Vérification statique des propriétés de conception

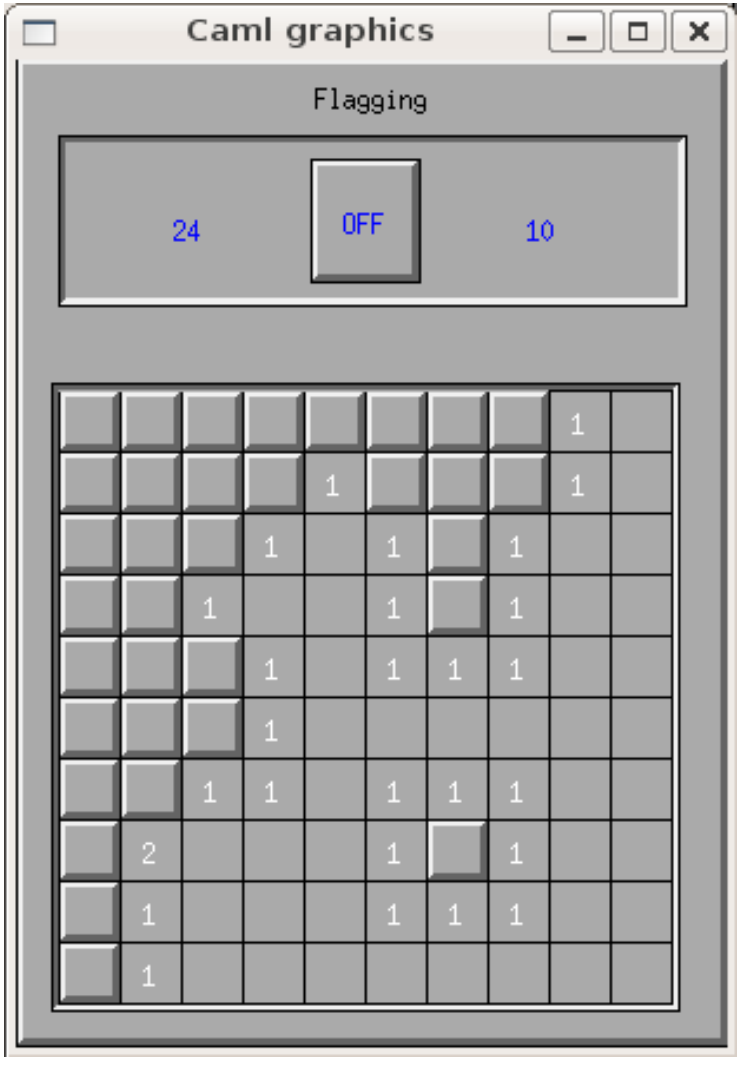
Patrick Lam
McGill University

4 avril 2007

But

Vérifier des propriétés de conception des systèmes logiciels.

Démineur : une propriété de conception



Les cellules minées ne sont jamais exposées,
sauf si la partie est terminée.

Extrait d'une implémentation de Démineur

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

  var cells:Cell[][];
  var gameOver:bool;

  proc getMined(c:Cell)
       returns m:bool {
    return c.isMined;
  }

  proc setGameOver() {
    gameOver = true;
  }

  // ...
}
```

Témoin *gameOver*

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

  var cells:Cell[][];
  var gameOver:bool;

  proc getMined(c:Cell)
       returns m:bool {
    return c.isMined;
  }

  proc setGameOver() {
    gameOver = true;
  }

  // ...
}
```

Le témoin `gameOver` est vrai lorsque la partie est terminée.

Cellules minées

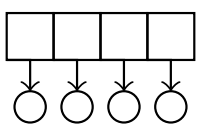
```
impl module Board {
  format Cell {
    isMined:bool;
    // ...
  }

  var cells:Cell[][];
  var gameOver:bool;

  proc getMined(c:Cell)
       returns m:bool {
    return c.isMined;
  }

  proc setGameOver() {
    gameOver = true;
  }

  // ...
}
```

Les cellules minées sont indiquées par une valeur `vrai` à leurs témoins `isMined`.

Structure de données : cellules exposées

```
impl module ExposedCells {
  format Cell {}

  reference d:Cell[];
  var s:int; var m:int; var full:bool;

  proc add(e:Cell) {
    d[s] = e;
    s = s + 1;
    if (d.length <= s)
      full = true;
  }

  // ...
}
```

Nous entreposons les cellules exposées dans un tableau `d`.



Propriété de conception Démineur

$$\texttt{Board.gameOver} \vee \mathsf{disjoint}(\{x : \texttt{Cell} \mid x.\texttt{isMined}\langle\texttt{Board}\rangle = \mathit{true}\}, \{x : \texttt{Cell} \mid \exists j.\ 0 \leq j \wedge j \leq s \wedge x = \texttt{ExposedCells}.d[j]\})$$

Propriété Démineur : langage des ensembles

$$\texttt{gameOver} \quad | \quad \text{disjoint}(\texttt{MinedCells}, \texttt{ExposedCells})$$

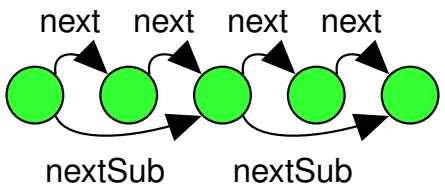
Fonctions d'abstraction

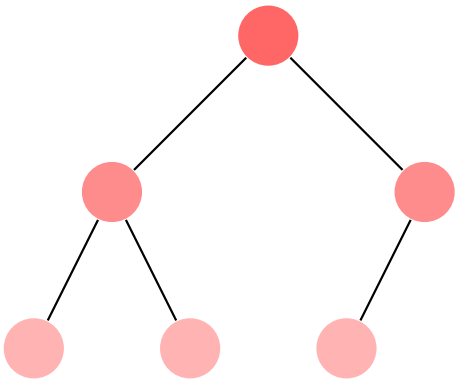
```
abst module Board {
  U = { x : Cell | "x.init = true" };
  MinedCells = U cap { x : Cell | "x.isMined = true" };
  predvar gameOver;
}

abst module ExposedCells {
  Content = { x : Node | "exists j. 0 <= j & j < s & x : d[j]"};
}
```

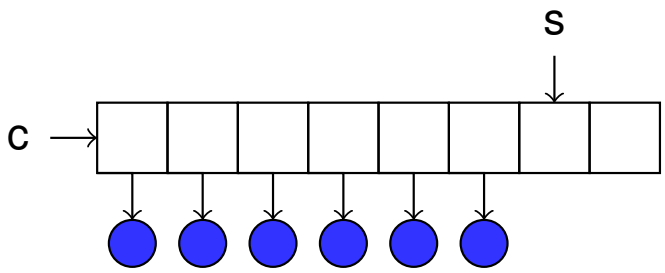
Propriété Démineur : langage des ensembles

$$\texttt{gameOver} \quad | \quad \text{disjoint}(\texttt{MinedCells}, \texttt{ExposedCells})$$

Manipulations des ensembles

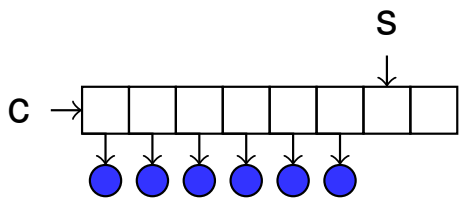







```
proc add (p:Node)
  requires p not in Content & card(p)=1 & (not full)
  modifies Content, full
  ensures Content' = Content + p;
```

Génération des conditions de vérification




```
proc add(p:Node) {
  d[s] = p;
  s = s + 1;
  if (d.length <= s)
    full = true;
}
```

```
proc add (p:Node)
  requires p not in Content
       & card(p)=1 & (not full)
  modifies Content
  ensures Content' = Content + p;
```

```
abst module ExposedCells {
  Content = { x : Node | "exists j. 0 <= j & j < s & x : d[j]"};
  invariant "0 < s";
}
```

1. Générer des obligations de preuve.
2. Prouver les obligations générées.

Vérification de la procédure `add`

```
proc add (p:Node)
  requires p not in Content & card(p)=1 & (not full)
  ensures Content' = Content + p
  {
    d[s] = p;
    s = s + 1;
    if (d.length <= s)
      full = true;
  }

abst module ExposedCells {
  Content = { x : Node | "exists j. 0 <= j & j < s & x : d[j]"};
  invariant "0 < s";
```

Vérification de la procédure `add`

```
proc add (p:Node)
  requires p ∉ {x : Node | ∃j. 0 ≤ j < s' ∧ x = d[j]} ∧ ···
  ensures {x : Node | ∃j. 0 ≤ j < s' ∧ x = d'[j]} =
                {x : Node | ∃j. 0 ≤ j < s ∧ x = d[j]} + p
  {
    d[s] = p;
    s = s + 1;
    if (d.length <= s)
      full = true;
  }
```

Vérification de la procédure `add`

```
proc add (p:Node)
  requires p ∉ {x : Node | ∃j. 0 ≤ j < s' ∧ x = d[j]} ∧ ···
  ensures {x : Node | ∃j. 0 ≤ j < s' ∧ x = d'[j]} =
                {x : Node | ∃j. 0 ≤ j < s ∧ x = d[j]} + p
  {
    d[s] = p;
    s = s + 1;
    if (d.length <= s)
      full = true;
  }
```

Il faudra donc vérifier l'implication suivante:

$$s' = s + 1 \wedge d'[s'] = p \wedge \cdots \Rightarrow$$
$$\{x : \texttt{Node} | \exists j. 0 \leq j < s \wedge x \in d'[j]\} = \{x : \texttt{Node} | \exists j. 0 \leq j < s \wedge x : d[j]\} + p$$

Rôle de l'analyse statique

But : Vérification pratique et automatique des propriétés de conception.

Solution : Utiliser des analyses statiques spécialisées, par exemple des « shape analysis ».

