Softw Syst Model manuscript No.
(will be inserted by the editor)

Static Analysis by Abstract Interpretation of Embedded Critical Software

Julien Bertrane · Patrick Cousot · Radhia Cousot · Jérôme Feret · Laurent Mauborgne · Antoine Miné · Xavier Rival

Received: date / Revised version: date

Abstract Formal methods are increasingly used to ensure the correctness of complex critical embedded software systems. We show how sound semantic static analyses based on Abstract Interpretation may be used to check properties at various levels of a software design: from high level models to low level binary code. After a short introduction to the Abstract Interpretation theory, we present a few current applications: checking for run-time errors in synchronous and parallel embedded applications at the C level, translation validation from C to assembly, and analyzing SAO models of communicating synchronous systems with imperfect clocks. We conclude by briefly proposing some requirements to apply Abstract Interpretation to modeling languages such as UML.

keywords: Abstract interpretation, Critical software, Embedded systems, Static analysis, System design, System modeling, System verification.

1 Introduction

Ensuring the correctness of software systems constitutes a large part of software development budgets. It is particularly important for critical embedded systems, such as found in automotive, aerospace, and medical applications, as the slightest programming "bug" may have a catastrophic financial and even human cost. In this article, we build a case for using static analysis based on Abstract Interpretation to help ensuring software correctness.

We illustrate a possible use for static analysis in Fig. 1. In this drastically simplified workflow inspired from a real industrial case [41], an engineer (not necessarily a programmer) models a control system using the SAO graphical language, a precursor and similar tool as

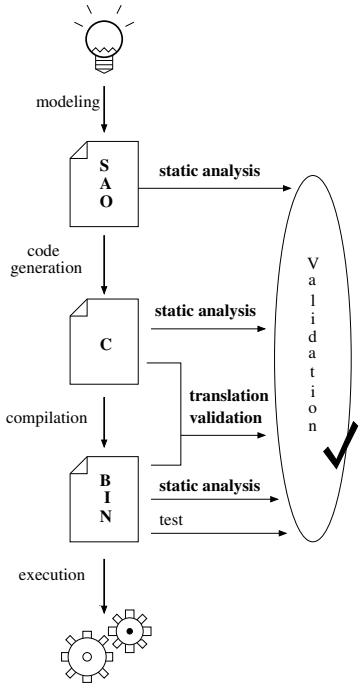


Fig. 1 Example workflow for designing an embedded application.

Julien Bertrane · Patrick Cousot · Radhia Cousot · Jérôme Feret · Antoine Miné · Xavier Rival
Équipe CNRS-ENS-INRIA Sémantique et Interprétation Abstraite. Département d'informatique, École normale supérieure, 45 rue d'Ulm, F-75230 Paris Cedex 05, France. E-mail: firstname.lastname@ens.fr

Laurent Mauborgne
Fundación IMDEA Software Facultad de Informática (UPM), office 3312, Campus Montegancedo 28660-Boadilla del Monte, Madrid, Spain. E-mail: laurent.mauborgne@imdea.org

Patrick Cousot
Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, N.Y., USA.

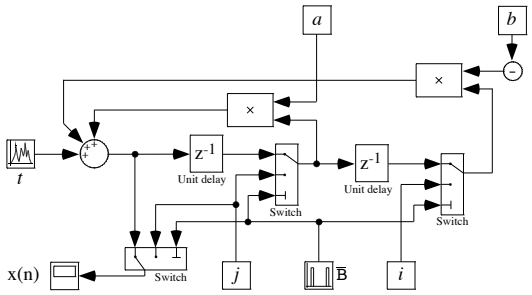


Fig. 2 A second-order digital filter specified in Simulink.

SCADE [20] — a Simulink™ fragment similar to SAO is also shown in Fig. 2. It is then automatically translated to the C programming language and then compiled to produce the actual binary software executed by the device. Validation includes testing, which requires executing (part of) the binary with some monitoring and is able to check a wide range of properties (including functional ones) but is costly and never achieves a full coverage of all possible executions (path- and data-coverage). Formal methods can also be employed. In particular, semantic-based static analysis, which always terminates and covers all executions, albeit in an over-approximated way. For example it can detect dead code (never executed) or dead data structures (constructed but never used) but cannot always prove their absence. As another example, it can prove that some code is free from arithmetic and memory overflow errors, but cannot always prove their presence (reported alarms may be actual errors or spurious alarms due to over-approximation). Static analysis can be applied at many levels: machine-readable specification, program source, or binary. The higher level the better, as it provides purer information to the tool and its feedback is easier to understand and act upon for the designer. However, higher levels abstract away some aspects of computations, which makes it impossible to check some properties of actual executions. For instance, SAO and SCADE have real arithmetics and do not specify how actual numerical computations are performed nor the type of numbers, so that a static analysis of numeric overflows (as done by Astrée [1,17,9], Sec. 3) or of the precision of floating-point computations (as done by Fluctuat [24]) must be done at the C level or below — a static analysis of real expressions at the SCADE level may however be used to determine its numerically most precise float compilation to C [29]. Likewise, neither SAO nor C make any guarantee about the worst-case execution time, so, such an analysis (as done by AbsInt's aiT [26]) is done at the binary level and for a specific processor. Nevertheless, some properties can only be checked at the model level. For instance, a SAO model can be enriched with non-software elements, such as real-time clocks and communication lines with delays, to enable a static analysis taking physical time into account (Sec. 6). Finally, static analysis can also improve the confidence in compilers and code generators: translation validation (Sec. 5) can check whether the source and binary are equivalent, at least with respect to a class of properties, so that the analysis for such properties at a higher level needs not be redone at the lowest level (which is often more difficult).

Ideally, a static analyzer should extract automatically precise properties from a complete mathematical specification of the analyzed system. Most properties are however undecidable, so, we resort to abstraction, i.e. the analyzer explores machine-representable supersets of actual behaviors of the system using tractable algorithms. As a consequence, the analyzer may consider spurious behaviors and miss properties, but the analysis is sound: all the properties that are found (absence of run-time errors, worst-case execution time, etc.) are indeed true for all executions. A specificity of static analysis is that it works directly on the concrete system that is input to compilers or code generators, and the abstracted system is derived automatically according to built-in abstraction mechanisms. No abstract system need to be provided — which would be time consuming and pose the question of whether it indeed corresponds to the concrete one. We use the Abstract Interpretation framework [13], a general theory of the approximation of semantics, to design static analyzers that are sound by construction. There is no silver bullet: each static analyzer should be tailored to a specific class of properties and programs to achieve both precision (low rate of missed properties) and efficiency. Thankfully, Abstract Interpretation provides a growing library of ready-to-use abstractions, and the mean to design new ones in a principled way.

After a short formal introduction to Abstract Interpretation theory (Sec. 2), this article describes more informally several static analysis applications: checking for run-time errors in critical embedded synchronous C software with Astrée (Sec. 3), ongoing extensions to parallel embedded software (Sec. 4), translation validation from C to assembly (Sec. 5), and analyzing communicating synchronous systems with imperfect clocks (Sec. 6). Section 7 concludes and suggests the application of Abstract Interpretation to modeling languages such as UML.

2 Abstract Interpretation

We provide a succinct introduction to Abstract Interpretation. More details are provided e.g. in [7].

2.1 Small-Step Operational Semantics

In order to analyze the behavior of the executions of a computer system, we start by providing a model of computations, that is, an operational semantics. An example of operational semantics for UML-Statecharts would be [5].

Such an operational semantics of a given program can be described as a transition system $\langle \mathcal{S}, \mathcal{I}, \mathcal{E}, t \rangle$. $\mathcal{S}$ is a set of states, including initial states $\mathcal{I} \subseteq \mathcal{S}$ and bad or erroneous states $\mathcal{E} \subseteq \mathcal{S}$. $t \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation between a state $s \in \mathcal{S}$ and its possible successors: for any state $s' \in \mathcal{S}$, $t(s, s')$ is true if, and only if, s' is a potential successor of s. The blocking states have no successor $\mathcal{B} \triangleq \{s \in \mathcal{S} \mid \forall s' \in \mathcal{S} : \neg t(s, s')\}$.

2.2 Big-Step Operational Semantics

The big-step operational semantics of $\langle \mathcal{S}, \mathcal{I}, \mathcal{E}, t \rangle$ is $\langle \mathcal{S}, \mathcal{I}, \mathcal{E}, t^\star \rangle$ where $t^\star \triangleq \bigcup_{n \geqslant 0} t^n$ is the reflexive transitive closure of t,[1] $t^0 \triangleq \mathbf{1}_{\mathcal{S}} \triangleq \{\langle s, s \rangle \mid s \in \mathcal{S}\}$ is the identity relation on $\mathcal{S}$, $t^{n+1} \triangleq t^n \circ t$ where $\circ$ is the composition of relations.[2]

We have $t^\star = T(t^\star)$ where $T(r) \triangleq \mathbf{1}_{\mathcal{S}} \cup r \circ t$ since a state s' is reachable from s in $n \geqslant 0$ steps if and only if $n = 0$ and $s = s'$ or $n > 0$ and s' is reachable from a successor of s in $n - 1$ steps. Moreover, if $T(r) = \mathbf{1}_{\mathcal{S}} \cup r \circ t \subseteq r$ then $t^\star \subseteq r$. It follows that $t^\star = \mathbf{lfp}^{\subseteq} T$, by definition of the $\subseteq$-least fixpoint $\mathbf{lfp}^{\subseteq} T$ of T.[3]

Because all non-trivial properties of programs are undecidable, $t^\star = \mathbf{lfp}^{\subseteq} T$ is not computable for infinite state transition systems $\langle \mathcal{S}, \mathcal{I}, \mathcal{E}, t \rangle$ (except for trivial programs t and specifications $\mathcal{E}$).

2.3 Specification

A typical verification problem is to prove that no execution starting in an initial state can reach a bad state (e.g. where the next execution step would raise an error). The correctness condition is $\forall s \in \mathcal{I} : \forall s' \in \mathcal{S} : t^\star(s, s') \implies s' \notin \mathcal{E}$ that is no state s' reachable from the initial states is a bad state. For example, $\mathcal{E}$ can be the set of blocking states in order to specify the absence of deadlocks. Error-freedom can also be written $\mathcal{R} \subseteq \mathcal{S} \setminus \mathcal{E}$ where $\mathcal{R} \triangleq \{s' \mid \exists s \in \mathcal{I} : t^\star(s, s')\}$ is the set of states reachable from the initial states and $X \setminus Y \triangleq \{x \in X \mid x \notin Y\}$.

