Static Analysis by Abstract Interpretation of Embedded Critical Software

Julien Bertrane  
ENS, Julien.bertrane@ens.fr

Patrick Cousot  
ENS & CIMS, Patrick.Cousot@ens.fr

Radhia Cousot  
CNRS & ENS, Radhia.Cousot@ens.fr

Järôme Feret  
INRIA & ENS, Järôme.Feret@ens.fr

Laurent Mauborgne  
IMDEA laurent.mauborgne@imdea.org

Antoine Miné  
CNRS & ENS, Antoine.Miné@ens.fr

Xavier Rival  
INRIA & ENS, Xavier.Rival@ens.fr

Sémantics and Abstract Interpreation team

November 16th, 2010
Fig. 1: Example workflow for designing an embedded application.

Semantics and Abstract Interpretation team

Static Analysis by A. I. of Embedded Critical Software
Which level should be statically analyzed?

- Static Analysis can be applied at many levels:
  - machine-readable specification
  - program source
  - binary

Static Analysis of high level, pros:
- purer information
- feedback easier
- has information on hardware (imperfections)
- de-synchronization analysis (made at Modeling level)

Static Analysis of high level, cons:
- some aspects of computations abstracted (real arithmetics VS actual implementation)
- numeric overflows analysis (made at C level)
- precision of floating-point computations analysis (made at C level)
- worst case execution time analysis (made at binary level)
Which level should be statically analyzed?

- Static Analysis can be applied at many levels:
  - machine-readable specification
  - program source
  - binary

- Static Analysis of high level, pros:
  - purer information
  - feedback easier
  - has information on hardware (imperfections)
    - de-synchronization analysis (made at Modeling level)
Which level should be statically analyzed?

- Static Analysis can be applied at many levels:
  - machine-readable specification
  - program source
  - binary

- Static Analysis of high level, **pros**:
  - purer information
  - feedback easier
  - has information on hardware (imperfections)
    - de-synchronization analysis (made at Modeling level)

- Static Analysis of high level, **cons**:
  - some aspects of computations abstracted (real arithmetics VS actual implementation)
Which level should be statically analyzed?

- Static Analysis can be applied at many levels:
  - machine-readable specification
  - program source
  - binary

- Static Analysis of high level, **pros**:
  - purer information
  - feedback easier
  - has information on hardware (imperfections)
    - de-synchronization analysis (made at Modeling level)

- Static Analysis of high level, **cons**:
  - some aspects of computations abstracted (real arithmetics VS actual implementation)
    - numeric overflows analysis (made at C level)
    - precision of floating-point computations analysis (made at C level)
    - worst case execution time analysis (made at binary level)
Static Analysis and Abstract Interpretation

Static analyzers should extract automatically properties.

Difficulties:
- most interesting properties are undecidable

Solutions:
- the analyzer explores machine-representable supersets of actual behaviors
Static Analysis and Abstract Interpretation

Static analyzers should extract automatically properties.

Difficulties:
- most interesting properties are undecidable
- the analyzer may consider spurious behaviors

Solutions:
- the analyzer explores machine-representable superset of actual behaviors
- refine the analysis, it is always sound
Static Analysis and Abstract Interpretation

Static analyzers should extract automatically properties.

Difficulties:
- most interesting properties are undecidable
- the analyzer may consider spurious behaviors
- what about errors in interpreting the specifications or the behavior of the code

Solutions:
- the analyzer explores machine-representable superset of actual behaviors
- refine the analysis, it is always sound
- work is done directly on the concrete system (i.e. input of compilers or code generators)
Static Analysis and Abstract Interpretation

Static analyzers should extract automatically properties.

