all traces: it is $\{\{\sigma\} \mid \sigma \in \mathcal{S}^\star\}$. Note that the standard reachability domain corresponds to the supremum element of $\mathfrak{T}$: we define $\delta_0 : \{0\} \rightarrow \mathcal{P}_{\preceq}(\mathcal{S}^\star)$ as $\delta_0(0) \stackrel{\text{def}}{=} \mathcal{S}^\star$. It is obvious that $(\mathcal{S}, \alpha_R)$ is isomorphic to $(D_R^{\delta_0}, \alpha_R^{\delta_0})$.

Definition 4 (partitioning abstract domain). *The* trace partitioning abstract domain*, $\mathbb{D}^\sharp$, is defined as: $\mathbb{D}^\sharp = \{(\delta, \mathcal{I}) \mid \delta \in \mathfrak{T} \wedge \mathcal{I} \in D_R^\delta\}$*

Application of Dynamic Partitioning: Choosing the partitions at analysis time is crucial in many cases. For instance, the analyzer should be able to decide whether or not to operate value based partitioning (Sect. 2.5) during the analysis; indeed, in case the analysis gives no precise information about the range of an integer variable i, partitioning the traces with the values of i would lead to a dramatic analysis cost, possibly for no precision improvement. Other applications include the dynamic choice of the number of unrolled iterations in a loop (Sect. 4.4) or the analysis of recursive functions [4].

Widening: Because the basis contains infinite ascending chains, we need a widening to use the trace partitioning domain in practice. We can produce a widening on $\mathbb{D}^\sharp$ as soon as we have a widening on the basis $\nabla_{\mathfrak{T}}$ and a widening on the set of states abstract domain. This widening ∇ can be derived by a construction similar to [17]. To compute $(\delta_1, \mathcal{I}_1)\nabla(\delta_2, \mathcal{I}_2)$, we compute $\delta = \delta_1 \nabla_{\mathfrak{T}} \delta_2$ and then widen the best approximations of $\mathcal{I}_1$ and $\mathcal{I}_2$ in D_R^δ.

3 Implementation of the Domain

We now provide an overview of the data structures and algorithms which turned out the most efficient for the implementation of the trace partitioning domain.

3.1 Partition Creation and Merge

Partitions are created at *partition begin* control points. Such points are defined by *partitioning directives* The choice of the points where partition directives should be inserted will be discussed in Sect. 4.1. The main directives we implemented are: If-partitioning, Loop-partitioning, Call-stack handling, Value-partitioning. A *partition end* point merges some or all partitions. Partition begins and partition ends may or may not be well parenthesized as far as the soundness of the analysis is concerned. We may imagine some partitioning strategies that would merge older partitions first and result in more precise results. ASTRÉE assumes that partition begins and partition ends are well parenthesized for the sake of efficiency only.

A token stands for an element of the partitions observed at a (set of) control point(s). As suggested in Sect. 2.5, we partition traces according to some conditions like the choice of a branch in a conditional structure. We let such a condition be denoted by a *pre-token*. Then, a *token* is defined as the series of such conditions the execution flowed through, hence can be considered a stack of tokens. We choose a stack here instead of a list or a set, since the last partition opened should be closed first so the order of pre-tokens should be preserved.

Definition 5 (tokens). Pre-tokens *($p \in \mathcal{P}$) and* tokens *($t \in \mathcal{T}$) are defined by the following grammar (which can be extended):*

$$\begin{array}{llcl} p ::= & \texttt{If\_true}(l) \mid \texttt{If\_false}(l) \mid \texttt{Val\_Var}(v,k,l) & \qquad & t ::= \epsilon \\ \mid & \texttt{While\_unroll}(l,k) \mid \texttt{While\_iter}(l) \mid \texttt{Fun\_Call}(f,l) & & \quad \mid \ t.p \end{array}$$

where f is a function name, l a program point, v a program variable, k an integer.

