Full Functional Verification of Linked Data Structures

Karen Zee[†], Viktor Kuncak[‡], and Martin Rinard[†]

[†]MIT CSAIL [‡]EPFL, I&C

Goal

Verify full functional correctness of linked data structure implementations

What is Full Functional Correctness?

- Complete, precise formal specification
- Captures every property client needs (except resource consumption)
- Implementation satisfies specification

Benefits of Full Functional Correctness

- Complete, precise, unambiguous interfaces for linked data structures
- Enables sound reasoning with specification (can discard implementation when reasoning)
  - Human developers
  - Automated analyses of client code
- First complete realization of concept of abstract data types

Example

Hashtable Specification

```
class Hashtable {
    //: public ghost specvar content :: “(obj * obj) set” = “{}”;
    //: public ghost specvar init :: “bool” = “false”;

    public Object put(Object key, Object value)
    /*: requires “init ∧ key ≠ null ∧ value ≠ null”
        modifies content
        ensures “content = old content - {(key, result)} ∪ {(key, value)} ^
          (result = null → ¬(∃v. (key, v) ∈ old content)) ∧
          (result ≠ null → (key, result) ∈ old content)” */
     { … }
    …
}
```

- Specifications at class granularity
- Specifications appear as comments
- Can use standard Java compilers

Abstract State as Sets, Relations

```
class Hashtable {
    //: public ghost specvar content :: "(obj * obj) set" = "{}";
    //: public ghost specvar init :: "bool" = "false";

    public Object put(Object key, Object value)
    /*: requires "init ∧ key ≠ null ∧ value ≠ null"
        modifies content
        ensures "content = old content - {(key, result)} ∪ {(key, value)} ^
          (result = null → ¬(∃v. (key, v) ∈ old content)) ∧
          (result ≠ null → (key, result) ∈ old content)" */
     { … }
    …
}
```

- Represent abstract state using specification variables
- Contents of hash table as set of key-value pairs

Method Preconditions, Postconditions

class Hashtable {
    //: **public ghost specvar** *content* :: “(obj * obj) set” = “{}”;
    //: **public ghost specvar** *init* :: “bool” = “false”;

    public Object put(Object key, Object value)
    /*: **requires** “init ∧ key ≠ null ∧ value ≠ null”
        modifies *content*
        ensures “content = old content - {(key, result)} ∪ {(key, value)} ^
            (result = null → ¬(∃v. (key, v) ∈ old content)) ∧
            (result ≠ null → (key, result) ∈ old content)” */
    { … }
    …
}

- Standard assume-guarantee reasoning for method interfaces
- Pre-, post-conditions in higher-order logic (HOL)

Requires Clause

```
class Hashtable {
    //: public ghost specvar content :: "(obj * obj) set" = "{}";
    //: public ghost specvar init :: "bool" = "false";

    public Object put(Object key, Object value)
    /*: requires "init ∧ key ≠ null ∧ value ≠ null"
        modifies content
        ensures "content = old content - {(key, result)} ∪ {(key, value)} ^
          (result = null → ¬(∃v. (key, v) ∈ old content)) ∧
          (result ≠ null → (key, result) ∈ old content)" */
     { … }
    …
}
```

Pre-condition requires that key and value be non-null

Modifies Clause

```
class Hashtable {
    //: public ghost specvar content :: “(obj * obj) set” = “{}”;
    //: public ghost specvar init :: “bool” = “false”;

    public Object put(Object key, Object value)
    /*: requires “init ∧ key ≠ null ∧ value ≠ null”
        modifies content
        ensures “content = old content - {(key, result)} ∪ {(key, value)} ^
           (result = null → ¬(∃v. (key, v) ∈ old content)) ∧
           (result ≠ null → (key, result) ∈ old content)” */
     { … }
    …
}
```

- Modifies clause gives frame condition
- **put** method modifies only *content*

Ensures Clause

```
class Hashtable {
    //: public ghost specvar content :: “(obj * obj) set” = “{}”;
    //: public ghost specvar init :: “bool” = “false”;

    public Object put(Object key, Object value)
    /*: requires “init ∧ key ≠ null ∧ value ≠ null”
        modifies content
        ensures “content = old content - {(key, result)} ∪ {(key, value)} ^
          (result = null → ¬(∃v. (key, v) ∈ old content)) ∧
          (result ≠ null → (key, result) ∈ old content)” */
     { … }
    …
}
```

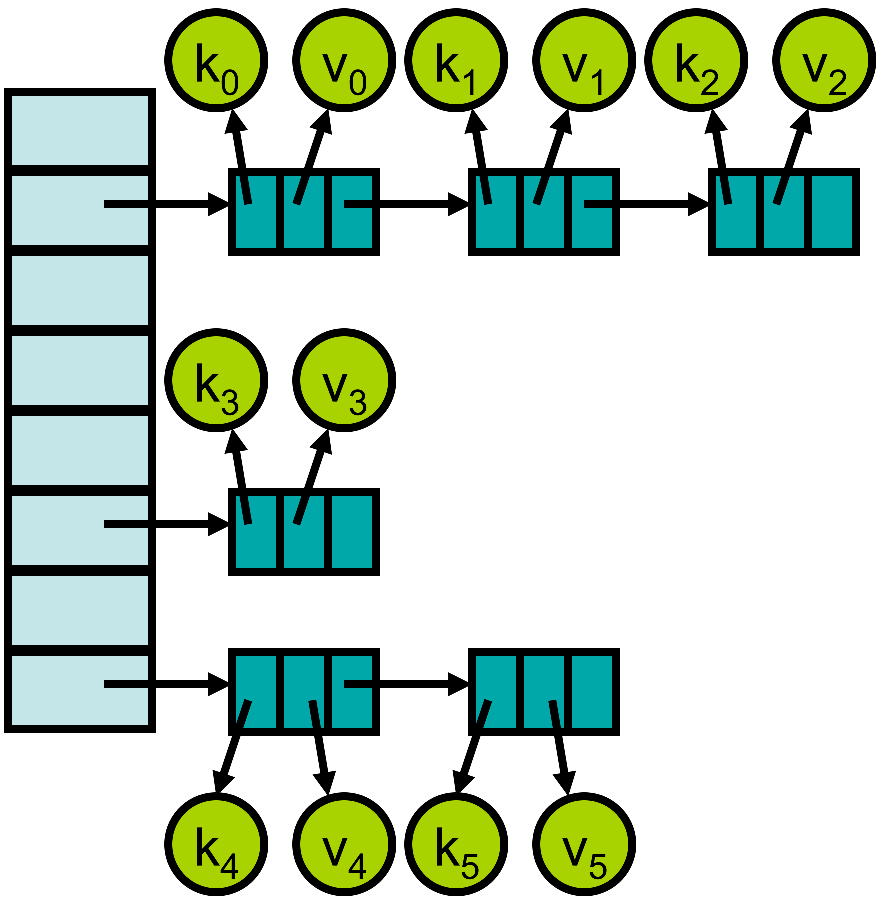
- Previous key-value binding is removed
- New binding is added

Ensures Clause

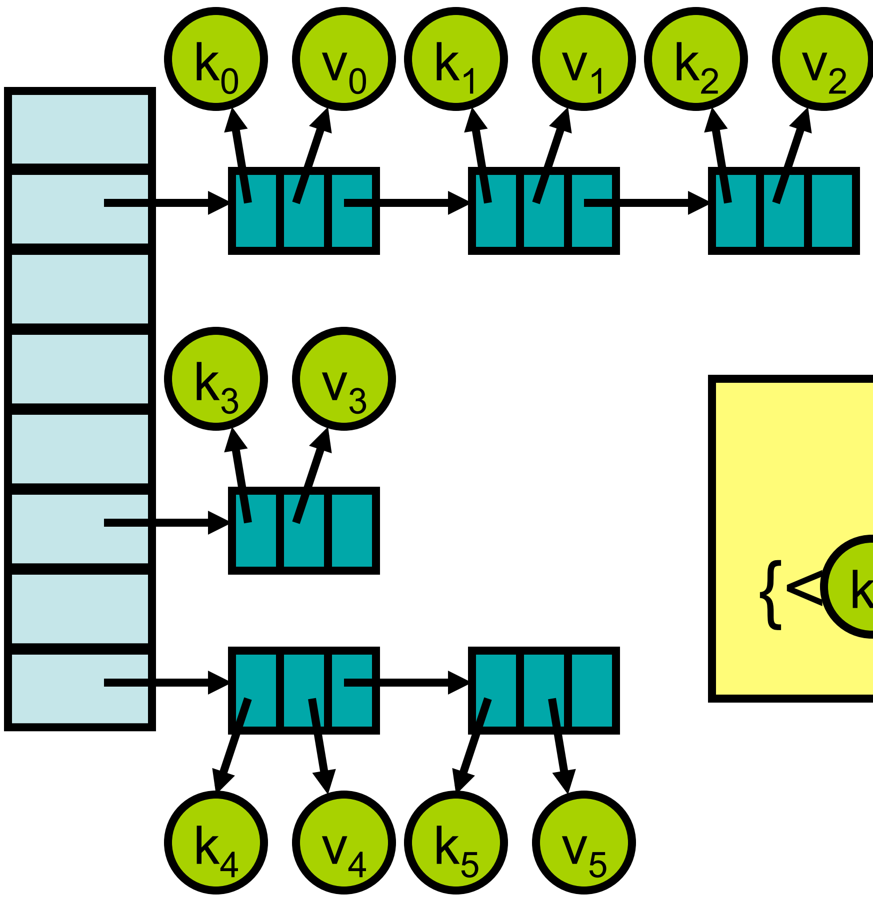
```
class Hashtable {
    //: public ghost specvar content :: “(obj * obj) set” = “{}”;
    //: public ghost specvar init :: “bool” = “false”;

    public Object put(Object key, Object value)
    /*: requires “init ∧ key ≠ null ∧ value ≠ null”
        modifies content
        ensures “content = old content - {(key, result)} ∪ {(key, value)} ^
          (result = null → ¬(∃v. (key, v) ∈ old content)) ∧
          (result ≠ null → (key, result) ∈ old content)” */
     { … }
    …
}
```

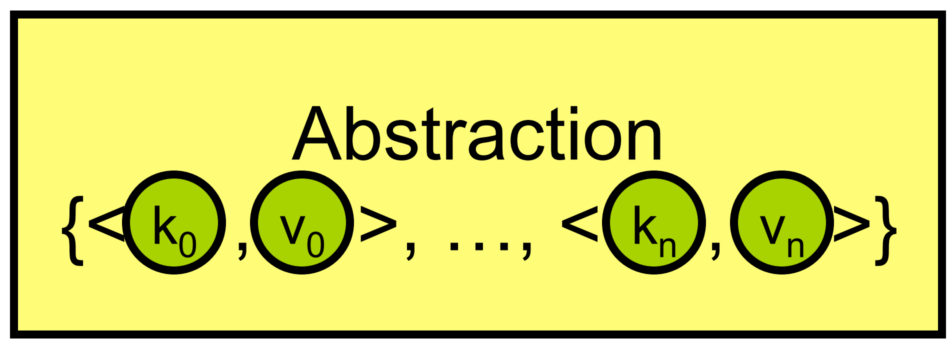
Returns previously-bound value or null

Hashtable Data Structure



Hashtable Data Structure





Abstraction Function

```
/*:  invariant ContentDef:
     "init → content = {(k,v). (∃ i. 0 ≤ i ∧ i < table.length ∧ (k,v) ∈ table[i].bucketContent)}"

     static ghost specvar bucketContent :: "obj ⇒ (obj * obj) set" = "λ n. {}"
     invariant bucketContentNull: "null.bucketContent = {}"
     invariant bucketContentDef: "∀x. x ∈ Node ∧ x ∈ alloc ∧ x ≠ null → x.bucketContent =
     {<x.key, x.value>} ∪ x.next.bucketContent ∧ (∀v. (x.key, v) ∉ x.next.bucketContent)"

     invariant Coherence:
     "init → (∀ i k v. (k,v) ∈ table[i].bucketContent → i = hash k table.length)"

     static specvar hash :: "obj ⇒ int ⇒ int"
     vardefs "hash == λ k. (λ n. (abs (hashFunc  k)) mod n)"

     static specvar abs :: "int ⇒ int"
     vardefs "abs == λ m. (if (m < 0) then -m else m)"
     …
*/
```

- Invariants
- Dependent specification variables

Abstraction Function

/*: **invariant** ContentDef:
“init → content = {(k,v). (∃ i. 0 ≤ i ∧ i < table.length ∧ (k,v) ∈ table[i].bucketContent)}”

static ghost specvar bucketContent :: “obj ⇒ (obj * obj) set” = “λ n. {}”
invariant bucketContentNull: “null.bucketContent = {}”
invariant bucketContentDef: “∀x. x ∈ Node ∧ x ∈ alloc ∧ x ≠ null → x.bucketContent = {<x.key, x.value>} ∪ x.next.bucketContent ∧ (∀v. (x.key, v) ∉ x.next.bucketContent)”

invariant Coherence:
“init → (∀ i k v. (k,v) ∈ table[i].bucketContent → i = hash k table.length)”

static specvar hash :: “obj ⇒ int ⇒ int”
vardefs “hash == λ k. (λ n. (abs (hashFunc k)) mod n)”

static specvar abs :: “int ⇒ int”
vardefs “abs == λ m. (if (m < 0) then -m else m)”
…
*/

Hash table contents consists of the contents of all the buckets

Abstraction Function

/*: **invariant** ContentDef:
"init → content = {(k,v). (∃ i. 0 ≤ i ∧ i < table.length ∧ (k,v) ∈ table[i].bucketContent)}"

static ghost specvar bucketContent :: "obj ⇒ (obj * obj) set" = "λ n. {}"
invariant bucketContentNull: "null.bucketContent = {}"
invariant bucketContentDef: "∀x. x ∈ Node ∧ x ∈ alloc ∧ x ≠ null → x.bucketContent =
{<x.key, x.value>} ∪ x.next.bucketContent ∧ (∀v. (x.key, v) ∉ x.next.bucketContent)"

invariant Coherence:
"init → (∀ i k v. (k,v) ∈ table[i].bucketContent → i = hash k table.length)"

static specvar hash :: "obj ⇒ int ⇒ int"
vardefs "hash == λ k. (λ n. (abs (hashFunc k)) mod n)"

static specvar abs :: "int ⇒ int"
vardefs "abs == λ m. (if (m < 0) then -m else m)"
…
*/

