Static Analysis by Abstract Interpretation of Sequential and Multi-Thread Programs

Antoine Miné

Static Analysis by Abstract Interpretation of Sequential and Multithreaded Programs

Antoine Miné*

CNRS & École Normale Supérieure
45, rue d'Ulm
75005 Paris, France
mine@di.ens.fr

Abstract. In the realm of embedded critical systems, it is crucial to guarantee the correctness of programs before they are deployed. Static analyzers can help by detecting at compile-time potentially erroneous program behaviors: they perform sound over-approximations to achieve an efficient analysis while not missing any potential behavior. We discuss the systematic design of such analyzers using abstract interpretation, a general theory of semantic approximation. After recalling the classic construction of static analyzers for sequential programs by abstraction of the concrete trace semantics, we introduce abstractions to derive thread-modular analyzers for multithreaded programs, borrowing ideas from rely/guarantee proof methods. Finally, we present two static analyzer tools, Astrée and AstréeA, that are used to check for run-time errors in large sequential and multithreaded embedded industrial avionic C applications.

1 Introduction

It is crucial to guarantee the correctness of programs before they are deployed, especially in the realm of embedded critical systems, where software cannot be corrected during missions and a single mistake can have dramatic consequences. Testing, the most widespread verification method employed in industry, is very efficient at catching errors, but it is costly and cannot, for efficiency reasons, test all executions. Hence, testing can miss errors. This is especially true for parallel and multithreaded programs: the huge number of possible thread interleavings causes a combinatorial explosion of executions, while some errors only appear in a tiny fraction of them (such as data-races). Formal methods, on the other hand, can provide rigorous guarantees that all the executions are correct.

We consider here static analyses, able to inspect program sources in order to find defects. We only consider semantic analyses, that are based on a mathematical notion of program behaviors, as opposed to syntax-based style checkers.

* This work was partially supported by the INRIA project "Abstraction" common to CNRS and ENS in France, and by the project ANR-11-INSE-014 from the French *Agence nationale de la recherche*.

Unlike proof methods, which require the user to provide annotations, these analyses run on the original, unannotated source without human intervention. Full automation and efficiency imply that such analyses are approximated. We impose soundness: unlike testing, program behaviors are over-approximated so that, if an error is present, it will be detected. However, the analysis can report spurious errors. Sound static analyzers have been used for decades in applications, such as program optimization, where precision is not critical. Recent progress had lead to the design of tools able to check for simple but important safety properties (such as the absence of run-time error) with few or zero false alarms. We participated in the design of two of them [2]: Astrée, an industrial-strength analyzer that checks for run-time errors in synchronous embedded C code, and AstréeA, its prototype extension targeting multithreaded embedded C applications.

These analyzers are based on abstract interpretation, which is a general theory of program semantics introduced by Cousot and Cousot in the late '70s [5]. Abstract interpretation stems from the observation that many semantics can be uniformly expressed as fixpoints of operators, after which seemingly unrelated semantics can be compared. It expresses formally the fact that some semantics are more abstract than (lose information with respect to) others as they compute approximations in less rich domains of properties. It provides tools to prove soundness: any property proved in an abstract semantic is still true in the concrete one. Finally, it provides tools to design and combine abstractions.

The aim of the article is to give a short overview of the theory underlying tools such as Astrée and AstréeA. Section 2 focuses on sequential programs and presents the classic construction of an effective analyzer by abstraction of the most concrete semantics of a program: its execution traces. Section 3 considers multithreaded programs and explains how, borrowing ideas from rely/guarantee proof methods [14], an efficient, thread-modular analysis can be constructed. Section 4 briefly presents the Astrée and AstréeA analyzers, and Sec. 5 concludes.

The modest contribution of this article is the formulation, in Sec. 3.4, of Jones' rely/guarantee method [14] in abstract interpretation form, as an abstraction of the execution traces of multithreaded programs by decomposition into intra-thread invariants and inter-thread interferences. This extends previous work in the 80's [7] that formalized earlier Owicki–Gries–Lamport methods [17,15] as abstract interpretation. It also extends our recent previous work [16], that only considered coarse abstract interferences, by exhibiting an intermediate layer of abstraction from which it can be recovered.

2 Abstractions for Sequential Programs

As a short introduction to abstract interpretation concepts, we review the formal construction of a static analyzer for *sequential* programs by abstraction of its trace semantics — see also [2, § II] for an extended introduction.

```
(1) i := 2;
(2) n := input_int();
(3) while i < n do
(4)     if input_bool() then i := i + 1;
(5) done;
(6) assert i >= 2;
```

(a)

at (1): $i = 0 \wedge n = 0$
at (2): $i = 2 \wedge n = 0$
at (3): $2 \leq i \leq \max(2, n)$
at (4): $2 \leq i \leq n - 1 \wedge n \geq 3$
at (5): $2 \leq i \leq n \wedge n \geq 3$
at (6): $i = \max(2, n)$

(b)

Fig. 1. A simple sequential program (a), and invariant assertions (b).

2.1 Transition Systems

The semantics of a program is defined classically [5] in a very general way in *small step operational form*, as a *transition system*: (Σ, τ, I), where Σ is a set of states, $I \subseteq \Sigma$ is the subset of initial states, and $\tau \subseteq \Sigma \times \Sigma$ is a transition relation. For sequential programs, the state set is defined as $\Sigma \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{M}$, where each state $\sigma = (\ell, m)$ has a control part $\ell \in \mathcal{L}$ denoting the current program point, and a memory part $m \in \mathcal{M} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{V}$ mapping program variables $V \in \mathcal{V}$ to values $v \in \mathbb{V}$. A transition models an atomic execution step, such as executing a machine-code instruction in a program. We denote by $(\ell, m) \to (\ell', m')$ the fact that $((\ell, m), (\ell', m')) \in \tau$, i.e, the program can reach the state (ℓ', m') from the state (ℓ, m) after one execution step. The transition system derives mechanically from the program code itself, e.g., from the sequence of binary instructions or, more conveniently, by induction on the syntax tree of the source code.

Example 1. Consider the simple programming language syntax:

$$ {}^{\ell}P^{\ell'} \ ::= \ {}^{\ell}x := e^{\ell'} \mid {}^{\ell}\texttt{if}\ e\ \texttt{then}\ {}^{\ell''}P^{\ell'} \mid {}^{\ell}\texttt{while}\ e\ \texttt{do}\ {}^{\ell''}P^{\ell'} \mid {}^{\ell}P;^{\ell''}P'^{\ell'} $$

where ${}^{\ell}$ and e denote respectively syntactic program points and expressions. The transition system $\tau[{}^{\ell}P^{\ell'}]$ is derived by induction on the syntax of P as follows:

