Modular Pluggable Analyses

Patrick Lam, Viktor Kuncak and Martin Rinard

MIT CSAIL

Two Kinds of Analyses

Precise Analyses

Sophisticated properties

- Data representation properties
- Correct operation sequencing

Small programs

Examples:

- Shape analysis
- Model Checking

Scalable Analyses

Simple properties

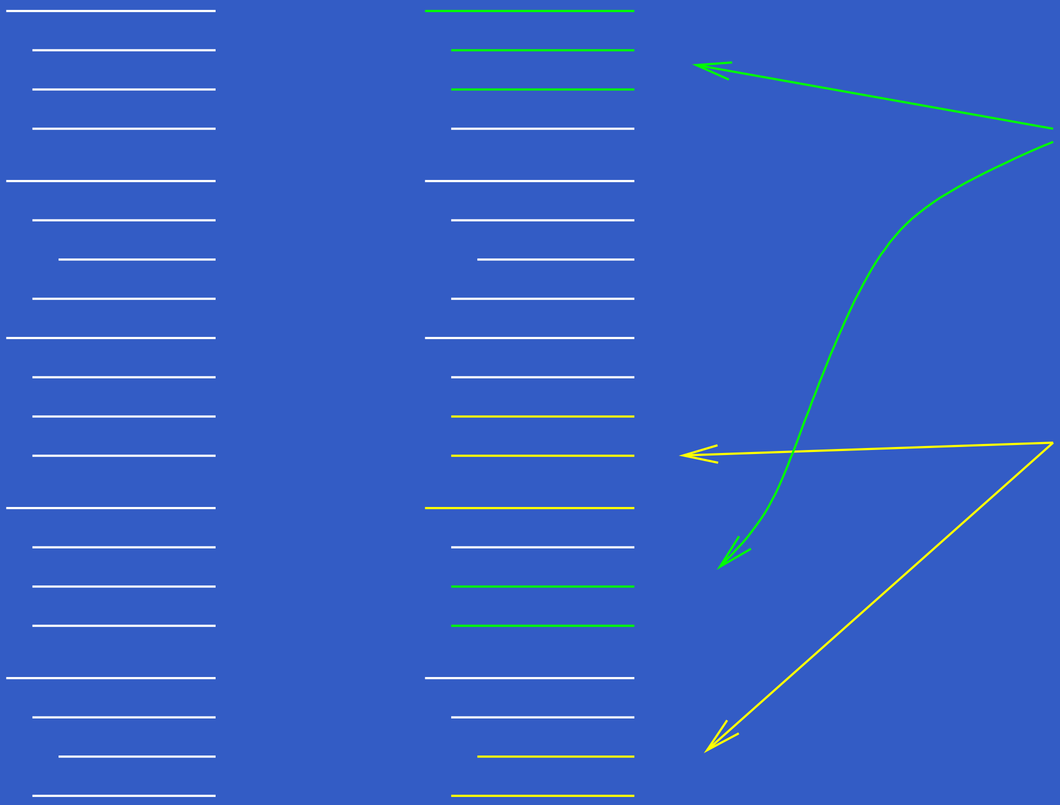
Large programs

Examples:

- Flow-insensitive pointer analysis
- Rapid type analysis

Is This Unavoidable?

For any property, have three kinds of regions:



Crucial Regions

Must analyze precisely to verify property

Relevant Regions

Must propagate simple information through region to verify property

Neutral Regions

No effect on property

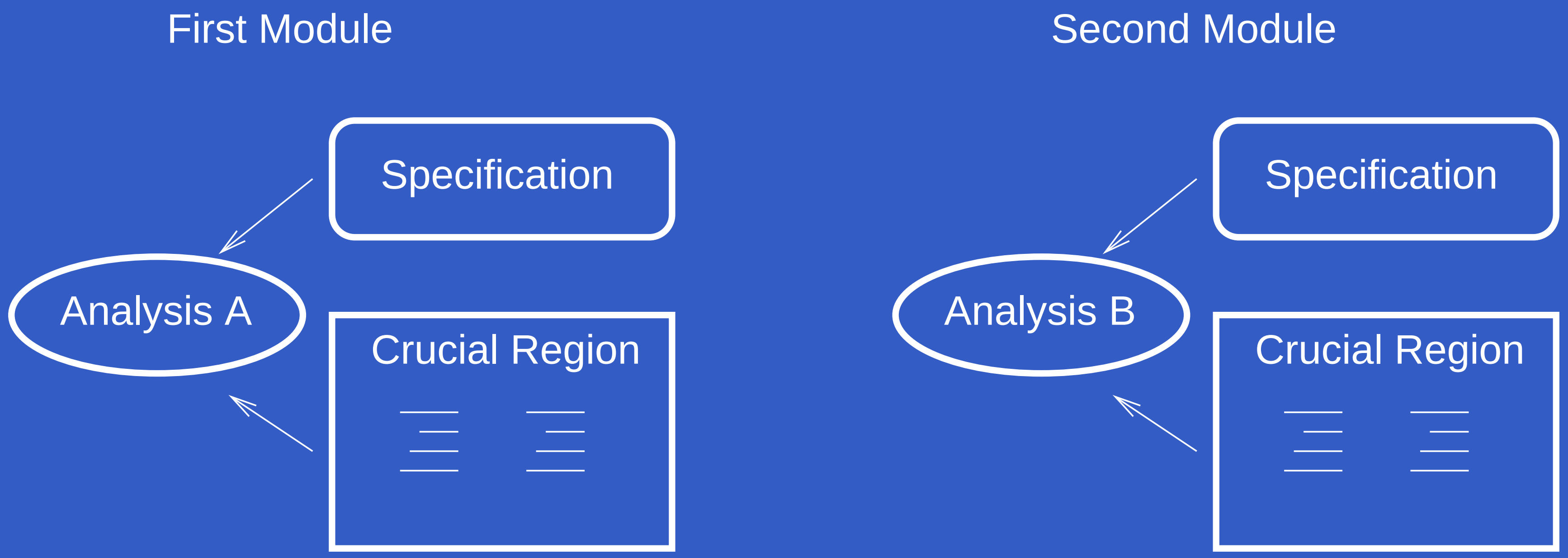
Should be able to exploit this structure