For instance, the pre-token $\texttt{Fun\_Call}(f, l)$ characterizes traces that called the function f at point l and have not returned yet. The pre-token $\texttt{If\_true}(l)$ characterizes the traces that flowed through the true branch of the conditional at point l and have not reached the corresponding merge point yet. The pre-token $\texttt{Val\_Var}(v, k, l)$ characterizes the traces that have reached l with the condition $v = k$ satisfied and have not reached the corresponding merge point yet. The pre-token $\texttt{While\_unroll}(l, k)$ characterizes the traces that spent exactly k iterations in the loop at point l; the pre-token $\texttt{While\_iter}(l)$ characterizes the traces that spent more iterations in the loop than the number of unrolls for this loop.

A partition in the sense of Sect. 2.3 is defined by a tuple (l, t) where l is a control state and t a token, since we partition the system with the control states. In other words, we fix $E = \mathcal{L} \times \mathcal{T}$.

3.2 Abstract Values

Let us consider a control point l and a set P of partitions observed at this point, during the analysis. Then, the prefix of the tokens corresponding to the partitions in P can be shared. By construction, the set of tokens corresponding to the partition at a given program point is prefix-closed; hence, a local invariant is represented by a tree; a path from the root to a leaf in such a tree corresponds to a uniquely defined token. We write $D^\sharp$ for the underlying domain abstracting sets of states. We assume that $D^\sharp$ features a least element $\bot$ and a lub $\sqcup$.

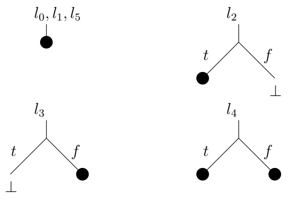
Definition 6 (abstract value). *The* abstract partitioning domain *is defined by $\mathbb{D}^\sharp = \mathcal{L} \to D^\sharp_p$, with the pointwise ordering, where elements of $D^\sharp_p$ are defined by induction as follows:*

$$d \ ::= \ \mathrm{leaf}[v]\,(v \in D^\sharp) \ \mid \ \mathrm{node}[\phi]\,(\phi \in \mathcal{P} \to D^\sharp_p)$$

Roughly speaking, an element of $D^\sharp_p$ encodes a function $\mathcal{T} \to D^\sharp$; so an element of $\mathbb{D}^\sharp$ corresponds to an element of $\mathcal{L} \to (\mathcal{T} \to D^\sharp)$ (which is equivalent to $(\mathcal{L} \times \mathcal{T}) \to D^\sharp$). Hence, $\mathbb{D}^\sharp$ is obtained from the trace discriminating reachability domain (Sect. 2.3) by composition the abstraction $\gamma : D^\sharp \to \mathcal{P}(\mathcal{S})$ pointwise. It is a particular instance of a partitioning abstract domain as defined in Sect. 2.6. Furthermore, we insure sharing of common parts of the underlying numerical invariants across partitions, which reduces memory requirements.

Let us consider the program below (example of Sect 2.4 continued). We assume that the conditional at point l_1 is partitioned and that the corresponding merge-point is l_4. The shape of the partitions (hence, the local abstract invariants) at various program points is displayed below (the $\bullet$ symbol stands for a value in $D^\sharp$).

```
$l_0$ : $s_0$;
$l_1$ : **if**($c$){
$l_2$ :         $s_1$
     }**else**{
$l_3$ :         $s_2$}
$l_4$ : $s_3$;
$l_5$ : $s_4$;
```



3.3 Iteration Scheme

The computation of an approximation of $[\![P]\!]$ requires some counterpart for the transition relation $\rightarrow$. For all $l, l' \in \mathcal{L}$, we let $\vartheta_{l,l'} : D_p^\sharp \rightarrow D_p^\sharp$ denote the abstract transfer function corresponding to the edge (l, l'). It is *sound* if and only if $\forall d \in D_p^\sharp,\ \forall \rho, \rho' \in \mathcal{M},\ \rho \in \gamma_p(d) \wedge (l, \rho) \rightarrow (l', \rho') \Rightarrow \rho' \in \gamma_p(\vartheta_{l,l'}(d))$.

