Static Verification of Dynamic Data Structures in Critical Embedded Systems

Pascal Sotin
ENS Paris

CITI, Lyon – April 12, 2012
Program Verification

- Need for verification of embedded critical software
  - Bugs are human life threats and economic issues
  - Huge amount of work for certification (avionic standard: DO-178C)

- Programs in embedded systems are...
  - getting bigger (reaching million lines of code in an A380)
  - getting more complex (e.g., modern interfaces)
    ⇒ Need for computer-assisted verification

- One aspect: verifying the absence of runtime errors
Definition (Runtime Error)

A runtime error is a behavior of the program which causes it to be killed by the operating system or to corrupt its own data.

Example

- Using integers: division by 0
- Using arrays: out of bound access
- Using call-stack: stack overflow
- Using heap-allocated structures: null pointer dereference
Static Analysis of C Programs

- Testing may find some bugs, not all
- Static analysis tools
  - Should be sound i.e. no bug should escape
  - May issue false alarm
  - Should scale to large program codes
- Static analysis tools used in avionics
  - **Coverity, CodeSonar**
    - Unsound: cannot prove the absence of bugs, can still find bugs
  - **Polyspace Verifier**
    - Sound, generalist
    - Might issue a lot of warnings
  - **Astrée**
    - Sound, Specialized for embedded systems
    - Issue little or no warning
Contributions

- Data-structure beyond the reach of existing tools

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>0x0</th>
<th>invalid elt</th>
<th>invalid elt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>8</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

- Found in critical avionics programs
- The array is statically allocated
- The list is ordered according to a priority field

Contributions (joint work with X. Rival, ENS)

- A hierarchical abstract domain (representation of this kind of structures)
- An implementation within the tool MemCAD
  - Combination of state-of-the-art analyses
  - Modular implementation (no monolithic ad-hoc solution)
Introduction

Program Verification
Static Analysis of C Programs

Static Analysis with MemCAD
Concrete Semantics
The MemCAD Static Analyzer
Memory Abstraction
Abstract Semantics

Hierarchical Abstract Domain
Memory Abstraction
Abstract Semantics
Implementation in MemCAD

Conclusion
Outline

1. Introduction
   Program Verification
   Static Analysis of C Programs

2. Static Analysis with MemCAD
   Concrete Semantics
   The MemCAD Static Analyzer
   Memory Abstraction
   Abstract Semantics

3. Hierarchical Abstract Domain
   Memory Abstraction
   Abstract Semantics
   Implementation in MemCAD

4. Conclusion
Concrete Semantics

- Usual semantics of programming languages:
  - Described in books
  - Implemented by compilers and architectures
- We rely on a formal semantics of ANCI C99
  - Describes mathematically the behaviors of programs
  - Supports non-determinism
  - Parameterized by the Application Binary Interface (ABI)
Collecting Semantics (1)

- Runtime errors are determined by the reachable states of the tr. system
  - No need to consider traces (Contrarily to liveness properties)
  - We reason with the collecting semantics

Collecting semantics

Let $\mathcal{L}$ be the set of program points. Let $\mathcal{M}$ be the set of memory states. The collecting semantics $F$ of the program belongs to:

$$\mathcal{L} \rightarrow \mathcal{P}(\mathcal{M})$$

- We need to prove disjointness of reachable and error states
Collecting Semantics(2)

- Example of collecting semantics computation
  - Assume a loop with guard $G$ and body $\text{Body}$

  \[
  F(Hd) = F(In) \\
  \cup \llbracket \text{Body} \rrbracket \circ \llbracket G = true \rrbracket (F(Hd))
  \]

  \[
  F(Out) = \llbracket G = false \rrbracket (F(Hd))
  \]

- Impossibility of an exact computation
  - The sets of states are huge
  - Generally, this problem is undecidable

- Use Abstract Interpretation to solve this issue

Data Structures in Critical Embedded Systems
The MemCAD Static Analyzer

Main features

- Static analysis tool for C programs
- Specialized for the analysis of heap-allocated structures
  - Shape analysis: abstract precisely the memory given by \texttt{malloc}
  - Based on the separation logic
- Embeds a value analysis

