THE OHIO STATE UNIVERSITY

Formal Verification of a Java Component Using the RESOLVE Framework

Laine Rumreich and Dr. Paul Sivilotti

Overview

- Unique combination of a Java component with RESOLVE specifications for full formal verification
  - Practicality of an industry-standard programming language
  - Robust full-functional verification possible in RESOLVE

Results

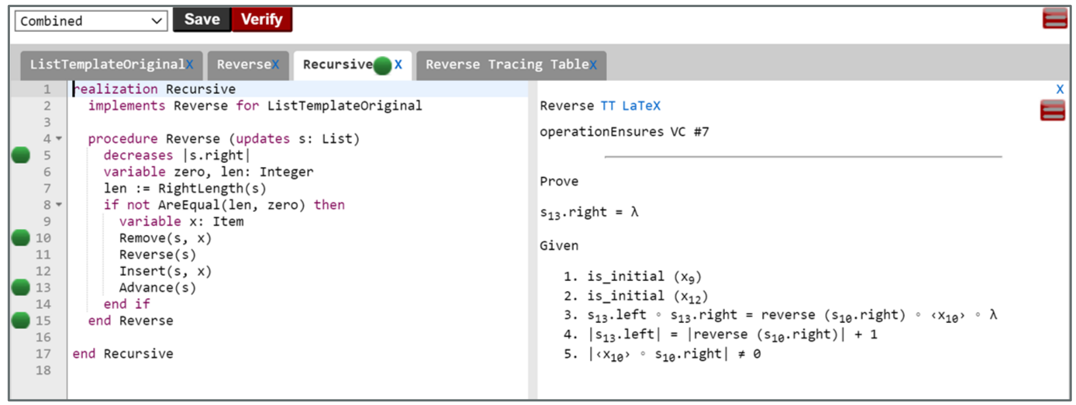
1. Example of the feasibility of combining Java and RESOLVE, a verification discipline that uses value semantics
2. Correctness proof for a Java-based Binary Decision Diagram (BDD) implementation
3. Correction of errors not revealed by an extensive test suite

Ongoing and Future Work

- Develop an automated theorem prover for a Java-based component with RESOLVE specifications
- Existing RESOLVE verifiers could be leveraged with only slight modifications to discharge many VCs in an automated way

RESOLVE

- Design discipline for software that allows for formal verification
- Uses clean, value-based semantics to ease client-side reasoning



Screenshot of the RESOLVE Verifier Web-IDE

- Defines a mathematical model as an abstract definition for client reasoning about the component
- Disallows aliasing by removing the assignment operator and replacing it with *swapping*