$$
\begin{array}{lcl}
\tau[{}^{\ell}x := e^{\ell'}] & \stackrel{\text{def}}{=} & \{\, (\ell, m) \to (\ell', m[x \mapsto v]) \mid m \in \mathcal{M},\ e \overset{m}{\rightsquigarrow} v \,\} \\
\tau[{}^{\ell}\texttt{if}\ e\ \texttt{then}\ {}^{\ell''}P^{\ell'}] & \stackrel{\text{def}}{=} & \{\, (\ell, m) \to (\ell'', m) \mid m \in \mathcal{M},\ e \overset{m}{\rightsquigarrow} \texttt{true} \,\} \cup \tau[{}^{\ell''}P^{\ell'}] \cup \\
 & & \{\, (\ell, m) \to (\ell', m) \mid m \in \mathcal{M},\ e \overset{m}{\rightsquigarrow} \texttt{false} \,\} \\
\tau[{}^{\ell}\texttt{while}\ e\ \texttt{do}\ {}^{\ell''}P^{\ell'}] & \stackrel{\text{def}}{=} & \{\, (\ell, m) \to (\ell'', m) \mid m \in \mathcal{M},\ e \overset{m}{\rightsquigarrow} \texttt{true} \,\} \cup \tau[{}^{\ell''}P^{\ell}] \cup \\
 & & \{\, (\ell, m) \to (\ell', m) \mid m \in \mathcal{M},\ e \overset{m}{\rightsquigarrow} \texttt{false} \,\} \\
\tau[{}^{\ell}P;^{\ell''}P'^{\ell'}] & \stackrel{\text{def}}{=} & \tau[{}^{\ell}P^{\ell''}] \cup \tau[{}^{\ell''}P'^{\ell'}]
\end{array}
$$

where $m[x \mapsto v]$ denotes the function mapping x to v and $y \neq x$ to $m(y)$, and $e \overset{m}{\rightsquigarrow} v$ states that e can evaluate to the value v in the memory m. ∎

2.2 Trace Semantics

We are not interested in the program itself, but in the properties of its executions. Formally, an execution *trace* is a finite sequence of states $\sigma_0 \to \sigma_1 \to \cdots \to \sigma_n$

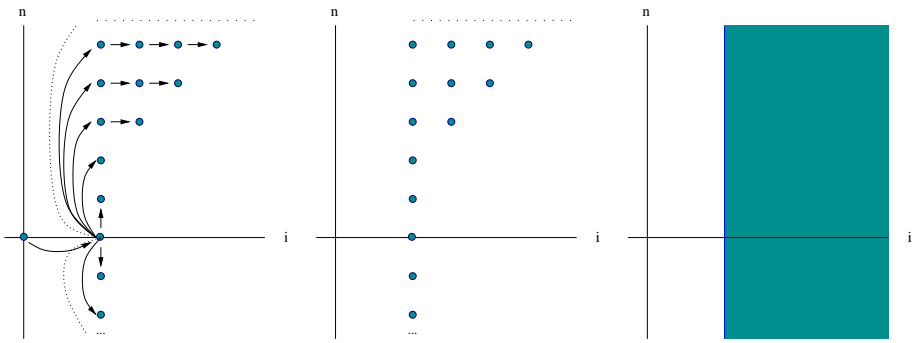

Fig. 2. Semantics of the program in Fig. 1 at various levels of abstractions: (a) traces, and (b) reachable states and (c) intervals at program point (3).

such that $\sigma_0 \in I$ and $\forall i$, $(\sigma_i, \sigma_{i+1}) \in \tau$. The semantics we want to observe, which is called a *collecting semantics*, is thus defined in our case as the set $T \in \mathcal{P}(\Sigma^*)$ of traces spawned by the program. It can be expressed [6] as the least fixpoint of a continuous operator in the complete lattice of sets of traces:

$$\begin{array}{l} T = \mathrm{lfp}\ F \text{ where} \\ F(X) \stackrel{\text{def}}{=} I \cup \{\, \sigma_0 \to \cdots \to \sigma_{i+1} \mid \sigma_0 \to \cdots \to \sigma_i \in X \wedge \sigma_i \to \sigma_{i+1} \,\} \end{array} \tag{1}$$

i.e., T is the smallest set containing traces reduced to an initial state and closed under the extension of traces by an additional execution step. Note that we are observing *partial* execution traces; informally, we allow executions to be interrupted prematurely; formally, T is closed by prefix (it also includes the finite prefixes of non-terminating executions). This is sufficient if we are interested in *safety* properties, i.e., properties of the form "no bad state is ever encountered," which is the case here — more general trace semantics, able to also reason about liveness properties, are discussed for instance in [4]. A classic result [6] is that T can be restated as the limit of an iteration sequence: $T = \bigcup_{n\in\mathbb{N}} F^n(\emptyset)$. It becomes clear then that computing T is equivalent to testing the program on all possible executions (albeit with an unusual exhaustive breadth-first strategy) and that it is not adapted to the effective and efficient verification of programs: when the program has unbounded or infinite executions, T is infinite.

Example 2. The simple program in Fig. 1.a increments `i` in a loop, until it reaches a user-specified value `n`. Figure 2.a presents its trace semantics starting in the state set $I = \{\, (1, [\mathtt{m} \mapsto 0, \mathtt{i} \mapsto 0]) \,\}$. The program has infinite executions (e.g., if `i` is never incremented). ∎

2.3 State Semantics

We observe that, in order to prove safety properties, it is not necessary to compute T exactly. It is sufficient to collect the set $S \in \mathcal{P}(\Sigma)$ of program states

encountered, abstracting away any information available in T on the ordering of states. We use an *abstraction function* $\alpha^{state} : \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma)$:

$$S = \alpha^{state}(T) \text{ where } \alpha^{state}(X) \stackrel{\text{def}}{=} \{\, \sigma \mid \exists \sigma_0 \to \cdots \to \sigma_n \in X, i \in [0;n],\, \sigma = \sigma_i \,\} \tag{2}$$

An important result is that, as T, S can be computed directly as a fixpoint:

$$S = \text{lfp}\ G \text{ where } G(X) \stackrel{\text{def}}{=} I \cup \{\, \sigma' \mid \exists \sigma \in X,\, \sigma \to \sigma' \,\}$$

or, equivalently, as an iteration sequence $S = \bigcup_{n \in \mathbb{N}} G^n(\emptyset)$, which naturally expresses that S is the set of states reachable from I after zero, one, or more transitions. The similarity in fixpoint characterisation of S and T is not a coincidence, but a general result of abstract interpretation (although, in most cases, the abstract fixpoint only over-approximates the concrete one: $\text{lfp}\ G \supseteq \alpha^{state}(\text{lfp}\ F)$, see [4]). Dually, given a set of states S, one can construct the set of traces abstracted by S using a *concretization* function $\gamma^{state} \stackrel{\text{def}}{=} \lambda S \,.\, \{\, \sigma_0 \to \cdots \to \sigma_n \mid n \in \mathbb{N} \wedge \forall i \in [0;n],\, \sigma_i \in S \,\}$. The pair $(\alpha^{state}, \gamma^{state})$ forms a Galois connection.[1] A classic result [5] states that the *best abstraction* of F can be defined as $\alpha^{state} \circ F \circ \gamma^{state}$, which in our case turns out to be exactly G. When the set of states is finite (e.g., when the memory is bounded), S can always be computed by iteration in finite time, even if T cannot. Obviously, S may be extremely large and require many iterations to stabilize, hence computing S exactly is not a practical solution; we will need further abstractions.

