Static Analysis of C Programs

Arnaud Venet
venet@email.arc.nasa.gov

Guillaume Brat
brat@email.arc.nasa.gov

Kestrel Technology
NASA Ames Research Center
Moffett Field, CA 94035

Agenda

• Motivation
• Introduction to Static Analysis
  – Definition
  – Defect classes
  – Applicability issues
  – Specialization
  – Analysis of MPF
• C Global Surveyor
  – Fact sheet
  – CGS phases
  – Example
• Conclusions
**Motivation**

A float overflow causes the crash of Ariane 501.

I shouldn’t have turned off the engine so soon...

A flag badly reset caused Mars Polar Lander to crash on Mars.

**Cost of Losing Missions**

- **Mars Polar Lander**: > $150M
  - Development + Operations: $120M
  - Deep Space 2 probes: $30M
- **Mars Climate Orbiter**: ~$85M
  - Development: $85M
  - Operations: $5M
- **Mars Surveyor 98 (MPL + MCO)** $328M
  - Development: $193M
  - Launch: $92M
  - Operations: $43M
- **Ariane 501**: > $500M
  - Investment over 10 years: $7B
  - Payload value: $500M
Static Analysis

- Static program analysis consists of automatically discovering properties of a program that hold for all possible execution paths of the program.
- Static analysis is not:
  - Testing: manually checking a property for some execution paths.
  - Model checking: automatically checking a property for all execution paths.

Static analysis offers compile-time techniques for predicting conservative, and computable, approximations to the set of values arising dynamically at run-time when executing the program.

C Global Surveyor uses abstract interpretation techniques to extract a conservative system of semantic equations which can be resolved using lattice theory techniques to obtain numerical invariants for each program point.
Is Static Analysis Useful?

- Optimizing compilers
- Program understanding
- Semantic preprocessing:
  - Model checking
  - Automated test generation
- **Program verification**
  - Discovering errors without executing the programs

Program Verification

- Check that every operation of a program will never cause an error (division by zero, buffer overrun, deadlock, etc.)
- **Example:**

```c
int a[1000];
for (i = 0; i < 1000; i++) {
    a[i] = ... ;  // 0 <= i <= 999
}
```

```
buffer overrun → a[i] = ... ;  // i = 1000;
```
Defect Classes

• Static analysis is well-suited for catching runtime errors
  – Array-out-bound accesses
  – Un-initialized variables/pointers
  – Overflow/Underflow
  – Invalid arithmetic operations
• Also for program understanding
  – Data dependences
  – Control dependences
  – Slicing
  – Call graphs

Defect Classes for DS1

• Defect classes for Deep Space One:
  – Concurrency: race conditions, deadlocks
  – Misuse: array out-of-bound, pointer mis-assignments
  – Initialization: no value, incorrect value
  – Assignment: wrong value, type mismatch
  – Computation: wrong equation
  – Undefined Ops: FP errors (tan(90)), arithmetic (division by zero)
  – Omission: case/switch clauses without defaults
  – Scoping Confusion: global/local, static/dynamic
  – Argument Mismatches: missing args, too many args, wrong types, uninitialized args
  – Finiteness: underflow, overflow
Issue 1: Incompleteness

- Discovering a sufficient set of properties (e.g., numerical invariants) for checking every operation of a program is an undecidable problem!
- **False positives:** operations that are safe in reality but which cannot be decided safe or unsafe from the properties inferred by static analysis.

Issue 2: Precision

- **Precision:** number of program operations that can be decided safe or unsafe by an analyzer
  - Precision and computational complexity are strongly related
  - Tradeoff precision/efficiency: limit in the average precision and scalability of a given analyzer
  - Greater precision and scalability is achieved through specialization
Specialization

- Tailoring the analyzer algorithms for a specific class of programs
  - flight control systems
  - digital signal processing, …
- CGS is specialized for the MPF s/w family
- Precision and scalability is guaranteed for this class of programs only
  - However, CGS works for every C program
  - But precision (and scalability) might not be as good for every C program as for MPF-based s/w

Practical Static Analysis

<table>
<thead>
<tr>
<th>Analyzer</th>
<th>Scalability (KLOC)</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coverity</td>
<td>1000</td>
<td>80%</td>
</tr>
<tr>
<td>PolySpace C-Verifier</td>
<td>500</td>
<td>95%</td>
</tr>
<tr>
<td>C Global Surveyor (NASA Ames)</td>
<td>500</td>
<td>95%</td>
</tr>
<tr>
<td>DAEDALUS</td>
<td>50</td>
<td>80%</td>
</tr>
<tr>
<td>GENERAL-PURPOSE ANALYZERS</td>
<td>50</td>
<td>95%</td>
</tr>
<tr>
<td>SPECIALIZED ANALYZERS</td>
<td>50</td>
<td>80%</td>
</tr>
</tbody>
</table>
Analysis of MPF

