Proving the Absence of Bugs

Case Studies on Automated Formal Verification of Java Programs Using KeY

TU Delft

Why bother with formal verification?

➜ To avoid the devastating impact of bugs in critical systems.

What is KeY?

➜ KeY is a semi-automatic deductive verifier for proving the correctness of Java programs.

What does 'correctness' even mean?

What are the logical foundations and how does KeY work in practice?

What kind of guarantees can it give?

What assumptions does it make?

How did we answer these questions?

We performed two case studies where we -

- verified a key-value store
- partially verified the insertion sort sorting algorithm

This involved providing specifications such that the prover could automatically verify that our implementations adhered them.

Picture of the Antares rocket exploding after take-off. The reason for the explosion was **not** a software bug in this case. Ariane 5 flight V88 is an infamous example of an explosion due to a bug, but the pictures aren't as pretty.

Correctness for KeY means proving that programs adhere to their specifications.

It can provide guarantees about functional and some non-functional properties like information flow and resource usage for sequential Java 7 code. However, it cannot directly check for some errors like StackOverflowError.

It makes assumptions like not allowing Objects to be null, not allowing integers overflows, etc. Many of these assumptions can be adjusted in its settings

How does KeY verify correctness?

- It takes programs with **pre and post conditions** provided as annotations in comments:

```
/*@ requires x > 0;
  @ ensures x > \old(x);
  @ // Example of a contract @*/
 void double() { this.x *= 2; }
```

- Each contract is reduced to a **sequent**: $P_1 \wedge P_2 \wedge \ldots \wedge P_n \implies Q_1 \vee Q_2 \vee \ldots \vee Q_m$
- Each term in the sequent is a (Java) Dynamic Logic expression and can be written as $p \rightarrow [q]r$ where $[q]r$ is a 'modality' stating that r **holds after executing** q.
- Thus, the contract for *double()* will be written as **true ==> x > 0 → [this.x*=2;]x > \old(x)**
- **Symbolic execution** is used to reduce code to logic statements, keeping track of state changes.
- Logical inference rules are applied to rewrite the sequent until its validity can be established, or no more rules can be applied automatically, in which case the user can apply rules by hand to guide the prover

Key takeaways:

- **KeY is capable of verifying complex software.** It has been used to find bugs in OpenJDK's LinkedList and Tim Sort, used by apps on billions of devices.
- **It has a steep learning curve.** It relies on user interaction for guidance, requiring an in-depth knowledge of the underlying proof theory.
- **Getting feedback on why proofs fail is difficult** as it requires inspecting the unfinished proof. Because of this, it cannot replace testing, which provides easy-to-understand feedback fast.

Tejas Kochar[1]
T.Kochar@student.tudelft.nl

Supervisors: Benedikt Ahrens[1], Kobe Wullaert[1]
[1]EEMCS, TU Delft, Netherlands