2.4 Program Proofs and Inductive Invariants

There is a deep connection [5] between the state semantics and the program logic of Floyd–Hoare [13] used to prove partial correctness. If we interpret each logic assertion A_ℓ at program point ℓ as the set of memory states $[\![A_\ell]\!] \subseteq \mathcal{M}$ satisfying it, and note $M \stackrel{\text{def}}{=} \{\, (\ell, m) \mid m \in [\![A_\ell]\!] \,\}$, then $(A_\ell)_{\ell \in \mathcal{L}}$ is a valid (i.e., inductive) invariant assertion if and only if $G(M) \subseteq M$. Moreover, the best inductive invariant assertion stems from lfp G, i.e., it is S. While, in proof methods, the inductive invariants must be devised by an oracle (the user), abstract interpretation opens the way to automatic invariant inference by providing a *constructive* view on invariants (through iteration) and allowing further abstractions.

Example 3. The optimal invariant assertions of the program in Fig. 1.a appear in Fig. 1.b, and Fig. 2.b presents graphically its state abstraction at point (3). ∎

2.5 Memory Abstractions

In order gain in efficiency on bounded state spaces and handle unbounded ones, we need to abstract further. As many errors (such as overflows and divisions by zero) are caused by invalid arguments of operators, a natural idea is to observe

[1] I.e., $\forall X \in \mathcal{P}(\Sigma^*), \forall Y \in \mathcal{P}(\Sigma),\ \alpha^{state}(X) \subseteq Y \iff X \subseteq \gamma^{state}(Y)$.

only the set of values each variable can take at each program point. Instead of concrete state sets in $\mathcal{P}(\Sigma) \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{L} \times (\mathcal{V} \to \mathbb{V}))$, we manipulate abstract states in $\Sigma_C \stackrel{\text{def}}{=} (\mathcal{L} \times \mathcal{V}) \to \mathcal{P}(\mathbb{V})$. The concrete and abstract domains are linked through the following Galois connection (so-called Cartesian Galois connection):

$$\begin{array}{lcl} \alpha^{cart}(X) & \stackrel{\text{def}}{=} & \lambda(\ell, V) \,.\, \{\, v \,|\, \exists (\ell, m) \in X,\, m(V) = v \,\} \\ \gamma^{cart}(X^\sharp) & \stackrel{\text{def}}{=} & \{\, (\ell, m) \,|\, \forall V,\, m(V) \in X^\sharp(\ell, V) \,\} \end{array} \tag{3}$$

Assuming that variables are integers or reals, a further abstraction consists in maintaining, for each variable, its bounds instead of all its possible values. We compose the connection $(\alpha^{cart}, \gamma^{cart})$ with (the element-wise lifting of) the following connection $(\alpha^{itv}, \gamma^{itv})$ between $\mathcal{P}(\mathbb{R})$ and $(\mathbb{R} \cup \{-\infty\}) \times (\mathbb{R} \cup \{+\infty\})$:

$$\begin{array}{lcl} \alpha^{itv}(X) & \stackrel{\text{def}}{=} & [\min X; \max X] \\ \gamma^{itv}([a;b]) & \stackrel{\text{def}}{=} & \{\, x \in \mathbb{R} \,|\, a \leq x \leq b \,\} \end{array} \tag{4}$$

yielding the interval abstract domain [5].

Another example of memory domain is the linear inequality domain [10] that abstracts sets of points in $\mathcal{P}(\mathcal{V} \to \mathbb{R})$ into convex, closed polyhedra. One abstracts $\mathcal{P}(\Sigma) \simeq \mathcal{L} \to \mathcal{P}(\mathcal{V} \to \mathbb{V})$ by associating a polyhedron to each program point, which permits the inference of linear relations between variables. The polyhedra domain is thus a relational domain, unlike the domains deriving from the Cartesian abstraction (such as intervals).

Example 4. Figure 2.c presents the interval abstraction of the program in Fig. 1.a at point (3). The relational information that $i \leq n$ when $n \geq 2$ is lost. ∎

Following the same method that derived the state semantics from the trace one, we can derive an interval semantics from the state one: it is expressed as a fixpoint lfp $G^\sharp$ of an interval abstraction $G^\sharp$ of G. While it is possible to define an optimal $G^\sharp$ as $\alpha \circ G \circ \gamma$ — where (α, γ) combines $(\alpha^{itv}, \gamma^{itv})$ and $(\alpha^{cart}, \gamma^{cart})$ — this is a complex process when G is large, and it must be performed for each G, i.e., for each program. To construct a general analyzer, we use the fact that F, and so, $G = \alpha^{state} \circ F \circ \gamma^{state}$, are built by combining operators for atomic language constructs, as in Ex. 1. It is thus possible to derive $G^\sharp$ as a combination of a small fixed set of abstract operators. More precisely, we only have to design abstraction versions of assignments, tests, and set union $\cup$. Generally, the combination of optimal operators is not optimal, and so, the resulting $G^\sharp$ is not optimal — e.g., $(\alpha \circ G_1 \circ G_2 \circ \gamma)(X) \subsetneq ((\alpha \circ G_1 \circ \gamma) \circ (\alpha \circ G_2 \circ \gamma))(X)$. Additionally, due to efficiency and practicality concerns, the base abstract operators may be chosen non-optimal to begin with. Achieving a sound analysis, i.e., $\gamma(\text{lfp } G^\sharp) \supseteq \text{lfp } G$, only requires sound abstracts operators, i.e., $\forall X^\sharp, (\gamma \circ G^\sharp)(X^\sharp) \supseteq (G \circ \gamma)(X^\sharp)$. The fact that $G^\sharp \neq \alpha \circ G \circ \gamma$, and even the existence of an α function, while sometimes desirable, is by no mean required — for instance, the polyhedra domain [10] does not feature an abstraction function α as some sets, such as discs, have no best polyhedral abstraction.

Example 5. The interval abstraction $[\![{}^{\ell}x := y + z^{\ell'}]\!]^\sharp$ of a simple addition maps the interval environment $[x \mapsto [l_x; h_x]; y \mapsto [l_y; h_y]; z \mapsto [l_z; h_z]]$ at point ℓ to $[x \mapsto [l_y + l_z; h_y + h_z]; y \mapsto [l_y; h_y]; z \mapsto [l_z; h_z]]$ at point ℓ', which is optimal. ∎

Example 6. The optimal abstraction $[\![{}^{\ell}y := -z; {}^{\ell'}x := y + z^{\ell''}]\!]^\sharp$ would map x to $[0; 0]$. However, the combination $[\![{}^{\ell}y := -z^{\ell'}]\!]^\sharp \circ [\![{}^{\ell'}x := y + z^{\ell''}]\!]^\sharp$ maps x to $[l_z - h_z; h_z - l_z]$ instead, which is coarser when $l_z < h_z$. ∎

Example 7. We can design a sound non-optimal fall-back abstraction for arbitrary assignments ${}^{\ell}x := e^{\ell'}$ by simply mapping x to $[-\infty; +\infty]$. ∎

We now know how to define systematically a sound abstract operator $G^\sharp$ which can be efficiently computed. Nevertheless, the computation of lfp $G^\sharp$ by naive iteration may not converge fast enough (or at all). Hence, abstract domains are generally equipped with a convergence acceleration binary operator $\triangledown$, called *widening* and introduced in [5]. Widenings extrapolate between two successive iterates and ensure that any sequence $X^\sharp_{i+1} \overset{\text{def}}{=} X^\sharp_i \triangledown G^\sharp(X^\sharp_i)$ converges in finite time, possibly towards a coarse abstraction of the actual fixpoint.