Composants du système Hob

Le système Hob vérifie des propriétés de conception des systèmes logiciels :

- il permet *d'exprimer* ces propriétés avec un langage de spécification basé sur les ensembles ; et
- il *vérifie* ces propriétés avec des techniques d'analyse statique.

Résumé de l'exemple

- Nous avons vu une propriété de conception de Démineur :
  - exprimée en langage naturel ;
  - exprimée directement en termes de l'implémentation ; et
  - exprimée avec des spécifications basées sur les ensembles.
- Des fonctions d'abstraction lient les spécifications et les implémentations.
- Des analyses statiques servent à vérifier que les postconditions des procédures sont satisfaites si les préconditions de ces procédures tiennent.

Première partie I

Les ensembles comme langage de spécification

Langage de spécification fondé sur les ensembles

$$
\begin{array}{rcl}
B & ::= & \text{true} \mid \text{false} \mid \text{var} \\
 & \mid & B \wedge B \mid B \vee B \mid \neg B \mid \exists S.B \mid \forall S.B \\
 & \mid & SE_1 = SE_2 \mid SE_1 \subseteq SE_2 \\
 & \mid & \text{disjoint}(SE_1, SE_2) \mid \text{card}(SE) = k \\
SE & ::= & \emptyset \mid [m.]\, S \mid [m.]\, S' \\
 & \mid & SE_1 \cup SE_2 \mid SE_1 \cap SE_2 \mid SE_1 \setminus SE_2
\end{array}
$$

Préconditions et postconditions des procédures

$$
\begin{array}{lll}
P & ::= & \text{proc } pn(p_1 : t_1, \ldots, p_n : t_n) \\
 & & \quad [\text{returns } r : t\,] \\
 & & \quad [\text{requires } B\,] \\
 & & \quad [\text{modifies } S^+] \\
 & & \quad \text{ensures } B
\end{array}
$$

La notion fondamentale est celle des spécifications des procédures.

Rationale

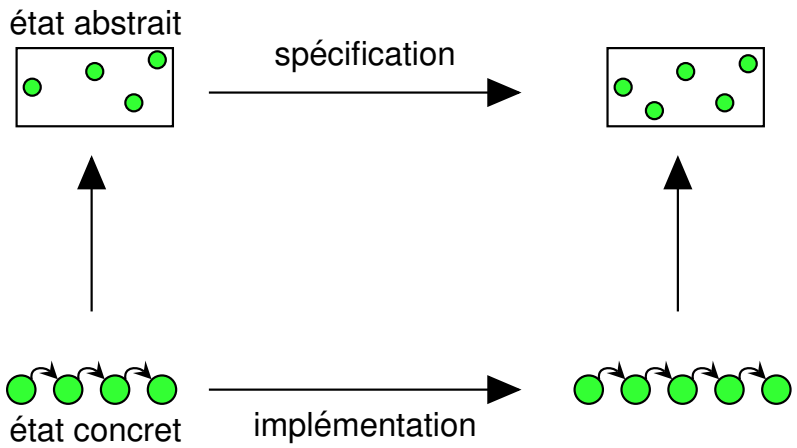
Nous avons choisi notre langage de spécification pour les raisons suivantes :

1. l'état du programme est témoin de ses propriétés de conception ;
2. les ensembles conviennent pour résumer l'état du programme ;
3. l'expressivité limitée des ensembles sert à
   - porter attention sur un aspect des spécifications ;
   - rendre nos spécifications automatiquement vérifiables.

Deuxième partie II

La vérification des specifications

Scénario de vérification

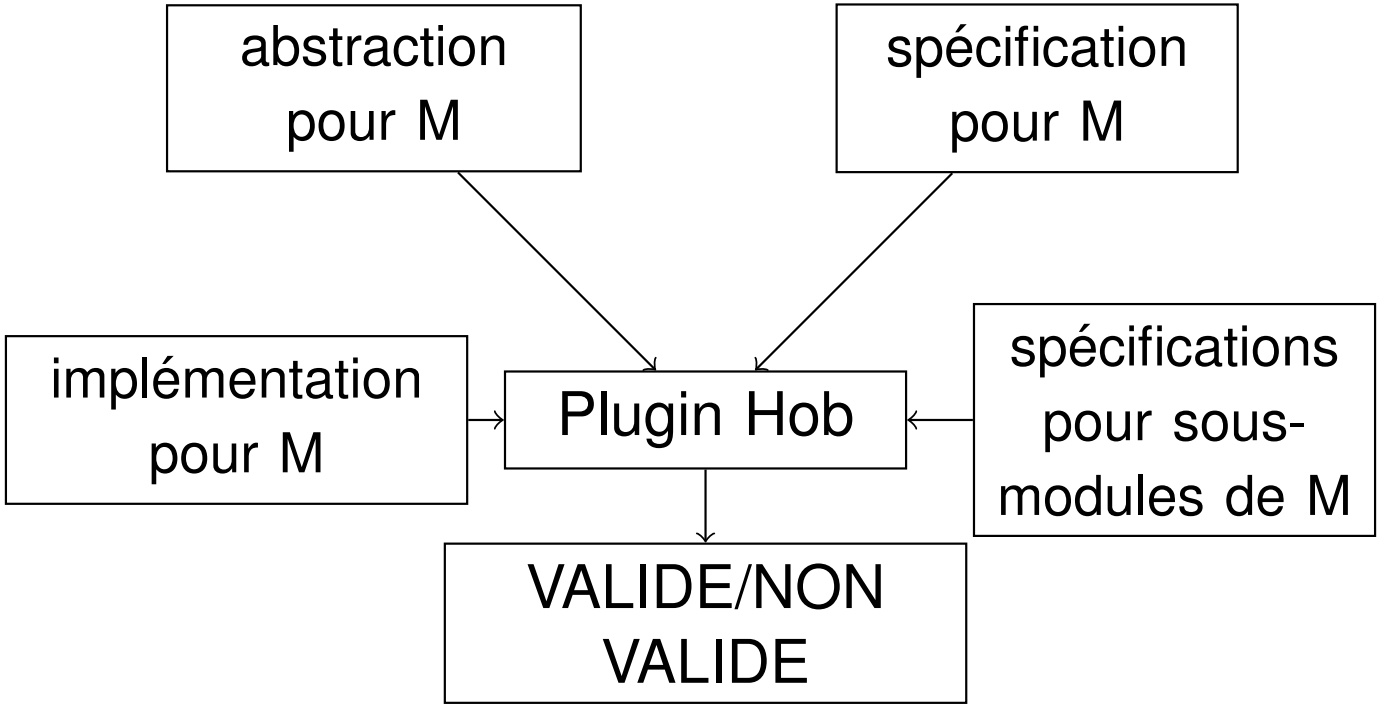


Toute implémentation de la procédure doit conformer à sa spécification :

- si la précondition de la procédure est vraie,
- l'analyse doit démontrer qu'après la termination du procédure, la postcondition du procédure est vraie.

Les procédures devront aussi préserver leurs invariants.

La vérification des programmes



Les plugins lient les implémentations et les spécifications :

- en se servant des fonctions d'abstraction, ils décident si les implémentations garantissent leurs spécifications ;
- assurent que les préconditions nécessaires sont valides.

Les techniques de vérification du système Hob

Le système Hob comprend divers plugins, chacun contenant une technique de vérification :

1. Démonstration de théorèmes : génération des conditions de vérification, qui devront être prouvées par des utilisateurs.
2. Shape analysis : permet le raisonnement automatique pour des implémentations qui manipulent des structures de données (par exemple, listes chaînées).
3. Analyse des témoins : permet la raisonnement pour les modules « coordination », ceux qui se servent des autres modules pour manipuler les structures des données.

Langage d'ensembles + plugins

Donc, Hob se sert des plugins pour vérifier différentes classes d'implémentations.

Notre langage de spécification permet l'application des analyses statiques à des programmes :

- de taille importante—la vérification se fait une procédure à la fois ; et
- avec diverses structures de données—les différentes analyses peuvent communiquer grâce aux spécifications basées sur les ensembles.

Troisième partie III

Résultats ; l'existant ; contributions

Implémentation du système Hob

Nous avons implémenté :

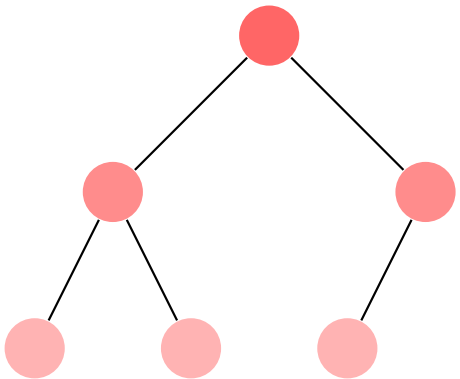
- Interpréteur et compilateur vers Java ;
- Le système Hob et
- Analyses statiques :
  - Analyse des témoins
  - « Shape analysis » Bohne
  - Adapteur pour démonstrateur des théorèmes Isabelle

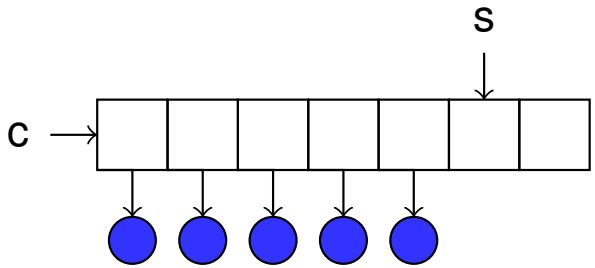
Logiciels vérifiés par Hob

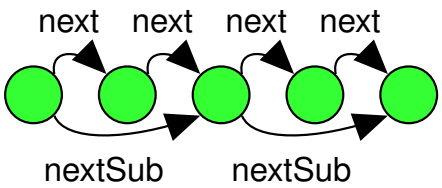
Nous avons vérifié les logiciels suivants :

- Des implémentations des structures de données ;
- Une simulation des molécules d'eau ;
- Un serveur HTTP ; et
- L'application Démineur.