- Contents of each bucket defined recursively over linked list
- Keys are unique

Abstraction Function

/*: **invariant** ContentDef:
"init → content = {(k,v). (∃ i. 0 ≤ i ∧ i < table.length ∧ (k,v) ∈ table[i].bucketContent)}"

static ghost specvar bucketContent :: "obj ⇒ (obj * obj) set" = "λ n. {}"
invariant bucketContentNull: "null.bucketContent = {}"
invariant bucketContentDef: "∀x. x ∈ Node ∧ x ∈ alloc ∧ x ≠ null → x.bucketContent = {<x.key, x.value>} ∪ x.next.bucketContent ∧ (∀v. (x.key, v) ∉ x.next.bucketContent)"

invariant Coherence:
"init → (∀ i k v. (k,v) ∈ table[i].bucketContent → i = hash k table.length)"

static specvar hash :: "obj ⇒ int ⇒ int"
vardefs "hash == λ k. (λ n. (abs (hashFunc k)) mod n)"

static specvar abs :: "int ⇒ int"
vardefs "abs == λ m. (if (m < 0) then -m else m)"
…
*/

Every key is in the correct bucket

Abstraction Function

```
/*:  invariant ContentDef:
     “init → content = {(k,v). (∃ i. 0 ≤ i ∧ i < table.length ∧ (k,v) ∈ table[i].bucketContent)}”

     static ghost specvar bucketContent :: “obj ⇒ (obj * obj) set” = “λ n. {}”
     invariant bucketContentNull: “null.bucketContent = {}”
     invariant bucketContentDef: “∀x. x ∈ Node ∧ x ∈ alloc ∧ x ≠ null → x.bucketContent =
     {<x.key, x.value>} ∪ x.next.bucketContent ∧ (∀v. (x.key, v) ∉ x.next.bucketContent)”

     invariant Coherence:
     “init → (∀ i k v. (k,v) ∈ table[i].bucketContent → i = hash k table.length)”

     static specvar hash :: “obj ⇒ int ⇒ int”
     vardefs “hash == λ k. (λ n. (abs (hashFunc  k)) mod n)”

     static specvar abs :: “int ⇒ int”
     vardefs “abs == λ m. (if (m < 0) then -m else m)”
     …
*/
```

set expressions

quantifiers

lambda expressions

Loop Invariants

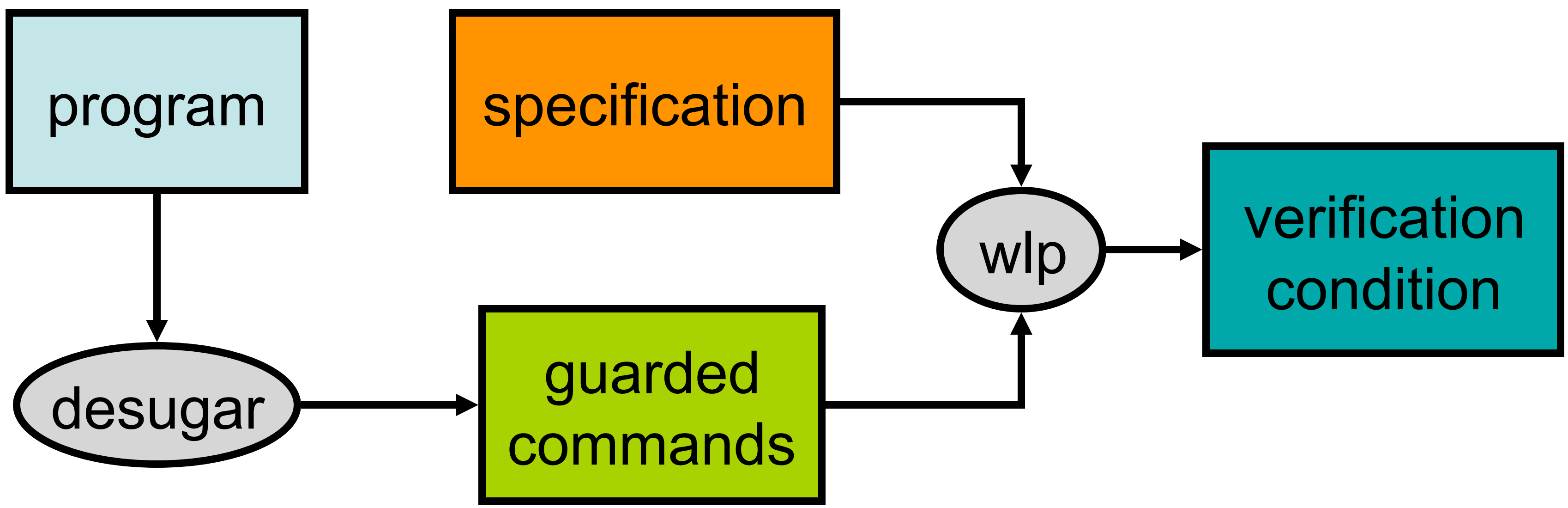
```
public Object get(Object k0)
/*:  requires "init ∧ k0 ≠ null"
     ensures "(result ≠ null → (k0, result) ∈ content) ∧
        (result = null → ¬(∃v. (k0, v) ∈ content))" */
{
     int hc = compute_hash(k0);
     Node current = table[hc];
     while /*: inv "∀v. ((k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent)" */
          (current != null) {
        if (current.key == k0) { return current.value; }
        current = current.next;
     }
       return null;
}
```

Source of Invariants

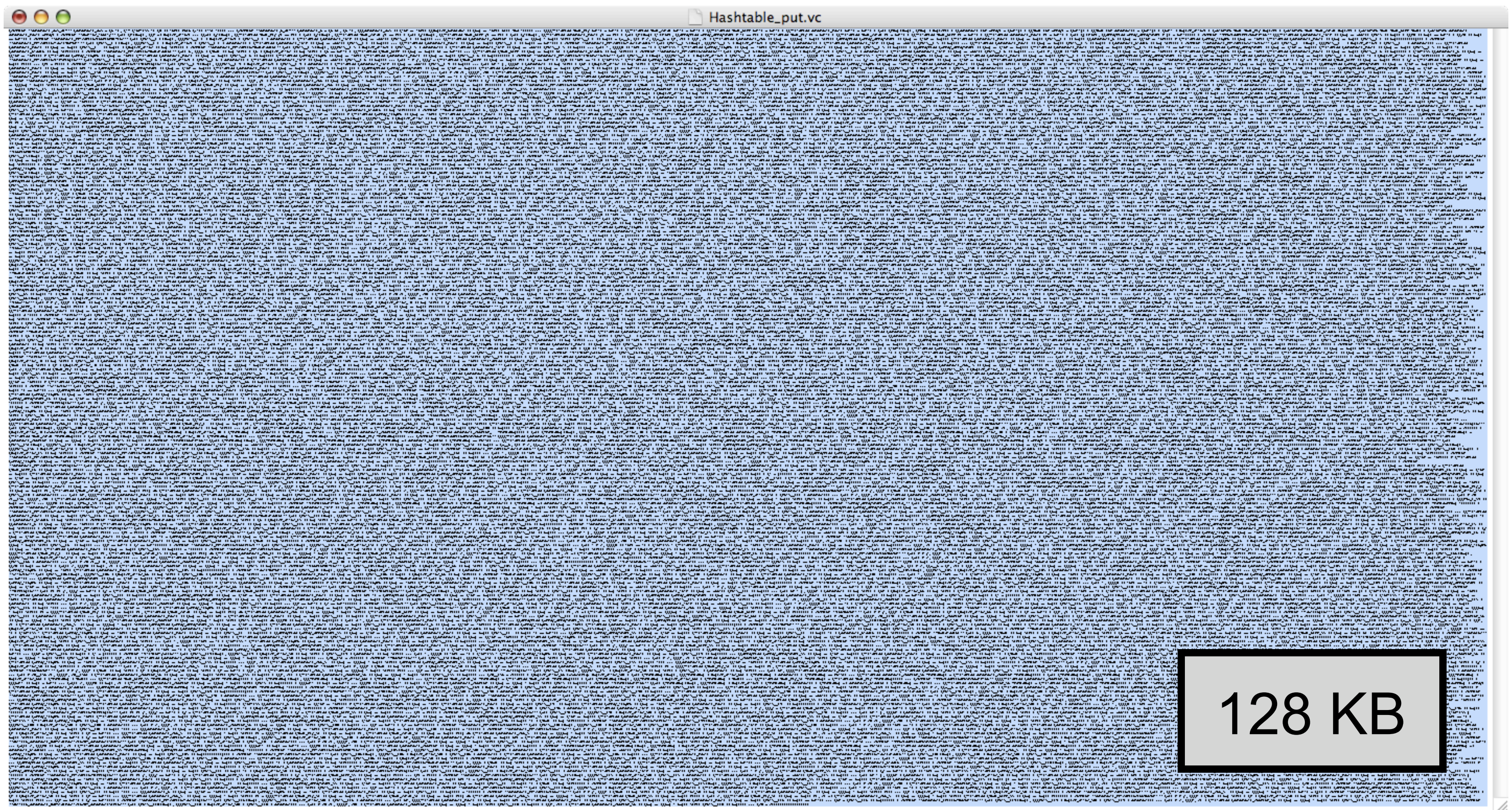
- Inferred by system
- Provided by developer
- Inferred by shape analysis

Generating Verification Conditions

- Convert to guarded commands
- Apply weakest liberal pre-conditions



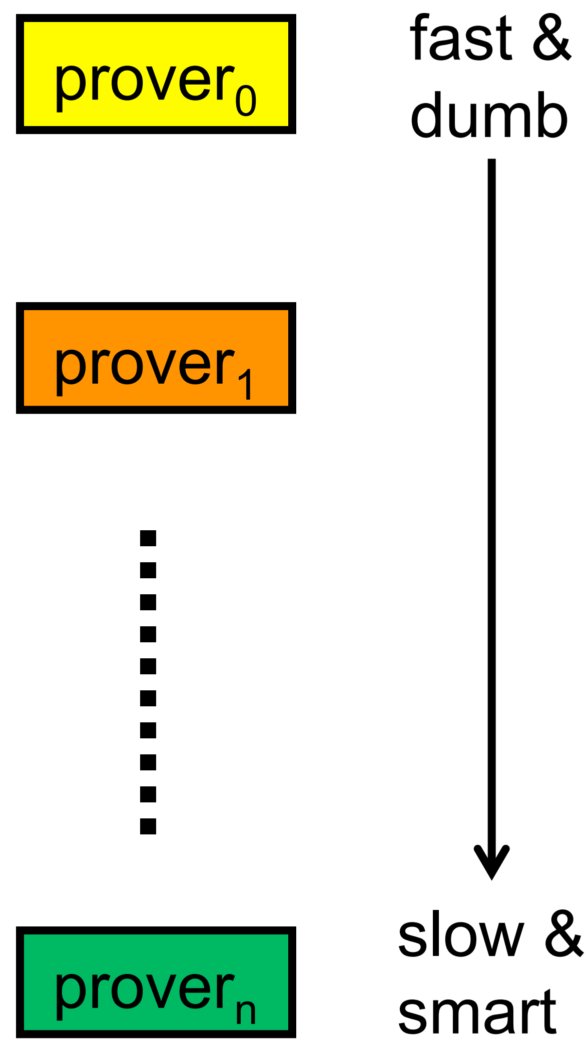
Verification Condition for Hashtable.put

