Intégration de méthodes de spécification et développement

Christian Attiogbé
Equipe COLOSS – LINA
Christian.Attiogbe@univ-nantes.fr

Equipe COLOSS

Composants et Logiciels Sûrs

Membres :

Christian	Attiogbé
Pascal	André
Gilles	Ardourel
Henri	Habrias
Cedric	Stoquer

Thèmes de recherche :

Génie logiciel, Méthodes formelles, Intégration de méthodes,

Vérification, Sureté des composants et logiciels

Sujets de stage :

1. COLOSS-P1-S1 : Processus composite de vérification de modèles
2. COLOSS-P2-S1 : Stratégies d'implémentation de modèle à composants flexibles
3. COLOSS-P3-S1 : Noyau d'un environnement de spécifications multi-paradigmes
4. COLOSS-P3-S2 : Vérification en B des propriétés fonctionnelles de composants Kmelia
5. COLOSS-P4-S1 : Etude sur l'intégration du GRAFCET au B événementiel

Thèmes abordés dans ce cours

. Motivations - Méthodes formelles pour les systèmes critiques (et les autres)

. Garantie de la correction des logiciels par la preuve des propriétés attendues

. Problématique générale de l'intégration de méthodes de développement formelles

. Les problèmes d'interactions logiques/sémantiques

. Application au développement de composants logiciels sûrs

Montrez-moi un modèle !

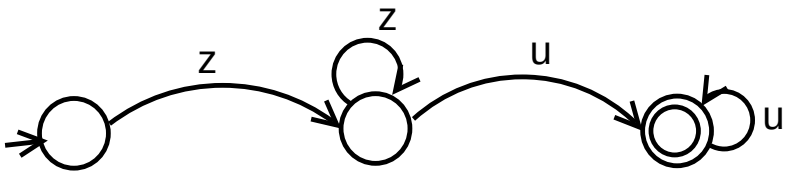


FIG. 1 – Exemple de modèle

Développement correct de logiciels

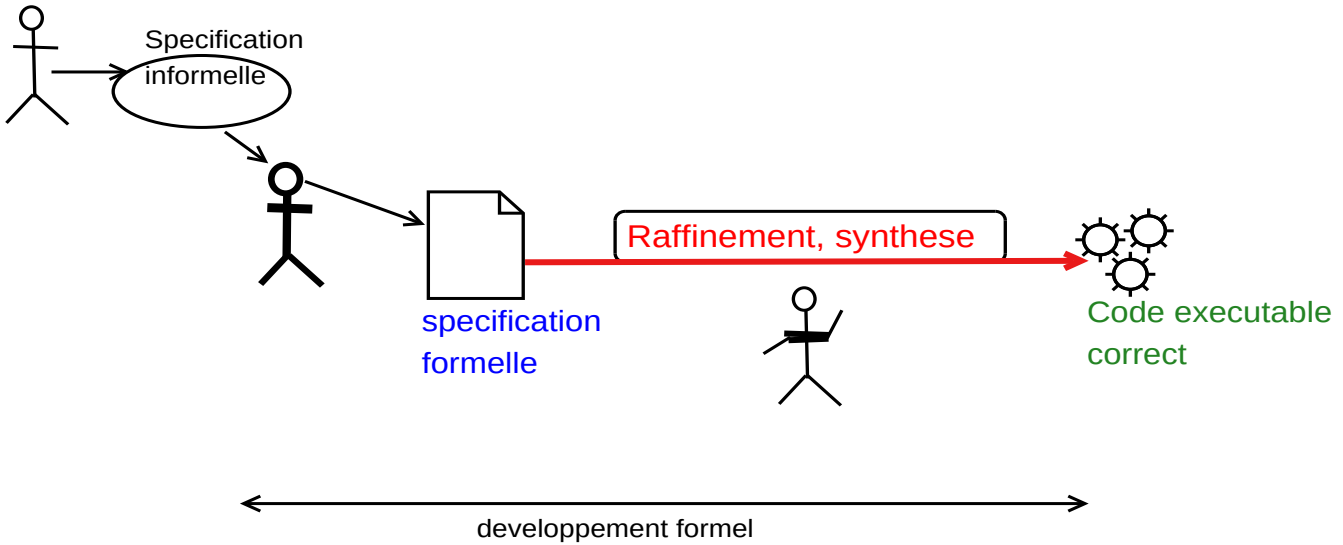


FIG. 2 – Activité de développement formel

Première partie

Méthodes Formelles

Introduction

Développement des systèmes informatiques

- Système informatique ?
- Développement ?
- Quels problèmes ?
- Quels concepts, théories, méthodes, techniques, outils ?
- Etat de l'art ?
- Besoins ?

