http://lara.epfl.ch

Proofs and Counterexamples for Java Programs

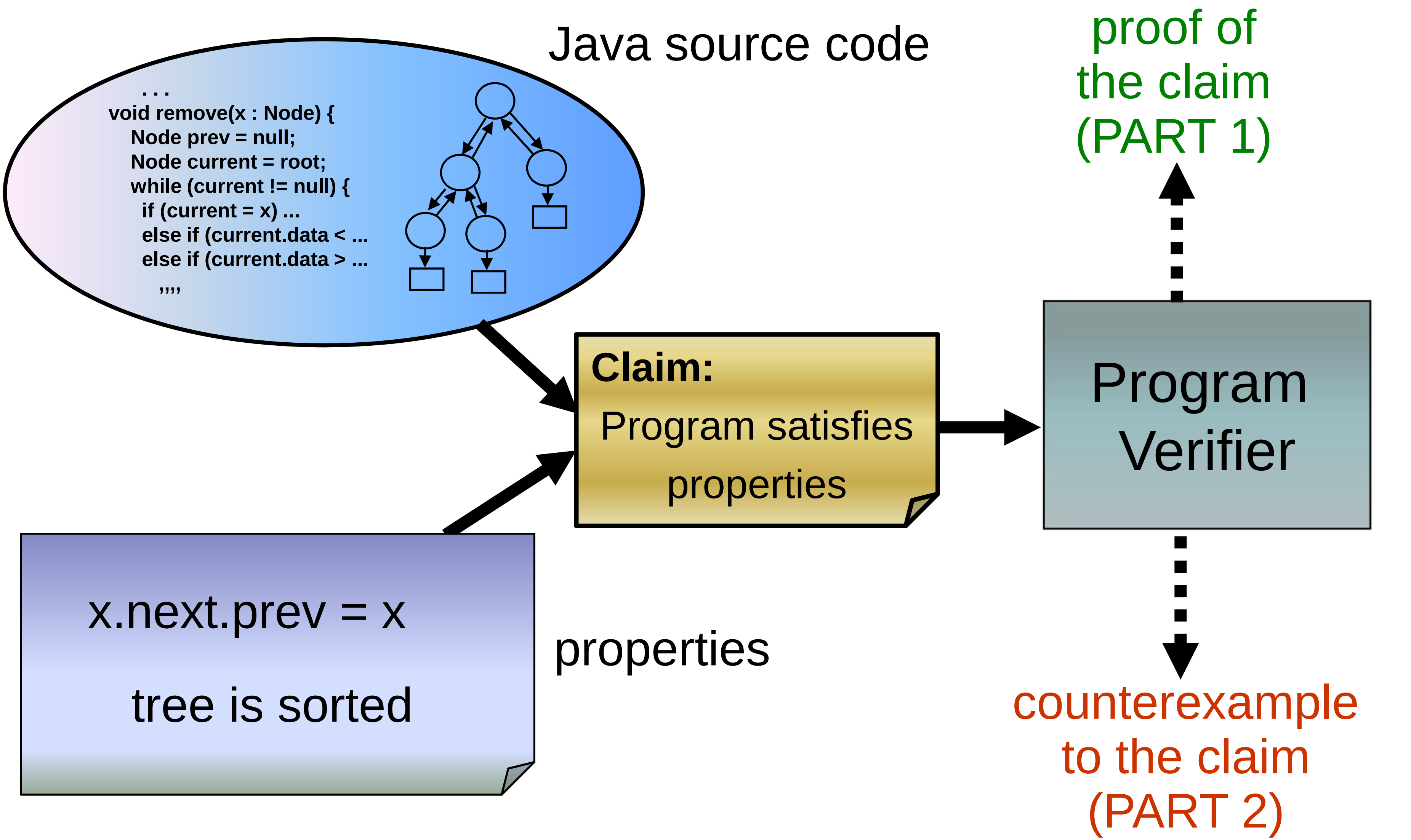
Viktor Kuncak

Laboratory for Automated Reasoning and Analysis

School of Computer and Communication Sciences

École Polytechnique Fédérale de Lausanne

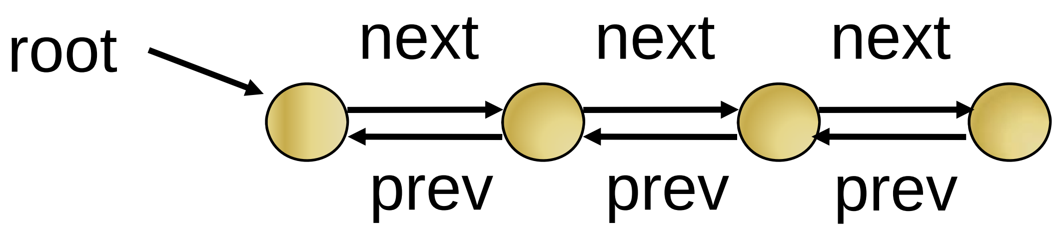
Proofs and Counterexamples for Java



PART 1: Proofs

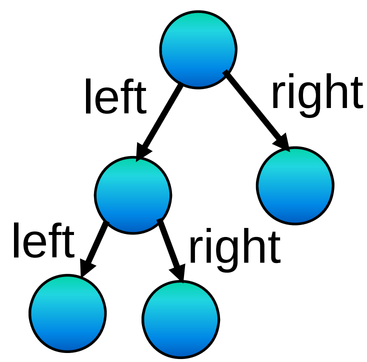
Jahob Verifier and Case Studies

Data Structures and Their Properties



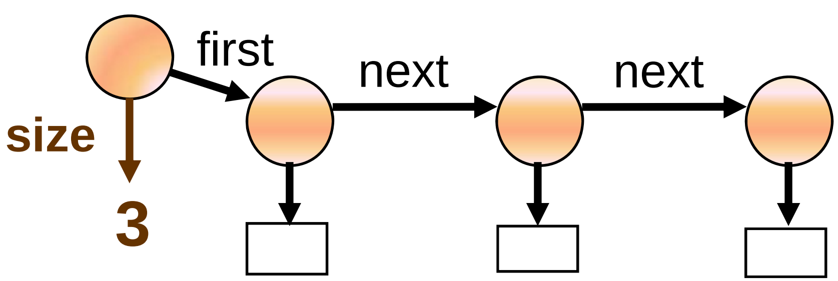
x.next.prev == x

acyclicity: $\sim next^{+}(x,x)$



graph is a tree

elements are sorted



value of size field equals the number of stored objects

size = $|\{data(n).\ (root,n) \in next^{*}\}|$

Data Structure Verification using Jahob

Verified correctness of

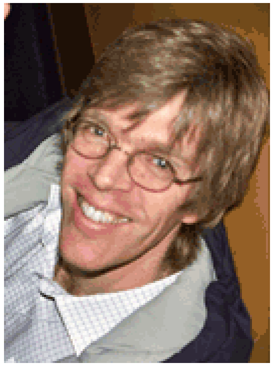
- hash table
- association list
- array-based list
- lists with cursor elements
- priority heap

More information on our site

http://JavaVerification.org

Joint work with:

Karen Zee



Martin Rinard

MIT

Full Functional Verification of Linked Data Structures

ACM Conf. ***P****rog.* ***L****anguage* ***D****esign and* ***I****mplementation*'08

An Integrated Proof Language for Imperative Programs

ACM Conf. ***P****rog.* ***L****anguage* ***D****esign and* ***I****mplementation*'09

Statically Enforcing Contracts of Widely Used Libraries

ArrayList documentation from http://java.sun.com :

