Formal Development Of Complex Systems

Binary relations

Axiomatisation

$S \leftarrow\!\!\!\!\twoheadrightarrow T$	If $r \in S \leftarrow\!\!\!\!\twoheadrightarrow T$ then $ran(r) = T$
$S \twoheadleftarrow\!\!\!\!\rightarrow T$	If $r \in S \twoheadleftarrow\!\!\!\!\rightarrow T$ then $dom(r) = S$
$S \twoheadleftarrow\!\!\!\!\twoheadrightarrow T$	If $r \in S \leftarrow\!\!\!\!\twoheadrightarrow T$ then $dom(r) = S \ \wedge \ ran(r) = T$

Manipulation of binary relations

- Restriction and Subtraction

Domain Restriction	$S \lhd r$
Range Restriction	$r \rhd T$
Domain Subtraction	$S ⩤ r$
Range Subtraction	$r ⩥ T$

Axiomatisation

$S \lhd r$	$S \lhd r = \{x \mapsto y \mid x \mapsto y \in r \ \wedge \ x \in S\}$
$r \rhd T$	$r \rhd T = \{x \mapsto y \mid x \mapsto y \in r \ \wedge \ y \in T\}$
$S ⩤ r$	$S ⩤ r = \{x \mapsto y \mid x \mapsto y \in r \wedge x \notin S\}$
$r ⩥ T$	$r ⩥ T = \{x \mapsto y \mid x \mapsto y \in r \wedge x \notin T\}$

Functions

Functions

Partial Function	$S ⇸ T$
Total Function	$S \rightarrow T$

Axiomatisation. A Function is a Relation

$f \in S ⇸ T$	$\triangleq$	$f \in S \leftrightarrow T \wedge f^{-1}; f = id(ran(f))$
		$f \in S \leftrightarrow T \wedge f^{-1}; f \subseteq T \lhd id$
$f \in S \rightarrow T$	$\triangleq$	$f \in S ⇸ T \wedge S = dom(f)$

Other Function definitions

Partial Injection	$S ⤔ T$
Total Injection	$S \rightarrowtail T$
Partial Surjection	$S ⤀ T$
Total Surjection	$S \twoheadrightarrow T$
Bijection	$S ⤖ T$

Axiomatisation

$S ⤔ T$	$S ⤔ T = \{f \cdot f \in S ⇸ T \wedge f^{-1} \in T ⇸ S\}$
$S \rightarrowtail T$	$S \rightarrowtail T = S ⤔ T \cap S \rightarrow T$
$S ⤀ T$	$S ⤀ T = \{f \cdot f \in S ⇸ T \wedge ran(f) = T\}$
$S \twoheadrightarrow T$	$S \twoheadrightarrow T = S ⤀ T \cap S \rightarrow T$
$S ⤖ T$	$S ⤖ T = S \rightarrowtail T \cap S \twoheadrightarrow T$

Modeling of systems

Before-After predicates as a relation on states

- Event *ev* defines a relation on states.
  - **BAA**$(\mathbf{x}, \mathbf{x}')$ is a Before-After Predicate characterising the event *ev*. Example.

    If **ev** is $\mathbf{x} := \mathbf{x} + \mathbf{1}$ then **BAA**$(\mathbf{x}, \mathbf{x}')$ is $\mathbf{x}' = \mathbf{x} + \mathbf{1}$ or

$$BAA(x, x') \equiv x' = x + 1$$

This definition is the assignment definition of Hoare Logic
$\{[x/E]\Psi\}x := E\{\Psi\}$

Event B

Machines and contexts

Machine Defines the system model as a state-transitions (state variables and events)

- `REFINES` an other machine
- `SEES` a context
- `VARIABLES` of the model
- `INVARIANTS` satisfied by the variables (state)
- `THEOREMS` satisfied by the variables (state) and deduced from invariants and seen contexts
- `VARIANT` decreasing
- `EVENTS` modifying state variables

Context It contains the definitions of the domain concepts needed to model the system. It also defines the proof context.