Méthodes de développement formel (résumé)

Interprétation de l'isomorphisme de Curry-Howard :

Preuve	*Axiomes* / *Theoremes*

équiv. à

Développement	*Specifications* / *Programmes*

♡

Développement de systèmes informatiques

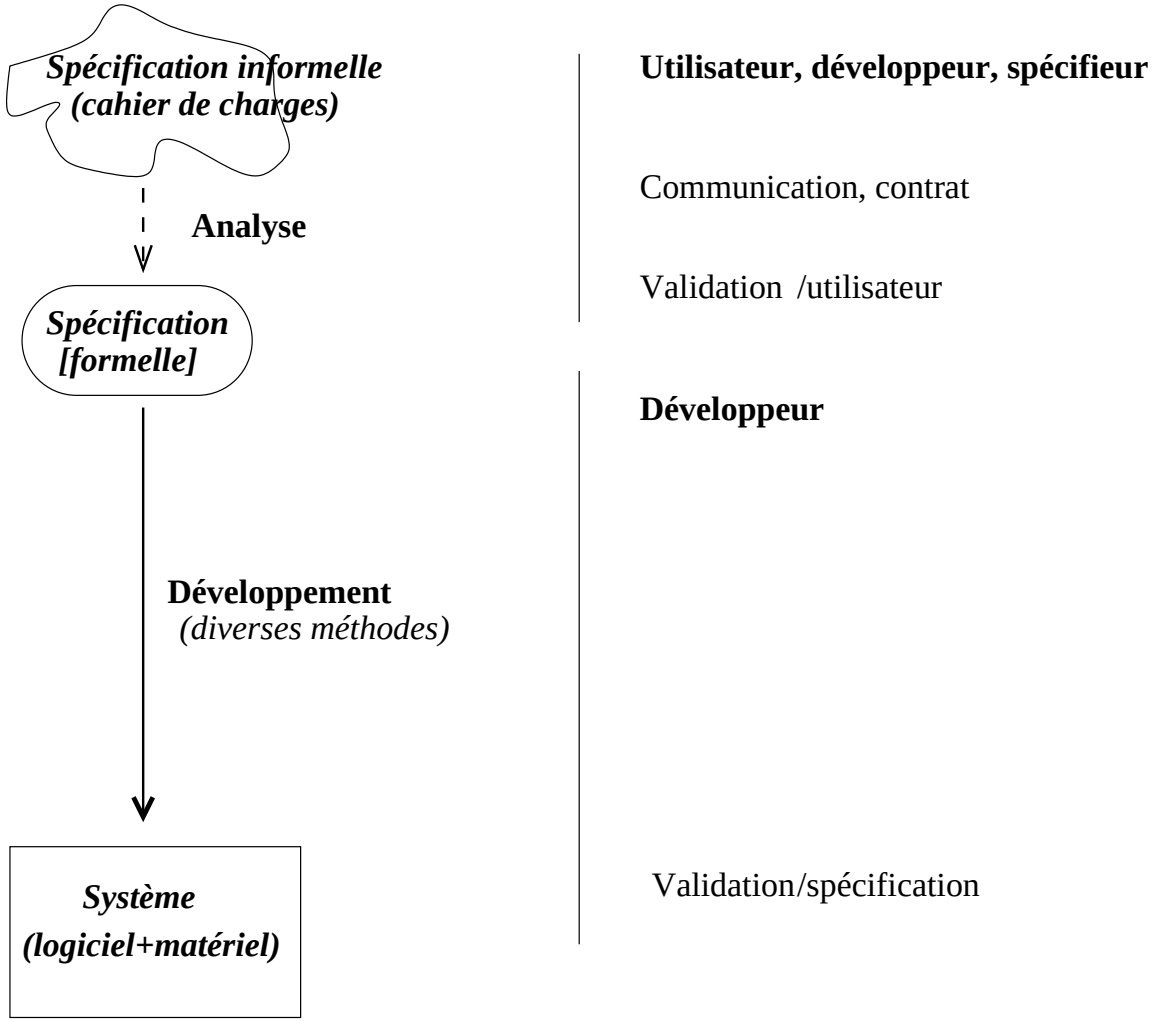


FIG. 3 – Problématique du développement de système

Développement

Cahier de charges (spécification informelle) → système informatique

Plusieurs étapes :

