Model Checking and Its Applications

Orna Grumberg
Technion, Israel

Verification and Deduction Mentoring Workshop
July 13, 2018
Personal data

- Ph.d. in (non-automated) verification
- Postdoc in Model Checking (MC) with Ed Clarke at CMU
- More than 30 years of research in MC: same title, different contents
Current research:

• Model checking for security
• New compositional methods
• Automated program repair
• Program difference
Why Computed-Aided Verification?

It is diverse:

• Logic
• Automata
• Algorithms
• Data structures
• Very efficient implementations that matters
Why Computed-Aided Verification?

Research, development, applications are done in academia, research labs, industry

A large variety of job types
Model Checking

- Given a system and a specification, does the system satisfy the specification.
Challenges in model checking

Model checking is successfully used, but...

- Scalability
- New types of systems
- New specifications (e.g. security)
- Applications in new areas
Technologies to help

Developed or adapted by the MC community

- SAT and SMT solvers
- Static analysis
- Abstraction - refinement
- Compositional verification
- Machine learning, automata learning

And many more...
In this talk

• We show how to exploit concepts and technologies from model checking to assist in stages of program development

  - Program difference
  - Automatic program repair
Modular Demand-Driven Analysis of Semantic Difference for Program Versions

[Trostanetski, Grumberg, Kroening, SAS 2017]
Programs change and evolve, raising the following interesting questions:

- Did the new version introduce new bugs or security vulnerabilities?
- Did the new version introduce the desired feature?
- More generally, how does the behavior of the program change?
Differences between program versions can be exploited for:

Regression testing of new version w.r.t. old version, used as “golden model”

Producing zero-day attacks on old version

characterizing changes in the program’s functionality
WHAT IS A DIFFERENCE IN BEHAVIOR?

```
void p1(int& x) {
    if (x < 0)
        x = -1;
    return;
    x--;  
    if (x >= 1)
        x=x+1;  
    return;
    else
        while (x == 1);
        x=0;
}

void p2(int& x) {
    if (x < 0)
        x = -1;
    return;
    x--;  
    if (x > 2)
        x=x+1;  
    return;
    else
        while (x == 1);
        x=0;
}
```
Difference for a pair of procedures $p_1, p_2$ is a triplet:

- **changed**: is the set of initial states for which both procedures terminate with different final states.
**FULL DIFFERENCE SUMMARY**

**Difference** for a pair of procedures $p_1, p_2$ is a triplet:

- **changed**: is the set of initial states for which both procedures terminate with different final states.
FULL DIFFERENCE SUMMARY

Difference for a pair of procedures $p_1, p_2$ is a triplet:

- **changed**: is the set of initial states for which both procedures terminate with different final states.

- **termination_changed**: is the set of initial states for which exactly one procedure terminates.
FULL DIFFERENCE SUMMARY

**Difference** for a pair of procedures $p_1, p_2$ is a triplet:

- **changed**: is the set of initial states for which both procedures terminate with different final states.
- **termination_changed**: is the set of initial states for which exactly one procedure terminates.

 violate **Mutual Termination**
FULL DIFFERENCE SUMMARY

**Difference** for a pair of procedures $p_1, p_2$ is a triplet:

- **changed**: is the set of initial states for which both procedures terminate with different final states.

- **termination_changed**: is the set of initial states for which exactly one procedure terminates.

- **unchanged**: is the set of initial states for which both procedures either terminate with the same final states, or both do not terminate.
FULL DIFFERENCE SUMMARY

**Difference** for a pair of procedures $p_1, p_2$ is a triplet:

- **changed**: is the set of initial states for which both procedures terminate with different final states.
- **termination\_changed**: is the set of initial states for which exactly one procedure terminates.
- **unchanged**: is the set of initial states for which both procedures either terminate with the same final states, or both do not terminate.
**FULL DIFFERENCE SUMMARY**

**Difference** for a pair of procedures $p_1, p_2$ is a triplet:

- **changed**: is the set of initial states for which both procedures terminate with different final states.

