Undecidable Rules and How to Deal with Them

Daniel Kästner, Sebastian Hahn, Jörg Herter, Thomas Karos
AbsInt GmbH, 2020

Coding Guidelines

- Aim to prevent certain coding constructs that in the past introduced defects
- In particular, prevent undefined behaviour
- Aim for certain coding style to increase readability, maintainability, and overall code quality

- Most prominent: MISRA C guideline sets

MISRA C Guideline Classification

- Directives
  - No fully-automatic compliance check on source code
    - Criteria not completely defined
    - Additional activities required
      (checking external documentation, …)
  - Example (Dir 4.1)
    Run-time failures shall be minimized
  - Example (Dir 3.1)
    All code shall be traceable to documented requirements

MISRA C Guideline Classification

- Rules
  - Automatic compliance check possible
  - Scope: system vs. single-translation-unit
  - Algorithmic class: undecidable vs. decidable

MISRA C Guideline Classification

- Decidable rules
  - syntactic properties
  - Example – Rule 5.7.
    A tag name shall be a unique identifier.
- Undecidable rules
  - semantic properties
  - Example – Rule 9.1:
    The value of an object with automatic storage duration shall not be read before it has been set.

Decidability

- Decision problem: Find an algorithm to determine for every possible parameter instance whether a certain parametric statement is true or false in a given axiomatic system [Hilbert1928].

- Program Equivalence: Given two computer programs, do both always compute the same output?

  Program Equivalence is undecidable

- Halting Problem: Given some algorithm and some input to the algorithm (both together are the parameter) does the algorithm halt when run on the given input.

  Halting Problem is undecidable [Turing1937]

… but also

- Planning flights in North America
  =>undecidable

- Determining whether a player has a winning strategy in a game of *Magic: The Gathering*
  => undecidable

Computational Complexity of Air Travel Planning, Carl de Marcken

Magic: The Gathering is Turing Complete, Alex Churchill et al.

AbsInt

Rice's Theorem

- Rice's Theorem: All non-trivial semantic statements about programs are undecidable.