In case $(\vartheta_{l,l'})$ is a family of sound abstract transfer functions, an abstract semantic function can be derived, such that $F \circ \gamma \subseteq \gamma \circ F^\sharp$; iterating it from the least element and using standard widening techniques yield a sound approximation of $[\![P]\!]_t$, i.e. the definition of a sound abstract semantics $[\![P]\!]^\sharp \in \mathbb{D}^\sharp$.

3.4 Abstract Transfer Functions

We let $\mathbb{C}$ denote the set of conditional expressions. We define the main abstract transfer functions below:

- **Non partitioning-related transfer functions**. we consider the case of the operator $\mathbf{guard} : \mathbb{C} \times D_p^\sharp \rightarrow D_p^\sharp$, which is the abstract counterpart of the concrete condition testing ($\mathbf{guard}_n$ is the operator provided by $D^\sharp$):

$$\begin{aligned} \mathbf{guard}(C, \mathrm{leaf}[v]) &= \mathrm{leaf}[\mathbf{guard}_n(C, v)] \\ \mathbf{guard}(D, \mathrm{node}[\phi]) &= \mathrm{node}[p \mapsto \mathbf{guard}(D, \phi(p))] \end{aligned}$$

- **Partition creation (create** : $(\mathcal{T} \rightarrow \mathbb{C}) \times D_p^\sharp$**):** if $C : \mathcal{T} \rightarrow \mathbb{C}$ is a family of conditions associated to all the created partitions, then $\mathbf{create}(C, d)$ creates a new partition defined by the condition $C(t)$ for each token t (Sect. 2.4). It can be written with an auxiliary function to accumulate prefixes:

$$\begin{aligned} \mathbf{create}(C, d) &= \mathbf{create}_0(C, \epsilon, d) \\ \mathbf{create}_0(C, t, \mathrm{leaf}[v]) &= \mathrm{node}[p \mapsto \mathrm{leaf}[\mathbf{guard}_n(C(t), v)]] \\ \mathbf{create}_0(C, t, \mathrm{node}[\phi]) &= \mathrm{node}[p \mapsto \mathbf{create}_0(C, t.p, \phi(p))] \end{aligned}$$

- **Partition merge (merge** : $\mathcal{P}(\mathcal{T}) \times D_p^\sharp \rightarrow D_p^\sharp$**):** $\mathbf{merge}(X, d)$ yields a new abstract value whose partitions are elements of the set X (where the elements of X denote covering of the traces at the current program point and form another prefix-closed set of tokens); basically **merge** merges some existing partitions so as to restrict to a smaller set of partitions (Sect. 2.5).

In practice X is a subset of the set of prefixes of the tokens corresponding to the partitions in d. It is defined in a similar way as **merge**:

$$\begin{aligned}
\mathbf{merge}(X, d) &= \mathbf{merge}_0(X, \epsilon, d) \\
\mathbf{merge}_0(X, t, \mathrm{leaf}[v]) &= \mathrm{leaf}[v] \\
\mathbf{merge}_0(X, t, \mathrm{node}[\phi]) &= \mathrm{leaf}[\sqcup_n \{v \mid \mathrm{node}[\phi] \text{ ancestor of } \mathrm{leaf}[v]\}] \text{ if } t \in X \\
\mathbf{merge}_0(X, t, \mathrm{node}[\phi]) &= \mathrm{node}[p \mapsto \mathbf{merge}_0(X, t.p, \phi(p))] \qquad \text{otherwise}
\end{aligned}$$

The program displayed in Sect. 3.2 exemplifies partition creation (between l_1 and l_2, l_3) and partition merge (between l_4 and l_5).

