Abstract Interpretation-based Static Analysis Tools:

Proving the Absence of Runtime Errors and Safe Upper Bounds on the Worst-Case Execution Time and Safe Upper Bounds on the Stack Usage

Christian Ferdinand
AbsInt GmbH
2013
AbsInt Angewandte Informatik GmbH

- Provides advanced development tools for embedded systems, and tools for validation, verification, and certification of safety-critical software.

- Founded in February 1998 by six researchers of Saarland University, Germany, from the group of programming languages and compiler construction of Prof. R. Wilhelm. Privately held by the founders.

- Selected Customers:
Background

- 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]
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 & Measurements
    - Test end criteria unclear
    - No full coverage possible
    - "Testing, in general, cannot show the absence of errors." [DO-178B]
Key Products: AI-based Static Analyzers

- aiT WCET Analyzer
  - Proving the correct timing behavior
  - Safe upper bounds on the worst-case execution time of tasks in real-time systems

- StackAnalyzer
  - Excluding stack overflows
  - Safe upper bounds on maximal stack usage of tasks

- Astrée
  - Proving the absence of runtime errors (division by zero, arithmetic overflow, invalid pointer accesses, etc.) in C programs
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.
    - Can find some bugs, but 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, StackAnalyzer, Polyspace Verifier, Astrée.
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.
    - Can find some bugs, but 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, StackAnalyzer, Polyspace Verifier, Astrée.
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.

- “Verification is not simply testing. Testing, in general, cannot show the absence of errors.”
### Table 1 — Topics to be covered by modelling and coding guidelines

<table>
<thead>
<tr>
<th>Topics</th>
<th>ASIL</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1a Enforcement of low complexity</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>1b Use of language subsets&lt;sup&gt;b&lt;/sup&gt;</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
</tbody>
</table>

<sup>b</sup>  
- 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:


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

<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>o</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.
**Table A.9 – Software verification**

(See 7.9)

<table>
<thead>
<tr>
<th>Technique/Measure *</th>
<th>Ref.</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>C.5.12</td>
<td>---</td>
<td>R</td>
<td>R</td>
<td>HR</td>
</tr>
<tr>
<td>2 Animation of specification and design</td>
<td>C.5.20</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
</tr>
<tr>
<td>3 Static analysis</td>
<td>B.6.4 Table B.8</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
<tr>
<td>4 Dynamic analysis and testing</td>
<td>B.6.5</td>
<td>R</td>
<td>HR</td>
<td>HR</td>
<td>HR</td>
</tr>
</tbody>
</table>

**Table B.8 – Static analysis**

(Referenced by Table A.9)

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

**Table A.9 – Software verification**

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

**NOTE 1** See Table C.18.

**NOTE 2** The references “B.x.x.x”, “C.x.x.x” in column 3 (Ref.) indicate detailed descriptions of techniques/measures given in Annexes B and C of IEC 61508-7.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. It is intended the only one of the alternate or equivalent techniques/measures should be satisfied. The choice of alternative technique should be justified in accordance with the properties, given in Annex C, desirable in the particular application.

**Criticality levels:**
- SIL1 (lowest)
- SIL4 (highest)

**Confidence levels:**
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.
Controllers in planes, cars, plants, … are expected to finish their tasks within **reliable time bounds**.

**Schedulability analysis** must be performed.

Hence, it is essential that an upper bound on the execution times of all runnables/tasks is known.

Commonly called the **Worst-Case Execution Time (WCET)**.
The Timing Problem

- Best-case execution time
- Exact worst-case execution time
The Timing Problem

Probabilistic Execution Time

Unsafe: execution time measurement

Exact worst-case execution time

Best-case execution time
The Timing Problem

Execution time

Exact worst-case execution time

Safe worst-case execution time estimate

Unsafe: execution time measurement

Best-case execution time

Probability

Execution time

AbsInt
The Timing Problem

Unsafe: execution time measurement

Safe worst-case execution time estimate

Probability

Execution time
aiT WCET Analyzer Internal Structure
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.

