Lecture 11

“Non”-Functional Software Verification

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]

- Formal technique: Abstract Interpretation

Required by
- DO-178B
- DO-178C
- ISO-26262
- EN-50128
- IEC-61508
Abstract Interpretation in Industry

Worst-Case Execution Time

Worst-Case Stack Usage

Runtime Errors
Real-Time Systems

- Controllers in planes, cars, plants, ... 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 **Worst-Case Execution Time**, computed at the code level.
- **WCET** of tasks prerequisite for scheduling analysis at **system-level**.
Two Levels of Timing Analysis

- **Code level**
  - **Single** process, task, ISR
  - Focus on
    - Control flow
    - Processor architecture with pipelines and caches
    - WCET

- **System level**
  - **Multiple** functions or tasks
  - Focus on
    - Integration and scheduling
    - End-to-end timing
    - Worst-Case Response Time (WCRT)

\[
R_i = C_i + \sum_{j \in SPT} C_j \left( \frac{R_j}{T_j} \right) \leq D_i = T_i
\]

- Fixed-point problem
- Response time
  - # of preemptions
  - Interference

- aiT (AbsInt)
- SymTA/S (Symtavision)
The Code Level: Methodology Overview

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

Execution time
The Timing Problem

x = a + b;

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

The Timing Problem

- **Timing coverage**: For safe time bounds at the task level, full path coverage for every possible hardware state is necessary. This can be achieved by Abstract Interpretation based static analysis.

- **Hardware predictability**: 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 the empty cache does not necessarily lead to the overall worst-case behavior

- **Composability**: Unpredicted timing interferences at the microcontroller level (switching costs) between different tasks impair composability and endanger successful software integration.
Cache

- The cache is a fast memory on chip.
- When the CPU wants to read/write at memory address \( a \), it sends a request for \( a \) to the bus.
- If the block \( m \) containing \( a \) is in the cache (hit), the request for \( a \) is served in the next cycle.
- If \( m \) is not in the cache (miss), it is transferred from main memory to the cache replacing some block in the cache. The request for \( a \) is served asap while the transfer still continues.
- Several replacement strategies: LRU, PLRU, FIFO, ... determine which line to replace.
Example: Direct Mapped I-Cache

<table>
<thead>
<tr>
<th>CPU</th>
<th>I-Cache</th>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter: 1028</td>
<td>1032: ble 1024</td>
<td>1024: add ...</td>
</tr>
<tr>
<td>Instruction:   mul 1024</td>
<td>1028: mul ...</td>
<td>1028: mul ...</td>
</tr>
<tr>
<td></td>
<td>1032: ble 1024</td>
<td>1032: ble 1024</td>
</tr>
</tbody>
</table>

Cache Hit: ~ 1 Cycle

Cache Miss: ~ +1 to +100 Cycles
### A-way Set Associative Cache

**CPU**

**Address:**
- **Address prefix**
- **Set number**
- **Byte in line**

<table>
<thead>
<tr>
<th>Set number</th>
<th>Byte in line</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- **Adr. prefix**
- **Tag**
- **Rep**
- **Data block**

<table>
<thead>
<tr>
<th>Set: Fully associative subcache of A elements with LRU, FIFO, rand. replacement strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

- **Compare address prefix**
- **If not equal, fetch block from memory**

- **Byte select & align**

**Main Memory**

**Data Out**
Cache Analysis

- **Must** Analysis:
  For each program point and calling context, find out which blocks *are* in the cache

- **May** Analysis:
  For each program point and calling context, find out which blocks *may be* in the cache
Example:
Fully associative data cache (2 elements) with LRU
## Result of the Cache Analyses

### Categorization of memory references

<table>
<thead>
<tr>
<th>Category</th>
<th>Abb.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>always hit</td>
<td>ah</td>
<td>The memory reference will always result in a cache hit.</td>
</tr>
<tr>
<td>always miss</td>
<td>am</td>
<td>The memory reference will always result in a cache miss.</td>
</tr>
<tr>
<td>not classified</td>
<td>nc</td>
<td>The memory reference could neither be classified as <strong>ah</strong> nor <strong>am</strong>.</td>
</tr>
</tbody>
</table>
Abstract Semantics: Transfer

concrete

```
| z | y | x | t |
```

abstract
```
| {x} | {s, t} | {y} |
```

```
| s | z | y | x |
```

```
| s | z | y | t |
```

[ s ]

“young”

Age

“old”

{ x }

{ }
Abstract Semantics: Join

Join (must)

{a}
{c, f}
{d}
{c}
{e}
{a}
{d}
{c, e}
{d}
{a, c}
{d}

"intersection + maximal age"

Interpretation: memory block a is definitively in the (concrete) cache
=> always hit

Question: How many references will a memory block surely survive in the cache?
Abstract Semantics: Join

Join (may)

Question: How many references can a memory block maximally survive in the cache?
aiT WCET Analyzer Structure
aiT WCET Analyzer

Combines
- global static program analysis by Abstract Interpretation: microarchitecture analysis (caches, pipelines, ...) + value analysis
- integer linear programming for path analysis in a single intuitive GUI.
Abstract Interpretation in Industry

- Worst-Case Execution Time
- Worst-Case Stack Usage
- Runtime Errors
Stack

- 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.

