Formal Specification and Verification of Object-Oriented Programs

Method Contracts

TECHNISCHE UNIVERSITÄT DARMSTADT

An Example

TECHNISCHE UNIVERSITÄT DARMSTADT

```
\javaSource "src/";

\programVariables{
 Person p;
 int j;
}

\problem {
  (\forall int i;
    (!p=null ->
      ({j := i}\<{p.setAge(j);}\>(p.age = i))))
}
```

Method Calls

TECHNISCHE UNIVERSITÄT DARMSTADT

Method Call with actual parameters $arg_0, \ldots, arg_n$

$$\langle c.\mathtt{m}(arg_0, \ldots, arg_n);\rangle\phi$$

where $\mathtt{m}$ declared as $\mathbf{void}\ \mathtt{m}(\mathtt{T}_0\ \mathtt{p}_0, \ldots, \mathtt{T}_n\ \mathtt{p}_n)$

Actions of rule methodCall

1. For receiver expression c (of type T): declare & init. new local variable T `obj` $=c$;
2. For each formal parameter $\mathtt{p}_i$ of $\mathtt{m}$: declare & init. new local variable T_i `p#i` $=arg_i$;
3. Check if receiver is $\mathbf{null}$. If yes, split proof in normal and exceptional branch.
4. Look up implementation class C of $\mathtt{m}$ and split proof
   if implementation cannot be uniquely determined
   (necessitated by dynamic dispatch in general)
5. Create statically resolved method invocation $c.\mathtt{m}(\mathtt{p\#0}, \ldots, \mathtt{p\#n})@C$

(Type conformance of arg_i to $\mathtt{T}_i$ guaranteed by JAVA compiler)

Method Calls Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Method Body Expand

1. Execute code that binds actual to formal parameters $\texttt{T}_i\ \texttt{p\#i} = arg_i;$
2. Call rule methodBodyExpand

$$\frac{\Gamma \Rightarrow \langle \pi\ \texttt{method-frame(source=}C\texttt{, this=}c\texttt{)\{}body\texttt{\}}\ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \langle \pi\ c.m(p\#0, \ldots, p\#n)@C;\ \omega \rangle \phi, \Delta}$$

   2.1 Rename p_i in `body` into `p#i`
   2.2 Replace method invocation by method frame and method body

Symbolic Execution

Only static information available, proof splitting (dynamic dispatch)

Method Calls Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Method Body Expand

1. Execute code that binds actual to formal parameters $\texttt{T}_i\ \texttt{p\#i} = arg_i;$
2. Call rule methodBodyExpand

$$\frac{\Gamma \Rightarrow \langle \pi\ \texttt{method-frame(source=}C\texttt{, this=}c\texttt{)\{}body\texttt{\}}\ \omega \rangle \phi, \Delta}{\Gamma \Rightarrow \langle \pi\ c.m(p\#0, \ldots, p\#n)@C;\ \omega \rangle \phi, \Delta}$$

   2.1 Rename p_i in `body` into `p#i`
   2.2 Replace method invocation by method frame and method body

Symbolic Execution
Runtime infrastructure required in calculus (method frames)

Demo
`methods/instanceMethodInlineSimple.key` `methods/inlineDynamicDispatch.key`

Localisation of Fields and Method Implementations

TECHNISCHE UNIVERSITÄT DARMSTADT

JAVA has complex rules for localisation of fields and method implementations

- Polymorphism
- Late binding (dynamic dispatch)
- Scoping (class vs. instance)
- Visibility (private, protected, public)

Proof split into cases when implementation not statically determined

Object initialization

TECHNISCHE UNIVERSITÄT DARMSTADT

JAVA has complex rules for object initialization

- Chain of constructor calls until Object
- Implicit calls to `super()`
- Visibility issues
- Initialization sequence

Coding of initialization rules in methods `<createObject>()`, `<init>()`,...
which are then symbolically executed

Limitations of Method Inlining

TECHNISCHE UNIVERSITÄT DARMSTADT

- Source code might be unavailable
  - source code often unavailable for commercial APIs,
    even for some JAVA API methods (& implementation vendor-specific)
  - method implementation deployment-specific
- Method is invoked multiple times in a program
  - n nested methods with 2 calls each: 2^n inlined method bodies
  - avoid multiple symbolic execution of identical code
- Cannot handle unbounded recursion
- Not modular! If method implementation changes or a new overwriting method is introduced, all proofs inlining the method must be redone.