- `EXTENDS` an other context
- `SETS` declares news sets
- `CONSTANTS` défines a list of constants
- `AXIOMS` defining properties of sets and constants
- `THEOREMS` a list of theorems deduced from axioms

Event B Contexts

CONTEXT
  ctx
EXTENDS
  actx
SETS
  s
CONSTANTS
  c
AXIOMS
  ax_i : ....
THEOREMS
  Tc_i : ....
END

Context

- `ac` extends the context `c` and adds new concepts
- `s` sets defined by comprehension or intention
- `k` definition of constants
- `ax1` axioms defining sets and constants
- `T(x)` set of theorems deduced from axioms and theorems.

Event B Machines

```
MACHINE
   m
REFINES
   am
SEES
   ctx
VARIABLES
   x
INVARIANTS
   I(x)
THEOREMS
   T(x)
VARIANT
   v
EVENTS
   ev1 = ...
   ev2 = ...
   ...
END
```

Machine

- `m` abstract machine corresponding to the system model
- `am` machine refined by `m`
- `c` visible contexts of machine `m`. They define the context $\Gamma(m)$
- `x` variables defining machine machine `m` state
- `I(x)` Invariants de la machine `m`
- `T(x)` Theorems deduced from the context and invariant
- `v` expression defining a decreasing variant (either a natural number or a set)
- `ev1, ...` list of machine events describing state changes with at least an `INITIALISATION` event

Modification of state variable

Skip	Null/Empty Action
$x := E$	Becomes expression E (Simple Assignement)
$x :\in S$	Becomes element of S (Arbitrary choice in a set S)
$x :\mid P$	Becomes such that P (Arbitrary choice such that P
$f(x) := E$	Equivalent to $f := f \mathbin{\lhd\!\!-} \{x \mapsto E\}$

Events

Event : E	Before-After Predicate (BAA)
`begin` $x : \mid P(x, x')$ `end`	$P(x, x')$
`when` $G(x)$ `then` $x : \mid P(x, x')$ `end`	$G(x) \ \wedge \ P(x, x')$
`any` t `where` $G(t, x)$ `then` $x : \mid P(x, x', t)$ `end`	$\exists t . (G(t, x) \ \wedge \ P(x, x', t))$

Event : E	Guard : grd(E)
`begin` S `end`	*TRUE*
`when` $G(x)$ `then` T `end`	$G(x)$
`any` t `where` $G(t, x)$ `then` T `end`	$\exists t . G(t, x)$

Context

```
Context C0
Sets s
Constants c
Axioms Ax(s, c)
Theorems Tc(s, c)
End
```

Contexts

- constants (c)
- sets (s)
- Axioms Ax(s,c)
- Theorems Tc(s,c)

Machine

- `Machines`. Initial state + events (transitions between states).

- `Machines` : Variables (state), Events (transitions),
- Invariants $I(s, c, x_A)$, Theorems $T(s, c, x_A)$
- Proof Obligations
- Non determinism
- Interleaving semantics with stuttering
- Traces correspond to sequences of event triggerings

```
Machine Spec
Sees C0
Variables x_A
Invariants Inv(s, c, x_A)
Theorems T(s, c, x_A)
Events
  Event Initialisation =
                  begin
                     x_A :| Init(s, c, x'_A)
                  end ;
  Event An_event = Any l
                  Where
                     G_A(l,s,c,x_A)
                  Then
                     x_A :| AC1(l,s,c,x_A,x'_A)
                  End
  Event Another_event =
                  When
                     GG_A(s,c,x_A)
                  Then
                     x_A :| AC2(s,c,x_A,x'_A)
                  End
       ...
            End
```

Example

```
contexts
  data
sets
  MESSAGES
   AGENTS
   DATA
constants
   n
   infile
axioms
   axm1 : n ∈ ℕ
   axm2 : n ≠ 0
   axm3 : infile ∈ 1 .. n → DATA
end
```

```
MACHINE agents
SEES data
VARIABLES
    sent
    got
    lost
INVARIANTS
   inv1 : sent ⊆ AGENTS × AGENTS
   inv2 : got ⊆ AGENTS × AGENTS
   inv4 : (got ∪ lost) ⊆ sent
   inv6 : lost ⊆ AGENTS × AGENTS
   inv7 : got ∩ lost = ∅
```

```
INITIALISATION
BEGIN
    act1 : sent := ∅
    act2 : got := ∅
    act4 : lost := ∅
END
```

```
sending a message
ANY
    a
    b
WHERE
    grd11 : a ∈ AGENTS
    grd12 : b ∈ AGENTS
    grd1 : a ↦ b ∉ sent
THEN
   act11 : sent := sent ∪ {a ↦ b}
END
```

```
getting a message
ANY
   a
   b
WHERE
   grd11 : a ∈ AGENTS
  grd12 : b ∈ AGENTS
  grd13 : a ↦ b ∈ sent \ (got ∪ lost)
THEN
   act11 : got := got ∪ {a ↦ b}
END
```

```
loosing  a messge
ANY
   a
   b
WHERE    grd1 : a ∈ AGENTS
   grd2 : b ∈ AGENTS
   grd3 : a ↦ b ∈ sent \ (got ∪ lost)
THEN
  act1 : lost := lost ∪ {a ↦ b}
END
```