- **termination_changed**: is the set of initial states for which exactly one procedure terminates.

- **unchanged**: is the set of initial states for which both procedures either terminate with the same final states, or both do not terminate.

\[
\text{changed} \cup \text{termination}\_\text{changed} \cup \text{unchanged} = \text{input space}
\]
The difference summary is:

\begin{align*}
\text{changed} & := \{3\} \\
\text{termination}\_\text{changed} & := \{2\} \\
\text{unchanged} & := \{c : (c < 2) \lor (c > 3)\}
\end{align*}

**EXAMPLE**

```c
void p1(int& x) {
    if (x < 0)
        x = -1;
    return;
    x--; 
    if (x >= 1)
        x=x+1;
    return;
else
    while ( x ==1);
    x=0;
}
```

```c
void p2(int& x) {
    if (x < 0)
        x = -1;
    return;
    x--; 
    if (x > 2)
        x=x+1;
    return;
else
    while ( x == 1);
    x=0;
}
```
DIFFERENCE SUMMARY - COMPUTATION

- The full difference summary is incomputable
- We compute under-approximations of changed and unchanged, ignoring termination_changed:
  - A set $\text{computed}_\text{changed} \subseteq \text{changed}$
  - A set $\text{computed}_\text{unchanged} \subseteq \text{unchanged}$
DIFFERENCE SUMMARY - COMPUTATION

Input space
DIFFERENCE SUMMARY - COMPUTATION

computed_unchanged

?

computed_changed
DIFFERENCE SUMMARY - COMPUTATION

computed
unchanged

? Under approximation
DIFFERENCE SUMMARY - COMPUTATION

computed unchanged

Over approximation
A program is represented by a call graph.
Every procedure is represented by a Control Flow Graph (CFG).

We are interested in the input-output behavior of the program.
HOW PROGRAMS CHANGE

Changes are small, programs are big

Can our work be $O(\text{change})$ instead of $O(\text{program})$?
WHICH PROCEDURES COULD BE AFFECTED
WHICH PROCEDURES ARE AFFECTED
MAIN IDEAS

- **Modular analysis** applied to one pair of procedures at a time
  - No inlining

- Only affected procedures are analyzed

- Procedures need not be fully analyzed:
  - Unanalyzed parts are abstracted using uninterpreted functions
  - Refinement is applied upon demand

- **Anytime analysis**:
  - Does not necessarily terminate
  - Its partial results are meaningful
  - The longer it runs, the more precise its results are
EXAMPLE

\[ R_\pi(x) = x \geq 0 \land (x - 1 \geq 1) \]
\[ \equiv x \geq 2 \]
$T_\pi(x) = x$

流程图:
- $x < 0$
  - $x := x - 1$
  - $x \geq 1$
    - $x = x + 1$
    - $x = 0$
  - $x = -1$
- $x = 1$
  - $x = x + 1$
- $x = -1$

PATH CHARACTERIZATION

For a finite path $\pi$ in CFG from entry node to exit node:

- The **reachability condition** $R_{\pi}$ is a First Order Logic Formula, which guarantees that control traverses $\pi$.

- The **state transformation** $T_{\pi}$ is an n-tuple of expressions over program variables, describing the transformation on the variables’ values along $\pi$.

Both given in terms of variables at the entry node of $\pi$. 
A procedure summary of procedure $p$ is

$$Sum_p \subseteq \{ (R_\pi, T_\pi) \mid \pi \text{ is a finite path in } p \}$$

The full set of path summaries often cannot be computed, and might not be needed.
A possible summary for procedure $p$ is:

$$sum_p = \{(x < 0, -1), (x \geq 2, x)\}$$

Its uncovered part is

$$x \geq 0 \land x < 2$$
For each \((r_1, t_1)\) in \(sum_{p_1}\), \((r_2, t_2)\) in \(sum_{p_2}\)