Example 8. The classic interval widening [5] sets unstable bounds to infinity:

$$[l_1; h_1] \triangledown [l_2; h_2] \overset{\text{def}}{=} \left[\begin{cases} -\infty & \text{if } l_2 < l_1 \\ l_1 & \text{otherwise} \end{cases} ; \begin{cases} +\infty & \text{if } h_2 > h_1 \\ h_1 & \text{otherwise} \end{cases} \right] \tag{5}$$

When analyzing Fig. 1, the iterations with widening at (3) give the following intervals for `i`: $i^\sharp_0 = [2; 2]$ and $i^\sharp_1 = [2; 2] \triangledown [2; 3] = [2; +\infty]$, which is stable. ∎

To balance the accumulation of imprecision caused by widenings and combining separately abstracted operators, it is often necessary to reason on an abstract domain strictly more expressive than the properties we want to prove — e.g., the polyhedra domain may be necessary to infer variable bounds that the interval domain cannot infer, although it can represent them, such as $x = 0$ in Ex. 6.

2.6 Further Considerations

We end this introduction with some pointers to additional material. Firstly, there exists a large library of abstract domains, in particular numeric ones, and associated operators with various precision versus cost trade-offs — [2] describes a few of them. An actual analyzer will use a combination, such as a reduced product, of many domains [9]. Secondly, we have assumed for simplicity that the set of variables $\mathcal{V}$ is finite, but abstract domains able to handle an unbounded memory exist; this is necessary when dynamic memory allocation is used. They often combine abstractions of the memory shape and of the contents of numeric fields, as in [19]. Likewise, an efficient handling of procedures requires abstracting the control state $\mathcal{L}$ (which can be large, or even unbounded in the case of recursivity). Call-string methods [18] are an instance of such abstractions. Finally, alternate iteration techniques exist [3], as well as methods to decompose the global fixpoint into several local ones to achieve a modular analysis [8].

3 Abstractions for Parallel Programs

We now consider multithreaded programs in a shared memory. We show how, starting from a classic concrete semantics based on interleaved executions [11] and applying abstract interpretation, we can construct an effective thread-modular static analysis which is similar to rely/guarantee proof methods [14].

3.1 Labelled Transition Systems

We assume a *finite* set $\mathcal{T}$ of threads. Each thread $t \in \mathcal{T}$ has its own control space $\mathcal{L}_t$ and transition relation $\tau_t \subseteq \Sigma_t \times \Sigma_t$, where $\Sigma_t \stackrel{\text{def}}{=} \mathcal{L}_t \times \mathcal{M}$. The state space and transitions for the whole program are derived from those of individual threads as follows. Program states live in $\Sigma \stackrel{\text{def}}{=} (\prod_{t \in \mathcal{T}} \{ t \} \to \mathcal{L}_t) \times \mathcal{M}$, i.e., each thread has its own control point, but the memory is shared. The semantics of the program is defined as a *labelled* transition system (Σ, τ, I), where the transition relation $\tau \subseteq \Sigma \times \mathcal{T} \times \Sigma$ is defined as:[2] $((\bar{\ell}, m), t, (\bar{\ell}', m')) \in \tau$ if $((\bar{\ell}[t], m), (\bar{\ell}'[t], m)) \in \tau_t$ and $\forall t' \neq t,\ \bar{\ell}[t'] = \bar{\ell}'[t']$. It states that an execution step of the program is an execution step of some thread and updates only that thread's control location. We denote such a step as $(\bar{\ell}, m) \xrightarrow{t} (\bar{\ell}', m')$. Labels $t \in \mathcal{T}$ are used to explicitly remember which thread caused each transition.

3.2 Interleaved Trace Semantics

As for sequential programs, we consider finite prefixes of executions and ignore liveness properties — in the context of parallel programs, liveness properties are related to fairness conditions, which is a very complex issue not discussed here; see [11] for a complete treatment. The trace semantics T is defined as in (1):

$$\begin{array}{l} T = \text{lfp } F \text{ where} \\ F(X) \stackrel{\text{def}}{=} I \cup \{ \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_i} \sigma_{i+1} \mid \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{i-1}} \sigma_i \in X \wedge \sigma_i \xrightarrow{t_i} \sigma_{i+1} \} \end{array} \tag{6}$$

which thus corresponds to executions consisting of arbitrary *interleavings* of transitions from any threads.

Given the similarity with the sequential trace semantics, a natural idea to analyze a parallel program is to forget about labels and apply the state and memory abstractions (Secs. 2.3, 2.5). The problem with this method is the huge number of control states to consider: $\prod_{t \in \mathcal{T}} |\mathcal{L}_t|$ instead of $|\mathcal{L}|$. Moreover, τ is much larger that $\bigcup_{t \in \mathcal{T}} \tau_t$ as a single transition in τ_t is repeated for each possible control state of other threads $\prod_{t' \neq t} \mathcal{L}_{t'}$. As a consequence, constructing an abstraction $G^\sharp$ of F from abstractions of atomic instructions requires a combination of many more such functions than for sequential programs. This makes even coarse Cartesian interval abstractions impracticable.

[2] The $\bar{x}$ notations indicates that $\bar{x}$ is a finite vector. $\bar{x}[i]$ is its value at index i. We also use vector functions $\bar{f}$. The i-th component of y's image $\bar{f}(y)$ is denoted $\bar{f}(y)[i]$. Finally, $\bar{\emptyset}$, $\bar{\cup}$, and $\bar{\subseteq}$ denote respectively $\emptyset$, $\cup$, $\subseteq$ extended element-wise to vectors.

```
(1) while true do
(2)     if x < y then
(3)         x := x + 1;
(4) done;
```

```
(5) while true do
(6)     if y < 10 then
(7)         y := y + 1;
(8) done;
```

(a)

at (1),(2),(4): $\mathtt{x} \leq \mathtt{y}$
at (3): $\mathtt{x} < \mathtt{y}$
at (5),(6),(8): $\mathtt{y} \leq 10$
at (7): $\mathtt{y} < 10$

(b)

Fig. 3. A simple multithreaded program (a), and its invariant assertions (b).

```
(1) while true do
(2)     x := x + 1;
(3)     x := x - 1;
(4) done;
```

```
(5) while true do
(6)     x := x - 2;
(7)     x := x + 2;
(8) done;
```

(a)

at (1),(2),(4): $\mathtt{x} \in \{-2, 0\}$
at (3): $\mathtt{x} \in \{-1, 1\}$
at (5),(6),(8): $\mathtt{x} \in \{0, 1\}$
at (7): $\mathtt{x} \in \{-2, -1\}$

(b)

Fig. 4. A program requiring flow-sensitivity (a), and its invariant assertions (b).

3.3 Rely/Guarantee

Proof methods for parallel programs, such as Owicki–Gries–Lamport [17,15] and rely/guarantee [14] solve this issue by attaching invariants to thread control points in $\bigcup_{t \in \mathcal{T}} \mathcal{L}_t$ instead of combinations of thread points in $\prod_{t \in \mathcal{T}} \mathcal{L}_t$. The price to pay is a refined notion of invariance: the properties attached to a thread t must also be proved stable by the effect of the other threads $t' \neq t$. In rely/guarantee methods, this effect is explicitly provided as extra annotations in t that are assumed to hold (*relied* on) when checking the invariants for t and proved correct (*guaranteed*) when checking the other threads $t' \neq t$. It then becomes possible to check each thread in a modular way, i.e., without looking at the code of the other threads, relying on the annotations instead.