Use method contract instead of method implementation

1. Show that requires clause is satisfied
2. Obtain postcondition from ensures clause
3. Delete updates with modifiable locations from symbolic state

Method Contract Rule: Normal Behavior Case

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public normal_behavior
  @ requires preNormal;
  @ ensures postNormal;
  @ assignable modified;
  @*/ // implementation contract of m()
```

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}\mathcal{F}(\texttt{preNormal}), \Delta \quad \text{(precondition)} \qquad \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}_{\mathit{modified}}\}(\mathcal{F}(\texttt{postNormal}) \rightarrow \langle \pi\, p\, \omega \rangle \phi), \Delta \quad \text{(normal)}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_n)\, p\, \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation from JML to Java DL
- $\mathcal{V}_{\mathit{modified}}$: anonymising update

Method Contract Rule: Normal Behavior Case

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public normal_behavior
  @ requires preNormal;
  @ ensures postNormal;
  @ assignable modified;
  @*/ // implementation contract of m()
```

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}\mathcal{F}(\texttt{preNormal}), \Delta \quad \text{(precondition)} \qquad \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}_{\mathit{modified}}\}(\mathcal{F}(\texttt{postNormal}) \rightarrow \langle \pi\, p\, \omega \rangle \phi), \Delta \quad \text{(normal)}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_n)\, p\, \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation from JML to Java DL
- $\mathcal{V}_{\mathit{modified}}$: anonymising update

Keeping the Context

TECHNISCHE UNIVERSITÄT DARMSTADT

- Want to keep part of prestate $\mathcal{U}$ that is unmodified by called method
- **assignable** clause of contract tells what can possibly be modified

```
@ assignable modified;
```

- How to erase all values of **assignable** locations in state $\mathcal{U}$?

  Analogous situation: $\forall$-Right quantifier rule $\Rightarrow \forall x; \phi$
  Replace x with a fresh constant c

  To change value of program location use update
- Anonymising updates $\mathcal{V}$ erase information about modified locations

$\mathcal{V}_{modified} = \{l_1 := c_1 \,||\, \cdots \,||\, l_n := c_n\}$
(c_i new constant symbol for each location in `modified`)

Anonymising Heap Locations

TECHNISCHE UNIVERSITÄT DARMSTADT

- How to erase values changed on the heap without throwing all away?

```
@ assignable o.a, this.*;
```

Define anonymising function `anon`:Heap $\times$ LocSet $\times$ Heap $\rightarrow$ Heap

The resulting heap `anon`(...) coincides with the first heap on all locations except for those specified in the location set. Those locations attain the value specified by the second heap.

The value of

$$\texttt{select}(\texttt{anon}(h1, locs, h2), o, f) = \begin{cases} \texttt{select}(h2, o, f) \\ \quad \text{if } \big((o, f) \in locs \ \wedge f \neq \texttt{<created>}\big) \vee \\ \qquad\qquad (o, f) \in freshLocs(h1) \\ \texttt{select}(h1, o, f) \quad \text{otherwise} \end{cases}$$

Anonymising Heap Locations: Example

TECHNISCHE UNIVERSITÄT DARMSTADT

```
@ assignable o.a, this.*;
```

To erase all knowledge about the values of the locations of the assignable expression:

- introduce a new (not yet used) constant of type Heap, e.g., $\texttt{heap}_c$
- anonymise the current heap

$$\texttt{anon}(\texttt{heap}, \{(\texttt{o}, \texttt{a})\} \cup \texttt{allFields}(\texttt{this}), \texttt{heap}_c)$$

- assign the current heap the new value

$$\mathcal{V}_{\mathit{modified}} = \{\texttt{heap} := \texttt{anon}(\texttt{heap}, \{(o, a)\} \cup \texttt{allFields}(\texttt{this}), \texttt{heap}_c)\}$$