Specifications (*.ais)
- Entry Point
- Worst Case Execution Time
- Visualization, Documentation
aiT WCET Analyzer Advantages

- WCET results determined automatically
- Valid for all inputs and all execution scenarios
- No modification of your code or tool chain required
- Integration into development tool chain
- Application areas
  - Timing verification
  - Feedback for optimization
  - Software integration
  - Architecture exploration
Stack Usage

- In safety-critical embedded systems the **Stack** is typically the only **dynamically** managed memory.

- The stack is used to store:
  - Local variables
  - Intermediate values
  - Function parameters
  - Function return addresses
Stack Usage Analysis

- Stack space has to be reserved at configuration time => maximal stack usage has to be known.

- Underestimating stack usage can cause stack overflows.
  Stack overflows are severe errors:
  - they can cause wrong reactions and program crashes,
  - they are hard to recognize,
  - they are hard to reproduce and fix.
Testing is difficult

- A traditional approach
  - Fill the stack area with a pattern (0xAAAA)
  - Let the system run for a long time
  - Monitor the maximum stack usage so far

- Error-prone and expensive!
  - Typical stack usage of a task can be very different from maximum stack usage. Dynamic testing typically cannot guarantee that the worst case stack usage has been observed.
Solution: StackAnalyzer

- **StackAnalyzer** computes save upper bounds of the stack usage of the tasks in a program for all inputs.
- Static program analysis based on Abstract Interpretation.

Executable (elf, coff, ...)

Function pointers, recursion depths, ...

Instruction "_main" + 1 computed calls 
"_fooA", "_fooB", "_fooC";
Routine "_fib" incarnates max 5;

Entry Points

StackAnalyzer

Stack Usage
- Visualization
- Documentation
StackAnalyzer: Static Stack Usage Analysis

- StackAnalyzer is an Abstract Interpretation based static analyzer which calculates safe and precise upper bounds of the maximal stack usage of the tasks in the system.

- It can prove the absence of stack overflows:
  - on binary code
  - without code modification
  - without debug information
  - taking into account loops and recursions
  - taking into account inline assembly and library function calls
StackAnalyzer Advantages

- Results are determined automatically, valid for all inputs and all execution scenarios.

- No modification of your code or tool chain required. Code is analyzed as executed in the final system.

- Stack optimization/Software integration.

- Successfully used for certification according to DO-178B/Level A.

- Available for various processor/compiler combinations.
The Static Analyzer Astrée

- Crashes or undefined behavior due to runtime errors are bad and 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-around
  - Floating point overflows and invalid operations (IEEE floating values Inf and NaN)
  - User-defined assertions, unreachable code, uninitialized variables
- Recommended: C programs without dynamic memory allocation and recursion.

```c
int main()
{
    char ArrayBlock[10];
    char *ptr;
    ptr = &ArrayBlock[0];
    *(ptr+9)=9;
    *(ptr+10)=10;
    return 0;
}
```

ALARM: invalid dereference: dereferencing 1 byte(s) at offset(s) 10 may overflow the variable ArrayBlock of byte-size 10 at [...]
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 worst-case of all possible rounding modes.

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

```c
#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
Alarm Analysis: Example

