Astrée
Proving the Absence of Runtime Errors

AbsInt GmbH

2012

Functional Safety

- Demonstration of functional correctness
  - Well-defined criteria
  - Automated and/or model-based testing
  - Formal techniques: model checking, theorem proving
- Satisfaction of non-functional requirements
  - No crashes due to runtime errors (division by zero, invalid pointer accesses, overflow and rounding errors)
  - Resource usage
    - Timing requirements (e.g. WCET, WCRT)
    - Memory requirements (e.g. no stack overflow)
  - Insufficient: tests and measurements
    - Test end criteria unclear
    - No full coverage possible
    - “Testing, in general, cannot show the absence of errors.” — DO-178B
    - Access to physical hardware: high effort due to limited availability and observability

Required by DO-178B/ DO-178C, ISO-26262, EN-50128, IEC-61508

Required by DO-178B/ DO-178C, ISO-26262, EN-50128, IEC-61508

Background

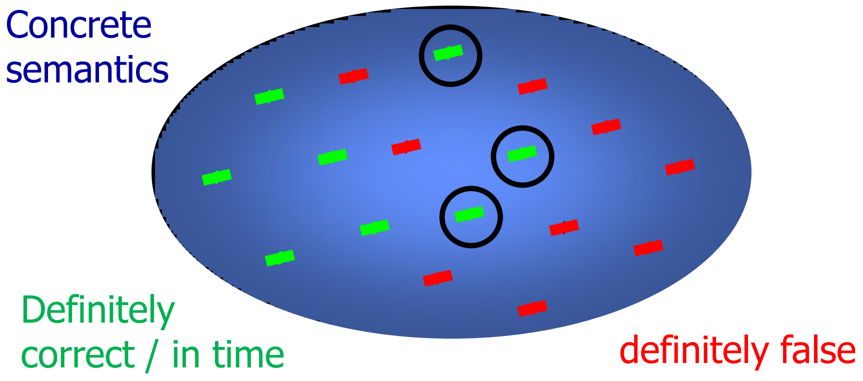
- Safety-critical embedded systems must satisfy high quality objectives
- Software failures can
  - in general: cause high costs, e.g. due to recall campaigns
  - in highly critical systems: endanger human beings
- Software test and validation responsible for significant part of development costs (frequently 50% and beyond)
- Challenge: comprehensively ensure system safety at reasonable costs
- AbsInt focuses on non-functional program errors (timing, memory consumption, runtime errors)
- Examples of related software failures:
  - Time drift in Patriot rockets in 1991 (rounding error)
  - Crash of railway switch controller 1995 in Hamburg-Altona (stack overflow)
  - Explosion of Ariane rocket 1996 (arithmetic overflow)
  - …

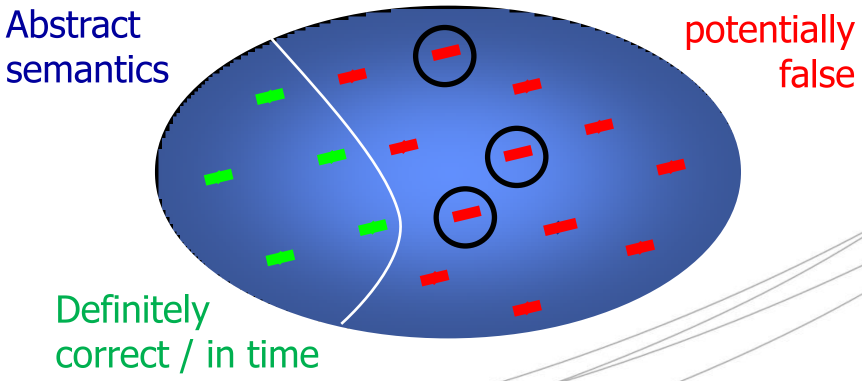
Static Analysis – an Overview

- General definition: results are only computed from the program structure, without executing the program under analysis
- Classification
  - Syntax-based: Style checkers (e.g. MISRA-C)
  - Unsound semantics-based: Bug finders/bug hunters
    - Cannot guarantee that all bugs are found
    - Examples: Splint, Coverity CMC, Klocwork K7,…
  - Sound semantics-based/abstract-interpretation–based
    - Can guarantee that all bugs from the class under analysis are found
    - Results valid for every possible program execution with any possible input scenario
    - Examples: aiT WCET Analyzer, StackAnalyzer, Astrée

Abstract Interpretation

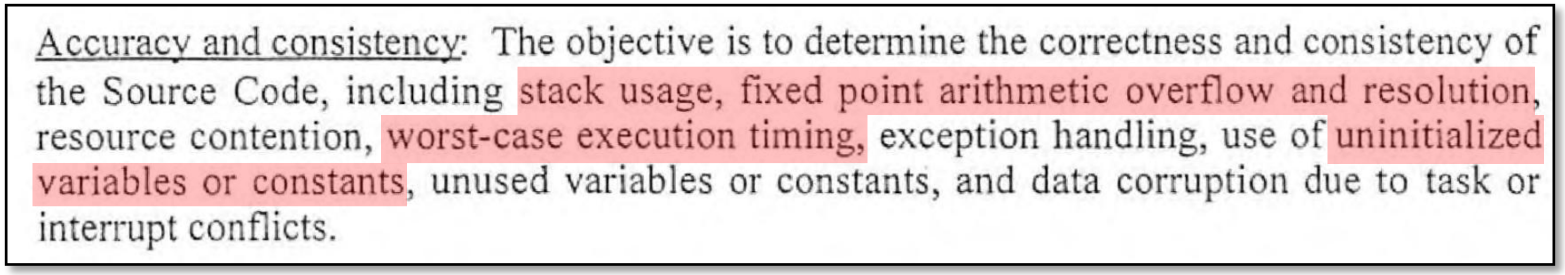
- Most interesting program properties are undecidable in the concrete semantics. Thus: concrete semantics mapped to abstract semantics where program properties are decidable (efficiency–precision trade-off). This makes analysis of large software projects feasible.
- Soundness: A static analysis is said to be sound when the data flow information it produces is guaranteed to be true for every possible program execution. Formally provable by abstract interpretation.
- Safety: Computation of safe overapproximation of program semantics: some precision may be lost, but imprecision is always on the safe side.





Aerospace: DO-178B/DO-178C

- "Verification is not simply testing.
  Testing, in general, cannot show the absence of errors."
- "The general objectives of the software verification process are to verify that the requirements of the system level, the architecture level, the source code level and the executable object code level are satisfied, and that the means used to satisfy these objectives are technically correct and complete."


Accuracy and consistency: The objective is to determine the correctness and consistency of the Source Code, including stack usage, fixed point arithmetic overflow and resolution, resource contention, worst-case execution timing, exception handling, use of uninitialized variables or constants, unused variables or constants, and data corruption due to task or interrupt conflicts.

- The DO-178C is a revision of DO-178B to bring it up to date with respect to current software development and verification technologies, e.g. the use of formal methods to complement or replace dynamic testing: theorem proving, model checking, abstract interpretation.

