Formal verification of automotive embedded software

Vassil Todorov - LRI, Groupe PSA, Université Paris-Saclay
Frédéric Boulanger - LRI, CentraleSupélec, Université Paris-Saclay
Safouan Taha - LRI, CentraleSupélec, Université Paris-Saclay
Need for safe advanced driver assistance systems ⇒ Formal methods?
Need for new software development process integrating formal methods

1983: Balzer proposed for the first time such a process placing the formal methods in the heart of the development (link between requirements, prototype, implementation)

2018: there is still no standard process integrating formal methods: each company needs to create it

Need to apply formal methods on automotive use cases to

- identify the potential perimeter of their application
- identify the impacts and difficulties to anticipate
- verify which method could be integrated best in which part of the process
Panorama of the formal verification tools

Automatic

Static analyzers

Model-checkers

Deductive provers

Interactive

Interactive provers

Basic safety

Total correctness

$10^{6}$ LOC

$10^{2}$ LOC

Panorama proposed by Xavier Leroy, INRIA
Abstract interpretation

- **Context**
  - AUTOSAR application software, 300 K lines of C code
  - Sound static analyzers were known to produce a big amount of false positives
  - People using model-based design think there is no need to analyze the code because it is generated automatically and is thus correct

- **Experiments**
  - MathWorks Polyspace Code Prover R2016b
  - AbsInt Astrée 17.04i

- **Results**
  - Reasonable number of alarms that could be analyzed by the engineers
  - Not the same number of alarms for the two tools although they are sound
  - Model-based design can produce runtime errors difficult/impossible to find by testing
  - ISO 26262 (v2018) introduces abstract interpretation

---

```
typedef unsigned char u8;
void main (void)
{
    u8 a = 1;
    u8 b = ~a;
}
```

**What is a real bug?**

---

**Table 7 — Methods for software unit verification**

<table>
<thead>
<tr>
<th>Methods</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Walk-through</td>
<td></td>
<td>++</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Peer-programming</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3. Inspection</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>4. Semi-formal verification</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>5. Formal verification</td>
<td>+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Control flow analysis</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>7. Data flow analysis</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>8. Static code analysis</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>9. Static analysis based on abstract interpretation</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10. Requirements-based test</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>11. Interface test</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>12. Fault injection test</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>13. Resource usage evaluation</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>14. Back-to-back comparison test</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
</tbody>
</table>
SAT/SMT-based model checking

- **Context**
  - Cruise Controller function allocated to body controller unit
  - 110 pages of textual requirements

- **Experiments**
  - Model all textual requirements in SCADE (formally defined language)
  - Transform the SCADE model in Lustre (via XSLT)
  - Compare the native Prover plugin with different Lustre model checkers (JKind, Kind2, GaTEL)
  - Prove the validity of the safety requirements about the Cruise controller deactivation

- **Example of property to prove**

  *In order to respect the safety objectives in the case the brake pedal sensor is not working, 2 seconds of deceleration under 144 without pressing the brake pedal shall turn off the function."

- **Results**
  - A bug previously found was confirmed only in few seconds
  - Certain properties like the previous one are not inductive and we need to explore all the states of the model ⇒ Problem : if we increase the time from 2s to 2min none of the model checkers is able to prove it within 24h
  - *PDR/IC3* is generating invariants starting from the property but in our case we needed to find an inductive relation between the variables of the property and the model. How can we find this relation automatically?
  - We need a better invariant generator and are working on its improvement

Deductive proof

- **Context**
  - SQRT function using linear integer interpolation over 40 values

- **Experiments**
  - Annotate the code using ACSL specification language
  - Use Frama-C to prove the correctness for a given precision
  - Develop the same function in Ada
  - Use SPARK to prove the correctness for a given precision

- **Results**
  - SPARK succeeded with 40 values at once using a little hint: bit vectors are easy to handle by the SMT solvers
  - SPARK can return a counterexample when a contract fails
  - Frama-C proved the correctness for 8 values but when extending the table to 40 values it didn’t scale
  - After submitting the problem to the developers of Frama-C we got a new version integrating the Colibri SMT solver which worked. The reason was that Colibri worked with modular arithmetic unlike the others
Conclusion and future work

- **Formal methods**
  - Our experiments showed that formal methods definitely bring more confidence in the software verification process finding bugs earlier and faster than testing
  - They also help thinking about and getting a better specification
  - They can be introduced incrementally

- **Formal tools**
  - Some are mature enough to be used in an industrial context
  - Tools are complex and may contain bugs ⇒ use of more than one tool?
  - Problem: Using mathematical numbers (reals and infinite integers) while we want to prove programs using floating point numbers and bounded integers
  - Challenges: scalability, floating point numbers, nonlinear arithmetic, timers, counters, lookup tables

- **Future work**
  - Can we get 0 false alarm using abstract interpretation tools and what is the prerequisite?
  - Can we get a better confidence in the software development using a formally defined language for MBD?
  - Can we get better invariant generation using a different methodology (e.g. functionally typing the variables)?