First Approach

Encapsulate each crucial region in a module

Analyze each module independently to verify properties (worst-case assumptions)

Each module uses an analysis appropriate for the important properties of that module



Applying First Approach

```
module List {
   reference root : Node;
   proc add(n : Node) {
   if (root==null) {
     root = n; n.next = null;
     n.prev = null;
    } else {
     n.next = root; root.prev = n;
     n.prev = null; root = n;
    }

   proc remove(n : Node)
    { ... }
}
```

Starting point:

- Implementation; and
- Property to verify:
  (doubly-linked list invariant)

  for all x in root.next*:
    x.prev.next = x and
    x.next.prev = x

Goal: use a shape analysis to verify invariant

- Assume invariant
- Analyze each method to see if it preserves invariant

Applying First Approach

```
module List {
   reference root : Node;
   proc add(n : Node) {
   if (root==null) {
     root = n; n.next = null;
     n.prev = null;
    } else {
     n.next = root; root.prev = n;
     n.prev = null; root = n;
    }

   proc remove(n : Node)
    { ... }
}
```

Analysis fails — needs preconditions

- add(n) — requires n not in list
- remove(n) — requires n in list
- Preconditions controlled by caller, not within List module

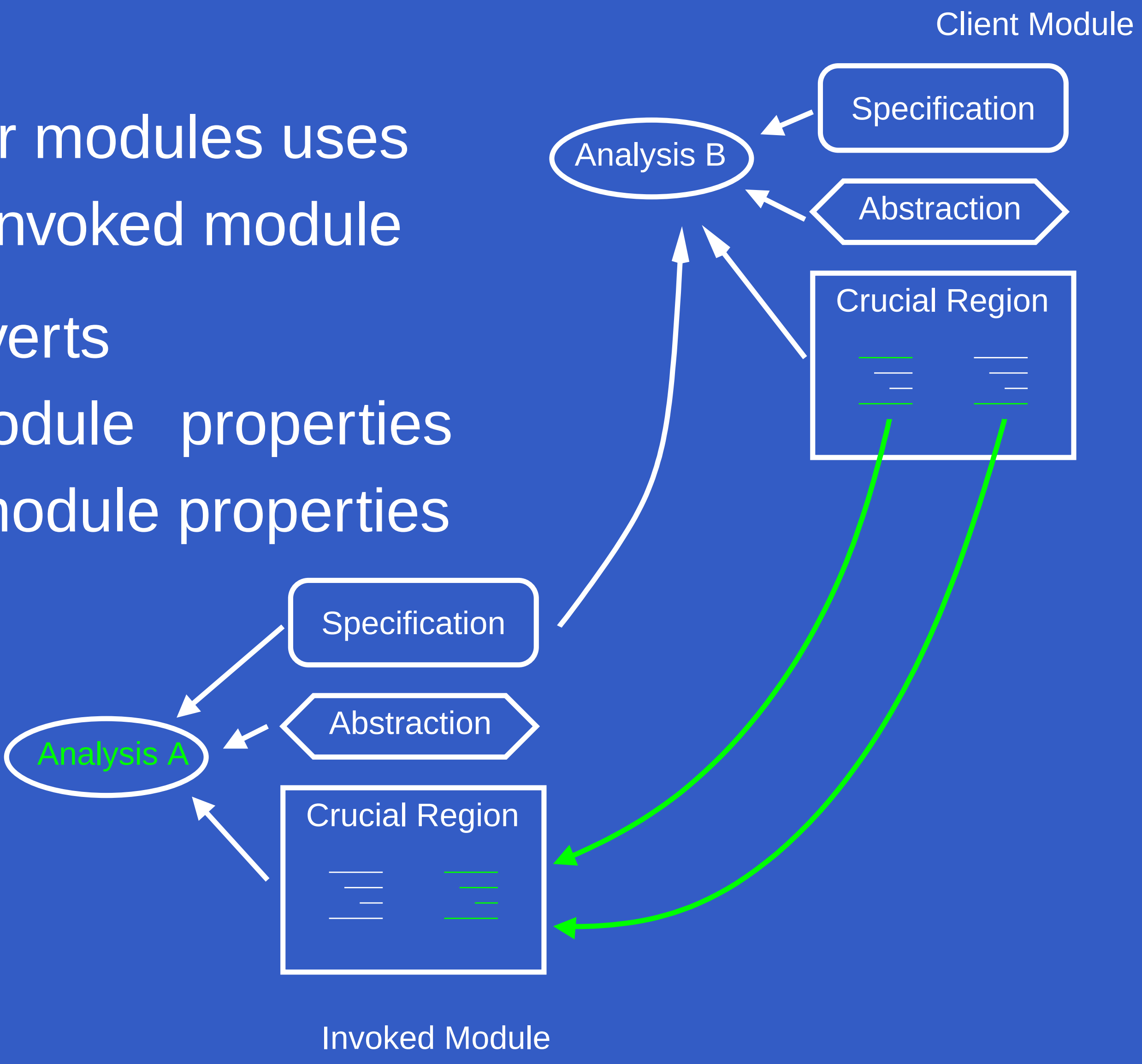
What this example tells us:

- Modules share objects
- Need to verify properties that span multiple modules
- Analyses must communicate
- Intermodule properties much simpler than intramodule properties

Our Approach

Key Points:

- Analysis of caller modules uses specification of invoked module
- Abstraction converts detailed intramodule properties to simpler intermodule properties



Our Approach

Each module has three parts:

Specification part

- Identifies abstract sets of objects
- Uses sets to specify requires and ensures clauses for each procedure

Abstraction part

- Defines sets in terms of concrete data structure
- Specifies representation invariants
- Specifies analysis plugin

Implementation part

- Implements computation
- Standard imperative language

Implementation Part in Example

```
implementation module List {
  format Node { next : Node; prev: Node; }
  reference root : Node;
  proc add(n : Node) {
    if (root==null) {
      root = n; n.next = null;
      n.prev = null;
   } else {
      n.next = root; root.prev = n;
      n.prev = null; root = n;
   }

  proc remove(n : Node) { ... }
}
```

Specification Part in Example

```
specifi cation module List {
  sets L:Node; // set L of objects in list
  proc add(n:Node) // procedure interface
    requires not n in L
    ensures L' = L + n
  proc remove(n:Node) // procedure interface
    requires n in L
    ensures L' = L - n;
}
```

Procedure interfaces:

- First-order formulas in boolean set algebra
- Decidable, yet expressive

Abstraction Part in Example

Defi nes sets in terms of concrete data structure

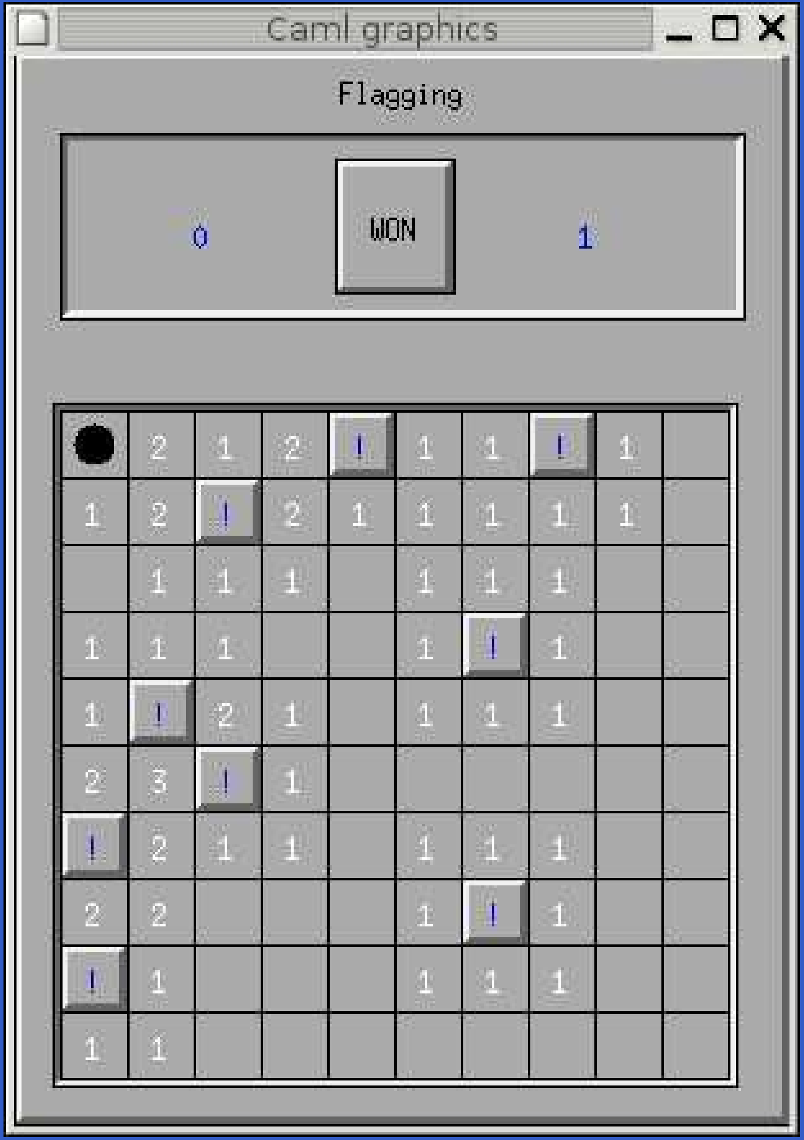
$$L = \{ n \mid n \text{ in root.next}^* \}$$

