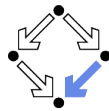


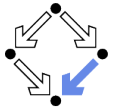
Verifying Java Programs with KeY

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
<https://www.risc.jku.at>



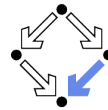
Verifying Java Programs



- **Extended static checking of Java programs:**
 - Even if no error is reported, a program may violate its specification.
 - Unsound calculus for verifying while loops.
 - Even correct programs may trigger error reports:
 - Incomplete calculus for verifying while loops.
 - Incomplete calculus in automatic decision procedure (Simplify).
- **Verification of Java programs:**
 - Sound verification calculus.
 - Not unfolding of loops, but loop reasoning based on invariants.
 - Loop invariants must be typically provided by user.
 - Automatic generation of verification conditions.
 - From JML-annotated Java program, proof obligations are derived.
 - Human-guided proofs of these conditions (using a proof assistant).
 - Simple conditions automatically proved by automatic procedure.

We will now deal with an integrated environment for this purpose.

The KeY Tool

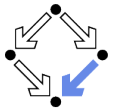


<http://www.key-project.org>

- **KeY:** environment for verification of JavaCard programs.
 - Subset of Java for smartcard applications and embedded systems.
 - Universities of Karlsruhe, Koblenz, Chalmers, 1998–
 - Beckert et al: “Deductive Software Verification – The KeY Book: From Theory to Practice”, Springer, 2016.
 - “Chapter 16: Formal Verification with KeY: A Tutorial”
- **Specification language:** JML.
 - Original: OCL (Object Constraint Language), part of UML standard.
- **Logical framework:** Dynamic Logic (DL).
 - Successor/generalization of Hoare Logic.
 - Integrated prover with interfaces to external decision procedures.
 - Z3, CVC4, CVC5.

Now only JML is supported as a specification language.

Dynamic Logic



Further development of Hoare Logic to a modal logic.

- **Hoare logic:** two separate kinds of statements.
 - Formulas P, Q constraining program states.
 - Hoare triples $\{P\}C\{Q\}$ constraining state transitions.
- **Dynamic logic:** single kind of statement.
 - Predicate logic formulas extended by two kinds of modalities.
 - $[C]Q$ ($\Leftrightarrow \neg\langle C\rangle\neg Q$)
 - Every state that can be reached by the execution of C satisfies Q .
 - The statement is trivially true, if C does not terminate.
 - $\langle C\rangle Q$ ($\Leftrightarrow \neg[C]\neg Q$)
 - There exists some state that can be reached by the execution of C and that satisfies Q .
 - The statement is only true, if C terminates.

States and state transitions can be described by DL formulas.

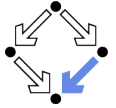


Dynamic Logic versus Hoare Logic

Hoare triple $\{P\}C\{Q\}$ can be expressed as a DL formula.

- **Partial correctness interpretation:** $P \Rightarrow [C]Q$
 - If P holds in the current state and the execution of C reaches another state, then Q holds in that state.
 - Equivalent to the partial correctness interpretation of $\{P\}C\{Q\}$.
- **Total correctness interpretation:** $P \Rightarrow \langle C \rangle Q$
 - If P holds in the current state, then there exists another state that can be reached by the execution of C in which Q holds.
 - If C is deterministic, there exists at most one such state; then equivalent to the total correctness interpretation of $\{P\}C\{Q\}$.

For deterministic programs, the interpretations coincide.

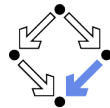


Advantages of Dynamic Logic

Modal formulas can also occur in the context of quantifiers.

- **Hoare Logic:** $\{x = a\} y := x * x \{x = a \wedge y = a^2\}$
 - Use of free mathematical variable a to denote the “old” value of x .
- **Dynamic logic:** $\forall a : x = a \Rightarrow [y := x * x] x = a \wedge y = a^2$
 - Quantifiers can be used to restrict the scopes of mathematical variables across state transitions.

Set of DL formulas is closed under the usual logical operations.