Obligation de preuve (Cours)

Guarded events

Let us consider an **abstract** event and the corresponding refining **concrete** event such that

```
EVENT  =          EVENT  =
  when              when
    G(x)              H(y)
  then              then
    x := E(x)         y := F(y)
  end               end
```

Invariant preservation proof obligation

Let $I(x)$ and $J(x,y)$ be the **invariants** , then we need to **prove** the **refinement invariant preservation** as

$$I(x) \wedge J(x,y) \wedge H(y) \implies G(x) \wedge J(E(x), F(y))$$

Parameterised events

Let us consider an **abstract** event and the corresponding refining **concrete** event such that

```
EVENT  =            EVENT  =
  any v where         any w where
    G(x,v)              H(y,w)
  then                then
    x := E(x,v)         y := F(y,w)
  end                 end
```

Invariant preservation proof obligation

Let $I(x)$ and $J(x,y)$ be the **invariants** , then we need to **prove** the **refinement invariant preservation** as

$$I(x) \wedge J(x,y) \wedge H(y,w) \implies \exists v \,.\, (\, G(x,v) \wedge J(E(x,v), F(y,w))\,)$$

New events

Let us consider a **new** event refining the **skip** event as follows

```
EVENT  =          EVENT  =
  SKIP              when
  end                 H(y)
                    then
                      y := F(y)
                    end
```

Invariant preservation proof obligation

Let $I(x)$ and $J(x,y)$ be the **invariants** , then we need to **prove** the **refinement invariant preservation** as

$$\boxed{I(x) \wedge J(x,y) \wedge H(y) \implies J(x, F(y))}$$

Obligation de preuve associé à un invariant associé à une action

Démontrer qu'un invariant reste vrai après une action

Exemple exam 2023 :

Obligation de preuve associée à `inv5` :

On doit démontrer que `Stored ∩ InTransit = ∅` reste vrai **après** l'exécution de l'événement `Send`.

1. **Hypothèses avant l'événement :**
   - L'invariant est vrai avant l'exécution de `Send` :

$$Stored \cap InTransit = \emptyset$$

2. **Actions de l'événement `Send` :**
   - `Stored` devient `Stored \ {m}`
   - `InTransit` devient `InTransit ∪ {m}`
3. **Obligation de preuve :** Après les actions de l'événement, on doit prouver que :

$$(Stored \setminus \{m\}) \cap (InTransit \cup \{m\}) = \emptyset$$

Ainsi il faut développer l' obligation de preuve pour la démontrer.

Les axiomes

Exemple exam 2023 :

3.1 Complétion de l'axiome `axm1` :

La constante `Network` représente un réseau sous forme d'un ensemble de liens directionnels entre les nœuds. Un lien directionnel peut être modélisé comme un couple de nœuds (relation binaire). Ainsi, le type de `Network` est donné par :

$$\text{axm1} : Network \subseteq NOEUDS \times NOEUDS$$

3.2 Complétion de l'axiome `axm2` :

On fait l'hypothèse que le réseau contient tous les nœuds de l'ensemble `NOEUDS`, ce qui signifie que chaque nœud du réseau doit apparaître au moins une fois en tant que source ou destination dans la relation `Network`. Cela peut être exprimé par :

$$\text{axm2} : \forall n \cdot (n \in NOEUDS \implies (\exists m \cdot ((n,m) \in Network \vee (m,n) \in Network)))$$

Types d'invariant

→ *Gluing invariant* : Un **gluing invariant** en Event-B est un invariant qui établit une relation entre les variables d'un modèle abstrait et celles d'un modèle raffiné, permettant de garantir que le raffinement respecte la structure du modèle original.