- Specifi es representation invariant – standard acyclic doubly linked list property
- Set and representation invariant languages specifi c to analysis plugin

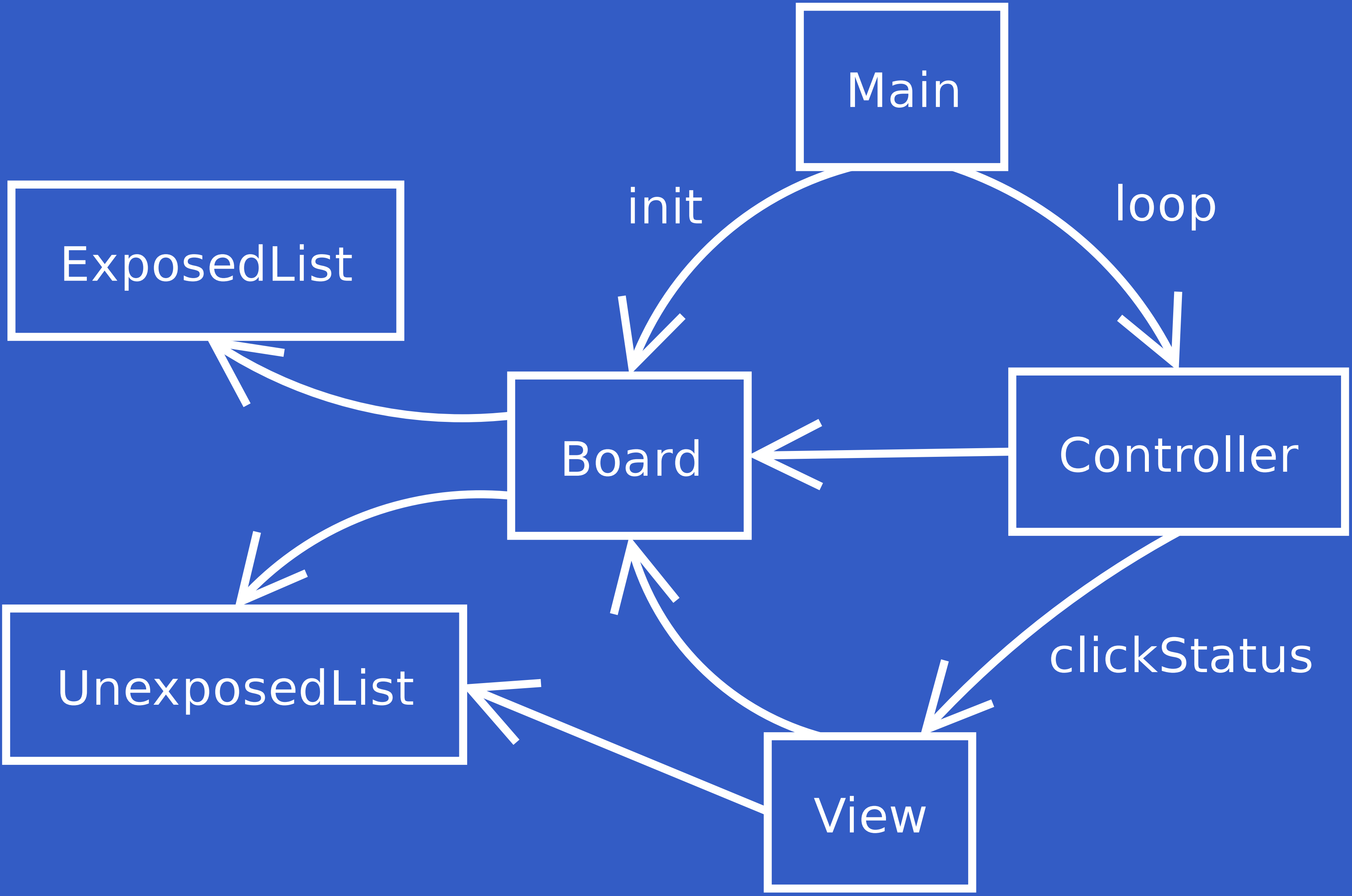
Key Elements

- Each analysis needs only specs of callees
- Flexible with respect to representation of abstract sets

