Software Model Checking of Floating-Point Programs

Malay Ganai,
Franjo Ivančić,
Aarti Gupta, Sriram Sankaranarayanan (UC Boulder)

NEC Labs America
R&D Overview in NEC Verification Group

Layered Infrastructure
- Symbolic constraint solvers
  - perform the search for bugs or proofs
- Verification methods
  - algorithms to formulate the search problem
- Modeling techniques
  - for deriving “effective” verification models from user designs or programs
- Application domains
  - Software program verification (F-Soft)
  - Concurrent system verification (ConSave)
  - Embedded/Hybrid Systems (TESSA)

Opportunities for exciting internships
- http://www.nec-labs.com/~fsoft
- http://www.nec-labs.com
CPS Design Flow & Verification

Mathematical Analysis

Stability, Robustness

Functional Correctness

Model-Based Flow is practiced
- Model-Based Design (MBD)
- Model-Driven Development (MDD)

Higher levels of abstraction help to design systems of larger scale & complexity

Mathematical Model
\[ x' = f(x, t) \]

Control Design

Closed Loop System Model (Simulink/Stateflow)

Code Gen

Platform Constraints

NEC Labs target

NEC Labs target

C code

Simulation

Test Coverage Program Analysis

Hardware Platform

Integration

NEC Labs target

CPS Needs

Unified design flow for control & cyber, Distributed sensing & control, Timing coordination, Uncertainties in data & environment, Human-computer interaction, Safety, reliability, security, privacy

Boeing 747 ~ 50 ECUs
~ 4 MLOC
Boeing 787 ~ 6 MLOC

ICE Train ~ 0.5MLOC
System-wide ~ 5MLOC

BMW [IEEE09] ~ 70-100 ECUs
~ 100 MLOC
**MBD and Verification**

### Model-Based Design (MBD)

- **Model** (UML, SysML, AADL, Simulink, …)
  - Model-based testing, Property verification
  - Property verification, Equivalence checking

- **Intermediate Form** (Lustre, …)
  - Runtime monitoring

### Model-level Verification

- **Discrete models**: Model checking works well (abstraction used to discretize continuous behavior), e.g. [SRI tool chain].
- **Hybrid (discrete+continuous) models**: Model checking does not scale well.
- **Architecture-level models** (SysML, AADL): Model checking can scale.

### Model-level Testing

- **Model-based Testing**: Main issue is coverage.
- **Concolic execution**: Mix of concrete and symbolic execution, helps to cover hard-to-reach regions.

### Source code Analysis & Verification

- **Numerical Precision Analysis**: Requires accurate modeling of floating point computations in source code, and efficient solvers for handling non-linear arithmetic.
Two Infamous Rounding Related Bugs

Ariane 5 (Flight 501)

Conversion of 64-bit floating-point number to unsigned 16-bit integer failed due to overflow. Finally, self-destruction was initiated. Cost about $400M in lost payload.

Patriot Missile Failure

Estimate of future position of incoming missile was inaccurate due to rounding error accumulation of fixed-point representation. Incoming Scud missile killed 28 soldiers.

May 24, 2012
Floating-Point Source Code Analysis: Motivating Example

```
float compute (floats b1,b2,a1,a2) {
    float a1b2 = a1*b2 ;
    float a2b1 = a2*b1 ;
    float denom = a2b1-a1b2 ;
    float res = b1/denom ;
    return res ;
}

int main() {
    float f = compute (-46099201,
                        -35738642,
                        37639840,
                        29180479 ) ;
    return f>0.0 ;
}
```

Subtraction leads to cancelation effects. The range of potential values for `denom` includes negative and positive numbers resulting in instability.

Since the computation of `denom` may result in sub-normal range, significant precision loss occurs.

The bounded model checking analysis in F-Soft uses NECLA's CORDIC solver to find the problem in about 30 seconds.