- All runtime error types are defined by program semantics! ;(

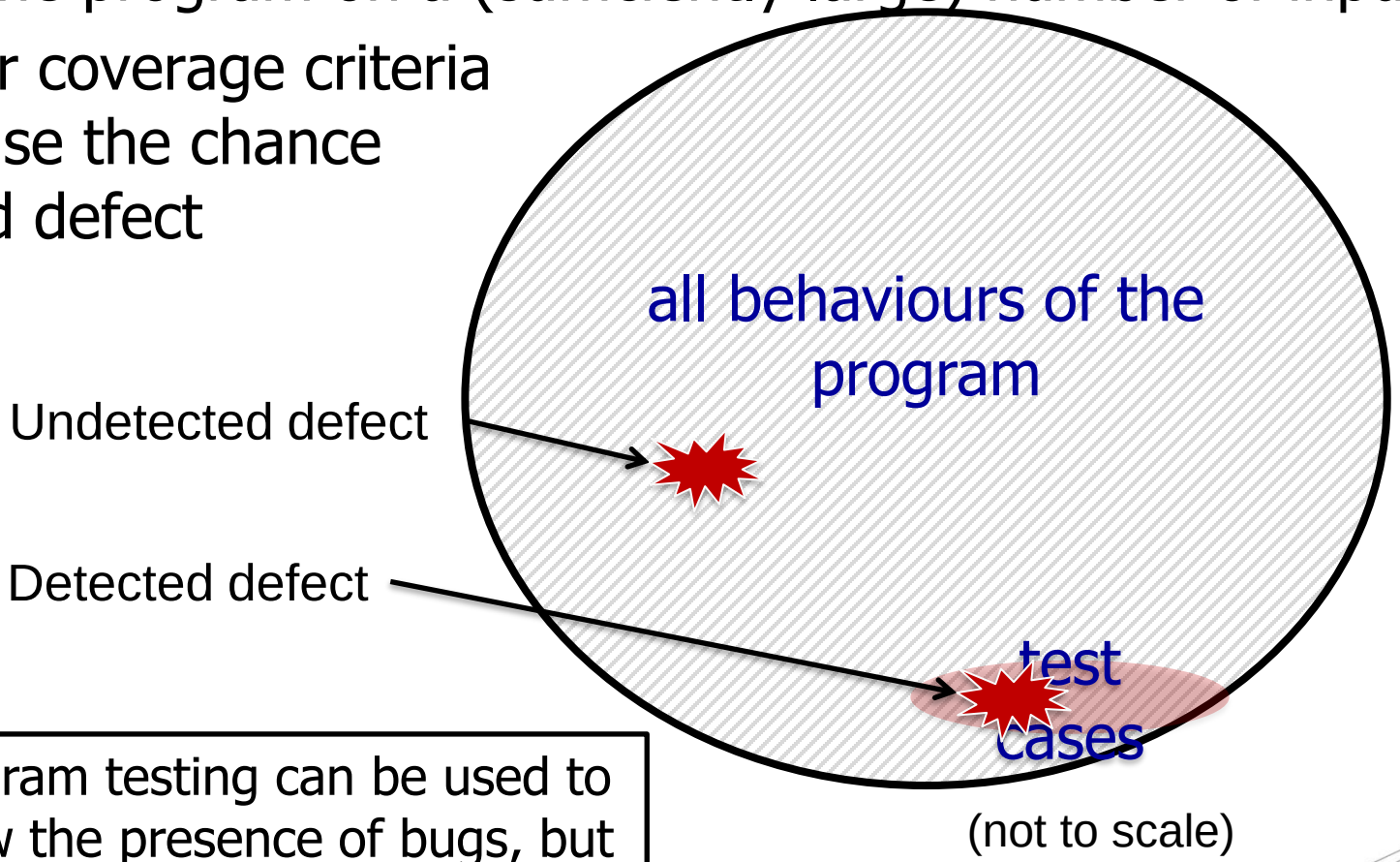
No Silver Bullet

Example property:
no division by 0
in a/b

- There is no verification method that is
  - automatic,
  - powerful: able to prove non-trivial properties,
  - sound: never claims the property holds if it does not hold,
  - complete: always proves the property if it holds, and
  - scalable: can handle industry-size applications.
- What to give up?
  - complete and sound: testing, unsound static analysis
  - complete: sound static analysis (abstract interpretation)
  - automatic: interactive theorem proving
  - scalable: model checking
    Note: BMC gives up soundness/completeness!

Testing

- Test the program on a (sufficiently large) number of inputs
- Higher coverage criteria increase the chance to find defect



Program testing can be used to show the presence of bugs, but never to show their absence!
Dijkstra (1970)

AbsInt

Static Program Analysis

- General Definition: results only computed from program structure, without executing the program under analysis.
- Categories, depending on analysis depth:
  - Syntax-based: checking syntactic ("decidable") coding rules
  - Semantics-based