Automotive: ISO-26262

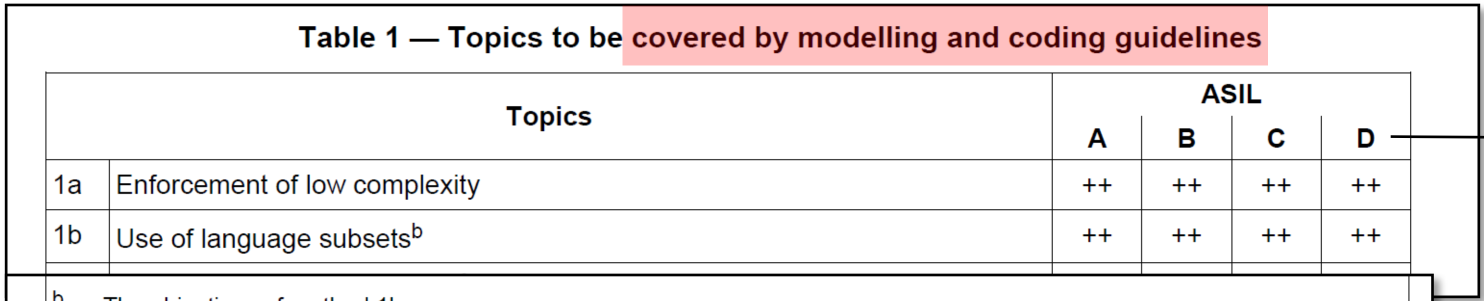
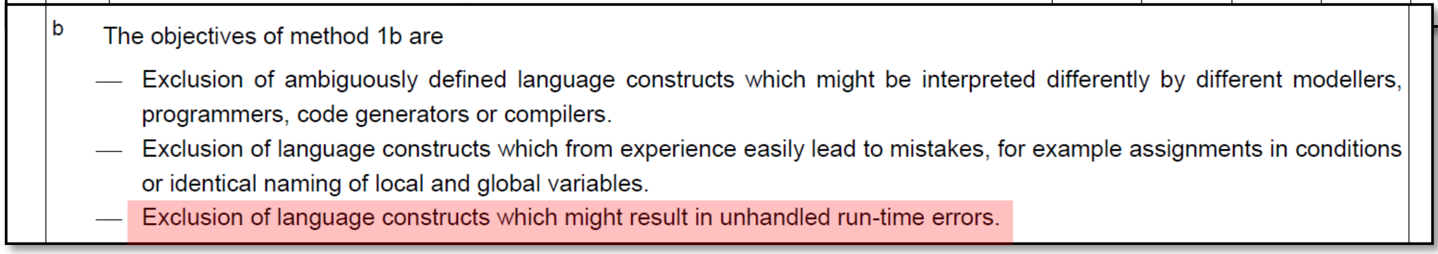


Table 1 — Topics to be covered by modelling and coding guidelines

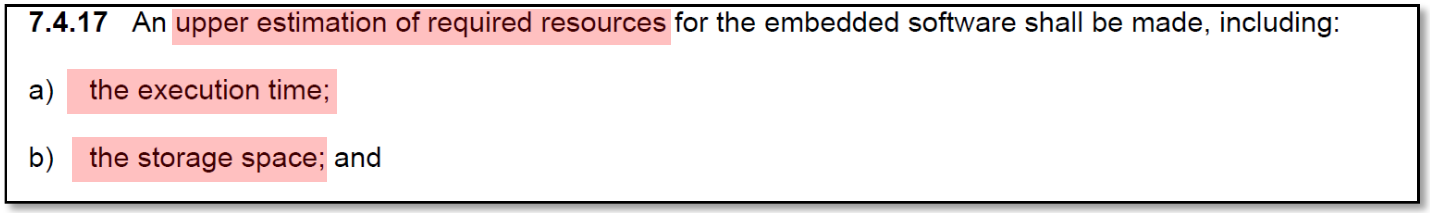
	Topics	ASIL A	ASIL B	ASIL C	ASIL D
1a	Enforcement of low complexity	++	++	++	++
1b	Use of language subsets[b]	++	++	++	++

Criticality levels:
A (lowest) to
D (highest)



[b] The objectives of method 1b are

- Exclusion of ambiguously defined language constructs which might be interpreted differently by different modellers, programmers, code generators or compilers.
- Exclusion of language constructs which from experience easily lead to mistakes, for example assignments in conditions or identical naming of local and global variables.
- Exclusion of language constructs which might result in unhandled run-time errors.



7.4.17 An upper estimation of required resources for the embedded software shall be made, including:

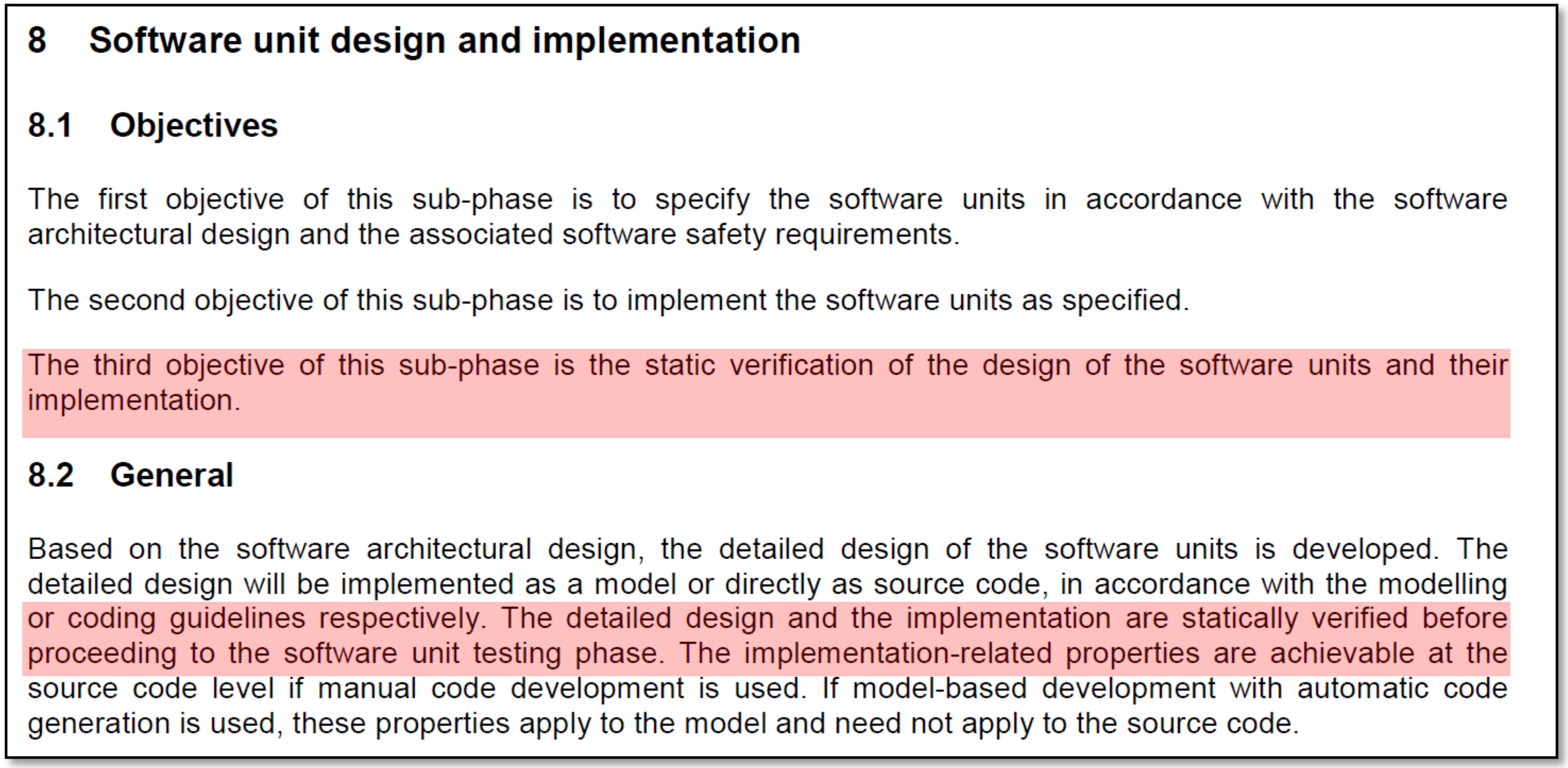
a) the execution time;