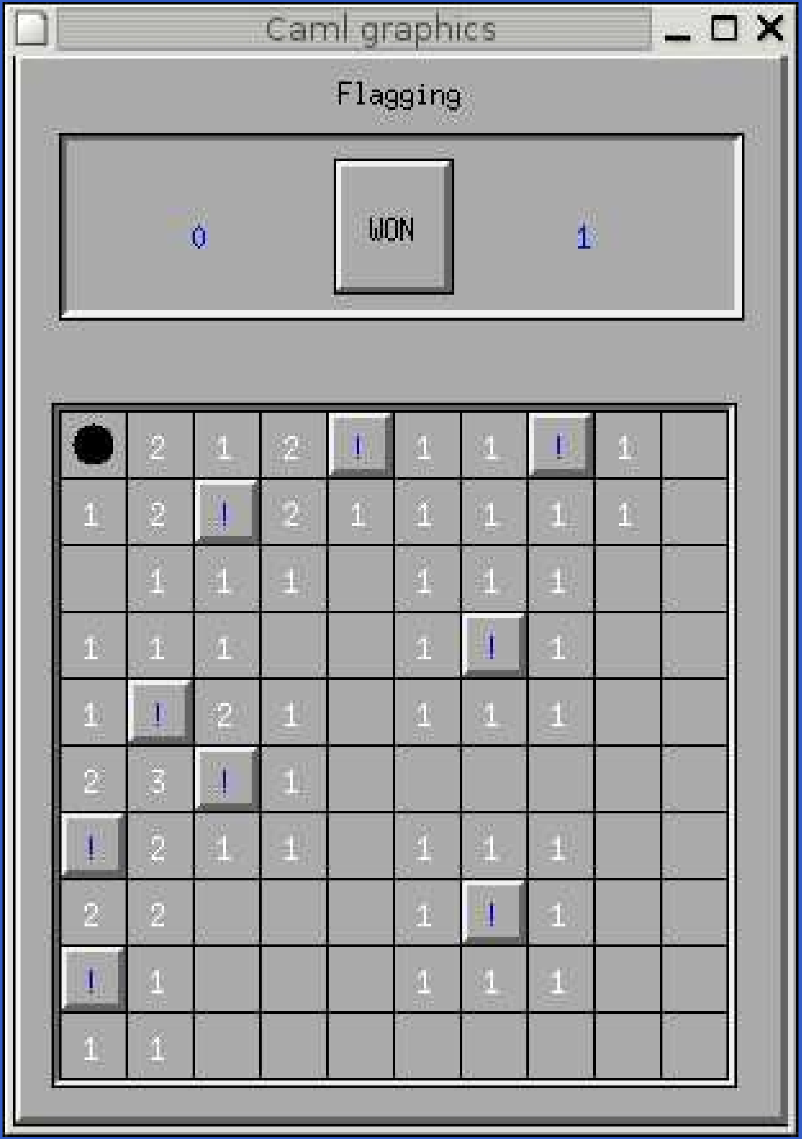
Minesweeper



Minesweeper: Modules



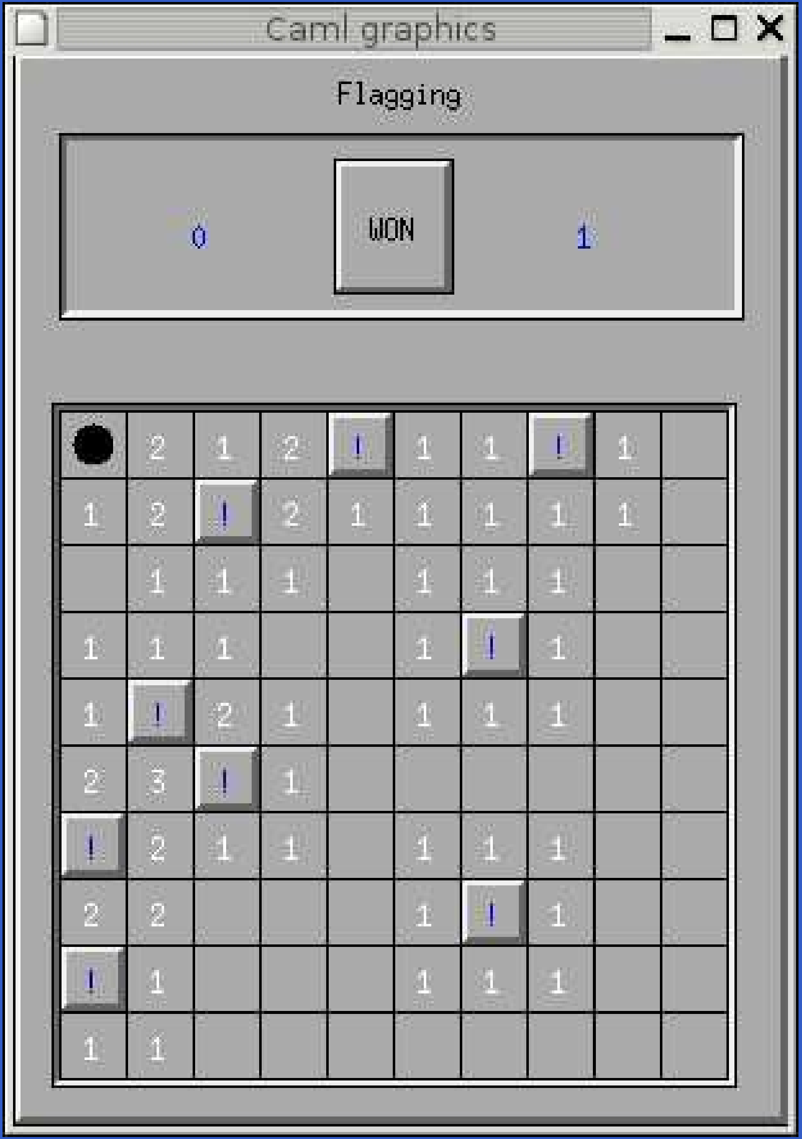
Minesweeper: State



The Board stores an array of Cells:

```
format Cell {
 isMined: bool;
 isExposed: bool;
 isMarked: bool;
}
```

Minesweeper: More State



Also have lists of cells:

```
module ExposedList {
 format Cell {
  next,prev: Cell;}
}

module UnexposedList {
 format Cell {
  next,prev: Cell;}
}
```

Challenge: Module Interaction

We allow modules to share objects.

