The Hob System for Verifying Data Structure Consistency Properties

Patrick Lam, Viktor Kuncak, Karen Zee, Martin Rinard

MIT CSAIL

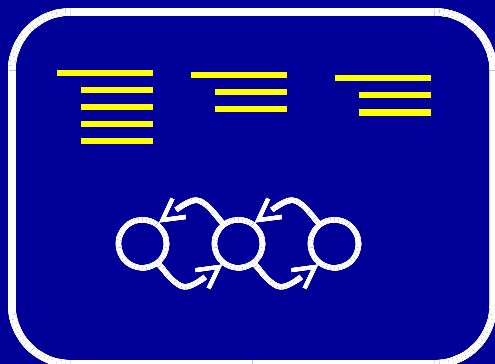
Massachusetts Institute of Technology

Cambridge, MA 02139

Process Scheduler Example

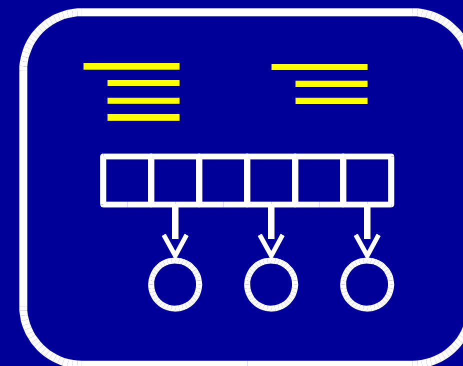
Idle Process Module

add(p)
empty()
del()



Linked List

Running Process Module



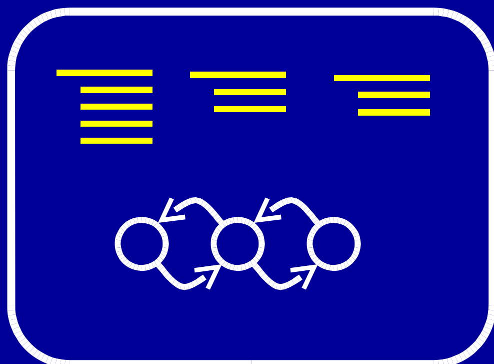
ins(p)
rem(p)

Hash Table

Consistency Properties

Idle Process Module

add(p)
empty()
del()

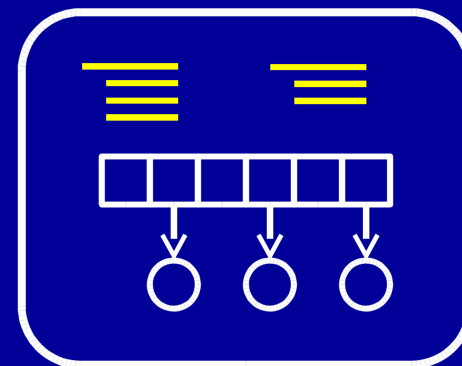


p.next.prev = p

p.prev.next = p

no cycles

Running Process Module



ins(p)
rem(p)

elements indexed correctly

no duplicates

No process is simultaneously idle and running

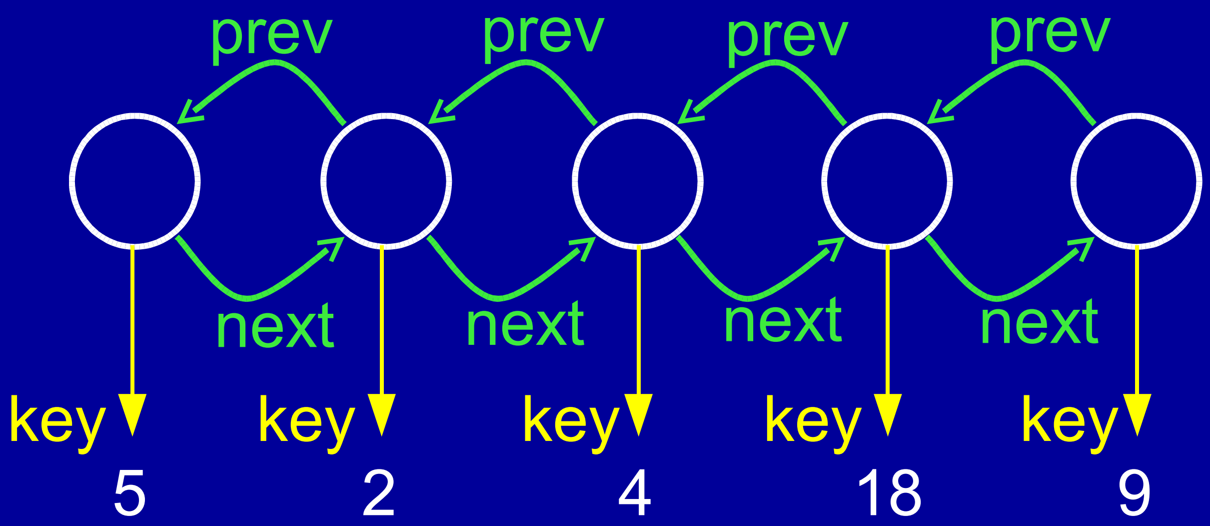
$$\text{Running} \cap \text{Idle} = \emptyset$$