Example 9. Figure 3 presents a program with two threads: the first one increments `x` up to `y` and the second one increments `y` up to 10. For conciseness, we exemplify the rely/guarantee conditions at a single program point. Consider the problem of proving that $\mathtt{x} < \mathtt{y}$ holds at (3), just after the test. We need to prove that the assertion is stable by the action of the second thread. It is sufficient to *rely* on the fact that the second thread does not modify `x` and only increments `y`. This property is, in turn, *guaranteed* by analyzing the second thread, relying only on the fact that the first thread does not modify `y`. ∎

3.4 Interference Semantics

We now rephrase the idea of rely/guarantee methods as an abstract interpretation of the interleaved trace semantics T (6). We decompose the trace semantics using two complementary abstractions: an abstraction into *thread-local invariants* (which is inspired from the formalization in [7] of Owicki–Gries–Lamport methods [17,15] as an abstract interpretation) and an abstraction into *inter-thread interferences* (which is new). Firstly, the local invariant $\bar{S}[t]$ of a thread t

is defined by projecting, with the bijection π_t, the reachable states $\alpha^{state}(T)$ (2) on t's control state $\mathcal{L}_t$:

$$\begin{array}{l}\bar{S}[t] \stackrel{\text{def}}{=} (\Pi_t \circ \alpha^{state})(T) \text{ where} \\ \Pi_t(X) \stackrel{\text{def}}{=} \{\, \pi_t(x) \mid x \in X \,\} \text{ and } \pi_t(\bar{\ell}, m) \stackrel{\text{def}}{=} (\bar{\ell}[t], m[\forall t' \neq t,\, p_{t'} \mapsto \bar{\ell}[t']])\end{array} \tag{7}$$

We keep the control state $\bar{\ell}[t']$ of other threads $t' \neq t$ encoded as extra variables $p_{t'}$ in the memory (called *auxiliary variables* in Owicki–Gries [17]). It is possible to remove these variables and keep only t's control information to get:

$$\alpha_t^{ctrl}(\bar{S}[t]) \text{ where } \alpha_t^{ctrl}(X) \stackrel{\text{def}}{=} \{\, (\ell, m) \mid \exists \bar{\ell},\, \pi_t(\bar{\ell}, m) \in X \wedge \bar{\ell}[t] = \ell \,\} \tag{8}$$

but α_t^{ctrl} is known to be an incomplete abstraction: some invariance properties on $\bar{S}[t]$ cannot be proved using $\alpha_t^{ctrl}(\bar{S}[t])$ only (see Ex. 12). Secondly, the interference $\bar{A}[t]$ of a thread t is defined as the possible actions it has on other threads, as observed in the interleaved trace semantics T:

$$\bar{A}[t] \stackrel{\text{def}}{=} \{\, (\sigma_i, \sigma_{i+1}) \mid \exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in T,\, t_i = t \,\} \tag{9}$$

Interferences are actually subsets of the transition relation *restricted* to those transitions appearing in actual traces starting in I. We can then express $\bar{S}[t]$ in fixpoint form as a function of $\bar{A}$ and the transitions caused by thread t:

$$\begin{array}{l}\bar{S}[t] = \text{lfp } G_t(\bar{A}) \text{ where} \\ G_t(\bar{Y})(X) \stackrel{\text{def}}{=} \Pi_t(I) \cup \{\, \pi_t(\sigma') \mid \exists \pi_t(\sigma) \in X,\, \sigma \xrightarrow{t} \sigma' \vee \exists t' \neq t,\, (\sigma, \sigma') \in \bar{Y}[t'] \,\}\end{array} \tag{10}$$

i.e., to reach a new program state from a known one, we either execute a step from thread t (in $\xrightarrow{t}$) or a step from another thread (in $\bar{A}[t']$). Moreover, we can express directly $\bar{A}$ as a function of $\bar{S}$:

$$\bar{A}[t] = \bar{B}(\bar{S})[t] \text{ where } \bar{B}(\bar{Y})[t] \stackrel{\text{def}}{=} \{\, (\sigma, \sigma') \mid \pi_t(\sigma) \in \bar{Y}[t] \wedge \sigma \xrightarrow{t} \sigma' \,\} \tag{11}$$

which expresses that $\bar{A}[t]$ corresponds to the transitions in τ_t starting from a reachable state. This yields the following *nested fixpoint* characterisation of $\bar{S}$:

$$\bar{S} = \text{lfp } \bar{H} \text{ where } \bar{H}(\bar{X})[t] \stackrel{\text{def}}{=} \text{lfp } G_t(\bar{B}(\bar{X})) \tag{12}$$

which can be computed in iterative form as: $\bar{S} = \bigcup_{n\in\mathbb{N}} \bar{H}^n(\bar{\emptyset})$ while $\bar{H}(\bar{X})[t] = \bigcup_{n\in\mathbb{N}} (G_t(\bar{B}(\bar{X})))^n(\emptyset)$. The benefit of this characterization is that the computation of $\bar{H}(\bar{X})[t]$ only depends on its argument $\bar{X}$ and the transition relation of the thread t, not on the transition relations of the other threads: the inner fixpoint is thus similar to the analysis of a sequential process. However, as analyzing a thread in a given set of interferences $\bar{B}(\bar{X})$ gathers a new, larger set of interferences $\bar{B}(\bar{H}(\bar{X}))$, the analysis must be performed again with this enriched set until it becomes stable. This is the role of the outer fixpoint.

3.5 Memory and Interference Abstractions

Our interference semantic $\bar{S}$ is very concrete (the state semantics $\alpha^{state}(T)$ can be recovered fully from it), and too large to be computed directly. An effective and efficient analyzer can be designed by applying abstractions independently to local invariants and to interferences, while keeping the nested fixpoint form of (12). Firstly, any memory abstraction, such as intervals (4) or polyhedra [10], can be applied directly to each local invariant $\bar{S}[t]$. As the number of variables is often a critical parameter in the efficiency of abstract domains, a faster but coarser analysis can be achieved by removing the auxiliary variables from $\bar{S}[t]$ with α^{ctrl}_t (8) prior to memory abstraction. Secondly, we abstract interferences $\bar{A}[t]$. For instance, it is possible to forget the control state of all threads using a *flow-insensitive* abstraction α^{flow}:

$$\alpha^{flow}(X) \stackrel{\text{def}}{=} \{\, (m, m') \mid \exists \bar{\ell}, \bar{\ell}',\ ((\bar{\ell}, m), (\bar{\ell}', m')) \in X \,\}$$

Intuitively, this means that, when analyzing a thread t, we consider that any instruction from any thread $t' \neq t$ may be executed at any point of thread t. Then, by noting that $\mathcal{P}((\mathcal{V} \to \mathbb{V}) \times (\mathcal{V} \to \mathbb{V})) \simeq \mathcal{P}((\{\,1, 2\,\} \times \mathcal{V}) \to \mathbb{V})$, we can see a set of pairs of maps on $\mathcal{V}$ as a set of maps on $\{\,1, 2\,\} \times \mathcal{V}$, with twice as many variables; hence, we can apply any memory abstraction to $\alpha^{flow}(X)$. Another solution consists in abstracting the relation $\alpha^{flow}(X)$ with its image, using the following abstraction α^{img}, before applying a memory abstraction:

$$\alpha^{img}(X) \stackrel{\text{def}}{=} \{\, m' \mid \exists m,\ (m, m') \in X \,\}$$

which is coarser but more efficient as it avoids doubling the number of variables. By combining α^{flow}, α^{img}, and α^{cart} (3), we abstract thread interferences in a non-relational and flow-insensitive way, as the set of values each thread can store into each variable during the thread execution. Further abstractions, such as intervals α^{itv} (4), can be applied. This yields an efficient analysis: the abstract interference is simply a global map from $\mathcal{T} \times \mathcal{V}$ to intervals, and we can build an abstraction $\bar{H}^\sharp(\bar{X}^\sharp)[t]$ of $\bar{H}(\bar{X})[t]$ by combining abstractions for each instruction of thread t (as Ex. 5 but slight modified to take interferences into account).