Method Contract Rule: Exceptional Behavior Case

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public exceptional_behavior
  @ requires preExc;
  @ signals (Exception exc) postExc;
  @ assignable modified;
  @*/
```

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}\mathcal{F}(\texttt{preExc}), \Delta \quad \text{(precondition)} \qquad \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}_{\mathit{modified}}\}((\mathit{exc} \doteq \neq \mathbf{null} \wedge \mathcal{F}(\texttt{postExc})) \rightarrow \langle \pi\ \mathbf{throw}\ \texttt{exc;}\ \texttt{p}\ \omega\rangle\phi), \Delta \quad \text{(exceptional)}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\ \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_n)\ p\ \omega\rangle\phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation from JML to Java DL
- $\mathcal{V}_{\mathit{modified}}$: anonymising update

Method Contract Rule: Combined

TECHNISCHE UNIVERSITÄT DARMSTADT

KeY actually uses only one rule for both specification cases

Therefore, translation of postcondition ϕ_{post} as follows (simplified):

$$((\texttt{exc} \doteq \textbf{null} \land \mathcal{F}(\texttt{\textbackslash old(preNormal)}) \rightarrow \mathcal{F}(\textit{postNormal})) \land$$
$$((\texttt{exc} \neq \textbf{null} \land \mathcal{F}(\texttt{\textbackslash old(preExc)})) \rightarrow \mathcal{F}(\textit{postExc}))$$

$$\frac{\begin{array}{l}\Gamma \Rightarrow \{\mathcal{U}\}(\mathcal{F}(\texttt{preNormal}) \lor \mathcal{F}(\texttt{preExc})), \Delta \quad \text{(precondition)} \\ \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}_{\textit{mod}_{\textit{normal}}}\}((\texttt{exc} \doteq \textbf{null} \land \phi_{post}) \rightarrow \langle \pi\ \texttt{p}\ \omega \rangle \phi), \Delta \quad \text{(normal)} \\ \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}_{\textit{mod}_{\textit{exc}}}\}((\texttt{exc} \neq \textbf{null} \land \phi_{post}) \\ \qquad\qquad \rightarrow \langle \pi\ \textbf{throw}\ \texttt{exc;}\ \texttt{p}\ \omega \rangle \phi), \Delta \quad \text{(exceptional)}\end{array}}{\Gamma \Rightarrow \mathcal{U}\langle \pi\ \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_n)\ p\ \omega \rangle \phi, \Delta}$$

- $\mathcal{F}(\cdot)$: translation of JML to Java DL
- $\mathcal{V}_{\textit{mod}_{\textit{normal / exc}}}$: anonymising updates

Method Contract Rule: Example

TECHNISCHE UNIVERSITÄT DARMSTADT