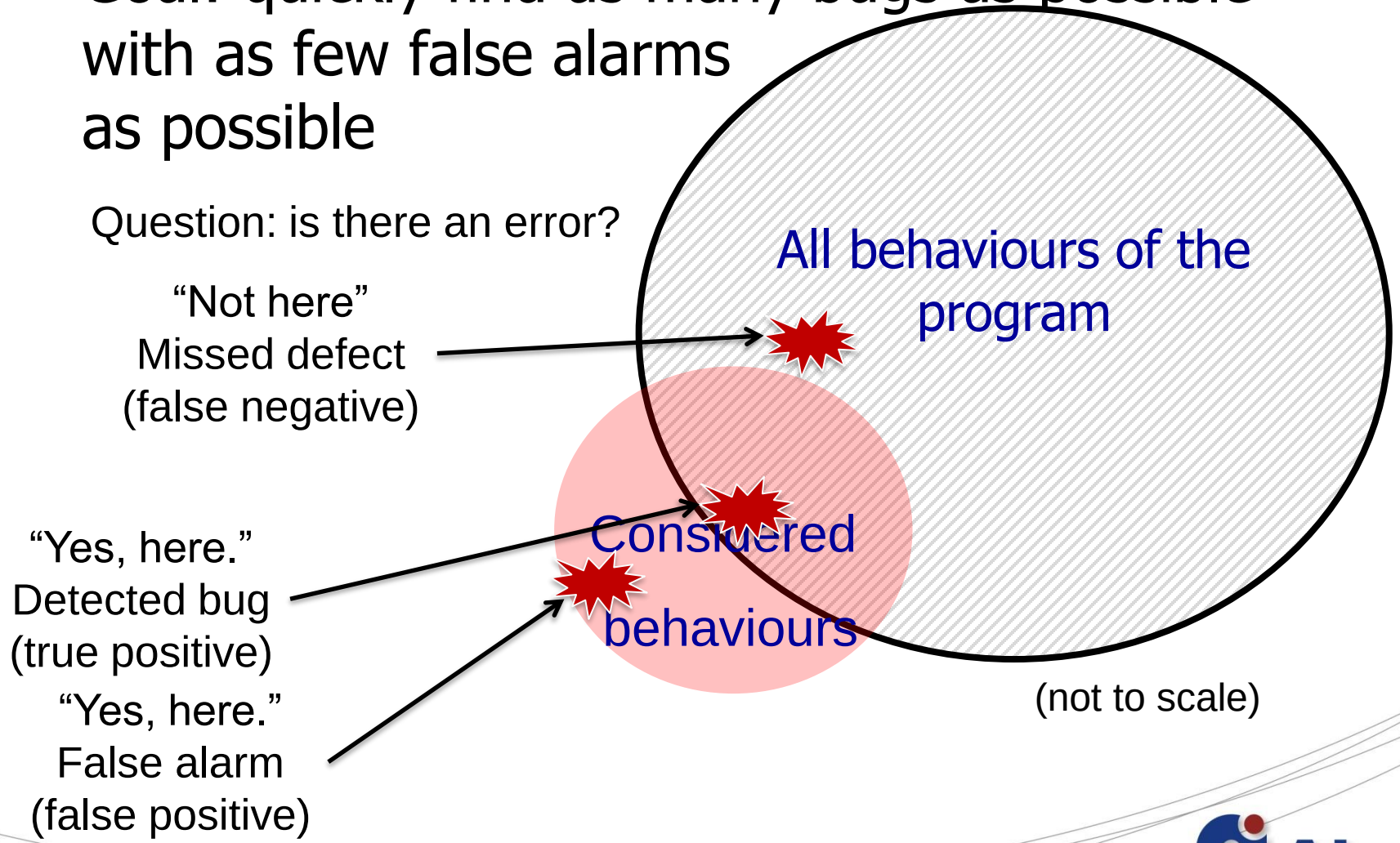
Question: Is there an error in the program?
- False positive: answer wrongly "Yes"
- False negative: answer wrongly "No" 

    - Unsound: Bug chasing
      - False positives: possible
      - False negatives: possible
    - Sound / Abstract Interpretation-based
      - False positives: possible
      - No false negatives ⇨ Soundness
        No defect missed

AbsInt

Unsound Static Analysis – Bug Chasing

- Goal: quickly find as many bugs as possible with as few false alarms as possible