Method Summary	
void	**add**(int index, Object element) **Inserts the specified element at the specified position in this list.**
boolean	**add**(Object element) **Appends the specified element to the end of this list.**

ArrayList verified contract from http://JavaVerification.org :

void	**add**(int index, Object element) **content = {(i,e). (i,e) : old content $\wedge$ i < index} $\cup$ {(index,element)}** **$\cup$ {(i,e). (i-1,e) : old content $\wedge$ index < i}**
boolean	**add**(Object element) **content = old content $\cup$ {(size,element)}**

Jahob Verifier for Java

Specifications written in subset of Isabelle/HOL

- ghost and ‘dependent’ specification variables of HOL type (sets, relations, functions)

Jahob proves

– data structure preconditions, postconditions

– data structure invariants

– absence of run-time errors

We can make verification easier through:

1. decison procedures for expressive logics
2. manual proof decomposition techniques
3. techniques that combine decision procedures

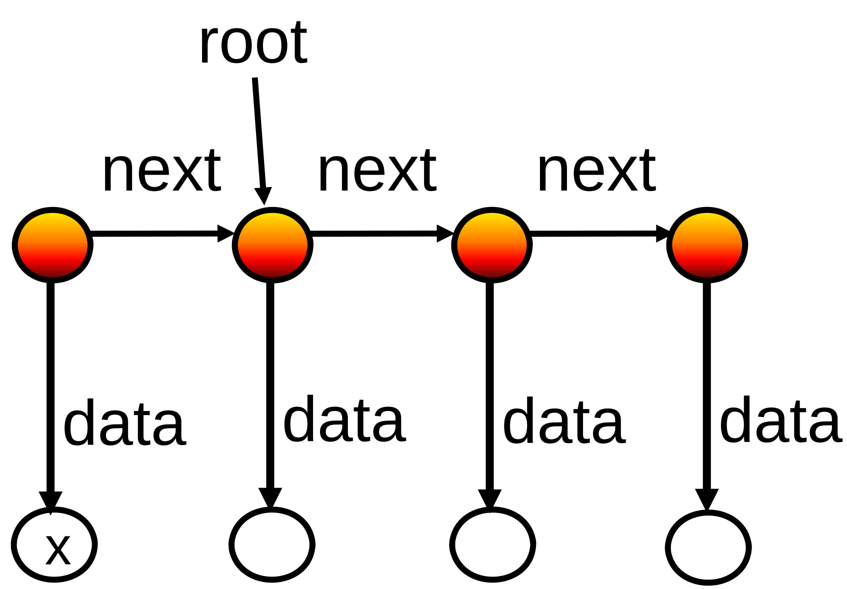
Example: Linked List

```
class List {
  private List next;
  private Object data;
  private static List root;
  private static int size;
```

invariant : size = |{data(n). next*(root,n)}|

```
  public static void addNew(Object x) {
     List n1 = new List();
     n1.next = root;
     n1.data = x;
     root = n1;
     size = size + 1;
  }
}
```

size: 4



Is invariant preserved?

Verification Condition for Example

$\neg next0^*(root0,n1) \wedge x \notin \{data0(n) \mid next0^*(root0,n)\} \wedge$
$next=next0[n1:=root0] \wedge data=data0[n1:=x] \rightarrow$

$|\{data(n) \,.\, next^*(n1,n)\}| =$

$|\{data0(n) \,.\, next0^*(root0,n)\}| + 1$

"The number of stored objects has increased by one."

This VC belongs to an expressive logic

- transitive closure * (in lists, but also in trees)
- uninterpreted functions (data, data0)
- cardinality operator on sets | ... |

How to prove such complex formulas?

Jahob Verifier for Java

Specifications written in subset of Isabelle/HOL

- ghost and 'dependent' specification variables HOL type (sets, relations, functions)

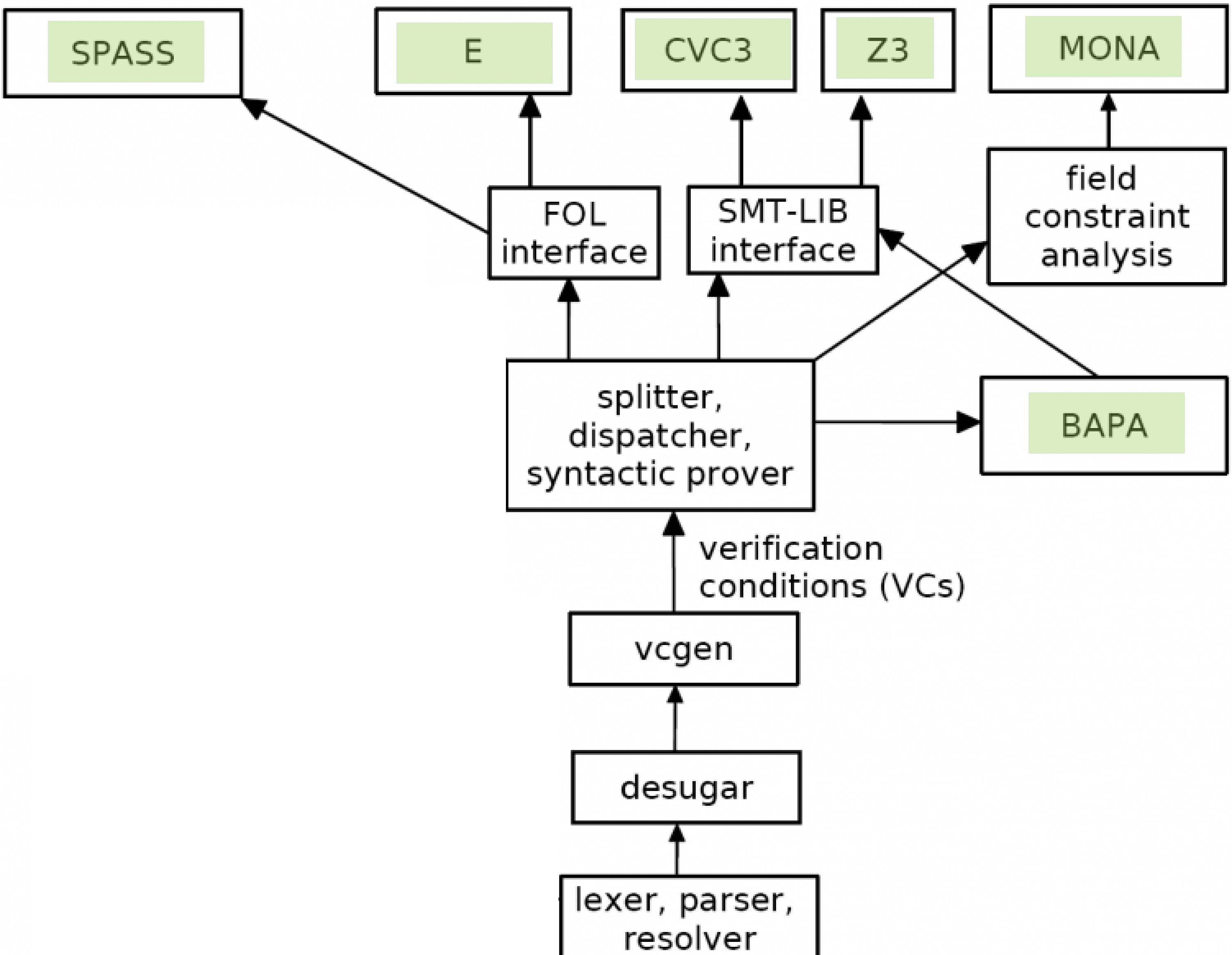
Jahob proves

- data structure preconditions, postconditions
- data structure invariants
- absence of run-time errors