Idle Process Module Implementation

```
impl module idle {
    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
key
key
key
key
5
2
4
18
9

Idle Process Module Implementation

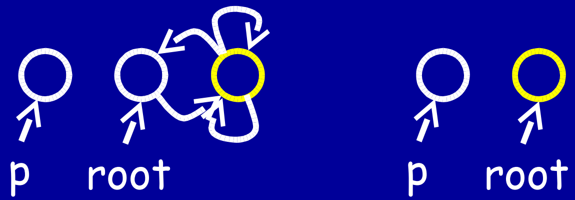
```
impl module idle {
    reference root : Process;
    format Process { next : Process; prev : Process; }

    proc add(p : Process) {
      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; { … }
}
```

What Do We Want to Verify?

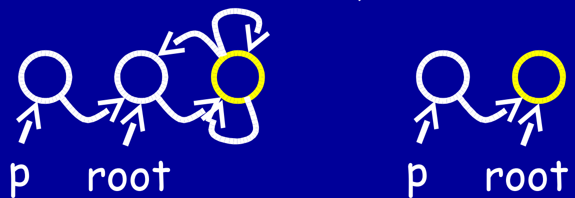
- Invariants for encapsulated list -
  on entry to and exit from add(p) and del()
  - $\forall p \in root.next^*\ \ p.next.prev = p$
  - $\forall p \in root.next^*\ \ p.prev.next = p$
  - acyclic $root.next^*$
- Whenever calling add(p), $p \notin root.next^*$
- Calls to del() return some p such that
  - $p \in root.next^*$ before call
  - $p \notin root.next^*$ after call
- No process simultaneously running and idle

$$Running \cap Idle = \emptyset$$

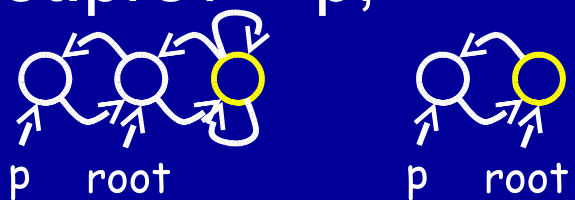
Apply Shape Analysis



p.next = root;



root.prev = p;



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

Two Problems

1) Preconditions outside module

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

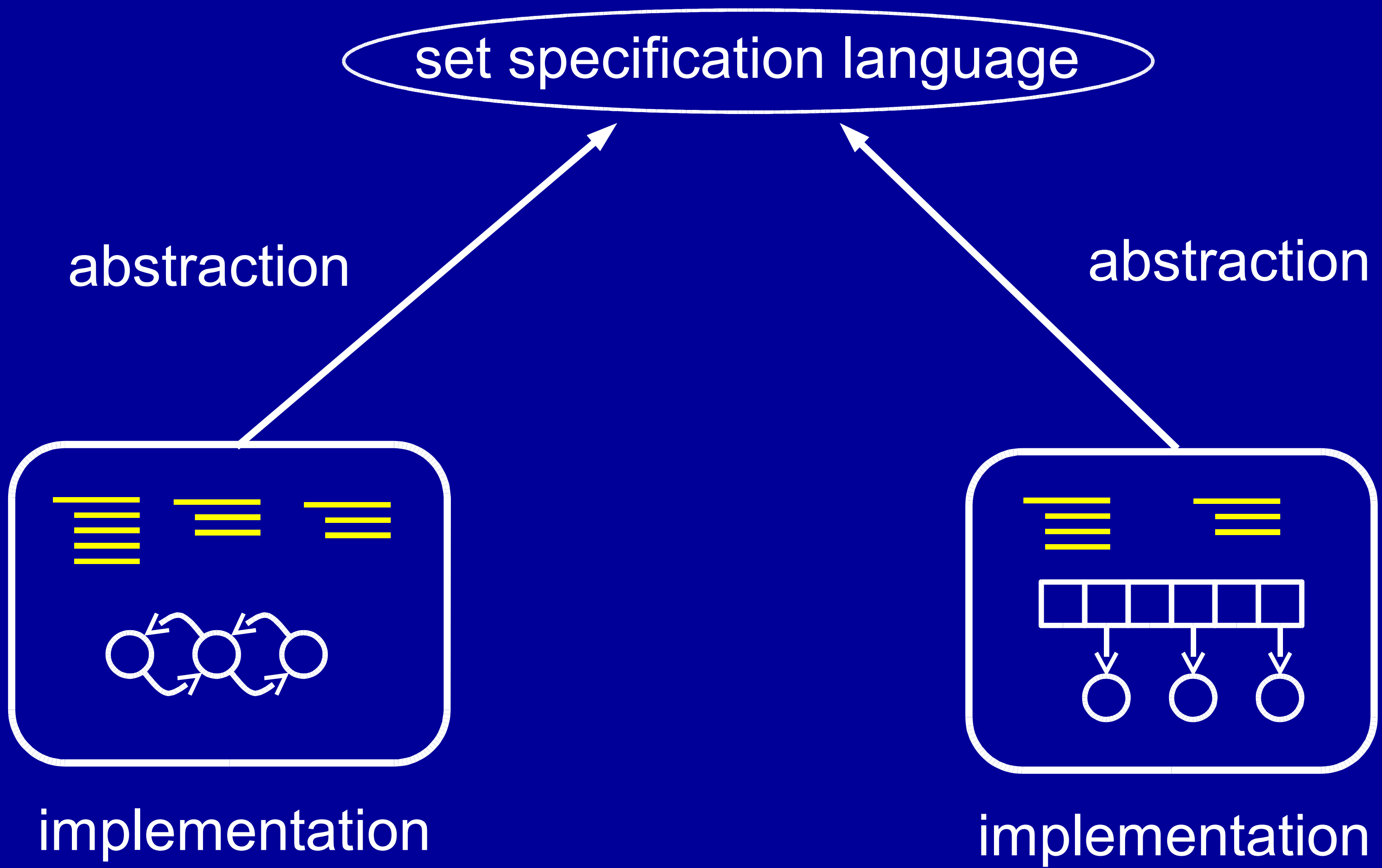
Infeasible to use shape analysis for entire program

2) Properties involving multiple modules

Running $\cap$ Idle = $\emptyset$

- Hash table and list analyses must (at least) exchange information
- But use dramatically different abstractions
  - Hash table analysis – array abstraction
  - List analysis – shape analysis abstraction

The Solution: A Layered Abstraction



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

Interface