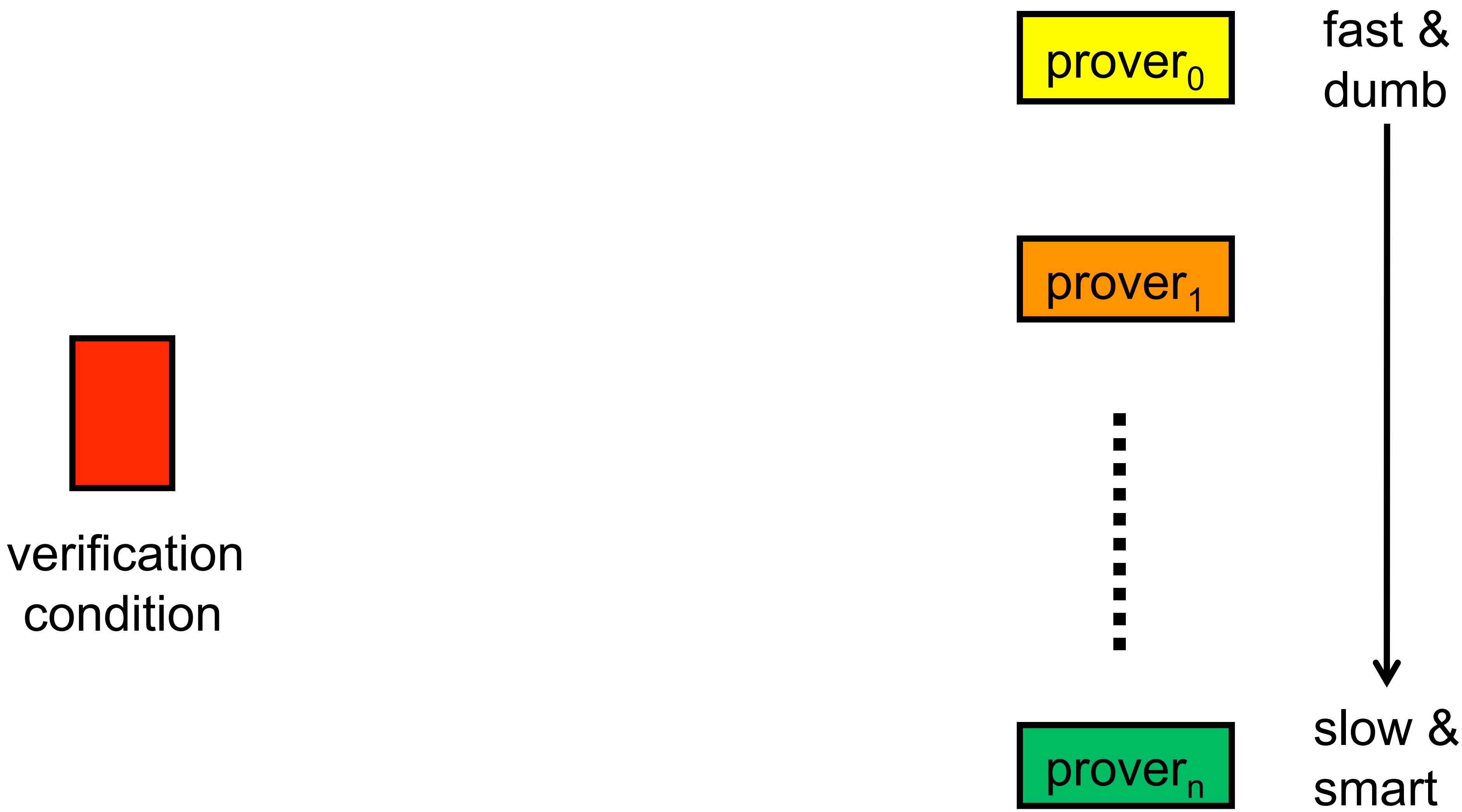


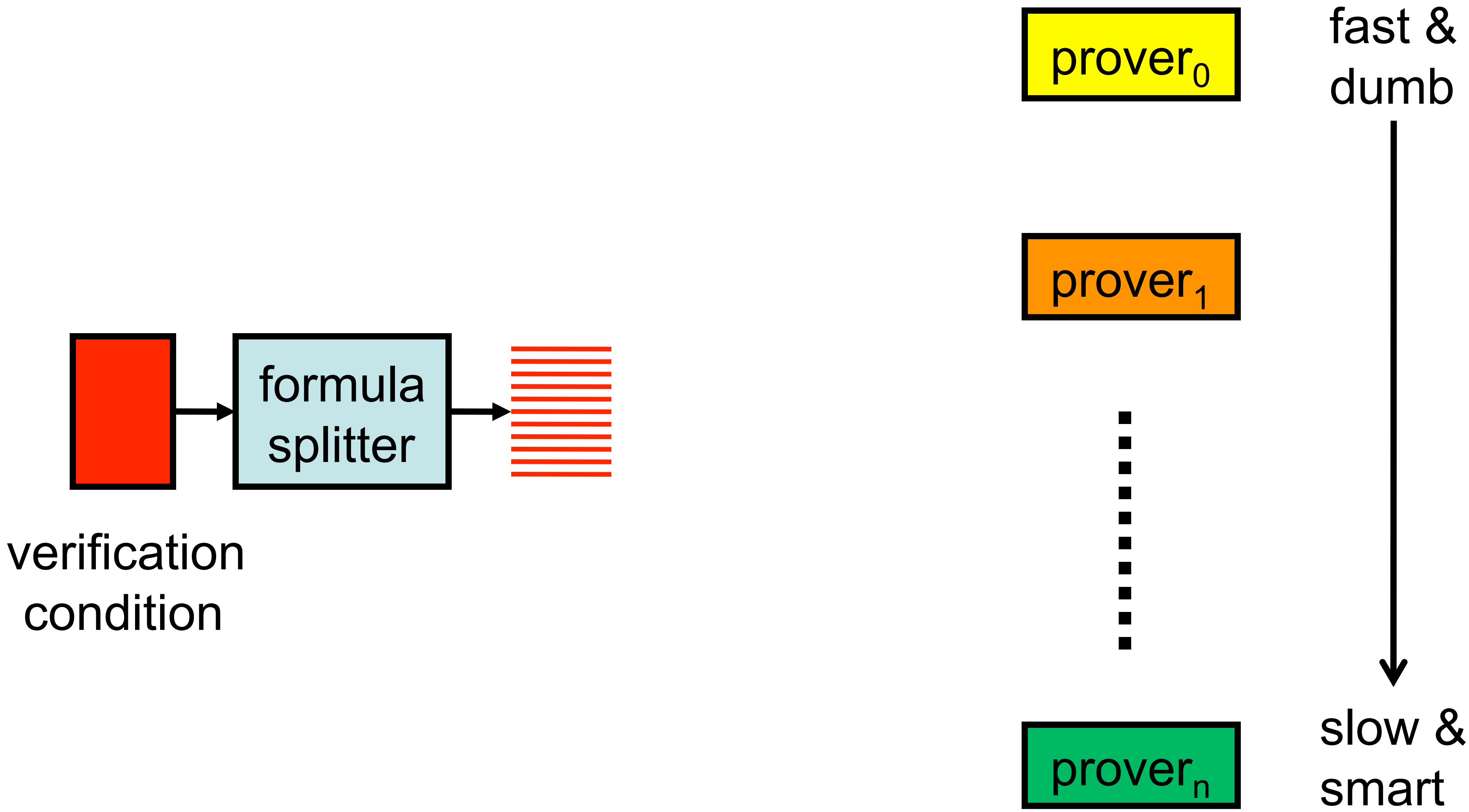
How to prove?

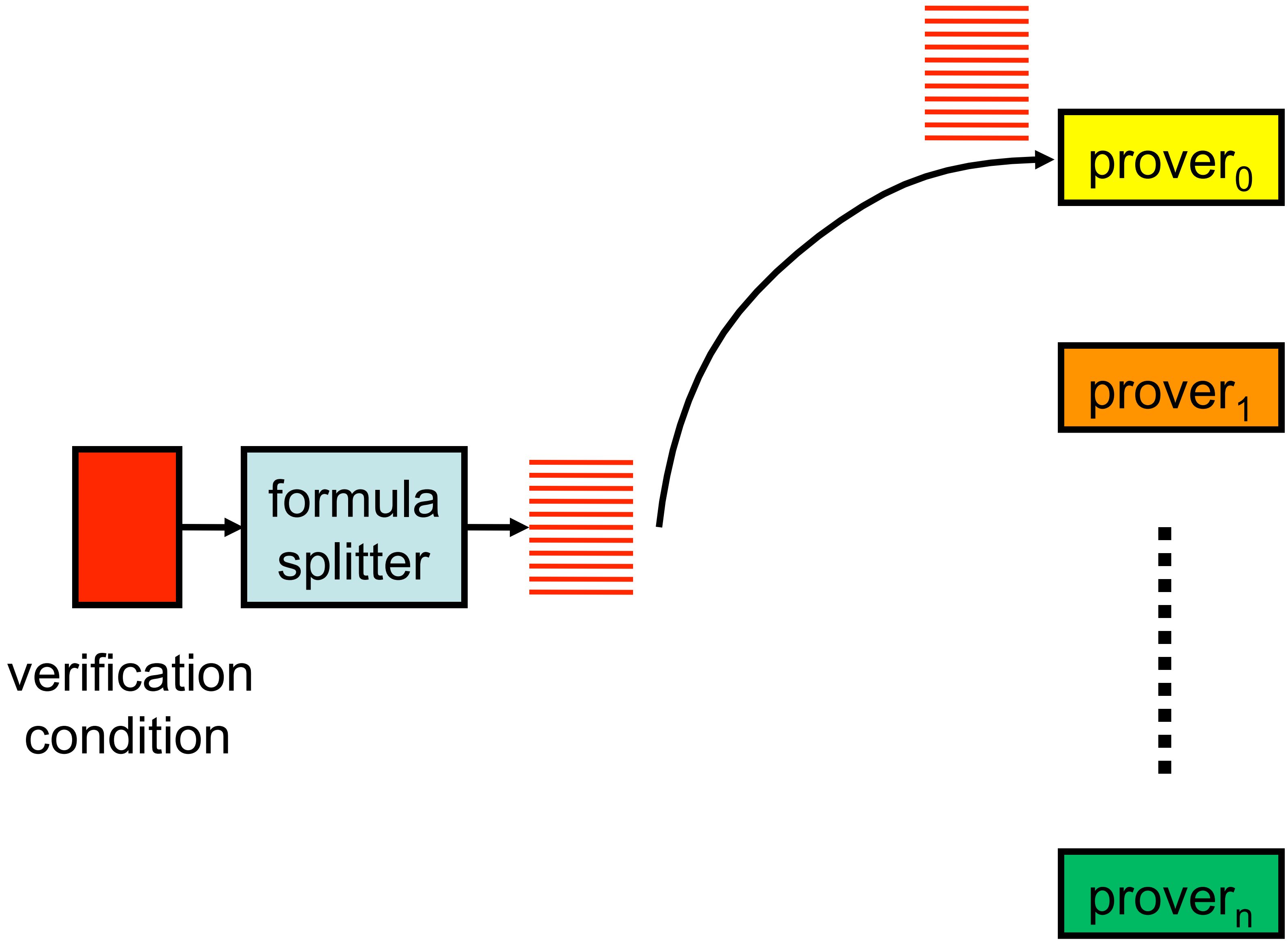
Available Provers

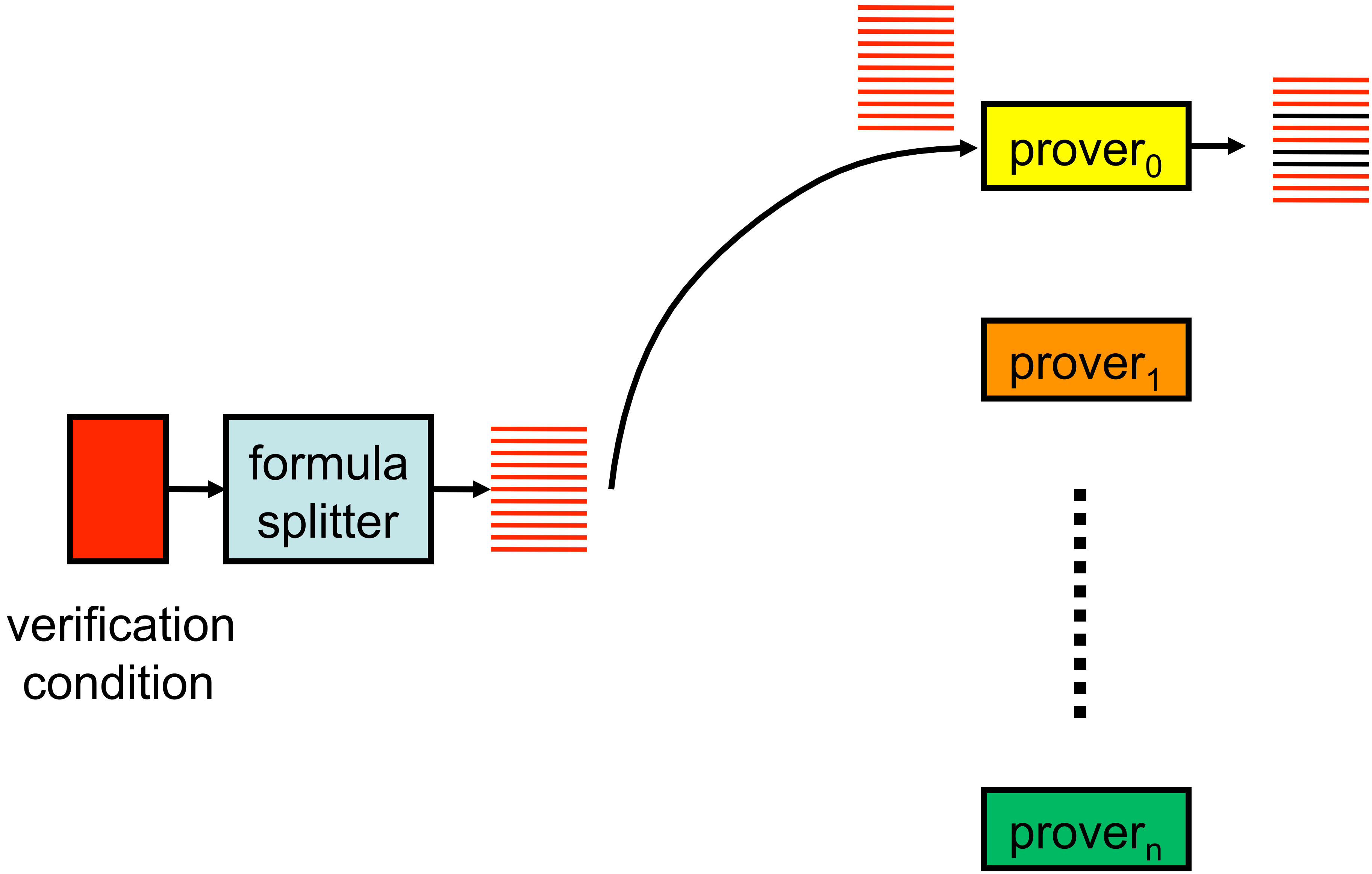
- Syntactic provers
- Nelson-Oppen provers
- Resolution-based provers
- Decision procedures
  - Monadic second-order logic
  - BAPA
- Proof assistants

