The Hob System for Verifying Generalized Data Structure Consistency Properties

Patrick Lam, Viktor Kuncak, Karen Zee, Martin Rinard

MIT CSAIL

Massachusetts Institute of Technology

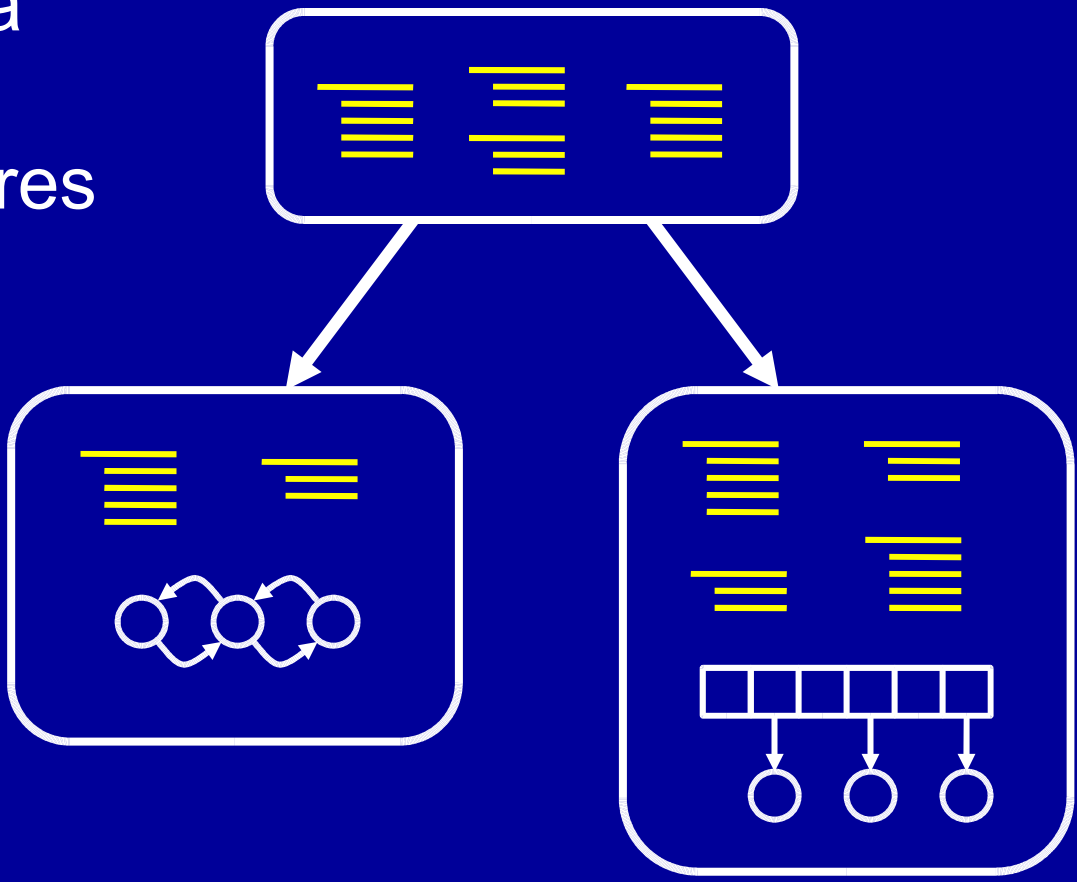
Cambridge, MA 02139

Context

Software System

Composed of modules, with:

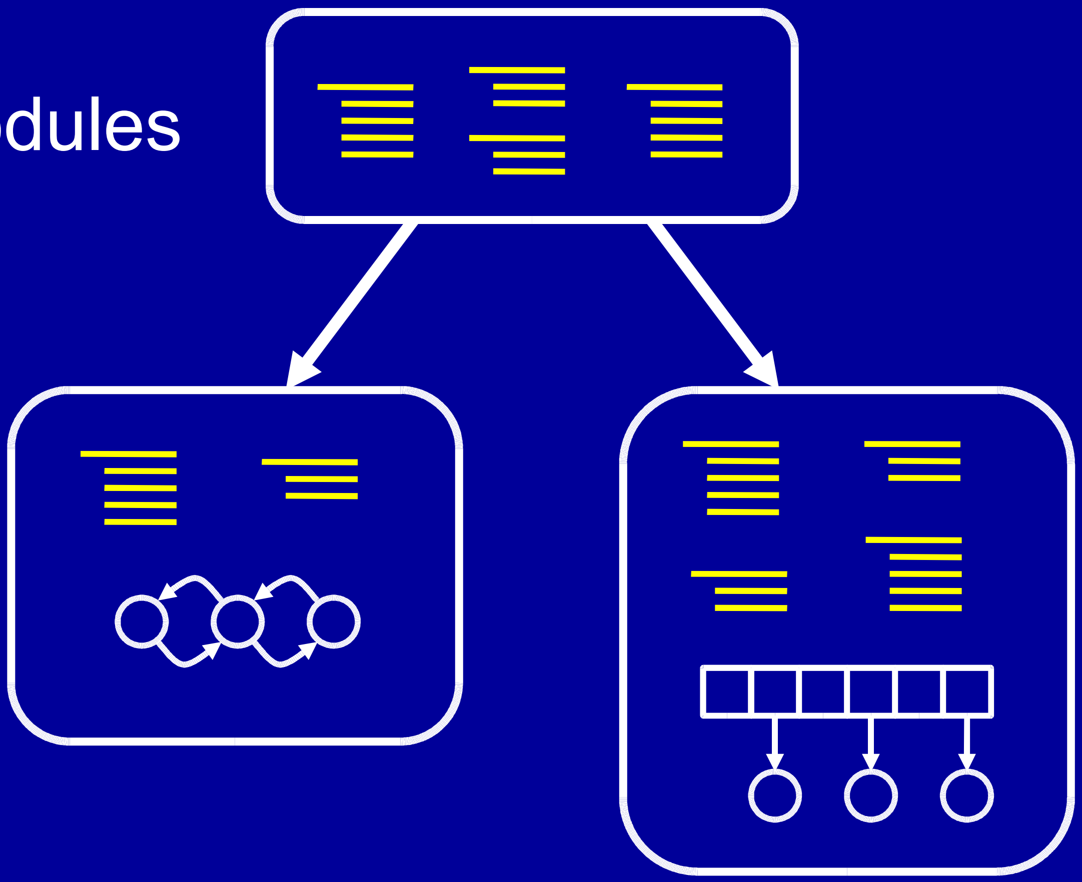
- Encapsulated data structures
- Exported procedures
- Code



Goal

Verify System Data Structure Consistency

- Within each module
  (e.next.prev = e)
- Across multiple modules
  (no object in both
  list and array)



1

Challenge 1: Scalability

Challenge 2: Diversity

Solution: Modular Analysis

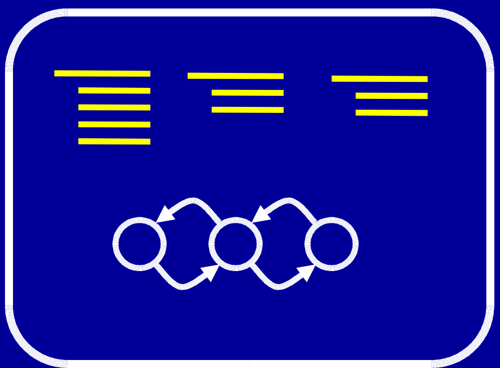
Outline

- Running Example
- Specifying Program Properties
- Linking Implementations and Specifications
- Establishing Local Program Properties
- Establishing Global Program Properties
- Experience
- Related Work & Conclusion

Process Scheduler Example

Idle Process Module

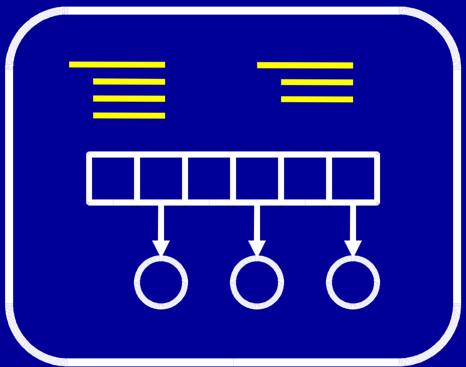
add(p)
empty()
del()



doubly-linked list

Running Process Module

ins(p)
rem(p)