b) the storage space; and

Excerpt from:
Final Draft ISO 26262-6 Road vehicles – Functional safety –
Part 6: Product development: Software Level.
Version ISO/FDIS 26262-6:2011(E), 2011.

Automotive: ISO-26262

- Importance of static verification emphasized:



8 Software unit design and implementation

8.1 Objectives

The first objective of this sub-phase is to specify the software units in accordance with the software architectural design and the associated software safety requirements.

The second objective of this sub-phase is to implement the software units as specified.

The third objective of this sub-phase is the static verification of the design of the software units and their implementation.

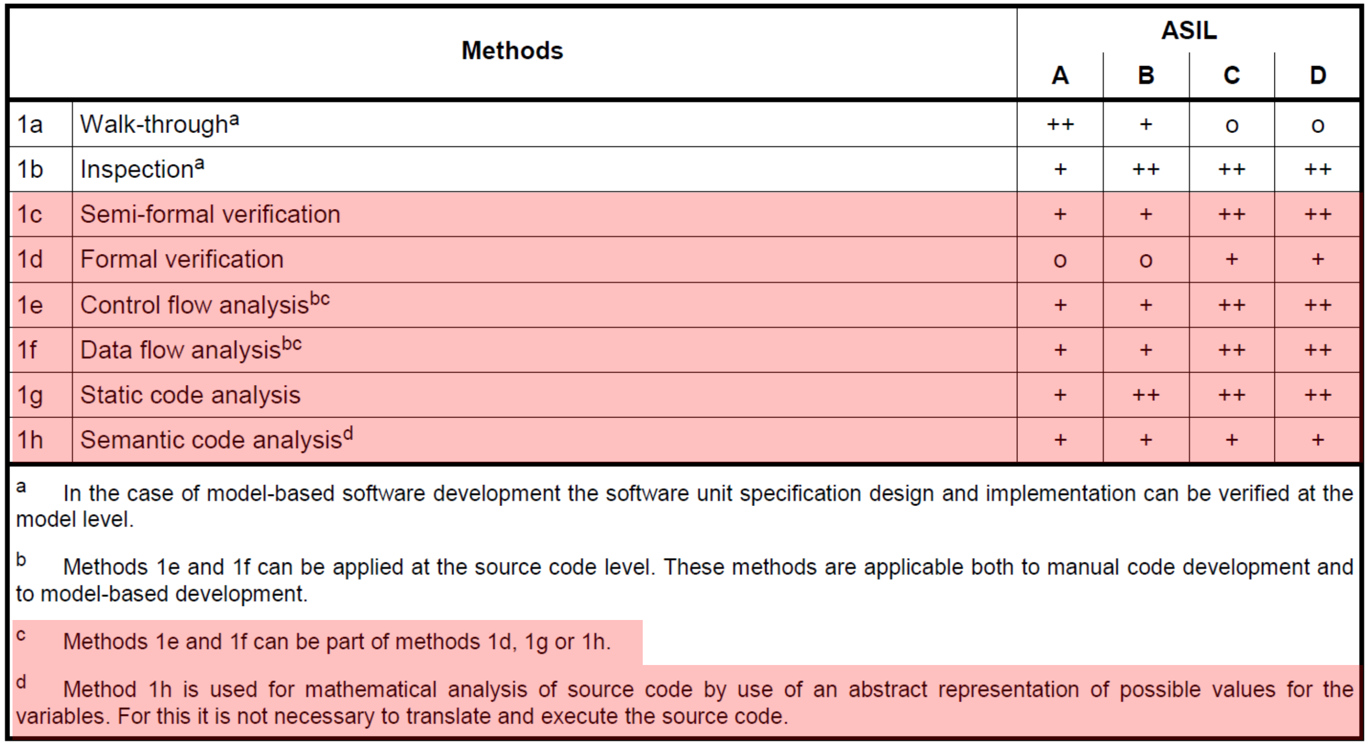
8.2 General

Based on the software architectural design, the detailed design of the software units is developed. The detailed design will be implemented as a model or directly as source code, in accordance with the modelling or coding guidelines respectively. The detailed design and the implementation are statically verified before proceeding to the software unit testing phase. The implementation-related properties are achievable at the source code level if manual code development is used. If model-based development with automatic code generation is used, these properties apply to the model and need not apply to the source code.

Excerpt from:
Final Draft ISO 26262-6 Road vehicles – Functional safety –
Part 6: Product development: Software Level.
Version ISO/FDIS 26262-6:2011(E), 2011.

Automotive: ISO-26262

Table 9 — Methods for the verification of software unit design and implementation



Methods		ASIL			
		A	B	C	D
1a	Walk-through[a]	++	+	o	o
1b	Inspection[a]	+	++	++	++
1c	Semi-formal verification	+	+	++	++
1d	Formal verification	o	o	+	+
1e	Control flow analysis[bc]	+	+	++	++
1f	Data flow analysis[bc]	+	+	++	++
1g	Static code analysis	+	++	++	++
1h	Semantic code analysis[d]	+	+	+	+

[a] In the case of model-based software development the software unit specification design and implementation can be verified at the model level.

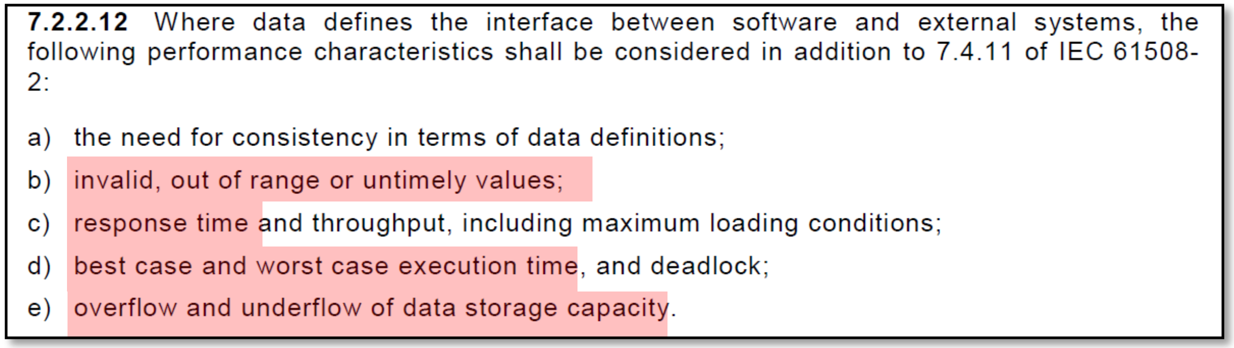
[b] Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.