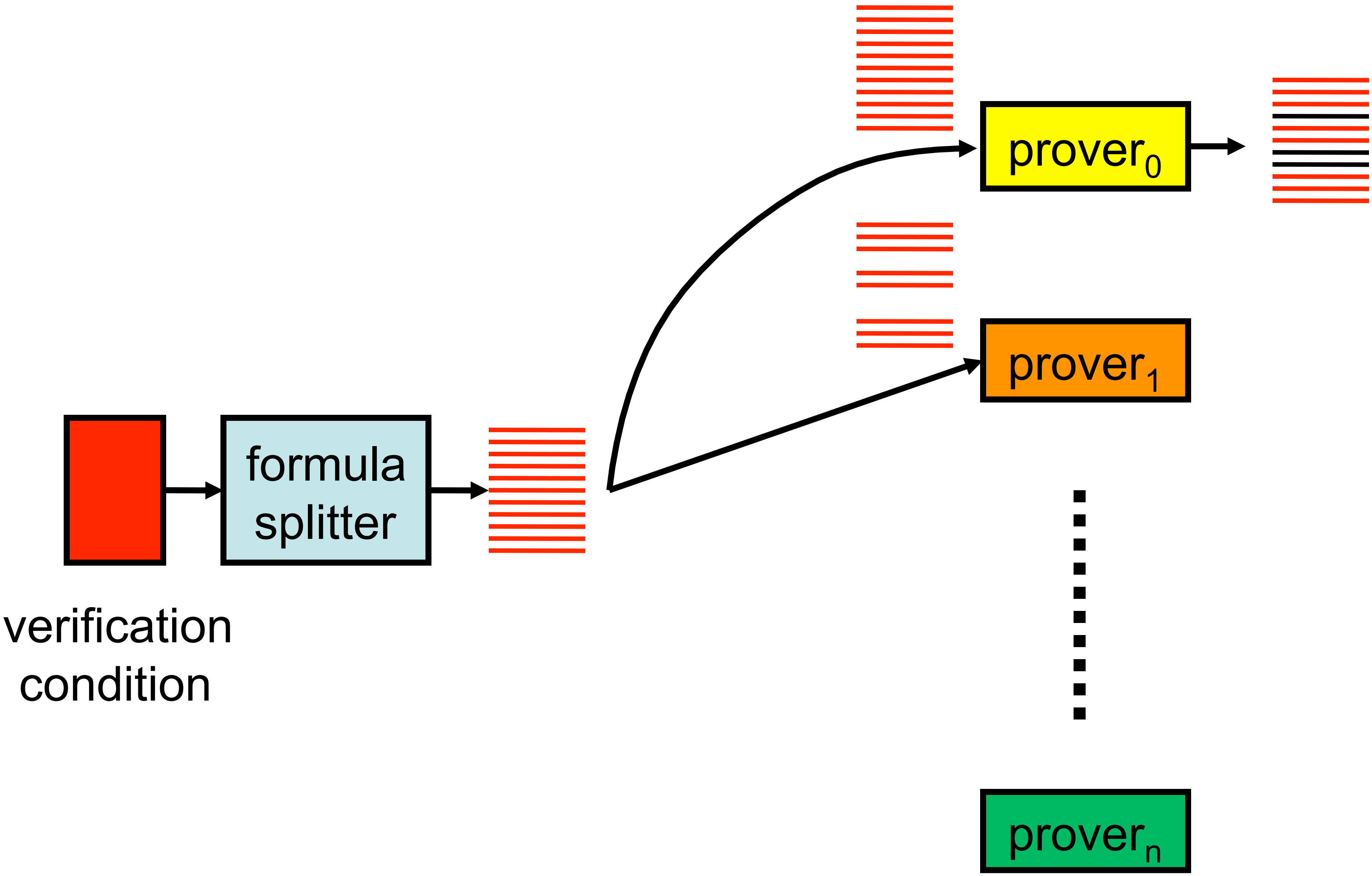


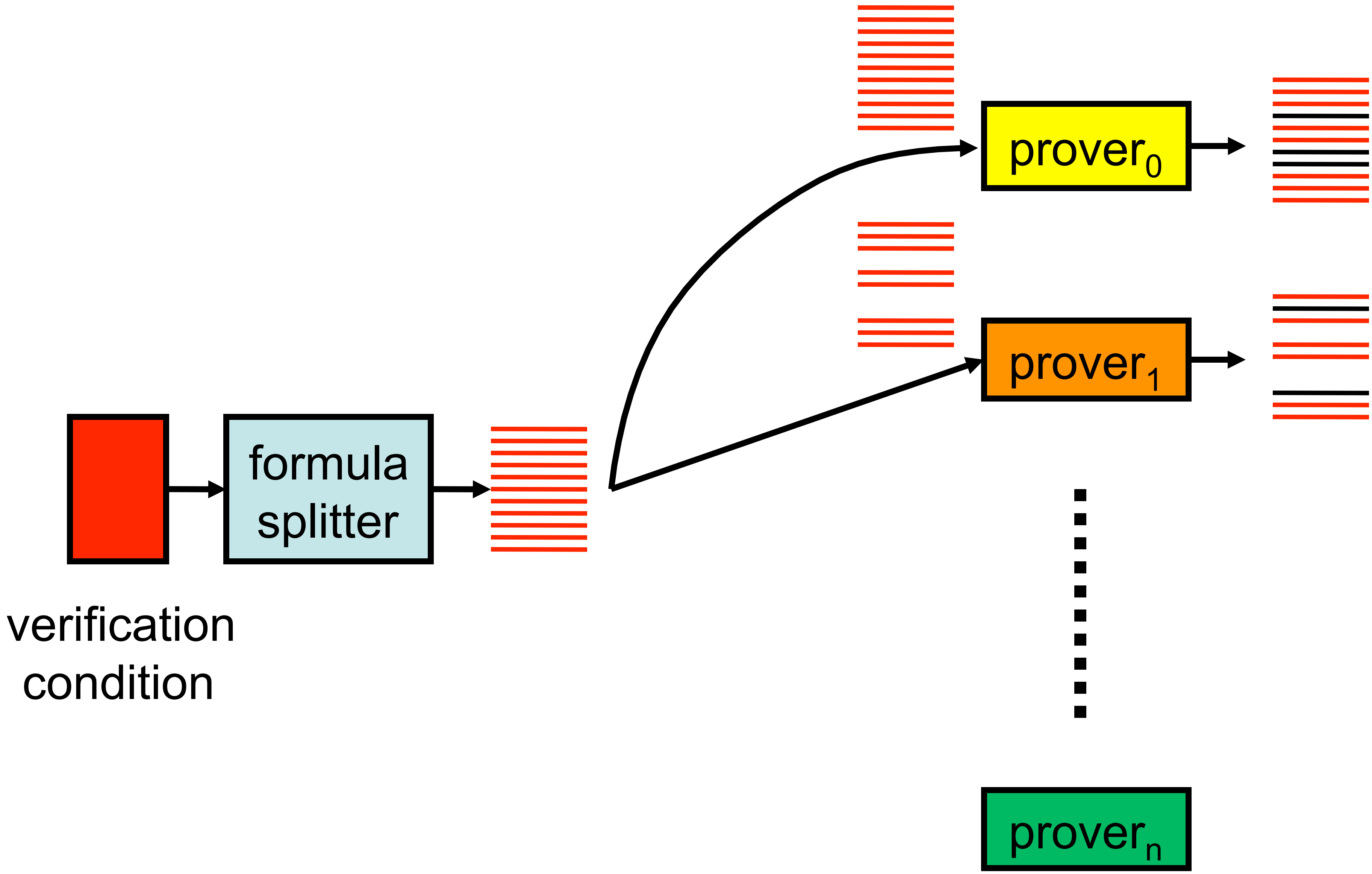

prover0
prover1
provern
fast & dumb
slow & smart
verification condition

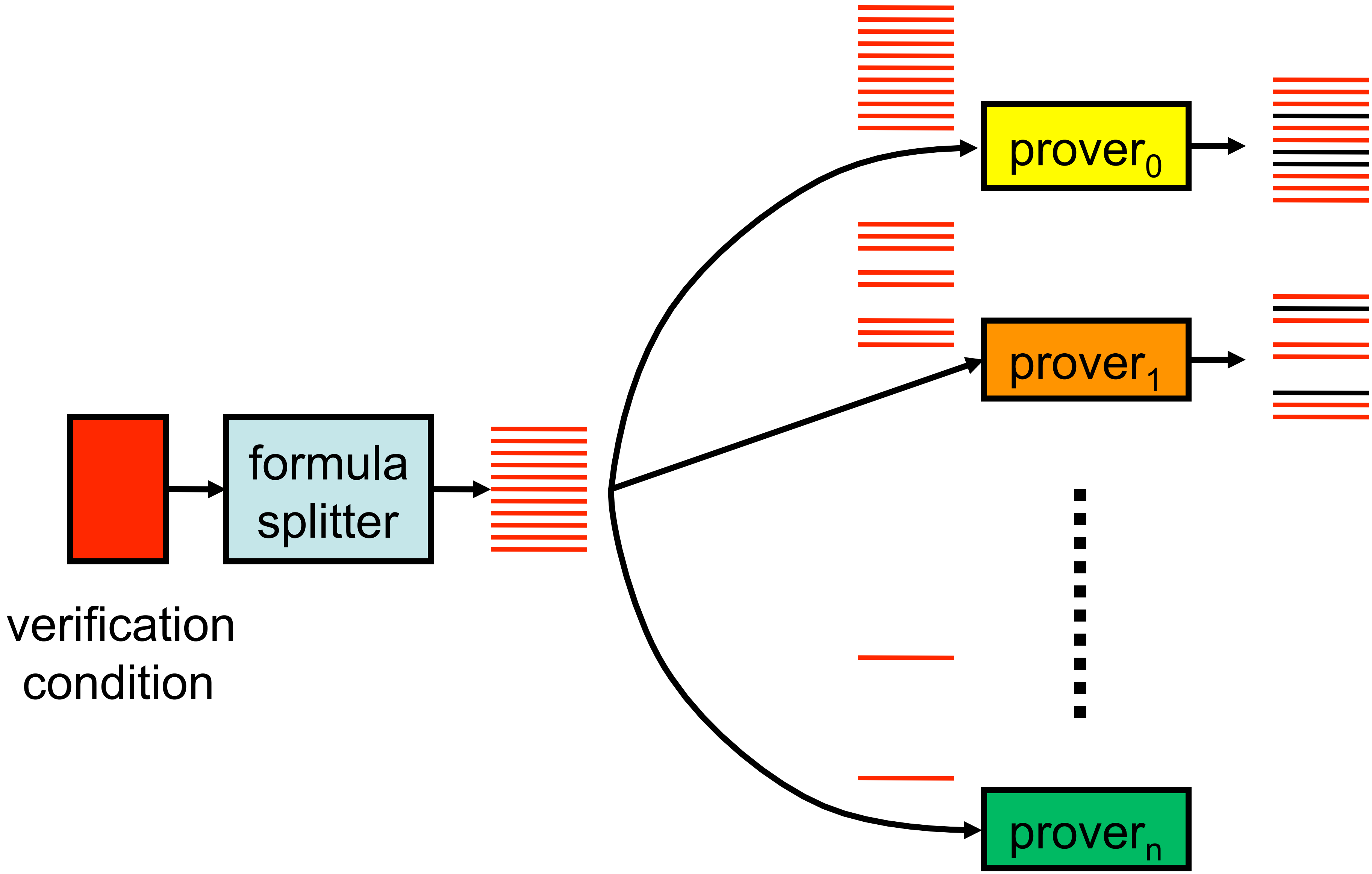

formula splitter
verification condition
$prover_0$
$prover_1$
$prover_n$
fast & dumb
slow & smart