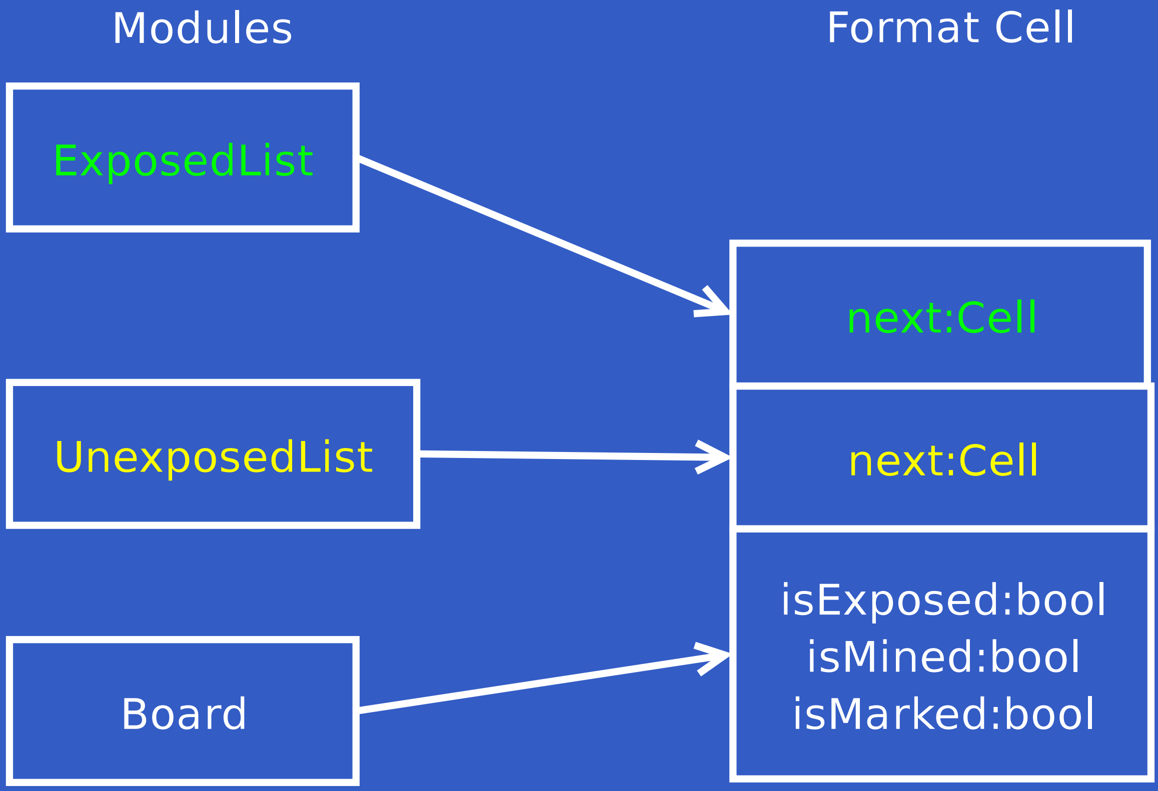
How do we know that other parts of program don’t break our invariants, especially through aliases?

- Each module has its own set of fi elds within the object.