A Calculus for Dynamic Logic

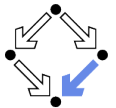
- **A core language of commands (non-deterministic):**

$X := T$... assignment
 $C_1; C_2$... sequential composition
 $C_1 \cup C_2$... non-deterministic choice
 C^* ... iteration (zero or more times)
 $F?$... test (blocks if F is false)

- **A high-level language of commands (deterministic):**

skip = true?
abort = false?
 $X := T$
 $C_1; C_2$
if F **then** C_1 **else** C_2 = $(F?; C_1) \cup ((\neg F)?; C_2)$
if F **then** C = $(F?; C) \cup (\neg F)?$
while F **do** C = $(F?; C)^*; (\neg F)?$

A calculus is defined for dynamic logic with the core command language.



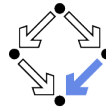
A Calculus for Dynamic Logic

- **Basic rules:**
 - Rules for predicate logic extended by general rules for modalities.

- **Command-related rules:**

$$\begin{array}{l}
 \frac{\Gamma \vdash F[T/X]}{\Gamma \vdash [X := T]F} \\
 \frac{\Gamma \vdash [C_1][C_2]F}{\Gamma \vdash [C_1; C_2]F} \\
 \frac{\Gamma \vdash [C_1]F \quad \Gamma \vdash [C_2]F}{\Gamma \vdash [C_1 \cup C_2]F} \\
 \frac{\Gamma \vdash F \Rightarrow [C]F}{\Gamma \vdash F \Rightarrow [C^*]F} \\
 \frac{\Gamma \vdash F \Rightarrow G}{\Gamma \vdash [F?]G}
 \end{array}$$

From these, Hoare-like rules for the high-level language can be derived.



Objects and Updates

Calculus has to deal with the pointer semantics of Java objects.

- **Aliasing:** two variables o, o' may refer to the same object.
 - Field assignment $o.a := T$ may also affect the value of $o'.a$.
- **Update formulas:** $\{o.a \leftarrow T\}F$
 - Truth value of F in state after the assignment $o.a := T$.

■ **Field assignment rule:**

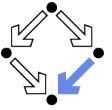
$$\frac{\Gamma \vdash \{o.a \leftarrow T\}F}{\Gamma \vdash [o.a := T]F}$$

■ **Field access rule:**

$$\frac{\Gamma, o = o' \vdash F(T) \quad \Gamma, o \neq o' \vdash F(o'.a)}{\Gamma \vdash \{o.a \leftarrow T\}F(o'.a)}$$

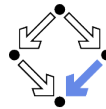
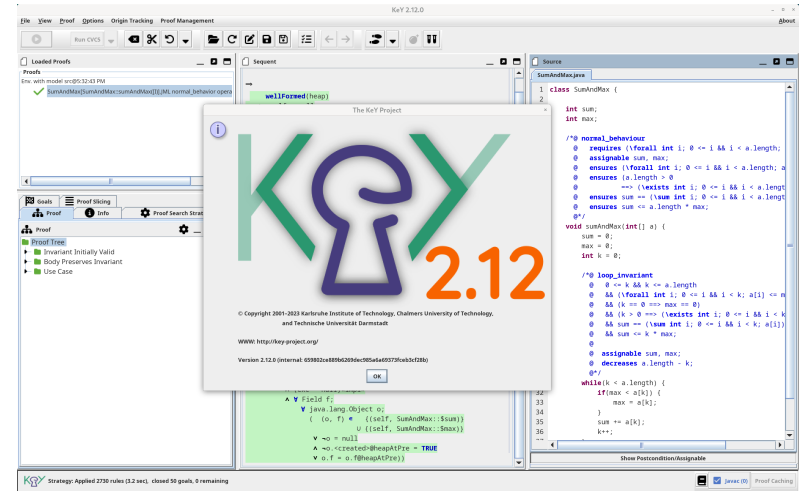
- Case distinction depending on whether o and o' refer to same object.
- Only applied as last resort (after all other rules of the calculus).

Considerable complication of verifications.