- Analyse (*Analysis*),
- Spécification, Modélisation (*Specification*, *Modelling*),
- Conception (*Design*),
- Implantation (*Implementation*)
- Maintenance (*Maintenance*)

Développement de systèmes (suite)

Nature des systèmes (logiciels) :

- séquentiels, parallèles (concurrents),
- autonomes (transformationnels), réactifs, temps-réels
- centralisés, répartis,
- embarqués,
- protocoles de communication
- ...

⇒ plusieurs types de systèmes

Quelques problèmes

- Décrire précisément le système voulu spécification
- Construire correctement le logiciel développement
- S'assurer que le logiciel construit est correct par rapport aux besoins
- Suivi du système

Pratique

- Nature des systèmes *complexes* → multifacette
- Plusieurs méthodes :
  - Méthodes semi-formelles
  - Méthodes formelles (intégrées) → traitement des systèmes complexes

⇒ maîtrise de plusieurs méthodes

Quelques définitions

Modélisation :

HOARE : A scientific theory is formalised as a mathematical model of reality, from which can be deduced or calculated the observable properties and of a well-defined class of processes in the physical world.

Il y a deux principales notions de modèles (en informatique).

1. Modèle = une approximation de la réalité par une structure mathématique.
   Un objet O est modèle d'une réalité R, si O permet de répondre aux questions que l'on se pose sur R.

En Mathématique, Physique, ... système d'équations portant sur des grandeurs (masses, énergie, ...) ou des lois hypothétiques.

2. (Logique, théorie des modèles)

   Un modèle d'une théorie T est une structure dans laquelle les axiomes de T sont valides.

   Une *structure* S est modèle d'une théorie T, ou bien S satisfait T si toute formule de T est satisfaite dans S.

   La réalité est un modèle d'une théorie !

   Théorie (du 1er ordre) = tout ensemble de formules logiques (du 1er ordre) dans un langage donné (précisément défini).

   Modèle comme interpétation d'une spécification - une algèbre comme modèle d'une spécification algébrique (axiomatisation).

Ces deux utilisations de *modèle* se retrouvent dans les approches *orientée modèle (ou état)* et *orientée propriétés.*

Dans le langage courant,

- *modèle* = (archétype), ce qui sert ou doit servir d'objet d'imitation pour reproduire quelque chose.
- *modèle* = (paradigme), modèle de déclinaison, de conjugaison, etc
- *modèle* = (référence), ...

Exemples de théorie :

La théorie des ensembles : elle est basée sur un ensemble d'axiomes.

Les objets de cette théorie sont appelés ensembles.

La classe des ensembles est appelée univers.

Les axiomes de la théorie des ensembles (de Zermelo+Fraenkel) sont les suivants :

- Axiome de l'ensemble vide : *il existe un ensemble qui ne contient aucun élément : c'est l'ensemble vide.*
- Axiome d'extensionalité : *deux ensembles sont égaux si et seulement si ils contiennent exactement les mêmes éléments*
- Axiome de l'union : *l'une union d'ensembles est un ensemble*
- Axiome de l'ensemble des parties :*les parties d'un ensembles forment une partie*
- Axiome du schéma de remplacement (Fraenkel, 1922) : *Lorsqu'on définit une fonction par des formules de la théorie des ensembles, alors éléments pour lesquels cette fonction vérifie une certaine propriété forment encore un ensemble.*

De plus, on ajoute à ces axiomes, l'axiome de l'infini : *il existe un ordinal infini.*

ZFC = ZF + axiome du choix

– Axiome du choix : *Soit une famille d'ensemble disjoints, si on considère un élément de chaque ensemble de la famille, alors on en forme un ensemble.*

Quelques définitions (suite)

Méthode semiformelle =

– Langage graphique [+ formel] (syntaxe précise et sémantique non précise) et

– Outils d'analyse divers.

→ Combinaison de langages/méthodes/techniques n'ayant pas tous une sémantique précise.

Exemples : JSD, OMT, OOX, UML

Quelques définitions (suite)

Méthode formelle =

– Langage formel (syntaxe précise et sémantique précise) et

– Système de preuve ou de raisonnement formel.