We can make verification easier through:

1. decison procedures for expressive logics
2. manual proof decomposition techniques
3. techniques that combine decision procedures

Automated Provers in Jahob



Using FOL Provers in Jahob

Two main classes of provers targetted

1. SMT provers (Z3, CVC3, Yices) using SMT-LIB interface
   - good for arithmetic

2. TPTP provers (E, Vampire, SPASS) using TPTP interface
   - good for quantified formulas

Idea of FOL Translations

- Use quantifiers for set algebra operations

content = old content U {(elem,len)} →

ALL x::obj. ALL y::int.
content(x,y) =(old_content(x,y) \/ (x=elem /\ y=len))

- Eliminate lambda exprs, fun. updates, if-then

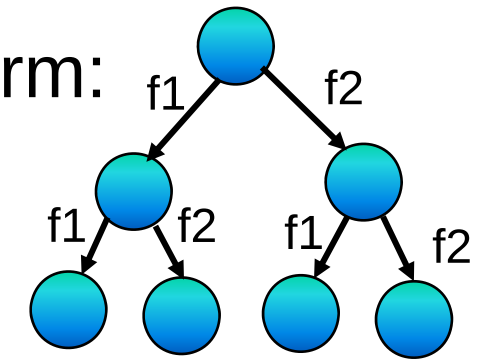
(icontent :: obj => (obj*int) set) :=

% o::obj. if (o=this) then this..icontent U {(elem,len)}
else o..icontent →

ALL o::obj. ALL x::obj. ALL y::int.
icontent(o,x,y) = ...

WS2S: Monadic 2^{nd} Order Logic

Weak Monadic 2^{nd}-order Logic of 2 Successors



In HOL, satisfiability of formulas of the form:

tree[f1,f2] & F(f1,f2,S,T)

where

- tree[f1,f2] means f1,f2 form a tree

F ::= x=f1(y) | x=f2(y) |x∈S | S⊆T | ∃S.F | $F_1 \wedge F_2$ | ¬F

- quantification is over finite sets of positions in tree
- transitive closure encoded using set quantification

Decision procedure

- recognize WS2S formula within HOL
- run the MONA tool (tree automata, BDDs)

New Decision Procedures: BAPA

Boolean Algebra with Presburger Arithmetic

$$\mathbf{S} ::= \mathbf{V} \mid \mathbf{S_1} \cup \mathbf{S_2} \mid \mathbf{S_1} \cap \mathbf{S_2} \mid \mathbf{S_1} \setminus \mathbf{S_2}$$

$$\mathbf{T} ::= \mathbf{k} \mid \mathbf{C} \mid \mathbf{T_1} + \mathbf{T_2} \mid \mathbf{T_1} - \mathbf{T_2} \mid \mathbf{C} \cdot \mathbf{T} \mid |\mathbf{S}|$$

$$\mathbf{A} ::= \mathbf{S_1} = \mathbf{S_2} \mid \mathbf{S_1} \subseteq \mathbf{S_2} \mid \mathbf{T_1} = \mathbf{T_2} \mid \mathbf{T_1} < \mathbf{T_2}$$

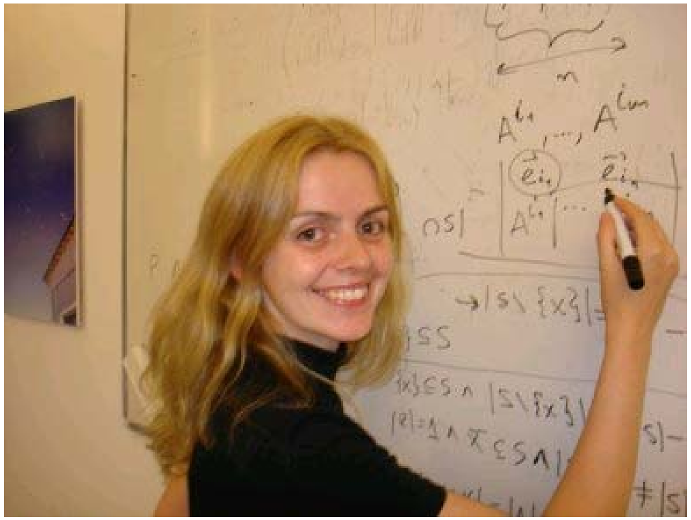
$$\mathbf{F} ::= \mathbf{A} \mid \mathbf{F_1} \wedge \mathbf{F_2} \mid \mathbf{F_1} \vee \mathbf{F_2} \mid \neg \mathbf{F} \mid \exists \mathbf{S.F} \mid \exists \mathbf{k.F}$$

Essence of decidability: Feferman, Vaught 1959

Our results

- first implementation for BAPA (CADE'05)
- first, exact, complexity for full BAPA (JAR'06)
- first, exact, complexity for QFBAPA (CADE'07)

Generalizations of BAPA



work with

Ruzica Piskac, 2nd year
PhD student in LARA group

sets & multisets with cardinalities

Decision Procedures for Multisets with Cardinality Constraints,
Verification, Model Checking, and Abstract Interpretation, 2008

Linear Arithmetic with Stars
Computer Aided Verification, 2008

Fractional Collections with Cardinality Bounds
Computer Science Logic, 2008

Recently: role of BAPA in combining logics

Jahob Verifier for Java

Specifications written in subset of Isabelle/HOL

- ghost and ‘dependent’ specification variables HOL type (sets, relations, functions)

Jahob proves

– data structure preconditions, postconditions

– data structure invariants

– absence of run-time errors

We can make verification easier through:

1. decison procedures for expressive logics
2. manual proof decomposition techniques
3. techniques that combine decision procedures

Manual Decomposition: Two Approaches

One option: use e.g. Isabelle to prove VC

Problem: users must map VC ⇔ Java code

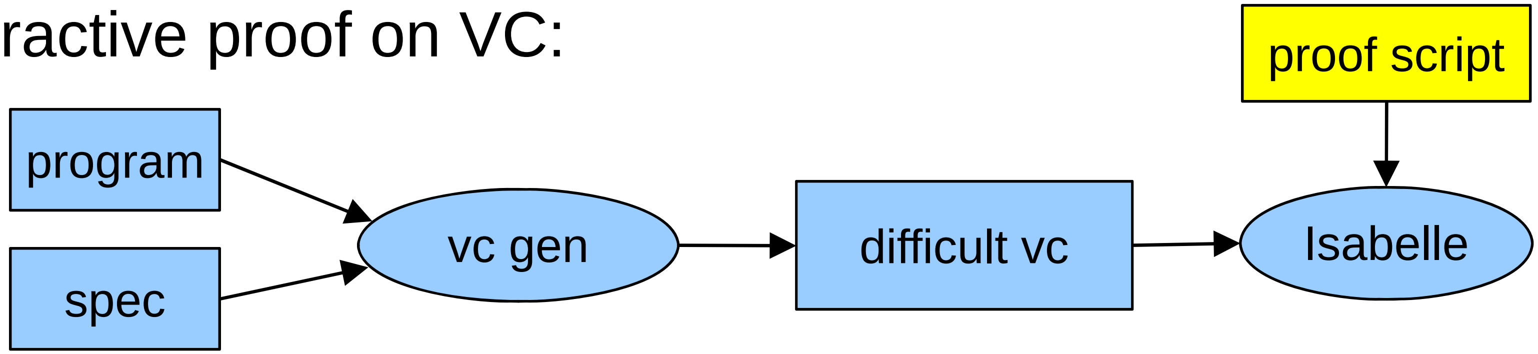
Alternatives:

1) Make interactive provers that work with Java
   – KeY

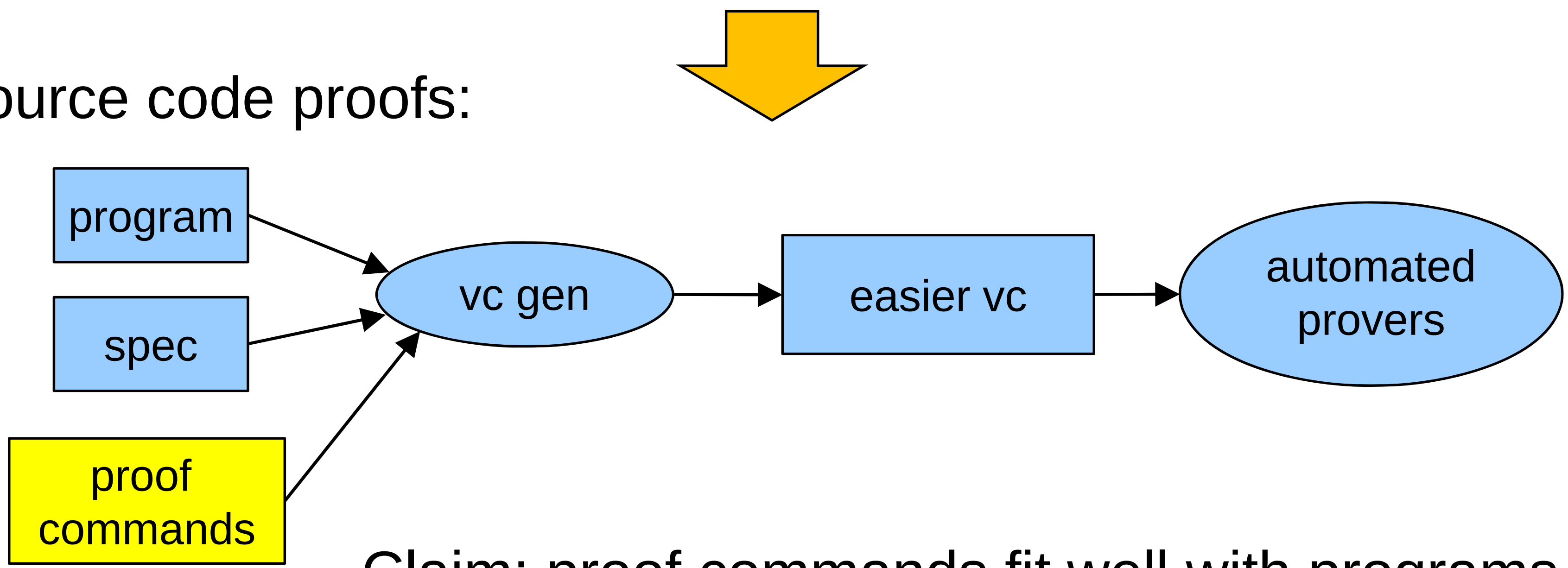
2) Programming-like constructs for verification
   – languages with dependent types (functional)
   – proof decomposition commands in Jahob (hope: make it easy for programmers to use)

Proving Difficult VCs in Jahob

interactive proof on VC:



source code proofs:

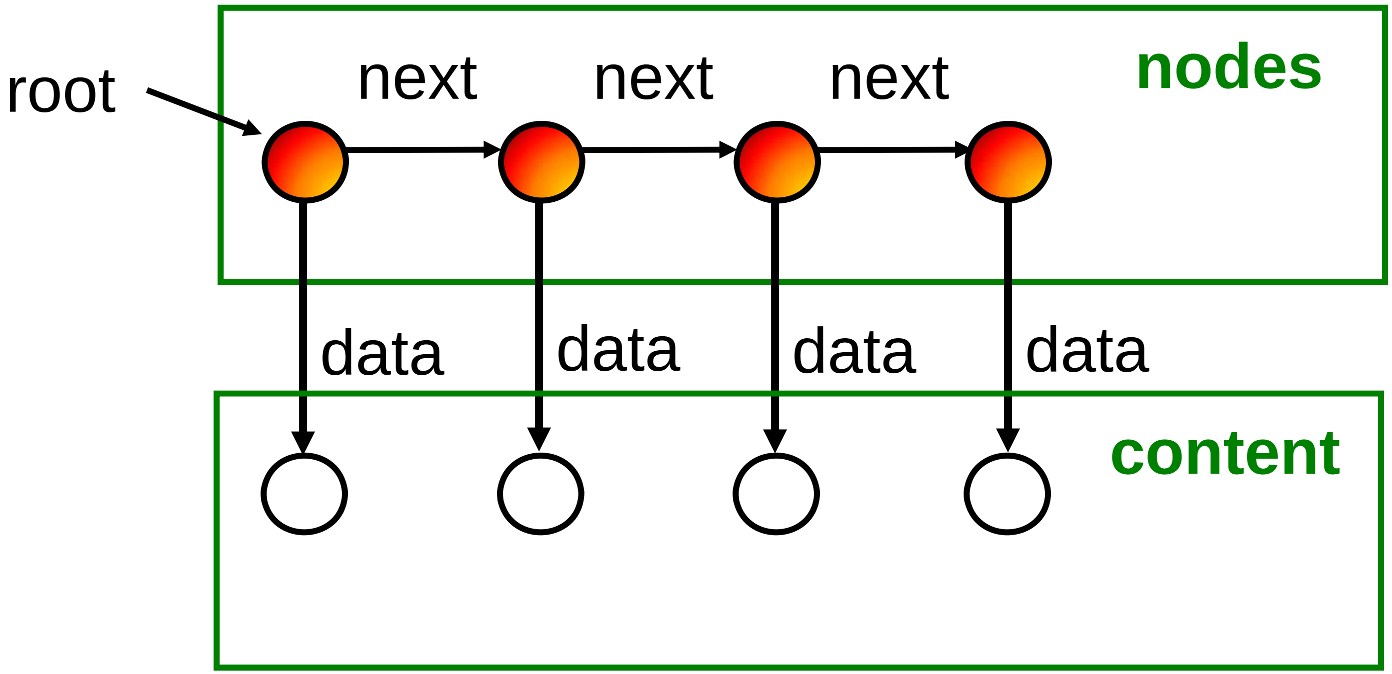


Claim: proof commands fit well with programs

Specification Variables
for Manual Proof Decomposition

for Combination of Provers,
and for Specification

Specifying Linked List in Jahob



Abstract the list with its content (data abstraction)