Technical characteristics

- Implemented in OCaml
- Relying on the numerical abstract domain library \texttt{APRON}
- Modular construction
  - Reusable components
  - Several trade-offs precision/efficiency
Memory Abstraction (1)

- Abstract memories ($\mathcal{M}^\#$) abstract set of memories $\mathcal{P}(\mathcal{M})$

**Example**

An abstract memory is made of:
- A shape graph
  - An edge abstracts a block of memory
  - A node abstracts a value
- An environment (eg. &hd)
- Constraints on the values (eg. 0x0)
Memory Abstraction (2)

- Structures are summarized according to their inductive definition
  - Depicted by thick edges in the shape graph

Example

- These edges can be unfolded for memory accesses
Abstract Semantics: Principle

- The abstract semantics $F^\#$ rely on abstract operations

$$F^\#(Hd) = F^\#(ln) \quad \square[\llbracket Body\rrbracket^\# \circ [G = true]^\#(F^\#(Hd))]$$

$$F^\#(Out) = [G = false]^\#(F^\#(Hd))$$

- Abstract operations are over-approximation of the concrete ones

- By construction, $F^\#$ is an over-approximation of $F$
  - $F^\#$ can be computed efficiently
  - No runtime error is forgotten
  - False alarms might appear
Abstract Semantics: Join operator

- Join operator
  - $\sqcup : \mathcal{M}^\# \times \mathcal{M}^\# \rightarrow \mathcal{M}^\#
  - Over-approximation of the union operator $\cup$ on $\mathcal{P}(\mathcal{M})$

**Example**

Computed in two steps
- Join of the shape graphs
- Join of the constraints on the values
Outline

1 Introduction
   Program Verification
   Static Analysis of C Programs

2 Static Analysis with MemCAD
   Concrete Semantics
   The MemCAD Static Analyzer
   Memory Abstraction
   Abstract Semantics

3 Hierarchical Abstract Domain
   Memory Abstraction
   Abstract Semantics
   Implementation in MemCAD

4 Conclusion
The data-structure we consider is a list within an array

<table>
<thead>
<tr>
<th></th>
<th>0x0</th>
<th>invalid elt</th>
<th>invalid elt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

Static analysis issue

- How can we abstract this structure?
- How can we infer automatically that a program builds this structure?
Memory Abstraction: Arrays

- Arrays can be represented using shape graphs
  - An array can be represented by a single edge
  - An array can be represented by the edges of its cells

(a) Concrete array.

(b) Pair of edges.
\[
\begin{align*}
\alpha & \quad 2 \\
& \quad 4 \\
\beta_0 & \quad \beta_1
\end{align*}
\]
\[
\begin{align*}
\beta_0 &= 24 \\
\beta_1 &= 48
\end{align*}
\]

(c) Single edge.
\[
\begin{align*}
\alpha & \quad 0 \\
& \quad 4 \\
\beta
\end{align*}
\]
\[
\begin{align*}
\beta &= 24 \times 65536 + 48
\end{align*}
\]

The contents need to be split and merged
Memory Abstraction: Not a solution (1)

- Idea: splitting the array and remembering the values

\[ \begin{align*}
\beta_0 &= \alpha + 8 \\
\beta_1 &= \alpha + 16 \\
\beta_2 &= 0x0 \\
\beta_0 &= \alpha + 16 \\
\beta_1 &= 0x0 \\
\beta_2 &= \alpha + 8 \\
\beta_0 &= \alpha + 8 \
\end{align*} \]

\( \beta = \alpha + 8k \)

- Pros: the memory describes a list
- Cons: the abstraction size is exponential in the length of the array
Memory Abstraction: Not a solution (2)

- Idea: splitting the array and abstracting the values

\[ \alpha \land \beta_0, \beta_1, \beta_2 \in \{\alpha, \alpha + 8, \alpha + 16, \ldots, 0x0\} \land \exists k \in \mathbb{N}, \beta = \alpha + 8k \]