- \(\text{diffCond} := r_1 \land r_2 \land (t_1 \neq t_2)\)
  - If \(\text{diffCond}\) is SAT, add it to \text{computed}_\text{changed}

- \(\text{eqCond} := r_1 \land r_2 \land (t_1 = t_2)\)
  - If \(\text{eqCond}\) is SAT, add it to \text{computed}_\text{unchanged}
SYMBOLIC EXECUTION
FOR COMPUTING \((R_\pi, T_\pi)\)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment (x := e)</td>
<td>(R_\pi^{i+1} = R_\pi^i)</td>
<td>(\forall y \neq x\ T_\pi^{i+1}[y] := T_\pi^i[y])</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(T_\pi^{i+1}[x] := e[V_p := T_\pi^i])</td>
</tr>
<tr>
<td>Test (B)</td>
<td>(R_\pi^{i+1} = R_\pi^i \land \bar{B})</td>
<td>(\forall x\ T_\pi^{i+1}[x] := T_\pi^i[x])</td>
</tr>
<tr>
<td>Procedure call (g(Y))</td>
<td></td>
<td>Inlined</td>
</tr>
</tbody>
</table>
COMPUTING THE SUMMARIES

To compute path summaries without in-lining called procedures:

We suggest modular symbolic execution
Modular Symbolic Execution

Path $\pi$ of procedure $p$ includes call $g(Y)$ at location $l_i$

$\text{sum}_g = \{ (r_1, t_1), \ldots, (r_n, t_n) \}$ previously computed

Instead of in-lining $g$ we compute:

$$R^{i+1}_\pi = R^i_\pi \land \bigvee_{j=1}^n r_j$$

$$T^{i+1}_\pi = ITE(r_1, t_1, \ldots, ITE(r_n, t_n, \text{error}) \ldots)$$
MODULAR SYMBOLIC EXECUTION

Path $\pi$ of procedure $p$ includes call $g(Y)$ at location $l_i$

$\text{sum}_g = \{ (r_1, t_1), ..., (r_n, t_n) \}$ previously computed

Instead of in-lining $g$ we compute:

$$R^{i+1}_\pi = R^i_\pi \land \bigvee_{j=1}^n r_j[V_g \leftarrow T^{i}_\pi(Y)]$$

$$T^{i+1}_\pi = \text{ITE}(r_1[V_g \leftarrow T^{i}_\pi(Y)], t_1[V_g \leftarrow T^{i}_\pi(Y)], ..., \text{ITE}(r_n[V_g \leftarrow T^{i}_\pi(Y)], t_n[V_g \leftarrow T^{i}_\pi(Y)], \text{error})...)$$
MODULAR SYMBOLIC EXECUTION

Path $\pi$ of procedure $p$ includes call $g(Y)$ at location $l_i$

$\text{sum}_g = \{ (r_1, t_1), ..., (r_n, t_n) \}$ previously computed

Instead of in-lining $g$ we compute:

$$R_{\pi}^{i+1} = R_{\pi}^i \land \bigvee_{j=1}^{n} r_j$$

$$T_{\pi}^{i+1} = \text{ITE}(r_1, t_1, ..., \text{ITE}(r_n, t_n, \text{error})...)$$
CAN WE DO BETTER?

- Use abstraction for the un-analyzed (uncovered) parts
- Later check if these parts are needed at all for the analysis of calling procedure
- If needed - refine
ABSTRACTION

Unanalyzed parts of a procedure are replaced by uninterpreted functions.

For matched procedures $g_1, g_2$ we have

- A common uninterpreted function $UF_{g_1, g_2}$
- Individual uninterpreted functions $UF_{g_1}$ and $UF_{g_2}$
For call $g_1(Y)$ with

$$sum_{g_1} = \{ (r_1, t_1), ..., (r_n, t_n) \}$$

$$R_{\pi}^{i+1} = R_{\pi}^i$$

$$T_{\pi}^{i+1} = \text{ITE}(r_1, t_1, \ldots \text{ITE}(r_n, t_n, \text{ITE(} \text{computed\_unchanged}, U_{Fg_1,g_2}, U_{Fg_1}) \text{))}$$

For $g_2(Y)$ we use $sum_{g_2}$ and $U_{Fg_2}$
REFINEMENT

Since we are using uninterpreted functions, the discovered difference may not be feasible:

```c
void p1(int& x) {
    if (x == 5) {
        abs1(x);
        if (x==0)
            x = 1;
    }
}
```

```c
void p2(int& x) {
    if (x == 5) {
        abs2(x);
        if (x==0)
            x = -1;
    }
}
```

abs1=abs2=abs
OVERALL ALGORITHM

- Start the analysis from the syntactically changed procedures. Analyze them with abstract modular symbolic execution up to a certain bound.
- Compute difference summaries for those procedures. If you can prove equivalence for all inputs – stop.
- Use refinement when needed.
- Repeat for calling procedures.
- Can be guided towards interesting procedures by the user.
We compared to two tools that prove equivalence between procedures:

- Regression Verification Tool (RVT)


- SymDiff

Sound and Complete Mutation-Based Program Repair

[Rothenberg, Grumberg, FM’16]
Mutation-Based Program Repair

- Sequential program
- Assertions in code
- Given set of mutations
- Can we use these mutations to make all assertions hold?
- Assignments, conditionals, loops, and function calls
- Assertion violation
- Operator replacement $ (+ \rightarrow - )$, constant manipulation $ (c \rightarrow c + 1) $
- Return all possible repairs
Example

```
int f(int x, int y){
1.   int z;
2.   if (x + y > 8) {
3.       z = x + y;
4.   } else {
5.       z = 9;
6.   }
7.   if (z ≥ 9) z = z - 1;
8.   assert(z > 8);
9.   return z;
}
```

\(x = 5, y = 2\)

\(z = 9\)

\(z = 8\)
Example

```c
int f(int x, int y){
1. int z;
2. if (x + y > 8) {
3.     z = x + y;
4. } else {
5.     z = 9;
6. }
7. if (z ≥ 9) z = z + 1;
8. assert(z > 8);
9. return z;
}
```

Mutation list:
- Replace + with –
- Replace – with +
- Replace ≥ with >

Repair list:
- option 1:
  - line 7: replace ≥ with >
- option 2:
  - line 7: replace – with +

At this point $z \geq 9$

Note: Repairs are minimal
Example

```c
int f(int x, int y){
1.   int z;
2.   if (x + y > 9) {
3.       z = x + y;
4.   } else {
5.       z = 10;
6.   }
7.   if (z ≥ 9)  z = z - 1;
8.   assert(z > 8);
9.   return z;
}
```

Mutation list:
- Replace + with –
- Replace – with +
- Replace > with ≥
- Replace ≥ with >
- Increase constants by 1

At this point \( z \geq 10 \)
Overview of our approach

Finding all correct programs from a finite set of programs

Output: All minimal repairs, sorted by size

Finding all unsatisfiable constraint sets from a finite set of constraint sets

Input: a buggy program

Translation

Mutation

Repair

\[\text{int } f(\text{int } x, \text{int } y)\{\]
1. \[\text{int } z;\]
2. \[\text{if } (x + y > 8) \{\]
3. \[\quad z = x + y;\]
4. \[\} \text{ else } \{\]
5. \[\quad z = 9;\]
6. \[\} \]
7. \[\text{if } (z \geq 9) \quad z = z - 1;\]
8. \[\text{assert}(z > 8);\]
9. \[\text{return } z;\]
\[\}\]
First step - Translation

Goal: Translate the program into a set of constraints which is satisfiable iff the program has a bug (i.e. there exists an input for which an assertion fails)

Work by Clarke, Kroening, Lerda (TACAS 2004) (CBMC)
- Simplification
- Unwinding of loops
  - a bounded number of unwinding
- Conversion to SSA