```
spec module idle {
```

Interface

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

- Modules export abstract sets of objects, which:
  - do not exist when program runs;
  - are simply a specification mechanism
  - characterize how objects participate in module's encapsulated data structures
  - used to define module's interface

Interface

spec module idle {

sets Idle : Process;

proc add(p : Process)

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

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

- 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 {
  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
```

- Can also have cardinality constraints on sets

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 $\text{Idle}' = \text{Idle} \cup \{p\}$

proc del() returns p : Process

requires $|\text{Idle}| \geq 1$ modifies Idle

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

proc empty() returns b : bool

ensures $b \Leftrightarrow |\text{Idle}| = 0$

Why Sets

- Capture important data structure aspects
  - Many data structures implement sets
  - Characterize data structure participation
- Can capture interface requirements
  - Object state preconditions reified as abstract set membership
  - Express required relationships between
    - Program state
    - Actions of program
- Membership in orthogonal sets supports
  - Useful polymorphism
  - Separation of concerns

Why Sets

- Provide productive perspective on program
  - Sets characterize changing object roles
  - Set membership changes reflect role changes
- Promote verified connection between design (object model) and implementation

Implementation

```
impl module idle {
    reference root : Process;
    format Process { next : Process; prev : Process; }
    proc add(p : Process) {
      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; { … }
}
```

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

abst module idle { analysis PALE;

- 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)

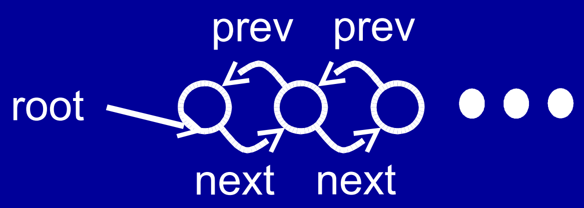
abst module idle { analysis PALE;