The KeY Prover

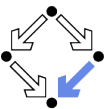
> KeY &



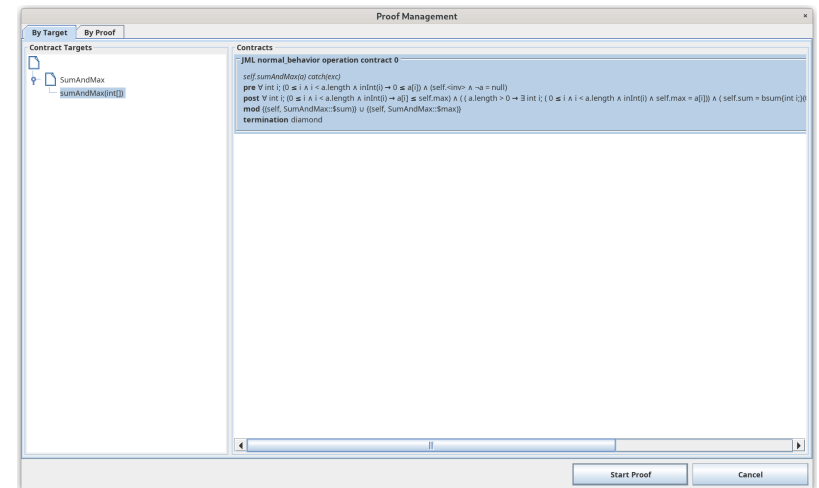
A Simple Example

File/Load Example/Getting Started/Sum and Max

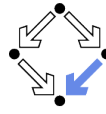
```
class SumAndMax {
  int sum; int max;
  /*@ requires (\forall forall int i;
    @ 0 <= i && i < a.length; 0 <= a[i]);
    @ assignable sum, max;
    @ ensures (\forall forall int i;
    @ 0 <= i && i < a.length; a[i] <= max);
    @ ensures (a.length > 0 ==>
    @ (\exists exists int i;
    @ 0 <= i && i < a.length;
    @ max == a[i]));
    @ ensures sum == (\sum int i;
    @ 0 <= i && i < a.length; a[i]);
    @ ensures sum <= a.length * max;
  @*/
  void sumAndMax(int[] a) {
    sum = 0;
    max = 0;
    int k = 0;
    /*@ loop_invariant
    @ 0 <= k && k <= a.length
    @ &&& (\forall forall int i;
    @ 0 <= i && i < k; a[i] <= max)
    @ &&& (k == 0 ==> max == 0)
    @ &&& (k > 0 ==> (\exists exists int i;
    @ 0 <= i && i < k; max == a[i]))
    @ &&& sum == (\sum int i;
    @ 0 <= i && i < k; a[i])
    @ assignable sum, max;
    @ decreases a.length - k;
    @*/
    while (k < a.length) {
      if (max < a[k]) max = a[k];
      sum += a[k];
      k++;
    } }
}
```



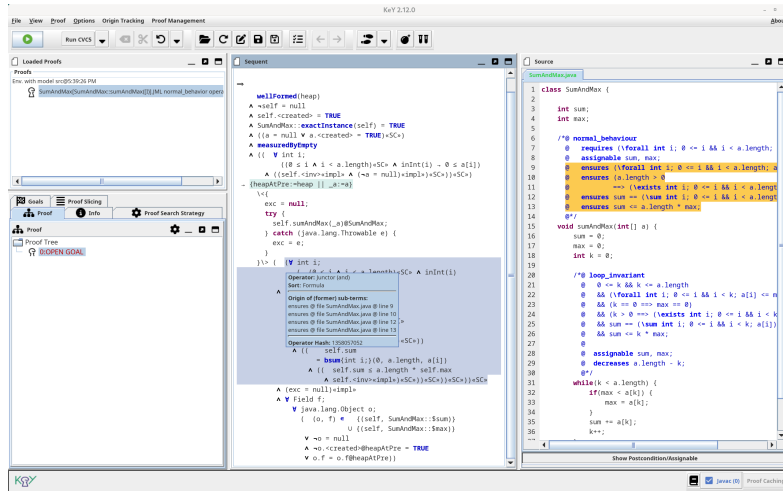
A Simple Example (Contd)



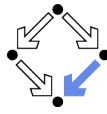
Generate the proof obligations and choose one for verification.