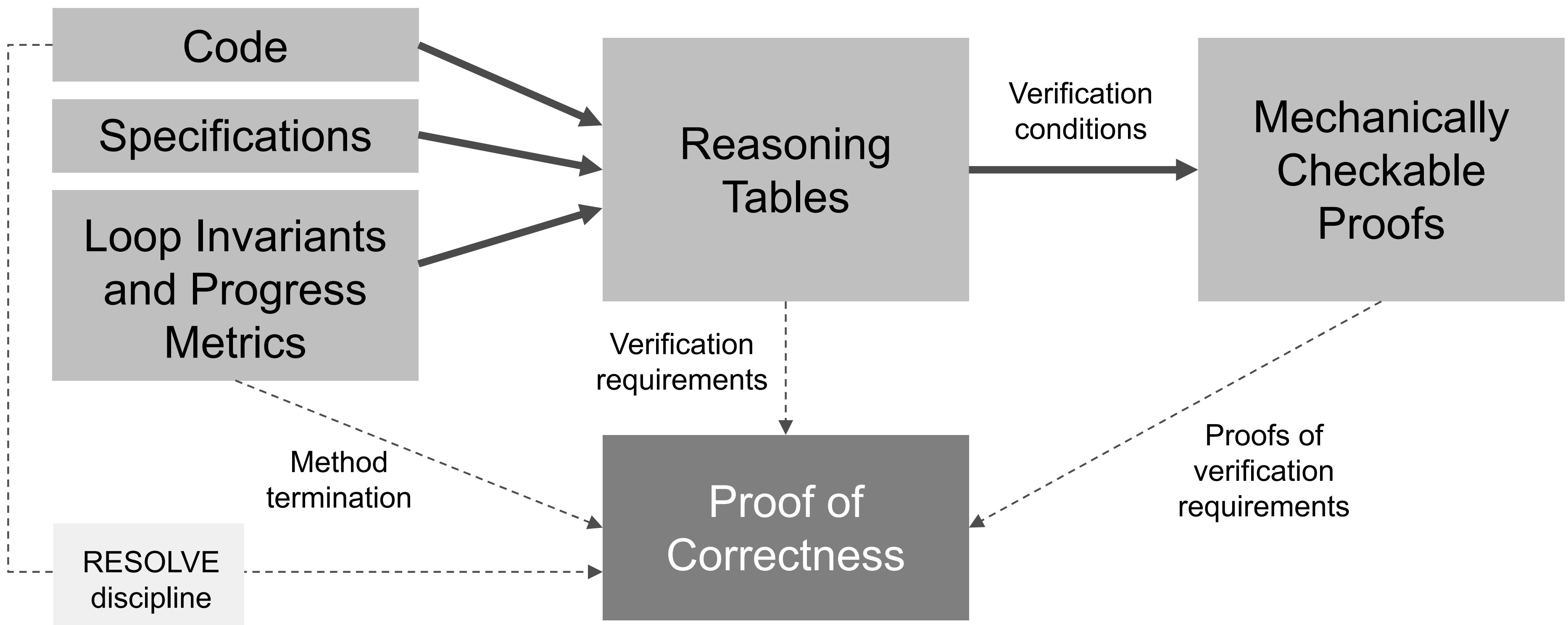
Challenges of Java Verification

- Aliasing and References
  - Assignment operator
  - Argument passing with repeated arguments allowed
- Presence of inheritance
  - Allows differing mathematical models for implementing classes

A Disciplined Approach to Java

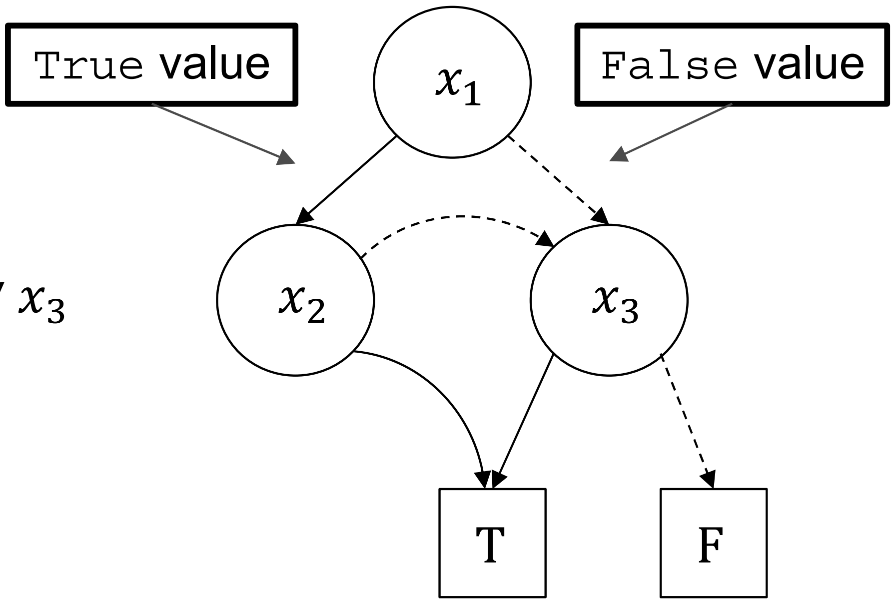
- Alias Control
  - Replace assignment with transferFrom method
  - Respect ownership of advertised aliases
- Disciplined use of inheritance
  - Requiring the same mathematical model for all implementing classes
  - Separating client and implementer states
  - Separating methods into *kernel* and *secondary*

Correctness Proof



The Binary Decision Diagram

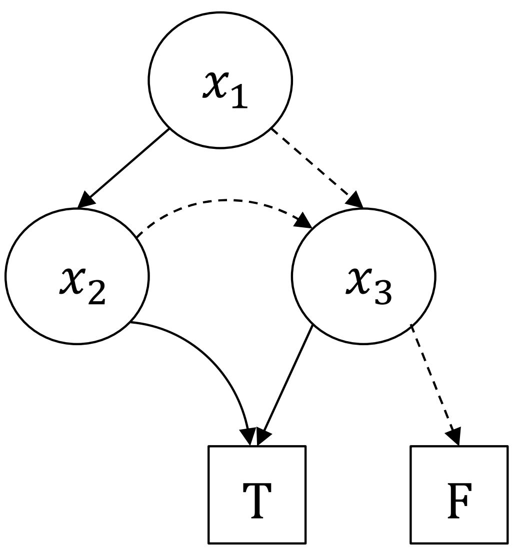
$(x_1 \land x_2) \lor x_3$



x_1	x_2	x_3	$(x_1 \land x_2) \lor x_3$
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	T
F	T	F	F
F	F	T	T
F	F	F	F