- Computation on Linux server, gcc, default rounding mode: f=0.343466...
- Newer quad-core machine using RedHat, gcc, default rounding: f=0.5
- E. Goubault reported on same computation on UltraSPARC: f=104,679,994
- Mathematically speaking correct (expected) result: f=-46,099,201
  - denom is mathematically speaking 1
Goals

- Utilize model checking to find numerically unstable computation paths
- Related work
  - Discovering potential instabilities in programs using abstract interpretation
    - Astree, Fluctuat
  - Floating-point analysis of hardware implementations
  - Software model checking
    - CBMC only (?) other tool that models floating-point computations
    - Focus is on exact bitwise computation (given particular rounding mode)

- Pros & Cons of using Model Checking
  - Completely path sensitive analysis
  - Counterexamples to aid user debugging
  - Utilize recent advances in SMT-solvers (SMT: Satisfiability Modulo Theory)
  - Scalability
    - Often, small program slices deal with numerical computation
  - Seriously? What about non-linear operations in SMT?
    - Good point. They are coming though – we at NEC have two different solvers now
Float Modeling using Intervals over Reals

int foo (float f, int x) {
    int ftmp = (int) f;
    float xtmp = (float) x;
    if (ftmp > 0)
        return x;
    if (f > xtmp)
        return ftmp;
    return -x;
}

enum floatStatus {
    NaN, Inf, -Inf, Reg
};
struct floatModel {
    real lowerBnd;
    real upperBnd;
    floatStatus lowS;
    floatStatus highS;
};
//some pre-defined
//functions, like float2Int

int foo (floatModel f, int x) {
    int ftmp = float2Int(f);
    floatModel xtmp = int2float(x);
    if (ftmp > 0)
        return x;
    if (floatCmpGT(f, xtmp))
        return ftmp;
    return -x;
}

- Note: After lowering, operations (such as +) use semantics of reals
  - The value of lowerBnd (upperBnd) is read only when status is not NaN, ±Inf

- Fully lowered expressions for bound variable types are large
  - Example: r=x*y
  - The actual bound variables updates are less complicated
    (basically, semantics of reals)

<table>
<thead>
<tr>
<th>x*y</th>
<th>NaN</th>
<th>Inf</th>
<th>&gt;0</th>
<th>0</th>
<th>&lt;0</th>
<th>-Inf</th>
</tr>
</thead>
<tbody>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>Inf</td>
<td>NaN</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>-Inf</td>
<td>-Inf</td>
</tr>
<tr>
<td>&gt;0</td>
<td>NaN</td>
<td>Inf</td>
<td>&gt;0/Inf</td>
<td>0</td>
<td>&lt;0/-Inf</td>
<td>-Inf</td>
</tr>
<tr>
<td>0</td>
<td>NaN</td>
<td>NaN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td>&lt;0</td>
<td>NaN</td>
<td>-Inf</td>
<td>&lt;0/-Inf</td>
<td>0</td>
<td>&gt;0/Inf</td>
<td>Inf</td>
</tr>
<tr>
<td>-Inf</td>
<td>NaN</td>
<td>-Inf</td>
<td>-Inf</td>
<td>NaN</td>
<td>Inf</td>
<td>Inf</td>
</tr>
</tbody>
</table>
Floating-Point Specific Checkers

- After instrumentation, interesting properties are easy to check
  - Computations should not result in Nan, Inf, -Inf
  - Operations should not be unspecified

- Type 1: Computations remain in floating-point number range
  - After every floating-point operation, add a check that lower bound type and upper bound type are Regular
  - Remove checks based on abstract interpretation

- Type 2: Unspecified situations
  - Check for casting from floating-point to integer-based type with too small bitwidth
  - Again, checks can be removed using abstract interpretation
SMT-based Bounded Model Checking

- CFG annotated with reals and integers is then analyzed using bounded model checker based on SMT solvers
  - SMT: Satisfiability modulo theories (such as Yices, Z3, CVC, OpenSMT, …)
- Problems generated from programs often involve non-linear operations
  - Current solvers are efficient for linear arithmetic
  - Available non-linear solver: HySAT (Uni Oldenburg)
  - We developed non-linear solver based on CORDIC linearization
  - See FMCAD’09(CORDIC) and FMCAD’10(ICP+SMT) for details of our solvers

```
Linear Arithmetic Constraints

SMT problem with Linear + Non-Linear operations on reals
```

```
CORD

SMT (LRA) Solver
```

```
Translate Non-linear Operations using CORDIC
```

```
DISE
```
Comparison of CORD with HySAT: Non-Linear Operations