Structures de données





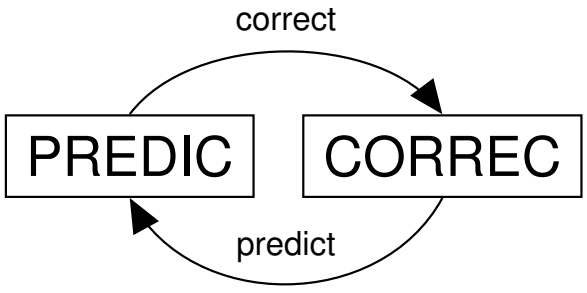


Spécifications partielles des propriétés, tels que :

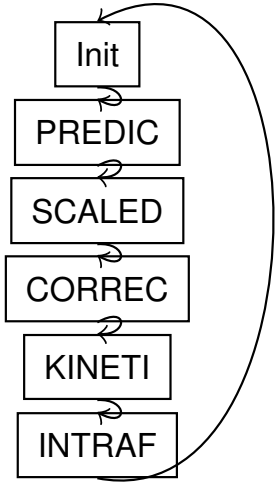
```
proc add(p : Entry)
  requires not (p in Content) & card(p) = 1
  modifies Content
  ensures Content' = Content + p;

proc remove(p : Entry)
  requires (p in Content) & card(p) = 1
  modifies Content
  ensures Content' = Content - p;
```

Simulation des molécules d'eau

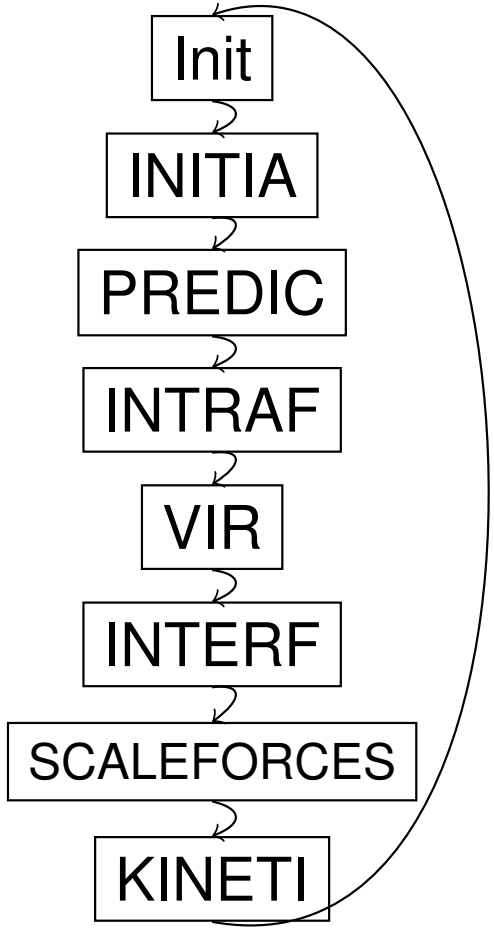
État des atomes :



État des molécules :



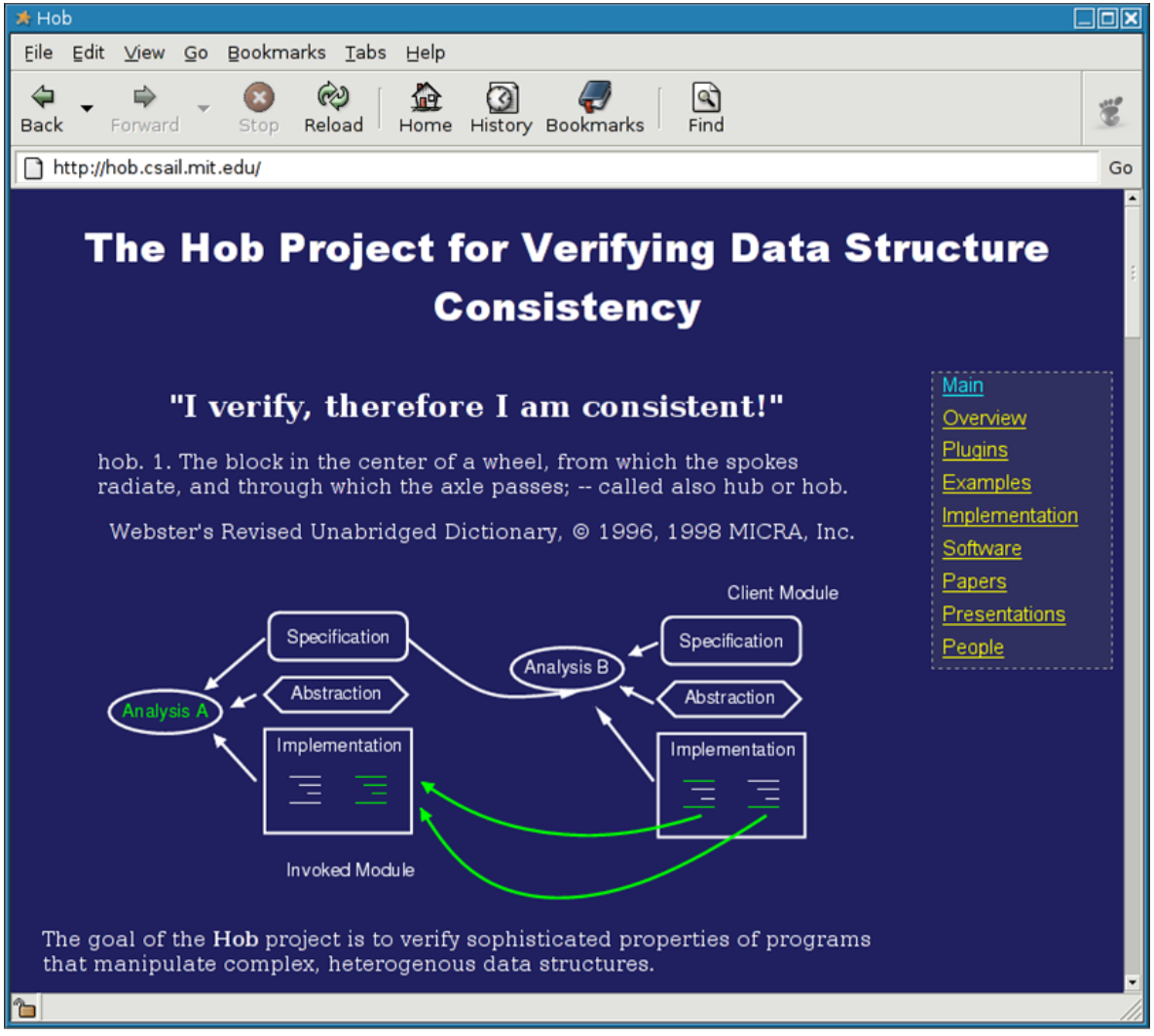
État de la simulation :



Simulation des molécules d'eau

- Lignes de code : 2000 ; lignes de spécification : 500
- Computation par étapes qui simule des molécules d'eau
- Les étapes :
  - Prédiction, correction, et application des limites de position
  - Calculations de forces inter- et intra-moléculaires
- Propriétés « typestate »
  - Les paramètres de la simulation sont initialisés
  - Les atomes portent les états appropriés
  - Les molécules portent les états appropriés
- Corrélations de l'état de la simulation, des atomes, et des molécules

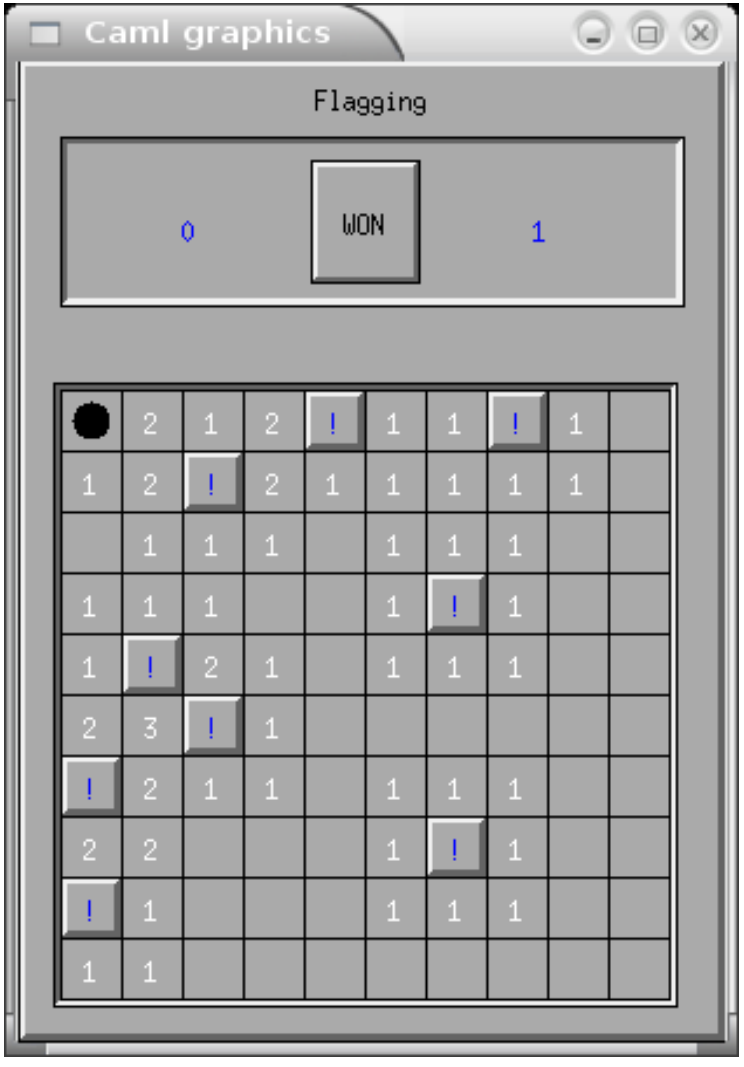
Serveur HTTP



Serveur HTTP

- Lignes de code : 1200 ; lignes de spécification : 430
- Répond à des requêtes HTTP/1.1
- Utilise des listes chaînées comme structure de données
- Propriétés de conception :
  - Les réponses aux requêtes sont servis de la mémoire cache ou ils sont sur une liste noire.
  - Les en-têtes de réponses sont toujours vides avant et après la traîtement des réponses.

