Static Analysis and Verification of Aerospace Software by Abstract Interpretation

Julien Bertrane
Département d’informatique, École normale supérieure

Patrick Cousot
Département d’informatique, École normale supérieure & Courant Institute of Mathematical Sciences, New York University

Radhia Cousot
CNRS & Département d’informatique, École normale supérieure

Jérôme Feret
INRIA & Département d’informatique, École normale supérieure

Laurent Mauborgne
AbsInt Angewandte Informatik

Antoine Miné
Sorbonne University, University Pierre and Marie Curie, CNRS, LIP6

Xavier Rival
INRIA & Département d’informatique, École normale supérieure

Full text available at: http://dx.doi.org/10.1561/2500000002
Editorial Scope

Topics

Foundations and Trends® in Programming Languages publishes survey and tutorial articles in the following topics:

- Abstract interpretation
- Compilation and interpretation techniques
- Domain specific languages
- Formal semantics, including lambda calculi, process calculi, and process algebra
- Language paradigms
- Mechanical proof checking
- Memory management
- Partial evaluation
- Program logic
- Programming language implementation
- Programming language security
- Programming languages for concurrency
- Programming languages for parallelism
- Program synthesis
- Program transformations and optimizations
- Program verification
- Runtime techniques for programming languages
- Software model checking
- Static and dynamic program analysis
- Type theory and type systems

Information for Librarians

Foundations and Trends® in Programming Languages, 2015, Volume 2, 4 issues. ISSN paper version 2325-1107, ISSN online version 2325-1131. Also available as a combined paper and online subscription.
Static Analysis and Verification of Aerospace Software by Abstract Interpretation

Julien Bertrane  
Département d’informatique, École normale supérieure

Patrick Cousot  
Département d’informatique, École normale supérieure &  
Courant Institute of Mathematical Sciences, New York University

Radhia Cousot  
CNRS & Département d’informatique, École normale supérieure

Jérôme Feret  
INRIA & Département d’informatique, École normale supérieure

Laurent Mauborgne  
AbsInt Angewandte Informatik

Antoine Miné  
Sorbonne University, University Pierre and Marie Curie, CNRS, LIP6

Xavier Rival  
INRIA & Département d’informatique, École normale supérieure
Contents

Nomenclature 2

1 Introduction 3

2 Theoretical Background on Abstract Interpretation 6
  2.1 Semantics ........................................ 8
  2.2 Collecting semantics ............................... 10
  2.3 Fixpoint semantics ................................. 12
  2.4 Abstraction functions .............................. 14
  2.5 Concretization functions .......................... 15
  2.6 Galois connections ................................ 16
  2.7 The lattice of abstractions ........................ 18
  2.8 Sound (and complete) abstract semantics ......... 19
  2.9 Abstract transformers .............................. 20
  2.10 Sound abstract fixpoint semantics ............... 22
  2.11 Sound and complete abstract fixpoints semantics 22
  2.12 Infinite abstraction example: interval abstraction 23
  2.13 Abstract domains and functions .................. 24
  2.14 Convergence acceleration by extrapolation and interpolation 25
  2.15 Combination of abstract domains .................. 27
  2.16 Partitioning abstractions ........................ 31

Full text available at: http://dx.doi.org/10.1561/2500000002
2.17 Static analysis .................................................. 32
2.18 Abstract specifications ........................................ 33
2.19 Verification ....................................................... 33
2.20 Verification in the abstract ................................. 33

3 Verification of Synchronous Control/Command Programs 35
3.1 Analyzed C subset ................................................. 35
3.2 Operational semantics of C ................................. 36
3.3 Analysis examples .............................................. 38
3.4 Flow- and context-sensitive abstractions ................. 42
3.5 Hierarchy of parameterized abstractions ................. 42
3.6 Trace abstraction ................................................. 43
3.7 Memory abstraction ............................................ 47
3.8 Pointer abstraction ............................................. 51
3.9 General-purpose numerical abstractions ................. 53
3.10 Domain-specific numerical abstractions ................. 65
3.11 Combination of abstractions ............................... 72
3.12 Abstract iterator .............................................. 76
3.13 Analysis parameters .......................................... 78
3.14 Application to aeronautic industry ....................... 79
3.15 Application to space industry .............................. 82
3.16 Industrialization ................................................ 82

4 Verification of Imperfectly-Clocked Synchronous Programs 84
4.1 Motivation ........................................................ 84
4.2 Syntax and semantics ......................................... 85
4.3 Abstraction ...................................................... 88
4.4 Temporal abstract domains .................................. 89
4.5 Application to redundant systems ........................ 90

5 Verification of Target Programs .............................. 92
5.1 Verification requirements and compilation ............... 92
5.2 Semantics of compilation ..................................... 93
5.3 Invariant translation applied to target level verification .................................................. 94
5.4 Compilation verification ....................................... 95
6 Verification of Parallel Programs 97
  6.1 Considered programs 97
  6.2 Program example 98
  6.3 Concrete collecting semantics 100
  6.4 Abstractions 103
  6.5 Preliminary application to aeronautic industry 107

7 Conclusion 109

References 112
Abstract

We discuss the principles of static analysis by abstract interpretation and report on the automatic verification of the absence of runtime errors in large embedded aerospace software by static analysis based on abstract interpretation. The first industrial applications concerned synchronous control/command software in open loop. Recent advances consider imperfectly synchronous programs, parallel programs, and target code validation as well. Future research directions on abstract interpretation are also discussed in the context of aerospace software.
Nomenclature