Correctness is bounded
int f (int x, int y) {
1.   int z;
2.   if (x + y > 8) {
3.       z = x + y;
4.   } else {
5.       z = 9;
6.   }
7.   if (z \geq 9) {
8.       z = z - 1;
9.   } else {
10.  assert (z > 8);
11.  return z;
12. }
}

\[ g_1 = x_1 + y_1 > 8 \]
\[ z_2 = x_1 + y_1 \]
\[ z_3 = 9 \]
\[ z_4 = g_1 ? z_2 : z_3 \]
\[ b_1 = z_4 \geq 9 \]
\[ z_5 = z_4 - 1 \]
\[ z_6 = b_1 ? z_5 : z_4 \]
\[ z_6 \leq 8 \]
Second step - Mutation

```c
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x - y;
    } else {
        z = 9;
    }
    if (z ≥ 9) {
        z = z - 1;
    }
    assert(z > 8);
    return z;
}
```

Mutation list:
- Replace + with –
- Replace – with +
- Replace > with ≥
- Replace ≥ with >

{ \( g_1 = x_1 + y_1 > 8 \), \( g_2 = x_1 - y_1 > 8 \), \( g_1 = x_1 + y_1 ≥ 8 \), \( g_2 = x_1 - y_1 ≥ 8 \) } }
Third step - Repair

```c
int f(int x, int y){
    SAT solver
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
}

SMT solver
if (z ≥ 9) {
    z = z + 1;
}
assert(z > 8);
return z;
}
```

\[
\begin{align*}
\{ g_1 = x_1 + y_1 > 8, g_1 = x_1 - y_1 > 8, g_1 = x_1 + y_1 ≥ 8 \} \\
\{ z_2 = x_1 + y_1, z_2 = x_1 - y_1 \} \\
\{ z_3 = 9 \} \\
\{ b_1 = z_4 ≥ 9, b_1 = z_4 > 9 \} \\
\{ z_5 = z_4 - 1, z_5 = z_4 + 1 \} \\
\{ z_6 = b_1? z_5: z_4 \} \\
\{ z_6 ≤ 8 \}
\end{align*}
\]
**Repair**

Choose candidate program of \( \text{size} = 1 \)

SAT solver

Blocking clause for specific assignment

Blocking clause for this assignment

And all other supersets of changes

SMT solver

\( g_1 = x_1 + y_1 > 8 \)

\( z_2 = x_1 + y_1 \)

\( z_3 = 9 \)

\( b_1 = z_4 \geq 9 \)

\( b_1 = z_4 > 9 \)

\( z_5 = z_4 - 1 \)

\( z_5 = z_4 + 1 \)

\( \text{SAT solver} \)

\( c_1 = 0 \)

\( c_2 = 0 \)

\( c_3 = 0 \)

\( c_4 = 1 \)

\( c_5 = 0 \)

\( c_6 = 1 \)

\( c_7 = 0 \)

\( c_8 = 0 \)

\( c_9 = 1 \)

\( c_{10} = 0 \)

\( \text{SAT} \)

\( g_1 = x_1 + y_1 > 8 \)

\( z_2 = x_1 + y_1 \)

\( z_3 = 9 \)

\( b_1 = z_4 \geq 9 \)

\( b_1 = z_4 > 9 \)

\( z_5 = z_4 - 1 \)

\( z_5 = z_4 + 1 \)

\( \text{UNSAT} \)
Summary

• We suggest a repair method which returns all minimal (bounded) correct programs, in increasing size
  • Based on a given set of mutations

• Minimal mutations: No change is made to the original program unless necessary

• If no repaired program is returned then the given mutations cannot repair the program
Summary

The method can assist a programmer in debugging in initial stages of development

• When bugs are simple, but many

• And also can help beginners
  • Educational tool for students

• Difference analysis can be used to prioritize the returned repaired programs
Evolving Software
Questions?