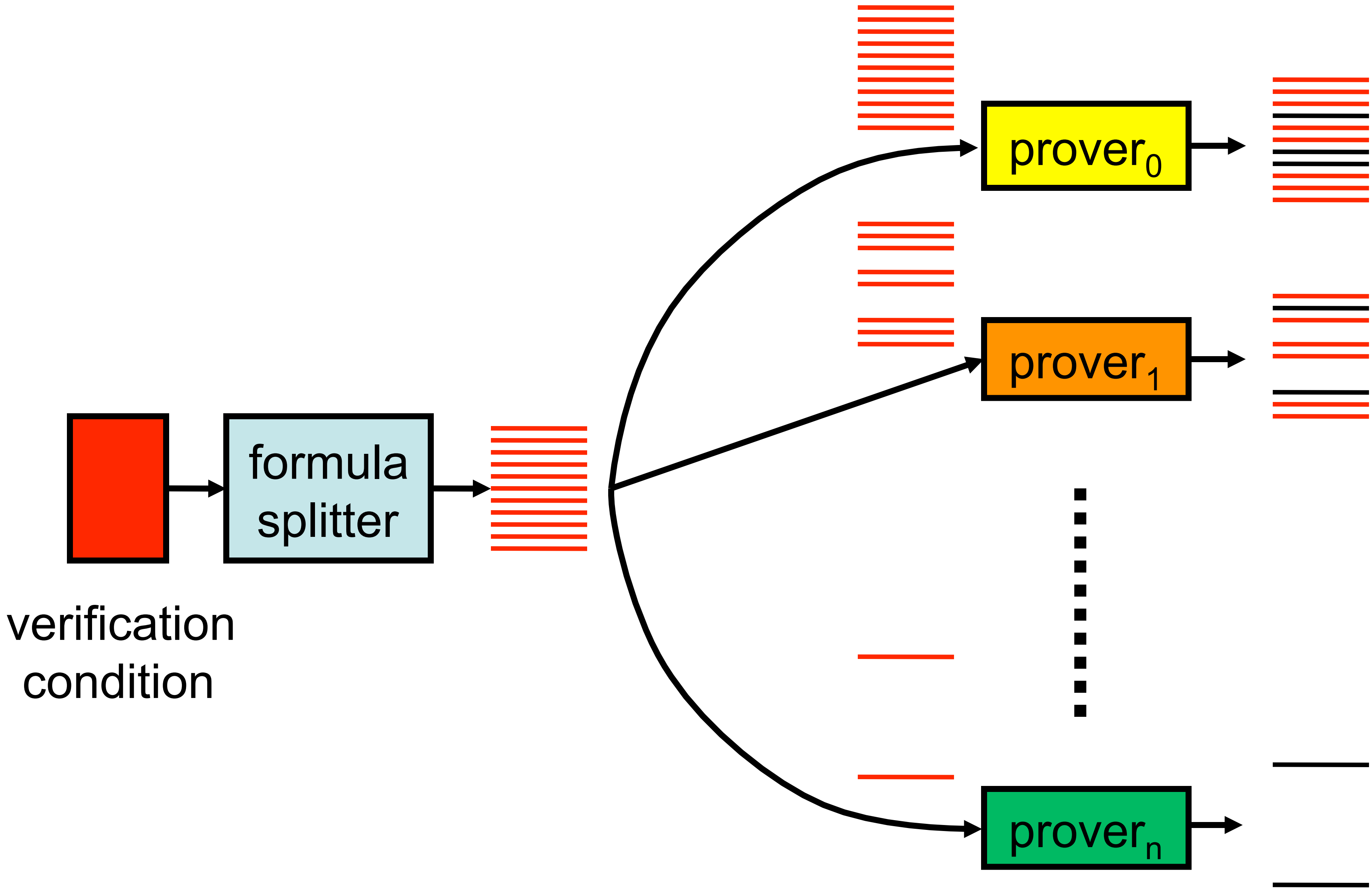

formula
splitter
verification
condition
prover0
prover1
proverN

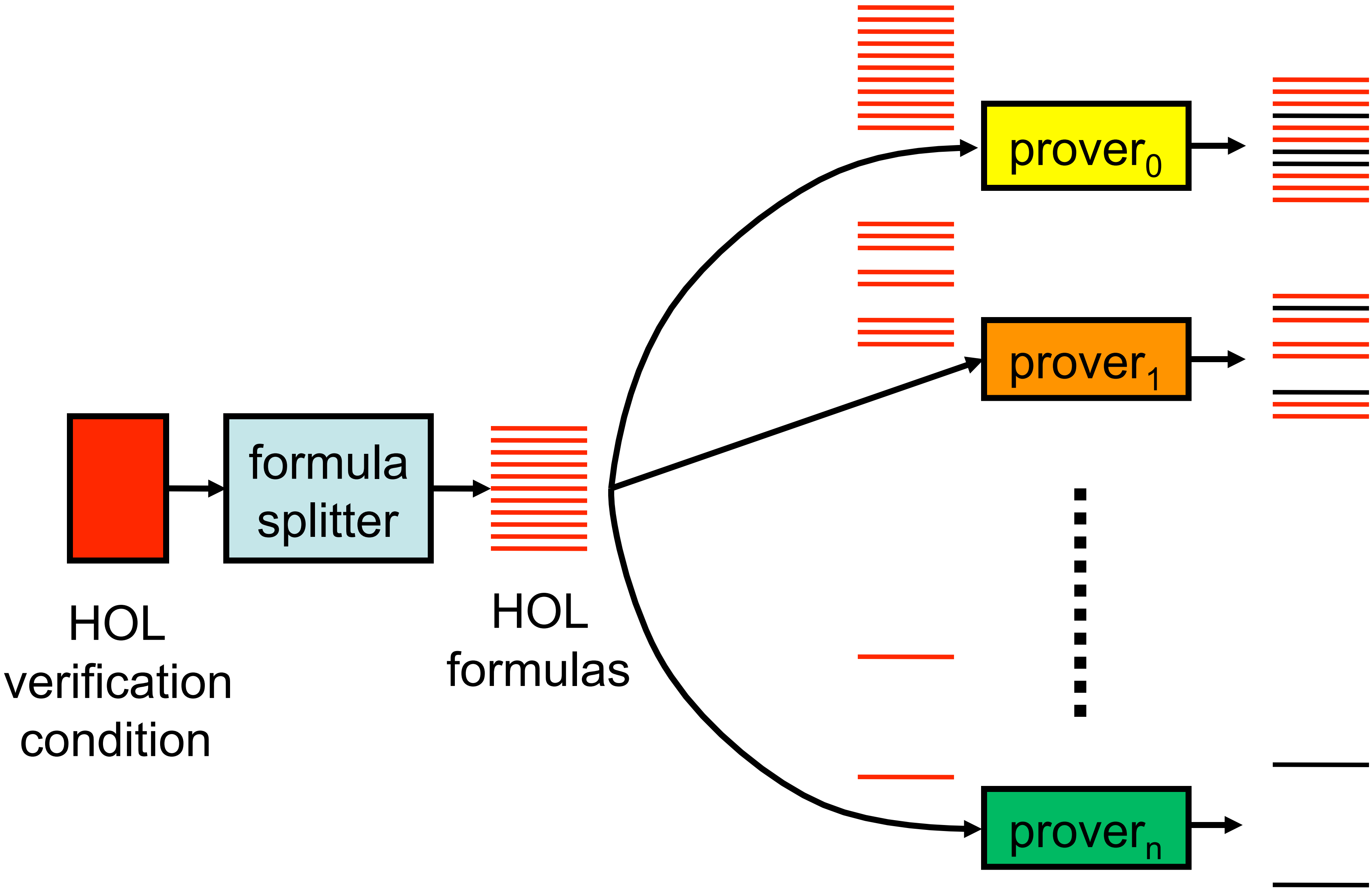

formula splitter
verification condition
prover$_0$
prover$_1$
prover$_n$

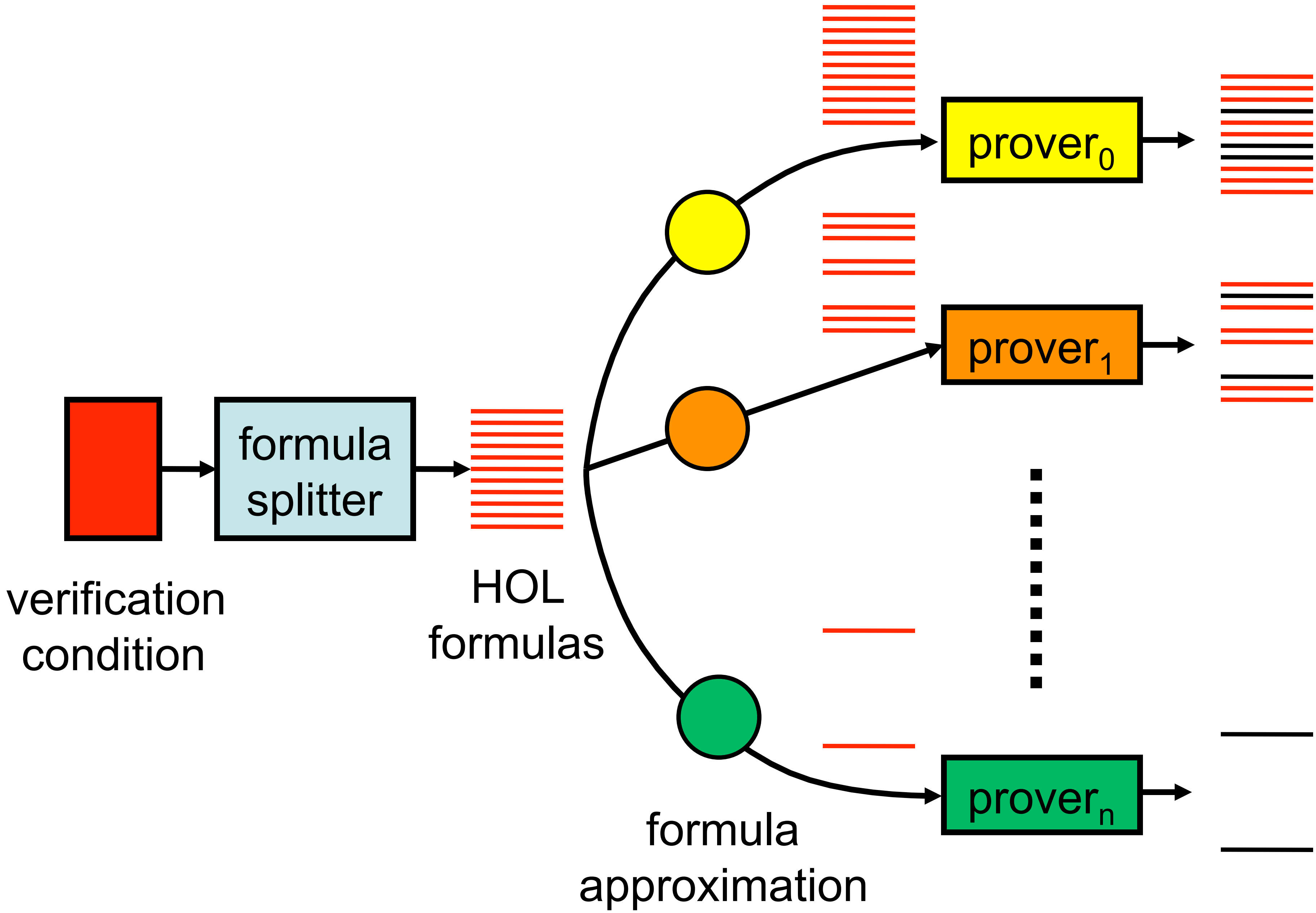

formula splitter
verification condition
prover0
prover1
provern

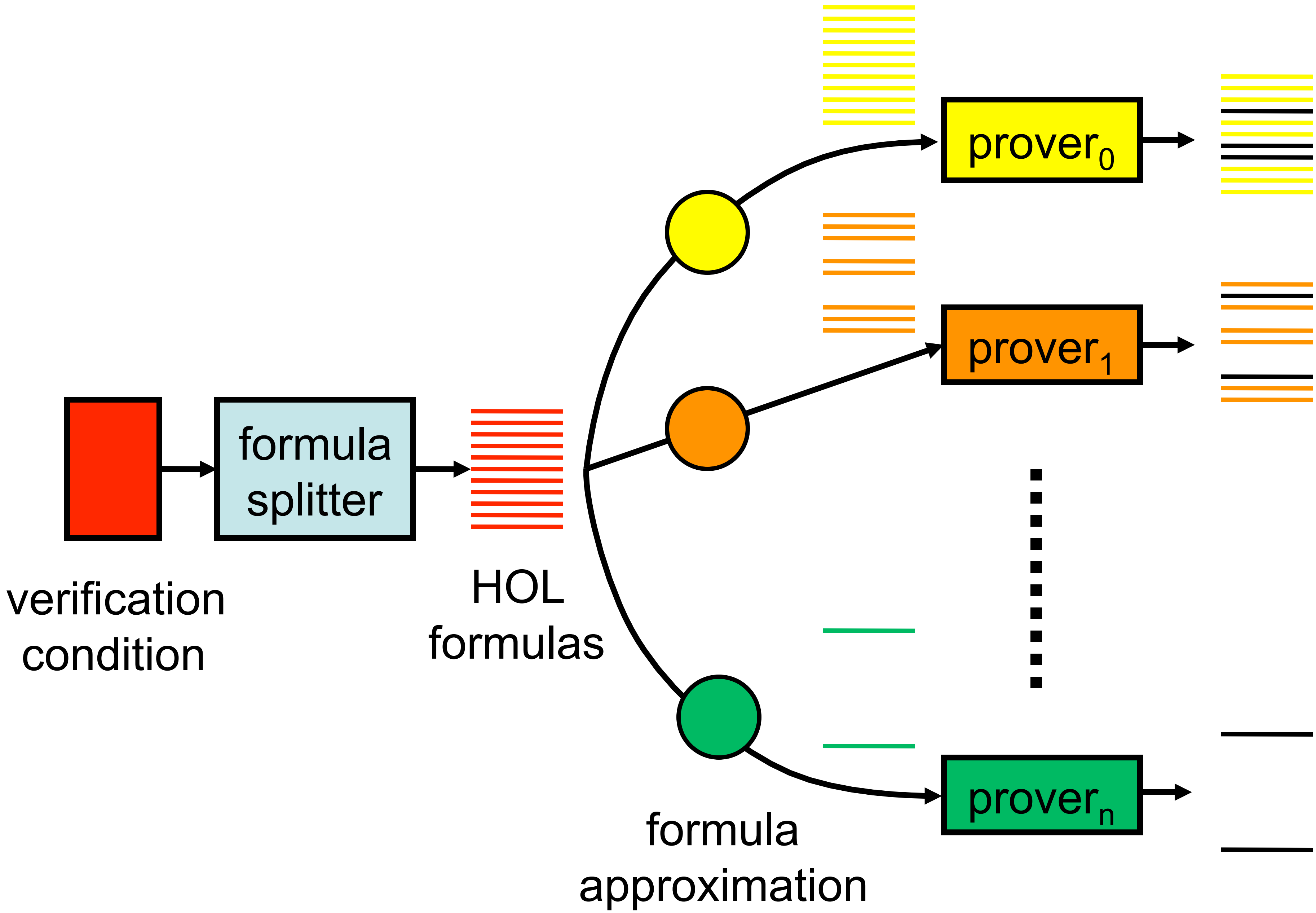

formula splitter
verification condition
$prover_0$
$prover_1$
$prover_n$


formula splitter
verification condition
prover0
prover1
provern


formula
splitter
verification
condition
prover0
prover1
provern

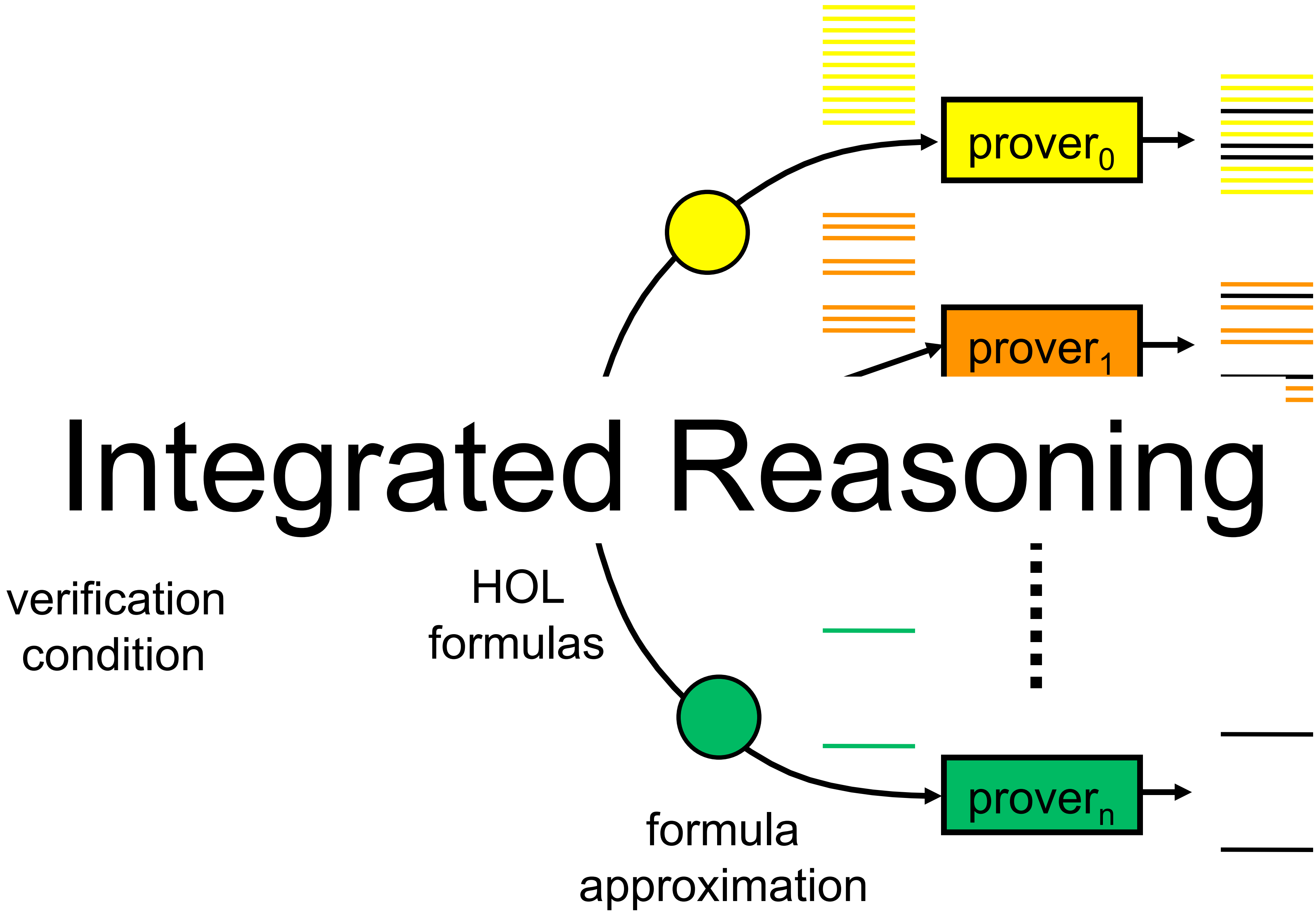

formula
splitter
HOL
verification
condition
HOL
formulas
prover0
prover1
provern


formula
splitter
verification
condition
HOL
formulas
formula
approximation
prover$_0$
prover$_1$
prover$_n$