- Analyzed 3 modules (~20KLoc each) of C code from the MPF flight software with PolySpace
- 80% Precision
  - 80% checks have been classified (correct or incorrect) with certainty
  - 20% warnings: need to be covered by conventional testing
- Found 2 certain errors in 30 minutes
  - But, average run is 12 hours
  - Average time spent manually analyzing RTE is 0.5 hours
- CGS analyzes all 140 KLoc of MPF in 1.5 hours with an 80% precision
  - Some array bounds are not know by CGS because they are passed dynamically in messages

Analysis of DS1

Polyspace:
analyzing 20-40 KLoc modules
took 8-12 hours
with an 80% precision

C Global Surveyor:
analyzing all 280 KLoc of DS1
took 2-3 hours
with a 90% precision
CGS fact sheet

• Static analyzer for finding runtime errors in C programs
  – Out-of-bound array accesses
  – Non-initialized variables
  – De-referencing null pointers
  – Tested on MPF and DS1 flight software systems
• Developed (20 KLoc of C) at NASA Ames in ASE group
  – A. Venet: arnaud@email.arc.nasa.gov
  – G. Brat: brat@email.arc.nasa.gov
• Runs on Linux and Solaris platforms
  – RedHat Linux 2.4
  – SUN Solaris 2.8
• Analysis can be distributed over several CPUs
  – Using PVM distribution system
• Results available using SQL queries
  – To the PostgreSQL database
  – Browser-based graphical interface

Example

```
dbm_ex.c
Main () {
   int i,j;
   volatile k;
   for (i=0; i<8; i++) {
      for (j=0; j<l; j++) {
         k++;
      }
   }
   return;
}
```
Setting up Analysis

• Creating a database
  – initdb cgsDB

• Starting the database in a separate shell
  – postmaster -i -D cgsDB

• Starting the PVM distribution system
  – pvm conf
  – Where conf lists all available machines

• Go to source directory: say src/

• Creating the intermediate form
  – cgsfe dbm_ex.c
  – The file dbm_ex.cil is created in src/CGS/

Initialization

• First, CGS reads the CIL files and prepare for the analysis
  – cgs init CGS/dbm_ex.cil

• In the database, one can see file and function tables:
  – psql src
  – select * from file_table;
  – select * from function_table;
Building Equations

• The second step of CGS consists of building the semantic equations abstracting the behavior of the program:
  – cgs build <options>
• This creates a table of equations in the database
  – Local numerical invariants available in DB
    – select * from num_inv_table where function=<name>;

Bootstraping

• This phase builds an abstract graph of the memory usage in the C program
  – cgs bootstrap <option>
• In the database the following information is now available:
  – Call graph
  – Memory graph, e.g., which global pointers points to what memory cell
Solving the Equations

• The next step is to solve the equations using the pointer analysis done in the previous phase
  - cgs solve <options>

• The following information is now available in the database:
  – Pointer table
  – All numerical invariants for all program points

ABC Analysis

• The only currently available analysis is the one checking the out-of-bound array accesses
  - cgs abc

• Results are available in the database
  - select * from abc_result_table;
  – Results are coded:
    • G for green: the access is correct
    • R for red: the access is incorrect
    • O for orange: the access may be incorrect
    • U for unreachable: dead code
Analysis Script for MPF

- cgs init CGS/*.cil (62s with eight 2.2MHz CPUs)
- cgs build –l –e –m Heap_alloc:2 –m IpcQ_Create:? –m BuggerMgr_alloc:? –s int-in-mem (527s)
- cgs bootstrap –c –k 3 –s taskSpawn:5 (445s)
- cgs solve –c –f –n (892s)
- cgs solve –c –b (471s)
- cgs solve –c –f –n (857s)
- cgs abc (510s) => roughly 1 hour for 60% precision
- cgs solve –c –b (526s)
- cgs solve –c –f –n (848s)
- cgs abc (503s) => roughly ½ hour for 80% precision

Conclusions

- Static analysis tools can be used to verify the absence of runtime errors in NASA code
  - No need for input test cases
  - Complete coverage of all data accesses (pointer aliasing) and execution paths
- Static analysis works well for errors such as
  - Out-of-bound array accesses
  - Un-initialized variables
  - De-references of null pointers
  - Some invalid arithmetic operations
- We have built a scalable, yet precise, static analyzer for C programs
  - Tested on MPF (140KLoc) and DS1 (280 KLoc)
  - Next test: MER (650 KLoc) and other NASA mission code
  - Available on Linux and Solaris platforms
- We plan on developing a static analyzer for MDS code
  - Will work for a simplified version of C++
  - Tentative availability date: 2005