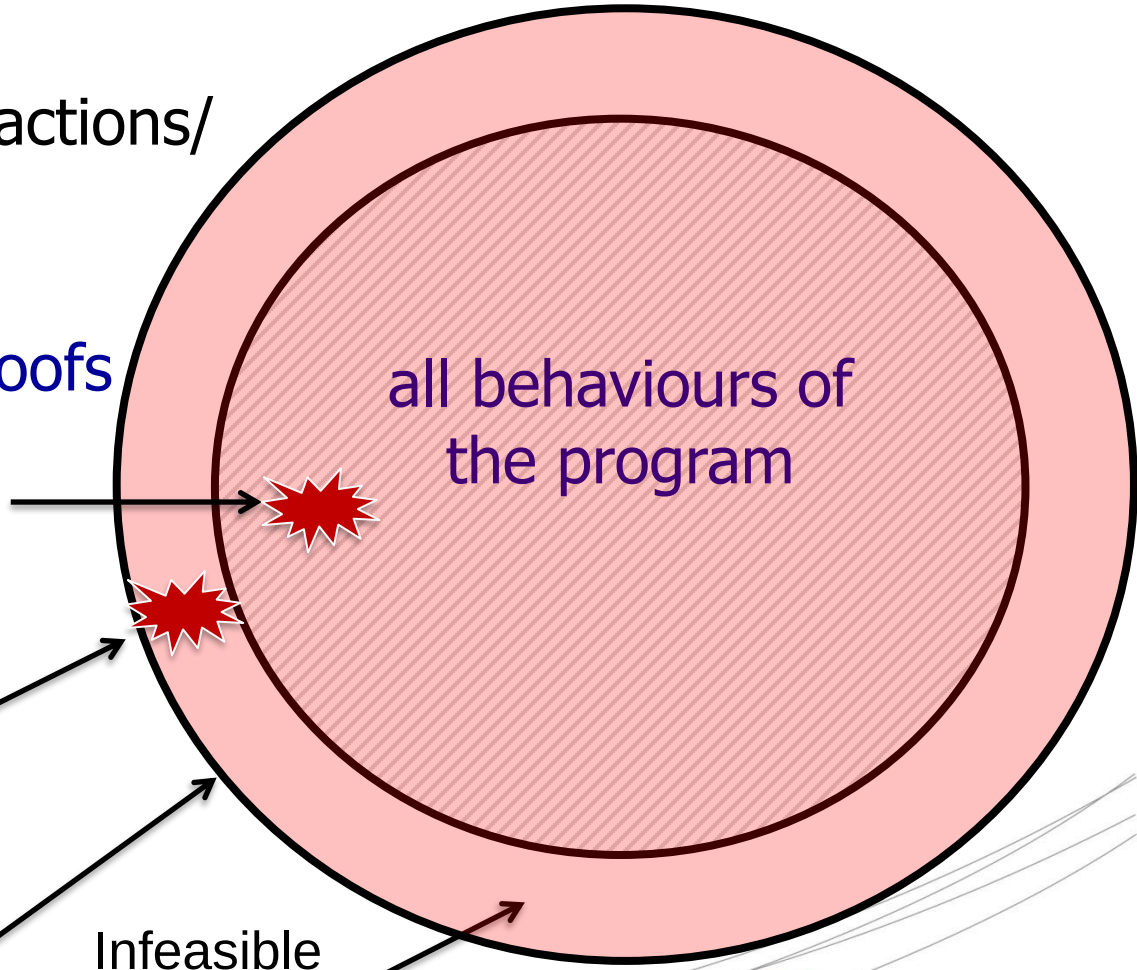


AbsInt

Abstract Interpretation

- Semantics based methodology for program analysis
- Give up completeness:
  gain scalability by abstractions/
  over-approximations
- Formal method:
  supports correctness proofs

Soundness of abstractions mathematically proven:
Never fails to report a defect under analysis



AbsInt

Analysis Depth

- Division by zero
  - `a/0` → division by zero always happens
    - can be detected syntactically
  - `a/b` → division by zero can occur if b might be zero
    - semantic information needed: value range of b
  - Alarm on `a/b`
    - Division by zero might happen
  - No alarm on `a/b`
    - Unsound analyzer:
      division by zero might happen in scenarios not considered
    - Sound analyzer:
      proof that b can never be zero here.

Enhanced Rule Classification

- Decidable rules can be fully checked by static analyzers
- Other guideline (directive or undecidable rule) is
  - Supportable:
    a static code analyzer can help checking this guideline
    - Example:
      Dir 4.1. Run-time failures shall be minimized
  - Opaque:
    a static code analyzer cannot reasonably contribute to checking this guideline
    - Example:
      Dir 3.1. All code shall be traceable to documented requirements

Enhanced Classification – Overview

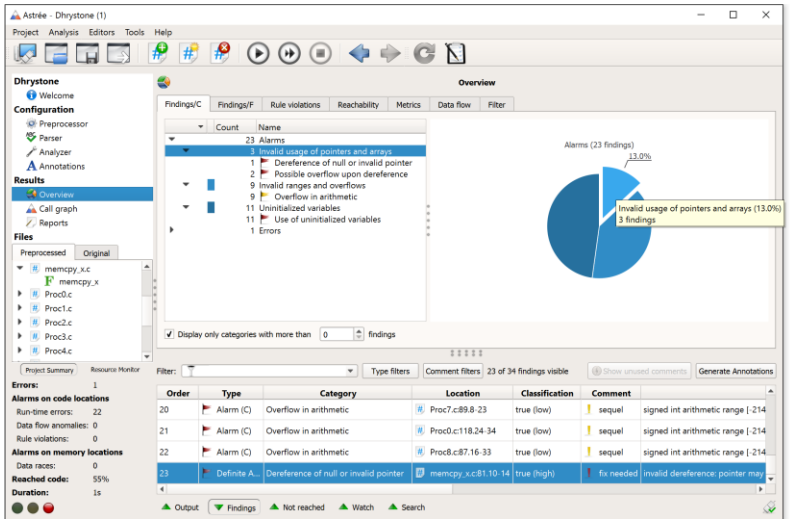
Type	Scope	Support Level	Elements
Directive	-	Supportable	9 (Dir 4.1, 4.5, 4.6, 4.7, 4.9, 4.10, 4.11, 4.12, 4.13)
Directive	-	Opaque	7 (Dir 1.1, 2.1, 3.1, 4.2, 4.3, 4.4, 4.8)
Rule	Undecidable STU	Supportable	1 (Rule 1.2: Language extensions should not be used)
Rule	Undecidable System	Supportable	36 (Rule 1.3, 2.1, 2.2, 8.13, 9.1, 12.2, 13.1, 13.2, 13.5, 14.1, 14.2, 14.3, 17.2, 17.5, 17.8, 18.1, 18.2, 18.3, 18.6, 19.1, 21.13, 21.14, 21.17, 21.18, 21.19, 21.20, 22.1, 22.2, 22.3, 22.4, 22.5, 22.6, 22.7, 22.8, 22.9, 22.10)

Code Guideline Checking

- Example tool: RuleChecker
  - Supports MISRA C:2004/2012, MISRA C++:2008, SEI CERT C, CWE, ISO/IEC 17961:2013 C Secure, Adaptive Autosar C++
  - Fast local data flow analyses (unsound)
  - Very fast analysis engine
  - Minimal setup effort
  - No false negatives and no false positives on decidable rules: every rule check violation is detected
  - Efficient exploration of analysis results
- RuleChecker can be coupled with sound Astrée analyzer
  - \+ No false negatives on semantical rules
  - \+ Very precise, low false alarm rate
  - – Computationally more intensive
  - – More complex setup

Runtime Errors and Data Races

- Abstract Interpretation-based static runtime error analysis
- Astrée detects all runtime errors* with few false alarms:
  - Array index out of bounds
  - Int/float division by 0
  - Invalid pointer dereferences
  - Uninitialized variables
  - Arithmetic overflows
  - Data races
  - Lock/unlock problems, deadlocks
  - Floating point overflows, Inf, NaN
  - Taint analysis (data safety / security), SPECTRE detection
  - \+ Floating-point rounding errors taken into account
  - \+ User-defined assertions, unreachable code, non-terminating loops
  - \+ RuleChecker integrated



* Defects due to undefined / unspecified behaviors of the programming language

AbsInt

Degrees of Automatic Rule Coverage

- Fully checked + Exact (FC+E): the analyzer will never miss a rule violation (no false negatives) and never issue a false alarm (no false positives). Only possible for decidable rules.

- Fully checked (FC): the checks adhere exactly to the rule text and the analysis will never miss a rule violation (no false negatives). Requires abstract interpretation. False positives may occur.

Degrees of Automatic Rule Coverage

- Partially checked with sound support (PC+S): rule is partially covered, but some aspects are covered by sound analysis. False positives and false negatives may occur, but for some aspects there will be no false negatives.

