Verification of Software Upgrades

Natasha Sharygina

*FORMAL VERIFICATION LAB
University of Lugano
Motivation

• Software evolves
  • Small frequent upgrades
  • Complete re-verification impractical / infeasible

• Incremental verification
  • Store information from previous verification runs
  • Speed-up consecutive runs

• Local upgrade checks
  • Incremental Bounded model checking
  • Interpolation-based function summarization
Context

*bounded model checking*

- Loops/recursion unwound
  - Up to a given bound
- Encoding into a BMC formula
- Satisfiability check by a solver
  - UNSAT $\rightarrow$ System is safe
  - SAT $\rightarrow$ Error found
    - Satisfying assignment identifies an error trace
Partitioning BMC

formula construction

\[ \varphi_1 \land \varphi_2 \land \varphi_3 \land \varphi_{\text{main}} \land \varphi_4 \land \varphi_5 \land \varphi_6 \land \text{error}_{\text{main}} \]

\[ \text{UNSAT} \]
Partitioning BMC
generation of summaries

\[ \varphi_1 \land \varphi_2 \land \varphi_3 \land \varphi_{\text{main}} \land \varphi_4 \land \varphi_5 \land \varphi_6 \land \text{error}_{\text{main}} \]

\[ \text{main} \]

\[ f_1 \land f_2 \land f_3 \land f_4 \land f_5 \land f_6 \]

\[ \varphi_1 \quad \varphi_2 \quad \varphi_3 \quad \varphi_4 \quad \varphi_5 \quad \varphi_6 \]

\[ A \quad B \]

\[ \text{UNSAT} \]
Partitioning BMC

properties of interpolant-based summaries

(ii) $I \land B$ is unsat

\[
\varphi_1 \land \varphi_2 \land \varphi_3 \land \varphi_{\text{main}} \land I_5 \land \text{error}_{\text{main}}
\]

(i) $A \rightarrow I$

(iii) shared symbols

\[
\varphi_1 \land \varphi_2 \land \varphi_3 \land \varphi_{\text{main}} \land \varphi_4 \land \varphi_5 \land \varphi_6 \land \text{error}_{\text{main}}
\]
Interpolation-based function summaries

• Function summary
  • An over-approximation of the real behavior
    • Considering the given bound
  • Contains only the relevant information
    • Generated from the proof of UNSAT
  • Expressed using function’s in/out parameters
Interpolation-based function summaries

• Function summary
  • An over-approximation of the real behavior
    • Considering the given bound
  • Contains only the relevant information
    • Generated from the proof of UNSAT
  • Expressed using function’s in/out parameters

• Usage
  1) Same code, different properties
    • To approximate the corresponding functions
  2) Same properties, different code
    • Upgrade checking

Sery O., Fedyukovich G., Sharygina N., Interpolation-based Function Summaries in Bounded Model Checking, HVC 2011; FunFrog tool, TACAS 2012
Interpolation-based function summaries

• Function summary
  • An over-approximation of the real behavior
    • Considering the given bound
  • Contains only the relevant information
    • Generated from the proof of UNSAT
  • Expressed using function’s in/out parameters

• Usage
  1) Same code, different properties
    • To approximate the corresponding functions
  2) Same properties, different code
    • Upgrade checking

Sery O., Fedyukovich G., Sharygina N., Upgrade Checking, FMCAD 2012
Upgrades – key idea

Observations:

• An old summary can remain a valid over-approximation of the new version of a modified function

• Old summaries are precise enough to prove the properties of interest

Idea: Do a cheap local check…
The eVolCheck algorithm overview

0) Verification of the base version of the software (*bootstrap*)
   • function summaries generated and stored
The eVolCheck algorithm overview

0) Verification of the base version of the software (*bootstrap*)
   • function summaries generated and stored