[c] Methods 1e and 1f can be part of methods 1d, 1g or 1h.

[d] Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

Excerpt from:
Final Draft ISO 26262-6 Road vehicles – Functional safety – Part 6: Product development: Software Level. Version ISO/FDIS 26262-6:2011(E), 2011.

E&E Systems: IEC-61508 – Edition 2.0



7.2.2.12 Where data defines the interface between software and external systems, the following performance characteristics shall be considered in addition to 7.4.11 of IEC 61508-2:

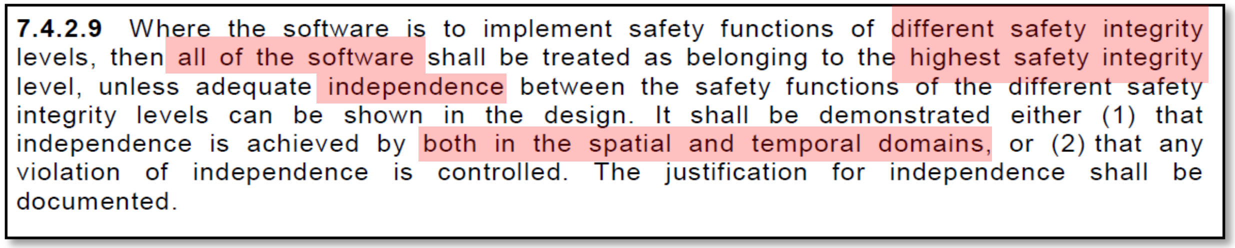
a) the need for consistency in terms of data definitions;

b) invalid, out of range or untimely values;

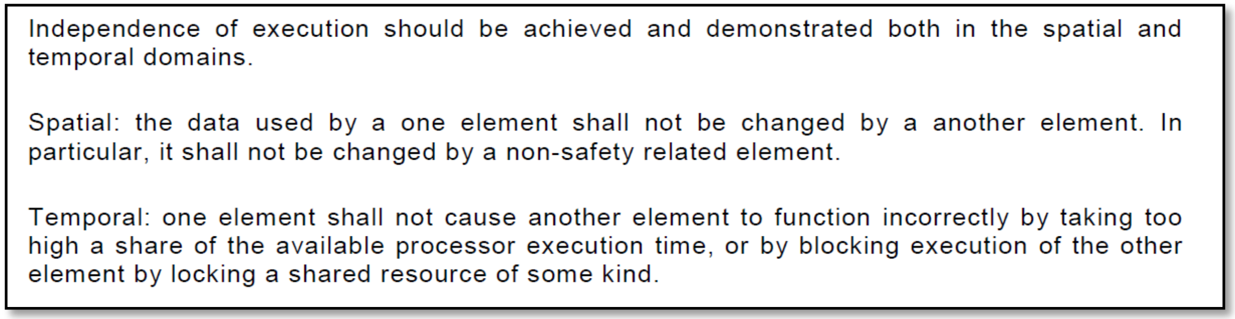
c) response time and throughput, including maximum loading conditions;

d) best case and worst case execution time, and deadlock;

e) overflow and underflow of data storage capacity.



7.4.2.9 Where the software is to implement safety functions of different safety integrity levels, then all of the software shall be treated as belonging to the highest safety integrity level, unless adequate independence between the safety functions of the different safety integrity levels can be shown in the design. It shall be demonstrated either (1) that independence is achieved by both in the spatial and temporal domains, or (2) that any violation of independence is controlled. The justification for independence shall be documented.



Independence of execution should be achieved and demonstrated both in the spatial and temporal domains.

Spatial: the data used by a one element shall not be changed by a another element. In particular, it shall not be changed by a non-safety related element.

Temporal: one element shall not cause another element to function incorrectly by taking too high a share of the available processor execution time, or by blocking execution of the other element by locking a shared resource of some kind.

Excerpt from:
IEC-61508, Edition 2.0. Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements

E&E Systems: IEC-61508 – Edition 2.0

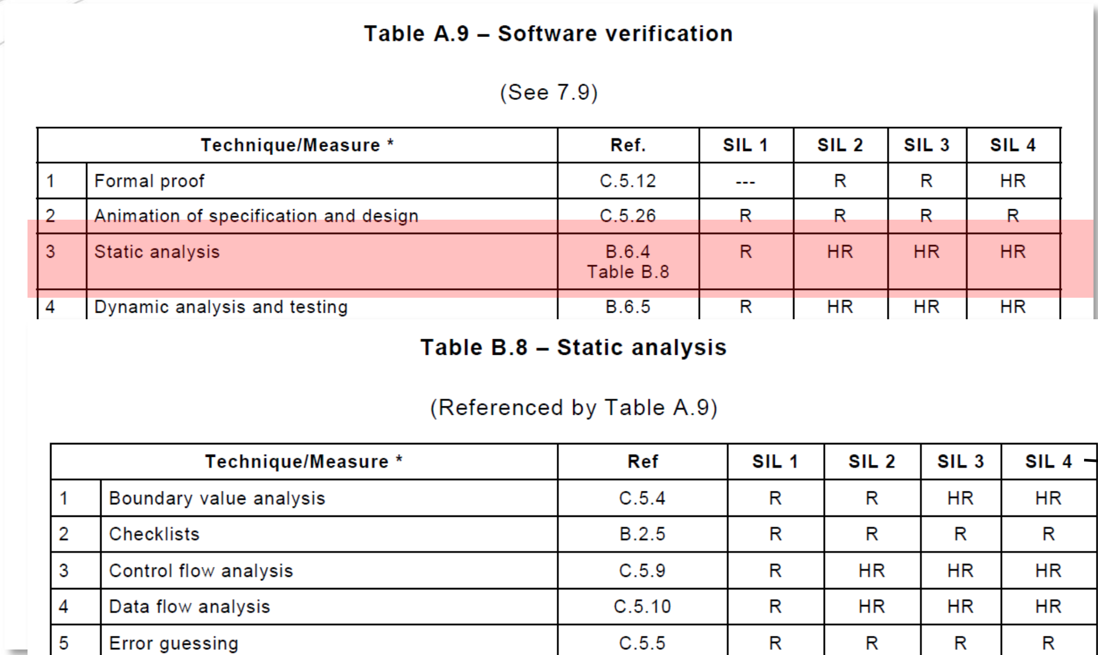


Table A.9 – Software verification

(See 7.9)

Technique/Measure *		Ref.	SIL 1	SIL 2	SIL 3	SIL 4
1	Formal proof	C.5.12	---	R	R	HR
2	Animation of specification and design	C.5.26	R	R	R	R
3	Static analysis	B.6.4 Table B.8	R	HR	HR	HR
4	Dynamic analysis and testing	B.6.5	R	HR	HR	HR