$1_S$ identity on $S$ (also $t^0$)
\( t \circ r \) composition of relations $t$ and $r$
\( t^n \) powers of relation $t$
\( t^* \) reflexive transitive closure of relation $t$
\( \text{lfp} \subseteq F \) least fixpoint of $F$ for $\subseteq$
$\wp(S)$ parts of set $S$ (also $2^S$)
\( x \) program variable
\( \mathbb{V} \) set of all program variables
\( S \) program states
\( I \) initial states
\( t \) state transition relation
\( C[t] \) collecting semantics
\( T[t] \) trace semantics
\( R[t] \) reachability semantics
\( P[t] \) prefix trace semantics
$F_P$ prefix trace transformer
$F_i$ interval transformer
$F_R$ reachability transformer
$\alpha$ abstraction function
$\gamma$ concretization function
$X^\#$ abstract counterpart of $X$
\( \rho \) reduction
\( \nabla \) widening
\( \Delta \) narrowing
\( \mathbb{Z} \) integers
\( \mathbb{N} \) naturals
\( \mathbb{R} \) reals
\( |x| \) absolute value of $x$
\( q \) quaternion
\( ||q|| \) norm of quaternion $q$
\( \overline{q} \) conjugate of quaternion $q$
The validation of software checks informally (e.g., by code reviews or tests) the conformance of the software executions to a specification. More rigorously, the verification of software proves formally the conformance of the software semantics (that is, the set of all possible executions in all possible environments) to a specification. It is of course difficult to design a sound semantics, to get a rigorous description of all execution environments, to derive an automatically exploitable specification from informal natural language requirements, and to completely automatize the formal conformance proof (which is undecidable). In model-based design, the software is often generated automatically from the model so that the certification of the software requires the validation or verification of the model plus that of the translation into an executable software (through compiler verification or translation validation). Moreover, the model is often considered to be the specification, so there is no specification of the specification, hence no other possible conformance check. These difficulties show that fully automatic rigorous verification of complex software is very challenging and perfection is impossible.
We present abstract interpretation [Cousot and Cousot, 1977] and show how its principles can be successfully applied to cope with the above-mentioned difficulties inherent to formal verification.

First, semantics and execution environments can be precisely formalized at different levels of abstraction, so as to correspond to a pertinent level of description as required for the formal verification.

Second, semantics and execution environments can be over-approximated, since it is always sound to consider, in the verification process, more executions and environments than actually occurring in real executions of the software. It is crucial for soundness, however, to never omit any of them, even rare events. For example, floating-point operations incur rounding (to nearest, towards 0, plus or minus infinity) and, in the absence of precise knowledge of the execution environment, one must consider the worst case for each floating-point operation. Another example is the range of inputs, like voltages, that can be overestimated by the full range of the hardware register where the value is sampled (anyway, a well-designed software should be defensive, i.e., have appropriate protections to cope with erroneous or failing sensors and be prepared to accept any value from the registers).

In the absence of an explicit formal specification or to avoid the additional cost of translating the specification into a format understandable by the verification tool, one can consider implicit specifications. For example, memory leaks, buffer overruns, undesired modulo in integer arithmetics, floating-point overflows, data-races, deadlocks, live-locks, etc. are all frequent symptoms of software bugs, the absence of which can be easily incorporated as a valid but incomplete specification in a verification tool, maybe using user-defined parameters to choose among several plausible alternatives.

Because of undecidability issues (which make fully automatic proofs on all programs ultimately impossible) and the desire not to rely on end-user interactive help (which can add a heavy, or even intractable cost), abstract interpretation makes an intensive use of the idea of abstraction, either to restrict the properties to be considered (which introduces the possibility to have efficient computer representations and algorithms to manipulate them) or to approximate the solutions of the
equations involved in the definition of the abstract semantics. Thus, proofs can be automated in a way that is always sound but may be imprecise, so that some questions about the program behaviors and the conformance to the specification cannot be definitely answered neither affirmatively nor negatively. So, for soundness, an alarm will be raised which may be false. Intensive research work is done to discover appropriate abstractions eliminating this uncertainty about false alarms for domain-specific applications.

In this article, we report on the successful scalable and cost-effective application of abstract interpretation to the verification of the absence of runtime errors in aerospace control software by the ASTRÉE static analyzer [Cousot et al., 2007a], illustrated first by the verification of the fly-by-wire primary software of commercial airplanes [Delmas and Souyris, 2007] and then by the validation of the Monitoring and Safing Unit (MSU) of the Jules Vernes ATV docking software [Bouissou et al., 2009]. We also discuss on-going extensions to imperfectly synchronous software, parallel software, and target code validation, and conclude with more prospective goals for rigorously verifying and validating aerospace software.

An early version of this article appeared in [Bertrane et al., 2010]. We extend that version with more thorough explanations, additional examples, and updated experimental results.


References


Full text available at: http://dx.doi.org/10.1561/2500000002


Full text available at: http://dx.doi.org/10.1561/2500000002
References


Euclid of Alexandria. Elementa geometriæ, book xii, proposition 17. fl. 300 BC.


References