List.java Screenshot

specs as verified comments

public interface is simple

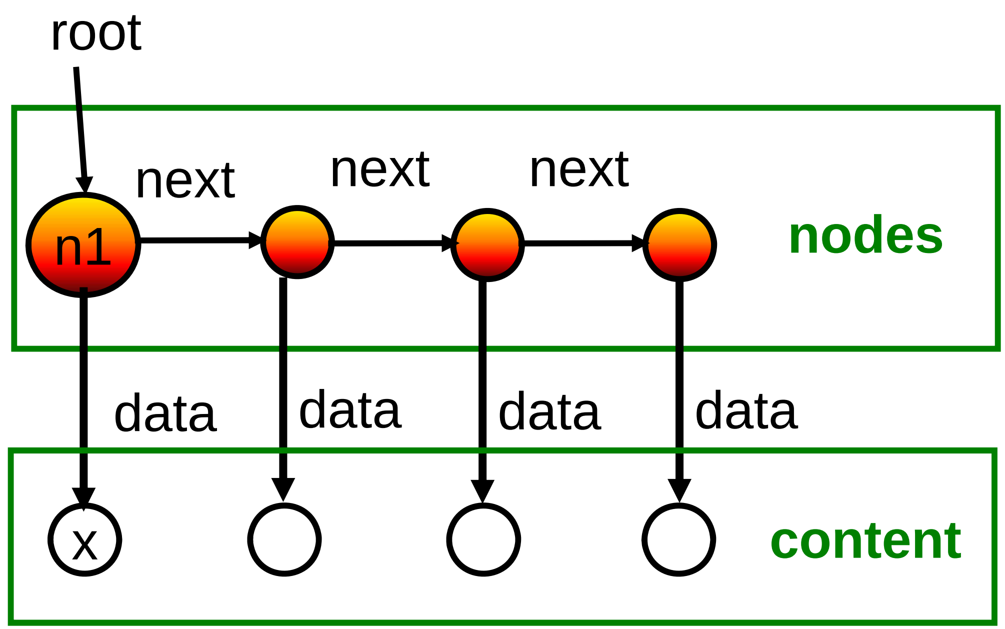
(a reason to focus on datastructures)

```
class List {
  private List next;
  private Object data;

  private static List root;
  private static int size;
  /*:
    private static ghost specvar nodes :: objset;
    public static ghost specvar content :: objset;

    invariant nodesDef: "nodes = {n. n ≠ null ∧ (root,n) ∈ {(u,
    invariant contentDef: "content = {x. ∃ n. x = List.data n ∧

    invariant sizeInv: "size = cardinality content";
    invariant treeInv: "tree [List.next]";
    invariant rootInv: "root ≠ null ⟶ (∀ n. List.next n ≠ root
    invariant nodesAlloc: "nodes ⊆ Object.alloc";
    invariant contentAlloc: "content ⊆ Object.alloc";
   */

  public static void addNew(Object x)
  /*: requires "(x ∉ content)"
      modifies content
      ensures "content = old content ∪ {x}"
  */
  {
    List n1 = new List();
    n1.next = root;
    n1.data = x;
```

Verification Condition for size

next0, data0, size0,
nodes0, content0

```
List n1 = new List();
n1.next = root;
n1.data = x;
root = n1;
size = size + 1;
//: nodes = nodes ∪ {n1}
```



next, data, size, nodes, content

next=next0[n1:=root0] ∧ data=data0[n1:=x] ∧ … →

MONA: nodes = nodes0 ∪ {n1}

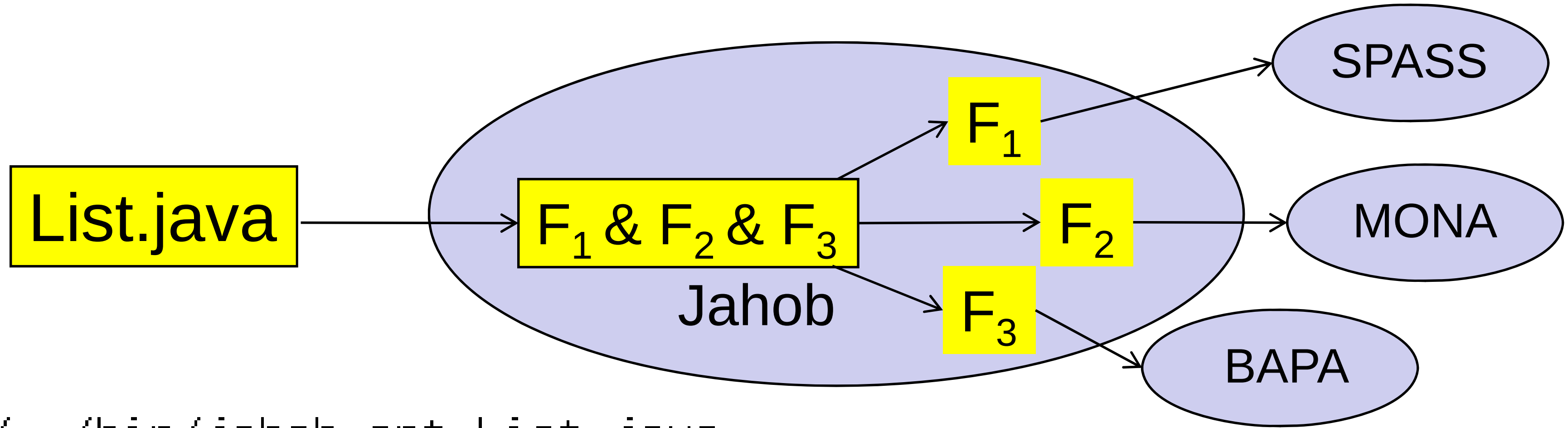
SPASS: content = content0 ∪ {x}

BAPA: |content| = |content0| + 1

∴ |{data(n) | (n1,n) ∈ next*} | =

|{data0(n) | (root0,n) ∈ next0*}| + 1

Verifying the addNew Method



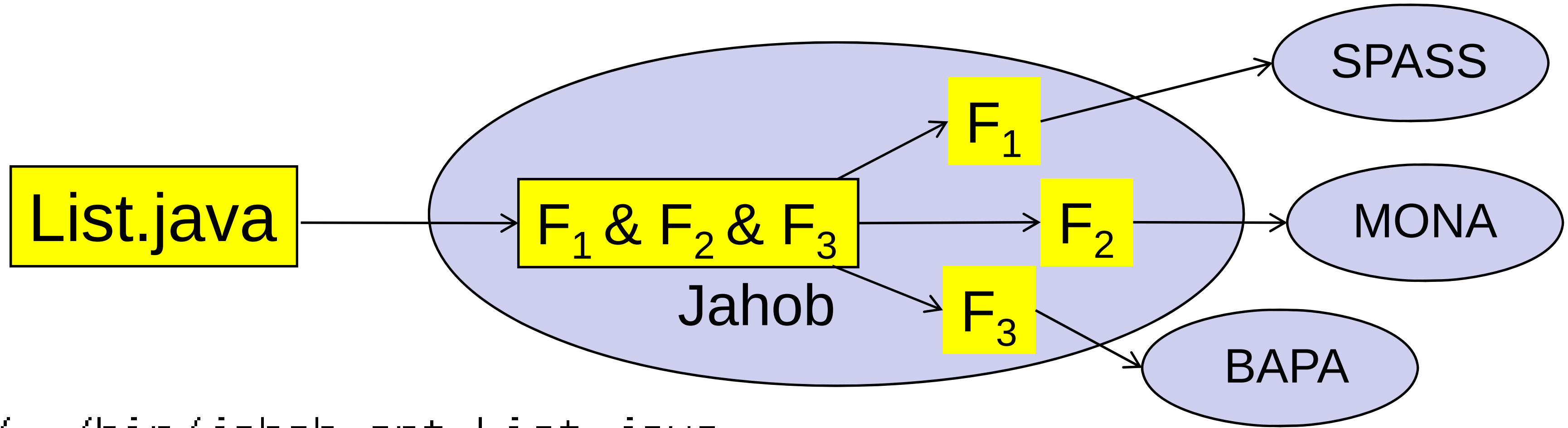

```
../../bin/jahob.opt List.java
-method List.addNew -usedp spass mona bapa
```