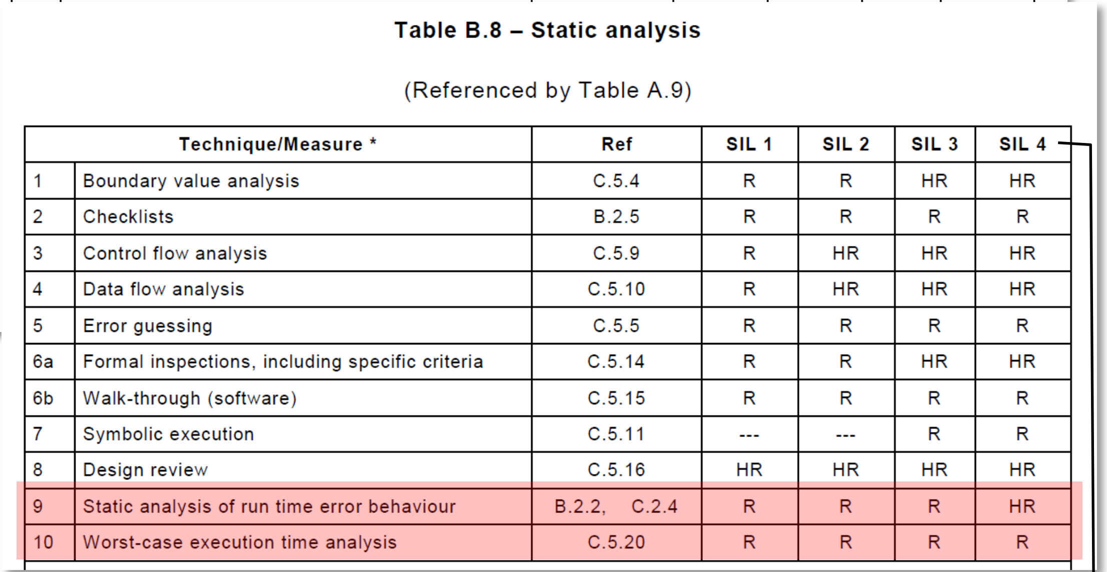
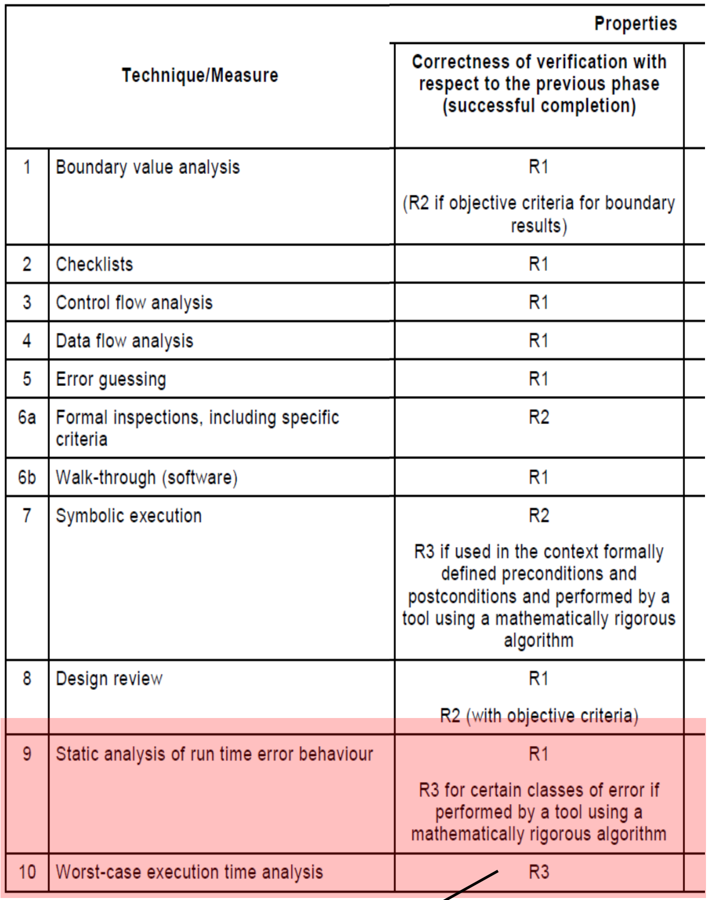


Table B.8 – Static analysis

(Referenced by Table A.9)

Technique/Measure *		Ref	SIL 1	SIL 2	SIL 3	SIL 4
1	Boundary value analysis	C.5.4	R	R	HR	HR
2	Checklists	B.2.5	R	R	R	R
3	Control flow analysis	C.5.9	R	HR	HR	HR
4	Data flow analysis	C.5.10	R	HR	HR	HR
5	Error guessing	C.5.5	R	R	R	R
6a	Formal inspections, including specific criteria	C.5.14	R	R	HR	HR
6b	Walk-through (software)	C.5.15	R	R	R	R
7	Symbolic execution	C.5.11	---	---	R	R
8	Design review	C.5.16	HR	HR	HR	HR
9	Static analysis of run time error behaviour	B.2.2, C.2.4	R	R	R	HR
10	Worst-case execution time analysis	C.5.20	R	R	R	R



Technique/Measure		Properties
		Correctness of verification with respect to the previous phase (successful completion)
1	Boundary value analysis	R1 (R2 if objective criteria for boundary results)
2	Checklists	R1
3	Control flow analysis	R1
4	Data flow analysis	R1
5	Error guessing	R1
6a	Formal inspections, including specific criteria	R2
6b	Walk-through (software)	R1
7	Symbolic execution	R2 R3 if used in the context formally defined preconditions and postconditions and performed by a tool using a mathematically rigorous algorithm
8	Design review	R1 R2 (with objective criteria)
9	Static analysis of run time error behaviour	R1 R3 for certain classes of error if performed by a tool using a mathematically rigorous algorithm
10	Worst-case execution time analysis	R3

Criticality levels:
SIL1 (lowest) to
SIL4 (highest)

Confidence levels:
R1 (lowest) to
R3 (highest)

Railway: prEN-50128

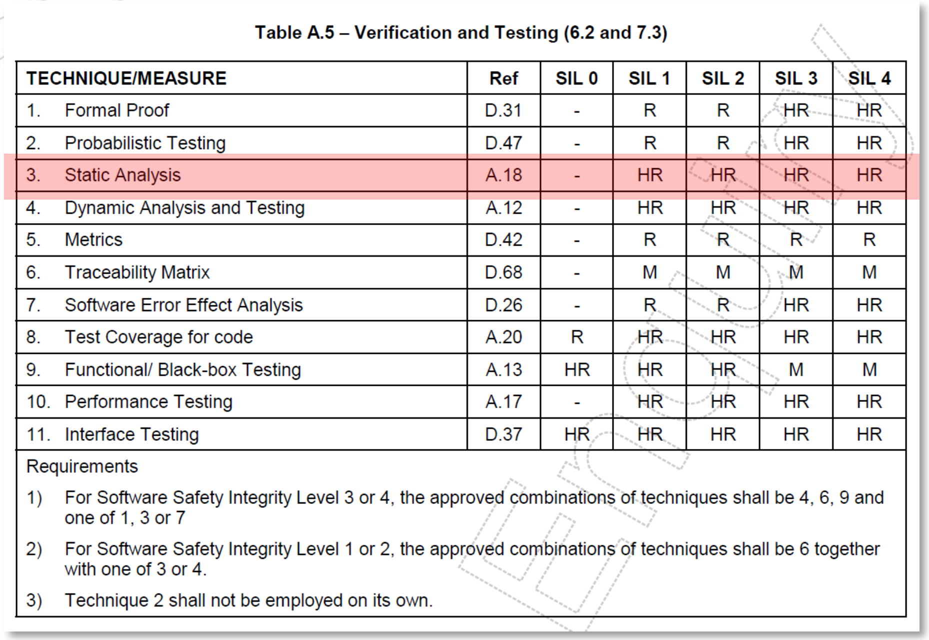


Table A.5 – Verification and Testing (6.2 and 7.3)