```c
#define BASE 0x80000000
#define OFFSET 0x38343031
volatile int SwitchPosition;
int main()
{
    int MODULE_NUMBER = BASE + OFFSET;
    char sp = SwitchPosition;
}
```
The Zero Alarm Goal

- With zero alarms, absence of runtime errors is **automatically** proven by the analysis run, 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
- 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
Other Issues in Safety Standards handled by Astrée

- Control flow analysis
- Data flow analysis
- Static code analysis
- Semantic code analysis
- Soon: Shared variables
- Soon: Coding rule checker (MISRA C)
Astrée Advantages

- **Low number of false alarms** by high analysis precision.
  - Special support for real-time systems, digital filters, ...
  - Floating-point rounding errors taken into account
- **Efficient elimination of false alarms**
  - High analysis speed
  - Local tuning of analysis precision
  - Efficient support for alarm analysis (intuitive GUI, variable values per context, ...)
- **High reliability.** No alarms shadowed (no "green follows orange")
- Seamless integration into development environment
- Flexible licensing models
- Qualified tool support (CET) and analysis service
Esterel SCADE + aiT/StackAnalyzer

SCADE

Traceability File (XML)

Generated C Code

External C Code

SCADE-aiT coupling

Analysis Results (XML)

aiT

Executable Code

Cross Compiler

Esterel SCADE + aiT/StackAnalyzer
GUI Integration
SCADE: Model-Level Feedback

**WCET and Space Analysis Result of Session 2**

**Root Node:** CruiseControl::CruiseControl
**Optimization Level:** 2
**Root reset function WCET:** 81
**Root cycle function WCET:** 1020

**Cycle functions**

<table>
<thead>
<tr>
<th>SCADE Path</th>
<th>Calls</th>
<th>WCET (sum)</th>
<th>WCET (max)</th>
<th>WCET (avg)</th>
<th>CWET (sum)</th>
<th>CWET (max)</th>
<th>CWET (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CruiseControl::CruiseControl</td>
<td>1</td>
<td>537</td>
<td>537</td>
<td>537.00</td>
<td>1020</td>
<td>1020</td>
<td>1020</td>
</tr>
<tr>
<td>CruiseControl::CruiseRegulation</td>
<td>1</td>
<td>188</td>
<td>188</td>
<td>188.00</td>
<td>277</td>
<td>277</td>
<td>277</td>
</tr>
<tr>
<td>CruiseControl::CruiseSpeedMgt</td>
<td>1</td>
<td>138</td>
<td>138</td>
<td>138.00</td>
<td>158</td>
<td>158</td>
<td>158</td>
</tr>
<tr>
<td>CruiseControl::SaturateThrottle</td>
<td>1</td>
<td>79</td>
<td>79</td>
<td>79.00</td>
<td>79</td>
<td>79</td>
<td>79</td>
</tr>
<tr>
<td>pwlinear::LimitUnSymmetrical</td>
<td>1</td>
<td>20</td>
<td>20</td>
<td>20.00</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>linear::Gain</td>
<td>2</td>
<td>10</td>
<td>5</td>
<td>5.00</td>
<td>10</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**Reset functions**

<table>
<thead>
<tr>
<th>SCADE Path</th>
<th>Calls</th>
<th>WCET (sum)</th>
<th>WCET (max)</th>
<th>WCET (avg)</th>
<th>CWET (sum)</th>
<th>CWET (max)</th>
<th>CWET (avg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CruiseControl::CruiseControl</td>
<td>1</td>
<td>65</td>
<td>65</td>
<td>65.00</td>
<td>81</td>
<td>81</td>
<td>81</td>
</tr>
<tr>
<td>CruiseControl::CruiseRegulation</td>
<td>4</td>
<td>47</td>
<td>14</td>
<td>11.75</td>
<td>47</td>
<td>14</td>
<td>11.75</td>
</tr>
<tr>
<td>CruiseControl::CruiseSpeedMgt</td>
<td>2</td>
<td>17</td>
<td>9</td>
<td>8.50</td>
<td>17</td>
<td>9</td>
<td>8.50</td>
</tr>
</tbody>
</table>

**Abbreviations**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WCET (sum)</td>
<td>WCET contribution of the function without descendants; i.e. sum of the function WCETs computed for each context.</td>
</tr>
<tr>
<td>WCET (max)</td>
<td>WCET of the function without descendants; i.e. max of the function WCETs computed for each context.</td>
</tr>
<tr>
<td>WCET (avg)</td>
<td>WCET average of the function without descendants.</td>
</tr>
</tbody>
</table>

**Function CruiseControl::CruiseRegulation**

**Cycle function**

- **Calls:** 1
  - WCET (sum): 277 (27.16%)
  - WCET (max): 188 (18.43%)
  - Cumulative WCET (max): 277 (27.16%)
  - WCET (avg): 277.00

**Reset function**

- **Calls:** 4
  - WCET (sum): 47 (58.02%)
  - WCET (max): 14 (17.28%)
  - Cumulative WCET (max): 14 (17.28%)
  - WCET (avg): 11.75
dspace TargetLink + Astrée

TargetLink Code Generator

Data Dictionary

Generated C Code

Preprocessor

Preprocessed C Code

TargetLink-aiT Coupling

Analysis Results (XML)

Astrée

Integratio n C Code
void wrapper(){
    while (1){
        subsys_function();
    }
}
Tool Coupling Advantages

- **Automatic transfer** of model-level information to implementation-level tools
- Analysis results are **conveniently accessible** from the modeling level
- **Direct feedback** on the effect of design decisions on resource usage
- **Automatic Verification**