Application Démineur



Application Démineur

- Lignes de code : 882 ; lignes de spécification : 384
- Interface graphique qui suit le patron « model/view/controller »
- Propriétés des structures de données :
  - Listes, tableaux de cellules respectent leurs invariants
  - Aucun élément parait deux fois dans le même ensemble ; propriétés de pointeurs
- Propriétés de conception du tableau :
  - Toute cellule est soit exposée soit cachée
  - Aucune cellule exposée est minée sauf si la partie est terminée
- Corrélations entre l'état du programme et ses actions :
  - Les cellules sont initialisées avant le début d'une partie
  - Le tableau n'est pas exposé avant la fin de la partie

L'existant

- Langages de spécification

  Z, VDM, Larch, JML, Modélisation objet (UML)

- Technologies d'analyse statique

  Interprétation abstraite, « typestate », « shape analysis », « model checking », démonstrateurs de théorèmes

- Systèmes de vérification

  Stanford Pascal Verifier, ESC/Java, Spec#

Idées fondamentales du système Hob

- Langage de spécification basé sur les ensembles, qui cible les propriétés de conception ;
- Approche modulaire de vérification ; et
- Gamme de plugins d'analyse statique.

Le futur

- Programmes construits avec processus indépendents (« multithreaded ») ;
- Analyse des propriétés de conception des systèmes embarqués.

Contributions principales

Conception et implémentation du système Hob :

- Langage de spécification des propriétés de conception basé sur les ensembles
- Analyses statiques modulaires pour vérifier les manipulations des ensembles.

hob.csail.mit.edu

Merci à mes collaborateurs :
Martin Rinard
Viktor Kuncak
Karen Zee
Thomas Wies

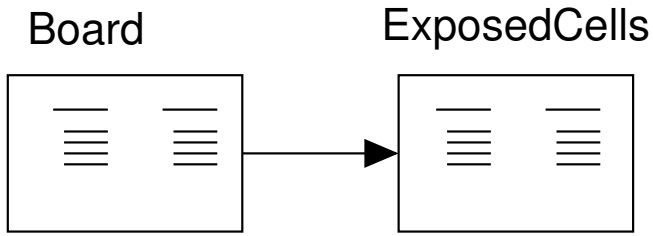
Encapsulation mechanism

Our implementation language encapsulates fields, not objects.

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

- Different modules share `Cell` objects.
- Only `Board` may read or write to the `isMined` field.
- Other modules may contribute other fields to `Cell` objects.

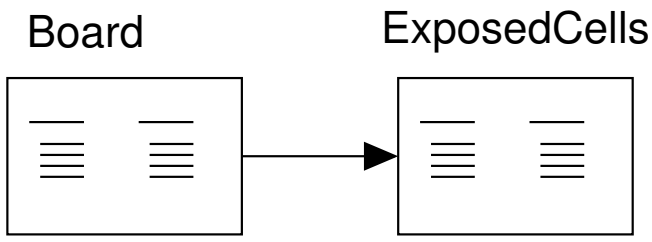
Modular Verification of Design Property



1. Show that the property holds upon exit to `Board`, assuming that it holds upon entry to `Board`.
2. Ensure that only the `Board` module calls `ExposedCells`.
3. Show that the property holds in program's initial state.

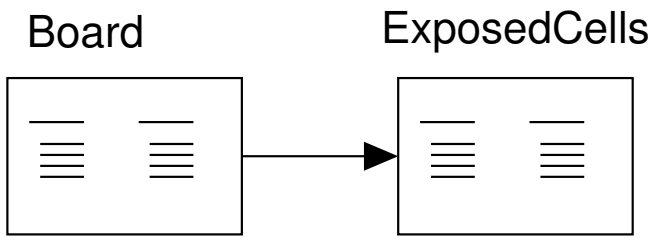
Key precondition : only `Board` and `ExposedCells` modify relevant state.

Stating the Design Property



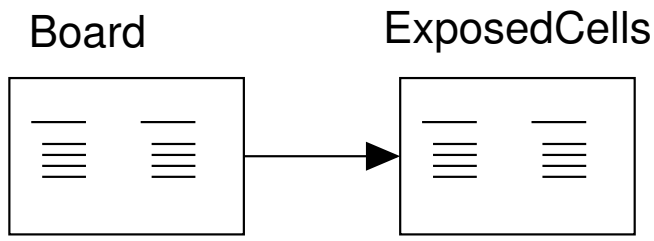

```
scope MinedExposedDisjoint {
  modules Board, ExposedCells;
  exports Board;

  invariant Board.gameOver |
      disjoint(Board.MinedCells,
               ExposedCells.Content);
}
```

Stating the Design Property



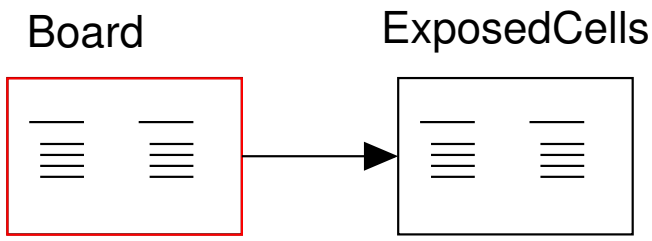

```
scope MinedExposedDisjoint {
  modules Board, ExposedCells;
  exports Board;

  invariant Board.gameOver |
       disjoint(Board.MinedCells,
                ExposedCells.Content);
}
```

Stating the Design Property




```
scope MinedExposedDisjoint {
  modules Board, ExposedCells;
  exports Board;

  invariant Board.gameOver |
       disjoint(Board.MinedCells,
                ExposedCells.Content);
}
```

Stating the Design Property




```
scope MinedExposedDisjoint {
  modules Board, ExposedCells;
  exports Board;

  invariant Board.gameOver |
       disjoint(Board.MinedCells,
                ExposedCells.Content);
}
```

Showing that Properties Hold : a `Board` procedure

```
proc setExposed (c :Cell; v :bool) {
  if (v && c.isMined) setGameOver();
  if (c.isExposed) ExposedCells.remove(c);
  else UnexposedCells.remove(c);

  c.isExposed = v;
  if (v) ExposedCells.add(c); else UnexposedCells.add(c);
}
```

```
proc setExposed(c :Cell; v :bool)
  requires (c in U) & (card(c)=1)
  modifies Exposed, Unexposed, ExposedCells.Content,
    UnexposedCells.Content, Board.gameOver
  ensures v=>(ExposedCells.Content' = ExposedCells.Content + c &
    Unexposed' = Unexposed - c) &
  ((not v)=>(ExposedCells.Content' = ExposedCells.Content - c &
    Unexposed' = Unexposed + c)) & (Unexposed' <= U');
```

`Board` verification relies on interface for `ExposedCells`.

Showing that Properties Hold : a `Board` procedure

```
proc setExposed (c :Cell; v :bool) {
  if (v && c.isMined) setGameOver();
  if (c.isExposed) ExposedCells.remove(c);
  else UnexposedCells.remove(c);

  c.isExposed = v;
  if (v) ExposedCells.add(c); else UnexposedCells.add(c);
}
```

```
proc setExposed(c :Cell; v :bool)
  requires (c in U) & (card(c)=1)
  modifies Exposed, Unexposed, ExposedCells.Content,
    UnexposedCells.Content, Board.gameOver
  ensures v=>(ExposedCells.Content' = ExposedCells.Content + c &
    Unexposed' = Unexposed - c) &
  ((not v)=>(ExposedCells.Content' = ExposedCells.Content - c &
    Unexposed' = Unexposed + c)) & (Unexposed' <= U');
```

`Board` verification relies on interface for `ExposedCells`.

Showing that Properties Hold : a `Board` procedure

```
proc setExposed (c :Cell; v :bool) {
  if (v && c.isMined) setGameOver();
  if (c.isExposed) ExposedCells.remove(c);
  else UnexposedCells.remove(c);

  c.isExposed = v;
  if (v) ExposedCells.add(c); else UnexposedCells.add(c);
}
```

```
proc setExposed(c :Cell; v :bool)
  requires (c in U) & (card(c)=1)
  modifies Exposed, Unexposed, ExposedCells.Content,
    UnexposedCells.Content, Board.gameOver
  ensures v=>(ExposedCells.Content' = ExposedCells.Content + c &
    Unexposed' = Unexposed - c) &
  ((not v)=>(ExposedCells.Content' = ExposedCells.Content - c &
    Unexposed' = Unexposed + c)) & (Unexposed' <= U');
```

`Board` verification relies on interface for `ExposedCells`.

Interfaces for Data Structure Implementations

```
spec module ExposedCells {
  specvar Content : Node set;

  proc add (p :Node)
    requires p not in Content & card(p)=1 &
      (not full)
    modifies Content, full
    ensures Content' = Content + p;

  ...
}
```

Flags Plugin

We designed the flags plugin for :

- coordination modules : such modules use other modules to manage sets.
- typestate-like set definitions : define set contents using integer and boolean field values.

Coordination Modules

Coordinate actions of other modules :

- Maintain references to objects ;
- Pass objects as parameters to other modules ; and
- Get references back as return values.