Exemples : CCS, CSP, HOL, Z, B

Développement formel =

– transformation systématique des spécifications en programmes en utilisant des lois prédéfinies.

Exemples : Synthèse, Raffinement

Quelques définitions (suite)

Vérification : montrer que le système (S)est correct par rapport à des propriétés(P)

$$S \models P$$

Validation : montrer que le système est correct par rapport aux spécifications informelles

$$S \sim S_{informelle}$$

Raisonnement formel : Consiste à **appliquer un système formel à une spécification**.

Exemples de raisonnement formel :

- Rafinement de specification,
- vérification des proprétés d'un système,
- validation par vérification,
- preuve de théorèmes (*theorem proving*),
- analyse d'un système (représenté par machine à états) par rapport à des propriétés (*model checking*).

$\Longrightarrow$ La logique est le fondement des approches formelles

Logique du premier ordre

– Proposition

Une proposition est une expression du langage dont la grammaire est la suivante :

$$\begin{array}{lll} prop & ::= & P, Q, E, \ldots \\ & | & prop \Rightarrow prop \\ & | & prop \wedge prop \\ & | & \neg\ prop \end{array}$$

Des parenthèses peuvent être utilisées si nécéssaire.

Exemples de proposition

le ciel est rugueux

Pierre conduit un velo

$$0 > 3$$

$$GrandesOreilles(elephant) \Rightarrow Africain(elephant)$$

Règles d'inférence du calcul propositionnel

système de raisonnement (preuve) (avec un métalangage)

$\wedge$ *intr*	$\dfrac{HYP \vdash P \quad HYP \vdash Q}{HYP \vdash P \wedge Q}$	souvent appliquée en arrière pour diviser un but en deux sous-buts plus simples avec les même hypothèses.
$\wedge$ *elim*	$\dfrac{HYP \vdash P \wedge Q}{HYP \vdash P \quad HYP \vdash Q}$	
$\Rightarrow$ *intr*	$\dfrac{HYP, P \vdash Q}{HYP \vdash P \Rightarrow Q}$	C'est la règle de déduction
$\Rightarrow$ *elim*	$\dfrac{HYP \vdash P \Rightarrow Q}{HYP, P \vdash Q}$	C'est l'anti-déduction

Modus Ponens	$\frac{HYP \vdash P \quad HYP \vdash P \Rightarrow Q}{HYP \vdash Q}$	
Contradiction	$\frac{HYP, \neg Q \vdash P \quad HYP, \neg Q \vdash \neg P}{HYP \vdash Q}$	première règle traitant du $\neg$
	$\frac{HYP, Q \vdash P \quad HYP, Q \vdash \neg P}{HYP \vdash \neg Q}$	deuxième règle du $\neg$

Logique des Prédicats

Le calcul des propositions : *vérité absolue.*

Le calcul des prédicats : *vérités relatives.*

On fait une extension du calcul propositionnel.

Formation des prédicats

$$
\begin{array}{ll}
\mathit{Predicat} ::= & \mathit{Predicat} \Rightarrow \mathit{Predicat} \\
& \mathit{Predicat} \wedge \mathit{Predicat} \\
& \neg\ \mathit{Predicat} \\
& \forall\ \mathit{Variable}.\mathit{Predicat} \\
& [\mathit{Variable} := \mathit{Expression}]\mathit{Predicat} \\
& \mathit{Expression} = \mathit{Expression} \\
\mathit{Expression} ::= & \mathit{Variable} \\
& [\mathit{Variable} := \mathit{Expression}]\mathit{Expression} \\
& \ldots \\
\mathit{Variable} ::= & \mathit{Identifier} \\
\ldots &
\end{array}
$$

Comment utiliser les prédicats

$$x > 2$$

$$x \in \mathbb{N} \Rightarrow x \geq 0$$

Dans les prédicats on utilise deux sortes de variables : les *variables libres* et les *variables liées.*

- Substitution

$$[x := 5](x \in \mathbb{N} \Rightarrow x \geq 0)$$

$$(5 \in \mathbb{N} \Rightarrow 5 \geq 0)$$

$$[x := elephant](GrandesOreilles(x) \Rightarrow Africain(x))$$

- Quantification

$$\forall\, x.GrandesOreilles(x) \Longrightarrow Africain(x),$$

Cycles de vie

- pendant longtemps, support méthodologique du développement du logiciel.
- les plus représentatifs de ces cycles de vie :
  - Cycle en V,
  - Cycle en cascade,
  - Cycle de Balzer