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

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

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}];
   };
   invariant data root : L;
}
```



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

What Happens Next?

Have interface, implementation, abstraction

Must verify implementation conforms to interface

General strategy:

- System constructs precondition
  (for each procedure)
  - Internal invariants from abstraction module
  - Requires clause from interface, translated through set definitions in abstraction module
- Similarly constructs postcondition
  (for each procedure)
- Invokes plugin to determine conformance

Other Plugins

- Typestate Analysis Plugin

  Manipulates boolean algebra formulas only; less expensive than shape analysis.

- Theorem Proving Plugin

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

On Cross-Module Properties

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

- linked list consistency properties
- array data structure properties

These properties serve to establish set abstractions.

Hob uses sets to state cross-module properties:

- set disjointness properties
- more general relations between set contents

Cross-Module Properties

Stated using common set specification language, e.g.:

$$Running \cap Idle = \emptyset$$

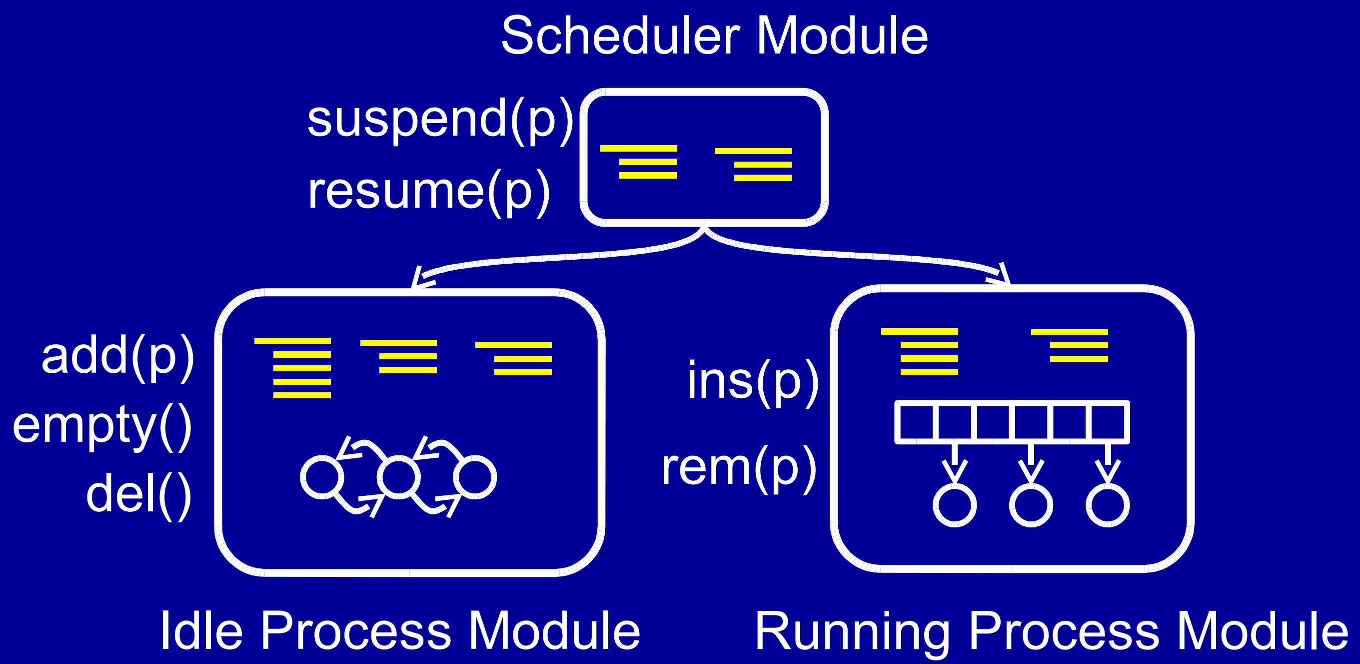
Invariants may be temporarily violated while program updates data structures.