Define $F(X) \triangleq \mathcal{I} \cup \mathbf{post}[t]X$ where $\mathbf{post}[r]X \triangleq r(X) \triangleq \{s' \mid \exists s \in X : r(s, s')\}$ is the image of the set X by the relation r. We have $F \in \wp(\mathcal{S}) \to \wp(\mathcal{S})$[4] is additive[5] hence strict[6] and increasing.[7] We have $\mathcal{R} = F(\mathcal{R})$ since a state is reachable iff it is an initial state or the successor of a reachable state. If $F(X) \subseteq X$ then X contains the initial states $\mathcal{I}$ and, transitively, any of its successors, so, $\mathcal{R} \subseteq X$. This implies that $\mathcal{R} = \mathbf{lfp}^{\subseteq} F$, which is not computable either.

2.4 Abstraction

2.4.1 Intervals

Let us start with the simple example of abstracting a set $V \subseteq \mathbb{Z}$ of integers (e.g. the set of possible values of an integer variable) by an interval of values $\alpha_i(V) \triangleq [\min V, \max V]$ (where $\min \mathbb{Z} \triangleq \max \emptyset \triangleq -\infty$ and $\max \mathbb{Z} \triangleq \min \emptyset \triangleq +\infty$). In particular, for the empty set $\emptyset$, $\alpha_i(\emptyset) = [+\infty, -\infty]$. In general, this is obviously an over-approximation since the interval $\alpha_i(V)$ may contain spurious values not in V. So from $z \notin \alpha_i(V)$ we can conclude $z \notin V$ whereas knowing that $z \in \alpha_i(V)$ in the abstract, we do not know whether $z \in V$ or $z \notin V$ in the concrete since z might be a spurious value. The concretization is $\gamma_i([\ell, h]) \triangleq \{z \in \mathbb{Z} \mid \ell \leqslant z \leqslant h\}$. Let us define the abstract domain of intervals $V_i^\sharp \triangleq \{[\ell, h] \mid \ell \in \mathbb{Z} \cup \{-\infty\} \wedge h \in \mathbb{Z} \cup \{+\infty\} \wedge \ell \leqslant h\} \cup \{[+\infty, -\infty]\}$. We have $\forall V \in \wp(\mathbb{Z}) : \forall [\ell, h] \in V_i^\sharp : \alpha_i(V) \subseteq [\ell, h] \iff V \subseteq \gamma_i([\ell, h])$ and so, by definition, the pair $\langle \alpha_i, \gamma_i \rangle$ is a Galois connection,[8] written $\langle \wp(\mathbb{Z}), \subseteq \rangle \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} \langle V_i^\sharp, \subseteq \rangle$.

2.4.2 Cartesian Abstraction

A set $V \subseteq \mathbb{Z}^n$ of vectors of $n \geqslant 1$ integer values (e.g. the set of possible values of n integer variables) can be abstracted by projection along each component $\alpha_C(V) \triangleq \prod_{i=1}^n \{z \mid \exists z_1, \ldots, z_{i-1}, z_{i+1}, \ldots, z_n \in \mathbb{Z} : \langle z_1, \ldots, z_{i-1}, z, z_{i+1}, \ldots, z_n \rangle \in V\}$. The concretization is $\gamma_C(\langle V_1,$

[1] It follows that $t^\star(s, s') = \exists n \geqslant 0 : \exists s_0, \ldots, s_n : s = s_0 \wedge t(s_0, s_1) \wedge \ldots \wedge t(s_{n-1}, s_n) \wedge s_n = s'$, including the case $s = s'$ for $n = 0$.

[2] $\circ$ is the composition of relations that is $r_1 \circ r_2 \triangleq \{\langle x, x'' \rangle \mid \exists x' : \langle x, x' \rangle \in r_1 \wedge \langle x', x'' \rangle \in r_2\}$.

[3] The $\sqsubseteq$-least fixpoint $\mathbf{lfp}^{\sqsubseteq} f$ of an increasing map f on a poset partially ordered by $\sqsubseteq$ is defined by $f(\mathbf{lfp}^{\sqsubseteq} f) = \mathbf{lfp}^{\sqsubseteq} f$ and $f(x) = x$ implies $\mathbf{lfp}^{\sqsubseteq} f \sqsubseteq x$.

[4] $\wp(X) = \{Y \mid Y \subseteq X\}$ is the set of all subsets Y of a set X.

[5] F is additive iff $F(\bigcup_{i \in \Delta} X_i) = \bigcup_{i \in \Delta} F(X_i)$.

[6] F is strict whenever $F(\emptyset) = \emptyset$.

[7] F is increasing whenever $X \subseteq Y$ implies $F(X) \subseteq F(Y)$.

[8] By definition, $\langle C, \preceq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq \rangle$ if and only if $\langle C, \preceq \rangle$ and $\langle A, \sqsubseteq \rangle$ are posets, $\alpha \in C \to A$, $\gamma \in A \to C$ and $\forall x \in C, y \in A : \alpha(x) \sqsubseteq y \iff x \preceq \gamma(y)$. $\Longleftarrow$ implies soundness in that y is an abstract over-approximation of the concrete x and $\Longrightarrow$ implies that $\alpha(x)$ is the best abstraction of x in that it is more precise than any other sound abstraction y.

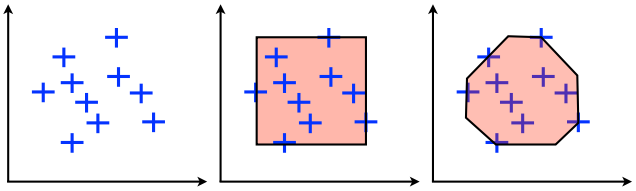


Fig. 3 A set and its interval and octagonal abstractions.

$\ldots, V_n\rangle) \triangleq \{\langle z_1, \ldots, z_n\rangle \mid \bigwedge_{i=1}^n z_i \in V_i\}$ so that $\langle \wp(\mathbb{Z}^n), \subseteq\rangle \underset{\alpha_C}{\overset{\gamma_C}{\leftrightarrows}} \langle \wp(\mathbb{Z})^n, \dot\subseteq\rangle$ where $\dot\subseteq$ is the componentwise ordering.[9] Composed with the interval abstraction, this specifies the interval analysis of [13] such that $\alpha_{i\circ C}(V) \triangleq \langle \alpha_i(V_1), \ldots, \alpha_i(V_n)\rangle$ where $\langle V_1, \ldots, V_n\rangle \triangleq \alpha_C(V)$ which implies $\langle \wp(\mathbb{Z}^n), \subseteq\rangle \underset{\alpha_{i\circ C}}{\overset{\gamma_{i\circ C}}{\leftrightarrows}} \langle {V_i^\sharp}^n, \dot\subseteq\rangle$. This abstraction forgets about relationships between values of variables (such as whether variables have equal values). To keep relations between values $X, Y, \ldots$ of numerical variables, more refined abstractions must be used such as octagons [34] that infer relations of the form $\pm X \pm Y \leq c$ (where c is a constant automatically inferred by the static analysis), see Fig. 3.

2.4.3 Transformers

Transformers, that is, relations between states, can also be abstracted. Consider for example the abstraction of a relation $r \subseteq S \times S$ by its right image $\alpha_I(r) \triangleq r(I) \triangleq \mathbf{post}[r]I \triangleq \{x' \mid \exists x \in I : r(x, x')\}$ of a given set $I \subseteq S$ (e.g. of initial states). For example, the reachable states of $\langle \mathcal{S}, \mathcal{I}, \mathcal{E}, t\rangle$ are $\mathcal{R} = \alpha_{\mathcal{I}}(t^\star)$ so reachability is an abstraction of the big-step semantics. We have:

$$
\begin{array}{lll}
& \alpha_I(r) \dot\subseteq R & \\
\Leftrightarrow & \{x' \mid \exists x \in I : r(x, x')\} \subseteq R & \wr\text{def. } \alpha_I\wr \\
\Leftrightarrow & \forall x' \in S : (\exists x \in I : r(x, x')) \implies x' \in R & \wr\text{def. } \subseteq\wr \\
\Leftrightarrow & \forall x' \in S : \forall x \in I : r(x, x') \implies x' \in R & \wr\text{generalization}\wr \\
\Leftrightarrow & \forall x, x' \in S : r(x, x') \implies (x \in I \implies x' \in R) & \wr\text{def. } \implies\wr \\
\Leftrightarrow & r \subseteq \{\langle x, x'\rangle \mid x \in I \implies x' \in R\} & \wr\text{def. } \subseteq\wr \\
\Leftrightarrow & r \subseteq \gamma_I(R) & \\
& \multicolumn{2}{r}{\wr\text{by defining } \gamma_I(R) \triangleq \{\langle x, x'\rangle \mid x \in I \implies x' \in R\}\wr}
\end{array}
$$

which is the characteristic property of the Galois connection $\langle \wp(\mathcal{S} \times \mathcal{S}), \subseteq\rangle \underset{\alpha_I}{\overset{\gamma_I}{\leftrightarrows}} \langle \wp(\mathcal{S}), \subseteq\rangle$.

2.4.4 Fixpoint Abstraction

For reachability, we have $\mathcal{R} = \alpha_{\mathcal{I}}(t^\star) = \alpha_{\mathcal{I}}(\mathbf{lfp}^{\subseteq} T) = \mathbf{lfp}^{\subseteq} F$ so that the abstraction $\alpha_{\mathcal{I}}(\mathbf{lfp}^{\subseteq} T)$ of the concrete fixpoint $\mathbf{lfp}^{\subseteq} T$ can be calculated as an abstract fixpoint $\mathbf{lfp}^{\subseteq} F$ not referring to the concrete world at all. This follows from a general result sketched below and the observation that:

$$
\begin{array}{lll}
& \alpha_I \circ T(r) & \\
= & \alpha_I(\mathbf{1}_{\mathcal{S}} \cup r \circ t) & \wr\text{def. } \circ \text{ and } T\wr \\
= & \alpha_I(\mathbf{1}_{\mathcal{S}}) \cup \alpha_I(r \circ t) & \wr\text{def. } \alpha_I\wr \\
= & I \cup \{x' \mid \exists x \in I : (r \circ t)(x, x')\} & \wr\text{def. } \mathbf{1}_{\mathcal{S}} \text{ and } \alpha_I\wr \\
= & I \cup \{x' \mid \exists x \in I : \exists x'' : r(x, x'') \wedge t(x'', x')\} & \wr\text{def. } \circ\wr \\
= & I \cup \{x' \mid \exists x'' \in \{x'' \mid \exists x \in I : r(x, x'')\} : t(x'', x')\} & \wr\text{def. } \exists, \in\wr \\
= & I \cup \{x' \mid \exists x'' \in \alpha_I(r) : t(x'', x')\} & \wr\text{def. } \alpha_I\wr \\
= & I \cup \mathbf{post}[t](\alpha_I(r)) & \wr\text{def. } \mathbf{post}[t]\wr \\
= & F \circ \alpha_I(r) & \wr\text{def. } F \text{ and } \circ\wr
\end{array}
$$

The above calculus also shows how to calculate the abstract transformer F from the concrete transformer T, which is the idea of the calculational design of static analyzers [12].