```
class Person2 {
 private /*@ spec_public @*/ int age;
 /*@ public normal_behavior
   @ requires age < 29;
   @ ensures age == \old(age) + 1;
   @ assignable age;
   @ also
   @ public exceptional_behavior
   @ requires age >= 29;
   @ signals (ForeverYoungException exc) age == \old(age);
   @ assignable \nothing;
   @*/
 public void birthday() throws ForeverYoungException {
   if (age >= 29) throw new ForeverYoungException();
   age++;
 } }
```

Method Contract Rule: Example Cont'd

TECHNISCHE UNIVERSITÄT DARMSTADT

Demo

`methods/useContractForBirthday2.key`

- Proof without contracts (all except object creation)
  - Method treatment: Expand
- Proof with contracts (until method contract application)
  - Method treatment: Contract
- Proof contracts used
  - Method treatment: Expand
  - Select contracts for `birthday()` in `src/Person2.java`
  - Prove "EnsuresPost"

Correctness of Method Contract Rule

TECHNISCHE UNIVERSITÄT DARMSTADT

Normal Behavior Contract

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}(\mathcal{F}(\texttt{preNormal}) \land \mathit{freePre}), \Delta \quad \text{(precondition)} \qquad \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}_{\mathit{modified}}\}(\mathcal{F}(\texttt{postNormal}) \rightarrow \langle \pi\ \texttt{p}\ \omega\rangle\phi), \Delta \quad \text{(normal)}}{\Gamma \Rightarrow \mathcal{U}\langle \pi\ \texttt{result} = \texttt{m}(\texttt{a}_1, \ldots, \texttt{a}_n)\ p\ \omega\rangle\phi, \Delta}$$

- Depends on correctness of used contract
- Correctness of a (normal behaviour) JML contract shown by proving (see previous lecture)

$$\mathcal{F}(\mathit{pre}) \land \mathit{freepre} \rightarrow \left\langle \begin{array}{l} \texttt{exc} = \textbf{null}; \\ \textbf{try}\{\texttt{o.m()};\}\textbf{catch}(\texttt{Throwable e})\{\texttt{exc} = \texttt{e};\} \end{array} \right\rangle (\texttt{exc} \doteq \textbf{null} \land \mathcal{F}(\mathit{post}) \land \mathit{frame})$$

(Demo: `methods/src/Person2.java`)

Recursion

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public normal_behavior
  @ requires n>=0;
  @ ensures \result >= n;
  @ assignable \nothing;
  @*/
public static int fib(int n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  }
  return fib(n-1) + fib(n-2);
}
```

```
/*@ public normal_behavior
  @ requires n>=0;
  @ ensures \result == n*(n + 1);
  @ assignable \nothing;
  @*/
  public static int gauss(int n) {
        if (n == 0) {
            return 0;
        }
        return n + gauss(n-1);
  }
}
```

Recursive Methods—Problems

TECHNISCHE UNIVERSITÄT DARMSTADT

How do we prove that a recursive method satisfies its contract?

Problem: Inlining

Inlining not possible in general

- Recursion depth does not have a fixed bound, i.e., depends on state
  $\Rightarrow$ infinite proof tree
- Even if a fixed bound exists:
  inefficient even for a relative small fixed upper bound

Problem: Contracts

- Cyclic dependency:
  Using same contract we are trying to prove correct

Recursive Methods—Induction to the Rescue

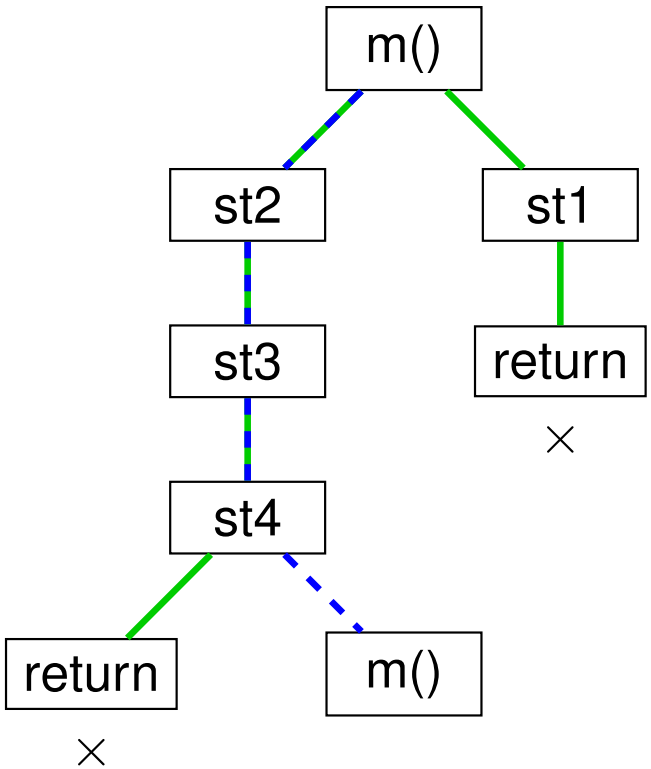
TECHNISCHE UNIVERSITÄT DARMSTADT

Idea: Use Induction and Contracts

- Base Case:
  Prove contract C_m of a recursive method m correct for recursion depth 0
- Induction Hypothesis:
  C_m correct for induction depth of n
- Induction Step:
  - Assume C_m correct for recursion depth of n
  - Prove C_m correct for recursion depth of $n+1$