Example 10. The interval abstraction $[\![\, {}^{\ell} x := y + 1^{\ell'} \,]\!]^\sharp$ in the environment $[x \mapsto [l_x; h_x]; y \mapsto [l_y; h_y]]$ at ℓ and the global interference map $[x \mapsto [l'_x; h'_x]; y \mapsto [l'_y; h'_y]]$ gives, at ℓ', the interval $x \mapsto [\min(l'_x, l_x + \min(l_y, l'_y) + 1); \max(h'_x, h_x + \max(h_y, h'_y) + 1)]$. ∎

Example 11. We analyze Fig. 3 using intervals, by applying α^{ctrl}_t, α^{cart}, and α^{itv} to local invariants, and α^{flow}, α^{img}, α^{cart}, and α^{itv} to interferences. We find that, at (3), $\mathtt{x} \in [0; 9]$. The information that $\mathtt{x} < \mathtt{y}$ is lost. The second thread produces the abstract interferences $[\mathtt{x} \mapsto \emptyset, \mathtt{y} \mapsto [0; 10]]$, i.e., we infer that it cannot modify $\mathtt{x}$ and can only put values from $[0; 10]$ into $\mathtt{y}$. We lose the information that $\mathtt{y}$ can only increase. ∎

Example 12. Consider the program in Fig. 4. When analyzing the first thread, we note that, at (2), `x` can be 0 or -2, depending on whether the second thread is at (6) or (7). Moreover, to prove that `x` stays in $\{-2, 0\}$ at (2), it is necessary to infer that the interference from the second thread can only increment `x` when it is at (7), and decrement it when it is at (6). A flow-insensitive abstraction using α_t^{ctrl} and α^{flow} would instead infer that `x` can be incremented arbitrarily and decremented arbitrarily. It would not allow us to prove that `x` is bounded. ∎

3.6 Further Considerations

A practical consequence of (12) is that a thread-modular static analyzer can be designed by slightly modifying a sequential analyzer: all we need to do is keep track of (abstract) interferences as well as reachable states, and add an external fixpoint iteration to re-analyze each thread until the inferred interferences stabilize. As for memory abstractions, there are many choices in how to abstract interferences, with various cost versus precision trade-offs and various levels of expressiveness. Although we presented only flow-insensitive non-relational abstract interferences, we can envision abstractions keeping (at least partially) flow and relational information. This is in contrast to our earlier formalization [16] which only allowed flow-insensitive non-relational interferences, and can now be seen a special case of the framework presented here. In the presence of mutual exclusion primitives and scheduling policies that restrict the interleaving of threads (such as thread priorities), a model of the scheduler can be incorporated into the semantics to refine the notion of interference. For instance [16] handles locks by partitioning flow-insensitive, non-relational abstract interferences with respect to the set of locks each thread holds. Another remark is that the interleaved trace semantics (6) is not realistic in the context of modern multicore architectures and compilers; non-coherent caches and optimizations can expose extra behaviors. However, we proved in [16] that, provided that a flow-insensitive abstraction of interferences is used, the abstract semantics is sound even in the presence of weakly consistent memory models, so that the results of the static analysis can still be trusted in realistic settings. Finally, we did not discuss the case of an unbounded number of threads, which requires further abstraction.

4 Applications

We now briefly describe two static analysis tools that were designed at the École normale supérieure using abstract interpretation principles.

4.1 Astrée

Astrée is a static analyzer that checks for run-time errors in synchronous (hence sequential) embedded critical C programs. It covers the full C language, including pointers and floats, but excluding dynamic memory allocation and recursivity (forbidden in the realm of critical software). It checks for arithmetic

overflows, and invalid arithmetic and memory operations. It is sound, and so, never misses any existing error. Due to over-approximations, it can signal false alarms. Although Astrée can analyze many kinds of codes, it is specialized to control-command avionic software, on which it aims at efficiency and certification, i.e., zero false alarm. It was able to prove quickly (2h to 53h) the absence of run-time error in large (100 K to 1M lines) industrial avionic codes. This result could be achieved using a specialization process: starting from a fast and coarse analyzer based on intervals, we added manually more powerful abstract domains as they were needed. Some were borrowed from the large library of existing abstractions, and others were developed specifically for the avionic application domain (such as an abstraction of digital filters [12]). Astrée is a mature tool: it is industrialized and made commercially available by AbsInt [1]. More details can be found in [2].

4.2 AstréeA

AstréeA aims at checking for run-time errors in multithreaded embedded critical C programs. It is based on the Astrée code-base and inherits all of its abstractions. Additionally, it implements the interference fixpoint analysis described in Sec. 3. Interferences are abstracted in a flow-insensitive and non-relational way, which was sufficient to give encouraging experimental results. On our target application, a 1.6 Mlines industrial avionic software with 15 threads, AstréeA reports around 1 300 alarms in 50h. More details can be found in [16,2]. AstréeA is a research prototype in development; we are currently improving it with more powerful memory and interference abstractions.

5 Conclusion

We have provided in this article a short glimpse of abstract interpretation theory and its application to the systematic construction of static analyzers for sequential and parallel programs. Two key reasons make abstract interpretation-based constructions attractive. Firstly, its ability to relate in a formal way the output of a static analyzer all the way down to the precise dynamic behavior of the input program (its execution traces), both in terms of soundness proof and information loss. Secondly, the ability to decompose the design of an analyzer as the combination of independent, reusable abstractions, which allows the modular implementation of analyzers (e.g., abstract domains designed for Astrée could be reused in AstréeA). Future work includes the design of new abstractions to improve the AstréeA analyzer prototype and reduce the number of alarms on selected embedded multithreaded C programs. In particular, the derivation of thread-modular abstract semantics for parallel programs through the use of concrete interferences we introduced here opens the way to the design of flow-sensitive and relational interference abstractions, which was not possible in the earlier framework underlying AstréeA.

Acknowledgments. We thank the anonymous referees on a earlier version of [16] for pointing out the link between the analysis performed by AstréeA and rely/guarantee methods, which was not apparent to us before. We also thank Patrick Cousot for urging us to provide a sequence of explicit abstraction functions detailing precisely which information is lost at each step of our construction, instead of providing a monolithic soundness proof as we did in [16].

