Projet GECCOO - Modularité et Raffinement

Marie-Laure Potet

LSR-IMAG, Grenoble, France

Marie-Laure.Potet@imag.fr

ProVal INRIA - LIFC -Projet Cassis - EVEREST - VASCO LSR

Les enjeux

Maîtriser la complexité de l'activité de spécification, de programmation et de validation

- composition horizontale : décomposer ou recomposer
  - définir des composants cohérents ayant des interfaces adéquates
- composition verticale : raffiner ou abstraire
  - introduire différents niveaux de détail (comportement, représentation des données)

Vérification : conditions de préservation des propriétés par recomposition et par introduction de détails.

Les difficultés : la notion d'état n'est pas modulaire, les propriétés ne sont pas toujours conservées par raffinement (vivacité, sécurité ...) ...

$\Rightarrow$ Compromis entre :

- Expressivité
- Composition des preuves
- Complexité de mise en oeuvre pour les développeurs

Les différents travaux (1)

Invariants dans les langages à composants :

- extension pour la méthode B
- invariant et redéfinition pour Krakatoa
- invariants de structures de données (Caduceus)

Preuve et spécification par abstraction / raffinement :

- introduction de types de données abstraits en JML (Jack, Krakatoa) (FTfJT 2006)
  - lien structure concrète / structure abstraite
- raffinement d'ordre supérieur (Sylvain Boulmé)
  - spécifications dépendant de l'état