Example exam 2023 :

- $\text{Stored} = \text{dom}(\text{StoredAt})$
- $\text{InTransit} = \text{dom}(\text{InTransitTo})$

→ *Invariants structurels :* Ils définissent des propriétés fondamentales concernant la structure du modèle

Send

guard :

- **Grd1** : Le message **m** doit être stocké quelque part, donc **m ∈ Messages** et doit être stocké dans un noeud, donc **StoredAt(m) ≠ Ø**.
- **Grd2** : Il doit exister un lien entre le noeud où le message est stocké et le noeud où il se dirige, c'est-à-dire qu'il doit y avoir une relation dans le réseau entre le noeud où le message est stocké et le noeud **n** vers lequel il se dirige. Ce lien est vérifié par la présence d'une relation dans le réseau, que l'on peut formaliser avec une relation **Network**.

Voici donc la version complétée de la garde :

text Copier le code

```
grd1: m ∈ Messages ∧ StoredAt(m) ≠ Ø
grd2: n ∈ NODES ∧ (StoredAt(m), n) ∈ Network
```

Action :

- Retirer le message **m** de **StoredAt**, c'est-à-dire qu'il n'est plus stocké.
- Ajouter le message **m** dans **InTransitTo**, en associant le message **m** au noeud **n** (le noeud vers lequel il se dirige).

Voici la version complétée des actions de **Send** :

text Copier le code

```
StoredAt := StoredAt \ {m ↦ StoredAt(m)} ⊎ {m ↦ n}
InTransitTo := InTransitTo ∪ {m ↦ n}
```

Receive : (utilisation de if)

```
EVENT Receive
ANY m, n
WHERE
    grd1: m ∈ MESSAGES
    grd2: n ∈ NODES
    grd3: n = Destination(m) ∨ n ≠ Destination(m)
THEN
    IF n = Destination(m) THEN
        Done := Done ∪ {m}  -- Ajouter le message à Done
    ELSE
        -- Le message reste en transit, donc rien à faire ici
    END
END
```

Obligation de preuve :

1. Obligation de preuve de la préservation de l'invariant

L'invariant que nous devons préserver après l'exécution de l'événement Send est que tous les messages sont soit stockés, soit en transit. En d'autres termes, les messages doivent toujours être présents dans StoredAt ou InTransitTo après l'exécution de l'événement.
Invariant de la machine Route_1 :

- inv1 : StoredAt ∈ MESSAGES 7→ NODES
- inv2 : InTransitTo ∈ MESSAGES 7→ NODES

Cela signifie que les messages sont soit stockés, soit en transit, et que chaque message a une position associée dans l'une des deux relations (soit dans StoredAt, soit dans InTransitTo).
→ Démonstration :
Avant l'exécution de l'événement Send, le message m doit être stocké dans StoredAt (selon la garde grd1). Lors de l'exécution de l'événement Send, le message m est déplacé de StoredAt vers InTransitTo, où il est maintenant associé au noeud n.
L'action de l'événement Send est donc :
StoredAt := StoredAt \ {m ↦ StoredAt(m)} ⊎ {m ↦ n}
InTransitTo := InTransitTo ∪ {m ↦ n}

- Avant l'exécution de Send, m est dans StoredAt, donc StoredAt(m) ≠ ∅.
- Après l'exécution de Send, m est retiré de StoredAt et ajouté à InTransitTo.

Cela garantit que le message m est toujours présent dans l'une des deux relations, soit StoredAt, soit InTransitTo, ce qui préserve l'invariant.
Conclusion : L'invariant inv1 et inv2 est préservé, car après l'exécution de l'événement Send, tous les messages sont toujours dans StoredAt ou InTransitTo.

2. Obligation de preuve de la correction du raffinement

La correction du raffinement stipule que l'événement raffiné Send doit correspondre à l'événement abstrait, mais avec une plus grande précision, en intégrant les détails du réseau et des noeuds. Cela signifie que l'événement raffiné doit bien respecter les actions et préconditions définies dans l'événement abstrait.
Spécification de l'événement abstrait Send :

- Le message m doit être stocké (donc dans StoredAt) et il doit exister un lien entre le noeud où il est stocké et le noeud où il se dirige.
- Le message m doit être déplacé de l'état StoredAt vers InTransitTo, où il se dirige vers le noeud n.

