# Static Analysis by Abstract Interpretation of Embedded Critical Software

Julien Bertrane ENS*
Patrick Cousot ENS* & CIMS§
Radhia Cousot CNRS & ENS*
Jérôme Feret INRIA & ENS*
Laurent Mauborgne IMDEA**
Antoine Miné CNRS & ENS*
Xavier Rival INRIA & ENS*

## Abstract

Formal methods are increasingly used to help ensuring 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 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 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 [27], an engineer (not necessarily a programmer) models a control system using the SAO graphical language, a precursor and similar tool as SCADE [14] — a Simulink™ fragment similar to SAO is also shown in Fig. 2. It is then automatically translated to the programming language C 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. Moreover, 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 to 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, 11, 5], Sec. 3) or of the precision of floating-point computations (as done by Fluctuat [17]) is done at the C level — a static analysis of real expressions at the SCADE level may however be used to determine its numerically most precise float compilation to C [20]. Likewise, neither SAO nor C make any guarantee about the worst-case execution time, so, such an analysis (as done by AbsInt's aiT [18]) is done at the binary level and for a specific processor. A SAO model can also be enriched with non-software elements, such as real-

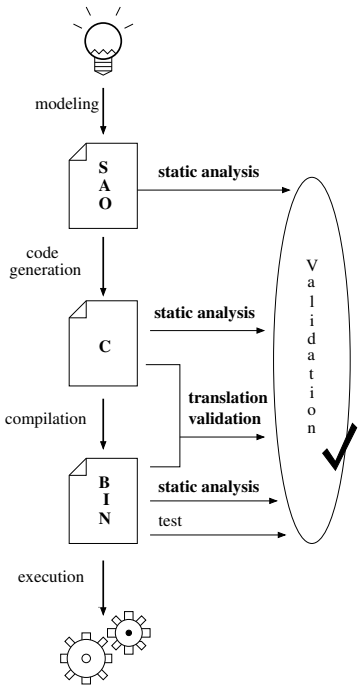


Figure 1: Example workflow for designing an embedded application.

(*)École Normale Supérieure, 45, rue d'Ulm, Paris, France, First.Last@ens.fr
(§)Courant Institute of Mathematical Sciences, NYU, New York, NY
(**)Fundación IMDEA Software, 28660-Boadilla del Monte, Madrid, Spain, laurent.mauborgne@imdea.org

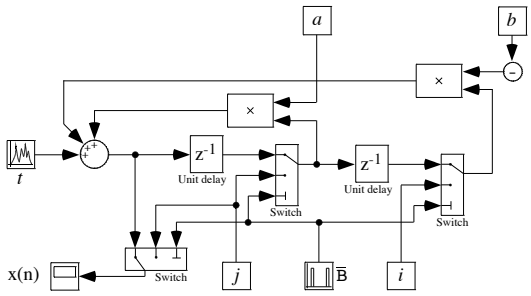


Figure 2: A second-order digital filter.

time clocks and communication lines with delays, to enable a static analysis taking time into account (Sec. 5). Finally, static analysis can also improve the confidence in compilers and code generators: translation validation (Sec. 4) can check whether the source and output 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 pose the question of whether it indeed corresponds to the concrete one. We use the Abstract Interpretation framework [8], 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 in Sec. 2, this article describes more informally several static analysis applications: checking for run-time errors with Astrée (Sec. 3), translation validation from C to assembly (Sec. 4), and analyzing communicating synchronous systems with imperfect clocks (Sec. 5). Section 6 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 [3].

## 2.1 Small-Step Operational Semantics

In order to analyze the behavior of a computer system during execution, we start by providing a model of computations, that is, an operational semantics. An example of operational semantics for UML-Statecharts is [31].

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 $T(r) = \mathbf{1}_{\mathcal{S}} \cup t \circ r \subseteq r$ 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 $s$ 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 $X \setminus Y \triangleq \{x \in X \mid x \notin Y\}$.

(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 $f(x) \sqsubseteq x$ implies $\mathbf{lfp}^{\sqsubseteq} f \sqsubseteq x$.

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 $\langle \alpha, \gamma \rangle$ 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 $\alpha_C(V) \triangleq \prod_{i=1}^n \{z \mid \exists z_1, \ldots, z_{i-1}, z_{i+1}, \ldots, z_n \in \mathbb{Z} : \langle z_1, \ldots, z_{i-1}, z, z_{i+1}, \ldots, v_n \rangle \in V\}$. The concretization is $\gamma_C(\langle V_1, \ldots, V_n \rangle) \triangleq \{\langle z_1, \ldots, z_n \rangle \mid \bigwedge_{i=1}^n z_i \in V_i\}$ 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 [8] 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 abstract relations between values $X$, $Y$, ... of numerical variables, more refined abstractions must be used such as

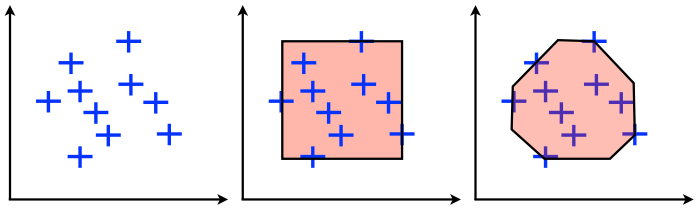

Figure 3: A set and its interval and octagonal abstractions.

octagons [25] 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:

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

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 at all to the concrete world. This follows from a general result sketched below and the observation that:

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

The above calculus also shows how to calculate the abstract transformer $F$ knowing the concrete transformer $T$,

[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 $y$.
[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 $\bigwedge_{i=1}^n (V_i \subseteq V_i')$.

which is the idea of the calculational design of static analyzers [7].

More generally, under suitable hypotheses of existence of joins in posets and fixpoints [8, 7], 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 $\alpha \circ f \mathrel{\dot\sqsubseteq} f^\sharp \circ \alpha$.[10] This is the case for example for interval analysis [8].

#### 2.4.5 Fixpoint Approximation

Under suitable hypotheses of existence of joins in posets and fixpoints [8, 7], fixpoints can be computed iteratively. For example, $\mathcal{R} = \bigcup_{i=0}^{n} F^n(\emptyset)$[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 $\triangledown$ [8] at the prejudice of precision. A naïve example of widening for intervals is $[\ell^i, h^i] \triangledown [\ell^{i+1}, h^{i+1}] \triangleq [\text{if } \ell^{i+1} < \ell^i \text{ then } -\infty \text{ else } \ell^i, \text{if } h^{i+1} > h^i \text{ then } +\infty \text{ else } \ell^i]$ 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 [8].

#### 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 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 $\alpha$ 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 $\alpha_i(X)$ [9] and standard abstractions (some already implemented in libraries [22]). 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, 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).

[10] The pointwise ordering is $f \mathrel{\dot\sqsubseteq} g$ 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 $f^{n+1} = f \circ f^n$.

# 3 Checking Run-Time Errors with Astrée

We now describe a first concrete application of Abstract Interpretation: the static analyzer Astrée [4] that checks for run-time errors in embedded C programs and has been successfully used in aeronautics [13] and aerospace [6]. It started in 2001 as an academic project [11] and is industrialized by AbsInt Angewandte Informatik [1] since 2009.

## 3.1 Semantics

### 3.1.1 Operational Semantics of C

Astrée accepts a fairly large subset of C, excluding dynamic memory allocation, recursivity, and parallelism, that are often unused (even forbidden) in embedded code. The language syntax and concrete semantics are based on the C99 norm [21], supplemented with the IEEE 754-1985 norm [19] 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).

### 3.1.2 Run-Time Errors

Astrée checks for run-time errors, that is, operations that are either undefined on the user-defined 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 and float arithmetics and casts, divisions by zero, out-of-bound array accesses, NULL, dangling, out-of-bound and misaligned pointer dereferences, and assertion failures (in calls to the `assert` C function).

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

## 3.2 Analyzed Codes

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. Once generated, such codes have the following structure:

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

(where the read, update and write instructions may be scattered in the source or encapsulated within functions) and feature a large number of global variables (the state), floating-point computations, few nested loops, and a limited number of recurring code patterns (a few macros widely instantiated).

Programs analyzed by ASTRÉE should be stand-alone, i.e., have no undefined symbols. Nevertheless, programs are run within an environment and typically fetch external data, e.g. sensor values, through memory-mapped registers. ASTRÉE allows specifying these 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.

## 3.3 Design of Astrée

ASTRÉE is an abstract interpreter: it traverses the program by structural induction on its syntax, iterating loops and stepping into functions, to collect an abstraction of all possible execution traces. The analysis is thus fully flow- and context-sensitive. Its termination is guaranteed by the use of widening operators $\triangledown$ [8] to accelerate loops and the absence of recursion.

### 3.3.1 Abstractions

ASTRÉE does not use a single abstraction, but rather many abstractions (more than 20) that can be switched on and off.

For instance, expressing the absence of many run-time errors requires information on variable bounds, and so, ASTRÉE implements the interval domain [8] based on interval arithmetics [26].[12] However, discovering precise bounds often requires stronger properties, especially for loops that requires invariant relations between variables (e.g. between the loop index and other variables used in the loop body). Hence, ASTRÉE uses relational domains, such as octagons [25] that infer relations of the form $\pm X \pm Y \leq c$, and boolean decision trees that handle disjunctions precisely — for instance $(B \wedge X > 1) \vee (\neg B \wedge X < -1)$.

Beside these classic, general-purpose domains, ASTRÉE features domain-specific abstractions, tailored to its use in control/command software. Examples include a domain to handle digital filters [15], as presented in Fig. 2, a domain of geometric-arithmetic deviations [16] $X \leq \alpha(1+\beta)^{\mathtt{clock}}$ used to bound the accumulation of floating-point rounding errors in the main loop, and a domain to handle quaternion computations.[13]

In addition to numerical domains, a specific memory domain [24] has been developed to abstract structures and arrays in a field-sensitive way, and to handle pointers and type-unsafe C constructions (such as type-punning, physical casts, or `union` types). Finally, an abstraction of execution traces [29] partitions the reachable states with respect to the history of computation and permits some path-sensitivity.

The various abstractions are independent modules, each with its own private abstract representation of properties and algorithms, that communicate using a sophisticated communication network [10] approximating the classic notion of reduced product [9].

### 3.3.2 Ensuring Precision and Efficiency

A key to the efficiency of ASTRÉE is its parsimonious and localized use of carefully limited abstract domains. One example is the use of octagons [25] which are less expressive but cheaper than general polyhedra [12] (cubic versus exponential cost). Moreover, relational domains, such as octagons and boolean trees, do not relate all variables together, but are limited to small packs of variables. Likewise, trace partitioning is only performed on limited portions of the source code, after which all partitions are merged together. Code portions and variable packs are determined automatically by simple, syntax-directed pre-analyses. Finally, domains only communicate a selected portion of the information they infer to other domains — a fully reduced product would not scale up given the amount of information inferred by the many 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 [4], 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 most 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 pre-analysis 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 [3]. The regularity of automatically generated code is very helpful to design robust full-scale pre-analyses by generalisation of test cases. When

[12] However interval arithmetics is used statically (at compile-time) with a widening to accelerate convergence of iterative fixpoint computations, not at all dynamically (at run-time) as in the traditional use of interval arithmetics to evaluate expressions of numerical programs.

[13] In mechanics, quaternions extend the complex numbers to designate rotations in the 3-dimensional space.

the information is inferred but fails to be exploited, new communication channels between domains can be opened. 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 [6]: the later required a domain to handle quaternion computations, which were absent in the former application.

### 3.4 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 [13]. 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 longer: up to 1 M code lines. They are analyzed in 50 h. Both analyses give zero false alarms.

A second application was the analysis of space software [6] or, more precisely, a 14 K lines C code generated from a SCADE [14] model designed by Astrium ST. After 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 [23].

## 4 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. 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 [30]. 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.

While analysis of the target code is adequate for some applications such as worst-case execution time [18], higher level properties such as 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 is the case for 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, yet that approach is hard to scale, 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. Note that translation validation is 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.

Both techniques can be formalized in the Abstract Interpretation framework [28], since the compiler semantic effect reduces to a classic fixpoint transfer, where C computation steps are translated into assembly computation steps. Such a formalization allows for the translation certification to interface well with the C code analysis.

The Lcertify tool was developed as a library of data-structures and algorithms for translation validation, following the framework exposed in [28]. It comes with a C front-end, but it 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.

## 5 Imperfectly-Clocked Synchronous Systems

### 5.1 Proving Temporal Specifications

The analysis presented in Sec. 4 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 are then certified. On the other hand, the temporal specifications are rather expressed at SAO level, where the synchronous hypothesis and the syntax make them easier to define and read. As many systems are designed in a distributed way, this implies studying multi-clock, inherently asynchronous systems. These systems are luckily not arbitrary asynchronous systems but rather imperfectly-clocked, meaning that we have some information about their physical clocks and their communication channels that may be enough to prove the desired specification. A typical specification for a clock is that its period (the time between two consecutive clock ticks) is equal to some fixed value $\delta$ with a possible imprecision of $x\%$.

A particular case is redundant sub-systems. 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. The synchronous language community proposed safe ways to handle this case, we thus developed abstract domains that can be very precise on the code produced following these guidelines.

### 5.2 Continuous-Time Semantics and Temporal Abstract Domains

We consider a continuous-time semantics since it allows abstract domains to compute a precise fixpoint abstracting this 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.

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 $\forall\langle a; b\rangle : \neg x$. In a more complex way, yet at the cost of only two additions, a communication channel transmitting information in a serial way with at least $\alpha$ and at most $\beta$ delay, submitted with a $\exists[a; b] : x$ constraint results in $\exists[a + \alpha; b + \beta] : x$.

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.

Additional domains on time properties were defined in [2]. This temporal aspect does not only enable proving temporal properties, but also allows the automatic definition of a reduced product [9], the time becoming a *common language* between them.

### 5.3 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.

## 6 Conclusion

We have 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 but 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.

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

## References

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

[2] Bertrane, J. Proving the properties of communicating imperfectly-clocked synchronous systems. In *Proceedings of the Thirteenth International Symposium on Static Analysis (SAS 06)* (Seoul, 29–31 Aug. 2006), K. Yi, Ed., vol. 4134 of *LNCS*, Springer, pp. 370–386.

[3] Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., and Rival, X. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace (I@A 2010)*, AIAA-2010-3385. AIAA (American Institute of Aeronautics and Astronautics), Apr. 2010, pp. 1–38.

[4] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, invited chapter. In *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, T. Mogensen, D. Schmidt, and I. Sudborough, Eds., LNCS 2566. Springer, 2002, pp. 85–108.

[5] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. A static analyzer for large safety-critical software. In *Proc. ACM SIGPLAN '2003 Conf. PLDI* (San Diego, 2003), ACM Press, pp. 196–207.

[6] Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Goubault, E., Ghorbal, K., Lesens, D., Mauborgne, L., Miné, A., Putot, S., Rival, X., and Turin, M. Space software validation using abstract interpretation. In *Proc. of the Int. Space System Engineering Conference, Data Systems In Aerospace (DASIA'09)* (Istanbul, Turkey, 26–29 May 2009), ESA publications, pp. 1–7.

[7] Cousot, P. The calculational design of a generic abstract interpreter. In *Calculational System Design*, M. Broy and R. Steinbrüggen, Eds. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[8] Cousot, P., and 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)* (Jan. 1977), pp. 238–252.

[9] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, 1979), ACM Press, pp. 269–282.

[10] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. Combination of abstractions in the Astrée static analyzer. In *Proc. of the 11th Annual Asian Computing Science Conference (ASIAN'06)* (Tokyo, Japan, 6–8 Dec. 2006), M. Okada and I. Satoh, Eds., vol. 4435 of *LNCS*, Springer, pp. 272–300.

[11] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., and Rival, X. The Astrée static analyzer. `http://www.astree.ens.fr`.

[12] Cousot, P., and 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)* (Tucson, USA, 1978), ACM Press, pp. 84–97.

[13] Delmas, D., and Souyris, J. Astrée: from research to industry. In *Proc. of the 14th Int. Static Analysis Symposium (SAS'07)*, G. Filé and H. Riis-Nielson, Eds., vol. 4634 of *LNCS*. Springer, Kongens Lyngby, Denmark, 22–24 Aug. 2007, pp. 437–451.

[14] Esterel Technologies. Scade suite™, the standard for the development of safety-critical embedded software in the avionics industry. `http://www.esterel-technologies.com/`.

[15] Feret, J. Static analysis of digital filters. In *Proc. of the 13th European Symp. on Programming Languages and Systems (ESOP'04)* (27 Mar.–4 Apr. 2004), D. Schmidt, Ed., vol. 2986 of *LNCS*, Springer, pp. 33–48.

[16] Feret, J. The arithmetic-geometric progression abstract domain. In *Proc. of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'05)* (Paris, France, 17–19 Jan. 2005), R. Cousot, Ed., vol. 3385 of *LNCS*, Springer, pp. 42–58.

[17] Goubault, E. Static analyses of floating-point operations. In *Proc. of the 8th Int. Static Analysis Symposium (SAS'01)* (2001), vol. 2126 of *LNCS*, Springer, pp. 234–259.

[18] Heckmann, R., and Ferdinand, C. Worst-case execution time prediction by static program analysis. In *Proc. of the 18th Int. Parallel and Distributed Processing Symposium (IPDPS'04)* (2004), IEEE Computer Society, pp. 26–30.

[19] IEEE Computer Society. IEEE standard for binary floating-point arithmetic. Tech. rep., ANSI/IEEE Std. 745-1985, 1985.

[20] 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* (17 Sep. 2010).

[21] ISO/IEC JTC1/SC22/WG14 Working Group. C standard. Tech. Rep. 1124, ISO & IEC, 2007.

[22] Jeannet, B., and Miné, A. `Apron`: A library of numerical abstract domains for static analysis. *Computer Aided Verification, CAV'2009 5643 of LNCS* (2009), 661–667.

[23] Kästner, D., Wilhelm, S., Nenova, S., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., and Rival, X. Astrée: Proving the absence of rutime errors. In *Proc. of Embedded Real-Time Software and Systems (ERTS'10)* (Toulouse, France, May 2010), pp. 1–5. (to appear).

[24] 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)* (June 2006), ACM Press, pp. 54–63.

[25] Miné, A. The octagon abstract domain. *Higher-Order and Symbolic Computation 19* (2006), 31–100.

[26] Moore, R. E. *Interval Analysis*. Prentice Hall, Englewood Cliffs N. J., USA, 1966.

[27] Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., and Schoen, D. Applying formal proof techniques to avionics software: A pragmatic approach. In *Proc. of the World Congress on Formal Methods (FM'99)* (1999), vol. 1709 of *LNCS*, Springer, pp. 1798–1815.

[28] 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)* (Venice, Italy, Jan. 2004), ACM Press, pp. 1–13.

[29] Rival, X., and Mauborgne, L. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst. 29*, 5 (2007).

[30] Technical Commission on Aviation, R. DO-178B. Tech. rep., Software Considerations in Airborne Systems and Equipment Certification, 1999.

[31] von der Beeck, M. A formal semantics of uml-rt. In *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings* (2006), O.Nierstrasz, J. Whittle, D. Harel, and G. Reggio, Eds., vol. 4199 of *LNCS*, Springer, pp. 768–782.