More generally, under suitable hypotheses of existence of joins in posets and fixpoints [13,12], if $f \in C \to C$, $\langle C, \preceq\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A^\sharp, \sqsubseteq\rangle$ and $f^\sharp \in A^\sharp \to A^\sharp$ satisfy $\alpha \circ f = f^\sharp \circ \alpha$, then $\alpha(\mathbf{lfp}^{\preceq} f) = \mathbf{lfp}^{\sqsubseteq} f^\sharp$.

Faced with undecidable problems, $\alpha(\mathbf{lfp}^{\preceq} f)$ is often non-computable, in which case it must be over-approximated $\alpha(\mathbf{lfp}^{\preceq} f) \sqsubseteq \mathbf{lfp}^{\sqsubseteq} f^\sharp$, which follows from the local condition $\alpha \circ f \dot\sqsubseteq f^\sharp \circ \alpha$.[10] This is the case for example for interval analysis [13].

2.4.5 Fixpoint Approximation

Under suitable hypotheses of existence of joins in posets and fixpoints [13,12], fixpoints can be computed iteratively. For example, $\mathcal{R} = \bigcup_n F^n(\emptyset)$[11] with the intuition that the reachable states are reachable in either 0, 1, 2, ..., n, ... computation steps. However, in general, convergence of the iterates to a fixpoint may require infinitely many iterations (for undecidable problems) or suffer a combinatorial explosion in time and memory (for finite but complex problems). But for rare cases where the fixpoint can be computed directly (e.g. by elimination), convergence must in general be accelerated, e.g. using a widening ∇ [13] at the prejudice of precision. A naïve example of widening for intervals is $[\ell^i, h^i]\nabla[\ell^{i+1}, h^{i+1}] \triangleq$ [if $\ell^{i+1} < \ell^i$ then $-\infty$ else ℓ^i, if $h^{i+1} > h^i$ then $+\infty$ else h^i] so that unstable bounds are pushed to infinity, which enforces rapid although imprecise convergence. A narrowing can then be used to remove some of the infinite bounds [13].

2.4.6 Design of a Static Analyzer

The design of a static analyzer for a (specification or programming) language starts with the definition of its

[9] The componentwise ordering is: $\langle V_1, \ldots, V_n\rangle \dot\subseteq \langle V'_1, \ldots, V'_n\rangle$ if and only if $\bigwedge_{i=1}^n (V_i \subseteq V'_i)$.

[10] The pointwise ordering is $f \dot\sqsubseteq g$ if and only if $\forall x : f(x) \sqsubseteq g(x)$.

[11] The powers of a function f are: f^0 is the identity, $f^1 \triangleq f$, and $f^{n+1} = f \circ f^n$.

semantics and the properties of interest for each program of the language, often in the form of fixpoints $\mathbf{lfp}^{\preceq} F$ corresponding to an abstraction of the notion of computation (e.g. reachability). F is then designed by induction on the language syntax. Then, an abstraction α is defined, which is a complex task, hence must be done by composition of simpler abstractions, using standard composition methods such as the reduced product $\alpha(X) \triangleq \bigwedge_{i=1}^{m} \alpha_i(X)$ combining different abstractions $\alpha_i(X)$ [14] and standard abstractions (some already implemented in libraries [31]). The static analyzer is then designed by induction on the language syntax. It reads a program, computes the abstract transformer $F^\sharp$ (designed to satisfy $\alpha \circ F \sqsubseteq F^\sharp \circ \alpha$) for that program, and over-approximates the abstract fixpoint $\mathbf{lfp}^{\sqsubseteq} F^\sharp$ by an iterative computation with convergence acceleration with widening/narrowing. This ensures that the result A (e.g. an abstract invariant for reachability) is sound (i.e. $\alpha(\mathbf{lfp}^{\preceq} F) \sqsubseteq \mathbf{lfp}^{\sqsubseteq} F^\sharp \sqsubseteq A$). The result A is then used for verification purposes (e.g. to prove the absence of run-time errors).

3 Checking Run-Time Errors in Embedded Synchronous Software with Astrée

We now describe a first concrete application of Abstract Interpretation: the static analyzer Astrée [8] that checks for run-time errors in embedded C programs and has been successfully used in aeronautics [19] and aerospace [10]. Astrée started in 2001 as an academic project [17] and is now a mature tool industrialized by AbsInt Angewandte Informatik GmbH [1].

3.1 Analyzed Software

Astrée accepts a fairly large subset of C, excluding dynamic memory allocation, recursion, and parallelism, that are often unused (even forbidden) in embedded code. The language syntax and concrete semantics are based on the C99 norm [30], supplemented with the IEEE 754-1985 norm [27] for floating-point computations. However, the C99 norm leaves many aspects of the semantics unspecified and lets implementations decide. As embedded software are rarely strictly conforming but rely instead on platform-specific features, Astrée also takes them into account and provides options for the user to tune some semantic aspects (e.g. the bit-size, alignment, and byte order of data-types) to fit its application.

Programs analyzed by Astrée should be standalone, i.e. have no undefined symbols. In particular, if the program uses a library, then its source code must be provided to Astrée as well. Alternatively, one can provide a *stub* for undefined functions instead of an actual implementation, which is particularly useful when only a specification of the function is known. Stubs generally provide only pre- and post-conditions on the function arguments and return values, abstracting away the actual computation. Moreover, as programs generally run within an environment and typically fetch external data (e.g. sensor values) through memory-mapped registers, Astrée allows specifying memory locations as "volatile" with a range of expected values (or possibly the full range of the type, including special *NaN* or $\pm\infty$ float values). The analysis naturally considers all possible sequences of inputs in the specified ranges.

Although Astrée accepts a large variety of C codes, it cannot analyze most of them precisely and efficiently. It is mainly specialized, by its choice of abstractions, to analyze control / command synchronous programs automatically generated from higher level specifications (e.g. as in Fig. 2). Once generated, such codes have the following structure:

```
initialize state variables
loop for 10 h
    read inputs from sensors
    update state and compute outputs
    write outputs to actuators
    wait for next clock tick (10 ms)
```

The read, update, and write instructions may be scattered in the source code by the code generator or encapsulated within functions. Such codes also feature a very large number of global variables representing the current state. They are generally numeric intensive, featuring much floating-point computations. There are, however, few nested loops and, except for the outer loop running for a very long time, inner loops generally have a fixed, small number of iterations. Finally, such codes are often generated from instancing a limited number of macros, and feature few, recurring code patterns.

3.2 Astrée usage

The usage of Astrée is depicted in Fig. 4. Astrée takes as argument the set of C source files (after preprocessing by a standard C preprocessor) and an optional configuration file describing the range of volatile variables. Astrée then computes an abstraction of the semantics of the program, leading to over-approximated invariants. It emits an alarm whenever the computed semantics leads to a run-time error. It can also output the computed invariants for selected variables and program points, which is useful to understand the origin of alarms.

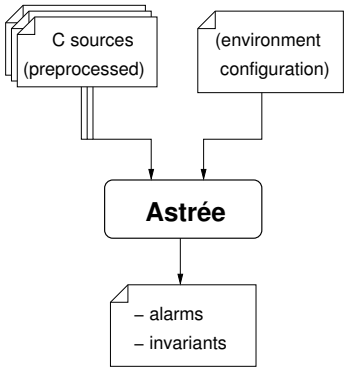


Fig. 4 Input and output of Astrée.

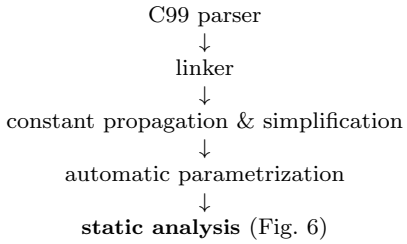


Fig. 5 Astrée phases.

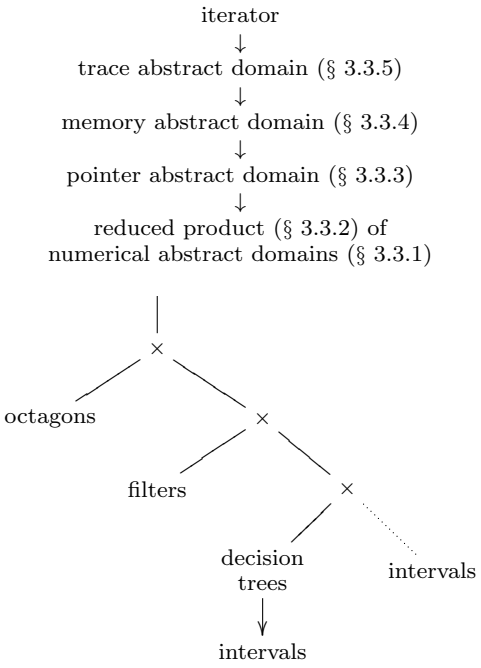


Fig. 6 Hierarchy of abstractions in Astrée.

The errors checked by Astrée correspond to operations that are either undefined on the user-chosen platform (e.g. out-of-bound array accesses) or have unintended results (e.g. wrap-around after integer overflows). The property to verify (absence of run-time errors) is thus implicit and derived from the program's own source. More precisely, Astrée checks overflows in unsigned and signed integer arithmetics and casts, divisions by zero, infinities and *not a number* special floating-point values (caused by overflows and invalid operations), out-of-bound array accesses, invalid uses of pointers (dereferencing NULL, dangling, out-of-bound, or misaligned pointers), and failure of user-specified assertions (using a construct similar to the C function `assert`).

Astrée does not stop at the first error it encounters but instead continues the analysis for all executions that have a well-defined semantics (e.g. for integer overflows with wrap-around, but not for divisions by zero). Thus, if there are no alarms, or if all executions leading to an alarm can be proved by other means to be spurious, then the program is indeed free from run-time errors.

3.3 Design of Astrée

Similarly to a compiler, Astrée operates in several phases (Fig. 5). The source files are first parsed and transformed into an internal, graph-based, explicitly typed representation. They are then linked together, resolving undefined symbols. A fast intra-procedural analysis performs constant propagation and eliminates unused variables in order to simplify the program before more costly phases. A simple syntax-based heuristics detects which parts of the program and which variables would benefit from a higher precision (such as relational or disjunctive analyses).