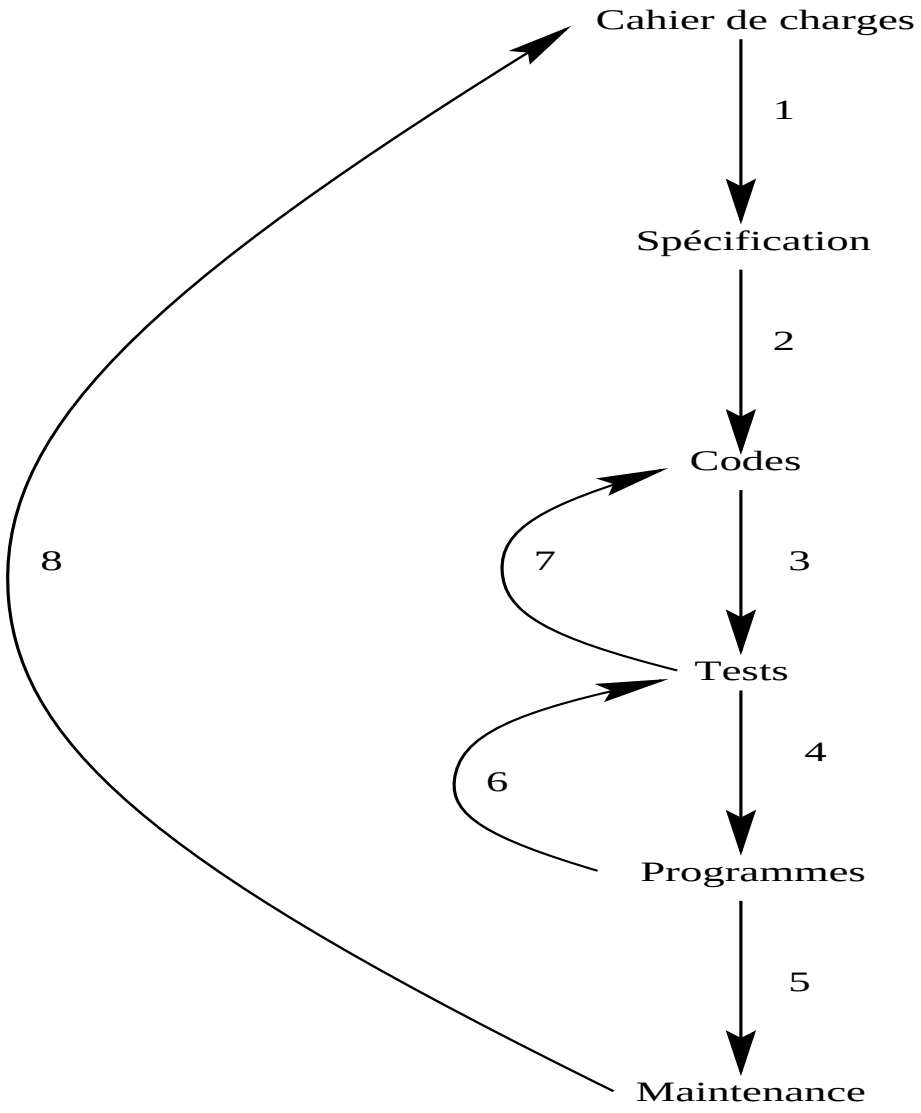


FIG. 4 – Cycle de vie en cascade (Boehm 1977)

Intérêts et limites des méthodes semi-formelles

– SADT, SA-RT, SSADM, ...
– JSD-JSP,
– Merise, Axial, ...
– OOA, OMT, UML
– ...

L'analyse du problème est faite.
Contribution positive même si suffisante.
Le problème est dégrossi.

→ impossible de raisonner formellement sur le système en vue.

→ Il peut y avoir des ambiguités, des erreurs.

Spécification formelle

⇒ Expression dans un langage formel du quoi d'un système à développer.

– Résultat de la phase d'analyse

– Plusieurs formes possibles selon la nature du système

On parle de **langages** ou **formalismes** de spécification formelle :

Logique, Z, Langages de spécification algébriques, algèbres de processus, etc

Démarche de spécification

Les aspects données ou les aspects opérations ?

- **Les données d'un système permettent de décrire les états du système**
- **les opérations du système permettent de décrire son fonctionnement ou son comportement par des axiomes**

On parle du paradigme des données et du paradigme des opérations.