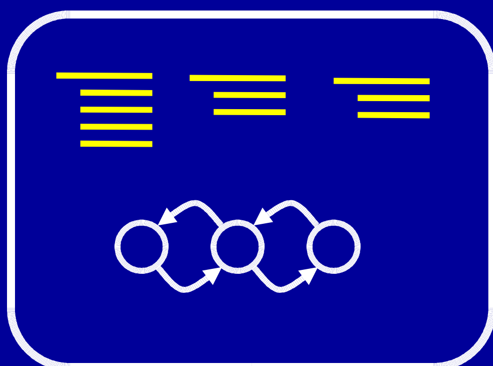
array

Consistency Properties

8 No process is simultaneously idle and running

Idle Process Module

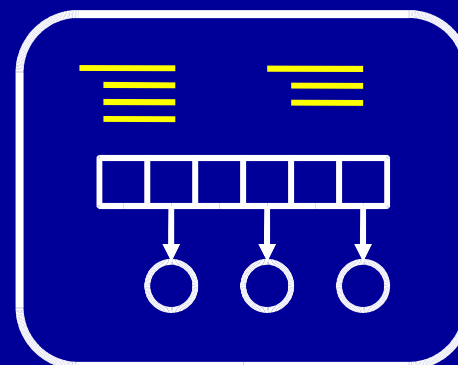
add(p)
empty()
del()



p.next.prev = p,
p.prev.next = p,
no cycles

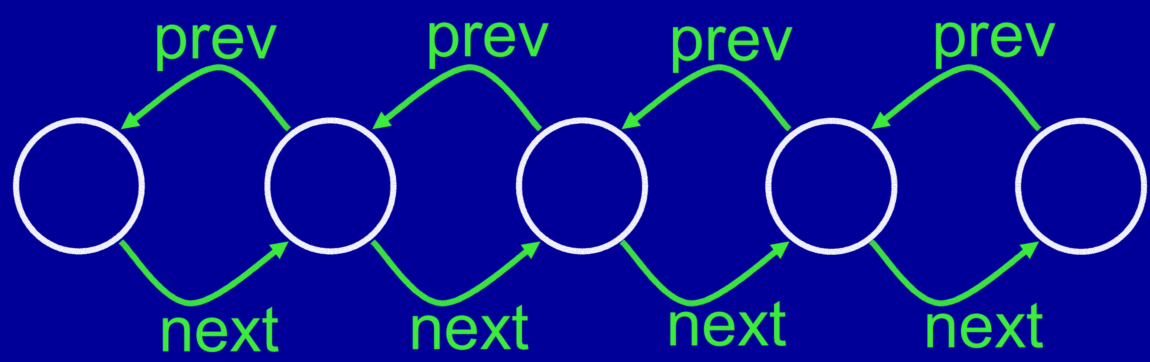
Running Process Module

ins(p)
rem(p)



7 elements indexed properly
no duplicates

Idle Process Module Implementation



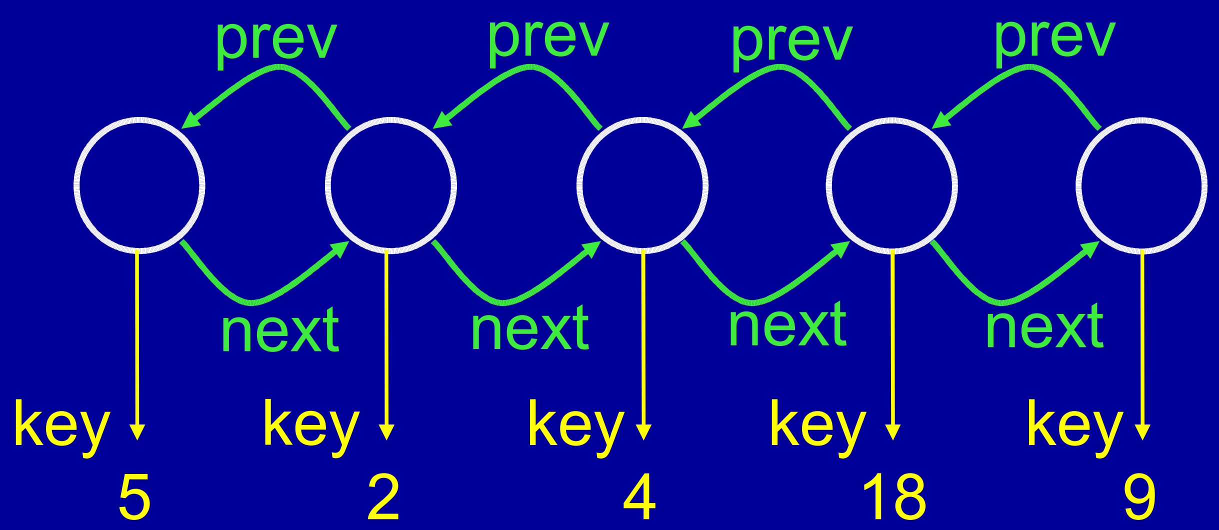
impl module idle { 3

reference root : Process;

format Process { next : Process; prev : Process; }

Format statements declare object fields.

On Formats


prev
prev
prev
prev
next
next
next
next
key
5
key
2
key
4
key
18
key
9

Idle Process Module Implementation

```
impl module idle {
    reference root : Process;
    format Process { next : Process; prev : Process; }

    proc add(p : Process) {        3
      if (root == null) {
        root = p; p.prev = null; p.next = null;
      } else {
        p.next = root; root.prev = p; p.prev = null; root = p;
      }
    }

    proc del() returns p : Process; { … }
    proc empty() returns b : bool; { … }
}
```

Outline

- Running Example
- Specifying Program Properties
- Linking Implementations and Specifications
- Establishing Local Program Properties
- Establishing Global Program Properties
- Experience
- Related Work & Conclusion

What Do We Want to Verify?

On entry to and exit from add(p) and del()

- $\forall$ p in root.next*: p.next.prev = p
- $\forall$ p in root.next*: p.prev.next = p
- acyclic root.next*

7

invariants

Whenever calling add(p), p$\notin$ root.next*

Calls to del() return some p such that

- p$\in$ root.next* before call
- p$\notin$ root.next* after call

7

usage constraints

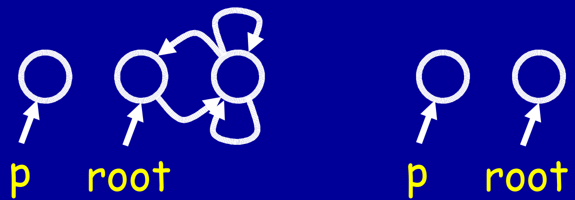
No process simultaneously running and idle

8

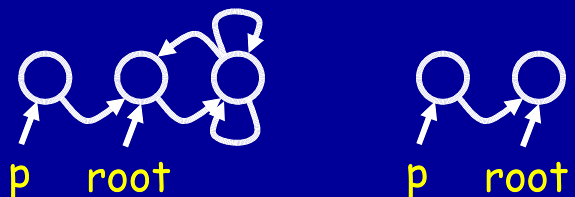
global conditions

Apply Shape Analysis

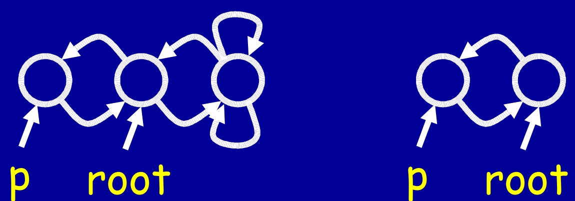
Detailed analysis, works with model of heap:



p.next = root;



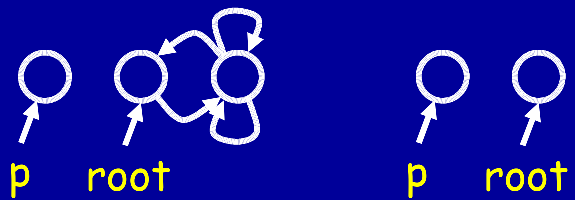
root.prev = p;



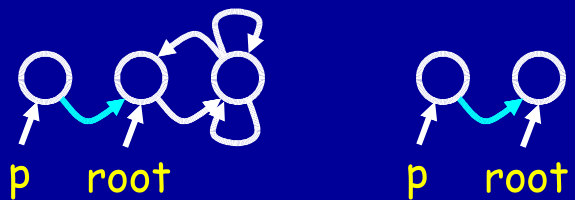
Should be able to use assume/guarantee reasoning to verify consistency conditions

Apply Shape Analysis

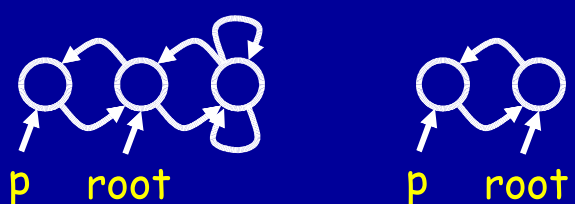
Detailed analysis, works with model of heap:



p.next = root;



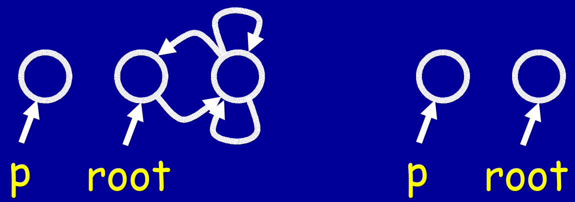
root.prev = p;



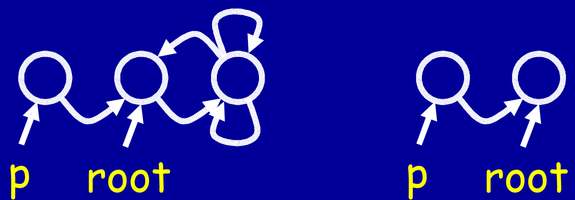
Should be able to use assume/guarantee reasoning to verify consistency conditions

Apply Shape Analysis

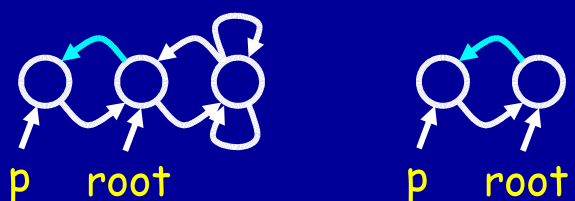
Detailed analysis, works with model of heap:



p.next = root;



root.prev = p;



Should be able to use assume/guarantee reasoning to verify consistency conditions

Two Problems

Preconditions outside module

Whenever calling add(p), $p \notin root.next^*$

Infeasible to use shape analysis for entire program

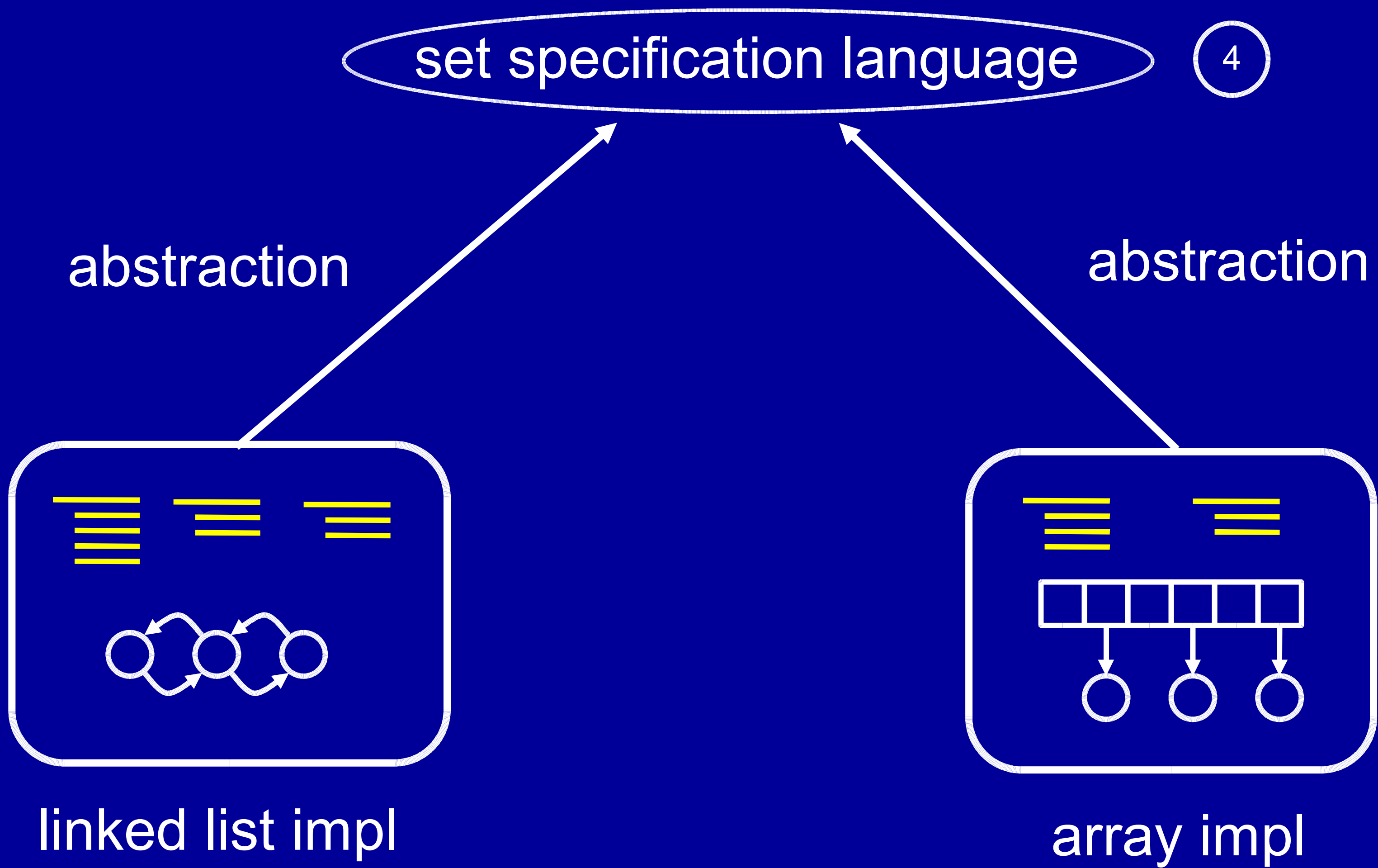
Properties involving multiple modules

No process simultaneously running and idle

Array and list analyses must exchange information

But use dramatically different abstractions

The Solution: A Layered Abstraction



Module Components

3 Implementation
- Encapsulated data structures
- Procedure implementations

4 Interface - requires, ensures, modifies clauses for each exported procedure

5 Abstraction
- Which analysis to apply to the implementation
- Internal data structure consistency properties
- Connection between
  - Encapsulated data structures in module
  - Shared interoperation abstraction

Let’s see what it is like to develop a module using this approach!

Interface

spec module idle {   4

    …

Interface

```
spec module idle {   4
  specvar Idle : Process set;
  ...
```

Modules export abstract sets of objects, which:

- are simply a specification mechanism
- do not exist when program runs
- characterize how objects participate in module’s encapsulated data structures
- used to define module’s interface

Interface

spec module idle {

specvar Idle : Process set;

proc add(p : Process)

requires $(p \notin \text{Idle}) \wedge p \neq \text{null}$ modifies Idle

ensures $\text{Idle}' = \text{Idle} \cup \{p\}$; 4

Each exported procedure has requires, modifies, and ensures clauses

Use (quantified) boolean algebra of sets

Boolean Algebra of Sets

$SE ::= \emptyset, p, p', S, S', S_1 \cap S_2, S_1 \cup S_2, S_1 - S_2$

$B ::= SE_1 = SE_2, SE_1 \subseteq SE_2,$

$p \in SE, p \notin SE, p = null, p \neq null,$

$|SE| = k, |SE| \geq k, |SE| \leq k,$

$\forall S.B, \exists S.B,$

$B_1 \wedge B_2, B_1 \vee B_2, \neg B,$

b, b'

Satisfiability, Entailment Decidable (Skolem 1919)

Interface

spec module idle { 4

specvar Idle : Process set;

proc add(p : Process)

requires $(p \notin Idle) \wedge p \neq null$ modifies Idle

ensures $Idle' = Idle \cup \{p\}$;

proc del() returns p : Process

requires $|Idle| \geq 1$ modifies Idle

ensures $Idle' = Idle - \{p\} \wedge p \in Idle \wedge p \neq null$;

- Can also have cardinality constraints on sets

Interface

```
spec module idle {    4
  specvar Idle : Process set;
  proc add(p : Process)
      requires (p ∉ Idle) ∧ p ≠ null modifies Idle
      ensures Idle' = Idle ∪ {p};
  proc del() returns p : Process
      requires |Idle| ≥ 1 modifies Idle
      ensures Idle' = Idle – {p} ∧ p ∈ Idle ∧ p ≠ null;
  proc empty() returns b : bool
      ensures b ⇔ |Idle| = 0;
}
```

Benefits of a Set Spec Language (1)

Capture important data structure aspects

Can capture interface requirements

Benefits of a Set Spec Language (2)

Membership in orthogonal sets supports

- Useful polymorphism
- Separation of concerns

Provide productive perspective on program

- Sets characterize changing object roles
- Set membership changes reflect role changes

Benefits of a Set Spec Language (3)

Promote verified connection between design (object model) and implementation

Outline

- Running Example
- Specifying Program Properties
- Linking Implementations and Specifications
- Establishing Local Program Properties
- Establishing Global Program Properties
- Experience
- Related Work & Conclusion

Connection Between Sets (Interface) and Data Structures (Implementation)

5 abst module idle { analysis PALE; 6

- analysis PALE statement tells system to use the PALE analysis plugin to analyze idle module
- In general, can use whatever analysis you want
- System comes with several
  - PALE is a shape analysis from Denmark (Anders Moeller and Michael Schwartzbach)
  - Also have array and field analysis plugins
- Or you can even implement your own

Connection Between Sets (Interface) and Data Structures (Implementation)

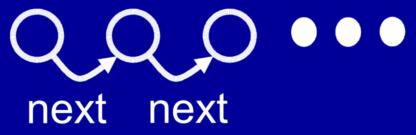
abst module idle { analysis PALE;

Idle = { p : Process | root<next*>p};

- This definition states that the Idle set contains all of the objects in root.next*
- Precise syntax of definition depends on plugin
- Abstraction modules use values in data structure to define meaning of exported abstract sets

Connection Between Sets (Interface) and Data Structures (Implementation)

```
abst module idle { analysis PALE;
   Idle = { p : Process | root<next*>p};
   invariant type L = {
      data next : L;
```



- PALE analysis works with data structures that have a backbone and routing pointers
- data next : L says that the backbone consists of the next references of the objects

Connection Between Sets (Interface) and Data Structures (Implementation)

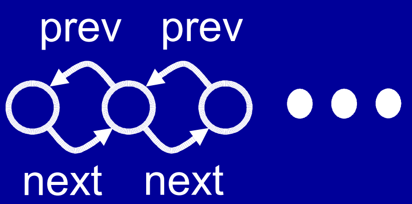
abst module idle { analysis PALE;

Idle = { p : Process | root<next*>p};

invariant type L = {

data next : L;

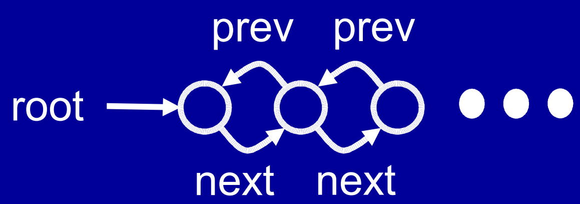
pointer prev : L [this^L.next = {prev}];



- prev is a routing pointer in the data structure
- prev is the inverse of next
- So p.next.prev = p.prev.next = p

Connection Between Sets (Interface) and Data Structures (Implementation)

5
```
abst module idle { analysis PALE;
    Idle = { p : Process | root<next*>p};
    invariant type L = {
        data next : L;
        pointer prev : L [this^L.next = {prev}];
    };
    invariant data root : L;
}
```



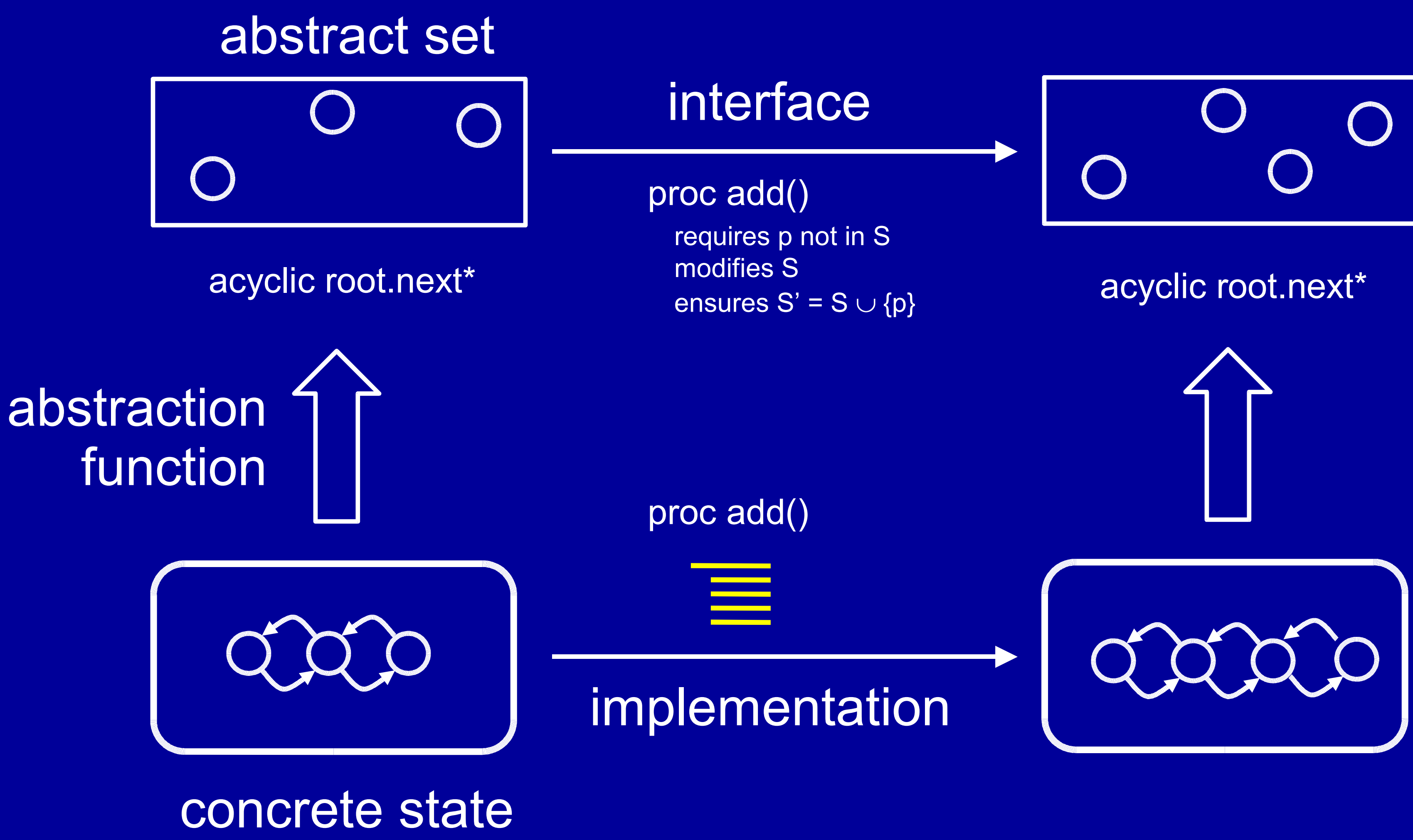
6

- root is the root of a data structure of L's

Outline

- Running Example
- Specifying Program Properties
- Linking Implementations and Specifications
- Establishing Local Program Properties
  - PALE, Flag, Theorem Proving Plugins
- Establishing Global Program Properties
- Experience
- Related Work & Conclusion

What Happens Next?



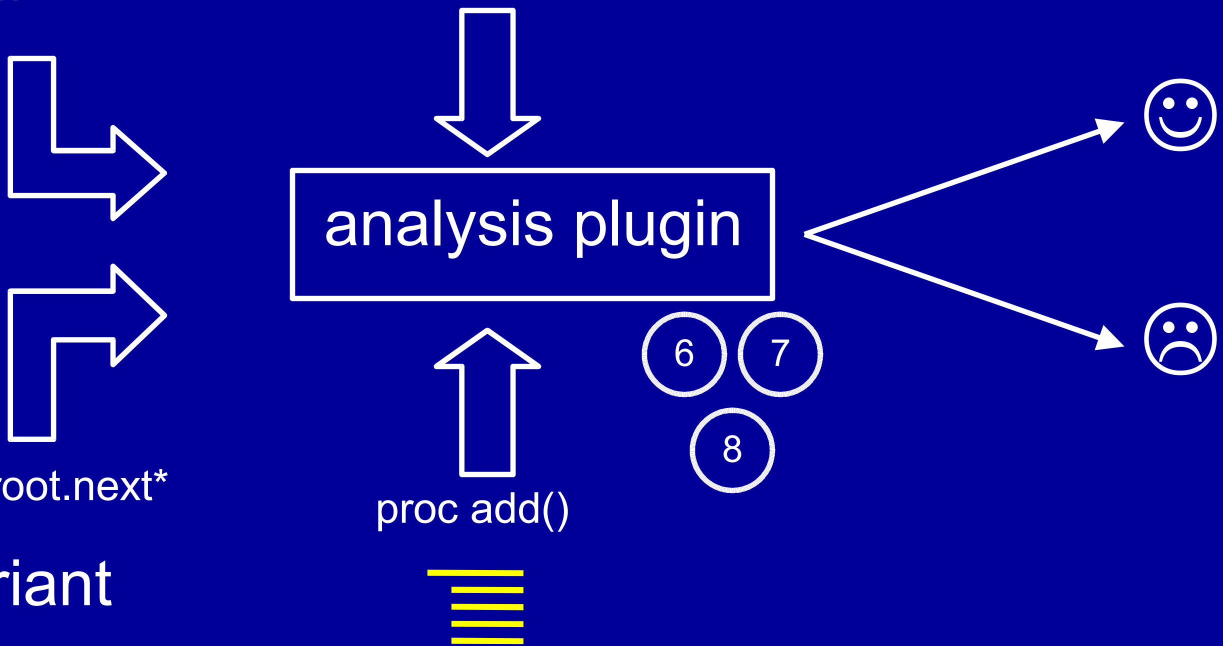
What Happens Next?

other set specifications

module Scheduler {
proc suspend() requires s in S;
proc resume() …
}

translated interface

proc add()
requires p not in root<next*>
ensures root<next*>’ = root<next*> ∪ {p} ∧ frame



acyclic root.next*

invariant

proc add()

implementation

Plugins in Hob

- Shape Analysis Plugin

  Invokes PALE shape analysis tool to assign set membership according to heap structure.

- Flag Analysis Plugin

  Manipulates boolean algebra formulas only; more scalable than shape analysis.

- Theorem Proving Plugin

  Invokes Isabelle interactive theorem prover to establish arbitrary statements about program execution.

Some modules are really simple

Coordination Modules

6

- Coordinate actions of other modules
  - Maintain references to objects
  - Pass objects as parameters to other modules
  - Get references back as return values
- No encapsulated data structures
- No abstraction functions
- Just interfaces and implementations

Example: Scheduler module coordinates Idle and Running process modules

Example Coordination Code

```
p1 = new Process();
p2 = new Process();
p3 = new Process();
add(p1);
add(p2);
add(p3);
x = del();
y = del();
```

What Does Set Analysis Know?

```
p1 = new Process();
p2 = new Process();
p3 = new Process();
add(p1);
add(p2);
add(p3);
x = del();
y = del();
```

Known Facts

- $p1 \neq p2$
- $p1 \neq p3$
- $p2 \neq p3$
- $x \neq y$
- $|Idle|=1$

Flag Plugin 6

- Extension of Set Analysis plugin
- Set membership given by values of primitive fields
- Example sets:

  Idle = { x : Process | x.status = 1 }
  Running = { x : Process | x.status = 2 }
- Also works for boolean flags
- Analysis
  - Same abstract set machinery as Set Analysis plugin
  - Also update sets when flags change

    x.status = 2:

    Idle’ = Idle – x

    Running’ = Running ∪ x

Analyzing Coordination Modules

Hob's Flag Analysis plugin manipulates set specifications to ensure needed preconditions and to guarantee postconditions

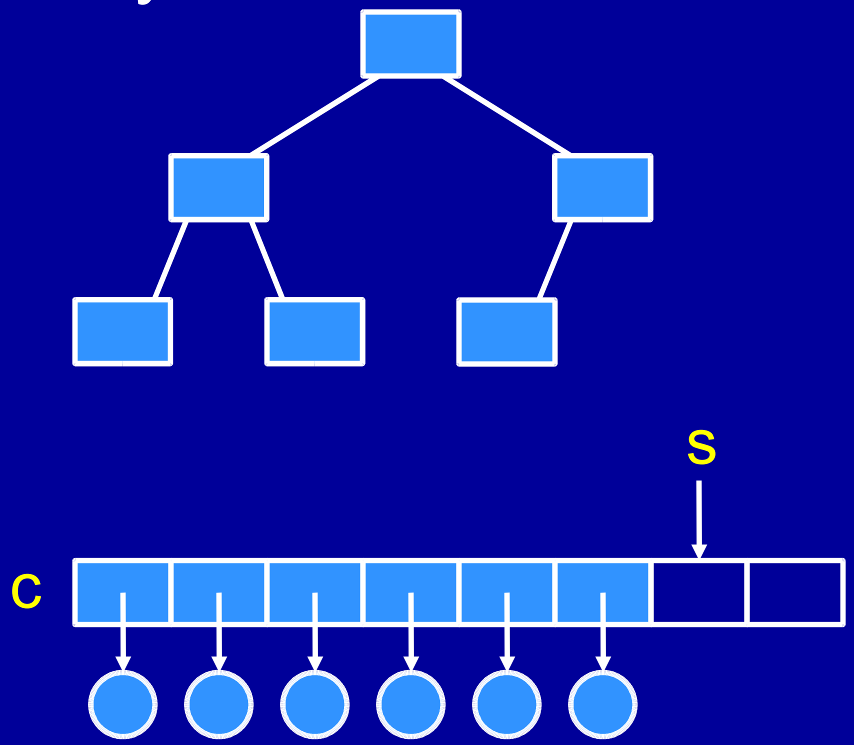
More details in VMCAI '05,

Lam, Kuncak and Rinard. “Verifying Set Interfaces based on Object Field Values”.

Some data structure invariants are even more complicated!

Priority Queue Implemented as an Array

- Complete binary tree up to last row
- Implementing tree in array
  - parent(i) = i/2
  - left(i) = 2i
  - right(i) = 2i + 1



Applying Theorem Proving 6

```
spec module SuspendedQueue {
   specvar InQueue : Process set;

proc insert(p: Process; priority: int)
   requires not (p in InQueue)
   modifies InQueue
   ensures InQueue' = InQueue + p;
   …
}
```

```
impl module SuspendedQueue {
   format Process { priority : int };
   var c: Process[];
   var s: int;

   proc insert(p: Process; priority: int) { ... }
   …
}
```

```
abst module SuspendedQueue {
   use plugin "vcgen";
   InQueue = { x : Process | exists j. 1 ≤ j & j ≤ s & x = c[j] };

   invariant "0 ≤ s";
   invariant "forall i. (forall j.
              ((1 ≤ i) & (i ≤ s) & (1 ≤ j) & (j ≤ s) & (c[i] = c[j])) => i = j"
}
```

Abstracting Arrays as Sets

Theorem Proving Plugin accepts arbitrary Isabelle formulas as set definitions:

InQueue = { x : Process | exists j. 1 ≤ j & j ≤ s & x = c[j] };

We generate proof obligations from the implementation code.

How well does this work?

- insert example
- Generates 11 sequents
- Of these:
  - Isabelle discharges 5 automatically
  - We proved 6 manually
    - Shortest proof: 1 line (introducing an arithmetic lemma)
    - Longest proof: 38 lines
    - Average proof length: 14.2 lines

For more on Theorem Proving...

... see our SVV 2004 paper,

Zee, Lam, Kuncak and Rinard. “Combining Theorem Proving with Static Analysis for Data Structure Consistency”.

Outline

- Running Example
- Specifying Program Properties
- Linking Implementations and Specifications
- Establishing Local Program Properties
- Establishing Global Program Properties
- Experience
- Related Work & Conclusion

Moving to More General Properties

So far, we've discussed intra-module properties:

- linked list consistency properties
- array data structure properties

These properties serve to establish set abstractions.

Can we productively use the set abstraction?

Using and Improving Hob's Spec Language

Hob uses sets to state cross-module properties:

- set disjointness properties
- more general relations between set contents

9 10

Hob also includes *scopes* and *defaults*, which allow developers to write better (more concise) module specifications.

Cross-Module Properties 9

Stated using common specification abstraction, e.g.:

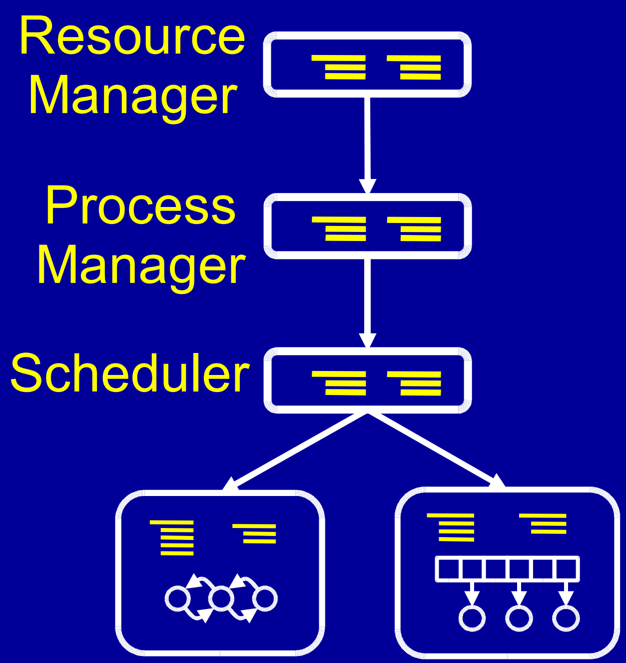
$$\text{Running} \cap \text{Idle} = \emptyset$$

Such invariants cross-cut multiple modules and hold at many different program points.

In principle, could manually conjoin these invariants to all appropriate points.

Specification Aggregation

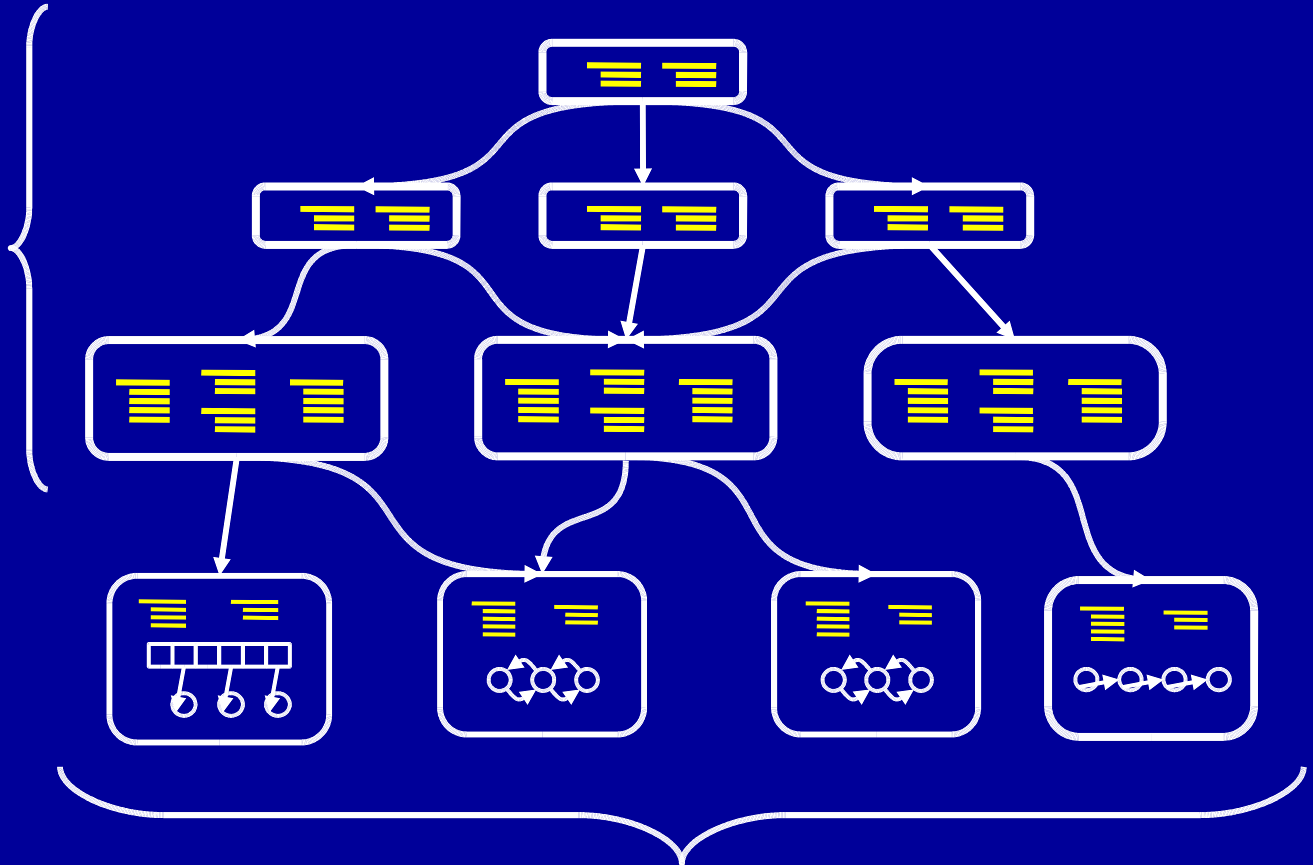
- Hierarchy of modules
- Standard approach:
  - Weave into preconditions through program
  - Weave into call sites where they are needed



Result is that specifications aggregate, moving up the hierarchy

Standard Usage Scenario

Modules
Coordinate
Data
Structure
Operations



Leaf Modules Encapsulate Data Structures

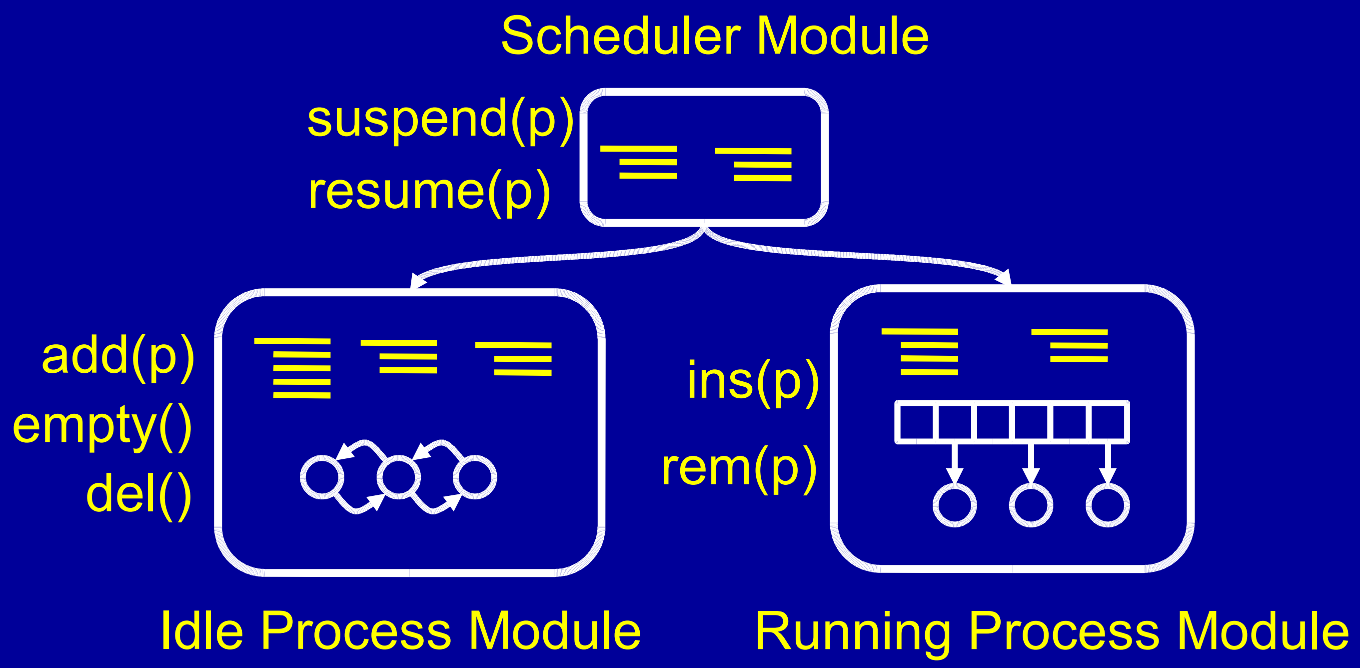
Even more aggregation!

Example Scope


```
scope S {
    invariant Running ∩ Idle = ∅;
    modules scheduler, idle, running;
    export scheduler;
}
```

- Property holds except within modules in scope
- Sets of invariant included in modules in scope
- Outside scope
  - Use invariants to prove other properties
  - Invoke procedures in exported modules only

Scopes in Example

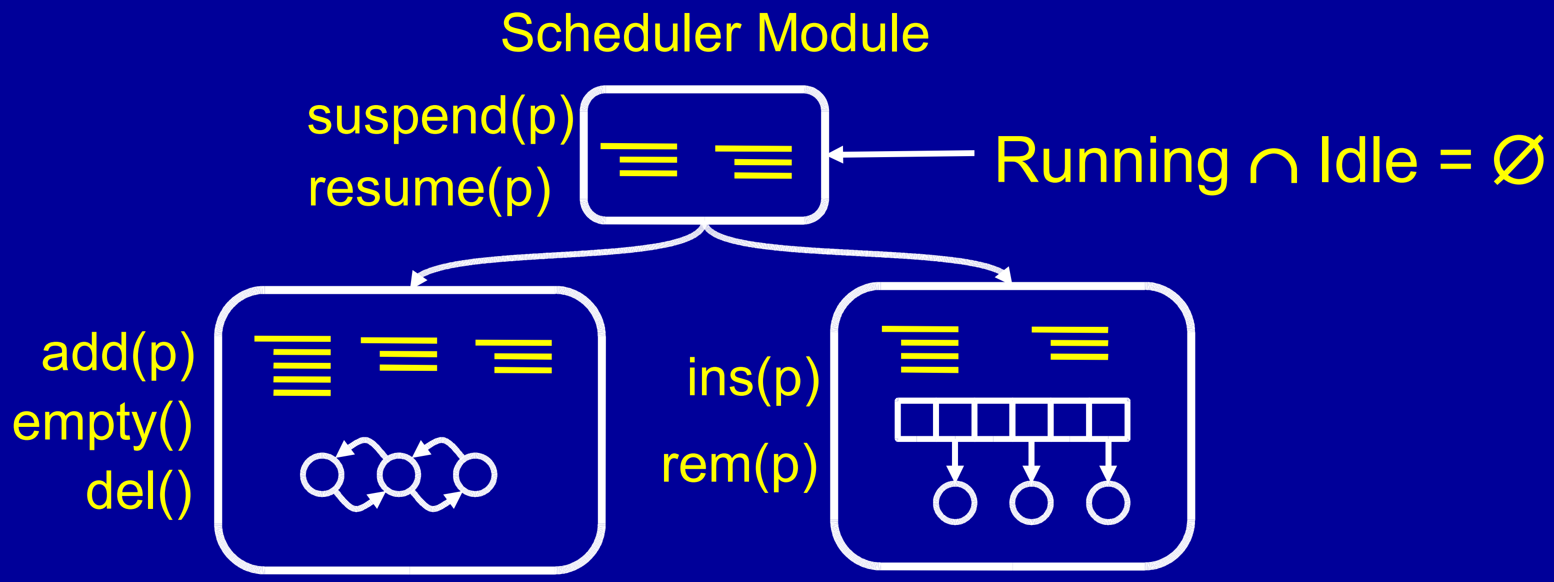


- Running ∩ Idle = ∅ may be violated anywhere within Scheduler, Idle Process, or Running Process modules
- Scheduler must coordinate operations on Idle Process and Running Process Modules
- Otherwise property may become violated outside scope
- Concept of internal and exported modules in a scope

Scopes and Analysis

System conjoins property to preconditions and postconditions of exported modules

Analysis verifies procedures preserve property



Why Scope Invariants Work

Hob verifies scope invariants:

- in program's initial state, and
- whenever exiting the scope.

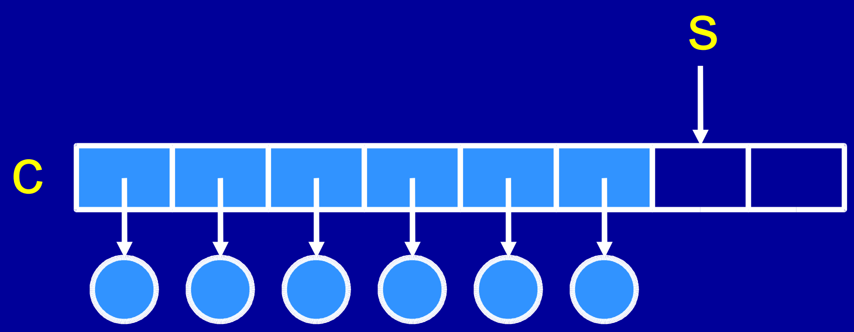
Truth or falsity of the invariant never changes outside the scope.

Hob may therefore assume that the invariant holds upon entry to the scope.

Guards

10

Consider an array-based data structure.



Must allocate the array before calling data structure operations!

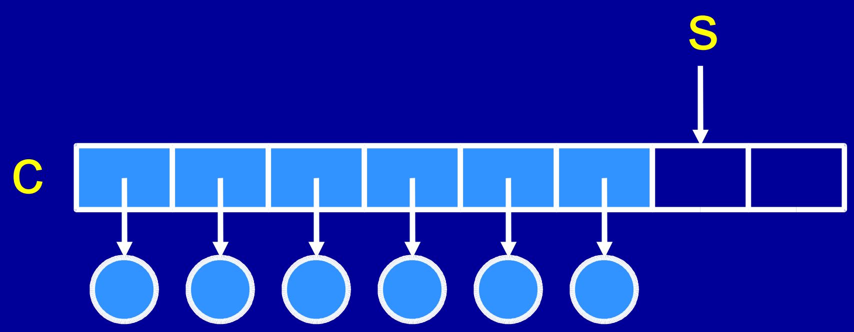
specvar Init : bool;

proc init() ensures Init';

proc add(p) requires Init ... ;

Guards

Consider an array-based data structure.



Must allocate the array before calling data structure operations!

specvar Init : bool;

proc init() ensures Init';

proc add(p) requires $\boxed{\text{Init}}$ ... ;

explicit initialization constraint

Applying Defaults

Hob automatically conjoins defaults to appropriate ensures and requires clauses:

```
proc init()

   ensures Init';
proc add(p)
   requires Init & p != null
   ensures …;
proc del(p)
   requires Init & …
   ensures ….;
```

```
default I : Init;
proc init()
   suspends I
   ensures Init';
proc add(p)
   requires       p != null
   ensures … ;
proc del(p)
   requires       …
   ensures …;
```

Applying Defaults Appropriately

Developers may specify a pointcut for the default:

```
default padRead(q) :        10
        pre(all(scope C)) =
  (card(q) = 1) & (q in M.Reading)
```

Default Pointcut Language

P ::= $P_1 – P_2$ | P_1 & P_2 | $P_1|P_2$ | not P

| pre S | post S | prepost S

S ::= $S_1 – S_2$ | S_1 & S_2 | $S_1|S_2$ | not S

| proc pn(tn_1, …, tn_k) returns tn_r

| exports (module ms) | exports (scope ss)

| local (model ms) | local (scope ss)

| all (module ms) | all (scope ss)

| all

Defaults Improve Specifications

- Convert errors of omission (i.e. missing clauses) into errors of commission.
- Allow developers to write more concise specifications focussing on locally important properties.

For more on Scopes and Defaults

See our AOSD '05 article:

- Lam, Kuncak, and Rinard. “Cross-Cutting Techniques in Program Specification and Analysis.”

Outline

- Running Example
- Specifying Program Properties
- Linking Implementations and Specifications
- Establishing Local Program Properties
- Establishing Global Program Properties
- Experience
- Related Work & Conclusion

Hob Framework & Benchmarks

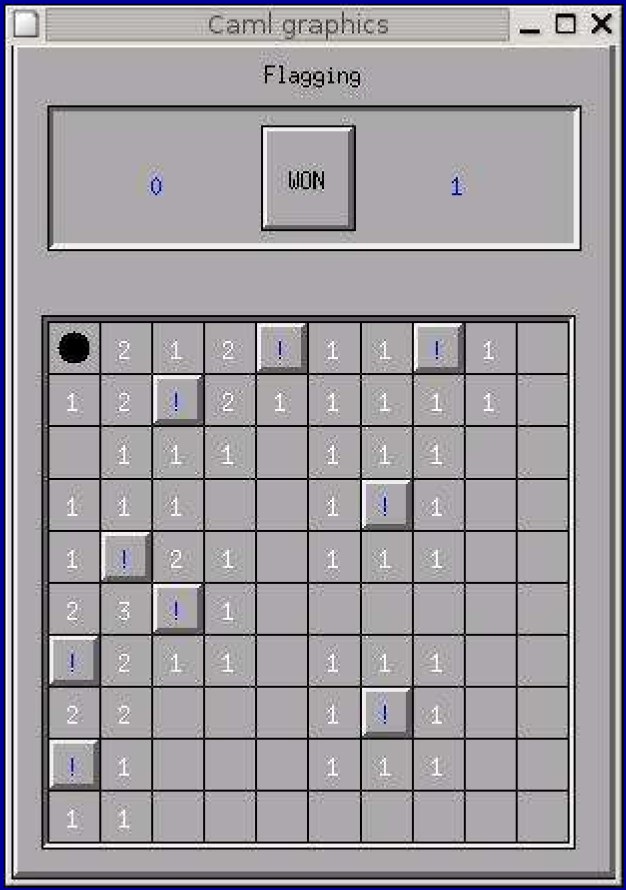
- Implemented Hob System components:
  - Interpreter
  - Analysis framework
  - Pluggable analyses
    - Set/flag analysis
    - PALE analysis interface
    - Array analysis (VCs discharged via Isabelle)
- Modules and programs
  - Data structures
  - Minesweeper, Water

Data Structures

- Lists (doubly and singly linked)
- List-based data structures

  (stacks, sets, queues, priority queues)
- Array data structure (set)

Minesweeper



Minesweeper

- 750 lines of code, 236 lines of specification
- Full graphical interface (model/view/controller)
- Data structure consistency properties
  - Lists, arrays of board cells are consistent
  - No duplicates; pointer consistency properties
- Board cell state correlations
  - All cells are exposed or hidden
  - No exposed cell has a mine unless game over
- Correlations between state and actions
  - Cells initialized before game starts
  - Can’t reveal entire board until game over
  - Iterators used correctly

Water

- Time step computation, simulates liquid water
- Computation consists of sequence of steps
  - Predict, correct, boundary box enforcement
  - Inter and intra molecular force calculations
- 2000 lines of code, 500 lines of specification
- Typestate properties
  - Simulation parameters properly initialized
  - Atoms are in correct states for each step
  - Molecules are in correct states for each step
- State correlations – simulation, atoms, molecules

Set Abstraction Worked Great

Captured data structure participation in a powerful, intuitive way

- Individual data structure consistency
- Correlations between data structures

Powerful interface specification language

- Procedure call sequencing requirements
- Object use requirements
- Connections between state and actions

Able to deploy multiple analyses productively

(the first time anyone has been able to do this)

Framework Made Everything Better

Better design

- Sets helped us conceptualize design
- Enabled us to identify and verify high-level properties

Better implementation

- Better structure
- Easier to understand
- Fewer errors

Guaranteed correspondence between implementation and (aspects of) design

Outline

- Running Example
- Specifying Program Properties
- Linking Implementations and Specifications
- Establishing Local Program Properties
- Establishing Global Program Properties
- Experience
- Related Work & Conclusion

Related Work

Shape analyses
- Moeller, Schwartzbach PLDI 2001
- Ghiya, Hendren POPL 1996

Typestate
- Strom, Yellin IEEE TOSEM 1986
- DeLine, Fahndrich ECOOP 2004, PLDI 2001

Theorem provers
- Isabelle, Athena, HOL, PVS, ACL2

Program specification
- Eiffel, JML, Spec#

Verifiers – Program Verifier, Stanford Pascal Verifier, Larch, ESC/Modula-3/Java, Boogie

Primary Contribution

Hob framework for modular program analysis:

- Abstract set specification language
- Scope invariants; defaults and guards

Enables multiple (very precise and unscalable) analyses to interoperate

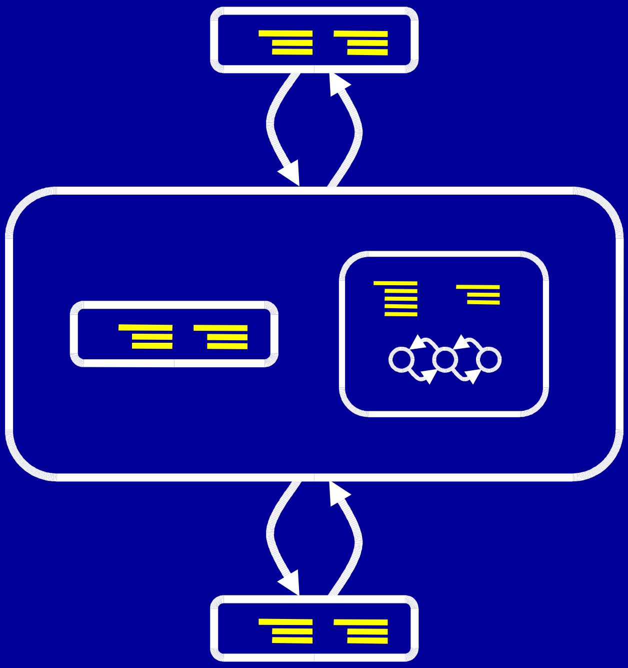
Verifies data structure consistency properties

First system to combine high-level properties from markedly different analyses http://cag.csail.mit.edu/~plam/hob

Outcalls

- So far, all calls enter and exit scopes from top
- What about outcalls from scope?

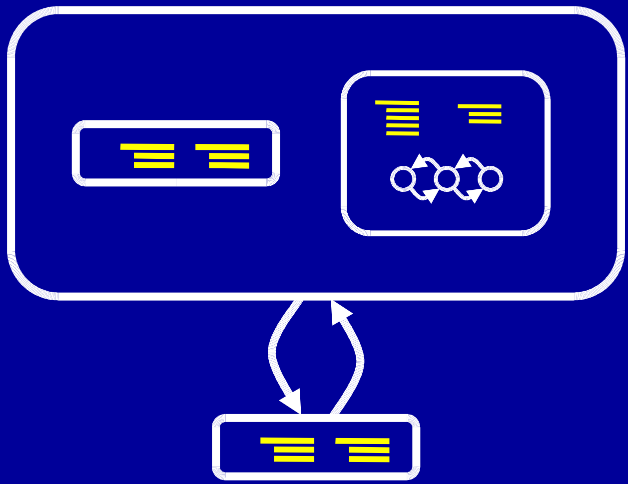
Scope



Invariant Issue

- Invariant may be violated inside scope
- If callee uses invariant (transitively), must reestablish invariant before call
- If callee does not use invariant (transitively), should be able to call with invariant violated

Scope



- Our approximation: restore invariant before reentrant outcalls

Potential policy variants

- Could have outcalls without invariant restoration when appropriate
  - A procedure can declare invariants it uses
  - If so, can only call procedures that use at most these invariants
  - If an outcalled procedure does not use invariant, do not need to restore it