4 Trace Partitioning in Practice

The theoretical framework and our implementation of the trace partitioning abstract domain allow the introduction of a huge number of different partitions to refine the analysis. One last issue is to find which partition will indeed help the analysis while not impending too much on the complexity.

4.1 Manual Directives and Automatic Strategies

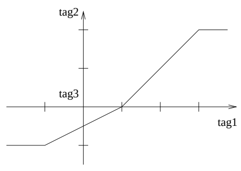
Our implementation allows the end-user to make such choices by specifying partitioning directives such as control flow or value partitions, or partition merges in the program code. Some functions to be partitioned according to the control flow (as in [16]) can also be specified (a merge is inserted at the return point). Although this possibility proved very useful for static analysis experts to improve the precision of an analysis, it is quite likely that most end-users would miss opportunities to partition or propose too costly partitions. That is why we believe that static analyzer designers should also devise automatic strategies adapted to the kind of programs they wish to analyze precisely.

Automatic strategies for trace partitioning stem from imprecisions observed in the analysis of the target programs. When those imprecisions are understood, they can be captured by semantic patterns. In the three following sections, we present typical examples of imprecisions solved by a partitioning, together with the strategy which provides the ad-hoc partitions.

4.2 Linear Interpolations

Example: Computing functions defined by linear interpolation is a rather common task in real-time embedded software (e.g. functions describing the command reaction laws). We consider here the case of the piece of code below that inputs a value in variable x and computes in variable y the result of a linear interpolation. The loop checks in which range the variable x lies; then, the corresponding value is computed accordingly.

```
$l_0$ :  int $i = 0$;
$l_1$ :  **while**($i < n$ && $x > tx[i+1]$)        $tc = \{0; 0.5; 1; 0\}$
$l_2$ :      $i++$;                                  $tx = \{0; -1; 1; 3\}$
$l_3$ :  $y = tc[i] \times (x - tx[i]) + ty[i]$      $ty = \{-1; -0.5; -1; 2\}$
$l_4$ :  ...
```



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

Fig. 2. Linear interpolation

Non Relational Analysis: The execution of this fragment of code is expected to yield a value in the range $[-1, 2]$ whatever the value of x. 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 imprecisions 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.

Analysis with Trace Partitioning: Our approach to this case 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:

– no iteration: $i = 0$; $x < -1$; $y = -1$
– one 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 values that could be observed in concrete executions).

The partitions generated in this example correspond to l_0, $(l_1, 0)$, $(l_2, 0)$, $(l_1, 1)$, $(l_2, 1)$, $(l_3, 0)$, $(l_3, 1)$, l_4; the partition associated to l_i is the set of traces ending at l_i; the partition associated with (l_i, j) is the set of traces ending at l_i after completing j iteration in the loop. This set of partitions is determined during the analysis, with directives requesting partitioning at point l_1 and merge at point l_4.

As we noted before, the trace partitioning turns out to be a reasonable alternative to the design of a more involved relational domain.

Strategy Implemented in Astrée**:** The imprecision observed when analyzing the linear interpolation have two causes: first, the expression at point l_3 computes the sum of two expressions which are not independent. Non-relational domains are quite imprecise in such cases (e.g. if $x \in [-1, 1]$, a dumb interval analysis will find $x - x \in [-2, 2]$). The second cause is that, through the use of arrays, the expression makes an implicit disjunction. Most efficient relational domains are imprecise on disjunctions (unions is usually the abstract operation that loses the most precision).

In Astrée, we use the following strategy to build partitions solving this kind of problem: first, we identify expressions e with an array access in a sum, such that another element of the sum is related to the index of the array access (it is the case for the expression at l_3, Fig 2). Then, we look backward for the last non-trivial assignment of that index. If it is in a loop, then we partition the loop, otherwise, we partition the values of the index after its assignment. In addition, we partition all the control flow between the index assignment and the expression e. We keep the analysis efficient by merging those partitions right after the expression e is used.