Démonstration :
La garde de l'événement raffiné Send vérifie que le message est stocké et que le lien entre le noeud de stockage et le noeud de destination existe dans le réseau. Cette garde est formulée comme suit :
text
Copier le code
grd1: m ∈ Messages ∧ StoredAt(m) ≠ ∅
grd2: n ∈ NODES ∧ (StoredAt(m), n) ∈ Network

- grd1 assure que le message m est dans Messages et est stocké dans StoredAt.
- grd2 assure que le noeud n existe dans NODES et que le lien entre le noeud où le message est stocké et le noeud n existe dans le réseau ((StoredAt(m), n) ∈ Network).

L'action de l'événement Send consiste à déplacer le message de StoredAt vers InTransitTo en associant m au noeud n :
text
Copier le code
StoredAt := StoredAt \ {m ↦ StoredAt(m)} ⊎ {m ↦ n}
InTransitTo := InTransitTo ∪ {m ↦ n}

Cela respecte la spécification abstraite, car :

- Le message m est bien retiré de StoredAt et ajouté à InTransitTo.
- Le message m est bien envoyé vers le noeud n, comme spécifié dans la garde.

Conclusion : L'événement raffiné Send respecte bien la spécification de l'événement abstrait, car il assure que le message est bien stocké, que le lien existe dans le réseau, et que le message est déplacé vers le noeud approprié.

<u>Propriété de terminaison :</u>
Le type de propriété associée à cette question est une ***propriété de terminaison*** *(ou* ***propriété de l'éventuelle fin****), aussi appelée* ***propriété de l'éventuelle complétude****. Cette propriété garantit qu'une fois lancée, l'exécution du système atteindra un état final où tous les messages auront été traités, sans rester bloqués dans un état intermédiaire.*

<u>Définir une séquence :</u>
Exemple 2023 :

```
Route == seq NODES
```

ajouter élément : `# <n> }`

Q 10.3 2023 :

Hypothèse à ajouter pour garantir la validité de la route inversée :

Pour garantir que la route inversée est correcte, nous devons ajouter l'hypothèse suivante sur le réseau : **le réseau est acyclique et orienté**. Cela signifie que chaque lien entre deux noeuds a une direction bien définie et qu'il n'y a pas de cycles dans le réseau. Cela permet d'assurer que l'inversion de la route ne crée pas de boucle et que chaque noeud est accessible depuis son prédécesseur dans la route inversée.