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CORD</th>
<th>HySAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>SAT?</td>
<td>ld δ</td>
</tr>
<tr>
<td>e1</td>
<td>no</td>
<td>-10</td>
</tr>
<tr>
<td>e2</td>
<td>yes</td>
<td>-13</td>
</tr>
<tr>
<td>e3</td>
<td>no</td>
<td>-15</td>
</tr>
<tr>
<td>e4</td>
<td>yes</td>
<td>-20</td>
</tr>
<tr>
<td>s1</td>
<td>no</td>
<td>-18</td>
</tr>
<tr>
<td>s2</td>
<td>no</td>
<td>-18</td>
</tr>
<tr>
<td>s3</td>
<td>yes</td>
<td>-18</td>
</tr>
<tr>
<td>s4</td>
<td>yes</td>
<td>-18</td>
</tr>
</tbody>
</table>

Small handcrafted benchmarks

Properties related to numerical stability of float computations
Conclusions

- Software model checking to find numerical instabilities in programs
- Utilizes (non-linear) SMT-based Bounded Model Checking
- Model of programs is based on interval arithmetic
- More experiments needed on larger programs
  - Heuristics for choice of rounding precision
  - Experiment with affine arithmetic instead of interval arithmetic
  - Propose combination of abstraction-refinement techniques to scale better
F-Soft Platform Prototype

Properties

Source code (C, stubs)

Automated checkers

Static Analysis

Bug report (HTML, gdb, XML)

Testbench Generator

Abstraction

Ctrex Analysis & Refinement

Model Transformation, Translation

Model Checker (VeriSol)

Proof

Program slicing

Constant Folding

Semantic Slicing

Limitation Analysis

Loop Optimizations

Invariant Generation

Predicate Abstraction

VeriSol
Floating-Point Analysis Experiment: Effect of Varying Rounding Precision

<table>
<thead>
<tr>
<th>$\log_2 \delta$</th>
<th>n=5(d=68)</th>
<th>n=10(d=118)</th>
<th>n=15(d=168)</th>
<th>n=20(d=218)</th>
<th>n=25(d=268)</th>
<th>n=30(d=318)</th>
<th>n=35(d=368)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-22</td>
<td>2.2s</td>
<td>23.3s</td>
<td>440.8s</td>
<td>TO</td>
<td>60.5s</td>
<td>279.7s</td>
<td>TO</td>
</tr>
<tr>
<td>-20</td>
<td>10.1s</td>
<td>15.1s</td>
<td>36.8s</td>
<td>TO</td>
<td>93.5s</td>
<td>571.8s</td>
<td>TO</td>
</tr>
<tr>
<td>-18</td>
<td>2.3s</td>
<td>15.6s</td>
<td>TO</td>
<td>123.5s</td>
<td>185.3s</td>
<td>197.3s</td>
<td>TO</td>
</tr>
<tr>
<td>-16</td>
<td>2.7s</td>
<td>5.1s</td>
<td>8.2s</td>
<td>162.9s</td>
<td>32.9s</td>
<td>149.1s</td>
<td>TO</td>
</tr>
<tr>
<td>-14</td>
<td>2.5s</td>
<td>11.6s</td>
<td>35.0s</td>
<td>34.9s</td>
<td>577.2s</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>-12</td>
<td>1.1s</td>
<td>7.8s</td>
<td>58.0s</td>
<td>16.4s</td>
<td>91.3s</td>
<td>175.4s</td>
<td>TO</td>
</tr>
<tr>
<td>-10</td>
<td>1.2s</td>
<td>6.8s</td>
<td>34.3s</td>
<td>24.3s</td>
<td>162.8s</td>
<td>245.5s</td>
<td>TO</td>
</tr>
<tr>
<td>-9</td>
<td>1.3s</td>
<td>36.5s</td>
<td>34.5s</td>
<td>61.7s</td>
<td>TO</td>
<td>SP(273.4s)</td>
<td>SP(457.8s)</td>
</tr>
<tr>
<td>-8</td>
<td>0.8s</td>
<td>SP(14.1s)</td>
<td>SP(52.6s)</td>
<td>SP(56.8s)</td>
<td>SP(120.5s)</td>
<td>SP(226.3s)</td>
<td>TO</td>
</tr>
<tr>
<td>-6</td>
<td>SP(0.9s)</td>
<td>SP(2.8s)</td>
<td>SP(8.9s)</td>
<td>SP(43.6s)</td>
<td>SP(97.9s)</td>
<td>SP(231.7s)</td>
<td>TO</td>
</tr>
<tr>
<td>-4</td>
<td>SP(0.8s)</td>
<td>SP(3.4s)</td>
<td>SP(32.2s)</td>
<td>SP(73.5s)</td>
<td>SP(79.5s)</td>
<td>SP(229.6s)</td>
<td>SP(496.5s)</td>
</tr>
<tr>
<td>-2</td>
<td>SP(0.9s)</td>
<td>SP(2.4s)</td>
<td>SP(34.5s)</td>
<td>SP(51.5s)</td>
<td>SP(78.0s)</td>
<td>SP(44.3s)</td>
<td>SP(435.4s)</td>
</tr>
</tbody>
</table>