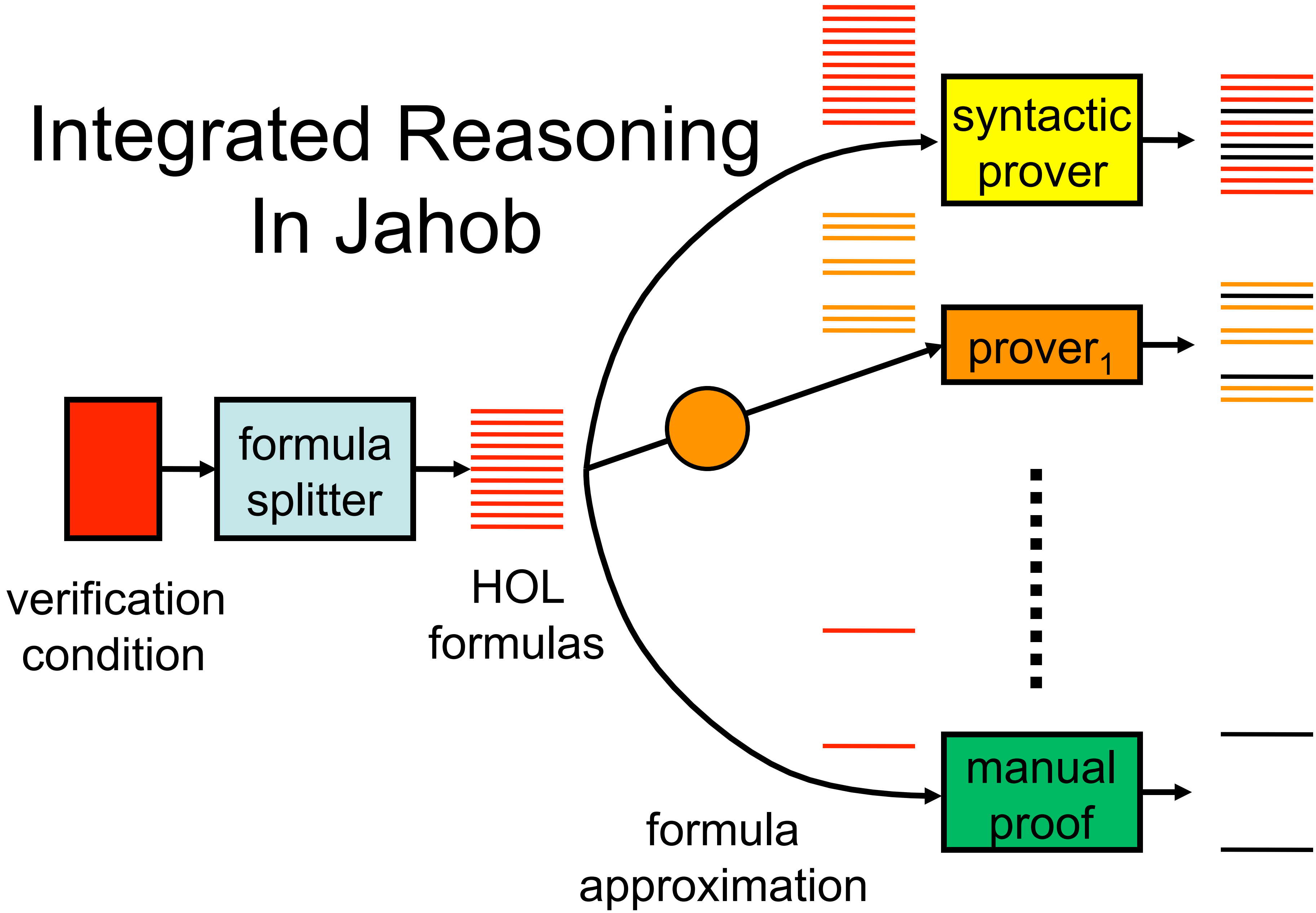


formula splitter
verification condition
HOL formulas
prover$_0$
prover$_1$
prover$_n$
formula approximation

Integrated Reasoning



Integrated Reasoning In Jahob



Formula Splitting

- Transform verification condition into equivalent conjunction of smaller formulas

$$A \rightarrow G_1 \wedge G_2 \quad \Rightarrow \quad A \rightarrow G_1,\ A \rightarrow G_2$$

$$A \rightarrow (B \rightarrow G^{[p]})^{[q]} \quad \Rightarrow \quad (A \wedge B^{[q]}) \rightarrow G^{[pq]}$$

$$A \rightarrow \forall x.G \quad \Rightarrow \quad A \rightarrow G[x:=x_{fresh}]$$

- Enables application of different solvers to solve different parts of verification condition

Formula Approximation

- Feed *any* formula to *any* prover or decision procedure
- Approximate given formula with a stronger formula in appropriate logic subset
- Translate constructs where possible
  - Transform set expressions to predicates
  - Apply beta-reduction to lambda expressions
  - Rewrite tuples into elements
  - Apply extensionality
- Approximate where necessary
  - Descend formula recursively
  - Approximate inexpressible subformulas according to arity

Formula Approximation Rules

$\alpha : (0,1) \times C$

$\alpha^p(f_1 \wedge f_2) \equiv \alpha^p(f_1) \wedge \alpha^p(f_2)$

$\alpha^p(f_1 \vee f_2) \equiv \alpha^p(f_1) \vee \alpha^p(f_2)$

$\alpha^p(\neg f) \equiv \neg \alpha^{\neg p}(f)$

$\alpha^p(\forall x.f) \equiv \forall x.\alpha^p(f)$

$\alpha^p(\exists x.f) \equiv \exists x.\alpha^p(f)$

$\alpha^0(f) \equiv$ false, for f not representable in C

$\alpha^1(f) \equiv$ true, for f not representable in C

$\alpha^p(f) \equiv$ e, for f directly representable in C as e

Formula Approximation Rules

$\alpha : (0,1) \times C$

$\alpha^p(f_1 \wedge f_2) \equiv \alpha^p(f_1) \wedge \alpha^p(f_2)$

$\alpha^p(f_1 \vee f_2) \equiv \alpha^p(f_1) \vee \alpha^p(f_2)$

$\alpha^p(\neg f) \equiv \neg \alpha^{\neg p}(f)$

$\alpha^p(\forall x.f) \equiv \forall x.\alpha^p(f)$

$\alpha^p(\exists x.f) \equiv \exists x.\alpha^p(f)$

$\alpha^0(f) \equiv$ false, for f not representable in C

$\alpha^1(f) \equiv$ true, for f not representable in C

$\alpha^p(f) \equiv$ e, for f directly representable in C as e

Formula Approximation Rules

$\alpha : (0,1) \times C$

$\alpha^p(f_1 \wedge f_2) \equiv \alpha^p(f_1) \wedge \alpha^p(f_2)$

$\alpha^p(f_1 \vee f_2) \equiv \alpha^p(f_1) \vee \alpha^p(f_2)$

$\alpha^p(\neg f) \equiv \neg \alpha^{\neg p}(f)$

$\alpha^p(\forall x.f) \equiv \forall x.\alpha^p(f)$

$\alpha^p(\exists x.f) \equiv \exists x.\alpha^p(f)$

$\alpha^0(f) \equiv$ false, for f not representable in C

$\alpha^1(f) \equiv$ true, for f not representable in C

$\alpha^p(f) \equiv$ e, for f directly representable in C as e

Formula Approximation Rules

$\alpha : (0,1) \times C$

$\alpha^p(f_1 \wedge f_2)$	$\equiv \alpha^p(f_1) \wedge \alpha^p(f_2)$
$\alpha^p(f_1 \vee f_2)$	$\equiv \alpha^p(f_1) \vee \alpha^p(f_2)$
$\alpha^p(\neg f)$	$\equiv \neg \alpha^{\neg p}(f)$
$\alpha^p(\forall x.f)$	$\equiv \forall x.\alpha^p(f)$
$\alpha^p(\exists x.f)$	$\equiv \exists x.\alpha^p(f)$
$\alpha^0(f)$	$\equiv$ false, for f not representable in C
$\alpha^1(f)$	$\equiv$ true, for f not representable in C
$\alpha^p(f)$	$\equiv$ e, for f directly representable in C as e

Why Does Formula Approximation Work?

- Formula splitting preserves assumptions
- Proof of a given subformula depends only on a subset of assumptions

- Some HOL formulas directly translatable into formulas in simpler logics

Dealing With Proof Complexity

- Proof complexity issues
  - Provers overwhelmed by assumptions
  - Proof of a single subformula requires expertise of multiple provers
- Note statements

  //: note f: "…" from f_0, f_1, … f_n;
  - Tell provers which assumptions to use
  - Introduce intermediate lemmas into verification conditions
- In effect, developer guides proof decomposition

Note Example (get)

```
public Object get(Object k0)
/*:   requires "init ∧ k0 ≠ null"
      ensures "(result ≠ null → (k0, result) ∈ content) ∧
          (result = null → ¬(∃v. (k0, v) ∈ content))" */
{
      int hc = compute_hash(k0);
      Node current = table[hc];
      //: note ThisProps: "this ∈ old alloc ∧ this ∈ Hashtable ∧this.init";
      //: note HCProps: "0 ≤ hc ∧hc < table.length ∧ hc = hash key (table.length)";
      /*: note InCurrent: "∀v. ((k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent)"
            from ContentDef, HCProps, Coherence, ThisProps, InCurrent; */
      while /*: inv "∀v. (k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent" */
          (current != null) {
          if (current.key == k0) { return current.value; }
          current = current.next;
      }
       return null;
}
```

Note Example (get)

```
public Object get(Object k0)
/*:   requires "init ∧ k0 ≠ null"
      ensures "(result ≠ null → (k0, result) ∈ content) ∧
           (result = null → ¬(∃v. (k0, v) ∈ content))" */
{
      int hc = compute_hash(k0);
      Node current = table[hc];
      //: note ThisProps: “this ∈ old alloc ∧ this ∈ Hashtable ∧this.init”;
      //: note HCProps: “0 ≤ hc ∧hc < table.length ∧ hc = hash key (table.length)”;
      /*: note InCurrent: “∀v. ((k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent)”
            from ContentDef, HCProps, Coherence, ThisProps, InCurrent; */
      while /*: inv "∀v. (k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent” */
           (current != null) {
           if (current.key == k0) { return current.value; }
           current = current.next;
      }
       return null;
}
```

Label known facts

Note Example (get)

```
public Object get(Object k0)
/*:   requires "init ∧ k0 ≠ null”
      ensures "(result ≠ null → (k0, result) ∈ content) ∧
          (result = null → ¬(∃v. (k0, v) ∈ content))" */
{
      int hc = compute_hash(k0);
      Node current = table[hc];
      //: note ThisProps: “this ∈ old alloc ∧ this ∈ Hashtable ∧this.init”;
      //: note HCProps: “0 ≤ hc ∧hc < table.length ∧ hc = hash key (table.length)”;
      /*: note InCurrent: “∀v. ((k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent)”
          from ContentDef, HCProps, Coherence, ThisProps, InCurrent; */
      while /*: inv "∀v. (k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent” */
          (current != null) {
          if (current.key == k0) { return current.value; }
          current = current.next;
      }
       return null;
}
```

State proof goal
(intermediate fact or final goal)

Note Example (get)

```
public Object get(Object k0)
/*:   requires "init ∧ k0 ≠ null"
      ensures "(result ≠ null → (k0, result) ∈ content) ∧
          (result = null → ¬(∃v. (k0, v) ∈ content))" */
{
      int hc = compute_hash(k0);
      Node current = table[hc];
      //: note ThisProps: "this ∈ old alloc ∧ this ∈ Hashtable ∧this.init";
      //: note HCProps: "0 ≤ hc ∧hc < table.length ∧ hc = hash key (table.length)";
      /*: note InCurrent: "∀v. ((k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent)"
          from ContentDef, HCProps, Coherence, ThisProps, InCurrent; */
      while /*: inv "∀v. (k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent" */
          (current != null) {
          if (current.key == k0) { return current.value; }
          current = current.next;
      }
       return null;
}
```

Identify a set of known facts

Note Example (get)

```
public Object get(Object k0)
/*:   requires "init ∧ k0 ≠ null”
      ensures "(result ≠ null → (k0, result) ∈ content) ∧
          (result = null → ¬(∃v. (k0, v) ∈ content))" */
{
      int hc = compute_hash(k0);
      Node current = table[hc];
      //: note ThisProps: “this ∈ old alloc ∧ this ∈ Hashtable ∧this.init”;
      //: note HCProps: “0 ≤ hc ∧hc < table.length ∧ hc = hash key (table.length)”;
      /*: note InCurrent: “∀v. ((k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent)”
          from ContentDef, HCProps, Coherence, ThisProps, InCurrent; */
      while /*: inv "∀v. (k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent” */
          (current != null) {
          if (current.key == k0) { return current.value; }
          current = current.next;
      }
       return null;
}
```

- Instruct prover to use set to prove goal
- Proven fact inserted into assumption base