1) The user upgrades the software
2) Upgraded version of the software is preprocessed
3) eVolCheck identifies the modified code
   • by comparing parse trees for both the base and the upgraded version
4) eVolCheck attempts to verify the upgraded version
   • using cheap local checks based on the function summaries of the last version
5a) If successful, eVolCheck updates function summaries for next checks
5b) If unsuccessful, eVolCheck reports violation + an error trace
6) The user fixes the reported errors and continues from step 2)
The algorithm

overview

0) Verification of the base version of the software (*bootstrap*)
   • function summaries generated and stored

1) The user upgrades the software

2) Upgraded version of the software is parsed by goto-cc

3) eVolCheck identifies the modified code
   • by comparing parse trees for both the base and the upgraded version

4) **eVolCheck attempts to verify the upgraded version**
   • using cheap local checks based on the function summaries of the last version

5a) If successful, eVolCheck updates function summaries for next checks

5b) If unsuccessful, eVolCheck reports violation + an error trace

6) The user fixes the reported errors and continues from step 2)
Incremental upgrade check

Functions $f_1$ and $f_4$ upgraded...
Incremental upgrade check

Functions $f_1$ and $f_4$ upgraded...
Incremental upgrade check

We attempt to verify that summaries $I_1$ and $I_4$ are still valid over-approximations.
Incremental upgrade check

Check for $I_1$:

$\varphi'_1 \rightarrow I_1 \quad \checkmark$ upgrade is safe
Incremental upgrade check

Check for $I_1$:

$\varphi'_1 \rightarrow I_1$  ✔️ upgrade is safe

Check for $I_4$:

$\varphi'_4 \rightarrow I_4$  ✗ propagate upwards
Incremental upgrade check

Check for \( I_1 \):

\[ \varphi'_1 \rightarrow I_1 \quad \checkmark \text{ upgrade is safe} \]

Check for \( I_4 \):

\[ \varphi'_4 \rightarrow I_4 \quad \times \text{ propagate upwards} \]

\[ \varphi'_4 \land \varphi_5 \land I_6 \rightarrow I_5 \quad \times \text{ refine downwards} \]

\[ \varphi'_4 \land \varphi_5 \land \varphi_6 \rightarrow I_5 \quad \checkmark \text{ upgrade is safe} \]
Incremental upgrade check

Check for $I_1$:

$\varphi'_1 \rightarrow I_1$ ✓ upgrade is safe

Check for $I_4$:

$\varphi'_4 \rightarrow I_4$ X propagate upwards

$\varphi'_4 \land \varphi_5 \land I_6 \rightarrow I_5$ X refine downwards

✓ upgrade is safe

Note that $l_5$ is checked as UNSAT of:

$\varphi'_4 \land \varphi_5 \land \varphi_6 \land \neg l_5$

... we can regenerate summaries
Incremental upgrade check

Check for \( I_1 \):
\[
\varphi'_1 \rightarrow I_1 \quad \checkmark \text{ upgrade is safe}
\]

Check for \( I_4 \):
\[
\varphi'_4 \rightarrow I_4 \quad \times \text{ propagate upwards}
\]
\[
\varphi'_4 \land \varphi_5 \land I_6 \rightarrow I_5 \quad \times \text{ refine downwards}
\]
\[
\varphi'_4 \land \varphi_5 \land \varphi_6 \rightarrow I_5 \quad \checkmark \text{ upgrade is safe}
\]

Note that \( I_5 \) is checked as UNSAT of:
\[
\varphi'_4 \land \varphi_5 \land \varphi_6 \land \neg I_5
\]