Recursive Methods—Induction to the Rescue

TECHNISCHE UNIVERSITÄT DARMSTADT



- Paths with no recursive call
- Paths with recursion

Recursive Methods—Induction to the Rescue

TECHNISCHE UNIVERSITÄT DARMSTADT

Observation: Branches of

- symbolic paths without recursive call relate to base case
- symbolic paths with recursive call relate to step case

Does that mean we are done? No changes for recursive methods needed?

Yes and No

Partial Correctness: Yes, we are done!

Partial correctness is only concerned with terminating cases. Diverging execution paths of the method satisfies the contract trivially.

Total Correctness: No, not yet.

For total correctness, we still have to prove that our recursion eventually terminates.

Recursive Methods—Ensuring Termination

TECHNISCHE UNIVERSITÄT DARMSTADT

How to ensure termination of recursive methods?

Idea

Define an integer typed term (a variant), and prove that at time of recursive call

- its value is non-negative (i.e., the measure has a lower bound)
- and strictly smaller than in the pre-state of the invoking method

If that can be proven, termination is guaranteed!

Recursive Methods—Defining a Measure in JML

TECHNISCHE UNIVERSITÄT DARMSTADT

```
/*@ public normal_behavior
  @ requires n >= 0;
  @ ensures \result >= 0;
  @ measured_by n;
  @ assignable \nothing;
  @*/
```

Meaured_By Clause

Specifies an integer typed JML expression that has to be proven non-negative and strictly decreasing at invocation time.

Recursive Methods—Translation of measured_by to DL

TECHNISCHE UNIVERSITÄT DARMSTADT

Functional DL contract F for a method m

$$F = (pre, post, div, var, mod)$$

with

- a precondition DL formula *pre* ✓ ,
- a postcondition DL formula *post* ✓ ,
- a divergence indicator $div \in \{TOTAL, PARTIAL\}$ ✓ ,
- a variant *var* a term of type `any` ,
- a modifies set *mod* is either a term of type `LocSet` or `\strictly_nothing` ✓

The variant term is defined as

$$var = \mathcal{E}(\mathbf{measured\_by})$$

Recursive Methods—Completing PO Generation & Method Contract Rule

TECHNISCHE UNIVERSITÄT DARMSTADT

Generation of Proof Obligations

$$(\mathit{freePre} \land \mathit{pre} \land \texttt{measuredBy}(\mathit{var}_{\texttt{m}})) \rightarrow$$
$$\{\texttt{heapAtPre} := \texttt{heap}\}\langle \texttt{exc=null; try \{ self.m(args)\} catch ...}\rangle$$
$$(\mathit{post} \ \& \ \mathit{frame})$$

- `measuredBy` is a predicate (captures value of the variant in the pre-state)

Recursive Methods—Completing PO Generation & Method Contract Rule

TECHNISCHE UNIVERSITÄT DARMSTADT

Method Contract Rule (here only normal behavior for others analog)

$$\frac{\Gamma \Rightarrow \{\mathcal{U}\}(\mathit{pre} \wedge \texttt{measuredByCheck}(\mathit{var}_{\mathtt{m}})), \Delta \quad \text{(precondition)} \qquad \Gamma \Rightarrow \{\mathcal{U}\}\{\mathcal{V}_{\mathit{modified}}\}\mathit{post} \rightarrow \langle \pi\, p\, \omega \rangle \phi), \Delta \quad \text{(normal)}}{\Gamma \Rightarrow \{\mathcal{U}\}\langle \pi\, \texttt{result} = \mathtt{m}(\mathtt{a}_1, \ldots, \mathtt{a}_n)\; p\; \omega \rangle \phi, \Delta}$$

The predefined predicate `measuredByCheck` is true iff.