BooleanStructure Math Model

$(x_1 \wedge x_2) \vee x_3$




```
ASSIGNMENT is finite set of integer

BOOLEAN_STRUCTURE is
  (sat: finite set of ASSIGNMENT, vars: string of integer)
    exemplar exp
    constraint
      for all a: ASSIGNMENT where ( a in exp.sat )
        ( a is subset of entries(exp.vars) ) and
      | exp.vars | = | entries(exp.vars) |
```

No repeated variables

```
sat = { {3}, {1, 2}, {1, 3},
              {2, 3}, {1, 2, 3} }
vars = <1, 2, 3>
```

Verified Concrete Component

```
ASSIGNMENT is finite set of integer

BOOLEAN_STRUCTURE is
  (sat: finite set of ASSIGNMENT, vars: string of integer)
    exemplar exp
    constraint
      for all a: ASSIGNMENT where ( a in exp.sat )
        ( a is subset of entries(exp.vars) ) and
      | exp.vars | = | entries(exp.vars) |

@convention
     NO_EXTRANEOUS_VARIABLES($this.sat, $this.vars) and
     NO_DUPLICATES_IN_VARS($this.vars)

@correspondence this = ($this.sat, $this.vars)
```

BooleanStructure Math Model

BooleanStructure Convention and Correspondence

Loop Invariants

```
/**
 * @updates newSat, ~allAssignments
 *
 * @maintains entries(~allAssignments.seen) \ $this.sat = newSat
 *
 * @decreases |~allAssignments.unseen|
 */
for (Set<Integer> a : allAssignments) {
    // a = allAssignments.next()

    // Add assignment to new sat if it isn't in the original one
    if (!(processAssignment(this.sat, this.vars, a))) {
        newSat.add(a);
    }
}
```

~allAssignments.seen * ~allAssignments.unseen = allAssignments

Required for reasoning tables

Reasoning Tables: copyFrom

Secondary Layered Method

State	Path	Facts	Obligations
`public void copyFrom(BooleanStructure x) {`			
0		for all a: ASSIGNMENT where (a in this.sat) (a is subset of entries(this.vars)) $\lvert$ this.vars $\rvert$ = $\lvert$ entries(this.vars) $\rvert$	
`BooleanStructure newExp = this.newInstance();`			
1		$newExp.sat_1$ = { {} } $newExp.vars_1$ = <>	$newOrder_6$ = ~$order.seen_6$
`/** * @updates newOrder, ~order * * @maintains newOrder = ~order.seen * * @decreases |~order.unseen| */ for (int v : order) {`			
7	$\lvert$~$order.unseen_7 \rvert > 0$	$newOrder_7$ = ~$order.seen_7$	$\lvert$~$order.unseen_7 \rvert > 0$
`// v = order.next()`			
8		~$order.seen_8$ = ~$order.seen_7$ * <v> <v> * ~$order.unseen_8$ = ~$order.unseen_7$	
`newOrder.add(newOrder.length(), v);`			
9		$newOrder_9 = newOrder_7$ * <v>	$newOrder_9$ = ~$order.seen_8$ $\lvert$~$order.unseen_8 \rvert < \lvert$~$order.unseen_7 \rvert$
`} // end for`			
10		$newOrder_{10}$ = ~$order.seen_{10}$ $\lvert$~$order.unseen_{10} \rvert = 0$	entries($newExp.vars_4$) = entries($newOrder_{10}$) $\lvert$ $newExp.vars_4$ $\rvert$ = $\lvert$ entries($newOrder_{10}$) $\rvert$
`this.transferFrom(newExp);`			
13		this.vars = $newExp.vars_{12}$ this.sat = $newExp.sat_{12}$	this.vars = x.vars this.sat = x.sat
`} // end copyFrom`			

of code

nstraint

nsures

Proofs

- Mechanically checkable proofs for each Verification Condition from Reasoning Tables

Correction of Errors

- Incorrect Specification
- Incorrect Implementation
  - Errors are despite a rigorous test suite
    - 314 unit test cases
    - 96.3% code coverage
- Design Pattern Limitation