Il convient de distinguer les opérations exprimant le comportement d'un système des opérations caractérisant les données du système. Les premières opèrent sur les données du système alors que les dernières permettent de construire et exploiter les données du système.

Il y a un troisième paradigme qui semble transversal.

C'est le paradigme des processus (les algèbres de processus).

Dans ce paradigme des processus les systèmes sont décrits par des règles ou des équations exprimant leur comportement ou leurs états.

Les principales algèbres de processus à la base des autres formalismes sont :

- CSP (Hoare)
- CCS (Milner)
- ACP (Bergstra)

Eléments de classification

La principale classification des approches consiste à distinguer **deux principales approches** de spécifications formelles et de méthodes formelles :

– l'approche opérationnelle : orientée modèle ou *état*

– et l'approche axiomatique : orientée propriétés.

D'autres classifications (de bas niveau) sont rencontrées :

→ classification basée sur les paradigmes :

- *données* : data-oriented
  + *états*,
  + *opérations*,
- *processus* : process-oriented
- *hybride*

Classification(suite)

→ une classification basée sur les langages :

- les langages généraux : Z, spécifications algébriques, HOL, PVS, etc
- les langages orientés 'contrôle' : CCS, CSP, $SIGNAL$, $ESTEREL$, $LUSTRE$, etc

Spécifications orientées états

(model-based approach ou state-based approach)

On privilégie les données du système et l'état (les états) du système.

- **Principe** Une spécification permet de construire, à l'aide des concepts de base fournis, un modèle du système (logiciel) à développer. Ce modèle doit avoir les propriétés du système. On peut ensuite raisonner sur le fonctionnement du système en utilisant le modèle.
- **Concepts de base** On utilise essentiellement les mathématiques discrètes (théorie des ensembles), la Logique, les systèmes de transitions

– **Quelques exemples de formalismes** Z, VDM, B pour les systèmes séquentiels,
Réseaux de Pétri, CCS, CSP, Unity, ... pour les systèmes concurrents et distribués. (CCS et CSP relèvent du paradigme des processus) et les 'hybrides' RAISE (Combine VDM et CSP), VDM-SL et Réseaux de Petri, ZCCS (combine Z et CCS),

Spécifications orientées propriétés ou axiomatiques

(axiomatic approach, algebraic approach, operation oriented)

- **Principe** La spécification permet de construire une axiomatisation qui fournit les propriétés (comportementales) du système à développer en utilisant les concepts de base cités ci-après.
- **Concepts de base**

  Logique,

  Algèbre,

  Domaines définis inductivement.

– **Exemples de formalisme** les langages de spécification algébriques : LPG, ASL, ACT ONE, CLEAR, OBJ, PLUSS, LARCH, CASL ... pour les systèmes séquentiels
Les logiques modales pour les systèmes concurrents.
l'algèbre de processus : LOTOS (combine ACT ONE et CCS, CSP) pour les systèmes concurrents.

Spécifications hybrides (hybrid approach)

- **Principe** La spécification permet de compléter une axiomatisation par un modèle de données ou bien de compléter un modèle de données par une axiomatisation.
- **Concepts de base**
  Concepts de base des approches orientées état et propriétés.
- **Exemples de formalisme**
  - Extended LOTOS (combine ACT ONE, CCS et CSP),
  - ...

Méthodes de développement formel

Il y a plusieurs approches dans les méthodes formelles :

- l'**approche opérationnelle** :
  le développement est basé sur la description d'un système par un modèle qui a les propriétés du système.
  Vérification des propriétés sur le modèle.
  Des raffinements *corrects* successifs sont effectués sur le modèle.
- l'**approche axiomatique** :
  le développement est basé sur l'axiomatisation qui décrit le comportement du système.
  Des vérifications des propriétés sont effectuées, raffinements possibles.
- l'**approche hybride**

Méthodes de développement formel (suite)