- its argument is strictly greater than 0 and
- its argument is strictly less than an expression for which `measuredBy` is true (intuitively, if its argument is strictly smaller than at invocation time of the calling method)

measuredByCheck

$$\frac{\Gamma, \texttt{measuredBy}(s) \Rightarrow t \geq 0 \wedge t < s, \Delta}{\Gamma, \texttt{measuredBy}(s) \Rightarrow \texttt{measuredByCheck}(t)), \Delta}$$

Understanding Unclosed Proofs

TECHNISCHE UNIVERSITÄT DARMSTADT

Reasons why a proof may not close

- Buggy or incomplete specification
- Bug in program
- Maximal number of steps reached: restart or increase # of steps
- Automatic proof search fails: manual rule applications necessary

Understanding Unclosed Proofs

TECHNISCHE UNIVERSITÄT DARMSTADT

Understanding open proof goals

- Follow the control flow from the proof root to the open goal
- Branch labels give useful hints
- Identify unprovable part of post condition or invariant
- Sequent remains always in “pre-state”: Constraints on program variables refer to value at start of program (exception: formula is behind update or modality)
- NB: $\Gamma \Rightarrow o \doteq \mathbf{null}, \Delta$ is equivalent to $\Gamma, o \not\doteq \mathbf{null} \Rightarrow \Delta$

Formal Methods: Ready For Prime Time?

TECHNISCHE UNIVERSITÄT DARMSTADT

Functional Verification of Executable Source Code

- Doable: JavaCard API reference impl. verification, VeriSoft, TimSort etc.
- BUT
  - Expertise in formal methods required (verification engineer)
  - Not “push-button”
  - Writing provable specs is a huge bottleneck
  - Time and cost expensive (only for secure or safety-critical systems)

On the other hand . . .

Lightweight Formal Methods Work Well

- Automatic test case generation from proof attempts
- Fixed application domain and specs (MS Static Driver Verifier)
- Efficient omniscient visual state debugging (Visual State Debugger)

The Visual State Debugger (Topic of Guest Lecture)

TECHNISCHE UNIVERSITÄT DARMSTADT

New kind of debugger based on the following technologies:

Symbolic Execution

- Debugging can start at any statement of interest
  (no need to build a concrete initial state)
- Generation of a symbolic execution tree representing precisely all concrete execution paths until finite depth
- Efficient omniscient (reverse) debugging
- Semantic watchpoints (“free floating conditional breakpoints”)
  for advanced execution control
- Symbolic preconditions prune unreachable states

The Visual State Debugger (Topic of Guest Lecture)