A Simple Example (Contd'2)



The proof obligation in Dynamic Logic.



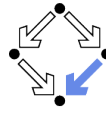
Proof Obligation

Two lists of formulas separated by a horizontal line.

$$\begin{array}{l} A_1 \\ \dots \\ A_n \\ \hline B_1 \\ \dots \\ B_m \end{array}$$

- Interpretation: $(A_1 \wedge \dots \wedge A_n) \Rightarrow (B_1 \vee \dots \vee B_m)$.
 - Proof is completed if some A_i is false or some B_j is true.
- All formulas are *unnegated*:
 - $(A_1 \wedge \neg A_2) \Rightarrow (B_1 \vee B_2) \rightsquigarrow A_1 \Rightarrow (B_1 \vee B_2 \vee A_2)$
 - $(A_1 \wedge A_2) \Rightarrow (B_1 \vee \neg B_2) \rightsquigarrow (A_1 \wedge A_2 \wedge B_2) \Rightarrow B_1$

A formula below the line may represent a “negated assumption”; a formula above the line may represent a “negated goal”:



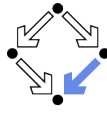
A Simple Example (Contd'3)

```

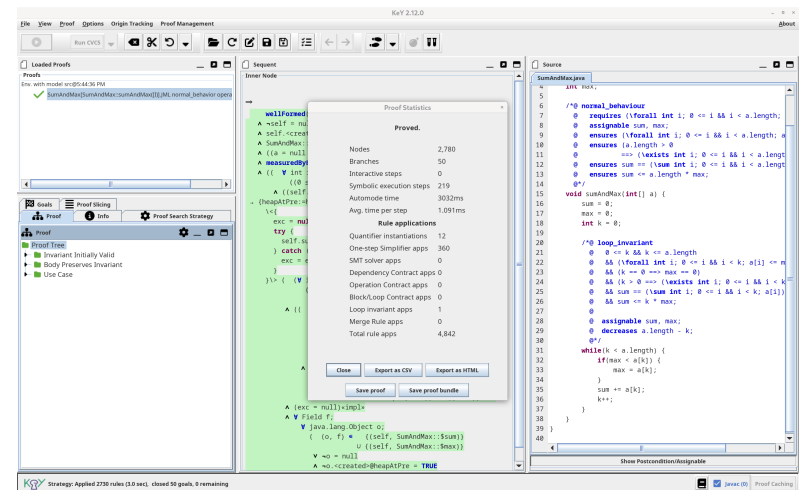
=>
wellFormed(heap)
& ...
& (( \forall forall int i;
  ((0 <= i & i < a.length) & inInt(i) -> 0 <= a[i])
  & ((self_25.<inv> & (!a = null))))))
-> {heapAtPre_0:=heap || _a:=a}
  \{
    exc_25=null; try {
      self_25.sumAndMax(_a)@SumAndMax;
    } catch (java.lang.Throwable e) { exc_25=e; }
  } \{
    ( 0 <= i & i < a.length) & inInt(i) -> a[i] <= self_25.max
    & (( a.length > 0
      -> \exists exists int i;
        (( 0 <= i & i < a.length) & inInt(i) & self_25.max = a[i]))
      & (( self_25.sum = javaCastInt(bsum(int i;)(0, a.length, a[i]))
        & (( self_25.sum <= javaMulInt(a.length, self_25.max)
          & self_25.<inv>))))))
    & (exc_25 = null)
    & \forall forall Field f;
      \forall forall java.lang.Object o;
        ( (o, f) \in { (self_25, SumAndMax::$sum) }
          \cup { (self_25, SumAndMax::$max) }
          | !o = null
            & !o.<created>@heapAtPre_0 = TRUE
              | o.f = o.f@heapAtPre_0)
  }

```

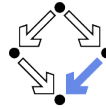
Press button “Start/stop automated proof search” (green arrow).



A Simple Example (Contd'4)



The proof runs through automatically.