- Pros: reasonable size
- Cons: the list invariant is lost
Memory Abstraction: Solution

- We represent the structure with a hierarchical abstract domain
  - The main memory contains an array
  - The content of the array is itself a memory

\[
&\text{hd} \rightarrow \alpha_{800} \rightarrow \beta \\
\land \beta = \text{list}
\]

- Advantages:
  - Captures that the content is a list
  - Abstraction size independent of the size of the array
Memory Abstraction: Features

- **Symbolic offsets**
  - Allows symbolic access to the content of the array
  - Provides compact loop invariants

- **Sub-memories as values**
  - Described by a shape graph
  - Interfaced through an environment

- A common numerical abstract domain
Abstract Semantics: Reading

We illustrate the abstract read access to `hd->next`;

Step 1: read the content of `hd`

\[ \Rightarrow \alpha + 8\delta_1 \]

Step 2: read the content at address \( \alpha + 8\delta_1 \)

- We show \( 0 \leq \delta_1 < \delta_2 \)
- Failure to show that \( 0 \leq \delta_1 < 100 \) would raise an alarm

\[ \Rightarrow \text{delegating to sub-memory } \beta \]
Abstract Semantics: Reading (Cont’d)

- Step 3: read the content of $\alpha + 8\delta_1$ in $\beta$
  - An inductive edge is found: unfolding is needed

$$\beta = \alpha + 8\delta_1 \rightarrow \alpha + 8\delta' \rightarrow \alpha + 8\delta_2 \rightarrow 0x0$$

- Step 3’: reading respectively $\alpha + 8\delta'$ and $0x0$
  - Note that reading refines our view of the memory but does not modify it
Abstract Semantics: Join

- In **MemCAD**, the join operation is in two steps
  - Join of the shape graphs
  - Join of the constraints on the values

First step:
- Sub-memories are joined using the usual join on graphs
- Numerical values are joined using the usual numerical join

Second step:
Abstract Semantics: Join

- In MemCAD, the join operation is in two steps
  - Join of the shape graphs
  - Join of the constraints on the values

First step:

- Sub-memories are joined using the usual join on graphs
- Numerical values are joined using the usual numerical join
Abstract Semantics: Join

In MemCAD, the join operation is in two steps

- Join of the shape graphs
- Join of the constraints on the values

First step:

\[ \begin{align*}
\alpha & \& \text{free\_pool} \\
0 & \delta_4 \\
800 & \delta_4 \\
\end{align*} \]

Second step:

- Sub-memories are joined using the usual join on graphs
- Numerical values are joined using the usual numerical join
Implementation in MemCAD

- Respect of the modularity
  - The main memory is not aware of the nature of its sub-memories
  - A sub-memory is not aware of being “sub”

- Fast analysis (timings in seconds)

<table>
<thead>
<tr>
<th>List Building Programs</th>
<th>Allocation method</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>malloc</td>
</tr>
<tr>
<td>head-insertion</td>
<td>0.019</td>
</tr>
<tr>
<td>tail-insertion</td>
<td>0.027</td>
</tr>
<tr>
<td>tail-insertion then traversal</td>
<td>0.056</td>
</tr>
<tr>
<td>ordered insertion</td>
<td>0.195</td>
</tr>
</tbody>
</table>
Conclusion: Static Analysis

- Need for software verification
  - Cost of verification
  - Cost of the remaining bugs
  - Essential for embedded systems

- Various ways to do so
  - Semi-automatic
    - Proofs
    - Type systems
  - Automatic
    - Model checking
    - Static analysis by abstract interpretation

- Static analysis
  - Effective tool for software verification
  - Sound foundations: strong guarantees
Conclusion: Contributions

- Problem addressed: embedded dynamic data-structures
  - Systematic construction based on simpler abstract domains
  - Efficient tool provided
- On-going work: lists containing arrays
  - Frequent pattern in C programs
  - Inductive definitions requires memory as value
- Other contributions
  - Analyses for quantitative properties
  - Analyses for properties mixing booleans and numbers
Thanks for your attention!