aiT
Worst-Case Execution Time Analyzer
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

© AbsInt GmbH 2012
Real-Time Systems

- Controllers in planes, cars, plants, etc. are expected to finish their tasks within reliable time bounds.
- It is essential that an upper bound on the execution times of all tasks is known: commonly called the worst-case execution time (WCET), computed at the code level.
- WCET of tasks prerequisite for scheduling analysis at system level (e.g. SymTA/S from Symtavision).
Worst-Case Execution Time

- Best-case execution time
- **Unsafe**: execution time measurement
- **Safe** worst-case execution time estimate
- Exact worst-case execution time

Probability vs. Execution time graph
Modelling Hardware

\[ x = a + b; \]

LOAD r2, _a
LOAD r1, _b
ADD r3, r2, r1

68K (1990)

MPC 5xx (2000)

PPC 755 (2001)

Execution time (clock cycles)

<table>
<thead>
<tr>
<th></th>
<th>Best case</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>68K (1990)</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>MPC 5xx (2000)</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>PPC 755 (2001)</td>
<td>3</td>
<td>321</td>
</tr>
</tbody>
</table>

Execution time depending on flash memory

<table>
<thead>
<tr>
<th></th>
<th>0 wait cycles</th>
<th>1 wait cycle</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>68K (1990)</td>
<td>20</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>MPC 5xx (2000)</td>
<td>3</td>
<td>8</td>
<td>30</td>
</tr>
<tr>
<td>PPC 755 (2001)</td>
<td>3</td>
<td>321</td>
<td></td>
</tr>
</tbody>
</table>
The Timing Problem

- **Timing coverage**: For safe time bounds at the task level, MC/DC coverage is not enough. **Full path coverage** for every possible hardware state is necessary.

- **Hardware complexity**: Caches, pipelines, etc. must be taken into account. **Timing anomalies** create complex scenarios. The appropriate analysis strategy is dictated by the hardware architecture.
  - Example: Cache misses on certain accesses do not necessarily lead to the overall worst-case behavior.
  - Example: Starting with an empty cache does not necessarily lead to the overall worst-case behavior.

- **Predictability**: The hardware must be configured in a predictable way.
  - Bad/no predictability on single cores due to: unified instruction/data caches, FIFO/random caches, caches in write-back mode, etc.
  - Bad/no predictability on multi cores additionally due to: access conflicts on shared caches/memories/flash prefetch buffers, bus conflicts on shared memory buses, etc.
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.
Table 1 — Topics to be covered by modelling and coding guidelines

<table>
<thead>
<tr>
<th>Topics</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Enforcement of low complexity</td>
<td>++</td>
</tr>
<tr>
<td>1b Use of language subsets</td>
<td>++</td>
</tr>
</tbody>
</table>

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:
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.
# Automotive: ISO-26262

<table>
<thead>
<tr>
<th>Methods</th>
<th>ASIL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>1a Walk-through&lt;sup&gt;a&lt;/sup&gt;</td>
<td>++</td>
</tr>
<tr>
<td>1b Inspection&lt;sup&gt;a&lt;/sup&gt;</td>
<td>+</td>
</tr>
<tr>
<td>1c Semi-formal verification</td>
<td>+</td>
</tr>
<tr>
<td>1d Formal verification</td>
<td>0</td>
</tr>
<tr>
<td>1e Control flow analysis&lt;sup&gt;bc&lt;/sup&gt;</td>
<td>+</td>
</tr>
<tr>
<td>1f Data flow analysis&lt;sup&gt;bc&lt;/sup&gt;</td>
<td>+</td>
</tr>
<tr>
<td>1g Static code analysis</td>
<td>+</td>
</tr>
<tr>
<td>1h Semantic code analysis&lt;sup&gt;d&lt;/sup&gt;</td>
<td>+</td>
</tr>
</tbody>
</table>

---

<sup>a</sup> In the case of model-based software development the software unit specification design and implementation can be verified at the model level.

<sup>b</sup> 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.

<sup>c</sup> Methods 1e and 1f can be part of methods 1d, 1g or 1h.

<sup>d</sup> 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:

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:
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)

<table>
<thead>
<tr>
<th>TECHNIQUE/MEASURE</th>
<th>Ref</th>
<th>SIL 0</th>
<th>SIL 1</th>
<th>SIL 2</th>
<th>SIL 3</th>
<th>SIL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Formal Proof</td>
<td></td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>2. Probabilistic Testing</td>
<td></td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3. Static Analysis</td>
<td>A.18</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>4. Dynamic Analysis and Testing</td>
<td>A.12</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>5. Metrics</td>
<td></td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>6. Traceability Matrix</td>
<td>D.68</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>7. Software Error Effect Analysis</td>
<td>D.26</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>8. Test Coverage for code</td>
<td>A.20</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>10. Performance Testing</td>
<td>A.17</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>11. Interface Testing</td>
<td>D.37</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
</tbody>
</table>

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.

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 performance (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.
aiT WCET Analyzer

Combines

- **global static program analysis** by abstract interpretation: microarchitecture analysis (caches, pipelines,...) + value analysis
- integer linear programming for **path analysis** to provide safe and precise bounds on the WCET
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
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 as the 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.