Difficulties:

▶ most interesting properties are undecidable
▶ the analyzer may consider spurious behaviors
▶ what about errors in interpreting the specifications or the behavior of the code

Solutions:

▶ the analyzer explores machine-representable supersets of actual behaviors
▶ refine the analysis, it is always sound
▶ work is done directly on the concrete system (i.e. input of compilers or code generators)

Abstract Interpretation framework:

▶ an analyzer focuses on a subset of properties and programs
▶ growing library of abstraction domains
▶ modularity of domains or close cooperation between them
Semantics and Specifications

Semantics:
- Semantics defined for each primitive

Specifications:
- We consider a set of bad states $\epsilon$ that shouldn't be reached.
- So $\text{lfp}(T) \cap \epsilon = \emptyset$
Semantics and Specifications

Semantics:

- Semantics defined for each primitive
- We focus on safety properties
Semantics and Specifications

Semantics:

- Semantics defined for each primitive
- We focus on safety properties
- The set of reachable states is therefore enough

Specifications:

- We consider a set of bad states $\varepsilon$ that shouldn't be reached.
- So $\text{lfp} \ T \cap \varepsilon = \emptyset$
Semantics and Specifications

Semantics:

- Semantics defined for each primitive
- We focus on safety properties
- The set of reachable states is therefore enough
- Theoretically computable as a fixpoint of an operator $T$ summarizing all the effects of primitives used in the program
Semantics and Specifications

Semantics:

- Semantics defined for each primitive
- We focus on safety properties
- The set of reachable states is therefore enough
- Theoretically computable as a fixpoint of an operator $T$ summarizing all the effects of primitives used in the program

Specifications:

- We consider a set of bad states $\varepsilon$ that shouldn’t be reached.
Semantics:

- Semantics defined for each primitive
- We focus on safety properties
- The set of reachable states is therefore enough
- Theoretically computable as a fixpoint of an operator $T$ summarizing all the effects of primitives used in the program

Specifications:

- We consider a set of bad states $\varepsilon$ that shouldn’t be reached.
- So $\text{lfp}^T \cap \varepsilon = \emptyset$
We abstract a set \( V \subseteq \mathbb{Z} \) of integers by the interval
\[
\alpha_i(V) \triangleq [\min V, \max V]
\]
Example of Abstract Domain: Intervals

- We abstract a set \( V \subseteq \mathbb{Z} \) of integers by the interval \( \alpha_i(V) \triangleq [\min V, \max V] \)

- this is an over-approximation
Example of Abstract Domain: Intervals

- We abstract a set \( V \subseteq \mathbb{Z} \) of integers by the interval
  \[ \alpha_i(V) \triangleq [\min V, \max V] \]
- this is an over-approximation
- \( z \notin \alpha_i(V) \Rightarrow z \notin V \)
Example of Abstract Domain: Intervals

- We abstract a set $V \subseteq \mathbb{Z}$ of integers by the interval $\alpha_i(V) \triangleq [\min V, \max V]$
- this is an over-approximation
- $z \notin \alpha_i(V) \implies z \notin V$
- $z \in \alpha_i(V) \nRightarrow z \in V$

Semantics and Abstract Interpretation team

Static Analysis by A. I. of Embedded Critical Software
Example of Abstract Domain: Intervals

- We abstract a set $V \subseteq \mathbb{Z}$ of integers by the interval $\alpha_i(V) \triangleq [\min V, \max V]$
- this is an over-approximation
- $z \notin \alpha_i(V) \Rightarrow z \notin V$
- $z \in \alpha_i(V) \not\Rightarrow z \in V$
- Concretization function $\gamma_i([\ell, h]) \triangleq \{z \in \mathbb{Z} | \ell \leq z \leq h\}.$
We abstract a set $V \subseteq \mathbb{Z}$ of integers by the interval $\alpha_i(V) \triangleq [\min V, \max V]$.

This is an over-approximation:

- $z \notin \alpha_i(V) \Rightarrow z \notin V$
- $z \in \alpha_i(V) \nRightarrow z \in V$

Concretization function $\gamma_i([\ell, h]) \triangleq \{z \in \mathbb{Z} | \ell \leq z \leq h\}$.

For all $V \in \wp(\mathbb{Z}) : \forall [\ell, h] \in V^\#: \alpha_i(V) \subseteq [\ell, h] \iff V \subseteq \gamma_i([\ell, h])$

And so, by definition, the pair $\langle \alpha, \gamma \rangle$ is a Galois connection.

$$\langle \wp(\mathbb{Z}), \subseteq \rangle \xleftarrow{\gamma_i} \langle V^\#, \subseteq \rangle \xrightarrow{\alpha_i} \langle \wp(\mathbb{Z}), \subseteq \rangle$$
Intermediate goal: $\alpha(\text{lfp} \preceq T) = A$ with $\gamma(A) \cap \varepsilon = \emptyset$
Intermediate goal: \( \alpha(\text{lfp} \preceq T) = A \) with \( \gamma(A) \cap \varepsilon = \emptyset \)

\( \alpha(\text{lfp} \preceq T) \) is often non-computable
Intermediate goal: $\alpha(\text{lfp} \preceq T) = A$ with $\gamma(A) \cap \varepsilon = \emptyset$.