TECHNIQUE/MEASURE	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Formal Proof	D.31	-	R	R	HR	HR
2. Probabilistic Testing	D.47	-	R	R	HR	HR
3. Static Analysis	A.18	-	HR	HR	HR	HR
4. Dynamic Analysis and Testing	A.12	-	HR	HR	HR	HR
5. Metrics	D.42	-	R	R	R	R
6. Traceability Matrix	D.68	-	M	M	M	M
7. Software Error Effect Analysis	D.26	-	R	R	HR	HR
8. Test Coverage for code	A.20	R	HR	HR	HR	HR
9. Functional/ Black-box Testing	A.13	HR	HR	HR	M	M
10. Performance Testing	A.17	-	HR	HR	HR	HR
11. Interface Testing	D.37	HR	HR	HR	HR	HR

Requirements

1) For Software Safety Integrity Level 3 or 4, the approved combinations of techniques shall be 4, 6, 9 and one of 1, 3 or 7

2) For Software Safety Integrity Level 1 or 2, the approved combinations of techniques shall be 6 together with one of 3 or 4.

3) Technique 2 shall not be employed on its own.

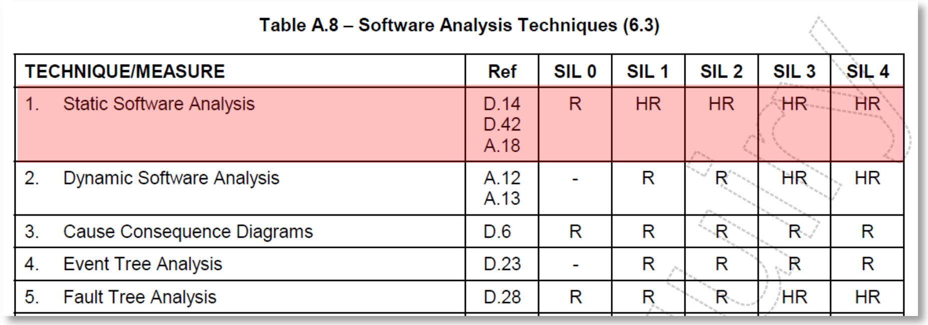


Table A.8 – Software Analysis Techniques (6.3)

TECHNIQUE/MEASURE	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Static Software Analysis	D.14 D.42 A.18	R	HR	HR	HR	HR
2. Dynamic Software Analysis	A.12 A.13	-	R	R	HR	HR
3. Cause Consequence Diagrams	D.6	R	R	R	R	R
4. Event Tree Analysis	D.23	-	R	R	R	R
5. Fault Tree Analysis	D.28	R	R	R	HR	HR

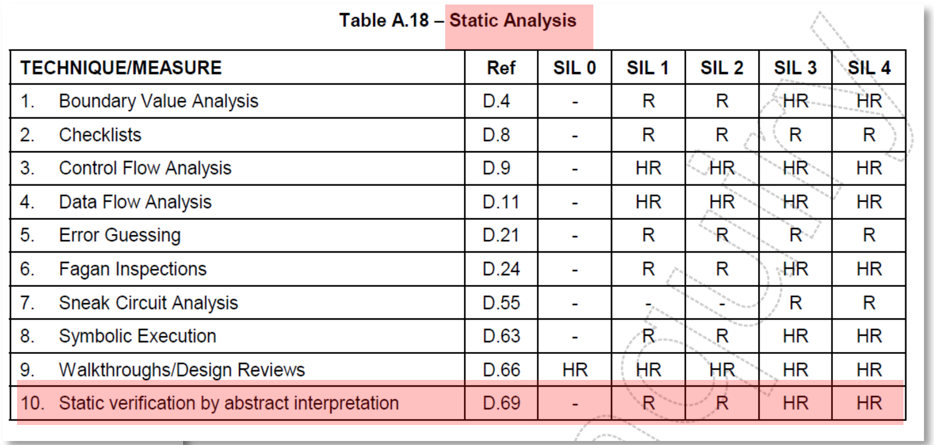
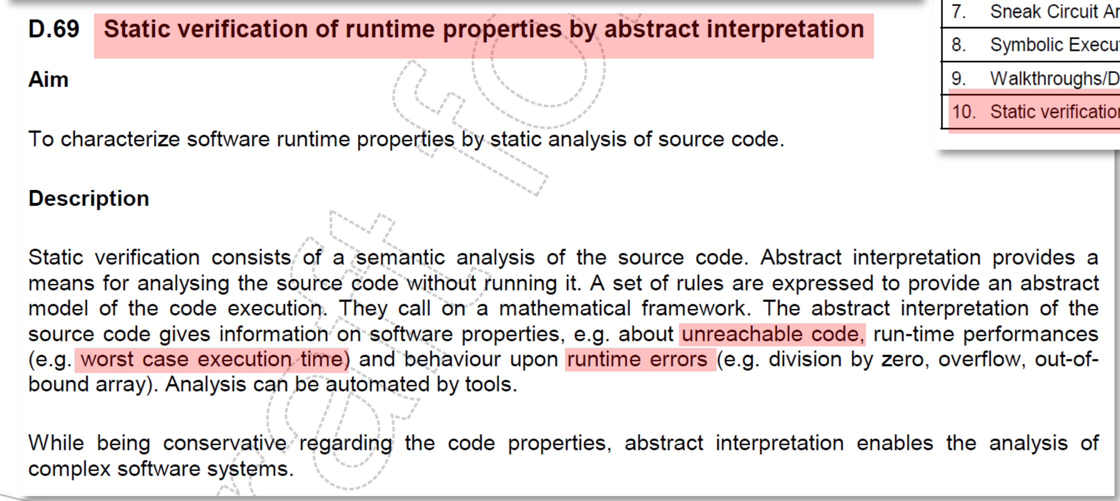


Table A.18 – Static Analysis

TECHNIQUE/MEASURE	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Boundary Value Analysis	D.4	-	R	R	HR	HR
2. Checklists	D.8	-	R	R	R	R
3. Control Flow Analysis	D.9	-	HR	HR	HR	HR
4. Data Flow Analysis	D.11	-	HR	HR	HR	HR
5. Error Guessing	D.21	-	R	R	R	R
6. Fagan Inspections	D.24	-	R	R	HR	HR
7. Sneak Circuit Analysis	D.55	-	-	-	R	R
8. Symbolic Execution	D.63	-	R	R	HR	HR
9. Walkthroughs/Design Reviews	D.66	HR	HR	HR	HR	HR
10. Static verification by abstract interpretation	D.69	-	R	R	HR	HR



D.69 Static verification of runtime properties by abstract interpretation

Aim

To characterize software runtime properties by static analysis of source code.