Concept of scope:

- Region where invariant may be violated
- Sets in invariant must belong to scope

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

Scopes in Example



- Running $\cap$ Idle = $\varnothing$ may be violated anywhere within Scheduler, Idle Process, or Running Process modules
- Scheduler must coordinate operations on Idle Process and Running Process Modules
- Otherwise invariant may become permanently violated
- Concept of internal and exported modules in a scope

Example Scope

scope S {

invariant Running $\cap$ Idle = $\emptyset$;

modules scheduler, idle, running;

export scheduler;

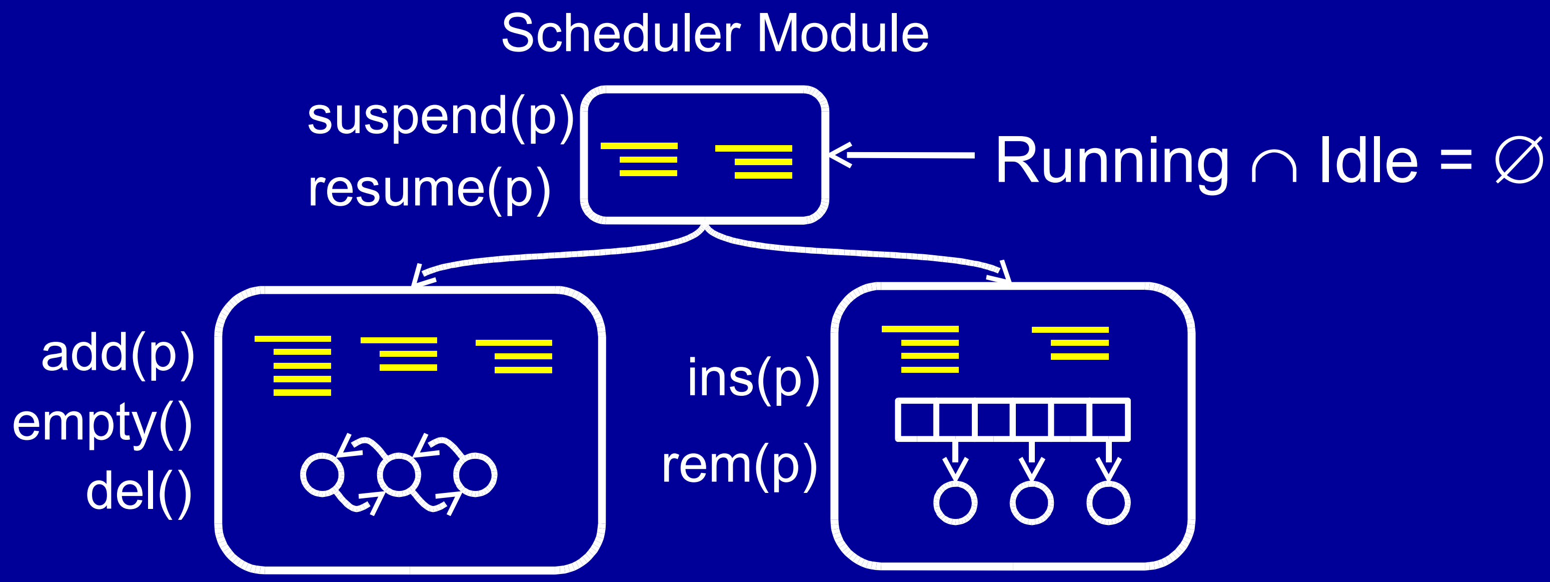
}

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

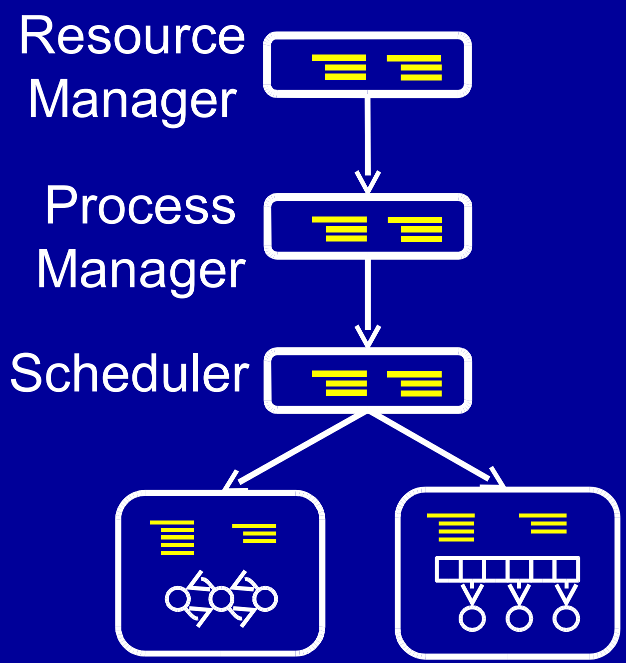
Scopes and Analysis

System conjoins invariant to preconditions and postconditions of exported modules

Analysis verifies procedures preserve invariant



Specification Aggregation



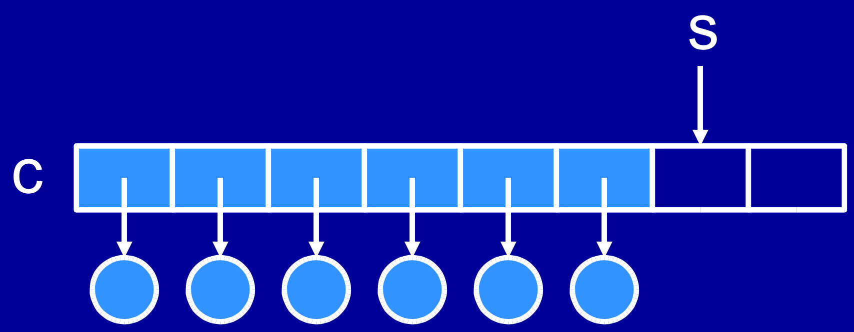
- Hierarchy of modules
- Standard approach:
  - Weave into preconditions through program
  - Weave into call sites where they are needed

Result is that specifications aggregate, moving up the hierarchy

- Scope invariants eliminate specification aggregation for invariant properties

Guards

Consider an array-based data structure.



Must allocate the array before calling data structure operations!

proc init() ensures Init';

proc add(p) requires Init ... ;

Guards and Defaults