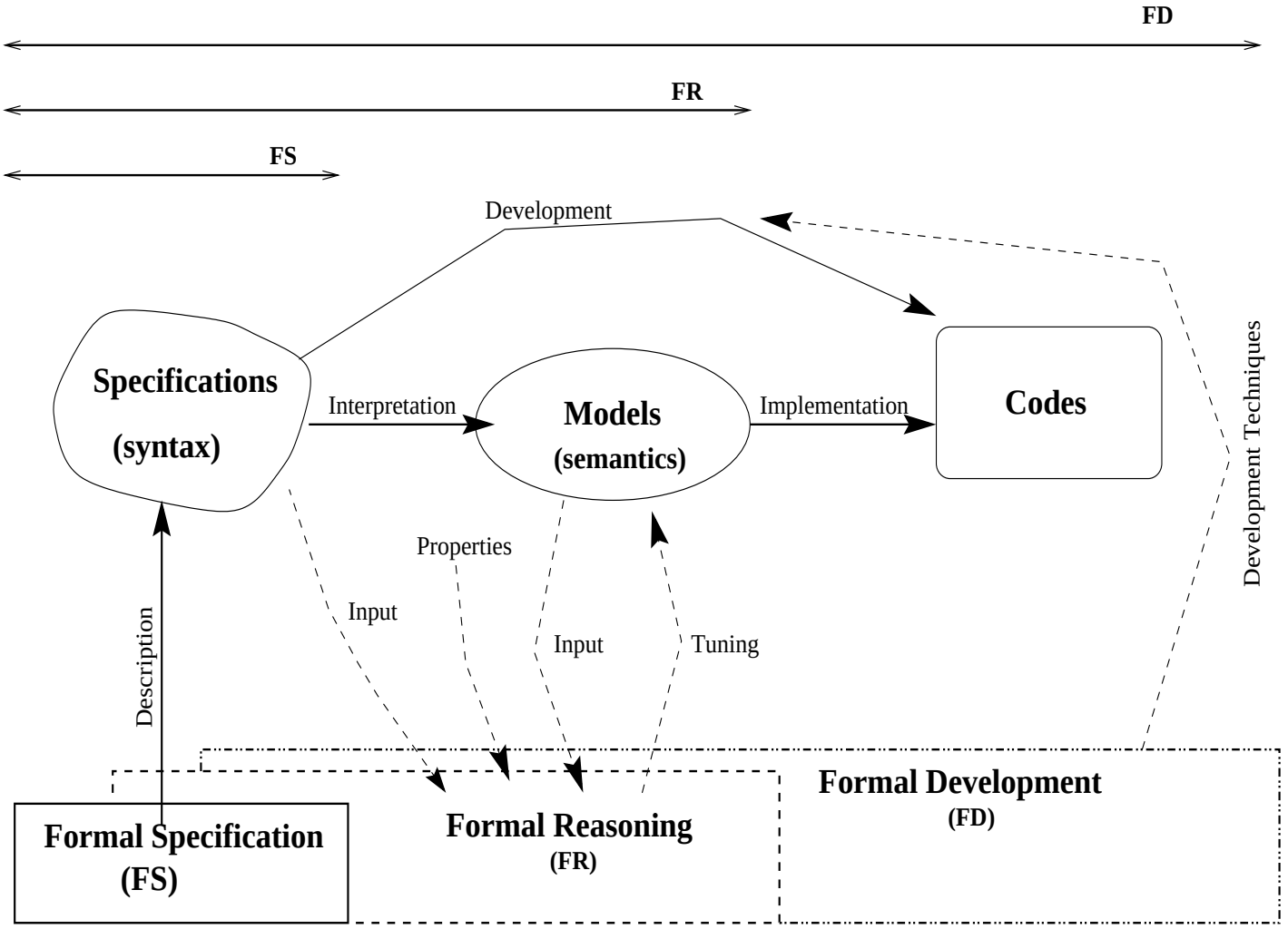


FIG. 5 – Eléments de développement formel

Méthodes de développement formel (suite)

Pour résumer l'essentiel des idées nous reprenons le slogan :

prouver c'est programmer et
programmer c'est prouver !

La deuxième façon de résumer est de considérer l'interprétation suivante de l'isomorphisme de Curry-Howard.

Preuve $\dfrac{\textit{Axiomes}}{\textit{Theoremes}}$ équiv. à **Développement** $\dfrac{\textit{Specifications}}{\textit{Programmes}}$

Cycle de vie adapté aux méthodes formelles

Avec les méthodes formelles les cycles de vie classiques ne marchent plus.
Il faut les adapter.

Le cycle de vie de Balzer représente la nouvelle famille de cycles de vie adaptés au développement formel (ou rigoureux !).

