Trust in Formal Methods Toolchains

Arie Gurfinkel
Software Engineering Institute
Carnegie Mellon University

July 14, 2013
VeriSure
About Me

Researcher at Carnegie Mellon Software Engineering Institute
Working on Software Model Checking and Static Analysis

Developer of many verification tools and libraries
• Xchek
• TLQSolver
• Yasm
• Linear Decision Diagrams
• Whale
• UFO
• Vinta
• REK

UFO/Vinta won the 2nd Software Verification Competition (SVCOMP)
Automated Software Analysis

Program \rightarrow \text{Automated Analysis} \rightarrow \text{Property}

Correct + Proof \rightarrow \text{Incorrect + Counterexample}

Software Model Checking with Predicate Abstraction
e.g., Microsoft’s SDV

Abstract Interpretation with Numeric Abstraction
e.g., ASTREE, Polyspace
No expensive testing!
  • Verification is exhaustive

Simpler certification!
  • Just check formal arguments

Can we trust formal methods tools? What can go wrong?
Tusting Automated Verification Tools

How should automatic verifiers be qualified for certification?

What is the basis for automatic program analysis (or other automatic formal methods) to replace testing?

Verify the verifier

- (too) expensive
- verifiers are often very complex tools
- difficult to continuously adapt tools to project-specific needs

Proof-producing (or certifying) verifier

- Only the proof is important – not the tool that produced it
- Only the proof-checker needs to be verified/qualified
- Single proof-checker can be re-used in many projects
Evidence Producing Analysis

Active research area
- proof carrying code, certifying model checking, model carrying code etc.
- Few tools available. Some preliminary commercial application in the telecom domain.
- Static context. Good for ensuring absence of problems.
- Low automation. Applies to source or binary. High confidence.

Not that simple in practice !!!
A 3-Depth Look…

Environment model

Low level property
Program = (Text, Semantics)

Hard to get right

Diff sem used by diff tools

Hard to verify

Verifier

No + Counterexample

Yes + Proof

Proof Checker

Good

Bad

Compiler

Executable

Hardware

Front-End

VC

Hard to verify

 compiler

Executable

Hardware

Real Env

Good

Bad

?=?
Five Hazards (Gaps) of Automated Verification

Soundness Gap
- Intentional and unintentional unsoundness in the verification engine
  - e.g., rational instead of bitvector arithmetic, simplified memory model, etc.

Semantic Gap
- Compiler and verifier use different interpretation of the programming language

Specification Gap
- Expressing high-level specifications by low-level verifiable properties

Property Gap
- Formalizing low-level properties in temporal logic and/or assertions

Environment Gap
- Too coarse / unsound / unfaithful model of the environment
Mitigating The Soundness Gap
Mitigating The Soundness Gap

Proof-producing verifier makes the soundness gap explicit

• the soundness of the proof can be established by a “simple” checker
• all assumptions are stated explicitly

Open questions:

• how to generate proofs for explicit Model Checking
  – e.g., SPIN, Java PathFinder
• how to generate partial proofs for non-exhaustive methods
  – e.g., KLEE, Sage
• how to deal with “intentional” unsoundness
  – e.g., rational arithmetic instead of bitvectors, memory models, …
Mitigating the Property Gap: Vacuity in Model Checking

Joint work with Marsha Chechik
Vacuity: Mitigating Property Gap

Model Checking Perspective: Never trust a *True* answer from a Model Checker

When a property is violated, a counterexample is a certificate that can be examined by the user for validity

When a property is satisfied, there is no feedback!

It is very easy to formally state something very trivial in a very complex way
MODULE main

VAR
    send : {s0,s1,s2};
    recv : {r0,r1,r2};

    ack : boolean;
    req : boolean;

ASSIGN
    init(ack):=FALSE;
    init(req):=FALSE;

    init(send):= s0;
    init(recv):= r0;

next (send) :=
    case
        send=s0:{s0,s1};
        send=s1:s2;
        send=s2&ack:s0;
        TRUE:send;
    esac;

next (recv) :=
    case
        recv=r0&req:r1;
        recv=r1:r2;
        recv=r2:r0;
        TRUE: recv;
    esac;

next (ack) :=
    case
        recv=r2:TRUE;
        TRUE: ack;
    esac;

next (req) :=
    case
        send=s1:FALSE;
        TRUE: req;
    esac;

SPEC AG (req -> AF ack)
Can A TRUE Result of Model Checker be Trusted

Antecedent Failure [Beatty & Bryant 1994]

• A temporal formula $\text{AG}(p \Rightarrow q)$ suffers an *antecedent failure* in model $M$ iff $M \vDash \text{AG}(p \Rightarrow q)$ AND $M \vDash \text{AG}(\neg p)$

Vacuity [Beer et al. 1997]

• A temporal formula $\varphi$ is satisfied *vacuously* by $M$ iff there exists a sub-formula $p$ of $\varphi$ such that $M \vDash \varphi[p \leftarrow q]$ for every other formula $q$

• e.g., $M \vDash \text{AG}(r \Rightarrow \text{AF } a)$ and $M \vDash \text{AG}(r \Rightarrow \text{AF } \neg a)$ and $\text{AG}(r \Rightarrow \text{AF } \neg r)$ and $\text{AG}(r \Rightarrow \text{AF } \text{FALSE})$, …
Vacuity Detection: Single Occurrence

\( \varphi \) is vacuous in \( M \) iff there exists an occurrence of a subformula \( p \) such that

- \( M \not\models \varphi[p \leftarrow \text{TRUE}] \) and \( M \not\models \varphi[p \leftarrow \text{FALSE}] \)

\[
\begin{array}{c}
M \not\models AG (req \Rightarrow AF \text{ TRUE}) \\
\quad \Rightarrow M \not\models AG \text{ TRUE}
\end{array}
\]

\[
\begin{array}{c}
M \not\models AG (req \Rightarrow AF \text{ FALSE}) \\
\quad \Rightarrow M \not\models AG \neg \text{req}
\end{array}
\]

\[
\begin{array}{c}
M \not\models AG (\text{TRUE} \Rightarrow AF \text{ ack}) \\
\quad \Rightarrow M \not\models AG \text{ AF ack}
\end{array}
\]

\[
\begin{array}{c}
M \not\models AG (\text{FALSE} \Rightarrow AF \text{ ack}) \\
\quad \Rightarrow M \not\models AG \text{ TRUE}
\end{array}
\]
Detecting Vacuity in Multiple Occurrences

Is $AG\ (req \Rightarrow AF\ req)$ vacuous? Should it be?

\[
\begin{align*}
M \models AG\ (TRUE \Rightarrow AF\ TRUE) & \quad M \not\models AG\ TRUE \\
M \models AG\ TRUE & \quad M \not\models AG\ TRUE
\end{align*}
\]

Is $AG\ (req \Rightarrow AX\ req)$ vacuous? Should it be?

\[
\begin{align*}
M \models AG\ (TRUE \Rightarrow AX\ TRUE) & \quad M \not\models AG\ TRUE \\
M \models AG\ TRUE & \quad M \not\models AG\ TRUE
\end{align*}
\]
Detecting Vacuity in Multiple Occurrences: ACTL

An \( \text{ACTL} \varphi \) is vacuous in \( M \) iff there exists an a subformula \( p \) such that

\[
M \not\models \varphi[p \leftarrow x], \text{ where } x \text{ is a non-deterministic variable}
\]

Is \( \text{AG} (\text{req} \Rightarrow \text{AF req}) \) vacuous? Should it be?

\[
\begin{align*}
M \not\models \text{AG} (x \Rightarrow \text{AF} x) \\
\hline
M \not\models \text{AG} \text{ TRUE} \\
\end{align*}
\]

Always vacuous!!!

Is \( \text{AG} (\text{req} \Rightarrow \text{AX req}) \) vacuous? Should it be?

\[
\begin{align*}
M \not\models \text{AG} (x \Rightarrow \text{AX} x) \\
\hline
\text{can’t reduce} \\
\end{align*}
\]

Not vacuous!!!

\[
M \models \text{AG} \text{ TRUE}
\]

can’t reduce
Mitigating the Environment Gap:

Environment Guarantee

Joint work with Marsha Chechik and Mihaela Gheorghiu
Validity of Vacuity Results

Properties may hold for wrong reasons
vacuity detection [Beer’97] – are all parts of property relevant?
• Example: “every time there is a request for a resource, it is fulfilled”
• ... holds if there are no requests!

Pros
• May help identify errors

Cons
• General definition yields many false positives
  – Vacuous but not “wrong”
• Hard to go from a violation to a fix
• Property-centric
System Structure

Traffic Light Example:

Environment

Component

Interface

Constructed for verification

Can we trust it?