- Partially checked (PC): the checks either check only some aspects of the rule, make simplifying assumptions, or may miss rule violations. Rule violations might be missed. False positives & false negatives may occur.

Degrees of Automatic Rule Coverage

- Supported in a sound way (S): No dedicated check, but an analysis run will provide sound evidence about rule violations.

- Not checked (NC)

RuleChecker – Rule Coverage Degree

Type	Scope	Support Level	Elements
Directive	-	Supportable	9 (Dir 4.1, 4.5, 4.6, 4.7, 4.9, 4.10, 4.11, 4.12, 4.13)
Directive	-	Opaque	7 (Dir 1.1, 2.1, 3.1, 4.2, 4.3, 4.4, 4.8)
Rule	Undecidable STU	Supportable	1 (Rule 1.2: Language extensions should not be used)
Rule	Undecidable System	Supportable	36 (Rule 1.3, 2.1, 2.2, 8.13, 9.1, 12.2, 13.1, 13.2, 13.5, 14.1, 14.2, 14.3, 17.2, 17.5, 17.8, 18.1, 18.2, 18.3, 18.6, 19.1, 21.13, 21.14, 21.17, 21.18, 21.19, 21.20, 22.1, 22.2, 22.3, 22.4, 22.5, 22.6, 22.7, 22.8, 22.9, 22.10)

FC: Fully checked

PC+S: Partially checked with sound support

PC: Partially checked (31)

S: Supported in sound way

NC: Not covered (22)

AbsInt

Astrée – Rule Coverage Degree

Type	Scope	Support Level	Elements
Directive	-	Supportable	9 (Dir 4.1, 4.5, 4.6, 4.7, 4.9, 4.10, 4.11, 4.12, 4.13)
Directive	-	Opaque	7 (Dir 1.1, 2.1, 3.1, 4.2, 4.3, 4.4, 4.8)
Rule	Undecidable STU	Supportable	1 (Rule 1.2: Language extensions should not be used)
Rule	Undecidable System	Supportable	36 (Rule 1.3, 2.1, 2.2, 8.13, 9.1, 12.2, 13.1, 13.2, 13.5, 14.1, 14.2, 14.3, 17.2, 17.5, 17.8, 18.1, 18.2, 18.3, 18.6, 19.1, 21.13, 21.14, 21.17, 21.18, 21.19, 21.20, 22.1, 22.2, 22.3, 22.4, 22.5, 22.6, 22.7, 22.8, 22.9, 22.10)

FC: Fully covered (6)
PC+S: Partially covered with sound support (10)
PC: Partially covered (15)
S: Supported in sound way (6)
NC: Not covered (16)

AbsInt

Some Detailed Examples

- Dir 4.1: Run-time failures shall be minimized
  - PC: at source level, some defects cannot be covered, e.g., stack overflows, deadline violations, compiler bugs. Only undefined/unspecified behaviors.
  - RuleChecker (PC) vs Astrée (PC+S): Astrée performs sound analysis of undefined / unspecified behaviors.
- Rule 1.3: There shall be no occurrence of undefined or critical unspecified behavior.
  - RuleChecker (PC) vs Astrée (PC+S): Astrée performs sound analysis of undefined / unspecified behaviors.
- Rule 2.1.: A project shall not contain unreachable code.
  - RuleChecker (PC) vs. Astrée (PC+S): Reduced coverage in RuleChecker, Astrée can guarantee code reported as unreachable to be unreachable (no false positives).
- Rule 9.1: The value of an object with automatic storage duration shall not be read before it has been set.
  - RuleChecker (PC) vs. Astrée (FC): no false negatives with Astrée

Summary

- Static code analysis is predominant technique to check coding guidelines, such as MISRA C/C++.
- Syntactic ("decidable") rules can be checked without false positives and false negatives.
- Semantic (undecidable) rules:
  - Is this rule violated? No precise yes/no answer possible.
  - Unsound static analyzers: false positives and false negatives.
  - Sound static analyzers: false positives, but no false negatives on many aspects
  - Different tools vary in degree of rule coverage

email: info@absint.com
http://www.absint.com