No encapsulated data structures.

No abstraction functions.

Just interfaces and implementations.

Coordination Module Example

```
p1 = new Object();
p2 = new Object();
p3 = new Object();

S.add(p1);
S.add(p2);
S.add(p3);

x = S.removeFirst();
y = S.removeFirst();
```

Known facts at end :

$$p1 \neq p2 \wedge p1 \neq p3 \wedge p2 \neq p3 \wedge x \neq y \wedge \text{card}(S.\text{Content})=1$$

Flags Analysis

- Set membership may be given by values of primitive fields.

  Example set definition :

  $$\texttt{Mined = \{ x : Cell | x.isMined = true \};}$$

- Also works for integer constants.
- Update sets when relevant fields change, e.g.

  $$\texttt{x.isMined = true} \Rightarrow \texttt{Mined'} = \texttt{Mined} \cup \{ \texttt{x} \}$$

Flags Analysis Formalism

Internal representation :

Formulas in the boolean algebra of sets, e.g :

$$q' \in \text{Content} \wedge \text{Content}' = \text{Content} - q'$$

- unprimed sets represent initial state, primed sets represent current state ;
- formula is a relation between initial state and current state.

Generic Flags Analysis Transfer Function

Propagate facts using strongest postconditions ; use relation composition to update the relation summarizing current state.

$$post_{st}(F) = F \circ m(st)$$

$m(st)$: the effect of statement st on sets ;

- for procedure calls, use contracts ;
- for assignments, generate set insertions and removals

Use relation composition to compose effects :

$$F_1(\vec{S}, \vec{S'}) \circ F_2(\vec{S}, \vec{S'}) = \exists \vec{T}.\ F_1(\vec{S}, \vec{T}) \wedge F_2(\vec{T}, \vec{S'})$$

Flags Analysis Transfer Function Example

When a module's abstraction section contains,

$$R = \{x : T \mid x.\mathtt{f} = c\}$$

we use the following transfer function :

$$\mathcal{F}(\mathtt{x}.\mathtt{f} = \mathtt{c}) := R' = R \cup \mathtt{x} \wedge \bigwedge_{S \in \text{alts}(R)} S' = S \setminus \mathtt{x} \wedge \text{frame}_{\{R\} \cup \text{alts}(R)}$$

where

$$\begin{aligned} \text{alts}(\mathtt{R}) &= \{S \mid \text{ abstraction module contains} \\ &\qquad S = \{x : T \mid x.\mathtt{f} = c_1\}, c_1 \neq c.\} \\ \text{frame}_F &= \bigwedge_{S \neq F} S' = S \end{aligned}$$

Flags Analysis Mechanism

Bottom line : do procedures implement their interfaces ?

For each procedure, the flags analysis :

- computes a relation summarizing the program state at the end of the procedure ;
- verifies if this relation implies that the procedure postcondition holds upon procedure exit.

Analysis also checks preconditions at procedure callsites.

MONA as Flags Plugin Decision Procedure

Use MONA decision procedure for monadic second-order logic.

- Handles formulas over sets and trees.
- Flags analysis doesn't use MONA's full generality.
- Much faster for sets (used in flags plugin) than for trees.

Naive approach does not scale—formulas grow exponentially in the presence of conditionals. Apply optimizations.

Loops

Consider the following code :

```
proc clear() {
  bool e = isEmpty();
  while (!e) {
    Entry q = removeFirst();
    e = isEmpty();
  }
  return;
}
```

Notice : potentially unbounded number of executions.

Ways to analyze loops

General idea : summarize loop body with a loop invariant.

Two choices :

1. Use developer-provided (explicit) invariants ; or
2. Automatically infer loop invariants.

Verifying explicitly-provided invariants

```
proc clear() {
  bool e = isEmpty();
  while "e' <=> card(Content') = 0" (!e) {
    Entry q = removeFirst();
    e = isEmpty();
  }
  return;
}
```

- Loop invariant must hold before entering loop ;
- Loop body must preserve the loop invariant ; and
- Loop invariant can be used as program state upon exit from loop.

Automatically Inferring Loop Invariants

- Propagate candidate loop invariants ; weaken until we reach a valid invariant.

Loop Invariant Inference Results

Inferred loop invariants for our benchmarks.

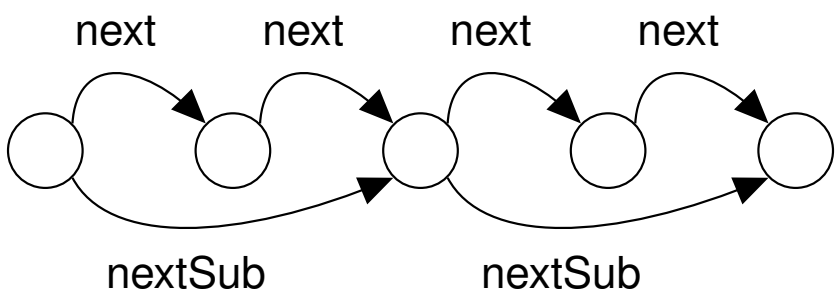
Compared inferred and explicitly-provided invariants :

- Inferred invariant always implied explicit invariant ;
- Never vice-versa.

Found 15 nontrivial loop invariants in benchmarks.
Our approach worked especially well due to set specification language.

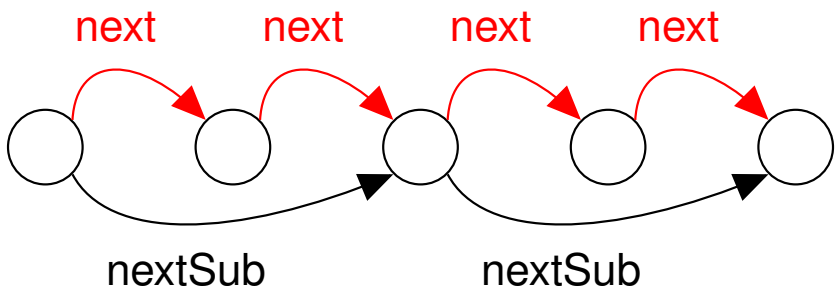
Shape Analysis in Hob

Analysis target : two-level skip list.



Shape Analysis in Hob

Analysis target : two-level skip list.

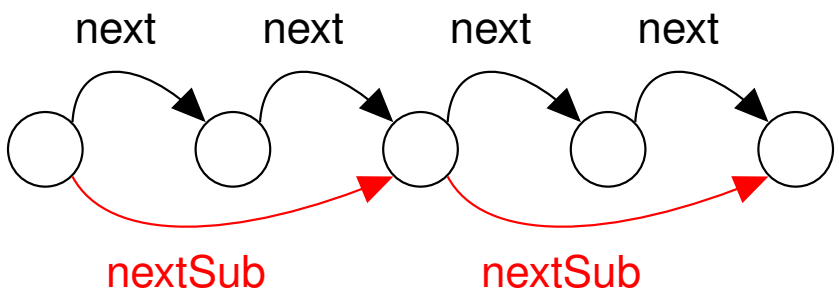


`next` fields form tree-structured data structure backbone :

```
Content = { n : Node |
  "rtrancl (lambda v1 v2. next v1 = v2) (next root) n" };
```

Shape Analysis in Hob

Analysis target : two-level skip list.



`nextSub` fields controlled by nondeterministic field constraints :

```
invariant "ALL x y.
   prev x = y --> (x  = null &
       (EX z. next z = x) --> next y = x) &
   ((x = null | (ALL z. next z  = x)) --> y = null)";
```

Bohne Example

```
abst module List {
  use plugin "Bohne decaf";
  Content = { n : Node | "rtrancl
      (lambda v1 v2. next v1 = v2) (next root) n" };

  invariant "ALL x y.
       prev x = y --> (x ~= null &
       (EX z. next z = x) --> next y = x) &
       ((x = null | (ALL z. next z ~= x))
          --> y = null)";

  invariant "init --> (ALL x. ~(next x = root))";
  // Other invariants omitted.
}
```

Evaluating the Bohne Plugin

- Bohne shape analysis plugin developed by Thomas Wies.
- Uses monadic second-order logic (and MONA decision procedure) to reason about reachability.
- Predicate abstraction for inferring loop invariants.
- Verified a number of data structures, including linked lists, skip lists, and tree insertion.
- Hob enables Bohne verification results to be used for ensuring software design properties.

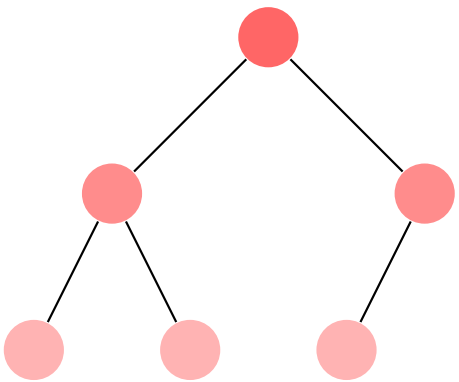
Theorem Proving Plugin

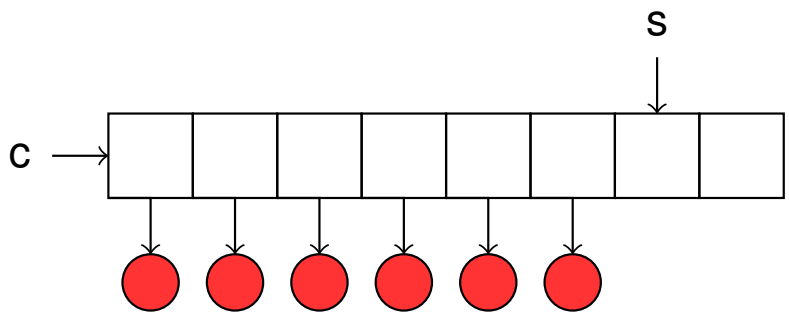
Some program properties rely on arbitrarily complicated reasoning.