Cost savings estimation during European IST research project INTEREST on real-life software for SCADE-aiT coupling. Main findings:
- SCADE-Workflow with aiT coupling saves **2 days per KLOC**
- In average, for a 19 KLOC SW, 40 days are saved, which represent **66%** of the total effort in the classical approach.
The Confidence Argument

- Absence of hazards has to be shown with adequate confidence: the evidence provided can be trusted beyond reasonable doubt.
- Abstract Interpretation is a formal verification method enabling provably sound analyses to be designed.

Reasoning strategy:
1. Soundness proof of mathematical analysis specification.
2. Automatic generation of analyzer implementation from mathematical specification, enabling high implementation quality.
3. Empirical validation of chosen abstraction, i.e., analysis model.
5. Qualification Software Life Cycle Data reports: soundness of tool development and validation process.
Qualification Support Kits

Tool Operational Requirements a3 for PowerPC

<table>
<thead>
<tr>
<th>Date</th>
<th>December 08, 2011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Final</td>
</tr>
<tr>
<td>Reference</td>
<td>a20111208</td>
</tr>
<tr>
<td>Baseline</td>
<td>Revision: 173159</td>
</tr>
</tbody>
</table>

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 asee.

Writing and evolution of the document

All the operational requirements are given in textual representation.

Verification Test Plan a3 for PowerPC

<table>
<thead>
<tr>
<th>Date</th>
<th>December 08, 2011</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status</td>
<td>Final</td>
</tr>
<tr>
<td>Reference</td>
<td>a20111208</td>
</tr>
<tr>
<td>Baseline</td>
<td>Revision: 173159</td>
</tr>
</tbody>
</table>

Introduction

Purpose of the document

This document describes all tests verifying the operational requirements for a3 StackAnalyzer for PowerPC listed in the Tool Operational Requirements Report (TOR).

Each test case description illustrates:

- its dependencies on other test cases,
- how the test can be performed,
- what requirements from the requirements report are covered,
- the test objective,
- the test environment (described in the Tool Operational Requirements Report (TOR)) and
- the expected results of the test.

The test objectives are:

- verifying the conformity of the tool with its operational specification and
- provide the traceability matrix between operational requirements and test cases.
Airbus

For many years Airbus France has been using aIT, StackAnalyzer and Astrée in the development of safety-critical avionics software for several airplane types. The analyzers were used as part of their certification strategy to demonstrate compliance to DO-178B, up to Level A.

Honda

aIT, StackAnalyzer and ValueAnalyzer have been successfully used by Honda in the FADEC software development of an aircraft turboprop engine.

ESA

In April 2008, Astrée was used to prove the absence of any real-time errors in a C version of the automatic docking software of the Jules Verne Automated Transfer Vehicle (ATV), enabling ESA to transport payloads to the International Space Station. The analysis was performed completely automatically.

Daimler

Daimler has been successfully using aIT and StackAnalyzer in many automotive software projects, including the powertrain control system of the new Actros truck.

OHB

OHB successfully uses aIT and StackAnalyzer in the development of onboard software for the mission success of the SmallGEO satellite platform for geostationary communication satellites and the GALILEO FOC platform for satellite navigation.

Toyota Unintended Acceleration Investigation

In 2010, aIT was used by NASA as an industry-standard static analysis tool for demonstrating the absence of timing-related software defects in the Toyota Motor Corporation Unintended Acceleration Investigation.

MTU Aero Engines

MTU successfully uses aIT, StackAnalyzer and Astrée to demonstrate the correctness of control software for emergency power generators in power plants. The tools in combination with their qualification packages (QSK and QSLCD) are part of the certification process according to the IEC60880.
Summary

- Current safety standards also require to demonstrate non-functional program properties. In all of them, variants of static analysis are recommended or highly recommended as a verification technique.

- AI-based static analyzers for non-functional properties are increasingly used in industry and can be considered as the state of the art.
  - aiT Worst-Case Execution Time Analyzer
  - StackAnalyzer
  - runtime error analyzer Astrée

- They can easily be integrated into (model-based) development processes and can contribute to reducing the...