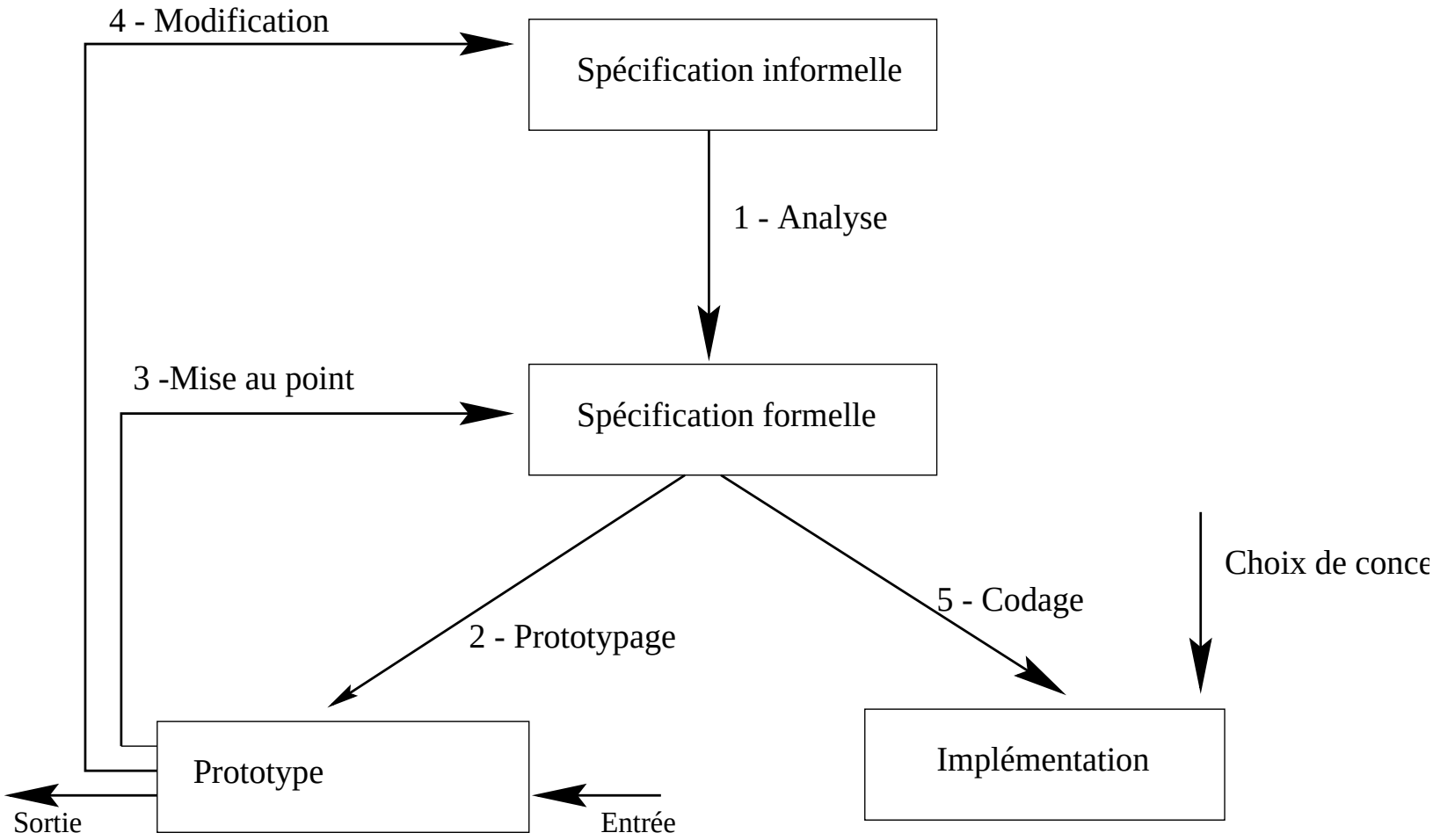


FIG. 6 – Cycle de vie de Balzer

Caractérisation des méthodes : un aperçu

Plusieurs caractéristiques peuvent être utilisées pour caractériser les méthodes.

Considérons les caractéristiques syntaxiques et sémantiques :

- L'aspect syntaxique → les moyens de description des spécifications formelles
- L'aspect sémantique → les possibilités de description sémantique associées aux spécifications

Formalismes	Bases théoriques	Modèles sémantiques
VDM, Z	logique 1er ordre, Théorie des ensembles	Modèles à états
B	logique 1er ordre, