Automatic Qualification of Abstract Interpretation-based Static Analysis Tools

Christian Ferdinand, Daniel Kästner
AbsInt GmbH
2013
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/DO-178C]
  - Formal technique: **Abstract Interpretation**
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, CodeSonar, ...
  - 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.
**Abstract Interpretation (AI)**

- 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**.

---

**Concrete semantics**

**Abstract semantics**

- Definitely correct / in time
- Definitely false
- Potentially false
AI – Industry Perspective

- Abstract Interpretation-based static analyzers are in wide industrial use: state-of-the-art for validating non-functional safety properties.
- Examples:
  - Static WCET and memory usage analysis (aiT, StackAnalyzer)
  - Static runtime error analysis (Astrée)

- aiT and StackAnalyzer are in wide use by avionics companies, e.g., for safety-critical Airbus software in many airplane types (A380, ...).
- The aiT WCET Analyzer has been used by NASA as an industry-standard tool for demonstrating the absence of timing-related software defects in the Toyota Motor Corporation 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.
Static WCET Analysis

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 worst-case execution time (WCET).

Application Code

Specifications (*.ais)

Compiler

Linker

Executable (*.elf / *.out)

Worst Case Execution Time: 572 cycles

- Worst Case Execution Time
- Visualization, Documentation
Static Stack Usage Analysis

- The required stack space has to be reserved for each task at configuration time => **maximal stack usage** has to be **statically known**.
- **Underestimating** the maximal stack usage can cause **stack overflows**.
- **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.
Astrée: Runtime Error Analysis

- AI-based static analyzer to **prove the absence of runtime errors** in C99 code.
- Astrée detects **all runtime errors** with **few false alarms**:
  - Array index out of bounds
  - Integer/floating-point division by 0
  - Invalid pointer dereferences
  - Arithmetic overflows and wrap-aro****
  - Floating point overflows and invalid operations (IEEE floating values Inf and NaN)
  - User-defined assertions, unreachable code, uninitialized variables
- Efficient support for **alarm analysis** (variable values, contexts, ...).
- Elimination of false alarms by local tuning of analysis precision.
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.
4. **Qualification Support Kits**: demonstrating implementation correctness in operational context of tool users.
5. **Qualification Software Life Cycle Data reports**: soundness of tool development and validation process.
Theory & Soundness Proofs

- **Abstract Interpretation**

- **aiT/StackAnalyzer**

- **Astrée**
Generating Program Analyzers

- **Program Analyzer Generator PAG**: generates efficient data flow analyzers from concise mathematical specification.
  - Binary-level analyzers: *variety of processors* to be supported.

![Diagram showing the workflow of generating program analyzers]

- Instruction set / processor specification: formats, instruction classes, name space definitions (NET)
- Analysis specification:
  - Domains
  - Transfer functions
  - Join functions
  - Interface

- Executable or assembly
  - *exec2crl*
  - CFG in CRL
  - Analysis 1
  - Analysis 2
  - Analysis 3
  - C code

- PAG: Program Analyzer Generator
Model Validation

- Especially important for **WCET analysis**: analyzer operates on processor model. It has to be shown that model conservatively approximates behavior of physical processor.

- Basic validation:
  - compare analytically computed WCET bounds $T_A$ for code snippets or representative programs with measured times $T_M$.
  - $T_A \geq T_M$ must always hold.
  - Local underestimations might be shadowed by overestimations in other parts, therefore additional validation required.

- In-depth validation:
  - aiT result includes prediction of all possible execution paths with all potential hardware states.
  - Different hardware states correspond to different observable events.
  - All observed events must be predicted by model.
Model Validation (2)

- Observable Events:
  - **Performance monitoring**: clock ticks, cache misses, dispatched instructions, mispredicted branches
  - **Bus traces**: requests generated by the core illustrate branch prediction behavior and other advanced pipeline features.
  - **Instruction traces** (e.g. NEXUS): every instruction emits an event at beginning and end of its execution.

- Prediction graph:
  - aiT model can be configured to **emit abstract traces** for all these event types.
  - **Prediction graph** is sound overapproximation of all possible traces of events observable in reality.
  - Any **measured event trace** has to be **part of prediction graph**.
Automatic Trace Validation

- Measure execution behavior on physical hardware and create traces
- Determine prediction graph from aiT analysis
- Check whether measured event trace is contained in prediction graph
- Validation successful iff there is path in prediction graph covering all events in exactly the same order in which they have been observed.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Application Type</th>
<th>Binary Size</th>
<th>Event Types</th>
<th>Trace Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>M68020</td>
<td>Avionics</td>
<td>14 MB</td>
<td>bus</td>
<td>4.232.000</td>
</tr>
<tr>
<td>MPC 5xx</td>
<td>Avionics</td>
<td>3 MB</td>
<td>instruction</td>
<td>3.879.000</td>
</tr>
<tr>
<td>MPC 755</td>
<td>Avionics</td>
<td>120 MB</td>
<td>bus</td>
<td>9.468.000</td>
</tr>
</tbody>
</table>

*aiT trace validation data for exemplary architectures*
Qualification Support Kits

- **Goal:** demonstrate that the tool *works correctly* in the *operational context* of the user.

1. **Report Package**
   - Tool Operational Requirements (TOR)
     - Low-level requirements to tool behavior under normal operating conditions.
     - Tool operational context, conditions for obtaining valid results, tool restrictions
   - Verification Test Plan (VTP)
     - Defines test cases for demonstrating all requirements specified in the TOR, including test setup, and structural and functional description of each test case.

2. **Test Package**
   - Extensible set of test cases, including all test cases specified in the VTP.
   - GUI providing convenient access to analyzer results and support for *fully automatic execution and evaluation* of all test cases.
   - Result of tool qualification run can be stored together with other certification documents.
Qualification Support Kits

Tool Operational Requirements a3 for PowerPC

Date: December 08, 2011
Status: Final
Reference: a20111208
Baseline: Revision: 173159

Introduction

Purpose of the document

This document describes the operational requirements specification for the stack analysis component StackAnalyzer of a tool that 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 atiSee.

Writing and evolution of the document

All the operational requirements are given in textual representation.
Qualification Software Life Cycle Data

- Goal: demonstrate that tool development process fulfills safety demands, e.g., regarding quality assurance, traceability, requirements engineering and verification activities.

- Document structure meets requirements of the DO-178B standard, but is applicable to other safety standards, as well.

- Available documents:
  - Software Development Plan (SDP)
  - Software Configuration Management Plan (SCMP)
  - Software Quality Assurance Plan (SQAP)
  - Software Verification Plan (SVP)
  - Software Verification Results (SVR)
  - Compliance Certificate (CC)
Conclusion

- Current safety standards require to demonstrate 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 is formal method for statically verifying dynamic program properties. It defines the state of the art for validating non-functional software properties: WCET, stack usage, absence of runtime errors.

- Confidence in correctness of analysis results:
  - Rigorous mathematical analysis theory
  - Soundness proofs of individual analysis specification
  - Automatic generation of analyzer implementation
  - Model validation, e.g., by automatic trace validation
  - Qualification Support Kits: tool operational requirements satisfied in operational context of tool user
  - Qualification Software Life Cycle Data: demonstrate that tool development process is compliant to safety requirements
email: info@absint.com
http://www.absint.com