Verifying an Array-Based Data Structure

Example : Complete binary tree up to last row.
We implemented the tree using an array and verified its operations.

- parent(i) $= i/2$
- left(i) $= 2i$
- right(i) $= 2i + 1$





Theorem Proving Set Definitions

Theorem proving plugin accepts arbitrary Isabelle formulas as set definitions :

$$\textbf{InQueue} = \{x : \textbf{Node} \mid \textbf{exists}\ j.\ 1 <= j\ \&\ j <= s\ \&\ x = c[j]\}$$

The plugin generates proof obligations from implementations and set definitions.

Evaluating the Theorem Proving Plugin

We used the theorem proving plugin to verify tree insertion.

Plugin generates 11 sequents, of which :

- Isabelle discharges 5 sequents automatically.
- We proved 6 manually ;
  - Shortest proof : 1 line (introducing an arithmetic lemma).
  - Longest proof : 38 lines
  - Average proof length : 14.2 lines

Custom Formula Transformations

- Peephole optimizations :

$$A \wedge B \wedge A \wedge \text{true} \rightarrow A \wedge B$$

- Substitution :

$$\exists x.x = E \wedge F(x) \rightarrow F(E)$$

- Convert implications to negated disjunctions :

$$f \Rightarrow g \rightarrow \neg(f \vee \neg g)$$

  (Amplifies the effect of substitution)

- Factoring :

$$(A \wedge B) \vee (A \wedge C) \rightarrow A \wedge (B \vee C)$$

  (Reduces exponential explosion for path sensitive analysis)

- Quantifier scope minimization :

$$(\exists x.\ F(x,y) \wedge G(y)) \rightarrow (\exists x.\ F(x,y)) \wedge G(y)$$

Impact of Formula Transformations

We recorded the maximum size (in nodes) of formulas with and without optimizations.

	# nodes (opt)	# nodes (unopt)	ratio
prodcons	466	4487	9.64
compiler	5749	$>$ 451628	$>$ 78.56
scheduler	296	1169	3.95
board	11778	$>$ 945887	$>$ 80.31
controller	28904	4528	6.38
view	19311	N/A	N/A
atom	28317	584384	20.65
ensemble	668110	N/A	N/A
h2o	79249	$>$ 1257883	$>$ 15.87
sendfile	2672	119287	44.64
httpserver	1094	68198	62.34
httprequest	9521	61041	6.41

Importance of Formula Transformations

Reduce formula sizes—empirically, growth no longer seems exponential.

- Larger formulas, larger formula size reduction ratio.

Without simplifications :

- Many formulas cannot be stored in memory.

With simplifications :

- Producing and verifying formulas no longer a bottleneck.
- All of our examples go through (up to 96-line procedures).

Challenges to Program Verification

1. Scalability
2. Diversity

Challenge 1 : Scalability

Want to apply detailed (and therefore expensive) static analyses to sizeable programs.

Existing shape analyses handle programs with at most hundreds of lines.

Challenge 2 : Diversity

Hard to imagine any single analysis successfully verifying all data structures.

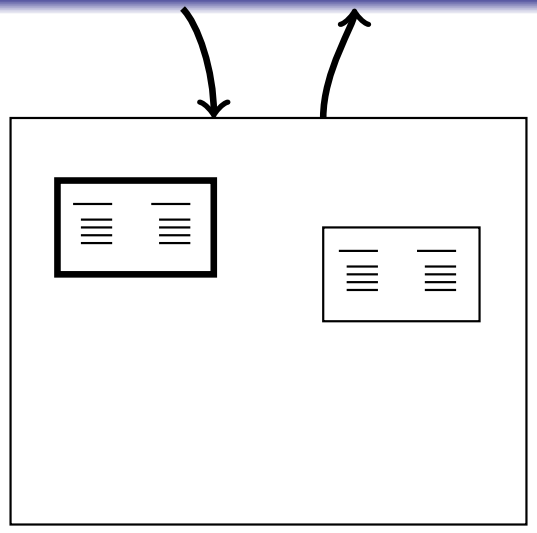
Need to somehow combine results from different analyses.

Key idea : Modular Verification

Assume/guarantee reasoning enables modular verification by overcoming the scalability and diversity challenges.

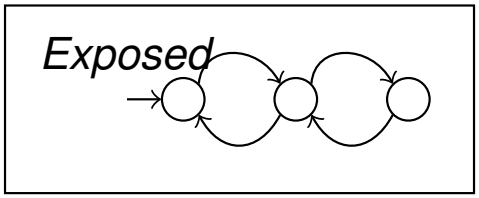
Hob uses assume/guarantee reasoning at two levels :

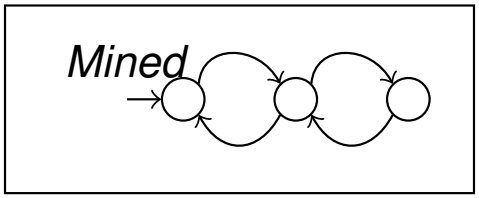
- to verify properties local to each module ; and
- to verify inter-module invariants.



How do we express this property ?

A possible minesweeper implementation :

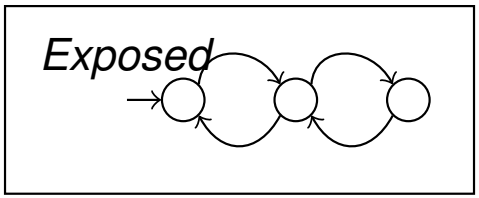


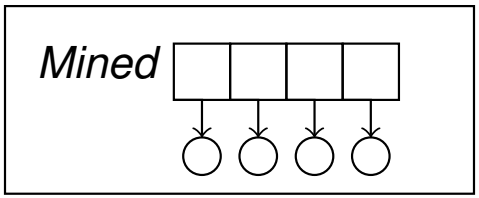


Want to state that $\mathit{Exposed}.\texttt{next}^* \cap \mathit{Mined}.\texttt{next}^* = \emptyset$.

But design information should not need contain low-level implementation details like heap reachability properties.

How do we express this property ?



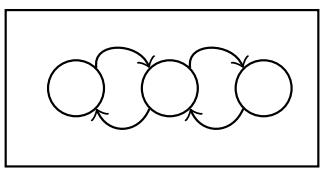


Furthermore, what if the mined cells data structure is array-based ?

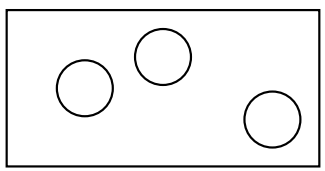
Our Proposal : Set-Based Specifications

Hob supports formulas in the boolean algebra of sets for stating design properties.

- Developers express design information using abstract sets of heap objects and relationships between these sets.



concrete list



abstract set

Verifiability of Hob's Set Specifications

1. Set language doesn't contain numbers.
2. Cannot express relations between objects, e.g. a mapping between keys and values.
3. Set language can't express ordering between elements.

Module-Structured Implementation Language

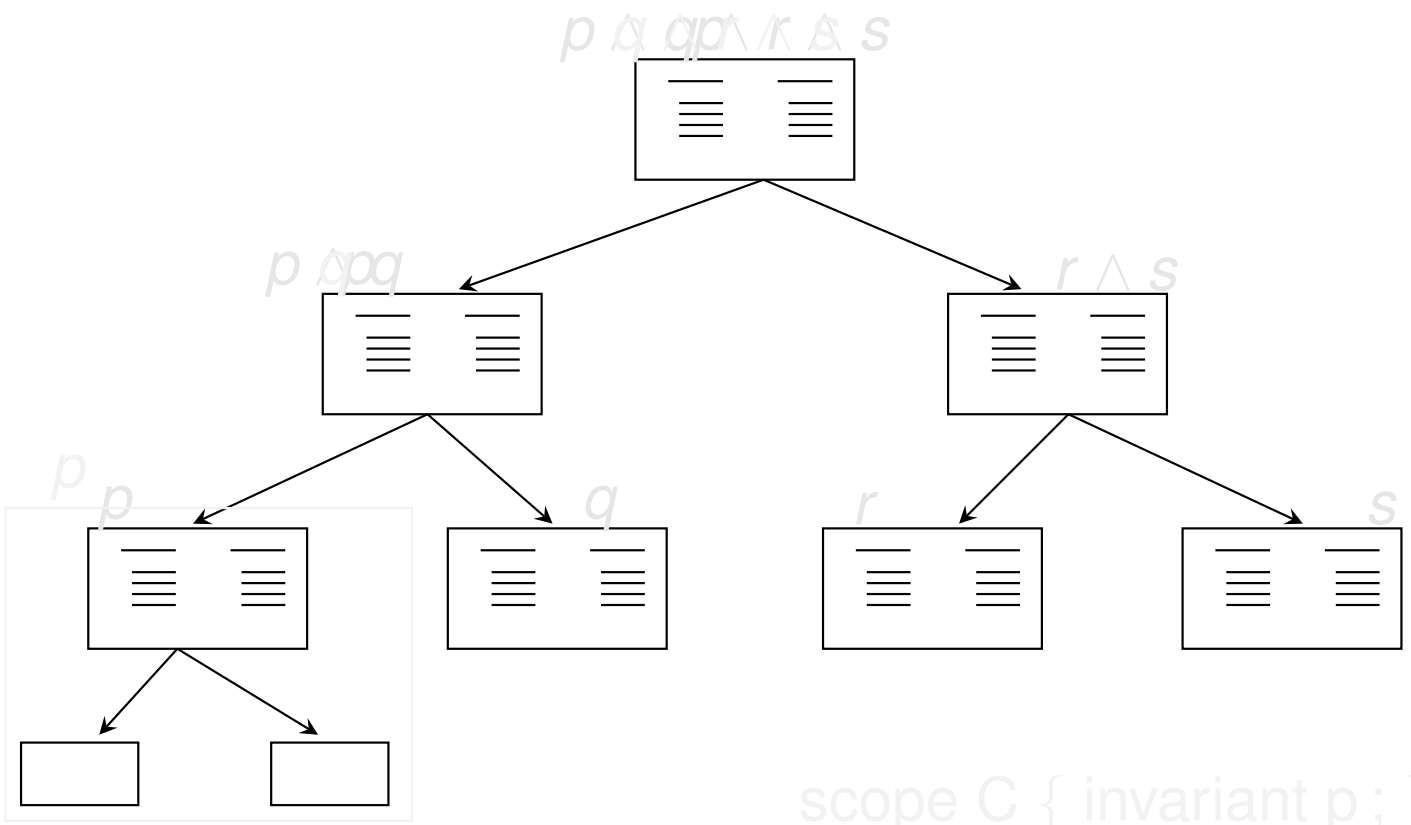
- Standard imperative language.
- Finite number of instantiable modules.