... we can regenerate summaries
eVolCheck
architecture
### eVolCheck experimentation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Benchmark Name</th>
<th>Bootstrap Total [s]</th>
<th>Bootstrap Itp [s]</th>
<th>Upgrade check Total [s]</th>
<th>Upgrade check Diff [s]</th>
<th>Upgrade check Itp [s]</th>
<th>Speedup</th>
<th>Result</th>
<th>ISR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABB_A</td>
<td>8.644</td>
<td>0.008</td>
<td>0.04</td>
<td>0.009</td>
<td>0.003</td>
<td>220x</td>
<td>SAFE</td>
<td>0/7</td>
<td></td>
</tr>
<tr>
<td>ABB_B</td>
<td>6.236</td>
<td>0.009</td>
<td>0.006</td>
<td>0.006</td>
<td>—</td>
<td>935x</td>
<td>SAFE</td>
<td>0/9</td>
<td></td>
</tr>
<tr>
<td>ABB_C</td>
<td>8.532</td>
<td>0.015</td>
<td>0.059</td>
<td>0.008</td>
<td>0.003</td>
<td>157x</td>
<td>SAFE</td>
<td>0/8</td>
<td></td>
</tr>
<tr>
<td>VTT_A</td>
<td>0.512</td>
<td>0.001</td>
<td>0.006</td>
<td>0.006</td>
<td>—</td>
<td>85.5x</td>
<td>SAFE</td>
<td>0/9</td>
<td></td>
</tr>
<tr>
<td>VTT_B</td>
<td>0.514</td>
<td>0.001</td>
<td>0.031</td>
<td>0.006</td>
<td>—</td>
<td>0.7x</td>
<td>BUG</td>
<td>1/9</td>
<td></td>
</tr>
<tr>
<td>euler_A</td>
<td>12.56</td>
<td>0.099</td>
<td>0.179</td>
<td>0.001</td>
<td>0.016</td>
<td>70.4x</td>
<td>SAFE</td>
<td>1/6</td>
<td></td>
</tr>
<tr>
<td>euler_B</td>
<td>12.547</td>
<td>0.095</td>
<td>2.622</td>
<td>0.001</td>
<td>0.031</td>
<td>4.74x</td>
<td>SAFE</td>
<td>3/5</td>
<td></td>
</tr>
<tr>
<td>life_A</td>
<td>13.911</td>
<td>1.366</td>
<td>0.181</td>
<td>0.001</td>
<td>&lt;0.001</td>
<td>77.0x</td>
<td>SAFE</td>
<td>0/5</td>
<td></td>
</tr>
<tr>
<td>life_B</td>
<td>13.891</td>
<td>1.357</td>
<td>6.774</td>
<td>0.001</td>
<td>—</td>
<td>0.31x</td>
<td>BUG</td>
<td>5/5</td>
<td></td>
</tr>
<tr>
<td>arithm_A</td>
<td>0.147</td>
<td>0.007</td>
<td>0.355</td>
<td>0.001</td>
<td>—</td>
<td>0.39x</td>
<td>BUG</td>
<td>3/3</td>
<td></td>
</tr>
<tr>
<td>diskperf_A</td>
<td>0.167</td>
<td>0.001</td>
<td>0.024</td>
<td>0.008</td>
<td>&lt;0.001</td>
<td>5.79x</td>
<td>SAFE</td>
<td>0/21</td>
<td></td>
</tr>
<tr>
<td>diskperf_B</td>
<td>0.137</td>
<td>0.001</td>
<td>0.062</td>
<td>0.009</td>
<td>—</td>
<td>2.25x</td>
<td>BUG</td>
<td>3/21</td>
<td></td>
</tr>
<tr>
<td>floppy_A</td>
<td>2.146</td>
<td>0.229</td>
<td>0.422</td>
<td>0.022</td>
<td>&lt;0.001</td>
<td>5.02x</td>
<td>SAFE</td>
<td>0/226</td>
<td></td>
</tr>
<tr>
<td>floppy_B</td>
<td>2.183</td>
<td>0.237</td>
<td>2.277</td>
<td>0.206</td>
<td>—</td>
<td>0.82x</td>
<td>BUG</td>
<td>79/226</td>
<td></td>
</tr>
<tr>
<td>kbfiltre_A</td>
<td>0.288</td>
<td>0.011</td>
<td>0.081</td>
<td>0.023</td>
<td>0.001</td>
<td>3.40x</td>
<td>SAFE</td>
<td>1/63</td>
<td></td>
</tr>
<tr>
<td>kbfiltre_B</td>
<td>0.320</td>
<td>0.009</td>
<td>0.088</td>
<td>0.023</td>
<td>0.001</td>
<td>1.85x</td>
<td>SAFE</td>
<td>3/63</td>
<td></td>
</tr>
</tbody>
</table>