References

1. AbsInt, Angewandte Informatik. Astrée run-time error analyzer. http://www.absint.com/astree.
2. J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace*, number 2010-3385 in AIAA, pages 1–38. AIAA (American Institute of Aeronautics and Astronautics), Apr. 2010.
3. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93)*, volume 735 of *LNCS*, pages 128–141. Springer, June 1993.
4. P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1–2):47–103, 2002.
5. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM, Jan. 1977.
6. P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.
7. P. Cousot and R. Cousot. Invariance proof methods and analysis techniques for parallel programs. In *Automatic Program Construction Techniques*, chapter 12, pages 243–271. Macmillan, New York, NY, USA, 1984.
8. P. Cousot and R. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Procs. of the 2d Int. Conf. on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet (SSGRR'01)*. Scuola Superiore G. Reiss Romoli, Aug. 2001.
9. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. In *Proc. of the 11th Annual Asian Computing Science Conf. (ASIAN'06)*, volume 4435 of *LNCS*, pages 272–300. Springer, Dec. 2006.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the 5th ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM, 1978.
11. R. Cousot. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles.* Thèse d'État ès sciences mathématiques, Institut National Polytechnique de Lorraine, Nancy, France, 15 Nov. 1985.
12. J. Feret. Static analysis of digital filters. In *Proc. of the 13th Europ. Symp. on Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 33–48. Springer, Mar. 2004.
13. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.
14. C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference.* Phd thesis, Oxford University, Jun. 1981.

15. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Software Engineering*, 3(2):125–143, Mar. 1977.
16. A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):1–63, Mar. 2012.
17. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, Dec. 1976.
18. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.
19. A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Proc. of the 9th Int. Symp. on Static Analysis (SAS'02)*, volume 2477 of *LNCS*, pages 36–51, 2002.

A Proofs

All the results from Sect. 2 are very classic, so, we do not prove them here (see for instance [4]). Instead, we focus on the new results presented in Sect. 3: the nested fixpoint characterization of the reachable states of a parallel program using interferences.

A.1 Proof of (6)

The proof that the operator F in (6) indeed has a least fixpoint, T, is analogous to that of F in (1). As F is a continuous morphism in the complete partial order of trace sets ordered by inclusion, i.e., $F(\bigcup_i X_i) = \bigcup_i F(X_i)$, we can apply Kleene's fixpoint theorem [4], which states that lfp F exists and is exactly equal to $\bigcup_{n\in\mathbb{N}} F^n(\emptyset)$.

We now repeat this classic proof for the sake of completeness. Let us note $X \overset{\text{def}}{=} \bigcup_{n\in\mathbb{N}} F^n(\emptyset)$. We first prove that X is indeed a fixpoint of F, then that it is smaller than any other fixpoint. Indeed, $F(X) = F(\bigcup_{n\in\mathbb{N}} F^n(\emptyset)) = \bigcup_{n\in\mathbb{N}} F^{n+1}(\emptyset)$, by continuity. Moreover, $F^0(\emptyset) = \emptyset \subseteq F^1(\emptyset)$, hence $F(X) = \bigcup_{n\in\mathbb{N}} F^{n+1}(\emptyset) \cup F^0(\emptyset) = \bigcup_{n\in\mathbb{N}} F^n(\emptyset) = X$. Hence, X is a fixpoint. Let Y be another fixpoint, i.e., $F(Y) = Y$. We have $F^0(\emptyset) = \emptyset \subseteq Y$ and, if $F^n(\emptyset) \subseteq Y$, then $F^{n+1}(\emptyset) = F(F^n(\emptyset)) \subseteq F(Y)$, as the continuity of F implies its monotonicity, hence $F^{n+1}(\emptyset) \subseteq Y$ as $Y = F(Y)$. Thus, by recurrence, $\forall n$, $F^n(\emptyset) \subseteq Y$, which implies that $X \subseteq Y$. Hence, X is the least fixpoint of F.

A.2 Proof of (10)

By definition (7), $\bar{S}[t] \overset{\text{def}}{=} \Pi_t(\alpha^{state}(T))$ and $T = \text{lfp}\ F$ with $F(X) \overset{\text{def}}{=} I \cup \{\, \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_i} \sigma_{i+1} \,|\, \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{i-1}} \sigma_i \in X \wedge \sigma_i \xrightarrow{t_i} \sigma_{i+1} \,\}$. We wish to prove that $\bar{S}[t] = \text{lfp}\ G_t(\bar{A})$, where $G_t(\bar{A})(X) \overset{\text{def}}{=} \Pi_t(I) \cup \{\, \pi_t(\sigma') \,|\, \exists \pi_t(\sigma) \in X, \sigma \xrightarrow{t} \sigma' \vee \exists t' \neq t, (\sigma,\sigma') \in \bar{A}[t'] \,\}$. To lighten notations, we note $G \overset{\text{def}}{=} G_t(\bar{A})$.

We first prove by recurrence on n that $\Pi_t(\alpha^{state}(F^n(\emptyset))) = G^n(\emptyset)$. Intuitively, $G^n(\emptyset)$ corresponds to the states, projected on thread t, that can be

reached from I after at most n execution steps. When $n = 0$, $\Pi_t(\alpha^{state}(F^0(\emptyset))) = G^0(\emptyset) = \emptyset$. Assume that the property holds for some n and let us prove that it holds for $n+1$. As π_t is one-to-one, we can reason equivalently on $G^{n+1}(\emptyset)$ and the states σ such that $\pi_t(\sigma) \in G^{n+1}(\emptyset)$, so, let us consider such a σ. Then, either (a) $\sigma \in I$, or σ can be reached from some σ' such that $\pi_t(\sigma') \in G^n(\emptyset)$ and either (b) $\sigma' \xrightarrow{t} \sigma$ or (c) $\exists t' \neq t$, $(\sigma', \sigma) \in \bar{A}[t']$. By definition, $(\sigma', \sigma) \in \bar{A}[t']$ is equivalent to the existence of a trace $\cdots \sigma' \xrightarrow{t'} \sigma \cdots$ in T. Moreover, by recurrence hypothesis, $\sigma' \in \alpha^{state}(F^n(\emptyset))$, i.e., there exists a trace in T with at most n transitions and ending in σ'. This means that, if a transition $\cdots \sigma' \xrightarrow{t'} \sigma \cdots$ exists in a trace in T, it also exists in a trace of length at most $n+1$ in T (the set of traces is "closed by fusion" [11]). Hence, case (c) is equivalent to the existence of $t' \neq t$ and a trace $\cdots \sigma' \xrightarrow{t'} \sigma \cdots$ in $F^{n+1}(\emptyset)$. Case (b) is equivalent to the existence of a trace $\cdots \sigma' \xrightarrow{t} \sigma \cdots$ in $F^{n+1}(\emptyset)$. Thus, the disjunction of (a), (b), and (c) is equivalent to the existence of a trace in $F^{n+1}(\emptyset)$ containing σ, which is equivalent to $\sigma \in \alpha^{state}(F^{n+1}(\emptyset))$. This ends the proof by recurrence.