After these simple and fast phases, the actual static analysis starts. The analysis is performed as an interpretation in the abstract: an iterator traverses the program by structural induction on its syntax, iterating loops and stepping into functions, to collect an abstraction of all possible execution traces. As a result, the analysis is fully flow and context sensitive. The termination is guaranteed by the use of widening operators $\triangledown$ [13] to accelerate loops and by the absence of recursion.

The abstraction computation is performed using abstract domains, which are modules providing a family of abstract properties together with a machine representation and algorithms to manipulate them according to the semantics of basic program instructions (assignments, tests, control-flow joins, etc.). The abstraction used in Astrée is composed of several abstract domains organized as shown in Fig. 6. Some abstract domains are parametrized by other abstract domains (shown using the $\downarrow$ symbol in the figure), while others are combined through a reduced product (shown as $\times$ in the figure) which is a generic way to represent conjunctions of heterogeneous properties (Sec. 3.3.2). The

```
if (X > Y) X = Y;
/* X ≤ Y */
if (Y <= 10) {
  /* X ≤ 10 */
}
```

```
x = 10;
for (i=0; i<1000; i++) {
  /* x ≤ i + 10 */
  if (...) x++;
  if (...) x = 0;
}
/* 0 ≤ x ≤ 1010 */
```

Fig. 7 Two programs benefiting from the octagon domain.

rest of the section presents a selection of the domains available in Astrée.

3.3.1 Numerical Abstract Domains

A numerical abstract domain focuses on properties of the integer and floating-point variables of the program. There exist many general-purpose domains, with various expressiveness and cost versus precision trade-offs. Astrée implement several of them (e.g. interval, octagon, and decision tree domains). Additionally, Astrée implements some specific-purpose domains specially designed to handle the kinds of computations found in embedded control / command software (e.g. digital filter, arithmetic-geometric progression, quaternion domains).

Interval Domain. Most run-time errors considered by Astrée (with the exception of divisions by zero) can be eliminated by inferring precise bounds on variables. Hence, Astrée implements the classic interval domain able to express and infer such bounds (see Sec. 2.4.1 and [13]). In floating-point arithmetics, the soundness with respect to all rounding modes is guaranteed by simply rounding upper bounds towards $+\infty$ and lower bounds towards $-\infty$. To obtain an efficient implementation that scales up to thousands of variables and millions of lines of code, environments mapping variables to intervals are stored in functional maps with sharing based on balanced binary trees. Indeed, such data-structures enjoy a constant-time copy and a very fast component-wise join operator (in time $k \log n$, where k is the number of variables that actually differ in both environments and n is the total number of variables), compared to a linear cost for plain arrays.

Octagon Domain. In order to infer precise bounds, it is often necessary to infer locally much stronger properties. Consider, for instance, the program on the left of Fig. 7, that computes the minimum of X and Y into X, and then branches when Y is less than 10. An interval analysis would only be able to discover that $Y \leq 10$ after the test `Y<=10`. To discover that, additionally, $X \leq 10$, it is necessary to infer and use the relation $X \leq Y$. Another example is the case of the loop on the right of Fig. 7. It is necessary to infer the loop invariant $x \leq i + 10$ to deduce that, when the loop ends, $x \leq 1010$. Both cases require relational properties, which cannot be expressed in a domain using the Cartesian abstraction (Sec. 2.4.2) such as the interval domain.

The octagon domain, proposed in [34], is a relational domain able to infer conjunctions of constraints of the form $\pm X \pm Y \leq c$. It is a restriction of the polyhedra[12] domain [18]. Octagons are based on a matrix representation with quadratic memory cost, and shortest-path closure algorithms with cubic time cost. Hence, octagons are much more efficient than general polyhedra (which have an exponential cost). Moreover, the octagon domain can easily be implemented using floating-point arithmetics by rounding towards $+\infty$ (as it only manipulates upper bounds). It is sufficient to precisely analyze both programs in Fig. 7.

Despite being more efficient than polyhedra, a cubic cost is still too high to scale up to thousands of variables. Hence, Astrée does not try to relate all variables at once in a single large octagon, but considers instead a collection of small octagons. The static set of octagons is determined by the automatic parametrization preceding the analysis (Fig. 5) in a simple way: variables used together in the same or in nearby linear expressions (say, within the same syntactic block) are put in the same octagon. This yields a number of octagons linear in the code size, each packing few variables (from two to a dozen), and so, the overall time and memory cost is linear in the program size.

Decision Tree Domain. The interval and octagon domains can only represent convex sets of points, which is not precise enough to handle disjunctive properties,

```
b = (x >= 6);
...
if (b) y = 10 / (x-4);
```

Fig. 8 Program benefiting from the decision tree domain.

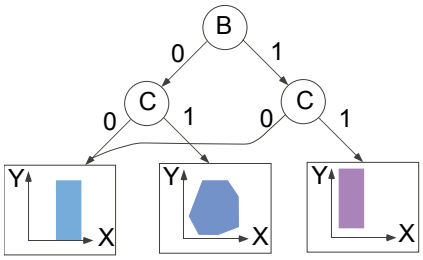


Fig. 9 Example decision tree, with boolean variables B and C, and numerical variables X and Y.

[12] The name "octagon" comes from the shape of such restricted polyhedra in two dimensions.

```
float I[2], O[2];
while (1) {
  X = input();
  if (init()) { O[0] = X; P = X; I[0] = X; }
  else
    P = 0.4677826 * X
      - 0.7700725 * I[0] + 0.4344376 * I[1]
      + 1.5419 * O[0] - 0.6740477 * O[1];
  I[1] = I[0]; I[0] = X;
  O[1] = O[0]; O[0] = P;
}
```

Fig. 10 Second order filter example.

such as case analyses or boolean reasoning, nor to express the absence of division by zero. Consider, for instance the program of Fig. 8, where the truth value of a numerical predicate `x >= 6` is stored into a boolean `b`, which is used later in the program. In order to prove that there is no division by zero in `10 / (x-4)`, it is necessary to infer and use the disjunctive predicate $(b = 1 \wedge x \geq 6) \vee (b = 0 \wedge x < 6)$ (in C, booleans are simply integers, and a predicates returns either 0, for false, or 1, for true).

In Astrée, such properties are encoded by a decision tree where internal nodes denote boolean variables (with a branch for each value, zero and one) and leaves store arbitrary abstract properties of one or several numerical variables. An example is shown in Fig. 9. Such trees bear some resemblance with classic BDDs [11] and enjoy similar sharing properties. The decision tree domain is parametrized by the choice of a domain for leaves (e.g. the interval domain), a set of boolean variables, and a set of numerical variables. As for octagons, we do not use a single large tree, but many small trees, where the set of variables is fixed by a heuristics occurring before the static analysis, so that the overall cost is still linear in the size of the program. This heuristics collects sets of related booleans and numerical variables using a simple and approximate dependency analysis.

Filter Domain. A first example of application domain specific abstraction is the digital filter abstract domain. Digital filters often appear in control / command embedded software to smooth the stream of values input by sensors. Figure 2 presents an example second order digital filter as programmed by an engineer using Simulink and Fig. 10 shows one possible generated code. It implements a recurrence relation where, at each clock tick, a new output `P` is computed as a linear combination of the last three inputs, `X`, `I[0]`, and `I[1]`, and the last two previously computed outputs, `O[0]` and `O[1]` (except for the reinitialization case where the output is set to the input). Figure 11 plots an example behavior and hints at the form of the invariant we seek, which is an ellipse aligned on the first bisector. Such a quadratic

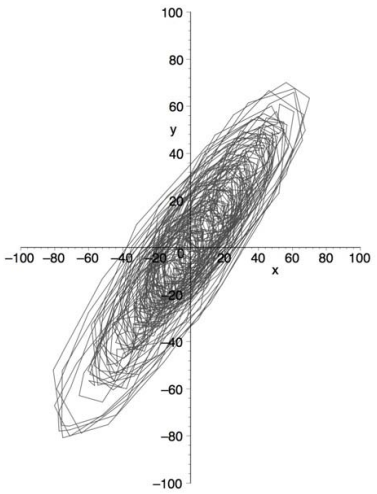


Fig. 11 Typical behavior of a second order digital filter. One axis shows the current output `O[0]`, and the other shows the previous one `O[1]`.

invariant is out of reach of the interval and octagon domains.

Astrée implements a specific domain, the digital filter domain [21], that is able to detect recurrence relations denoting filters, and synthesize a quadratic invariant from the recurrence coefficients and a bound on the input. Often, the filter depends on the value of the input at several successive iterations. A simple solution would be to abstract these contributions as a single global interval. However, it often occurs that the various contributions of an input read at a given iteration partially cancel each others. This is for instance the case in Fig. 10, as the coefficient of `I[0]` has a sign which is the oposite of the one of both `X` and `I[1]`. To exploit this, the digital filter domain performs a formal unrolling of the recurrence relation, up to a limited number of iterations. The residue is generally a small fraction of the bound of the input, which provides an increase in precision. Lastly, by linearity, the overall contribution of this residue along the iterations of the filter, can be abstracted by using quadratic inequalities.

Arithmetic-Geometric Progression Domain. The arithmetic-geometric progression domain [22] is another domain for non-linear properties, able to infer invariants of the form: $X \leq \alpha(1+\beta)^{\texttt{clock}}$. Such invariants correspond to slowly diverging computations such as, for instances, an assignment `X = X +` ϵ or `X =` $(1+\varepsilon)$`X` performed at each clock tick. Given a bound on the number of clock ticks (i.e. a bound on the execution time), it is possible to derive a bound on `X`. Such slowly diverging computations occur in our target programs due to the slow accumulation of floating-point rounding errors throughout the whole execution. The values ϵ and ε are very small:

```
float Q[4], R[4];
int i;
float norm =
  sqrt( Q[0] * Q[0] + Q[1] * Q[1] +
        Q[2] * Q[2] + Q[3] * Q[3] );
assert(norm >= 0.1);
for (i=0; i<4; i++)
  R[i] = Q[i] / norm;
/* ∀i, R[i] ∈ [−1 − ϵ, 1 + ϵ] */
```

Fig. 12 Quaternion normalization.

they correspond respectively to the absolute error on denormalized floating-point numbers and the relative error on normalized numbers (respectively 10^{-45} and 10^{-5} for 32-bit floating-point numbers), and so, there is generally no actual overflow, even after hours of uninterrupted computation.