Linux, x64, Intel i-7, 3.4GHz, 16GB
### Table: Benchmarks Results

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bootstrap</th>
<th>Upgrade check</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total [s]</td>
<td>Itp [s]</td>
</tr>
<tr>
<td>ABB_A</td>
<td>8.644</td>
<td>0.008</td>
</tr>
<tr>
<td>ABB_B</td>
<td>6.236</td>
<td>0.009</td>
</tr>
<tr>
<td>ABB_C</td>
<td>8.532</td>
<td>0.015</td>
</tr>
<tr>
<td>VTT_A</td>
<td>0.512</td>
<td>0.001</td>
</tr>
<tr>
<td>VTT_B</td>
<td>0.514</td>
<td>0.001</td>
</tr>
<tr>
<td>euler_A</td>
<td>12.56</td>
<td>0.099</td>
</tr>
<tr>
<td>euler_B</td>
<td>12.547</td>
<td>0.095</td>
</tr>
<tr>
<td>life_A</td>
<td>13.911</td>
<td>1.366</td>
</tr>
<tr>
<td>life_B</td>
<td>13.891</td>
<td>1.357</td>
</tr>
<tr>
<td>arithm_A</td>
<td>0.147</td>
<td>0.007</td>
</tr>
<tr>
<td>diskperf_A</td>
<td>0.167</td>
<td>0.001</td>
</tr>
<tr>
<td>diskperf_B</td>
<td>0.137</td>
<td>0.001</td>
</tr>
<tr>
<td>floppy_A</td>
<td>2.146</td>
<td>0.229</td>
</tr>
<tr>
<td>floppy_B</td>
<td>2.183</td>
<td>0.237</td>
</tr>
<tr>
<td>kbfilt A</td>
<td>0.288</td>
<td>0.011</td>
</tr>
<tr>
<td>kbfilt B</td>
<td>0.320</td>
<td>0.009</td>
</tr>
</tbody>
</table>

---

**Linux, x64, Intel i-7, 3.4GHz, 16GB**
if(delta0 != 0 && delta1 == 0 && delta2 == 0) vMax = 3 * vMax;
if(delta0 == 0 && delta1 != 0 && delta2 == 0) vMax = 3 * vMax;
if(delta0 == 0 && delta1 == 0 && delta2 != 0) vMax = 3 * vMax;

int s = 0;

for(i = 0; i < NUM_JOINTS; i++)
{
    currentPosition[i] = in[i];

    InitProfile(i, fb[i], // start
               in[i], // end
               jl[0], // max velocity
               jl[1], // initial velocity
               jl[2], // acceleration
               jl[3]  // deceleration
             );

    assert(jp[i].v <= jl[0]);

    totalTime = jp[i].t1 + jp[i].t2 + jp[i].t3;

    if(totalTime > maxTime) maxTime = totalTime;
    assert(maxTime >= 0);
}

//Profile Extension
Goto-diff execution

Goto-diff compares source code with respect to the previous successful eVolCheck execution.