Linear Search

```

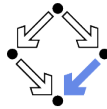
/*@ requires a != null;
 @ assignable \nothing;
 @ ensures
 @ (\result == -1 &&
 @ (\forallall int j; 0 <= j && j < a.length; a[j] != x) ||
 @ (0 <= \result && \result < a.length && a[\result] == x &&
 @ (\forallall int j; 0 <= j && j < \result; a[j] != x));
 @*/
public static int search(int[] a, int x) {
int n = a.length; int i = 0; int r = -1;
/* loop_invariant
 @ a != null && n == a.length && 0 <= i && i <= n &&
 @ (\forallall int j; 0 <= j && j < i; a[j] != x) &&
 @ (r == -1 || (r == i && i < n && a[r] == x));
 @ decreases r == -1 ? n-i : 0;
 @ assignable r, i; // required by KeY, not legal JML
 @*/
while (r == -1 && i < n) {
if (a[i] == x) r = i; else i = i+1;
}
return r;
}

```

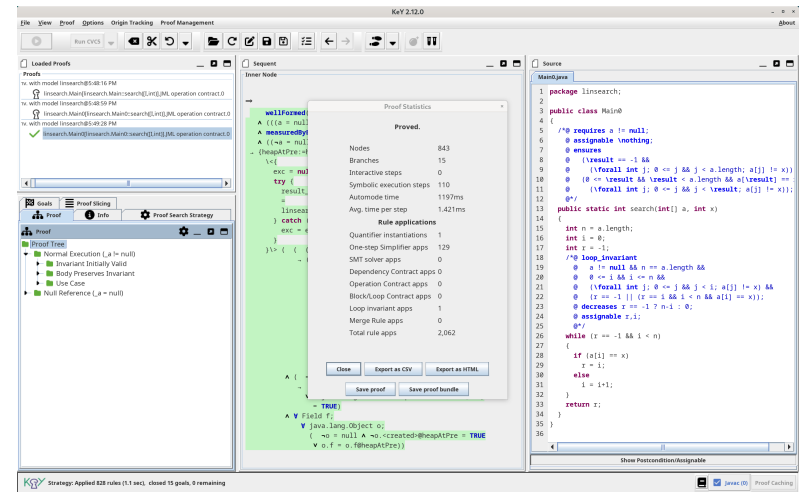
Wolfgang Schreiner

<https://www.risc.jku.at>

17/20



Linear Search (Contd)

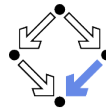


Also this verification is completed automatically.

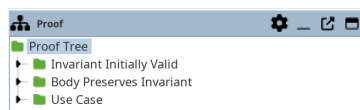
Wolfgang Schreiner

<https://www.risc.jku.at>

18/20



Proof Structure



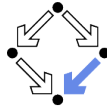
- Multiple conditions (Taclet option "javaLoopTreatment::teaching"):
 - Invariant Initially Valid.
 - Body Preserves Invariant.
 - Use Case (on loop exit, invariant implies postcondition).
- If proof fails, elaborate which part causes trouble and potentially correct program, specification, loop annotations.

For a successful proof, in general multiple iterations of automatic proof search (button "Start") and invocation of separate SMT solvers required (button "Run CVC5").

Wolfgang Schreiner

<https://www.risc.jku.at>

19/20



Summary

- Various academic approaches to verifying Java(Card) programs.
 - Jack: <http://www-sop.inria.fr/everest/soft/Jack/jack.html>
 - VeriFast: <https://github.com/verifast/>
 - Various tools for byte code verification.
- Do not yet scale to verification of full Java applications.
 - General language/program model is too complex.
 - Simplifying assumptions about program may be made.
 - Possibly only special properties may be verified.
- Nevertheless very helpful for reasoning on Java in the small.
 - Much beyond Hoare calculus on programs in toy languages.
 - Probably all examples in this course can be solved automatically by the use of the KeY prover and its integrated SMT solvers.
- Enforce clearer understanding of language features.
 - Perhaps constructs with complex reasoning are not a good idea. . .

In a not too distant future, customers might demand that some critical code is shipped with formal certificates (correctness proofs). . .

Wolfgang Schreiner

<https://www.risc.jku.at>

20/20