$\alpha(\text{lfp} \preceq T)$ is often non-computable.

However, if $\alpha \circ T \sqsubseteq T^\# \circ \alpha$, then $\alpha(\text{lfp} \preceq T) \sqsubseteq \text{lfp} \sqsubseteq T^\#$. 
Intermediate goal: $\alpha(\text{lfp} \preceq T) = A$ with $\gamma(A) \cap \varepsilon = \emptyset$

$\alpha(\text{lfp} \preceq T)$ is often non-computable

However, if $\alpha \circ T \subseteq T^\# \circ \alpha$, then $\alpha(\text{lfp} \preceq T) \subseteq \text{lfp} \sqsubseteq T^\#$.

New goal: $\gamma(\alpha(\text{lfp} \preceq T)) \cap \varepsilon = \emptyset$. 
The iterates of the $T^\#$ operation converge to the fixpoint:

- but maybe in infinitely many iterations
Abstract Fixpoint Approximation

The iterates of the $T^\#$ operation converge to the fixpoint:

- but maybe in infinitely many iterations
- and at a combinatorial time and memory cost
The iterates of the $T^\#$ operation converge to the fixpoint:

- but maybe in infinitely many iterations
- and at a combinatorial time and memory cost
- convergence has to be accelerated using a widening $\nabla$
The iterates of the $T^\#$ operation converge to the fixpoint:
- but maybe in infinitely many iterations
- and at a combinatorial time and memory cost
- convergence has to be accelerated using a widening $\nabla$
- A naïve example of widening for intervals is

$$[\ell^i, h^i] \nabla [\ell^{i+1}, h^{i+1}]$$

$$\triangleq [\text{if } \ell^{i+1} < \ell^i \text{ then } -\infty \text{ else } \ell^i, \text{ if } h^{i+1} > h^i \text{ then } +\infty \text{ else } \ell^i]$$
Semantics:

- **ASTRÉE** input written in large subset of C, *(not dynamic memory allocation, recursivity, and parallelism)*

Properties proved:

- 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,
- Assertion failures (in calls to the `assert` C function).
ASTREE

Semantics:

- **ASTRÉE** input written in large subset of C, *(not dynamic memory allocation, recursivity, and parallelism)*
- syntax and semantics based on the C99 norm, supplemented with the IEEE 754-1985 norm

Properties proved:
- run-time errors
  - 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,
- assertion failures (in calls to the `assert` C function).
ASTREE

Semantics:

- **ASTRÉE** input written in large subset of C, *(not dynamic memory allocation, recursivity, and parallelism)*
- syntax and semantics based on the C99 norm, supplemented with the IEEE 754-1985 norm

**Properties proved**: run-time errors

- 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,
- assertion failures (in calls to the `assert` C function).
Semantics:

- **ASTRÉE** input written in large subset of C, *(not dynamic memory allocation, recursivity, and parallelism)*

- Syntax and semantics based on the C99 norm, supplemented with the IEEE 754-1985 norm

Properties proved: run-time errors

- Overflows in unsigned and signed integer and float arithmetics and casts

- Divisions by zero
Semantics:

- **ASTRÉE** input written in large subset of C, *(not dynamic memory allocation, recursivity, and parallelism)*
- Syntax and semantics based on the C99 norm, supplemented with the IEEE 754-1985 norm

Properties proved: run-time errors

- Overflows in unsigned and signed integer and float arithmetics and casts
- Divisions by zero
- Out-of-bound array accesses
Semantics:

- **ASTRÉE** input written in large subset of C, \textit{(not dynamic memory allocation, recursivity, and parallelism)}
- Syntax and semantics based on the C99 norm, supplemented with the IEEE 754-1985 norm