- Hob supports boolean guard variables:
  - Initially false
  - Program can explicitly set to true or false
  - Set constraints can use guards
- Annoying to always have to mention guard
- Hob supports defaultly-true properties
  (explicitly *suspended* when not known to be true)

For more on Scopes and Defaults

See our AOSD '05 submission:

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

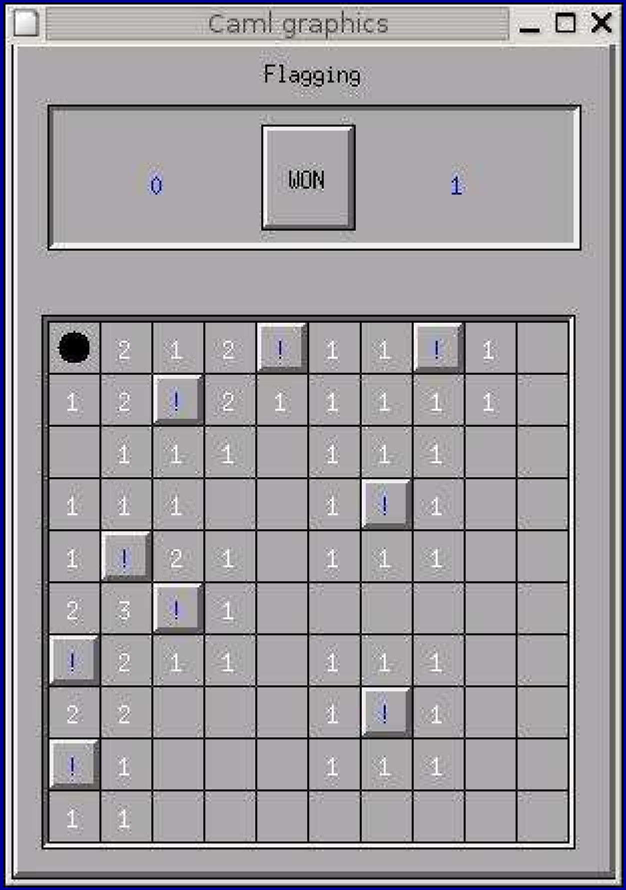
Hob Framework & Benchmarks

- 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, priority queue)

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

Related Work

- Shape analyses
  - Moeller, Schwartzbach PLDI 2001
  - Ghiya, Hendren POPL 1996
- Array analyses
  - Gupta, Mukhopadhyay, Sinha PACT 1999
  - Rugina, Rinard PLDI 2000
- 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 Contributions

Hob framework for modular program analysis:

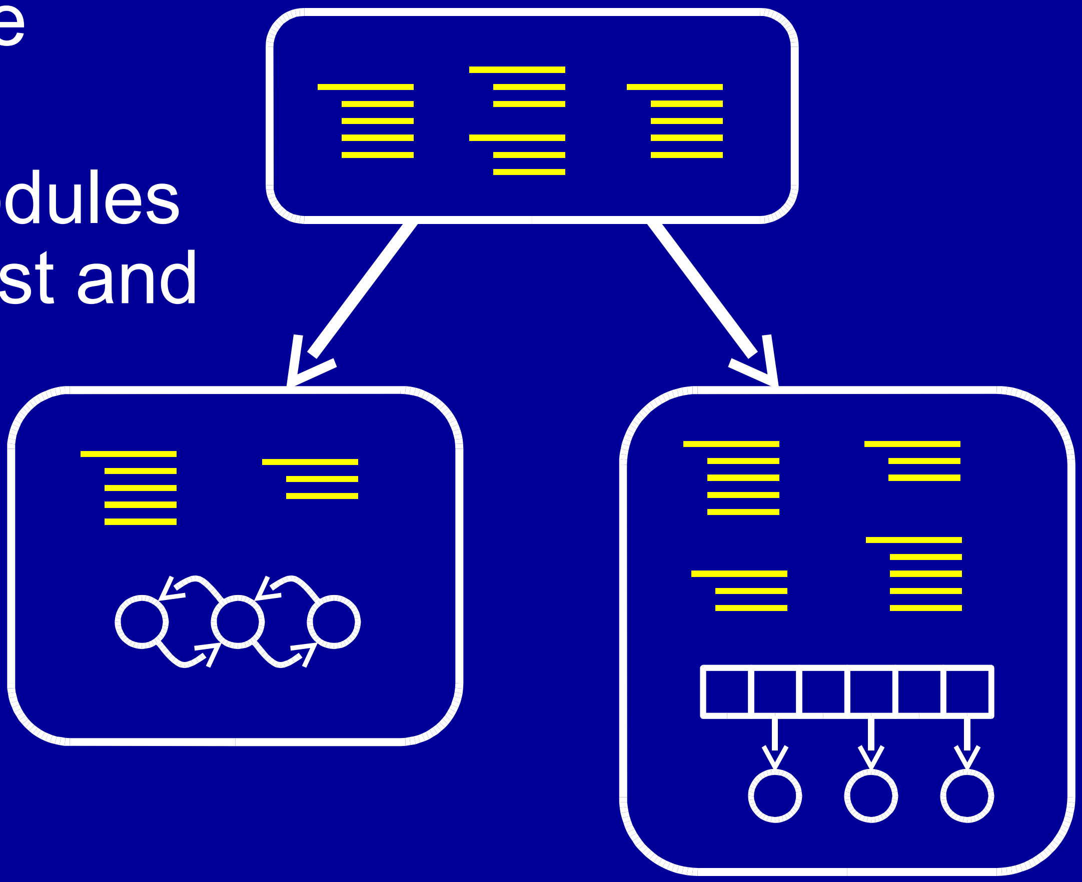
- Verifies data structure consistency properties
- Enables multiple (very precise and unscalable) analyses to interoperate
- First system to combine high-level properties from markedly different analyses

http://cag.csail.mit.edu/~plam/hob

Goal

Verify System Data Structure Consistency

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

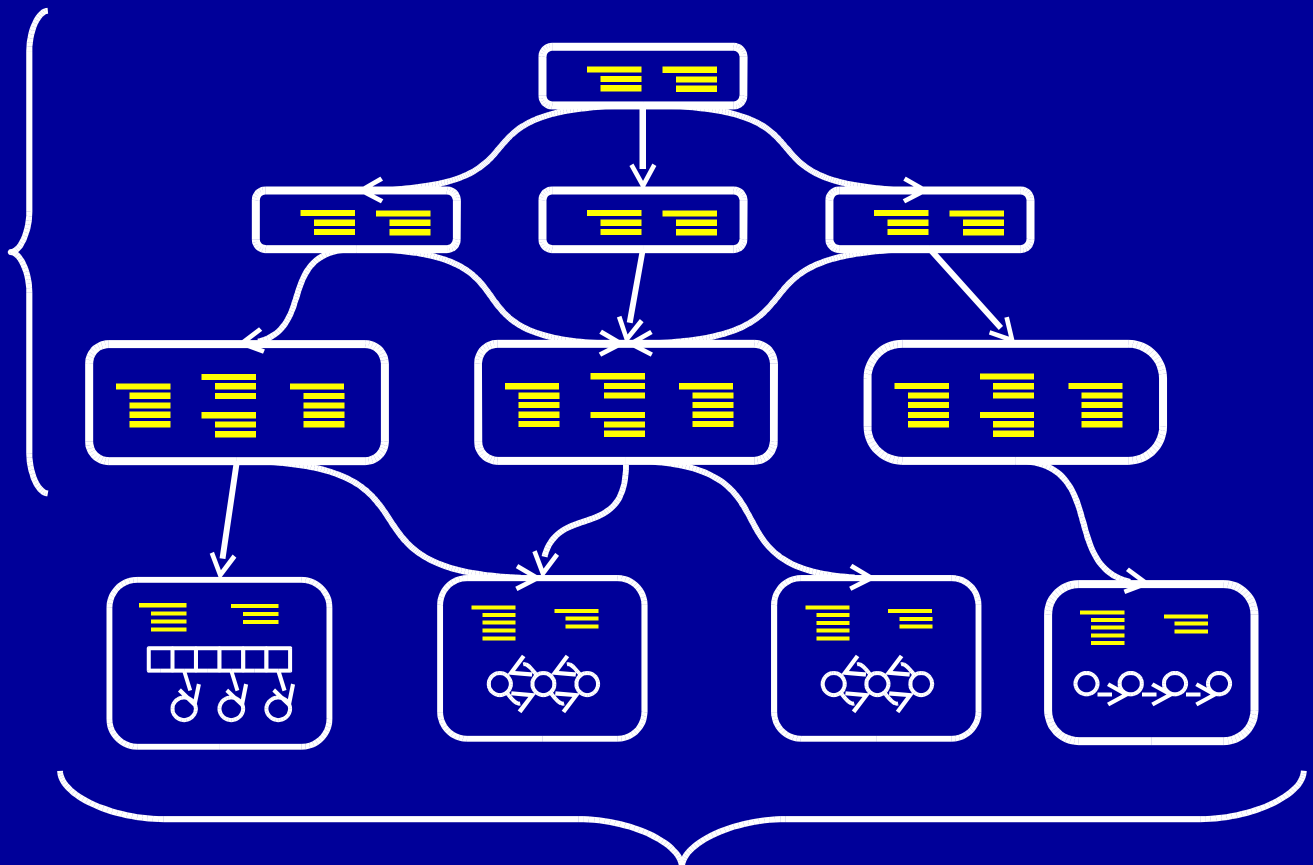


Issues

- Scalability
  - Powerful analyses can verify complex data structure consistency properties
  - Very detailed model of data structures