Exemple

classes Java / spécifications algébriques

$\Rightarrow$ décrire des structures abstraites au lieu de les représenter, faciliter la spécification et les preuves.

- ```
  class IntSet {
     int size ; int t[] ;
     public boolean mem (int n) { ... }
  ...  }
  ```

- ```
     type inset ;
     intset emptyset() ; logic intset singleton (int n);
     intset union(intset s1, s2) ; in (int n, intset s);
  ```

- ```
     predicate Intset_models(intset as, IntSet cs) = ...
  ```
  ```

Les différents travaux (2)

Prise en compte des propriétés de sécurité dans le processus de raffinement

- condition de préservation de propriétés temporelles par raffinement (LIFC), application à l'étude de cas demoney
  - raffinement dans JAG basé sur le raffinement en B événementiel
- raffinement de politiques de contrôle d'accès
  - application aux politiques réseaux (AFADL'06, B'07)

Politique de sécurité réseau

$\Rightarrow$ en lien avec l'ACI Potestat (test de politique de sécurité)

- raffinement de données suivant les couches TCP/IP (utilisateur, service, machine, port, ...)
- relation de conformité à une politique abstraite :
  - accepté au niveau i $\Rightarrow$ accepté au niveau i-1
  - refusé au niveau i $\Rightarrow$ refusé au niveau i-1
  - en conflit au niveau i $\Rightarrow$ en conflit ou accepté au niveau i-1
- application à la vérification et à la génération de tests de conformité

Invariants de module (1)

$\Rightarrow$ Appréhender la notion d'invariant localement à un composant est simple :

- établi par l'état initial
- préservé par le jeu d'opérations.

$\Rightarrow$ Etendre la notion d'invariant à l'extérieur d'un composant implique un certain nombre de choix (MPHL04) :

Invariants de module (2)

1. un principe d'encapsulation qui décrit quelles parties d'un programme peut modifier les variables mises en jeu par un invariant

2. la définition de la notion *d'invariants admissibles* qui décrit quelles variables peuvent être contraintes par l'invariant d'un composant donné.

3. une sémantique de la notion d'invariant qui décrit en quel point d'un programme les invariants doivent être vérifiés (invariant fort/invariant faible).

Invariants de module (3)

- Module à la Hoare. Invariant admissible : sur les variables déclarées dans le module. Principe d'encapsulation fort. Sémantique : invariant fort.
- B. Possibilité de contraindre les variables importées, avec des restrictions fortes sur les architectures admises.
- JML (V1). Peu de contrainte sur les invariants, pas de principe d'encapsulation imposé, sémantique théoriquement complexe (simplifiée pour les outils).
- Spec#. Approche 1 sauf dans des portions de programme bien déterminées + possession dynamique des composants.

The SPEC# *approach*

SPEC# proposes a flexible methodology for *modular verification* of objects invariants. Main characteristics:

- authorizes invariants violation outside the scope of objects definition, but in a controlled way (ownership relation).
- each object has a ghost field, indicating *dynamically* the status of its invariant.
- safe use of objects, with respect to their invariant status, is *monitored* by proofs.

Advantages: compatible with aliasing, late-binding, subtyping.

Drawbacks: annotations hard to manage, a great number of proof obligations (but many are trivial).

Ghost status variable

Each component N is extended by a ghost variable $\texttt{st}$ such that:

$$\texttt{st} \in \{\texttt{invalid}, \texttt{valid}, \texttt{committed}\}$$

.

- if $N.\texttt{st} = \texttt{invalid}$, then its invariant may be violated. In particular, any modification on N variables is authorized.
- if $N.\texttt{st} = \texttt{valid}$, then its invariant is established, and it has no **dynamic owner** M (i.e such $(M, N) \in owns \wedge M.\texttt{st} \neq \texttt{invalid}$).
- if $N.\texttt{st} = \texttt{committed}$, then its invariant is established, and it has a single dynamic owner M.

Formally expressed by *meta-invariant* $\mathcal{I}$ which is the conjunction of:

- $\mathcal{I}_1$: $M.\texttt{st} \in \{\texttt{valid}, \texttt{committed}\} \Rightarrow M.\text{Inv}$
- $\mathcal{I}_2$: $M.\texttt{st} \in \{\texttt{valid}, \texttt{committed}\} \wedge (M, N) \in owns \Rightarrow N.\texttt{st} = \texttt{committed}$
- $\mathcal{I}_3$: $N.\texttt{st} = \texttt{committed} \wedge\ (A, N) \in owns \wedge (B, N) \in owns \wedge\ A.\texttt{st} \neq \texttt{invalid} \wedge B.\texttt{st} \neq \texttt{invalid} \Rightarrow A = B$

`pack` *and* `unpack` *substitutions*

Each affectation $M.x := e$ has precondition $M.st = \texttt{invalid}$.

Substitutions are extended with two commands: $\texttt{pack}(M)$ and $\texttt{unpack}(M)$. Only these commands modify directly `st` variables.

- $\texttt{pack}(M)$ requires the establishment of M invariant.
- $\texttt{unpack}(M)$ allows violation of M invariant.

subst	**trm**	$\mathbf{prd_{st}}$
$\texttt{pack}(M)$	$\forall N \,.\, ((M,N) \in owns \Rightarrow N.\texttt{st} = \texttt{valid})$ $\wedge\, M.\texttt{st} = \texttt{invalid}$ $\wedge\, M.\text{Inv}$	$\forall N \,.\, ((M,N) \in owns \Rightarrow N.\texttt{st}' = \texttt{committed}$ $\wedge\, M.\texttt{st}' = \texttt{valid}$
$\texttt{unpack}(M)$	$M.\texttt{st} = \texttt{valid}$	$\forall N \,.\, ((M,N) \in owns \Rightarrow N.\texttt{st}' = \texttt{valid})$ $\wedge\, M.\texttt{st}' = \texttt{invalid}$

Results

- interpreting B in this meta-model with interesting extensions (B'07)
  - systematic annotations of B operations
  - equivalence between termination proof obligations of annoted substitutions and classical B proof obligations
- exploiting the static aspect of B to propose a static typing algorithm (less general but expressive enough) (submitted)
  - pattern operations, verification and preconditions inference
- To be done : compatibility with refinement

Examples of B initializations

- Invariant establishment by initialization of stand-alone M:

PRE $M.\mathtt{st} = \mathtt{invalid}$
THEN $M.Init$; $\mathtt{pack}(M)$
END

- Initialization performing another initialization (M INCLUDES N):

PRE $M.\mathtt{st} = \mathtt{invalid} \wedge N.\mathtt{st} = \mathtt{invalid}$
THEN $N.Init$; $\mathtt{pack}(N)$; $M.Init$; $\mathtt{pack}(M)$
END

- Initialization depending on another initialization (M SEES N):

PRE $M.\mathtt{st} = \mathtt{invalid} \wedge (N.\mathtt{st} = \mathtt{valid} \vee N.\mathtt{st} = \mathtt{committed})$
THEN $M.Init$; $\mathtt{pack}(M)$
END

$\Rightarrow$ we recover B proof obligations.

Interpreting B operations

B operation PRE P THEN S END defined in M:

- for a read-only operation on variables of $owns^{+}[\{M\}]$:

> PRE $P \wedge (M.\texttt{st} = \texttt{valid} \vee M.\texttt{st} = \texttt{committed})$
> THEN S
> END

- otherwise:

> PRE $P \wedge M.\texttt{st} = \texttt{valid}$
> THEN $\texttt{unpack}(M)\ ;\ S\ ;\ \texttt{pack}(M)$
> END

$\Rightarrow$ we recover B proof obligations.

A case study

Embedding a byte-code interpreter in smart card (BOM Project – FME'03). A natural 3 components architecture:

- a cap file: a specific byte-code exchange format between the card and the development platform
- a Javacard converter which produces a well-formed cap file (off-card operation)
- an embedded interpreter which executes the cap file (on-card operation)

MACHINE CAP VARIABLES cap_file INVARIANT $ICAP$ OPERATIONS ... END	MACHINE $Converter$ INCLUDES CAP VARIABLES $convert\_state$ INVARIANT $ICONV$ OPERATIONS ... END	MACHINE $Interpreter$ SEES CAP VARIABLES $interp\_state$ INVARIANT $IINTP$ OPERATIONS ... END

Not authorized in B

Example of byte-code interpreter

MACHINE $Interpreter$

INCLUDES CAP

VARIABLES ip, ...

INVARIANT $IINTP$

INITIALIZATION $start$ = PRE $CAP.\texttt{st} = \texttt{valid} \wedge P$ THEN $ip := 1$ END

OPERATIONS . . .

END

where $\mathcal{I} \wedge CAP.\texttt{st} = \texttt{valid} \wedge P \Rightarrow [ip := 1]IINTP$

transfer of CAP ownership

Main: $Converter.start; ...; \boxed{\texttt{unpack}(Converter); Interpreter.start}\ ; ...$

Perspectives

Proposer des patterns permettant de maîtriser la pose d'invariant (extension B, structure de données en Caduceus, classe et redéfinition en Krakatoa).

- facilité de mise en oeuvre pour les développeurs
- maîtrise de l'activité de preuve et de raffinement

Propriétés de sécurité :

- méthodologie Critères Communs : exigences et traçabilité à travers un développement
- classes de propriétés et conditions de préservation par raffinement