Note Example (get)

```
public Object get(Object k0)
/*:    requires "init ∧ k0 ≠ null”
       ensures "(result ≠ null → (k0, result) ∈ content) ∧
            (result = null → ¬(∃v. (k0, v) ∈ content))" */
{
       int hc = compute_hash(k0);
       Node current = table[hc];
       //: note ThisProps: “this ∈ old alloc ∧ this ∈ Hashtable ∧this.init”;
       //: note HCProps: “0 ≤ hc ∧hc < table.length ∧ hc = hash key (table.length)”;
       /*: note InCurrent: “∀v. ((k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent)”
             from ContentDef, HCProps, Coherence, ThisProps, InCurrent; */
       while /*: inv "∀v. (k0, v) ∈ content) = ((k0, v) ∈ current.bucketContent” */
            (current != null) {
            if (current.key == k0) { return current.value; }
            current = current.next;
       }
        return null;
}
```

Proved trivially by syntactic prover

Constructs for Directly Controlling Proof

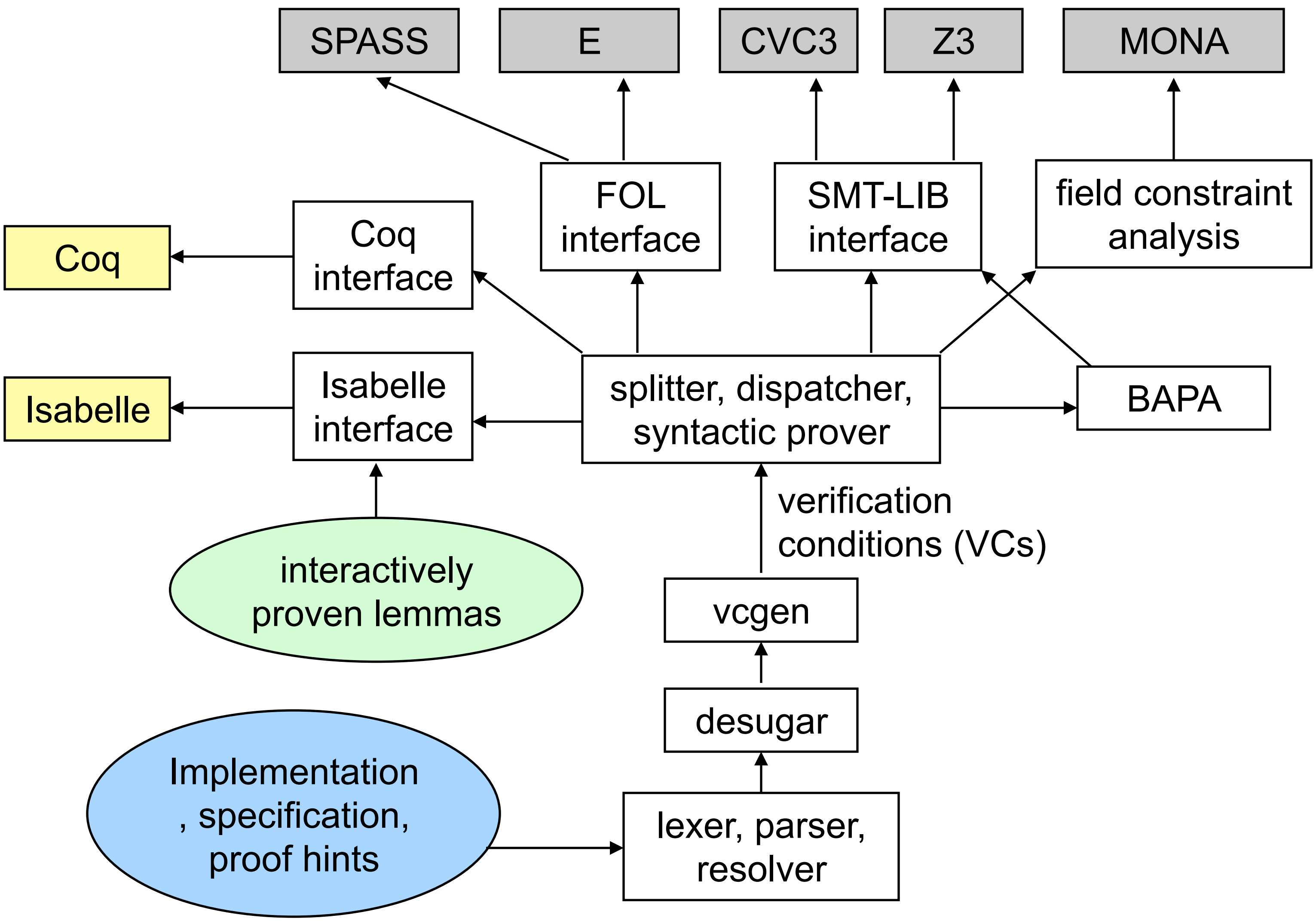
- havoc $x_0,\ldots,x_n$ suchThat f

  Instantiates $\exists x_0,\ldots,x_n.f$
- pickAny $x_0,\ldots,x_n$ in (c; note g)

  Prove $\forall x_0,\ldots,x_n.g$
- assuming f in (c_{pure}; note g)

  Prove $f \rightarrow g$
- Desugar into standard guarded commands

Jahob System Diagram



Experimental Results

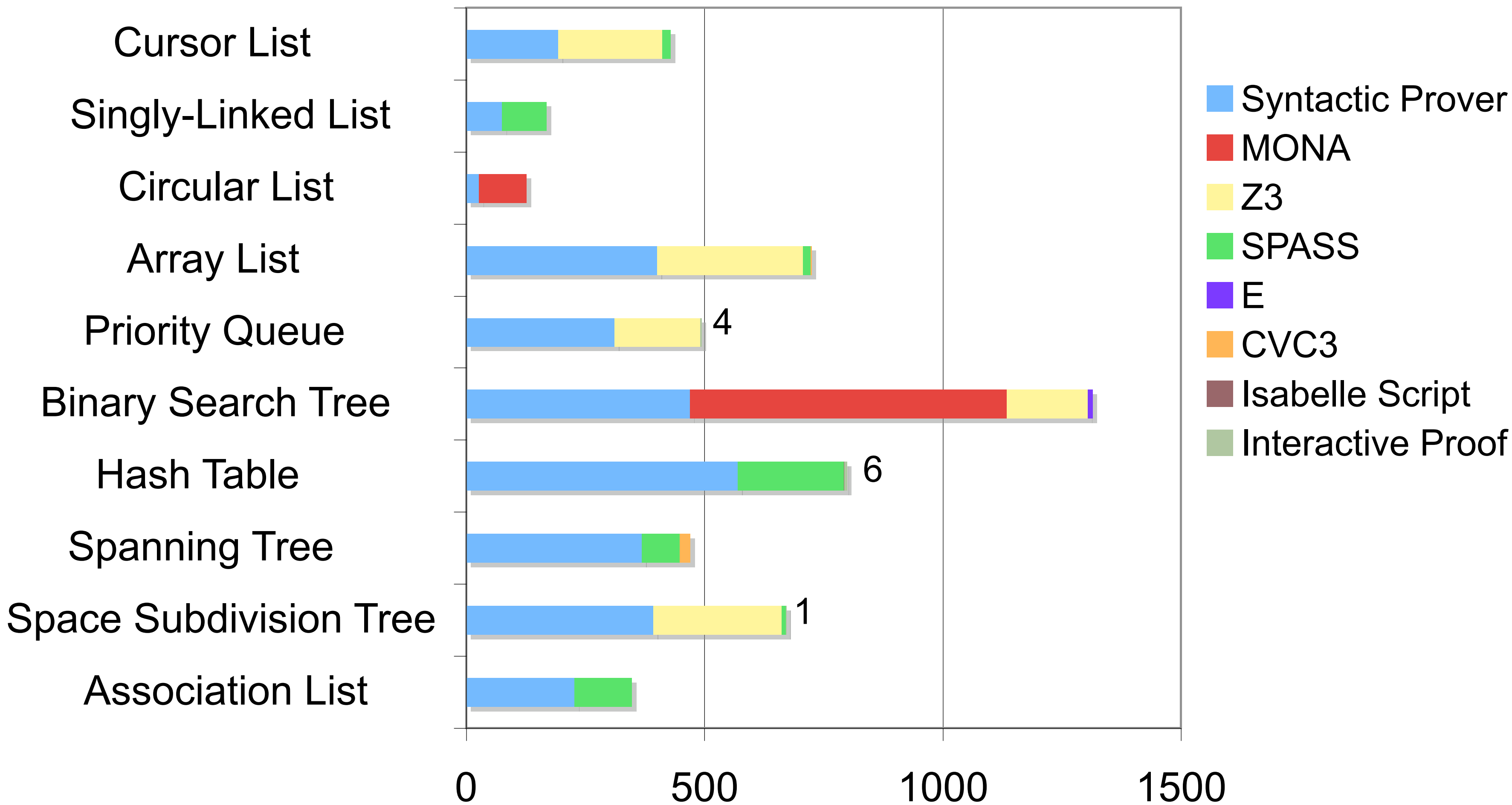
Verified Data Structures

- Cursor List
- Singly-Linked List
- Circular List
- Array List
- Priority Queue
- Binary Search Tree
- Hash Table
- Spanning Tree
- Space Subdivision Tree
- Association List

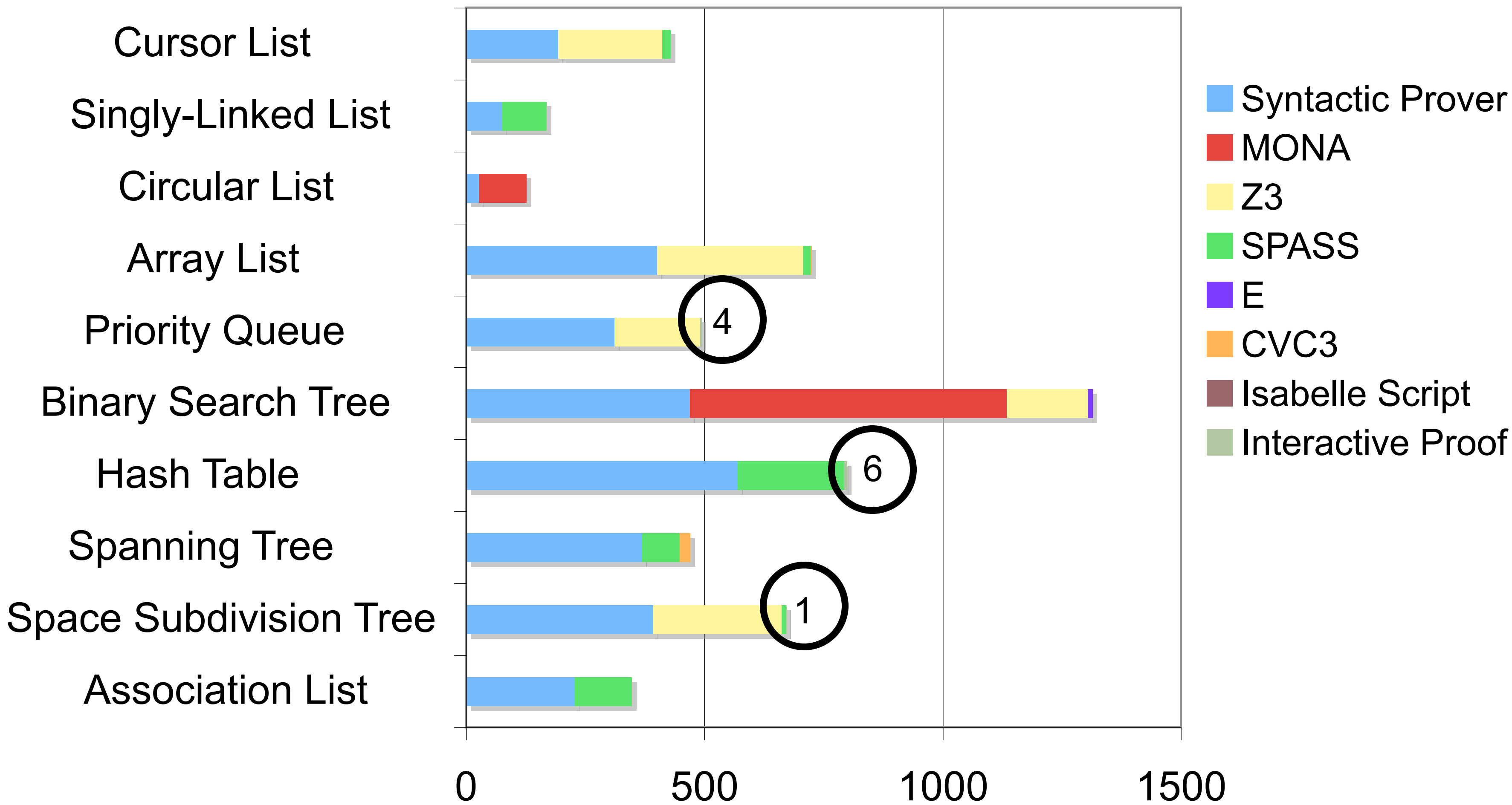
Provers

- Syntactic prover
- MONA
- Z3
- SPASS
- E
- CVC3
- Isabelle (simplifier)
- Isabelle proof assistant

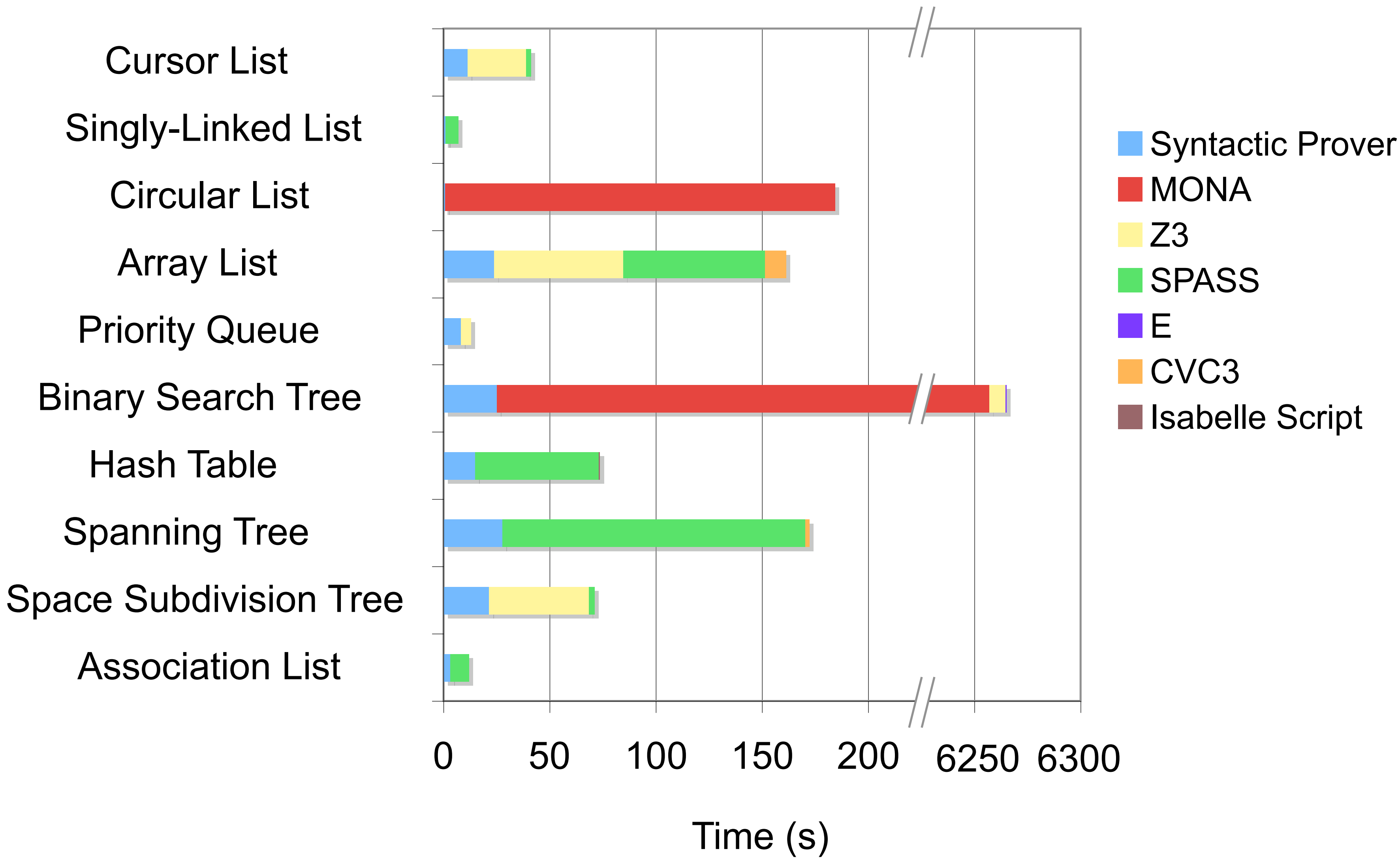
Formulas Verified



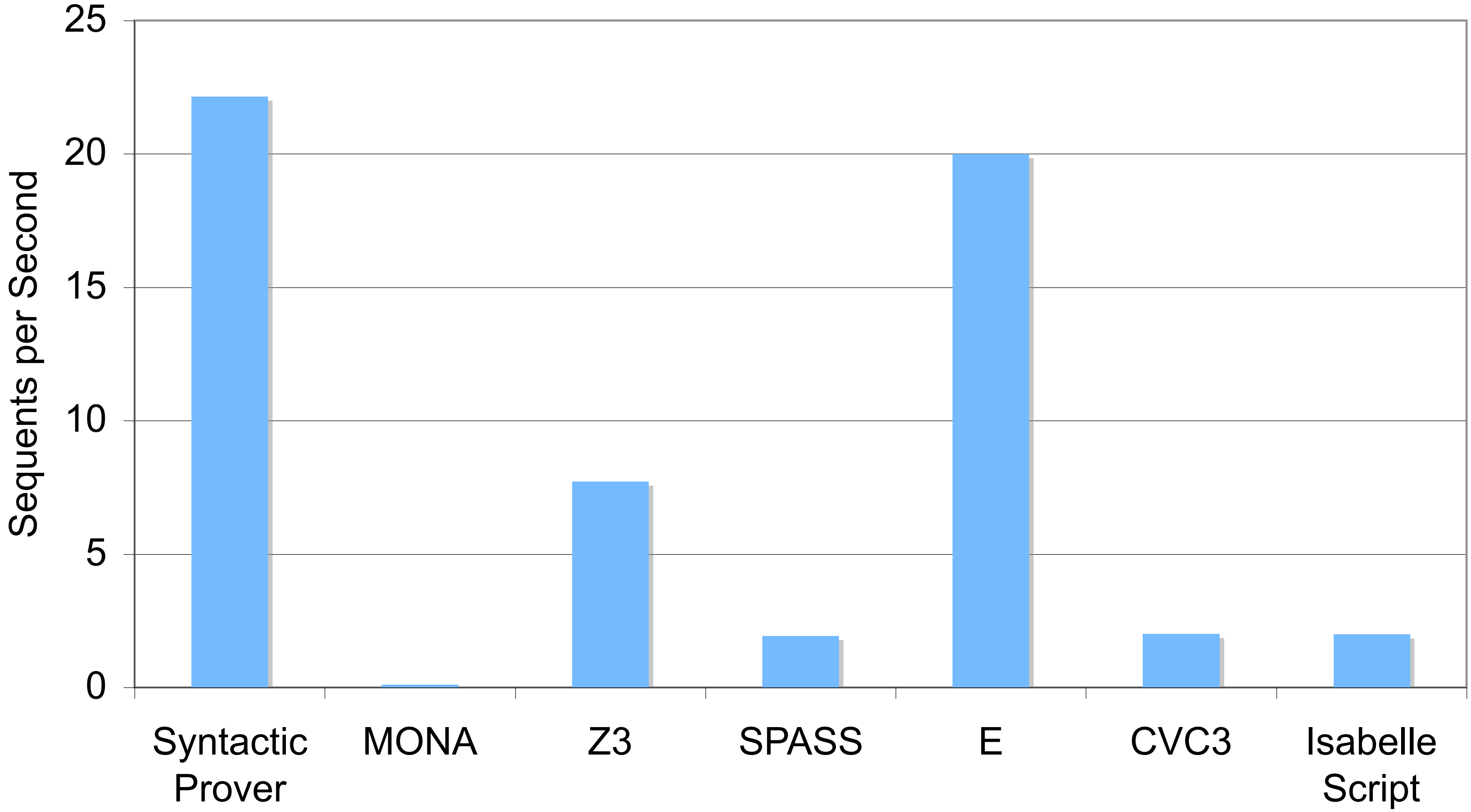
Formulas Verified



Verification Time



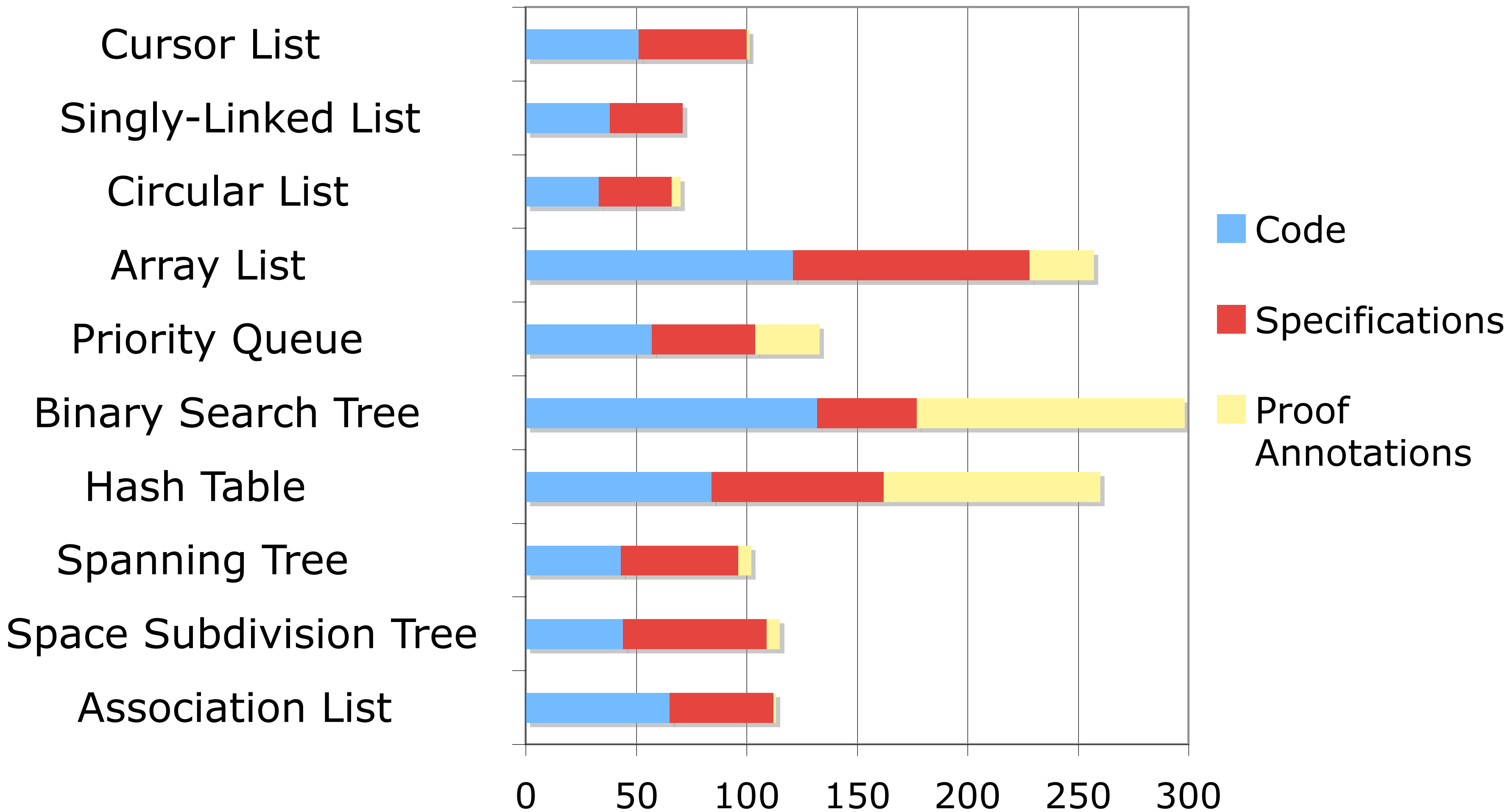
Prover Efficiency



Manual Proofs

- Space Subdivision Tree
  - 1 proof, 4 lines of proof script
  - 2 case splits, 1 quantifier instantiation
- Priority Queue
  - 2 proofs, 78 + 4 lines
  - Inductive proof with modulo arithmetic
- Hashtable
  - 1 proof for add, 7 lines, 3 case splits
  - 4 proofs for remove, 19 + 2 + 2 + 10 lines, case splits

Logical Lines of Code



Observations

- Implementation easier than specification
  (but more current expertise with implementation)
- Easy to prove – callers, observer methods
- Difficult to prove
  - Destructive update (constructors*, add, remove*)
  - Leaf methods (reasoning about concrete + abstract state)
- Incentive to decompose larger methods
- Incentive to reuse existing methods
- Must understand why implementation is correct to obtain proof

Implications

- Verified data structure libraries
- Integrated reasoning in other contexts
- New program analysis techniques
  - Client analyses based on sets and relations (no more reasoning about pointers)
  - More precise data structure analyses
- Semantic commutativity analysis for parallel programs

Related Work: Hob

- Kuncak, Lam, Zee, and Rinard [TSE 2006]; Lam [PhD. Thesis MIT 2007]
- Sets summarize data structure state
- Full functional verification only for data structures with set interface
- Multiple decision procedures (early form of integrated reasoning)

Related Work (cont’d)

Software Verification Tools

- Spec#: Barnett, DeLine, Fähndrich, Leino, and Schulte [J. Obj. Tech. 2004]
- ESC/Modula-3: Detlefs, Leino, Helson, Saxe [TR159 COMPAQ SRC 1998]
- ESC/Java: Flanagan, Leino, Lilibridge, Nelson, Saxe and Stata [PLDI 2002]
- ESC/Java: Chalin, Hurlin, and Kiniry [VSTTE 2005]
- Krakatoa: Filliatre [J. Func. Programming 2003]; Marche, Paulin-Mohring, and Urbain [J. Logic & Alg. Prog. 2003]
- KIV: Balser, Reif, Schellhorn, Stenzel, and Thums [FASE 2000]
- KeY: Ahrendt, Baar, Beckert, Bubel, Giese, Hähnle, Menzel, Mostowski, Roth, Schlager, and Schmitt [Soft. & Sys. Modeling 2005]
- LOOP: van der Berg and Jacobs [TR CSI-R0019 U. Nijmegen 2000]

Related Work (cont'd)

Shape Analysis

- Chong and Rugina [SAS 2003]
- Role analysis: Kuncak, Lam, and Rinard [POPL 2002]
- Grammar-based shape analysis: Lee, Yang, and Ki [ESOP 2005]
- TVLA: Sagiv, Reps, and Wilhelm [TOPLAS 2002]
- Symbolic shape analysis: Podelski and Wies [SAS 2005]
- Guo, Vachharajani, and August [PLDI 2007]

Separation Logic

- Smallfoot: Berdine, Calcagno, and O'Hearn [FMCO 2005]
- Nguyen, David, Qin, and Chin [VMCAI 2007]
- Nguyen and Chin [CAV 2008]

- Yang, Lee, Berdine, Calcagno, Cook, Distefano, and O'Hearn [CAV 2008]

Unrelated Work

Bounded model checking

- Bogor: Robby, Rodríguez, Dwyer and Hatcliff [STTT 2006]
- JACK: Bouali, Gnesi and Larosa [EATCS 1994]
- Forge: Dennis, Chang and Jackson [ISSTA 2006]
- J-Sim: Sobeih, Mahesh, Marinov and Hou [IPDPS 2007]

Testing

- Korat: Boyapati, Khurshid and Marinov [ISSTA 2002]
- TestEra: Khurshid and Marinov [Autom. Soft. Eng. 2004]
- Cute: Sen, Marinov and Agha [FSE 2005]

Conclusions

- Full functional correctness for linked data structure implementations
  - Formula splitting
  - Formula approximation
  - Integrated reasoning
  - Constructs for guiding proofs
- Complete realization of abstract data types
  - Precise, complete, and verified specifications
  - Enables new, more precise and scalable client program analyses

Questions?