Quaternion Domain. Embedded space software make a large use of quaternions, in particular in attitude control, as they provide a convenient way to represent and manipulate orientations in a three dimensional space. Quaternions are represented as four numbers (a rotation axis and an angle), and feature mathematical operators ($+$, $-$, $\times$, $/$, conjugate, norm) with natural algebraic properties. Quaternion computations are generally performed on normalized quaternions, which implies that all components are in $[-1, 1]$, avoiding any risk of overflow. However, because quaternions are implemented using floating-point arithmetics with rounding errors, the norm is never exactly one. Moreover, to avoid any drift of the quaternion components, the quaternions need to be re-normalized periodically, using for instance the algorithm presented Fig. 12. To prove that a program is free from overflow, a static analyzer must first prove that, after normalization, all components are in $[-1 - \epsilon, 1 + \epsilon]$ (where ϵ accounts for rounding errors during the normalization step), and so, must embed some knowledge of quaternion arithmetics [10]. This is quite difficult because the computations are highly non-linear, and a single quaternion operation is split into a sequence of many floating-point operations on distinct variables representing quaternion components. Thus, Astrée uses a special quaternion abstract domain to detect which tuples of variables form quaternions, track their norm, infer which quaternion operations are performed based on sequences of operations on individual components, and deduce precise bounds for all the components.

3.3.2 Reduced Product

Astrée employs dozens of numerical abstract domains, a few of the most important ones being presented above. Each program instruction is executed by each domain, so that the actual invariant inferred by the analyzer is the conjunction of the invariants found by each domain. Additionally, Astrée implements a powerful communication framework [16] which allows domains to interact and improve each other. A domain can then benefit from the information from other domains to refine its abstract element (this is called a reduction) or improve its predicate transformer. More precisely, although each abstract domain has its private representation of abstract properties, it also has access to a common, public language of predicates understood by all domains, as well as functions to request and export such predicates.

Consider, for instance, the case of interval information. It can be expressed by several domains but, for efficiency reasons, only the interval domain keeps an information on all variables at all time. Thus, when the octagon domain encounters an expression where only some of the variables are actually kept in the octagon, it needs to remove missing variables, replacing them with an interval, before processing the expression. It broadcasts a request for interval information, which is answered by the interval domain (and possibly other domains as well). Dually, when the octagon domain infers a bound on some variable that required non-trivial computations on relational properties, it broadcast this information. The interval domain, among others, will use this information to refine its abstract state and improve all its future computations.

For efficiency reasons, the predicate language is limited to a few predicates of interest by several domains, and the domains are very parsimonious in the information they communicate (a fully reduced product would not scale given the amount of information inferred by the many domains). However, the framework is easily extensible, so that new reductions can be added to the analyzer as needed.

3.3.3 Pointer Abstract Domain

Even in the absence of dynamic memory allocation, pointers and pointer arithmetics are widely used in embedded software. Astrée supports pointers using a straightforward and low-level semantics [33]: a pointer in the concrete semantics is simply a symbolic integer composed of a variable or function name (the base), and an optional integer byte displacement from the base address (the offset). A set of pointer values is then abstracted as a set of bases and a synthetic integer offset variable. Astrée implements a pointer abstract domain that is actually a domain functor taking as argument an arbitrary (reduced product of) numerical abstract domain(s) and adds support for pointers. The

pointer domain maintains the set of bases associated to each pointer, and delegates the abstraction of offsets to the underlying numerical domains. In particular, when using relational numerical domains, this allows the inference of relations between pointer offsets, and between pointer offsets and integer variables (e.g. between i and p in a loop such as `for (i=0;i<10;i++) p++;`). Likewise, pointer-related operators in expressions are handled in two steps: the pointer domain computes the effect of the operator on pointer bases to update its internal representation, and it translates pointer arithmetics into byte-based offset arithmetics to be handled by the underlying numerical domain. Delegating the abstraction of pointer offsets to numerical domains requires the addition of numerical domains specially adapted to offsets, such as the congruence domain [25] that can infer properties of the form $X \in a\mathbb{Z}+b$, necessary to express pointer alignment constraints.

3.3.4 Memory Abstract Domain

In addition to numerical and pointer variables, the C language features aggregates: structures, unions, and arrays. Astrée thus features a specific memory structure abstract domain [33] to handle these data-types. It is a functor that decomposes aggregate variables into collections of independent scalar variables, called cells, and hands them down to an underlying domain of numerical and pointer values. Given a program statement to execute, the memory domain must break down aggregate accesses into cell-level accesses (which requires some cooperation with numerical and pointer domains to evaluate array index and pointer offset expressions). Astrée cannot rely on the static type of variables to generate its cell map because, in the presence of pointer casts and type-punning, any memory location can be accessed dynamically with an arbitrary type. Thus, cells are managed dynamically and created on demand. Moreover, to handle C union types in a sound way, the memory domain must cope with cells denoting overlapping memory locations. When modifying a cell, it takes care to also update all overlapping ones, thus maintaining for underlying domains the illusion that cells are actually unrelated. Finally, the memory domain embeds some form of abstraction for arrays, which can be represented in either a field sensitive way (using distinct cells for each array element) or a field insensitive way (using a single cell representing the join of all array elements).

3.3.5 Trace Partitioning Abstract Domain

The decision tree numerical domain presented above provided a first example of disjunctive properties. It partitioned the value of a set of variables with respect to the value of a few boolean variables at the current program location. The trace partitioning domain [44] is another example, using a different partitioning criterion: it performs case analysis with respect to an abstraction of the history of computation.

```
float x,r;
const float tx[N] = { ... }, ty[N] = { ... };
assert(x >= t[0] && x <= t[N-1]);
for (i=0; i<N-1; i++)
  if (x <= tx[i+1]) break;
r = ty[i] +
    (ty[i+1]-ty[i]) * (x-tx[i]) / (tx[i+1]-tx[i]);
```

Fig. 13 Linear interpolation example.

Consider, for instance, the program Fig. 13 that implements a simple linear interpolation. First, a loop locates the input x within a sorted array tx of values and computes i such that $tx[i] \leq x \leq tx[i+1]$. Then, the interpolated value is computed as a combination of $ty[i]$ and $ty[i+1]$. Inferring a precise bound on r requires inferring the complex relation $(i = 0 \wedge t[0] \leq x \leq t[1]) \vee (i = 1 \wedge t[1] \leq x \leq t[2]) \vee \cdots$ holding before the assignment into r. The problem is, however, much simpler if, instead of looking for an invariant holding for all execution traces reaching the assignment, we perform a case analysis on the history of the computation: if the loop iterated n times, then $i = n \wedge t[n] \leq x \leq t[n+1]$. The assignment into r is then performed for each case n before the results are merged, hence achieving a limited form of path sensitivity. The trace partitioning domain formalizes this intuition, grouping execution traces according to criteria such as the number of iterations performed by a loop and which branches of tests where executed. It is implemented as a functor that lifts any abstraction of states to an abstraction of traces.

In order to scale up to programs of large size, it is important to limit the trace partitioning to small portions of code. These are computed by an automatic parametrization heuristic preceding the analysis, which performs an approximate dependency analysis of tests and assignments.

3.4 Ensuring Efficiency and Precision

A key to the efficiency of Astrée is its parsimonious and localized use of carefully limited abstract domains. First, we chose less expensive domains instead of more expressive ones (e.g octagonal invariants versus polyhedral, ellipses aligned on the first bisector versus polynomials). Moreover, expensive domains (such as octagons, decision trees, and trace partitioning) are only used on selected few variables or code portions. Finally, domains

only communicate a selected portion of the information they infer to other domains.

A key to the precision of Astrée is its design by refinement, which is made easy by its very modular design. We actually started from a simple interval-based analyzer and improved it until it reached zero alarms on a first family of realistic code [8], then considered other, more complex, codes. When encountering a new kind of codes, the analysis generally terminates quickly but with some false alarms, which must be investigated by hand to find the cause of imprecision. The analysis can then be improved in various ways, from easiest and most common, to more time consuming but thankfully seldom required. Often, it is sufficient to tune some parameters of the abstractions that are exposed through around 150 command-line options (such as iteration strategies, domain aggressiveness, pack size, etc.), which any trained end-user can do. When Astrée already contains a domain able to infer the missing information, a solution is to update the automatic parametrization of variable packs and code portions where the domain is activated — this happened, in particular, once while analyzing programs in the same application domain but with a new code generator [7]. The regularity of automatically generated code is very helpful to design robust full-scale automatic parametrization heuristics by generalisation of test cases. When the information is inferred but fails to be exploited, new reductions between domains can be added. These two cases require minor modifications to the analyzer source. When all fails, it is always possible to design and implement a new abstract domain focusing on the missing properties, but this is a research-grade activity. This last case happened when extending Astrée from aeronautic to space applications [10]: the later required a domain to handle quaternion computations, which were absent in the former application.

3.5 Applications

A first application of Astrée was the proof of absence of run-time errors in two families of industrial embedded avionic control / command software generated from SAO specifications [19]. Programs in the first family have around 100 K code lines and 10 K global variables (half of which are floats) and can be analyzed in around 2 h on a 64-bit 2.66 GHz intel workstation using a single core. Programs in the second family control more modern systems and are both more complex and larger: up to 1 M code lines. They are analyzed in 50 h. Both analyses give zero false alarm.

A second application was the analysis of space software [10] or, more precisely, a 14 K lines C code generated from a SCADE [20] model designed by Astrium ST. After a specialization required by a change of application domain and code generator, the code could be analyzed in 1 h, with zero false alarm.

Since its industrialization by AbsInt, Astrée is being adapted to handle code generated by dSPACE TargetLink (a popular code generator for MATLAB, Simulink and Stateflow) with encouraging preliminary results [32].

4 Checking Run-Time Errors in Parallel Embedded Software

Parallel programs are now ubiquitous, and modern programming languages (such as Java [23]) have been designed with built-in concurrency support, while older ones (such as C and C++) can access parallel features through the use of libraries (such as POSIX threads [28]). Embedded critical software are not immune to this trend. For instance, in the context of Integrated Modular Avionics (IMA), there is a tendency to replace a physical network of intercommunicating processors, running a single application each, with a single application running several threads in a shared memory. Achieving a good coverage when testing such systems is even more difficult than for sequential ones due to a combinatorial explosion in the number of possible interleavings of thread executions. Formal methods limiting interleavings to a fixed, small number of context switches, such as [40], can also miss bugs. On the other hand, it has been shown in [15] that Abstract Interpretation techniques could be applied to describe and abstract the semantics of parallel programs, paving the way for sound and efficient static analyses.

We now present on-going research ([7, § IV], [35]) on Thésée, an extension of Astrée that checks run-time errors in parallel programs.