<u>Questions de cours</u>

→ Différences entre contextes et machines, et leurs utilités respectives :
Un contexte (Context) définit des ensembles, des axiomes et des propriétés globales du système, comme des types, des constantes et des invariants. Il représente la structure statique du modèle.
Une machine (Machine) décrit des comportements dynamiques à travers des variables et des événements. Elle définit comment le système évolue en réponse aux événements.
Utilité : Le contexte fournit la base statique du système, tandis que la machine spécifie son comportement dynamique.
→ Interblocage (Deadlock) :
L'interblocage se produit lorsque deux ou plusieurs processus sont bloqués indéfiniment, chacun attendant que l'autre libère une ressource. Dans un modèle Event-B, il se caractérise par l'absence de progression possible des événements, généralement en raison de conditions de garde qui ne peuvent jamais être satisfaites simultanément.
→ Bonne définition (Well-definedness) :
Une formule est bien définie si son sens est clair et sans ambiguïté. Cela implique que chaque symbole dans la formule a une signification précise dans le contexte. Par exemple, l'expression x + y est bien définie si x et y sont des entiers. Cela se distingue de la correction syntaxique (respect des règles grammaticales) et du typage (adéquation entre types de variables et opérations).
→ Propriété qui relie les variables d'une machine concrète et d'une machine abstraite dans un raffinement :
La propriété de raffinement relie les variables d'une machine abstraite et d'une machine concrète. Elle garantit que la machine concrète respecte le comportement spécifié par la machine abstraite tout en étant plus détaillée ou optimisée. Son utilité est de s'assurer que le raffinement n'introduit pas d'erreurs tout en conservant les propriétés essentielles du modèle abstrait.
→ Une obligation de preuve déchargée (prouvée), peut-elle être considérée comme un théorème ?
Oui, une obligation de preuve déchargée est considérée comme un théorème car elle a été démontrée formellement. En d'autres termes, elle est une conséquence logique des axiomes et des invariants définis dans le modèle.
→ Expliquer pourquoi les obligations de preuve peuvent-elles générées automatiquement ?
Les obligations de preuve peuvent être générées automatiquement par des outils de vérification formelle. Ces outils analysent les invariants, les événements et les conditions de la machine pour en déduire les obligations nécessaires à la validation de la spécification.
→ Quel mécanisme de preuve est associé à la preuve d'un invariant ?
Le mécanisme de preuve associé à la preuve d'un invariant consiste à démontrer que l'invariant reste vrai à chaque étape du système, c'est-à-dire que pour toute exécution du système, l'invariant est préservé à chaque événement.
→ Une machine décrit des invariants et des théorèmes. Tout deux décrivent des propriétés de la machine. Pourquoi théorèmes et invariants ont-ils été différenciés ?
Les invariants sont des propriétés qui doivent toujours être vraies pendant l'exécution du système, tandis que les théorèmes sont des propriétés qui sont prouvées mais qui peuvent ne pas être constamment vérifiées. La différenciation permet de mieux organiser et structurer les propriétés dans le modèle.
→ Lorsqu'un variant est introduit, il doit décroître grâce aux événements dits "convergents". Quelles propriétés peut-on garantir grâce à la présence d'un variant décroissant ?
La présence d'un variant décroissant garantit que le système ne peut pas tourner indéfiniment sans progression, assurant ainsi que le système finit par atteindre un état terminal ou une condition d'arrêt.
→ Le raffinement établit une relation de simulation entre deux machines. Expliquer le rôle de la relation de simulation ?
La relation de simulation entre deux machines garantit que la machine concrète, qui est un raffinement de la machine abstraite, respecte les comportements de la machine abstraite tout en étant plus détaillée ou optimisée. Elle permet de prouver que la machine concrète conserve les propriétés essentielles du modèle abstrait.
→ Qu'est-ce qu'une obligation de preuve ?
Une obligation de preuve (ou preuve déchargée) est une condition que le système doit satisfaire pour garantir que la spécification est correcte, notamment que les invariants sont respectés et que les événements n'introduisent pas d'erreurs. Cela signifie qu'une fois prouvée, l'obligation peut être considérée comme validée sans besoin de nouvelles vérifications.
→ Comment ces obligations de preuve sont-elles générées ?
Les obligations de preuve sont générées automatiquement par un outil de preuve formelle (comme Atelier B). Elles sont dérivées des invariants, des événements et de la logique de raffinement de la machine, garantissant que les conditions spécifiées dans la machine sont toujours respectées à travers les différentes étapes de raffinement.
→ Quel est le rôle de l'invariant dans la spécification de propriétés de systèmes ?
L'invariant définit une propriété qui doit toujours être vraie pendant l'exécution du système. Il sert à spécifier les conditions que les variables doivent satisfaire à tout moment, assurant ainsi la cohérence du système tout au long de son fonctionnement.
→ Le maintien des invariants par l'événement d'initialisation et par chaque événement d'une machine permet de garantir le maintien de cet invariant pour tous les comportements décrits par cette machine. Expliquer pourquoi.
Le maintien des invariants est assuré par les événements qui respectent les conditions imposées par ces invariants. L'événement d'initialisation garantit que l'invariant est vrai dès le début du système, et chaque événement subséquent est vérifié pour s'assurer qu'il ne viole pas cet invariant, préservant ainsi la validité de la machine à chaque étape.
→ Lorsqu'un variant est introduit, il doit décroître grâce aux événements dits "convergents". Quelles sont les obligations de preuve du variant associées à ces événements ?
Les obligations de preuve pour un variant sont que celui-ci doit toujours décroître avec chaque événement convergent, et il doit finir par atteindre une valeur minimale, assurant ainsi que l'exécution du système finit par se stabiliser ou atteindre un état final.