Verification steps

- generate verification condition (VC) in HOL stating "The program satisfies its specification"
- split VC into a conjunction of smaller formulas F_i
- *approximate* each F_i with stronger F'_i in HOL subset

prove each F'_i conjunct w/ SPASS,MONA,BAPA

Verifying the addNew Method




```
../../bin/jahob.opt List.java
-method List.addNew -usedp spass mona bapa
```

```
[List.addNew:..s!.....xx...sx!....xx...s!....xx.......
==================================================
Built-in validity checker proved 2 sequents during splitting.
SPASS proved 5 out of 8 sequents. Total time : 0.2 s
MONA proved 2 out of 3 sequents. Total time : 0.7 s
BAPA proved 1 out of 1 sequents. Total time : 0.0 s
==================================================
A total of 10 sequents out of 10 proved.
:List.addNew]
0=== Verification SUCCEEDED.
```

decomposition was enabled by specification variables finer-grained decomposition

Natural Deduction Commands for Manual Proof Decomposition

Guarded Commands and wp

Verification condition generation (Jahob, Spec#):

programs,spec → guarded commands → VC

Guarded command c:	wp(c,G):
assume F	F → G
assert F	F /\ G
havoc x x=*	ALL x. G
c1 [] c2 if (*) c1 else c2	wp(c1,G) /\ wp(c2,G)

Assertions as Lemmas

Command	note(F)
Meaning	assert(F); assume(F)
soundness	note(F) $\leq$ skip
wp(note F, G)	F Λ (F → G)
verification conditions	F, F → G

- Useful and intuitive mechanism
- Programmers familiar with assertions

Constrained Choice for Quantifiers

Command
Buy one command
Get three uses
	fix x suchThat F {
	c;
	note(G)
	}

Meaning	assert (∃x.F);
	havoc(x);
	assume(F);
	c;
	assert(G);
	assume (∀ x.(F→G))

1) non-deterministic change havoc x suchThat F

2) pick witness for ∃x.F in c

3) prove universal assertion

Jahob Verifier for Java

Specifications written in subset of Isabelle/HOL

- ghost and ‘dependent’ specification variables HOL type (sets, relations, functions)

Jahob proves

– data structure preconditions, postconditions

– data structure invariants

– absence of run-time errors

We can make verification easier through:

✓ 1. decison procedures for expressive logics

✓ 2. manual proof decomposition techniques

➡ 3. techniques that combine decision procedures

Combining Decision Procedures

- Widely studied problem
- At the heart of SMT provers
- In practice: disjoint theories (share only ‘=‘)
- Our generalization: decide quantifier-free combination of quantified formulas sharing set variables and set operations
- Recent EPFL technical report:

On Combining Theories with Shared Set Operations

Formula Decomposition

Consider a formula

$$|\{data(x)\ .\ next^*(root,x)\}| = k+1$$

Introduce fresh variables denoting sets:

$A = \{x.\ next^*(root,x)\} \wedge$

$B = \{y.\ \exists\ x.\ data(x,y) \wedge x \in A\} \wedge$

$|B| = k+1$

1) WS2S

2) C^2

3) BAPA

Conjuncts belong to decidable fragments

Claim: quantifier-free combination is decidable

Combining Decidable Logics

Satisfiability problem expressed in HOL:

(all free symbols existentially quantified)

$\exists$ next,data,k. $\exists$ root,A,B.

A = {x. next*(root,x)} $\wedge$

B = {y. $\exists$ x. data(x,y) $\wedge$ x $\in$ A} $\wedge$

|B|=k+1

1) WS2S

2) C^2

3) BAPA

We assume formulas share only:

- **set variables** (sets of uninterpreted elems)
- individual variables, as a special case - {x}

Combining Decidable Logics

Conjunction of projections satisfiable → so is original formula

Satisfiability problem expressed in HOL, after moving fragment-specific quantifiers

$\exists$ root,A,B.

$\exists$ next. A = {x. next*(root,x)} $\wedge$ — F_{WS2S} : {root}$\subseteq$A

$\exists$ data. B = {y. $\exists$ x. data(x,y) $\wedge$ x $\in$ A} $\wedge$ — F_{C2} : |B|$\leq$|A|

$\exists$ k. |B|=k+1 — F_{BAPA} : 1 $\leq$ |B|

Extend decision procedures to **projection procedures** for WS2S,C^2,BAPA

applies $\exists$ to all non-set variables

$\exists$ root,A,B. {root}$\subseteq$A $\wedge$|B|$\leq$|A| $\wedge$ 1 $\leq$ |B|

Decision Procedure for Combination

1. Separate formula into WS2S, C^2, BAPA parts
2. For each part, compute projection onto set vars
3. Check satisfiability of conjunction of projections

Def: Logic is BAPA-reducible iff there is an algorithm that computes BAPA formula eqiuvalent to existential quantification over non-set vars.

Thm: WS2S, C^2, EPR, BAPA are BAPA-reducible.

Proof: WS2S – Parikh image of tree language is PA
C^2 – proof by Pratt-Hartmann reduces to PA
EPR – proof based on resolution

Details in technical report

Jahob Verifier for Java

Specifications written in subset of Isabelle/HOL

- ghost and 'dependent' specification variables HOL type (sets, relations, functions)