TECHNISCHE UNIVERSITÄT DARMSTADT

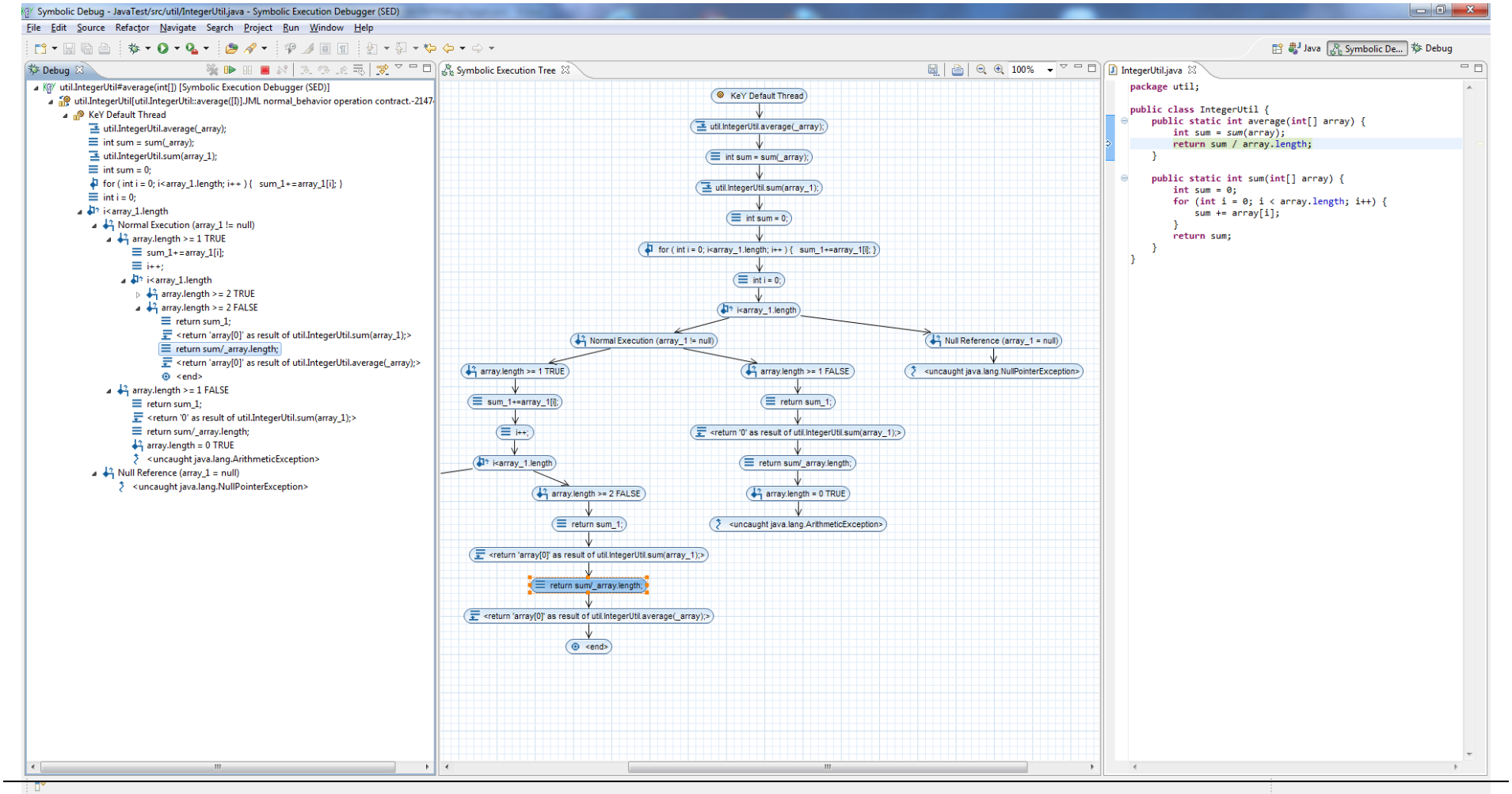
New kind of debugger based on the following technologies:

Symbolic execution based on KeY's verification engine

- Open branches correspond to symbolic program execution paths
- Closed branches indicate infeasible program paths

The Visual State Debugger: Screenshot

TECHNISCHE UNIVERSITÄT DARMSTADT



VSDB: State Visualization

TECHNISCHE UNIVERSITÄT DARMSTADT

Visual State Inspection

Program comprehension requires understanding intermediate states

- Succinct representation of datastructures by symbolic object diagrams
- Visualisation of all possible symbolic heap configurations
- Assistance in spotting unintended (possibly malformed) configurations

Summary

TECHNISCHE UNIVERSITÄT DARMSTADT

- Most features of sequential JAVA covered in KeY
- Most remaining features available in experimental version
  - Simplified multi-threaded JMM
  - Floats
- Degree of automation for loop-free programs is very high
- Method contracts can prevent combinatorial explosion
- Model search can give hints on counter examples
- Lightweight formal methods derivable from deductive verification

Left to do: recursive methods & verification of programs with loops—loop invariants

Literature for this Lecture

TECHNISCHE UNIVERSITÄT DARMSTADT

Essential

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 10: Using KeY

KeY Book Verification of Object-Oriented Software (see course web page), Chapter 3: Dynamic Logic, Sections 3.1, 3.2, 3.4, 3.6.5, 3.6.6

About the VSDB

ASE 2010 A Visual Interactive Debugger Based on Symbolic Execution: R. Hähnle, Markus Baum, Richard Bubel, Marcel Rothe. Proc. 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, 143–146, ACM Press, 2010

https://www.se.tu-darmstadt.de/se/group-members/martin-hentschel/