We now use the characterization of fixpoints by iteration: we proved in the proof of (6) that $T = \bigcup_{n\in\mathbb{N}} F^n(\emptyset)$. Kleene's theorem implies likewise that $\text{lfp}\ G = \bigcup_{n\in\mathbb{N}} G^n(\emptyset)$. By continuity of Π_t and α^{state}, we get that $\Pi_t(\alpha^{state}(T)) = \Pi_t(\alpha^{state}(\bigcup_{n\in\mathbb{N}} F^n(\emptyset))) = \bigcup_{n\in\mathbb{N}} \Pi_t(\alpha^{state}(F^n(\emptyset)))$. Applying the result of the preceding paragraph, we get $\Pi_t(\alpha^{state}(T)) = \bigcup_{n\in\mathbb{N}} G^n(\emptyset) = \text{lfp}\ G_t(\bar{A})$.

A.3 Proof of (11)

We now prove that $\bar{A}[t] = \{\, (\sigma, \sigma') \mid \pi_t(\sigma) \in \bar{S}[t] \wedge \sigma \xrightarrow{t} \sigma' \,\}$. By definition, $\bar{A}[t] \stackrel{\text{def}}{=} \{\, (\sigma_i, \sigma_{i+1}) \mid \exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in T,\ t_i = t \,\}$. As T is closed by prefix, we have $\bar{A}[t] = \{\, (\sigma_{n-1}, \sigma_n) \mid \exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in T,\ t_{n-1} = t \,\}$. As T is a fixpoint of F, $\exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} \sigma_n \in T$ is equivalent to $\exists \sigma_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-2}} \sigma_{n-1} \in T$ and $\sigma_{n-1} \xrightarrow{t_{n-1}} \sigma_n$. Finally, as $\bar{S}[t] = \Pi_t(\alpha^{state}(T))$, this is equivalent to $\pi_t(\sigma_{n-1}) \in \bar{S}[t]$ and $\sigma_{n-1} \xrightarrow{t_{n-1}} \sigma_n$, which ends the proof.

A.4 Proof of (12)

We now prove that $\bar{S} = \text{lfp}\ \bar{H}$, where $\bar{H}(\bar{X})[t] \stackrel{\text{def}}{=} \text{lfp}\ G_t(\bar{B}(\bar{X}))$, and give the iterative form of $\text{lfp}\ \bar{H}$.

Firstly, we prove that $\bar{S}$ is a fixpoint of $\bar{H}$. Indeed, for any t, $\bar{H}(\bar{S})[t] = \text{lfp}\ G_t(B(\bar{S})) = \text{lfp}\ G_t(\bar{A})$ by (11), which equals $\bar{S}[t]$ by (10).

Secondly, we prove that $\text{lfp}\ \bar{H}$ can be expressed in iterative form. To do so, we prove that $\bar{H}$ is continuous. Kleene's fixpoint theorem applied to the continuous function G_t gives the following characterization of $\bar{H}$: $\bar{H}(\bar{X})[t] = \bigcup_{n\in\mathbb{N}} (G_t(\bar{B}(\bar{X})))^n(\emptyset)$. To simplify notations, we define, for each t and n, the function $\bar{I}_n(\bar{X})[t] \stackrel{\text{def}}{=} (G_t(\bar{B}(\bar{X})))^n(\emptyset)$. Then, we have $\bar{H}(\bar{X}) = \bar{\bigcup}_{n\in\mathbb{N}} \bar{I}_n(\bar{X})$. We note that each $\bar{I}_n$ is continuous. Hence, we have: $\bar{H}(\bar{\bigcup}_i \bar{X}_i) = \bar{\bigcup}_{n\in\mathbb{N}} \bar{I}_n(\bar{\bigcup}_i \bar{X}_i) =$

$\bar{\bigcup}_{n\in\mathbb{N}}\bar{\bigcup}_i \bar{I}_n(\bar{X}_i) = \bar{\bigcup}_i\bar{\bigcup}_{n\in\mathbb{N}}\bar{I}_n(\bar{X}_i) = \bar{\bigcup}_i\bar{H}(\bar{X}_i)$, which proves the continuity of $\bar{H}$. We can thus apply Keene's fixpoint theorem to $\bar{H}$ itself to get lfp $\bar{H}$ in iterative form: lfp $\bar{H} = \bar{\bigcup}_{n\in\mathbb{N}}\bar{H}^n(\bar{\emptyset})$.

Finally, we prove that $\bar{S}\bar{\subseteq}$lfp $\bar{H}$. To do this, we prove by recurrence on n that, if σ_n is reachable after n steps, i.e., if there exists some trace $\sigma_0 \stackrel{t_0}{\rightarrow} \cdots \stackrel{t_{n-1}}{\rightarrow} \sigma_n$ in T, then, for any t, $\pi_t(\sigma_n) \in (\bar{H}^{n+1}(\bar{\emptyset}))[t]$. When $n = 0$, $\bar{H}(\bar{\emptyset})[t] =$ lfp $G_t(\bar{B}(\emptyset)) =$ lfp $G_t(\emptyset) \supseteq \Pi_t(I)$. As $\sigma_0 \in I$, we indeed have $\pi_t(\sigma_0) \in \Pi_t(I)$. Assume that the property is true at rank n and consider a trace $\sigma_0 \stackrel{t_0}{\rightarrow} \cdots \stackrel{t_n}{\rightarrow} \sigma_{n+1}$ in T. As T is closed by prefix, $\sigma_0 \stackrel{t_0}{\rightarrow} \cdots \stackrel{t_{n-1}}{\rightarrow} \sigma_n$ is also in T and we can apply the recurrence hypothesis to get $\pi_t(\sigma_n) \in (\bar{H}^{n+1}(\bar{\emptyset}))[t]$. Moreover, we have $\sigma_n \stackrel{t_{n+1}}{\rightarrow} \sigma_{n+1}$. We consider first the case $t_{n+1} = t$. Then as $(\bar{H}^{n+1}(\bar{\emptyset}))[t]$ is closed by reachability from thread t, we also have $\pi_t(\sigma_{n+1}) \in (\bar{H}^{n+1}(\bar{\emptyset}))[t]$, and so, $\pi_t(\sigma_{n+1}) \in (\bar{H}^{n+2}(\bar{\emptyset}))[t]$. Consider now the case $t_{n+1} \neq t$. Then, $\pi_t(\sigma_n) \in (\bar{H}^{n+1}(\bar{\emptyset}))[t]$ implies that $(\sigma_n, \sigma_{n+1}) \in \bar{B}(\bar{H}^{n+1}(\bar{\emptyset}))[t]$. As a consequence, $(\bar{H}^{n+2}(\bar{\emptyset}))[t]$ is closed under reachability through the transition $\sigma_n \stackrel{t_{n+1}}{\rightarrow} \sigma_{n+1}$, and so, $\pi_t(\sigma_{n+1}) \in (\bar{H}^{n+2}(\bar{\emptyset}))[t]$. We have proved that, for any $\pi_t(\sigma) \in \bar{S}[t]$, there exists some n such that $\pi_t(\sigma) \in (\bar{H}^n(\bar{\emptyset}))[t]$. As lfp $\bar{H} = \bar{\bigcup}_{n\in\mathbb{N}}\bar{H}^n(\bar{\emptyset})$, we get that $\pi_t(\sigma) \in (\text{lfp } \bar{H})[t]$, hence, $\bar{S} \bar{\subseteq}$ lfp $\bar{H}$.