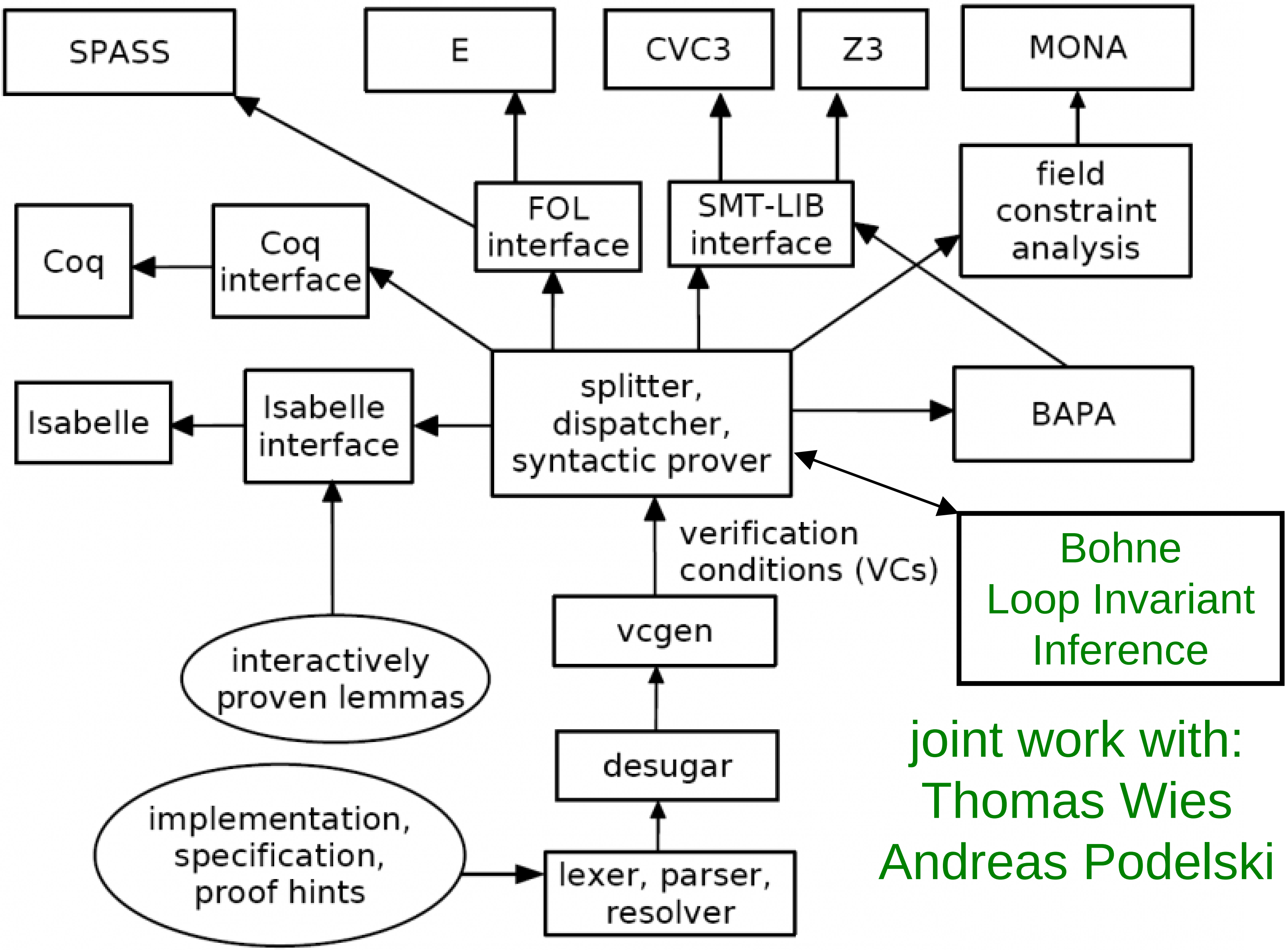
Jahob proves

- data structure preconditions, postconditions
- data structure invariants
- absence of run-time errors

We can make verification easier through:

1. decison procedures for expressive logics
2. manual proof decomposition techniques
3. techniques that combine decision procedures

Summary: Jahob Verifier



joint work with:
Thomas Wies
Andreas Podelski

Moving Forward: Rich Model Toolkit

Models for sw&hw (transition formulas)

Theorem proving and verification questions independent of the programming language

Applications to: Scala, Java, C

Goals

- common representation formats
- increase automation of verifiers, provers
  - challenge problems, run competitions
- inspire new verifiable language designs
- executing specifications, synthesis

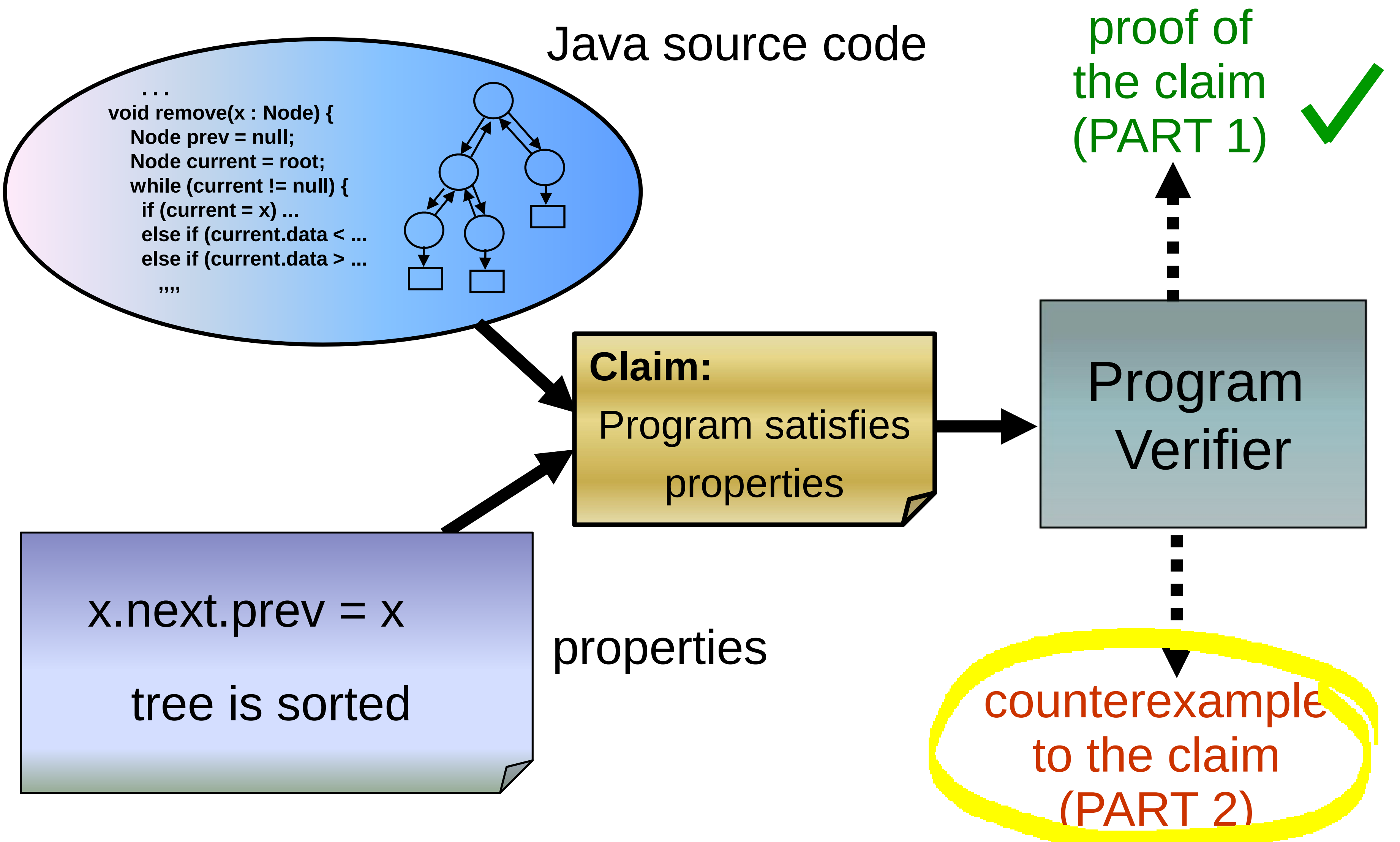
PART 2: Counterexamples

Joint work with

Milos Gligoric
Darko Marinov
UIUC

Tihomir Gvero
EPFL

Proofs and Counterexamples for Java



Adapting Our Proof Techniques to Generate Counterexamples?

Jahob was designed to generate proofs

Many approximations in one direction only:

- verification condition **implies** correctness claim
- approximated formula **implies** original one

Theorem provers give no counterexample

- FOL provers complete for proofs
  - no complete FOL proof system for non-validity (undecidability of FOL)
- or feature not implemented in prover

Possible Solutions

First approach: avoid approximation

- use decision procedures
- use complete combination methods
- keep track what was approximated, refine
- *promising in long term*

Second approach – rest of this talk

- systematic test-case generation
- end-to-end solution for counterexamples
- supports: loops, all computable invariants
- *effective, widely and immediately applicable*

Checking Hoare Triples

Program **c** with state (heap) h

Test Case Generation

havoc(h);	← arbitrary initial state
assume(size(h)<N);	← bound
assume(P(h));	← precondition