Each module contains :

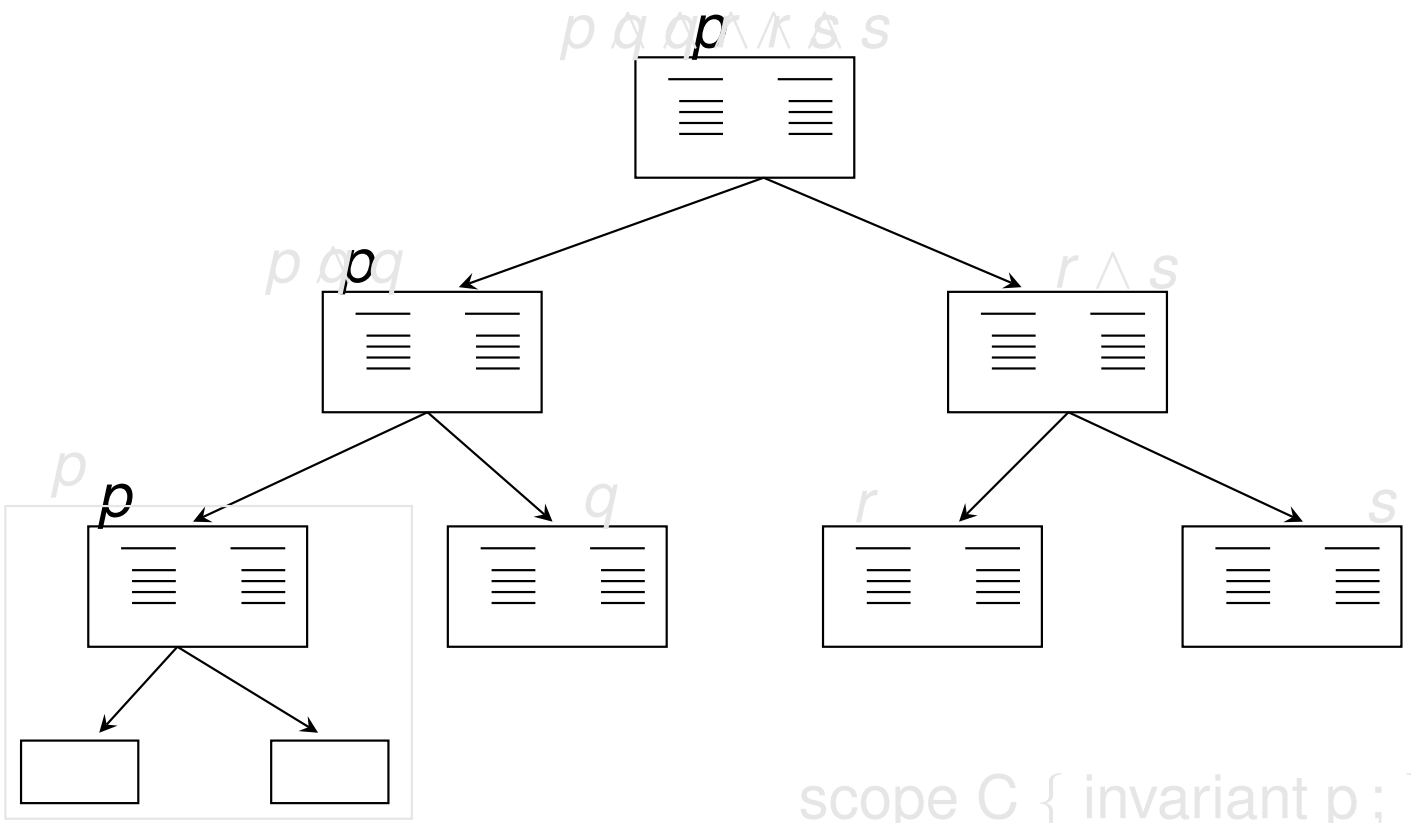
1. implementations—procedures, data structure roots, encapsulated fields ;
2. specifications—interfaces, set declarations, boolean variables ;
3. abstractions—link implementations and specifications.

```
impl module M {  (1)
  reference r : T;
  proc p(n1 :T;n2 :T) {
    // code
  }
}

spec module M {  (2)
  var S : T set;
  proc p(n1 :T;n2 :T)
    requires B ensures
B';
}

abst module M {  (3)
  S = { x : T | " ... "
};
}
```