4.1 Target Programs

We focus on applications for embedded real-time operating systems. Our main target is avionic applications running under an ARINC 653 operating system [3] (Aeronautical Radio Inc.'s Avionics Application Standard Software Interface). Such applications are composed of several threads that communicate implicitly through a shared memory, and explicitly through synchronisation objects provided by the OS: events, semaphores, message queues, blackboards, etc. We assume the same restrictions as in Astrée: we analyze C programs without dynamic memory allocation nor recursion. We also forbid the dynamic creation of threads

and synchronisation objects. These are created solely during an initialization phase, and are thus fixed before entering the parallel execution mode.

Another feature of our target programs is that all threads have fixed, distinct priorities. Due to the real-time nature of the operating system, thread priorities are enforced strictly: a thread that is runnable (i.e. not waiting for a semaphore or some external event) cannot be preempted by a lower priority thread — this is in contrast to desktop and server schedulers, where lower priority tasks always get to run, even when higher priority ones are not blocked. Moreover, we focus on programs where all threads are scheduled on the same execution core, excluding true parallelism. This means that the runnable thread of highest priority is always the only one to run. This property permits the implementation of a form of mutual exclusion based on priorities instead of semaphores. Note that a lower priority thread can still be preempted at any point by a higher priority thread that becomes runnable asynchronously (e.g. due to an external event or a timeout), and so, the number of possible interleavings authorized by a real-time scheduler is still very high.

We focus on proving the same properties as in Astrée: the absence of arithmetic and memory errors. Additionally, we report data-races (i.e. write / write and read / write accesses by two threads to the same shared memory location without enforcing mutual exclusion). However, we do not check deadlocks nor unbounded priority inversions; by construction, these cannot occur in our target programs (all blocking primitives have a timeout). Finally, other parallelism hazards, such as bounded priority inversions, livelocks, or starvation, which are much more complex to detect, are not considered.

4.2 Semantics and Abstractions

Our prototype, Thésée, is based on Astrée and inherits all of its abstract domains. It features also two additional domains as well as an extra iterator, which are sketched in Fig. 14 and described below.

4.2.1 Interference Semantics

For efficiency reasons, it is not possible to consider explicitly all interleavings of threads. Thus, we use an idea dating back to early proof methods for parallel programs [38], i.e. the decomposition of the global program invariant into a local invariant at each program point of each thread (similar to the kind of invariants inferred by Astrée) and a global interference property

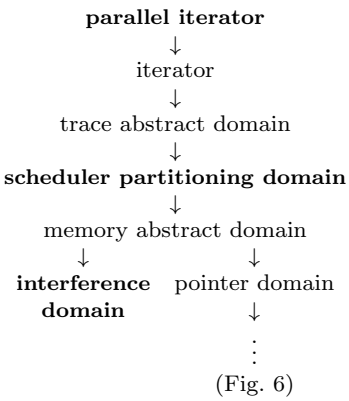


Fig. 14 Hierarchy of abstractions in Thésée. Abstractions added compared to Astrée are shown in boldface.

describing the effect on a given thread of all the other threads.

In the concrete, an interference is a triple (t, X, v), indicating that the thread t can set the variable X to the value v. A set of interferences is then abstracted by mapping each pair (t, X) to an abstract set of values (e.g. an interval). Interferences are thus non-relational and flow-insensitive. The memory abstract domain of Sec. 3.3.4 is modified to apply, when analyzing a given thread, a given abstraction of interferences to variables appearing in expressions. The modified expressions are passed down to underlying domains, that do not need further change to handle interferences. The memory domain also enriches the current thread's abstract interferences based on values assigned to variables. Given an abstract interference, it is then possible to analyse a single thread using Astrée's flow and context sensitive iterator to compute its local abstract invariants and collect new abstract interferences. To take into account the interferences between all threads in a sound way, we use a parallel iterator that reanalyzes all threads, starting from an initially empty abstract interference, until interferences stabilize. A widening $\triangledown$ is used to ensure that the parallel iterations terminate in few steps, so that the overall analysis is slower than a sequential program analysis by a small factor only.

This interference semantics is an abstraction of the interleavings of all threads, as proved in [35]. Moreover, it is also sound with respect to weakly consistent memory models [2]. In particular, it takes into account the optimizations and program transformations that are performed by processors and compilers, and that may expose behaviors not corresponding to any interleaving of threads.

4.2.2 Scheduler Partitioning

To achieve a good precision, it is also necessary to take into account the scheduler, which restricts the set of possible thread interleavings. For instance, it enforces mutual exclusion (two threads cannot lock the same semaphore) and thread priorities (a low priority thread cannot preempt a higher priority runnable thread). A concrete model of the ARINC 653 scheduler [3] that keeps track the state of threads and synchronization objects is developed in [35].

It is important to note that interesting scheduling properties, such as mutual exclusion, relate the control point of two threads. Thus, they cannot be expressed in a thread analysis that treats the effect of other threads in a flow insensitive way. Our solution is to partition the invariants (thread local invariants and thread interferences) with respect to an abstraction of the scheduler state. In particular, we distinguish the interferences performed by a thread when it owns a given lock and when it does not. When analyzing another thread, we partition the local invariant with respect to the same criterion. Then, the local invariants where the lock is owed by the analyzed thread is only influenced by interferences where the lock is not owned by the interfering thread. Another useful partitioning criterion is the ability to be preempted by another thread and involves thread priorities.

4.3 Preliminary Results

Our target parallel application is a large avionic software consisting of 1.6 M lines of C code and 15 threads. It runs under an ARINC 653 real-time OS [3]. The code is quite complex as it mixes string formatting, list sorting, network protocols, and automatically generated synchronous logic. The program was completed with a 2 500 line hand-written model of the ARINC 653 OS implementing the various API calls, in C enriched with analyzer-specific intrinsics (for locking and unlocking semaphores, etc.).

The analysis currently takes 14h on our 2.66 GHz 64-bit intel server using one core. An important result is that only four iterations are required to stabilize abstract interferences. Moreover, the number of required abstract scheduler partitions is quite low (52 partitions for interferences and 4 partitions for local invariants), and so, the analysis is memory efficient. The analysis generates around 7600 alarms. This high number is understandable: Thésée is naturally tuned for control / command software as it inherits abstract domains from Astrée, but the analyzed program is not limited to control / command processing. We plan to reduce the number of alarms in future work.

5 Translation Validation

The analysis described in Sec. 3 is performed at the source level, thus its results hold true at the assembly level only if the compilation of the C code is semantically correct. Indeed, if the compiler is incorrect, then the assembly code may contain bugs that could cause runtime errors: for instance, if an arithmetic expression is compiled in an incorrect manner, the assembly code may compute results that violate the invariants computed at the source level. In some application domains, such as avionics, certification rules state that the development process should be certified and that the final version of the software should be certified [4]. From this point of view, the analysis of the C code cannot be considered a sufficient guarantee, as the use of an incorrect compiler may ruin the whole certification effort.

Certification processes such as DO 178 B [4] also require each development step be documented, so that it can be verified correct at a later stage (either formally, or by manual inspection). From the certification point of view, compilation can also be considered a development step in itself. Therefore, certification in avionic systems puts a lot of emphasis on the verification of program translation stages.

An alternative technique would consist in certifying *only* the final code, and designing analysis tools that should be run on the compiled program. While an analysis of the target code is adequate for some applications such as worst-case execution time [26], higher level properties such as the absence of run-time errors are easier to reason about at the C level, since many simple C operations turn into complex pieces of assembly code, as in the case of floating point conversions or code in which the notion of error has disappeared (such as memory accesses). Thus, it is preferable to exploit the results obtained at the source level in order to cope with the target code certification in a more efficient way.

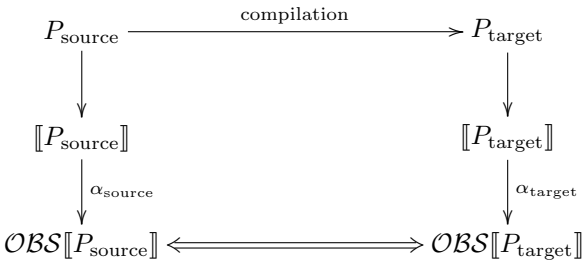
A first approach relies on the *translation of invariants* obtained in Sec. 3 into assembly level properties, which can be checked using an independent analysis. The advantage of this technique is that the low-level analysis only needs to *verify* invariants; the harder part (which consists in *inferring them*) can be carried out at the source level, using an analyzer such as Astrée. Proof carrying codes [36] were a first implementation of that idea, where properties to be tracked are related to the security of web services. It was later extended to work with Abstract Interpretation based static analyses [42]. This approach allows proving that the assembly

code enjoys some good properties, such as the absence of runtime errors. However, it does not provide a proof that the compilation is correct. This approach is also hard to implement and hard to get to scale up, as the analysis of low level operations is often complex.

A second technique proceeds by an automatic proof of equivalence (also known as *translation validation*) of both programs. This idea was initially proposed for synchronous languages [39] and later extended to languages like C [37,43]. The proof of equivalence is automatic: in [39], it is performed by a model checker whereas [43] relies on a theorem prover developed specifically for program equivalence checking. The algorithms presented in [37] and [43] proved to be scalable and extensible to rather large classes of compilation algorithms (even though they were developed for some specific compiler). This technique allows tackling both certification requirements of [4]: first, when the proof of equivalence succeeds, it means the results of the source code analysis applies to the target level; secondly, it also allows to document and verify a compilation stage in the development.

Note that translation validation is actually closely linked to the proof of absence of run-time errors at the source level: indeed, semantic equivalence can only be guaranteed on safe executions, whereas unsafe runs typically have undefined semantics. Indeed, the C 99 standard [30] leaves many behaviors undefined, when a program performs an incorrect operation. Such a situation cannot be accounted for in the equivalence checking phase, and the translation validator will always try to check that the target code behaves as specified in the source code, when the source code defines a valid behavior. Therefore, translation validation should always be run after an analysis tool like Astrée achieved a proof of absence of run-time errors in the source code.

Both techniques can be formalized in the Abstract Interpretation framework [43], since the compiler semantic effect can be modeled as a classic fixpoint transfer, where C computation steps are translated into assembly computation steps. Such a formalization allows the translation certification to interface well with the C code analysis, and also to reuse components of translation validators and invariant translators to certify the compilation of other languages, or when the compilation includes optimization stages. This formalization relies on the following principle: compilation should be expected to preserve the overall structure of programs, while modifying the representation of data, and the nature of small steps; therefore, before reasoning about compilation, one should abstract away the aspects of program behaviors that are not preserved, and focus on the "stable" part of the program semantics. This can be achieved using abstraction functions, that will take the standard semantics $[\![P_{\text{source}}]\!]$ (resp., $[\![P_{\text{target}}]\!]$) of the source program P_{source} (resp., assembly program P_{target}) into an abstract semantics $\mathcal{OBS}[\![P_{\text{source}}]\!]$ and $\mathcal{OBS}[\![P_{\text{target}}]\!]$, where only behaviors that are preserved by the compilation are observable. Typically, at the assembly level, the state at intermediate control points and the values stored in temporary registers will not be observable, as they do not correspond to anything in the source program semantics. Conversely, when a compiler determines that a variable is dead and discards it, it should be erased from the source semantics, as it will not be possible to relate it to anything in the assembly. Then, the equivalence between the "observational semantics" boils down to a simpler relation. This situation is described in the diagram below.