TO: Time-out  
SP: Spurious counterexample  
n: Length of input array  
d: depth of counterexample  
$\delta$: Rounding precision

- Potential for CEGAR-based loop to choose rounding precision ($\delta$)  
- Can be combined with other abstraction-refinement techniques such as predicates

- However: non-monotonic model checking performance  
  - Other CEGAR-based loops often have expected hardness increases with more precise models (more predicates means more state variables)  
  - Here: Performance here more related to how valid or invalid the query is  
  - That is, how precise to I have to compute in the real domain to find the answer
Validated Arithmetic: Interval Arithmetic

- IA is an arithmetic for validated computations
  - introduced by Moore several decades ago

- A computes a range in machine arithmetic that contains the result of the operation in real arithmetic

- An IA quantity \( x \) is a range \([x_l, x_u]\)

- Addition:
  \[ x + y = [x_l, x_u] + [y_l, y_u] = [x_l + y_l, x_u + y_u] \]

- Multiplication:
  \[ xy = [x_l, x_u][y_l, y_u] = [\min\{x_l y_l, x_l y_u, x_u y_l, x_u y_u\}, \max\{x_l y_l, x_l y_u, x_u y_l, x_u y_u\}] \]

- We can capture floating-point rounding errors by rounding down for the lower bound and rounding up for the upper bound
F-Soft’s Symbolic Models from CFG

- Our target for model checking: Bounded verification model
  - Recursive data structures are bounded up to some user-chosen depth
  - Recursive functions are modeled using bounded unwinding
    - Alternative: Boolean programs [Ball & Rajamani 01]
  - Functions are not inlined; calling context information is explicitly preserved

- This yields a CFG with only int type data variables
  - Program Counter (PC) variables are introduced to represent CFG nodes
  - Each data variable is interpreted as:
    - a vector of state-bits (latches) for bit-precise SAT- or SMT-based model checking
    - an infinite integer for numerical domain analysis and polyhedra-based model checking
  - Floating-point operations are modeled using reals and integers

- Memory layout is not related to physical memory view
  - Compiler-independent view of memory layout
  - Allows scalable analysis of pointer arithmetic without strides
  - Does not track compiler-dependent byte-level arithmetic (i.e. moving between structure elements through address offsets)
Handling of Denormalized Numbers

- Denormalized numbers are used to provide *gradual underflow*
  - These are numbers between the smallest positive *normal number* and zero (and their negative versions)
  - A *normal number* is defined using the regular floating-point format
  - Calculations can lose precision slower than without these numbers

- Modeling denormalized numbers
  - Relative error is large (½ to 1), but absolute error is small
    • Largest denormalized number in float: about $1.2 \times 10^{-38}$
    • Largest denormalized number in double: about $2.4 \times 10^{-308}$
  - Use absolute error to model denormalized numbers
  - Use denormalized absolute offset in addition to relative offset in all operations
    • Sound solution, since only one offset is actually required
    • For normal numbers, the denormalized offset is generally too small to matter
    • May be too imprecise in denormalized number range
  - Other solution would be more precise, but more expensive as well
    • Pre-compute whether an operation results in normal or denormalized range

Model for addition $x=y+z$:

- \[ x_{LB} = (y_{LB}+z_{LB}) \times (1 \pm \delta) - \varepsilon \]
- \[ x_{UB} = (y_{UB}+z_{UB}) \times (1 \pm \delta) + \varepsilon \]
Modeling Example: \( x = y + z \)