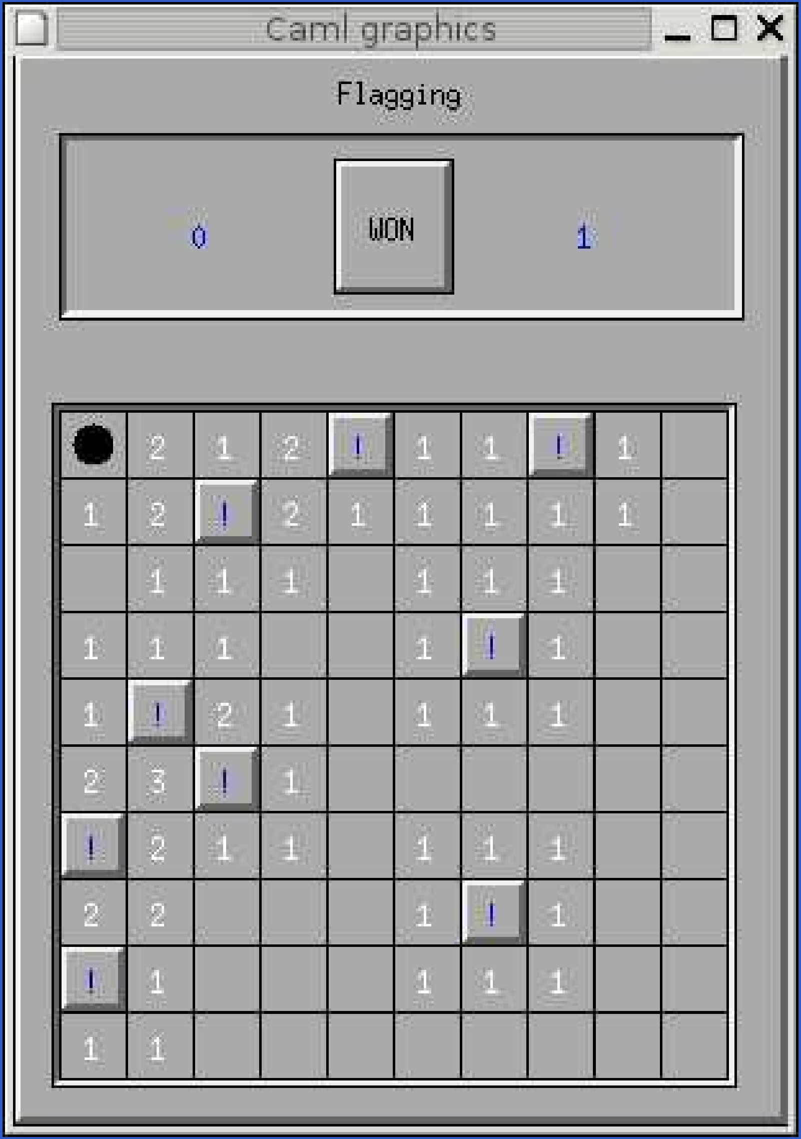
```
impl module ExposedList {format Cell {next:Cell;}}
impl module UnexposedList {format Cell {next:Cell;}}

impl module Board {
  format Cell { isExposed:bool; isMined:bool; ...  }
}
```

Illustration: Formats

Declare the fi elds of **`Cell`** across multiple modules:



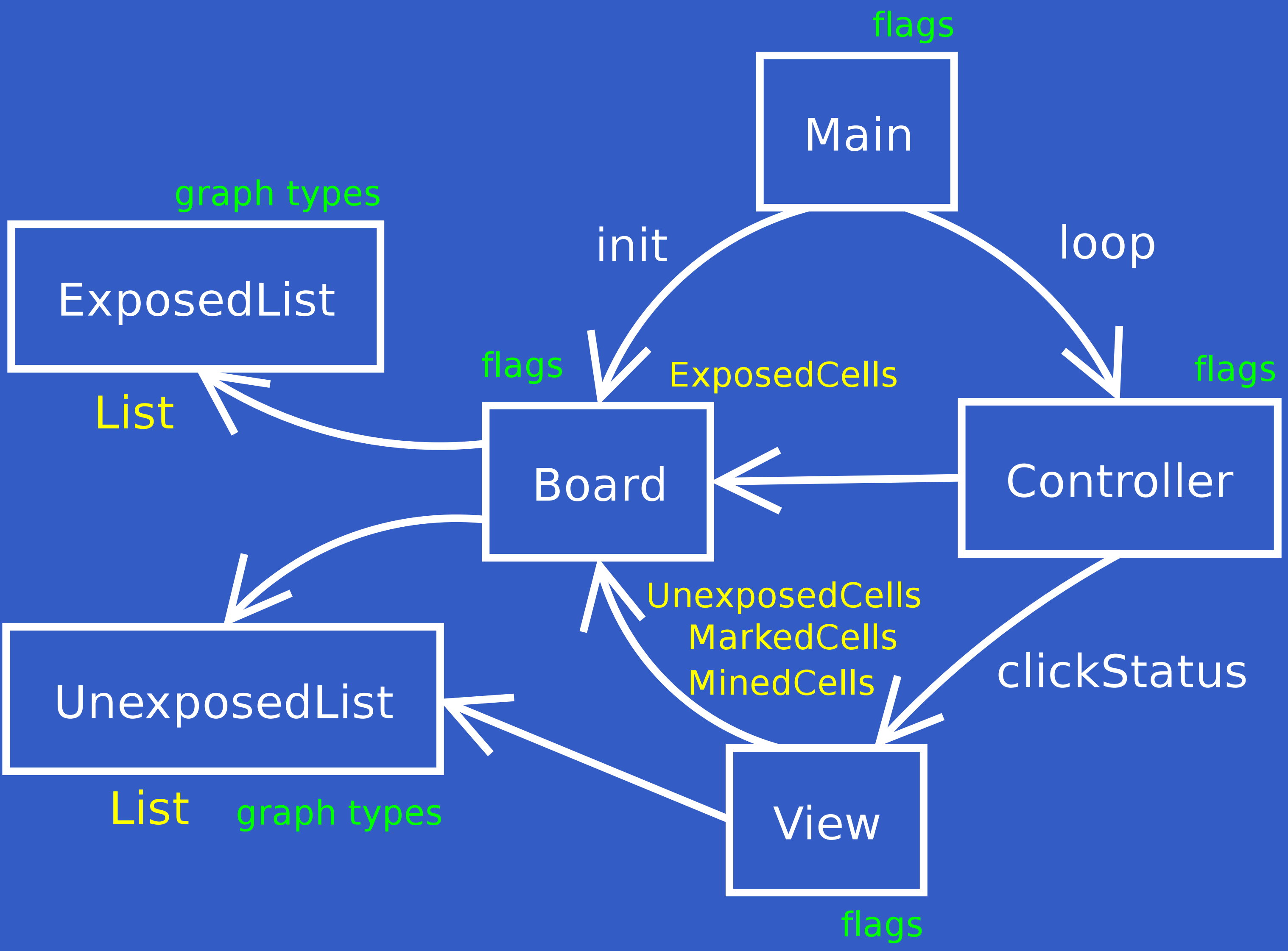
Minesweeper: Sets



Sets of `Cells`:

- AllCells
- ExposedCells
- ExposedList
- UnexposedCells
- UnexposedList
- MarkedCells
- MinedCells

Minesweeper: Modules



Minesweeper: Invariants

We maintain invariants on these sets:

- $\text{ExposedCells} = \text{ExposedList}$
- $\text{UnexposedCells} = \text{UnexposedList}$
- $\text{ExposedList} \cup \text{UnexposedList} = \text{AllCells}$
- $\neg$ **`markSwitchOn`** $\Rightarrow$
  $\text{MarkedCells}' = \text{MarkedCells}$
- $\text{MinedCells} \cap \text{ExposedCells} \neq \emptyset \Rightarrow$
  `gameOver`
- **`gameOver`** $\Rightarrow \text{UnexposedCells} = \emptyset$

Set Specification Language Example

Selected minesweeper procedure specifi cations:

```
proc Board.setMarked(c:Cell; v:bool)
 requires true
 modifies MarkedCells
 ensures v <=> c in MarkedCells;

proc View.drawFieldEnd()
 requires gameOver
 ensures card(UnexposedList.Content) = 0;
```

Ensuring Conformance

Analysis plugins show that procedures satisfy their specifi cations.

- Flag Plugin: Set membership is determined by values of integer flags.

  ```
  E = {x:Cell | x.seen=2};
  ```

- Graph Types Plugin: Set membership is determined by object reachability.

  ```
  Content = {x:Cell | root<next*>x};
  ```

In general, can use arbitrary plugins to analyze arbitrarily sophisticated and precise properties.

Plugin: Flags

Fields determine set membership:

```
abst module Board {
 use plugin flags;
 Exposed = {x:Cell | x.seen=2};
 Unexposed = {x:Cell | x.seen=1};
}
```

Dataflow analysis over 1$^{\text{st}}$-order theory of boolean formulas on sets:

$$[\![\texttt{x.seen = 2}]\!] = \text{Exposed}' = \text{Exposed} \cup \{x\}$$

Transfer Functions for Flag Plugin

Incorporation: Apply quantifi er elimination to

$$B \circ B_s = \exists \hat{S}_1, \ldots, \hat{S}_n.\ B[S_i' \mapsto \hat{S}_i] \wedge B_s[S_i \mapsto \hat{S}_i]$$

Applying set membership updates:
(let **`R = {x:t | x.f=c}`**)

$$\mathcal{F}(\mathbf{x.f}) = \mathbf{c} \;:=\; R' = R \cup \mathbf{x} \wedge \bigwedge_{S \in \mathsf{alts}(\mathbf{R})} S' = S \setminus \mathbf{x}$$

Generic transfer function:

$$[\![\mathrm{st}]\!](B) = \alpha(B \circ \mathcal{F}(\mathrm{st}))$$

Plugin: Graph Types

Analyzes specifi cations where the abstraction function is given by a regular expression over the heap.

```
proc add(e : Entry)
  requires not (e in Content)
  modifies Content
  ensures Content’ = Content + e;
```

where

```
   Content = {x:Entry | root<next*>x}
```

Uses MSOL over trees.

Experience

We have implemented a prototype system and tested computational patterns inspired by:

- compiler transformations
- CTAS
- water
- minesweeper

Experience

Most discovered errors were specifi cation errors.

Found some errors in the implementation.

At one point, we inadvertently changed the abstraction function and only partially updated the code. The tool found this error.

Generalized Typestate

We use sets in our view of the world, rather than attaching typestates to objects:

Typestate	Generalized Typestate
typestate$(x) =$ Exposed	$x \in$ Exposed

Generalizing typestate

Orthogonal Composition

Typestate	Generalized Typestate
Single atomic typestate	Multiple simultaneous typestates
	$x \in \text{Marked} \wedge x \in \text{Flagged}$

Sharing and Typestates:

No aliasing	Use simultaneous typestates for orthogonal memberships

Hierarchical Typestates:

Partial order	Subset inclusion
$T \leq S$	$S \subseteq T$

Expressive power of sets

Propagating boolean formulas enables (local) reasoning about object aliasing.

$$x = y$$

Can count cardinalities of sets.

$$|S| = 0; |S| = 1; |S| \geq 1$$

Conclusion

Goal: Enable the modular, scalable use of the full range of precise program analyses.

[Language design] Formats allow modules to share objects but not fi elds.

[Common set language] Analyses communicate using set algebra specifi cations.

[Verifi cation approach] Check that each module implements its specifi cation, using pluggable analyses and abstraction functions.