This diagram allows to clearly specify the compilation verification tools. An invariant translator should exploit the equivalence to translate invariants in a sound manner, thus it can only translate invariants that are more abstract than the observational semantics of choice. A translation validator should perform equivalence checking at the observational semantics level.

In general, the choice of the observational semantics is not unique. Depending on the level of granularity of the observation, the equivalence under consideration is more or less tight, which may make it more or less useful and easier or harder to verify.

The Lcertify tool was developed as a library of data-structures and algorithms for translation validation, following the framework exposed in [43]. It comes with a C front-end, but can also be used with a custom front-end, adapted to a user-specific imperative language, as used for some applications. At the assembly level, it takes 32-bit PowerPC binaries. It can certify the compilation of industrial-size codes in a few minutes including disassembly. It operates on a rather low level of observation, and will establish the correctness of the target code with respect to the source level on a per source statement basis. This makes the property it verifies very precise.

6 Imperfectly-Clocked Synchronous Systems

6.1 Proving Temporal Specifications

The analysis presented in Sec. 5 allows binaries obtained through the process explained in Fig. 1 to be proved safe with respect to specifications expressed at the C language level or at the assembly language level. This is a huge part of the control units of embedded systems which is then certified. On the other hand, the *temporal* specifications are rather expressed at a higher level, where the synchrony hypothesis and the syntax make them easier to define and read. This is also the right level for studying the redundancy added to systems in order for them to gain robustness to hardware failures. Indeed, some hardware characteristics are still present, like clocks and communication channels. As many systems are designed in a distributed way, these systems are in fact asynchronous. Having multiple clocks implies studying the desynchronizations resulting from the little imperfections of each clock. Similarly, the delays in communication channels, although usually bounded, cannot yet be considered as null or even constant, which leads to unexpected behaviors.

However, these systems are not arbitrary asynchronous systems but rather imperfectly-clocked systems. Their behaviors are expected to be close enough to those of an ideal synchronous systems in order to satisfy the desired specification. This has to be proved using the information about their physical clocks and their communication channels. For example, a typical hypothesis for an imperfect clock is that its period (the time between two consecutive clock ticks) is equal to some fixed value δ with a possible imprecision of $x\%$. This may enable the computation of the minimum and maximum number of cycles that may happen during a given time interval.

The case of redundant sub-systems is particular. The whole system is made safer by detecting the failures of one of the redundant units and by considering only the results of other equivalent units. Of course an automatic static analyzer should not trust this statement but has to prove it. We thus developed abstract domains based on the information we have from high level code (similar to UML *views*). These abstractions used during the development told us that in order to minimize disagreements between redundant units, the designer used strategies developed by the synchronous language community to *stabilize* the values (eliminating unstable ones) and perform votes on *several* cycles. We therefore do not try to detect which strategy is used in the system we analyze, yet our analyzer is designed so that it is very precise for these stabilized values and is also able to study dependencies on previous cycles. This is an example of how to extract information from the development process without trusting it excessively, since this information is only used as a hint to analyze lower level code that is actually going to be executed.

We now show how to perform this temporal analysis.

6.2 Continuous-Time Semantics

We consider a continuous-time semantics since it allows abstract domains to compute a more precise fixpoint abstracting this semantics than with a discrete semantics. This is because the mathematical properties of continuous spaces are richer than those of discrete ones, and maybe also because these systems are actually designed in a continuous world through differential equations.

In this semantics, it is for example easy to define the consequence of a small imperfection in the delay along a communication channel. If this delay belongs to the time interval $[\alpha; \beta]$ then if a message with value x is sent at time t, it is known that the value x is received between time $t + \alpha$ and $t + \beta$. This is the basis for our first Temporal Abstract Domain.

6.3 Temporal Abstract Domains

6.3.1 Abstract Constraints

By defining an *universal constraint* on a time interval $[a; b]$ and a boolean x, denoted $\forall\langle a; b\rangle : x$, we can describe signals that take the value x during the whole time interval $[a; b]$. Similarly, an *existential constraint*, denoted $\exists[a; b] : x$, describes signals that take the value x at least once during $[a; b]$. This abstract information can be propagated abstractly in an efficient way. For example, the effect of a negation operation on a $\forall\langle a; b\rangle : x$ signal turns it into $\forall\langle a; b\rangle : \neg x$. Indeed, if you compute repeatedly and instantaneously the negation of a continuous signal that is always equal to x between time a and b, the result is clearly always equal to $\neg x$ between time a and b.

In a more complex example, a communication channel transmitting information in a serial way with at least α and at most β delay, submitted with a $\exists[a; b] : x$ constraint results in $\exists[a + \alpha; b + \beta] : x$. The cost for propagating this abstract information is therefore of only two additions for each constraint, thus linear in the size of the initial set of constraints.

By considering conjunctions of these elements, we may express temporal properties. This abstract domain

thus plays a similar role as the one of the interval domain presented in Sec. 2 for the analysis of the code of one unit only.

6.3.2 Stability Domain

The *Changes Counting Domain* is the second Temporal domain. It was designed in order to discover automatically the properties of stability of some of the values in some units of the system and if possible to bound the variability of whole units of the embedded system.

For example, the data received from sensors is often stabilized in order to eliminate dubious values. The unit in charge of making the value changes smoother can for example be implemented in a synchronous language as the following node where `nmin` is a stability parameter:

```
stab (x:int)

if x unchanged since last cycle
then
   if counter c reaches nmin
   then
      return x
   else
      increment c;
      return previous stable value
 else
   initialize counter c;
   return previous stable value
```

The Changes Counting Domain has to find in an automatic and safe way that the output of the unit containing this code does not change more than once during each time interval of width `nmin`. Even if the analyzer is not extracting this value `nmin` directly from the code, this stability property is then much easier to discover in the code at this level than, for example, in the C code that could be generated from this synchronous program.

Another domain, expressing and proving additional quantitative temporal properties, such as average values, is proposed in [6].

6.3.3 Reduced Product

The temporal aspect does not only enable proving temporal properties, but also allows the automatic definition of a reduced product [14], the time becoming a *common language* between the domains.

For example, in Fig. 15, the upper part describes two abstract constraints $\exists[b;c] : x$ and $\forall[d;f] : x$. However, we assume that the time area between them is covered by an abstract value changes bound that ensures that between time a and e, at most one value change occurs, and we have $a < b < c < d < e < f$.

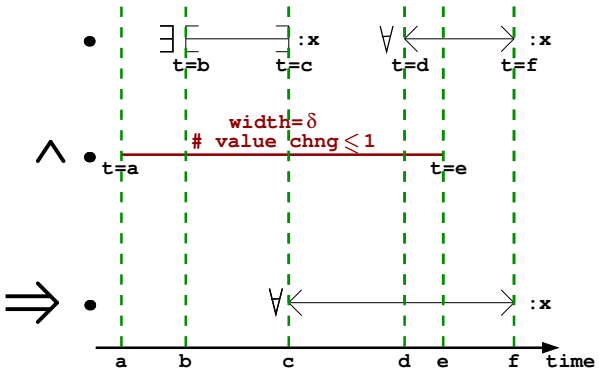


Fig. 15 Example of reduced product between temporal domains.

There are consequently only two possibilities between c and d: either there is no value change or there is one value change. Having one value change is actually not possible. Indeed, if there is at least one value change between b and c, then there are at least two, since, at some time $t \in [b;c]$, the value has to be $\neg x$, because at time d, it has to be x. Then, at some point t' between c and d, the value has to be x, which makes two value changes: one between t' and t, and one between t and d. This is excluded. As a consequence, the two constraints may be rewritten as $\forall[c;f] : x$.

The use of several temporal abstract domains thus makes the analysis more precise and more likely to prove the temporal specification.

6.4 Implementation and Experiments

Relaxing the synchrony hypothesis allows the certification of a bigger subset of the whole embedded system, at the price of a much more complex analysis that actually depends on the imprecision of the clocks and the communication systems. A prototype static analyzer has been developed according to these ideas and was able to prove some temporal properties of redundant SAO systems with a voting system arbitrating among them. Furthermore, when the analyzer cannot prove the specification, looking at the abstract fixpoint is sometimes sufficient to devise an erroneous trace. When this is not the case, it may be due false alarms that might be removed by creating more precise abstract domains. Incidentally, by trying the prototype on systems with different parameters, interesting information can be obtained, such as the minimal synchrony for the stabilization of the values read by sensors such that the specification is proved.

7 Conclusion

The development of critical embedded systems is often made following the V-Model paradigm. This paradigm states that the development can be separated into two distinct branches. The earlier starts with the definition of the purpose of the systems, turns them into specifications, and results in the writing of the low-level code that will be executed in the automated system. The later branch is assumed to be the integration of all the implemented subsystems, and the verifications and tests of the final system.

It is furthermore often suggested that a member of the development of the system belongs exclusively to one of the two branches, with the idea that if he belongs to the earlier design branch, he would validate it during the second branch. As soon as automatic and formal verifications are made, this argument cannot be considered as valid anymore. We propose however an analysis framework which is mainly based on the later branch. It is crucial to choose at which level an analysis has to be made. The reason for this may be syntactic: in low level code, some syntactic elements like variables or time, are not available, and consequently specifications on them cannot even be expressed.

Furthermore, some complex properties are difficult to prove when their purpose is not clearly known. This apparently violates the principle of separation of the two branches of V-model. Suppose, however, that the analysis designer is aware that some technique (say, digital filtering) is used by high level code in order to reach a complex property. A specialized domain, only knowing the *kind* of technique used may discover by itself that the ranges of coefficients used in the filter actually satisfy the desired property without contradicting the V-model. Similarly, a domain may implement the discovery of a proof that a property only holds if variables are stable enough in the system, and discover and prove the minimal stability for that without knowing the stability value that the designers chose.

Finally, the multiple levels of the earlier branch of development also give a chance to choose one level for the specifications of the systems and translations to other levels may be themselves validated so that the final level (binary) is certified.