Properties proved: run-time errors

- 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,
Semantics:

- **ASTRÉE** input written in large subset of C, \((not\) dynamic memory allocation, recursivity, and parallelism)
- Syntax and semantics based on the C99 norm, supplemented with the IEEE 754-1985 norm

Properties proved: run-time errors

- 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,
- Assertion failures (in calls to the assert C function).
Analyzed Codes

▶ **ASTRÉE** focuses on analyzing control/command synchronous programs automatically generated from Modeling Languages.
Analyzed Codes

- **ASTRÉE** focuses on analyzing control/command synchronous programs automatically generated from Modeling Languages.
- Communication by “volatile” memory locations allowed.

### Space software
- 14K lines C code generated from a SCADE model designed by Astrium ST.

### Aeronautics
- 100K code lines and 10K global variables — half of which are floats in around 2h up to 1M code lines analyzed in 50h.

### dSPACE TargetLink
- ASTRÉE now handles code generated by dSPACE TargetLink (code generator for MATLAB, Simulink and Stateflow) added by AbsInt.
Analyzed Codes

- ASTRÉE focuses on analyzing control/command synchronous programs automatically generated from Modeling Languages.
- Communication by "volatile" memory locations allowed.
- Proof of absence of run-time errors in 2 families of industrial embedded avionic control/command software generated from high level specifications:
  - Aeronautics: 100 K code lines and 10 K global variables — half of which are floats in around 2 h up to 1 M code lines analyzed in 50 h.
Astrée focuses on analyzing control/command synchronous programs automatically generated from Modeling Languages.

- communication by “volatile” memory locations allowed
- proof of absence of run-time errors in 2 families of industrial embedded avionic control/command software generated from high level specifications:
  - aeronautics 100 K code lines and 10 K global variables — half of which are floats in around 2 h up to 1 M code lines analyzed in 50 h
  - space software 14 K lines C code generated from a SCADE model designed by Astrium ST.
Astrée focuses on analyzing control/command synchronous programs automatically generated from Modeling Languages.

communication by “volatile” memory locations allowed

proof of absence of run-time errors in 2 families of industrial embedded avionic control/command software generated from high level specifications:

- aeronautics 100 K code lines and 10 K global variables — half of which are floats in around 2 h up to 1 M code lines analyzed in 50 h
- space software 14 K lines C code generated from a SCADE model designed by Astrium ST.

Astrée now handle code generated by dSPACE TargetLink (code generator for MATLAB, Simulink and Stateflow) (added by AbsInt).
relational domains bring fine-tuned preciseness (more precise than intervals)

at a bounded computational cost.
Imperfectly-Clocked Synchronous Systems
Imperfectly-Clocked Synchronous Systems

modeling → UML → static analysis

code generation → C → static analysis

validation

test
translation

compilation → BIN → static analysis

execution

static analysis by A. I. of Embedded Critical Software
Imperfectly-Clocked Synchronous Systems
A new semantics

This non-standard semantics: **continuous-time**
- allows a **more precise modeling** of reality
  - imperfect clocks
  - communication channels with unknown latency
- reuses **continuous theories**
  - integral theory
  - directed homology
- allows a **precise and efficient** static analysis
Imperfectly-Clocked Synchronous Systems

1st temporal abstract domain : constraints

express many local temporal properties
and prove some of these properties

Imperfectly-Clocked Synchronous Systems

More temporels abstract domains

value changes counting

integral boundings

▶ express stability specifications.

▶ express quantitative properties (average value, ...)

Reduce product Constraints - Value changes counting

\[ \text{width} = \delta \]

\[ \# \text{ value change} \leq 1 \]
Reduce product Constraints - Value changes counting

\[ \exists \left[ \quad \right] : x \quad \forall \leftrightarrow : x \]

\[ \land \quad \bullet \]

\text{width}=\delta

\# \text{ value chng} \leq 1

\[ \Rightarrow \quad \bullet \]

\[ \forall \leftrightarrow : x \]
Abstract Interpretation is able to define a static analysis at several levels of the development of embedded systems. It may help designers from early stages to product shipping. It may even check that the translation from one level to another is correct. Static analysis community can only benefit from a better formalization of different layers, as proposed by UML.