Error in Specification

```
/**
 * @mathdefinitions
 * EQUIVALENT(
 *    m: BOOLEAN_STRUCTURE,
 *    n: BOOLEAN_STRUCTURE,
 *   ): boolean is
 * for all p: ASSIGNMENT where
 *    ( p is subset of (entries(m.vars) union entries(n.vars)) )
 *       ( EVALUATION(m, p) iff EVALUATION(n, p) )
 */
public interface BooleanStructure extends BooleanStructureKernel {
```

```
VARIABLES(this) = VARIABLES(other) and
this.sat = other.sat
```

↓

```
isEquivalent = EQUIVALENT(this, other)
```

Error in Specification

```
/**
 * …
 * @requires |this.vars| < 64
 */
public String toStringTT() {
    Sequence<Integer> thisOrder = this.vars();
    …
    long variableMask = 1 << thisOrder.length() - 1;
    …
}
```

Overflow occurs if |`vars`| >= 64 *

* 1 (64 bits) left bit shifted by 63 is a very large negative number in two's complement

Error in Implementation

```
public void copyFrom(BooleanStructure other) {
  BooleanStructure newExp = this.newInstance();
  …
  PowerStringElements allAssignments = new PowerStringElements(this.vars());
  …
  for (Set<Integer> t : allAssignments) {
    if (other.evaluate(t)) {
          …
          newExp.disj(term);
     }
  }
  …
  newExp.reorder(newOrder);
}
```

Power set of the domain

hen
ty

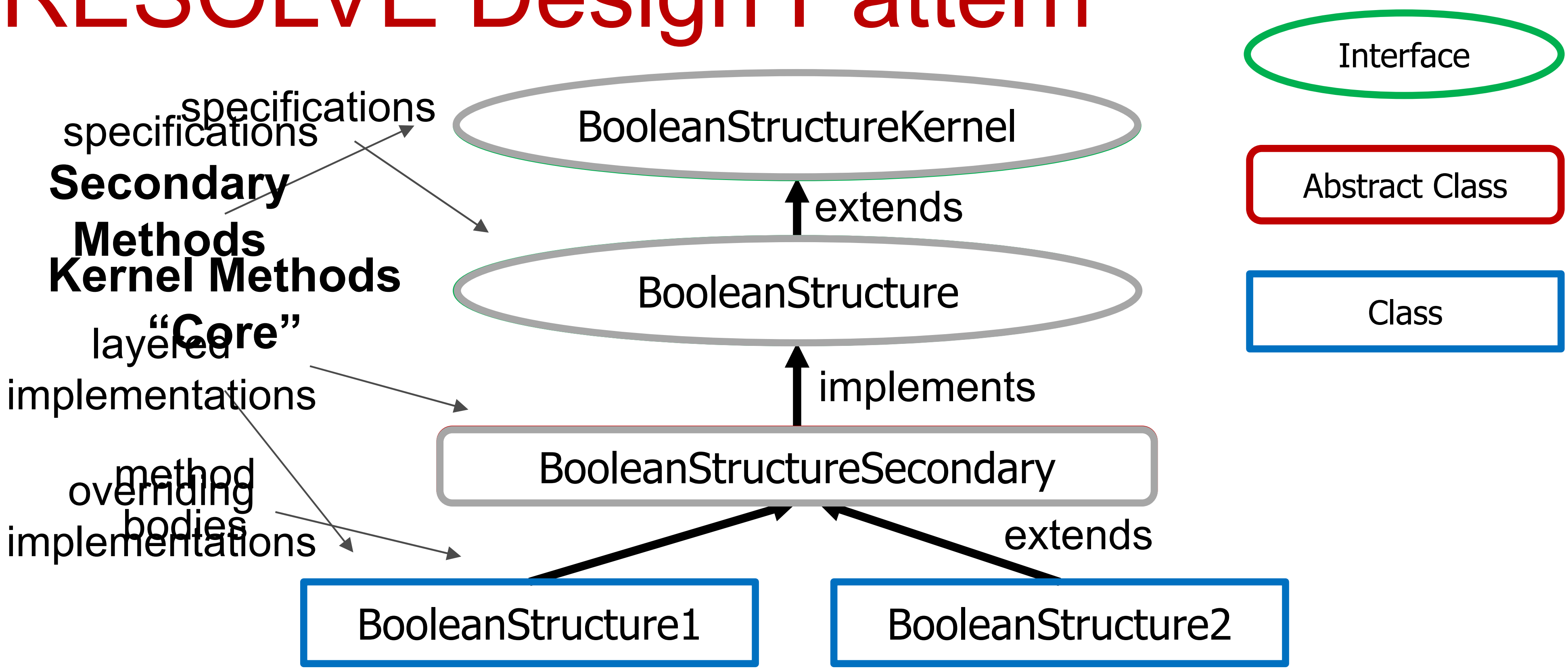
Enter the conditional if t is a satisfying assignment