4.3 Barycenter

Finding precise invariants when analyzing divisions sharing a variable in the dividend and divider require either complex ad-hoc transfer functions (as in [11]) or guessing an appropriate linear form [3]. If the variable found in the dividend and divider ranges in a small set (less than, say, a thousand) we can get very precise results by partitioning the traces according to the dynamic values of that variable. Such partition will be quite cheap because its scope will be very local: it is only necessary to partition right before the assignment, and then we can merge right after the assignment.

A simple example using division is the computing a barycenter between two values. One expects the barycenter to be between those two values. But it is in fact a difficult task with classical abstract domains. In the following figure, we show an example of classical barycenter computation. As it is the case in many real-time embedded systems, this computation is inside an infinite loop.

Using non-relational domains, one cannot prove that x will never overflow, whereas it is a simple matter, partitioning the values during just one instruction, to prove that x stays in $[-100, 100]$. If we suppose $x \in [-100, 100]$ and $r \in [0, 50]$, we get $(x * r + \mathrm{random}(-100, 100))/(r + 1)$ in $[-5100, 5100]$. whereas if we take any particular r in $[0, 50]$, we can compute that the expression is in $[-100, 100]$.

```
l0 :   int r = 0;  float x = 0.0;
l1 :   while(true){
l2 :          r = random(0, 50);
l3 :          x = (x * r + random(−100, 100))/(r + 1);
l4 :   }
```

4.4 Loop Unrolling

Analyzing separately the n first iterations of a loop may greatly improve the precision of the final result, as is the case of the following examples:

– Some families of embedded programs –as those addressed by ASTRÉE– consist in a large loop; the initialization of some variables might be done during the first iteration(s), so it might be helpful to distinguish the first iterations during the analysis so as to take into account their particular role.
– Unrolling a loop might turn weak updates into strong updates. For instance, let us consider the program **for**$(i = 0; i < n; i = i + 1)\{t[i] = i\}$, which initializes an array t (we assume that all cells are equal to 0 before the loop). If we perform no unrolling of the loop, a simple interval analysis infers the interval constraint $i \in [0, n-1]$; so the assignment to $t[i]$ is a weak update and in the end we get the family of constraints $\forall j,\ t[j] \in [0, n-1]$.
  The body of the loop is very small; hence, the complete unrolling of all the iterations of the loop is not too costly. It leads to the much more precise family of constraints $\forall j,\ t[j] = j$.

In practice, defining the control point corresponding to the loop as the partitioning point and the control points right after the loop as the merging point leads to the unrolling of the n first iterations. This allows for more precise results in the analysis of the loop; yet does not make the analysis of the subsequent statements more costly.

The analysis of an unrolled loop starts with the computation of an invariant for the n first iterations; after that an invariant for all the following iterations is achieved thanks to the standard widening techniques. It is also possible to start with a partition of the whole loop, and decide during the computation of the invariants, that because of the growth of n, this partition might not be finite (or be too large) and thus, as described in Sect. 2.6, to use a coarser partition.

4.5 Experimental Results

We tested ASTRÉE on a family of industrial size embedded codes. All partitions where chosen automatically. In the following table, we show the results for the analysis without partitioning and then **with partitioning**. For each program, we provide the size of the code as a number of LOCs, the number of iterations required for the analysis of the main loop (these programs all consist in a series of tasks executed at every clock tick, which is equivalent to a main loop), the analysis time in minutes (on a 3 GHz Bi-opteron, with 8 Gb of RAM), the memory consumption in megabytes and the number of alarms.