Some automatic transformations have been developed and proved correct by scientists and engineers, like automatic duplication of units for a safer redundant system satisfying the same specifications as the initial system when there is no hardware failure, and still functional in case of the failure of a single unit. Yet most of these automatic transformations require hypotheses that are not checked automatically, which on big systems cannot be considered as safe. Therefore, analyzing the high level code where this automatic transformations have not been performed is crucial in proving the specifications of the final system.

The use in UML of successive as well as independent abstracted *views* of the system being designed does actually reinforce the safety of the whole development process by providing simplified yet safe overapproximations of the final system. In the same way, performing several static analyses at several levels of the development process makes the certification easier and covers more cases.

All this is however possible only if each level has a formal semantics. We are therefore very interested in the high level becoming more and more formal. It would also be of high benefit if the transformation of the code from one level to a lower one were performed automatically, since it eases the discovery of correspondences between the two levels and the translation of the abstract property found at a higher level to lower ones.

We have thus shown that static analysis by Abstract Interpretation can be applied from the design to the implementation of software systems. Each level of description of the system must be checked, and the translation from one level to another one must be validated, since the different levels can significantly differ in their description of the target system.

A modeling language like UML describes nothing more that different abstractions of the target system, at different levels of abstraction, and so, Abstract Interpretation is certainly applicable both to formalize these abstractions and to develop static analysis techniques at each level of abstraction. However, modeling languages are usually not formalized and subject to multiple, if not contradictory, interpretations. As with any formal method, the first task towards the use of Abstract Interpretation on UML would therefore be to provide a rigorous mathematical definition of the meaning of the data, business, object, and component modeling, and their diagramatic representations. It will then be possible to define in what sense modeling languages do abstract the design process and ultimately the target system. Then, the development of tools, going beyond mere syntactic checks, will be possible.

Acknowledgements We thank Isabelle Perseil for her kind invitation to the UML&FM 2010 workshop.

References

1. AbsInt, Angewandte Informatik GmbH: Astrée run-time error analyzer. http://www.absint.com/astree/

2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Comp. **29**(12), 66–76 (1996)
3. Aeronautical Radio, Inc. (ARINC): ARINC 653. `http://www.arinc.com/`
4. Technical Commission on Aviation, R.: DO-178B. Tech. rep., Software Considerations in Airborne Systems and Equipment Certification (1999)
5. von der Beeck, M.: A formal semantics of UML-RT. In: O.Nierstrasz, J. Whittle, D. Harel, G. Reggio (eds.) Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings, *LNCS*, vol. 4199, pp. 768–782. Springer (2006)
6. Bertrane, J.: Proving the properties of communicating imperfectly-clocked synchronous systems. In: K. Yi (ed.) Proceedings of the Thirteenth International Symposium on Static Analysis (SAS 06), *LNCS*, vol. 4134, pp. 370–386. Springer, Seoul (2006)
7. Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Static analysis and verification of aerospace software by abstract interpretation. In: AIAA Infotech@Aerospace (I@A 2010), AIAA-2010-3385, pp. 1–38. AIAA (American Institute of Aeronautics and Astronautics) (2010)
8. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In: T. Mogensen, D. Schmidt, I. Sudborough (eds.) The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, LNCS 2566, pp. 85–108. Springer (2002)
9. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: Proc. ACM SIGPLAN '2003 Conf. PLDI, pp. 196–207. ACM Press, San Diego (2003)
10. Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Goubault, E., Ghorbal, K., Lesens, D., Mauborgne, L., Miné, A., Putot, S., Rival, X., Turin, M.: Space software validation using abstract interpretation. In: Proc. of the Int. Space System Engineering Conference, Data Systems In Aerospace (DASIA'09), pp. 1–7. ESA publications, Istanbul, Turkey (2009)
11. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **C-35**(8) (1986)
12. Cousot, P.: The calculational design of a generic abstract interpreter. In: M. Broy, R. Steinbrüggen (eds.) Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
13. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conf. Rec. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77), pp. 238–252 (1977)
14. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 269–282. ACM Press, San Antonio, Texas (1979)
15. Cousot, P., Cousot, R.: Invariance proof methods and analysis techniques for parallel programs. In: Automatic Prog. Construction Techniques, chap. 12, pp. 243–271. Macmillan, New York, NY, USA (1984)
16. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the Astrée static analyzer. In: M. Okada, I. Satoh (eds.) Proc. of the 11th Annual Asian Computing Science Conference (ASIAN'06), *LNCS*, vol. 4435, pp. 272–300. Springer, Tokyo, Japan (2006)
17. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: The Astrée static analyzer. `http://www.astree.ens.fr`
18. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Conf. Rec. of the 5th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'78), pp. 84–97. ACM Press, Tucson, USA (1978)
19. Delmas, D., Souyris, J.: Astrée: from research to industry. In: G. Filé, H. Riis-Nielson (eds.) Proc. of the 14th Int. Static Analysis Symposium (SAS'07), *LNCS*, vol. 4634, pp. 437–451. Springer, Kongens Lyngby, Denmark (2007)
20. Esterel Technologies: Scade suite™, the standard for the development of safety-critical embedded software in the avionics industry. `http://www.esterel-technologies.com/`
21. Feret, J.: Static analysis of digital filters. In: D. Schmidt (ed.) Proc. of the 13th European Symp. on Programming Languages and Systems (ESOP'04), *LNCS*, vol. 2986, pp. 33–48. Springer (2004)
22. Feret, J.: The arithmetic-geometric progression abstract domain. In: R. Cousot (ed.) Proc. of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05), *LNCS*, vol. 3385, pp. 42–58. Springer, Paris, France (2005)
23. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, third edn. Addison Wesley (2005)
24. Goubault, E.: Static analyses of floating-point operations. In: Proc. of the 8th Int. Static Analysis Symposium (SAS'01), *LNCS*, vol. 2126, pp. 234–259. Springer (2001)
25. Granger, P.: Static analysis of arithmetical congruences. Int. J. Comput. Math. **30**(3 & 4), 165–190 (1989)
26. Heckmann, R., Ferdinand, C.: Worst-case execution time prediction by static program analysis. In: Proc. of the 18th Int. Parallel and Distributed Processing Symposium (IPDPS'04), pp. 26–30. IEEE Computer Society (2004)
27. IEEE Computer Society: IEEE standard for binary floating-point arithmetic. Tech. rep., ANSI/IEEE Std. 745-1985 (1985)
28. IEEE Computer Society, The Open Group: Portable operating system interface (POSIX) - application program interface (API) amendment 2: Threads extension (C language). Tech. rep., ANSI/IEEE Std. 1003.1c-1995 (1995)
29. Ioualalen, A.: SARDANA: an abstract interpretation based tool for Optimization of numerical expressions in LUSTRE programs. In: Tools for Automatic Program AnalysiS (TAPAS 2010), Perpignan, France (2010)
30. ISO/IEC JTC1/SC22/WG14 Working Group: C standard. Tech. Rep. 1124, ISO & IEC (2007)
31. Jeannet, B., Miné, A.: `Apron`: A library of numerical abstract domains for static analysis. In: Computer Aided Verification (CAV'09), *LNCS*, vol. 5643, pp. 661–667 (2009)
32. Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Astrée: Proving the absence of rutime errors. In: Proc. of Embedded Real-Time Software and Systems (ERTS'10), pp. 1–5. Toulouse, France (2010). (to appear)
33. Miné, A.: Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In: Proc. of the ACM SIGPLAN-SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES'06), pp. 54–63. ACM Press (2006)

34. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation **19**, 31–100 (2006)
35. Miné, A.: Static analysis of run-time errors in embedded critical parallel C programs. In: Proc. of the 20th European Symposium on Programming (ESOP'11), LNCS. Springer, Saarbrücken, Germany (2011). To appear
36. Necula, G.C.: Proof-Carrying Code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97), pp. 106–119. Paris (1997)
37. Necula, G.C., Lee, P.: The Design and Implementation of a Certifying Compiler. In: Proc. of the Conference on Programming Languages, Design and Implementation (PLDI'98), pp. 333–344. ACM Press, Montréal, Canada (1998)
38. Owicki, S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Informatica **6**(4), 319–340 (1976)
39. Pnueli, A., Shtrichman, O., Siegel, M.: Translation Validation for Synchronous Languages. In: ICALP'98, pp. 235–246. Springer-Verlag (1998)
40. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: TACAS'05, *LNCS*, vol. 3440, pp. 93–107. Springer (2005)
41. Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., Schoen, D.: Applying formal proof techniques to avionics software: A pragmatic approach. In: Proc. of the World Congress on Formal Methods (FM'99), *LNCS*, vol. 1709, pp. 1798–1815. Springer (1999)
42. Rival, X.: Abstract interpretation-based certification of assembly code. In: L.D. Zuck, P.C. Attie, A. Cortesi, S. Mukhopadhyay (eds.) VMCAI, *LNCS*, vol. 2575, pp. 41–55. Springer (2003)
43. Rival, X.: Symbolic transfer functions-based approaches to certified compilation. In: Conf. Rec. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'04), pp. 1–13. ACM Press, Venice, Italy (2004)
44. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. **29**(5) (2007)

Julien Bertrane is a Teaching Assistant (ATER) at the Computer Science Departement of the École Normale Supérieure, Paris, France. He received his PhD from École Polytechnique in France. His research focuses on Abstract Interpretation applied to the static analysis of embedded system and, in particular, to their temporal specifications. He developed Abstract Domains dedicated to temporal analysis.

PLACE PHOTO HERE

Patrick Cousot

PLACE PHOTO HERE

Radhia Cousot

Jérôme Feret is a Junior Researcher (CR1) at the French National Institute for Research in Computer Science and Control (INRIA), in the project-team "Abstraction" which is hosted at the Computer Science Department of the École Normale Supérieure, Paris, France. He received his PhD from École Polytechnique in France. His research focuses on the static analysis of programs or models by the means of Abstract Interpretation. He is interested in the certification of mobile systems and critical embedded software, and in the static analysis of biological systems.

PLACE PHOTO HERE

Laurent Mauborgne

Antoine Miné is a Junior Researcher at Centre National de la Recherche Scientifique and at the Computer Science Departement of the École Normale Supérieure, Paris, France. He received his PhD from École Polytechnique in France. His research interests include the theory of Abstract Interpretation and its applications to static analysis, with a focus on numeric properties and the safety properties of synchronous and parallel embedded critical software.

Xavier Rival is a Junior Researcher (CR1) at INRIA Rocquencourt, and at the Computer Science Department of the École Normale Supérieure, Paris, France, and a part-time Asistant Professor at École Polytechnique, France. He received his PhD from École Polytechnique in France. His reserach focuses on the static analysis of programs by Abstract Interpretation, on the design of symbolic abstract domains, and on the verification of program transformations.