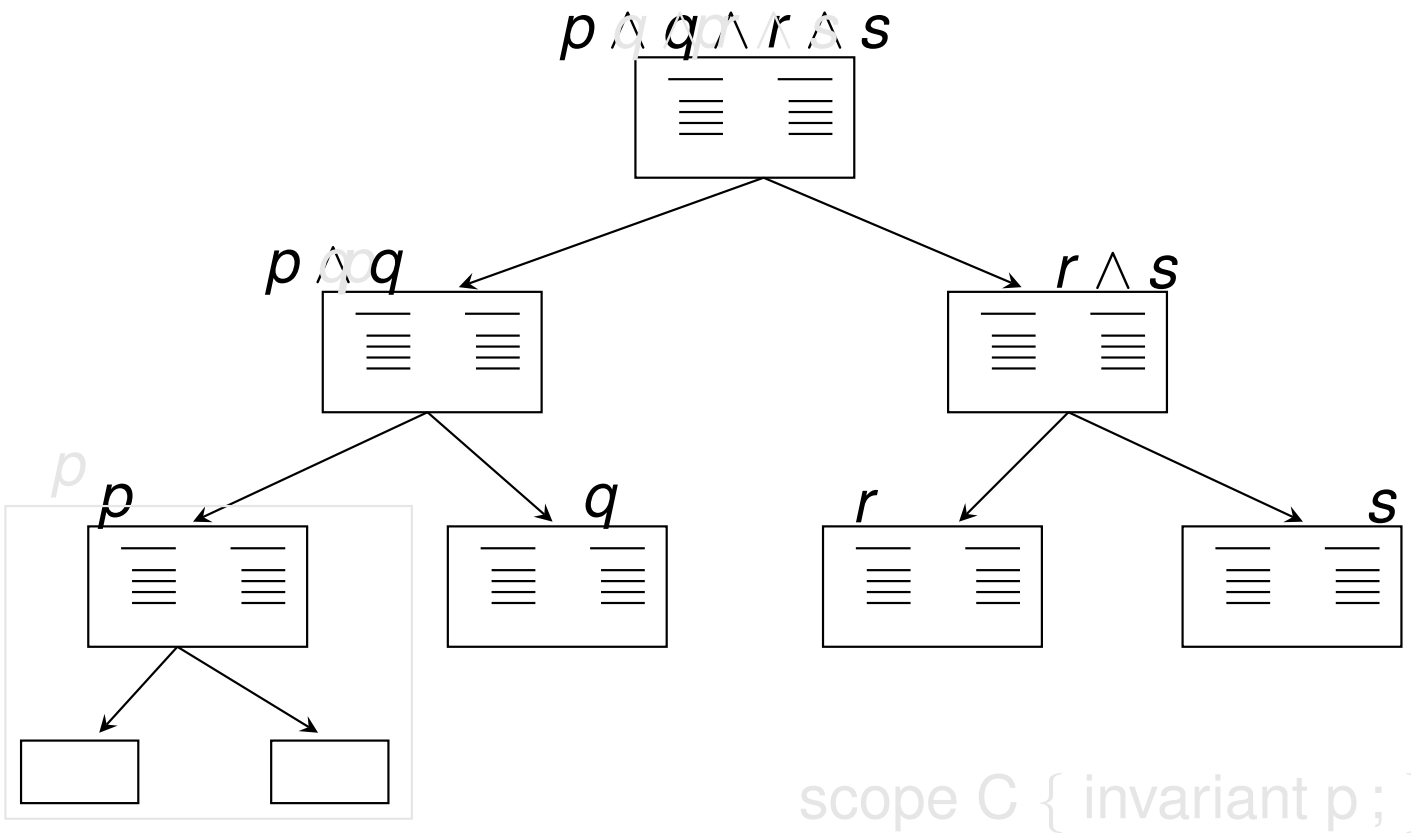
Specification Aggregation



Specification Aggregation

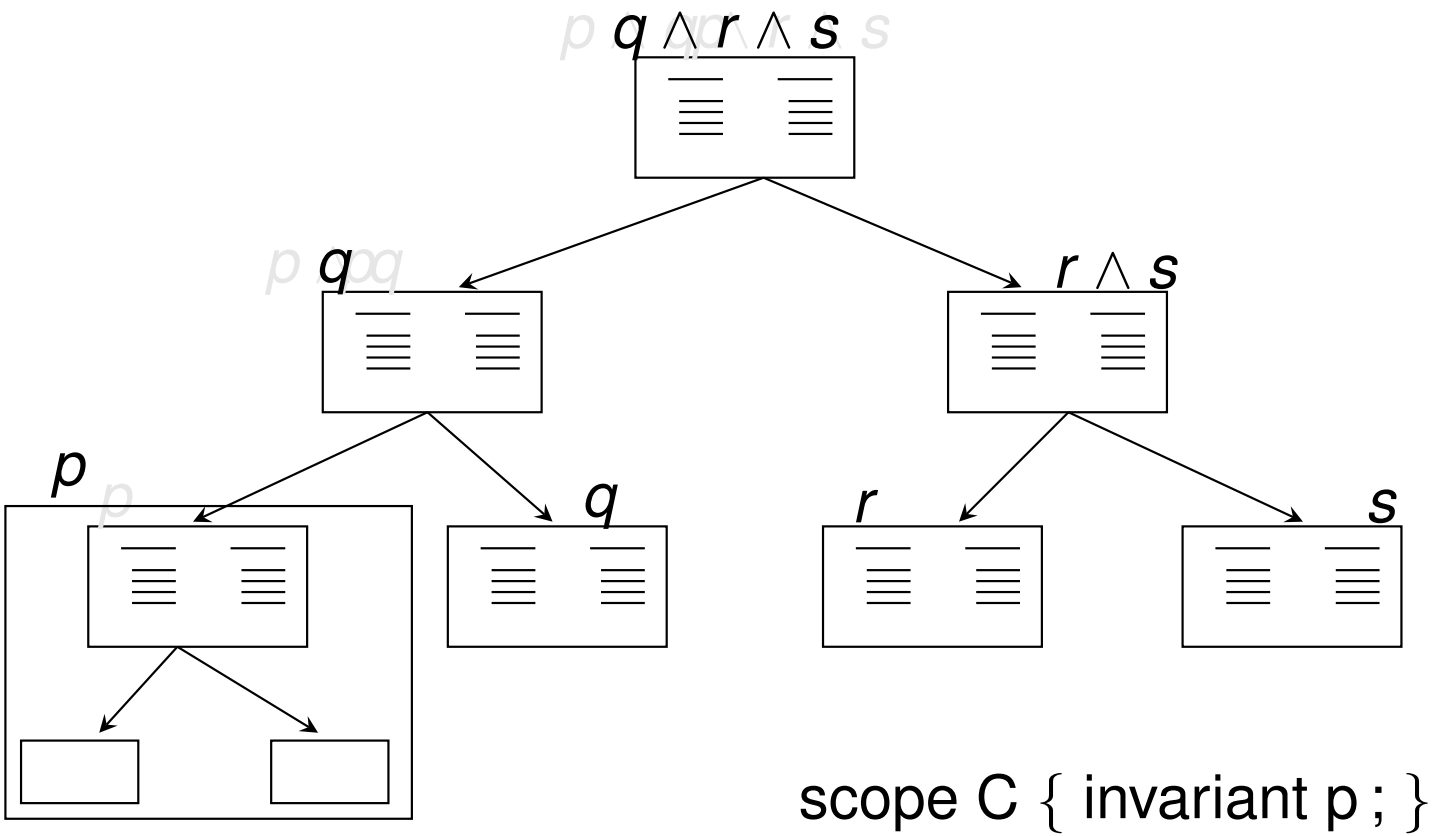


Specification Aggregation



scope C { invariant p ; }

Specification Aggregation



scope C { invariant p ; }

Scope Invariants

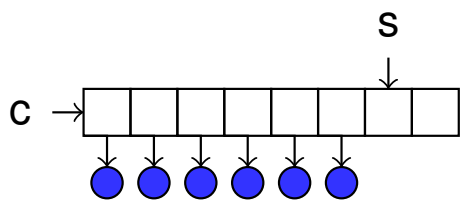
Developers must declare :

1. where the invariant may temporarily be violated ;
2. legitimate access points to the scope ; and
3. the invariant itself.

```
scope MinedExposedDisjoint {
  modules Board, ExposedCells; (1)
  exports Board; (2)

  invariant Board.gameOver |
     disjoint(Board.MinedCells,
          ExposedCells.Content); (3)
}
```

Defaults

Consider an array-based data structure.



Must allocate the array before calling data structure operations.

```
specvar ready : bool;
proc init() modifies ready ensures ready';
proc add(p) requires ready ...;
```

explicit initialization constraint

Explicit Initialization Constraints

```
spec module ArrayContainer {
  proc init() modifies ready ensures ready';
  proc add(p) requires ready & ...;
      ensures ...;

  proc remove(p) requires ready & ...;
      ensures ...;

  proc isEmpty(p) requires ready & ...;
      ensures ...;
}
```

Applying Defaults

Defaults must be true throughout their range unless suspended.

No defaults :

```
proc init()
   modifies ready
   ensures ready' ;

proc add(p)
   requires ready & p not in S
   ensures ... ;

proc del(p)
   requires ready & p in S
   ensures ... ;
```

Defaults :

```
default D : ready ;

proc init()
   suspends D modifies ready
   ensures ready' ;

proc add(p)
   requires      p not in S
   ensures ... ;

proc del(p)
   requires      p in S
   ensures ... ;
```

Verifying Scopes and Defaults

Scope invariants :

- conjoin to preconditions of exported modules ;
- conjoin to postconditions of exported modules ;
- verify in program's initial state.

Defaults : unless suspended,

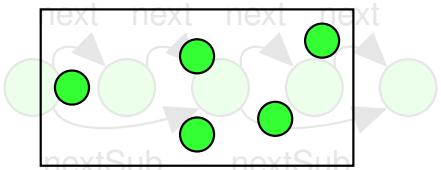
- conjoin to preconditions of specified modules ;
- conjoin to postconditions of specified modules ;

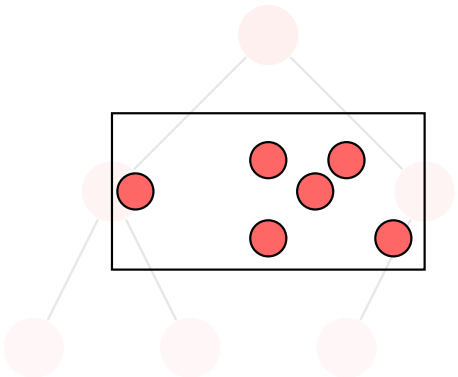
Ultimate verification responsibility lies at procedure level.

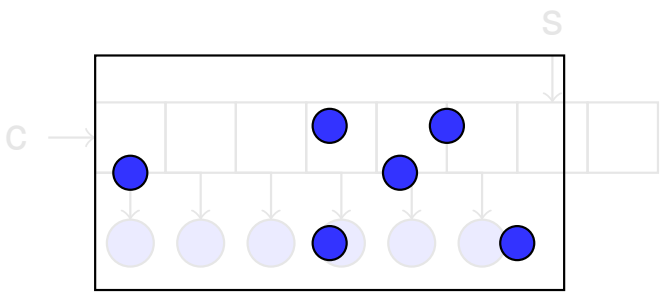
Another Scope Example

```
scope C {
  modules Scheduler, RunningList, IdleArray;
  exports Scheduler;

  invariant RunningList.Content
        sub Scheduler.Processes;
}
```

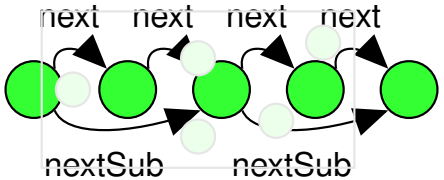
Grounding Out Set Definitions

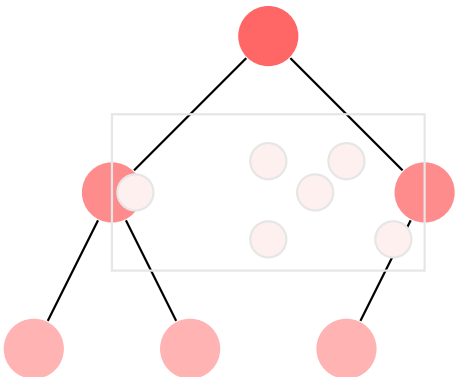


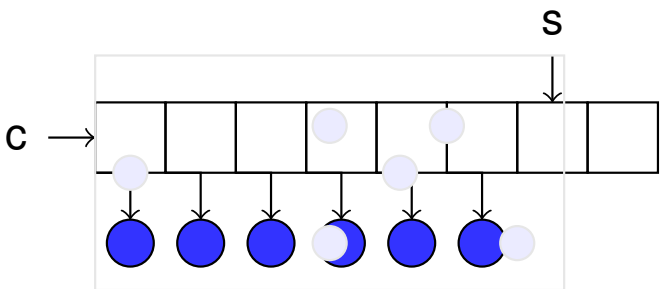




Grounding Out Set Definitions







Verifying Preconditions and Postconditions

Each module maintains a collection of abstract sets.
Modules are analyzed in isolation, each by one or more *analysis plugins*.
An analysis plugin verifies that procedures :

- preserve invariants ; and
- implement their interfaces.

Le démonstrateur de théorèmes Isabelle

Les scripts des sessions de démonstration permettent la réutilisation des modules.

Tout de même, l'utilisation du démonstrateur de théorèmes prend beaucoup d'effort.

Analysis Plugins

Outsource verification task to analysis plugins.

- Each analysis plugin handles a particular class of implementations.
- Different plugins specialized towards different heap abstractions.

Hob supports arbitrary classes of implementations given suitable analysis plugins.

Comparing Spec# and Hob

	Spec#	Hob
Specification language	C#	Set-based
Implementation language	Object-oriented	Procedural
Analysis approach	Simplify & co.	Plugin-based

Structures de données vérifiés

- Implémentations des ensembles, piles, files, et files à priorité, avec des listes (simplement chaînée, doublement chaînée, avec et sans itérateur)
- Implémentations des ensembles avec skip-lists à deux niveaux et par tableau.
- Insertion d'un élement dans un arbre binaire.