- Overestimating the maximal stack usage means waisting resources.
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.
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
Abstract Interpretation in Industry

Worst-Case Execution Time

Worst-Case Stack Usage

Runtime Errors
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-arounds
  - 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

Types of Runtime Errors (1)

- 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:

```c
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 reaction:**
  - **reports alarm (type A)** in order to signal a **potential runtime error**,  
  - **continues analysis** for scenarios where the runtime error **did not occur**.  
  - **Alarm type A**: contexts without continuation are pruned \(\Rightarrow\) Astrée reports an **error** and reports: **Analysis stopped for this context.**
Types of Runtime Errors (2)

- Runtime Errors causing **unspecified, but predictable behavior**:
  - Integer overflow
  - Invalid shifts <<, >>, or casts, ...

- Astrée reaction:
  - reports **alarm (type C)** in order to signal potential runtime error and
  - continues analysis with an overapproximation of all possible results.

  - No artificial restrictions on value ranges, so results are **always safe**.

```c
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));
}
```

"Compute-through-overflow" arithmetics.

Overflow detected in signed short -> unsigned short conversions.

Nevertheless:
precise range for z on two's complement hardware (configurable).
The Zero Alarm Goal

- With zero alarms, the 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
    - Alarms and error messages are linked: jump to location per 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, e.g. by semantic loop unrolling
    - making **external knowledge** available to Astrée
Constant Propagation

volatile int COND;
int main()
{
    int c = COND;
    int x=0, y=0;
    if (c<10) {
        x = 4;
        y = 6;
    }
    else {
        x = 12;
        y = 14;
    }
}
Interval Analysis

volatile int COND;
int main()
{
    int c = COND;
    int x=0, y=0;
    if (c<10) {
        x = 4;
        y = 6;
    } else {
        x = 12;
        y = 14;
    }
}
Octagon Analysis: $X \pm Y \leq c$

```c
volatile int COND;
int main()
{
    int c = COND;
    int x=0, y=0;
    if (c<10) {
        x = 4;
        y = 6;
    } else {
        x = 12;
        y = 14;
    }
}
```

- $(x,?), (y,?), (c,?)$
- $(x,?), (y,?), (c,[\text{MIN\_INT,MAX\_INT}])$
- $(x=0, \ y=0, \ x-y=0, \ x+y=0, \ c>=\text{MIN\_INT, c}<=\text{MAX\_INT})$
- $(x=0, \ y=0, \ x-y=0, \ x+y=0,...)$
- $(x=4, \ y=0, \ x-y=4, \ x+y=4,...)$
- $(x=4, \ y=6, \ x-y=-2, \ x+y=10,...)$
- $(x=0, \ y=0, \ x-y=0, \ x+y=0,...)$
- $(x=12, \ y=0, \ x-y=12, \ x+y=12,...)$
- $(x=12, \ y=14, \ x-y=-2, \ x+y=26,...)$
- $(4<=x<=12, \ 6<=y<=14, \ x=y-2, \ 10<=x+y<=26,...)$
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 $[-1.34218e+08, 1.34218e+08]$
- $r2 = 2.0$
int main()
{
    static int init = 0;
    int i=0;
    float x=0.0, div=0.0;
    while (i<10) {
        if (init) {
            x+=x/div;
        }
        else {
            init = 1;
            x = 1.0;
            div = 2.0;
        }
        i++;
    }
}

- unroll=0:
  - One invariant for all loop iterations: $i \in [0,9]; \ init \in [0,1]; div \in [0.0,2.0]$  
  - **false alarm: potential runtime error** in line 8: floating-point division by 0.
- unroll=1:
  - One invariant for first loop iteration: at entry: $i \in \{0\}; \ init \in \{0\}$
  - One invariant for all other loop iterations: $i \in [1,9]; \ init = 1; div = 2.0$
  - **No alarms reported.**
- Astrée default: unroll=3
- Automatic context-sensitive unrolling
- Local control: __ASTREE_unroll