What we are building

Component

Traffic light controller

Crossing environment

sensor

light
**Our Goal**

**Model-centric** identification of threats to validity

Pros:
- Ease of understanding
- No false positives
  - each reported error is present in the model

**Definition:** a property is **guaranteed by environment** if component is irrelevant for verification

![Diagram showing the relationship between Environment, Any Component, and Property P]

Environment

Property P

Any Component
Formalizing Environment Guarantees

An Environment $E$ is a tuple $(V, R)$
- $V$ - environment and component variables $(V_e, V_c)$
- $R$ – environment rules

A (Closed) Model $M$ is a tuple $(V, R, C)$
- ... a closure of $E$ with component rules $C$

Formal Definition:
- Environment $E$ guarantees a property $P$ iff
- for all closures $M$ of $E$, $M$ satisfies $P$
Special Case: Universal Properties

Universal properties: about all executions

- AG p (“in all states”)
- AF p (“on all paths”)

“Worst” component is the most non-deterministic

- Component variables change non-deterministically at each step
- = No constraints on component variables

Theorem

- A universal property is guaranteed by the environment iff it holds under the worst component.
Universal Properties: Implementation

Syntactic

Compose with Worst Component

Model Check

Environment Guarantee

Component

Compose

Model Check

Closed Model

Yes

No

Closed Model

Yes

No
Case Study: TCAS II

(Air) Traffic Collision Avoidance System
• Avoids collision between planes flying in the same space
• … by producing advisories to pilot for direction and strength of move

An example advisory:
• “CLIMB at 1500 to 2000 fpm”

[Leveson et al. 94]
TCAS II Model

NuSMV model [Chan et al. 98]
TCAS II Properties

Deterministic Advisory
  • advisory changes in a deterministic fashion

Increase Climb/Descent
  • increase climb does not change immediately to increase descent

Direction of the Other Aircraft
  • direction does not change unless noticed by Own Aircraft

Advisories are Consistent
  • direction (Up/Down) and strength (Pos/Neg) match
## Experimental Results

<table>
<thead>
<tr>
<th>Properties</th>
<th>Results</th>
<th>Time (sec.)</th>
<th>BDD nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Full</td>
<td>Env.</td>
</tr>
<tr>
<td>Reachability</td>
<td>-</td>
<td>-</td>
<td>1034.61</td>
</tr>
<tr>
<td>Deterministic Advisory</td>
<td>TRUE</td>
<td>TRUE</td>
<td>20.63</td>
</tr>
<tr>
<td>Increase Climb/Descend</td>
<td>TRUE</td>
<td>TRUE</td>
<td>20.81</td>
</tr>
<tr>
<td>Other Aircraft Direction</td>
<td>TRUE</td>
<td>TRUE</td>
<td>27.83</td>
</tr>
<tr>
<td>Consistent Advisory</td>
<td>TRUE FALSE</td>
<td>40.67</td>
<td>6.1</td>
</tr>
</tbody>
</table>


Mitigating the Semantic Gap
Mitigating the Semantics Gap

Combine a compile and a verifier in a single architecture

Use compiler intermediate representation as the “ground truth”

Propagate verification certificates from the verifier down to the compiled code
V&C Architecture (Verifier & Compiler)

Legend:
P – prog in ir
S – simplified
C – self-certified ir

Yes
No
Proofs are “Witnesses” to Success*

Program invariants are embedded as BEGIN and INV calls in compiled code.

```c
while(n < 10) {
    BEGIN();
    INV((n >= 0) && (n < 10));
    n = n + 1;
}
```

These invariants are sufficient to construct a proof of conformance to a claim (policy)

Adapt: From Low-Level to High-Level Certificate

Formal intermediate representation

Simplified/Transformed intermediate representation

P

S

Cert(P) ← Adapt ← Cert(S)

Certificate / Invariant for P

Certificate / Invariant for S
Iteratively Guess a Simulation and Adapt Cert.

Guess simulation between CFGs

Extend to states

Refine

Is adapted certificate safe?

Cert(S)
Five Hazards (Gaps) of Automated Verification

Soundness Gap
• Intentional and unintentional unsoundness in the verification engine
  • e.g., rational instead of bitvector arithmetic, simplified memory model, etc.

Semantic Gap
• Compiler and verifier use different interpretation of the programming language

Specification Gap
• Expressing high-level specifications by low-level verifiable properties

Property Gap
• Formalizing low-level properties in temporal logic and/or assertions

Environment Gap
• Too coarse / unsound / unfaithful model of the environment
Contact Information

Presenter
Arie Gurfinkel
RTSS
Telephone: +1 412-268-5800
Email: arie@cmu.edu

Web:
www.sei.cmu.edu
http://www.sei.cmu.edu/contact.cfm

U.S. mail:
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Customer Relations
Email: info@sei.cmu.edu
Telephone: +1 412-268-5800
SEI Phone: +1 412-268-5800
SEI Fax: +1 412-268-6257
But things are not that simple in practice !!!