s skipped when
pty

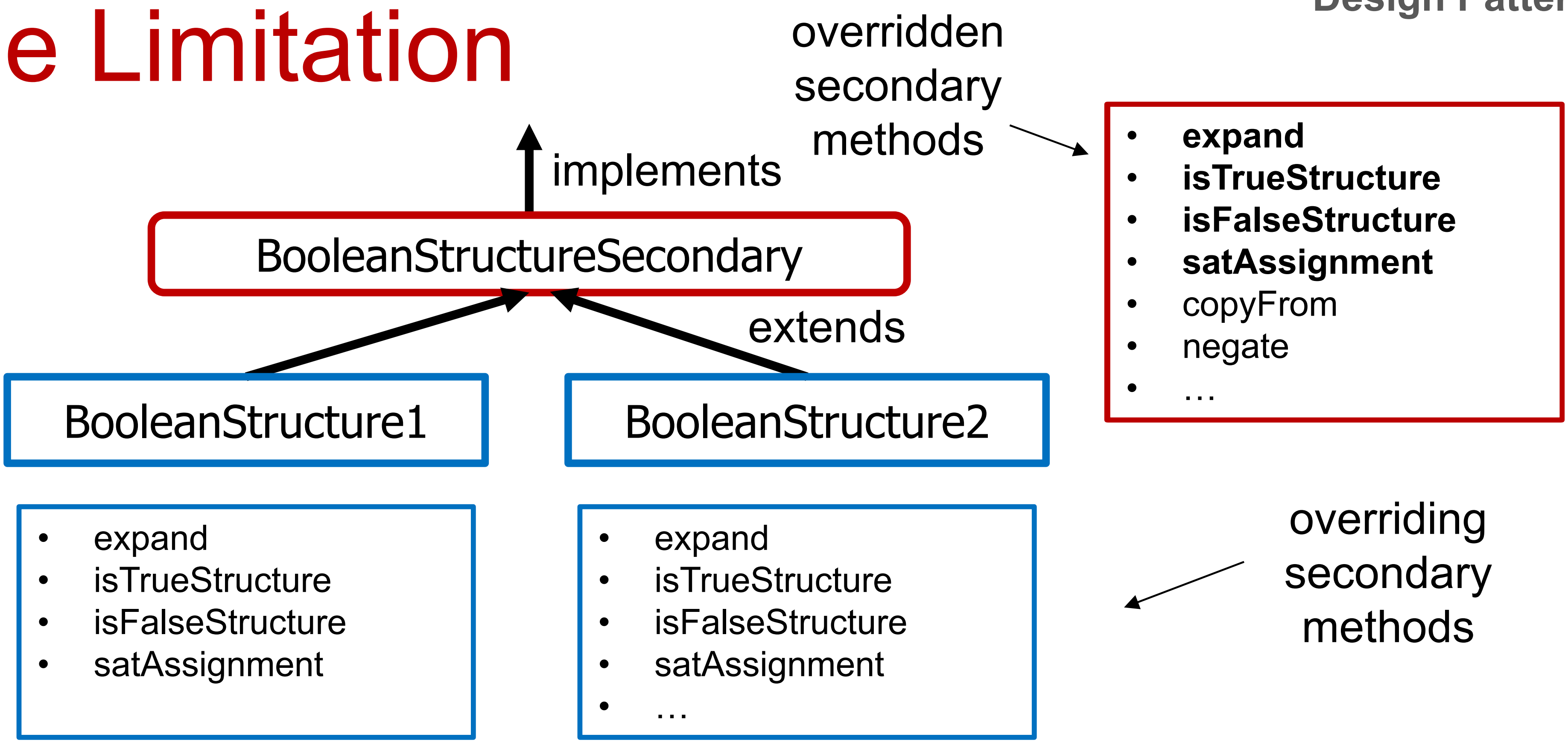
Order the variables to match the copied structure

Re
bec
/=
or
xp)

RESOLVE Design Pattern



The Limitation



The Limitation

- Limitation was present in Java component software used by thousands of students over many years
- Limitation corrected by adding a reference class that does not override any secondary methods

Conclusions

- Formal verification of a Java-based BDD implementation
- Groundwork for an automated verifier for a Java component with RESOLVE specifications
- Discoveries related to combining an industry-standard programming language and a specification notation designed with formal verification and client reasoning as the priority