Description

Static verification consists of a semantic analysis of the source code. Abstract interpretation provides a means for analysing the source code without running it. A set of rules are expressed to provide an abstract model of the code execution. They call on a mathematical framework. The abstract interpretation of the source code gives information on software properties, e.g. about unreachable code, run-time performances (e.g. worst case execution time) and behaviour upon runtime errors (e.g. division by zero, overflow, out-of-bound array). Analysis can be automated by tools.

While being conservative regarding the code properties, abstract interpretation enables the analysis of complex software systems.

Excerpt from:
DRAFT prEN 50128,
July 2009

Industry Perspective

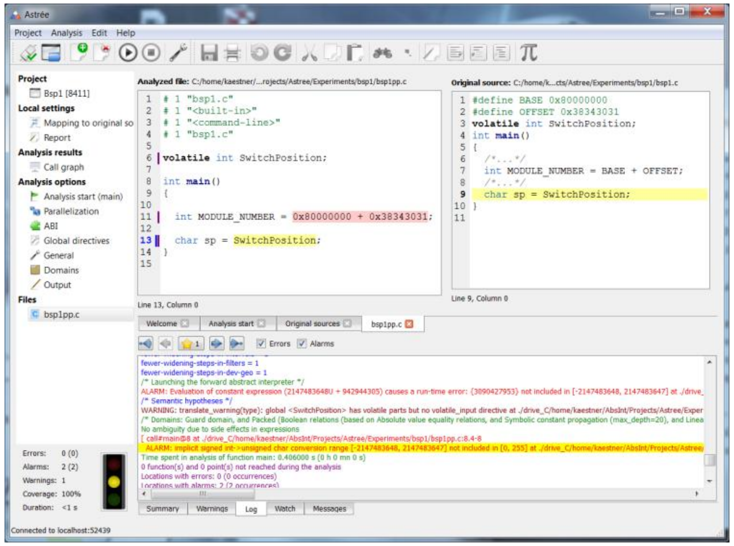
- In most current safety standards variants of static analysis are recommended or highly recommended as a verification technique
- Abstract-interpretation–based static analyzers are in wide industrial use: state-of-the-art for validating non-functional safety properties
- Examples:
  - Static WCET analysis (aiT)
  - Static stack usage analysis (StackAnalyzer)
  - Static runtime error analysis (Astrée): proving the absence of erroneous pointer dereferencing, out-of-bounds array indices, arithmetic overflows, division by zero,…
- aiT application examples:
  - safety-critical Airbus software in many airplane types (A380,…)
  - by NASA as an industry-standard tool for demonstrating the absence of timing-related software defects in the Toyota Unintended Acceleration Investigation (2010)*

* Technical Support to the National Highway Traffic Safety Administration (NHTSA) on the Reported Toyota Motor Corporation (TMC) Unintended Acceleration (UA) Investigation.

AbsInt

The Static Analyzer Astrée

- Crashes or undefined behavior due to runtime errors are bad
- Too many false alarms are bad
- Astrée detects all runtime errors with few false alarms
  - Array index out of bounds
  - Integer division by 0
  - Invalid pointer dereferences
  - Arithmetic overflows and wrap-arounds
  - Floating point overflows and invalid operations (IEEE floating values **`Inf`** and **`NaN`**)
  - User-defined assertions, unreachable code, uninitialized variables
  - Elimination of false alarms by local tuning of analysis precision



Analyzed file: C:/home/kaestner/...rojects/Astree/Experiments/bsp1/bsp1pp.c

```
# 1 "bsp1.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "bsp1.c"

volatile int SwitchPosition;

int main()
{

  int MODULE_NUMBER = 0x80000000 + 0x38343031;

  char sp = SwitchPosition;
}

```

Line 13, Column 0

Original source: C:/home/k...cts/Astree/Experiments/bsp1/bsp1.c

```
#define BASE 0x80000000
#define OFFSET 0x38343031
volatile int SwitchPosition;
int main()
{
  /*...*/
  int MODULE_NUMBER = BASE + OFFSET;
  /*...*/
  char sp = SwitchPosition;
}

```

Line 9, Column 0

```
fewer-widening-steps-in-filters = 1
fewer-widening-steps-in-dev-geo = 1
/* Launching the forward abstract interpreter */
ALARM: Evaluation of constant expression (2147483648U + 942944305) causes a run-time error: {3090427953} not included in [-2147483648, 2147483647] at ./drive_
/* Semantic hypotheses */
WARNING: translate_warning(type): global <SwitchPosition> has volatile parts but no volatile_input directive at ./drive_C/home/kaestner/AbsInt/Projects/Astree/Exper
/* Domains: Guard domain, and Packed (Boolean relations (based on Absolute value equality relations, and Symbolic constant propagation (max_depth=20), and Linea
No ambiguity due to side effects in expressions
[ call#main@8 at ./drive_C/home/kaestner/AbsInt/Projects/Astree/Experiments/bsp1/bsp1pp.c:8.4-8
  ALARM: implicit signed int->unsigned char conversion range [-2147483648, 2147483647] not included in [0, 255] at ./drive_C/home/kaestner/AbsInt/Projects/Astree/
Time spent in analysis of function main: 0.406000 s (0 h 0 mn 0 s)
0 function(s) and 0 point(s) not reached during the analysis
Locations with errors: 0 (0 occurrences)
Locations with alarms: 2 (2 occurrences)
```

Errors:	0 (0)
Alarms:	2 (2)
Warnings:	1
Coverage:	100%
Duration:	<1 s

Connected to localhost:52439

The Zero Alarm Goal

- With zero alarms, the absence of runtime errors is automatically proven by the analysis, without additional reasoning
- Design features of Astrée:
  - Precise and extensible analysis engine, combining powerful abstract domains (intervals, octagons, filters, decision trees,…)
  - Support for precise alarm investigation
    - Source code views/editors for original/preprocessed code
    - Alarms and error messages are linked: jump to location with one click
    - Detailed alarm reporting: precise location and context, call stack, etc.

    → Understanding alarms ⇒ Fixing true runtime errors + Eliminating false alarms
  - The more precise the analysis is, the fewer false alarms there are. Astrée supports improving precision by
    - parametrization: local tuning of analysis precision
    - making external knowledge available to Astrée
    - specialization: adaptation to software class and target hardware

Types of Runtime Errors (1/2)

- Runtime errors causing undefined behavior (with unpredictable results)
  - Modifications through out-of-bounds array accesses, dangling pointers,…
  - Integer divisions by zero, floating-point exceptions,…
- Example:

```
int main() {
  int n, T[1];
  n = 2147483647;
  printf("n = %i, T[n] = %i\n", n, T[n]);
}
```

PPC MAC:	`n=2147483647,T[n]=2147483647`	32-bit Intel:	`n=2147483647,T[n]=-135294988`
Intel MAC:	`n=2147483647,T[n]=-1208492044`	64-bit Intel:	`Bus error`

- Astrée's reaction:
  - Raise alarm in order to signal a potential runtime error
  - Continue analysis for scenarios where the runtime error did not occur
  - If the error definitely occurs in a given context,
    stop the analysis for this context and report the error

Types of Runtime Errors (2/2)

- Runtime errors causing unspecified, but predictable behavior
  - Integer overflow
  - Invalid shifts <<, >>, or casts,…
- Astrée’s reaction:
  1. Raise alarm in order to signal a potential runtime error
  2. Continue analysis with an overapproximation of all possible results