Testing (with Runtime Checks)

c;	← program
assert(Q(h))	← postcondition

Checking Hoare Triples

Program **c** with state (heap) h

Test Case Generation

havoc(h); ← arbitrary initial state

assume(size(h)<N); ← bound

assume(P(h)); ← precondition

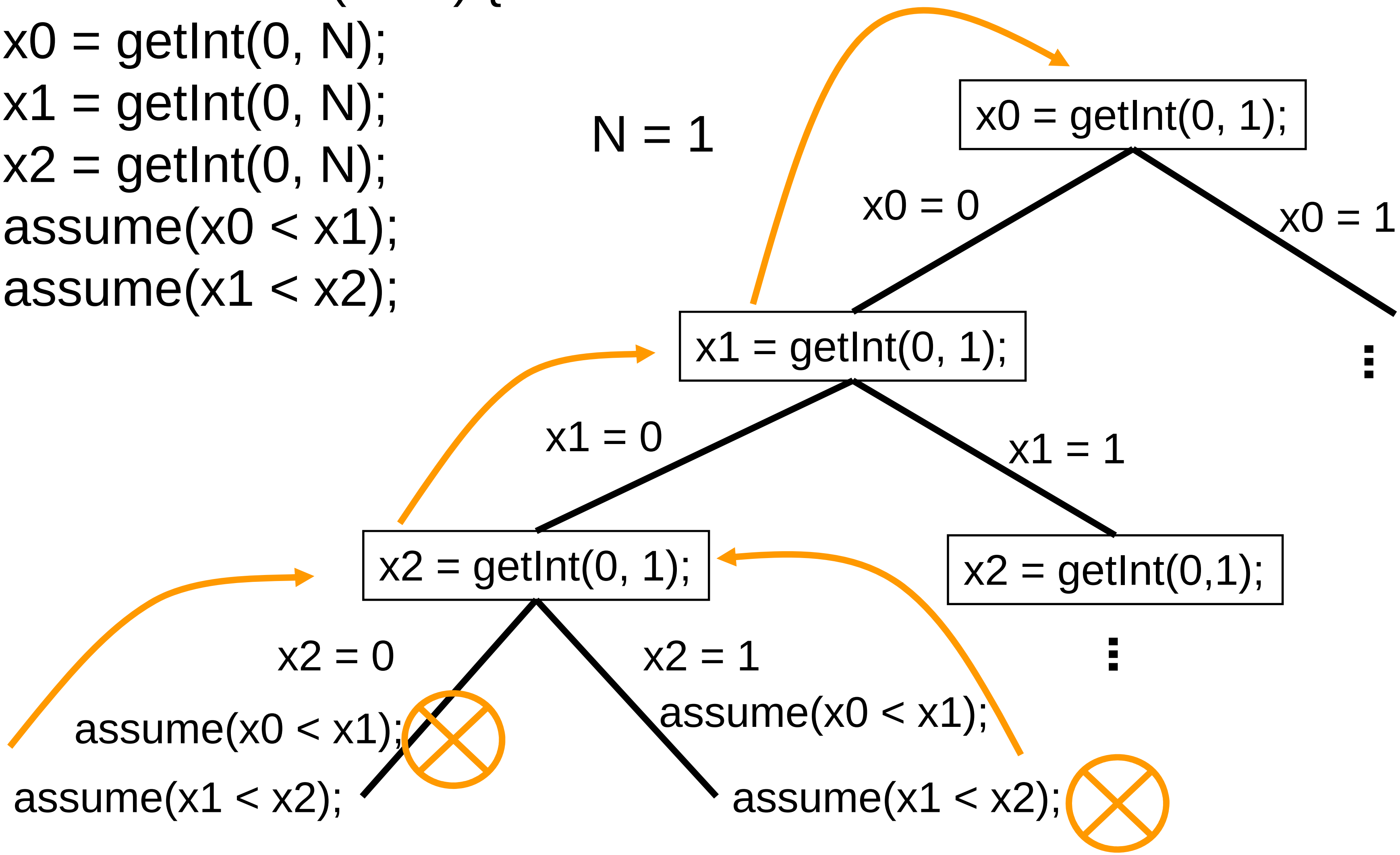
View test generation as systematic execution of (bounded-choice) guarded command language

x = getInt(0,N) means:

havoc(x); assume(0 < x <= N)

Systematic Execution Example

```
static void main(int N) {
  x0 = getInt(0, N);
  x1 = getInt(0, N);
  x2 = getInt(0, N);
  assume(x0 < x1);
  assume(x1 < x2);
}
```



Delayed Execution (for Integers)

eager evaluation

```
x0 = Susp(0,1);
force(x0);
x1 = Susp(0,1);
force(x1);
x2 = Susp(0,1);
force(x2);
assume(x0 < x1);
assume(x1 < x2);
```

original code

```
x0 = getInt(0,1);
x1 = getInt(0,1);
x2 = getInt(0,1);
assume(x0< x1);
assume(x1 < x2);
```

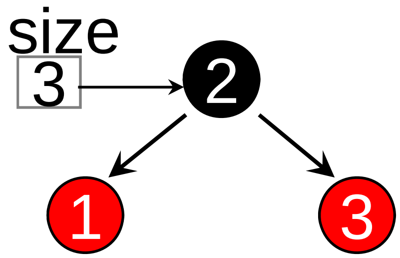
delayed execution

```
x0 = Susp(0,1);
x1 = Susp(0,1);
x2 = Susp(0,1);
force(x0);
force(x1);
assume(x0 < x1);
force(x2);
assume(x1 < x2);
```

Linked Data Structure: Red-Black Tree

```
class TreeSet implements Set {
  int size;
  Node root;

  static class Node {
    Node left, right, parent;
    boolean color;
    int value;
  }

  void add(int v) { ... }
  void remove(int v) { ... }
}
```

Example Tree



Red-black tree invariants:

– *treeness*
– *coloring*
– *ordering*

Object Pools: Abstractly Choosing Objects (avoids isomorphisms)

```
static void main(int N) {
  TreeSet  t = new TreeSet();
  t.initialize(N, N);
  assume(t.isRBT());
  int v = getInt(0, N);
  t.remove(v);
  assert(t.isRBT());
}
```

another way:
n= nodes.getNew()
(pick object distinct from previous ones)

```
void initialize(int maxSize, int maxKey) {
  size = getInt(1, maxSize);
  ObjectPool<Node> nodes =
         new ObjectPool<Node>(size);
  root = nodes.getAny();
  for (Node n : nodes) {
    n.left = nodes.getAny();
    n.right = getAny();
    n.parent = nodes.getAny();
    n.color = getBoolean();
    n.key = getInt(1, maxKey);
} }
```

Implementation

Implemented in Java Pathfinder from NASA

Explicit-state model checker working on bytecodes

Implemented delayed execution, object pools

Contribution incorporated by JPF developers

Delayed Choice is Essential for Efficiency

		Eager Choice	Delayed Choice
data structure	N	time [s]	time [s]
RedBlackTree	7	9.96	3.24
	8	65.67	13.85
	9	449.17	64.24
DAG	3	5.68	0.69
	4	out of mem	6.41
	5	-	1,013.75
HeapArray	6	16.66	4.12
	7	304.32	32.43
	8	8,166.77	318.59
SortedList	6	5.94	0.64
	7	900.67	2.38
	8	1865.55	9.85

Testing the Framework and Java Pathfinder

generator	time [s]	actual bugs found
AnnotatedMethod	24.77	2
RefactoringGet	5.30	1
DeclaredMethodsReturn	8.22	1
RefactoringSet	5.33	1
StructureClass	10.34	4
DeclaredFieldTest	51.67	1
ClassCastMethod	47.57	1

Conclusions

Jahob verifier

- specifications in subset of Isabelle/HOL
- applied to verify many data structures

Making verification easier through:

1. decison procedures for expressive logics
2. manual proof decomposition techniques
3. techniques that combine decision procedures

Finding counterexamples using test generation

- delayed execution essential for performance
- found bugs in real code, incorporated into JPF