  - Unthinkable to analyze complete program
- Diversity
  - Different analyses for different data structures
  - New analyses for new data structures
- Need for multiple targeted local analyses that interoperate to share information

Standard Usage Scenario

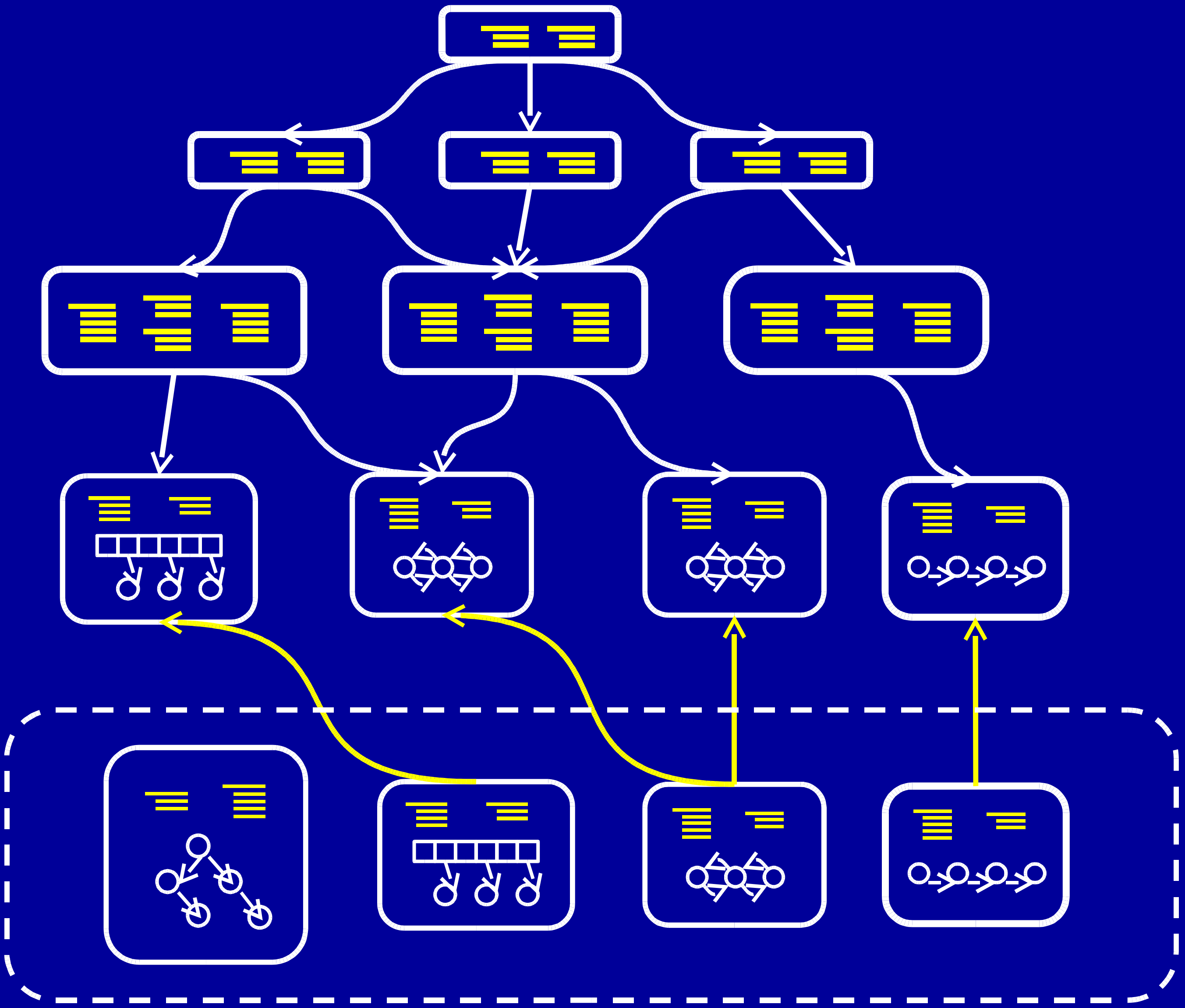
Modules
Coordinate
Data
Structure
Operations



Leaf Modules Encapsulate Data Structures

- Specification/analysis complexity in leaf modules
- Most developers never even see this complexity

Standard Usage Scenario



Module Library

Coordination Modules

- 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

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

- Extension of Set Analysis plugin
- Set membership given by values of primitive fields
- Example set (from Minesweeper):

  ExposedCells = { x : Cell | x.isExposed = true }

  UnexposedCells = { x : Cell | x.isExposed = false }
- Also works for integer flags
- Analysis
  - Same abstract set machinery as Set Analysis plugin
  - Also update sets when flags change

    x.f = false =>

    ExposedCells' = ExposedCells – x

    UnexposedCells' = UnexposedCells $\cup$ x

Analyzing Coordination Modules

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

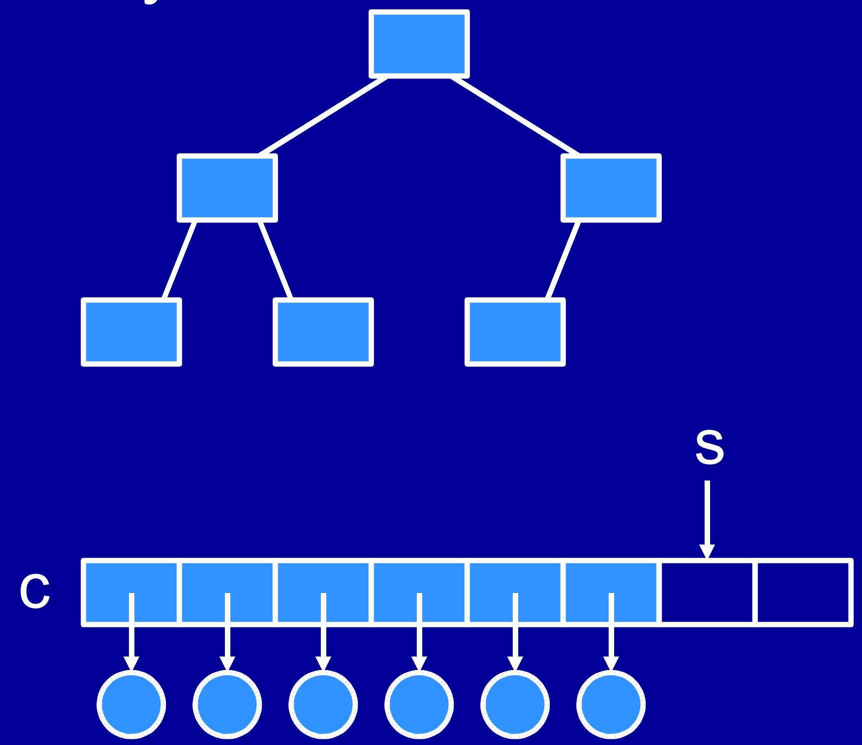
More details in VMCAI '05,

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

Some abstraction modules are even more complicated!

Heap 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