→ No artificial restrictions on value ranges, so the results are always safe

```
volatile short x,y;
__ASTREE_volatile_input((x, [-1,1]));
__ASTREE_volatile_input((y, [-1,1]));
void main()
{
  short z;
  z = (short)((unsigned short)x +
              (unsigned short)y);
  __ASTREE_assert((-2<=z && z<=2));
}
```

Overflow detected in signed short → unsigned short conversions

Nevertheless: precise range for **z** on two's complement hardware (configurable)

Astrée Domains

- Interval domain, Octagon domain
- Floating-point computations:
  - Control programs often perform massive floating-point computations
  - Rounding errors have to be taken into account for precise analysis
  - Astrée approximates expressions on variables V_k as

$$[a_0, b_0] + \sum_k [a_k, b_k] \cdot V_k$$

  - Rounding modes can be changed during runtime
  - Astrée considers the worst case of all possible rounding modes

```
#include <stdio.h>
int main () {
  double x; float a,y,z,r1,r2;
  a = 1.0; x = 1125899973951488.0;
  y = x+a; z = x-a;
  r1 = y - z; r2 = 2*a;
  printf("(x+a)-(x-a) = %f\n", r1);
  printf("2a          = %f\n", r2);
}
```

Output:

```
(x+a)-(x-a) = 134217728.0000
2a = 2.0000
```

Astrée result:

```
r1 in [-1.34218e+08, 1.34218e+08]
r2 = 2.0
```

- Further value domains: Decision tree domain, Digital filter domain, Clock domain, Memory domain,...

Analysis Process

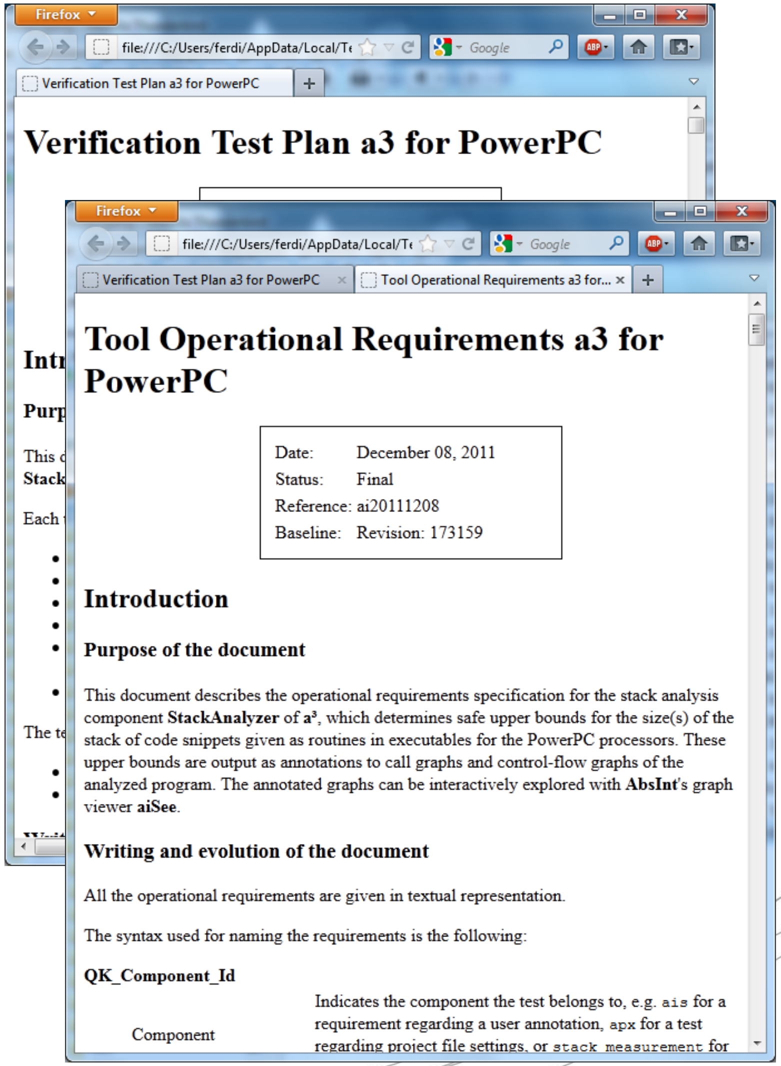
1. Preprocess the code
   - by adapting the build process, or
   - from the built-in Astrée preprocessor
2. Define appropriate analysis options
3. Run the analysis
   1. Investigate the alarms
   2. Fix true errors
   3. Use Astrée directives
      to fine-tune the analyzer
4. Generate final reports

Real-World Applications

- Astrée has been used successfully for industrial avionics, automotive and space applications
- Success stories include:
  - 132 000 lines of C code. 1200 false alarms on first run. After correction and parametrization: 11 false alarms. Analysis runtime: 110 min on a 2.4 GHz PC, 1 GB RAM.
  - 200 000 lines of preprocessed C code. 467 alarms on first run. After correction and parametrization: zero alarms. Runtime c. 6h on a 2.6 GHz PC, 16 GB RAM.
  - 755 197 lines of preprocessed C code. After correction and parametrization: zero alarms. Runtime c. 6h on Intel Core2Duo 2.66 GHz, 8 GB RAM.

Qualification Support Kits

- Report Package
  - Operational Requirements Report: lists all functional requirements
  - Verification Test Plan: describes one or more test cases to check each functional requirement
- Test Package
  - All test cases listed in the Verification Test Plan report
  - Scripts to execute all test cases including an evaluation of the results


Firefox
file:///C:/Users/ferdi/AppData/Local/Te
Google
Verification Test Plan a3 for PowerPC
Tool Operational Requirements a3 for...

Verification Test Plan a3 for PowerPC

Tool Operational Requirements a3 for PowerPC

Date:	December 08, 2011
Status:	Final
Reference:	ai20111208
Baseline:	Revision: 173159

Introduction

Purpose of the document

This document describes the operational requirements specification for the stack analysis component **StackAnalyzer** of **a³**, which determines safe upper bounds for the size(s) of the stack of code snippets given as routines in executables for the PowerPC processors. These upper bounds are output as annotations to call graphs and control-flow graphs of the analyzed program. The annotated graphs can be interactively explored with **AbsInt**'s graph viewer **aiSee**.

Writing and evolution of the document

All the operational requirements are given in textual representation.

The syntax used for naming the requirements is the following:

QK_Component_Id

Component	Indicates the component the test belongs to, e.g. `ais` for a requirement regarding a user annotation, `apx` for a test regarding project file settings, or `stack measurement` for

Summary

- Current safety standards require demonstrating that the software works correctly and the relevant safety goals are met, including non-functional program properties. In all of them, variants of static analysis are recommended or highly recommended as a verification technique.
- Abstract-interpretation–based static analysis tools compute results which hold for any possible program execution and any input scenario. They are in wide industrial use and can be considered state-of-the-art for validating non-functional safety properties.
  - aiT Worst-Case Execution Time Analyzer
  - StackAnalyzer for proving the absence of stack overflows
  - Astrée for proving the absence of runtime errors
- These tools enhance system safety and can contribute to reducing the V&V effort.

AbsInt

info@absint.com

www.absint.com