- \( x = \text{NONDET}_\text{REAL} \);
- \( x_{LB} = (y_{LB} + z_{LB}) \times (1 \pm \delta) \);
- \( x_{UB} = (y_{UB} + z_{UB}) \times (1 \pm \delta) \);
- \( x_{LBT} = (y_{LBT} == \text{NaN} \text{ or } z_{LBT} == \text{NaN}) \text{ ? } \text{NaN :} (y_{LBT} == \text{Inf} \text{ and } z_{LBT} == -\text{Inf}) \text{ ? } \text{NaN :} (y_{LBT} == -\text{Inf} \text{ and } z_{LBT} == \text{Inf}) \text{ ? } \text{NaN :} (y_{LBT} == \text{Inf} \text{ or } z_{LBT} == \text{Inf}) \text{ ? } \text{Inf :} (y_{LBT} == -\text{Inf} \text{ or } z_{LBT} == -\text{Inf}) \text{ ? } -\text{Inf :} (x_{LB} < \lambda) \text{ ? } -\text{Inf :} (x_{LB} > \mu) \text{ ? } \text{Inf : } \text{Regular} \)

(similar for \( x_{UBT} \))

- Constraints:
  - \( x_{LBT} \neq \text{Regular or } x_{LB} \leq x \)
  - \( x_{UBT} \neq \text{Regular or } x \leq x_{UB} \)

\( \delta \): Floating-point-type specific precision (float, double, long double)

\( \text{NaN, Inf, -Inf, Regular are symbols/constants. Constant propagation will be useful for simplification.} \)

\( \lambda \): Smallest float of the current type

\( \mu \): Largest float of the current type

Note that assignments to simple type-variables are only complicated expressions.
IEEE-754 Floating-Point Standard

Floating-point number formats
- Single precision ("float", 32-bit) vs double precision ("double", 64-bits)
- Special values: $\infty$, $-\infty$, not-a-number (NaN)
- Normal numbers vs sub-normal number range (very close to zero)

Five rounding modes
- Towards $\infty$, towards $-\infty$, towards 0, towards nearest with 2 bias strategies
- First compute precise solution, then round
- Thus, results in normal number range have maximal relative rounding error

Sub-normal numbers
- Provide gradual underflow for higher precision around 0
- Two near-by normal floating-point numbers have always non-zero difference
- Relative error can be large

Operations
- Standard defines many details about expected results for computations
Overview of Main NEC Business Domains and Services

Quality is the key strength of NEC products.

- **IT Services**
  - Cloud Oriented Service Platform Solutions
  - MegaOak
  - Grid
  - PanelDirector

- **Carrier Network**
  - Long Term Evolution Network Systems
  - Unity Cable Systems
  - WiMAX Network Systems
  - Compact Microwave Communications Systems

- **Social Infrastructure**
  - Digital Terrestrial TV Transmitters
  - Asteroid Explorer "HAYABUSA" (provided by Japan Aerospace Exploration Agency)

- **Platform**
  - Super Computer
  - Server
  - Integrated Operation/Management Middleware

- **Personal Solutions**
  - Personal Computers
  - Mobile Terminals

- **Others**
  - Electron Devices
  - Lithium-ion Batteries
  - Liquid Crystal Displays
NEC Global R&D

Promoting research collaboration between sites by utilizing regional characteristics (markets and technologies).

Europe

NEC Laboratories China
(Beijing, China)
- Service Platform
- Networking
- Sensor/Media processing

NEC Laboratories Europe
(Heidelberg, Germany / Acton, U.K.)
- Network Management & Control
- Wireless Access
- Distributed Service/Security
- Standardization

China

North America

Japan

NEC Laboratories America
(Princeton, Silicon Valley, U.S.A.)
- Information Analysis, Data Management
- Large Scale Distributed System
- Parallel Architecture
- System Analysis and Verification
- Broadband and Mobile Networking
- Machine Learning
- Quantum IT

Tsukuba, Ibaraki
- Energy Device
- Sensing Device
- Functional material
- Nanotechnology and Quantum IT

Ikoma, Nara
- Ubiquitous Computing
- Software for the Internet

Sagamihara, Kanagawa
- Energy Device
- Low Power Device and System
- Jisso Design/Analysis

Tamagawa, Kanagawa
- Service Platform
- Security
- Computer Architecture
- Network System
- Storage System
- Recognition/Automatic Translation
- SoC Design
- RF Circuit Design
- Mobile terminals Packaging
- Production Technologies for New Products
NECLA At a Glance

Researchers are drawn from global talent pool
Research strategy based on core competencies and inter-disciplinary synergies
Focus on creating innovative core technology that provides differentiation for products/services
Collaboration with major research universities, including CMU, Columbia, Georgia Tech, MIT, Princeton, Rutgers, Stanford, U. of California, U. of Pennsylvania, and with leading companies
Strong track record of tech transfer & commercialization
Successful exploitation of US-based location