There is no previous execution of eVolCheck.
Run bootstrapping.
<terminated, exit value: 0> Initial check of the program

Bootstrap is successful. Total time: 11.204

j[2],  // acceleration
j[3]  // deceleration

ASSERTION IS TRUE
INTERPOLATION TIME: 0.578
ASSERTION(S) HOLD(S)
Total number of steps: 1
TOTAL TIME FOR CHECKING THIS CLAIM: 11.204
#x: Done.
```c
// Code & comments start

int num, den, result;

num = d * (v0 * v0 + 3 * a * x);
den = d + a; // should both be +ve
```
<terminated> demo configuration [eVolCheck/Goto-Diff] Initial check of the program
demo configuration [eVolCheck/Goto-Diff]
<terminated> demo configuration [eVolCheck/Goto-Diff]

Debug

Substitutions | Refinements | Total time | SymbEx time | Slicer time
---------------|-------------|------------|-------------|----------------
0              | 0           | 1.762      | 0.017       | 0              

Error trace

Upgrade result
Upgrade is safe.
Total time: 1.762

VTT.c

```c
int num, den;
num = d * (v0 * v0 + 3 * a * x);
den = d + a; // should both be +ve
```

Project Explorer

Console

<terminated> demo configuration [eVolCheck/Goto-Diff] Upgrade checking

- summary was verified. go to the next check.
- checking validity of old summary for function: c::main
  - preserved. go to the next check.
Invalid summaries ratio: 0/20
TOTAL UPGRADE CHECKING TIME: 1.762
#X: Done.

257647 PINCETTE
Summary

• Technology for incremental upgrade checking
  • Local check → cheap check
  • Support for incremental SW development

• Tool: eVolCheck
  • http://www.verify.inf.usi.ch/evolcheck.html
Thank You!
Thank You!
Thank you for your attention!

- Software evolves
  - Small frequent upgrades
  - Complete re-verification impractical / infeasible

- Incremental verification
  - Store information from previous verification runs
  - Speed-up consecutive runs

- Our take
  - Bounded model checking
  - Interpolation-based function summaries

**Function summary**
- Over-approximation of real behavior
  - Considering the given bound
- Contains relevant information
  - Derived from UNSAT proof
- Expressed using function’s in/out parameters

**Usage**
1) Same code, different prop.
   - To approximate functions
2) Same prop., different code
   - In upgrade checking
Check with summaries

1) No error is reachable
   → OK, program is safe

2) Error is reachable
   A) due to over-approximation of summaries
   B) real error

Solution: Refine the abstraction
- Identify victim summaries
  - Error trace analysis
  - Dependency analysis
- Replace them by precise representation
Emergency slides
Correctness

\[ \varphi'_1 \land \varphi_2 \land \varphi_3 \land \varphi_{main} \land \varphi'_4 \land \varphi_5 \land \varphi_6 \land \text{error}_{main} \]

\[ \text{UNSAT} \]
Correctness

(iv) tree interpolant property:
\[ \varphi_f \land I_{\text{child}_1} \land \cdots \land I_{\text{child}_n} \rightarrow I_f \]

\[ \varphi'_1 \land \varphi_2 \land \varphi_3 \land \varphi_{\text{main}} \land \varphi'_4 \land \varphi_5 \land \varphi_6 \land \text{error}_{\text{main}} \]

UNSAT
Correctness

(iv) tree interpolant property:
\[ \varphi_f \land l_{\text{child}_1} \land \ldots \land l_{\text{child}_n} \rightarrow l_f \]

\[ \varphi'_1 \land \varphi_2 \land \varphi_3 \land \varphi_{\text{main}} \land \varphi'_4 \land \varphi_5 \land \varphi_6 \land \text{error}_{\text{main}} \]

\[ l_{\text{main}} \land \text{error}_{\text{main}} \]

UNSAT
Correctness

(iv) tree interpolant property:
\[ \varphi_f \land I_{\text{child}_1} \land \ldots \land I_{\text{child}_n} \rightarrow I_f \]

\[ \varphi'_1 \land \varphi_2 \land \varphi_3 \land \varphi_{\text{main}} \land \varphi'_4 \land \varphi_5 \land \varphi_6 \land \text{error}_{\text{main}} \]

\[ I_{\text{main}} \land \text{error}_{\text{main}} \]

UNSAT