Program	test 1		test 2		test 3		test 4	
Code size (LOCs)	70 000		65 000		215 000		380 000	
Iterations	48	**43**	33	**32**	80	**59**	163	**62**
Analysis time (minutes)	44	**70**	21	**28**	180	**330**	970	**680**
Memory peak (Mb)	520	**550**	340	**390**	1 100	**1 300**	1 800	**2 200**
Alarms	658	**0**	552	**2**	4 963	**1**	6 693	**0**

The results show the expected positive impact on the precision, as the number of alarms of the analyzer is always reduced with partitioning; in all cases the analysis with partitioning results in a very low number of alarms whereas the analysis without partitioning yields huge numbers of false positives –much beyond what the end-user could check by hand. The analysis being more precise, less iterations to reach a post fixpoint are required with trace partitioning. In the case of test 4, the number of iterations required by the analysis with partitioning disabled even causes a much higher analysis time. Of course, using partitioning each iteration takes longer, but the cost in time and memory is very reasonable.

5 Conclusion

The partitioning of abstract domains was first introduced in [6]; it describes trace partitioning on the concrete level (sets of traces). We proposed to use such partitions to guide a restricted kind of disjunctions. Disjunctive completion usually gives very precise results, but has an exponential cost, that is why in practice, one must restrict the number of disjunctions. The idea of using the control flow to chose which disjunctions to keep was first introduced in [16], but still their proposal was not practical, especially for large programs. What we proposed here is a more general and flexible framework which allowed the Astrée static analyzer to be very precise on industrial programs [3].

Future work includes the extension of the partitioning abstract domain with backwards transfer functions, so as to do backwards analysis. A second extension would be to partition traces with the number of times a property of the memory state $\mathcal{P}$ was satisfied at a control point l, generalizing the condition-guided partitioning we presented here. This would allow expressing some kind of temporal properties of traces, by distinguishing traces that satisfied $\mathcal{P}$ at least once and the others.

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 during the early developments of this work.

References

[1] Ammons, G., and Larus, J. R. Improving data-flow analysis with path profiles. In *Conference on Programming Language Design and Implementation (PLDI'98)* (1998), ACM Press, pp. 72–84.

[2] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. 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*, T. Mogensen, D. Schmidt, and I. Sudborough, Eds., no. 2566 in LNCS. Springer-Verlag, 2002.

[3] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. A static analyzer for large safety-critical software. In *Conference on Programming Language Design and Implementation (PLDI'03)* (2003), ACM Press, pp. 196–207.

[4] Bourdoncle, F. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming 4*, 2 (1992), 407–435.

[5] Cousot, P. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, Université de Grenoble, 1978.

[6] Cousot, P. 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, 1981, ch. 10.

[7] Cousot, P., and Cousot, R. Abstract intrepretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL'77)* (1977), ACM Press, pp. 238–252.

[8] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles of Programming Languages (POPL'79)* (1979), ACM Press, pp. 269–283.

[9] Cousot, P., and Cousot, R. Basic concepts of abstract interpretation. In *Building the Information Society*. Kluwer Academic Publishers, 2004, ch. 4.

[10] Cousot, P., and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages (POPL'78)* (1978), ACM Press, pp. 84–97.

[11] Feret, J. Static analysis of digital filters. In *European Symposium on Programming (ESOP'04)* (2004), no. 2986 in LNCS, Springer-Verlag.

[12] Granger, P. Static Analysis of Arithmetical Congruences. *Int. J. Computer. Math. 30* (1989).

[13] Holley, L. H., and Rosen, B. K. Qualified data flow problems. In *7th ACM Symposium on Principles of Programming Languages (POPL'80)* (1980), ACM Press, pp. 69–82.

[14] Mauborgne, L. ASTRÉE: Verification of absence of run-time error. In *Building the Information Society*. Kluwer Academic Publishers, 2004, ch. 4.

[15] Miné, A. The octagon abstract domain. In *AST* (2001), IEEE, IEEE CS Press.

[16] Tzolowski, S., and Handjieva, M. Refining static analyses by trace-based partitionning using control flow. In *Static Analysis Symposium (SAS'98)* (1998), vol. 1503 of *LNCS*, Springer-Verlag.

[17] Venet, A. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *Static Analysis Symposium (SAS'96)* (1996), vol. 1145 of *LNCS*, Springer-Verlag.