```
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”
}
```

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”.

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)

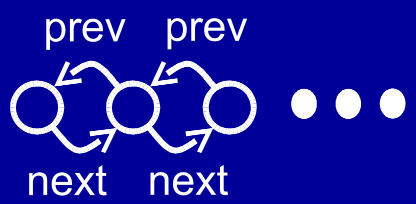
abst module idle { analysis PALE;

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

  invariant type L = {

    data next : L;

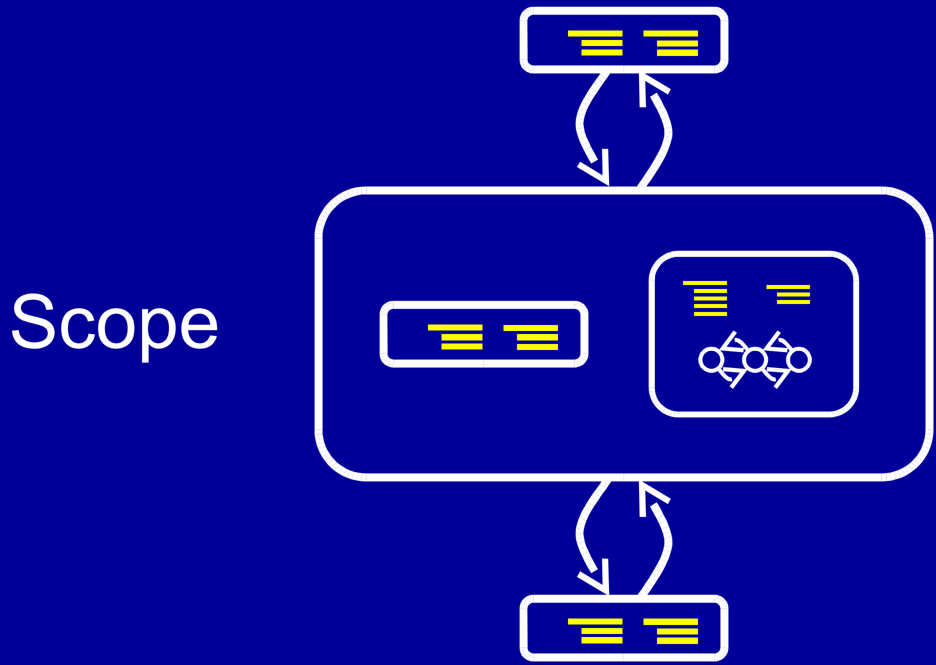
    pointer prev : L [this^L.next = {prev}];



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

Outcalls

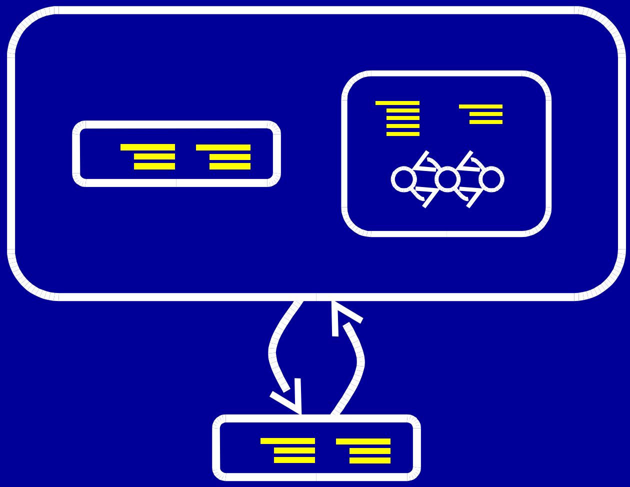
- So far, all calls enter and exit scopes from top
- What about outcalls from scope?



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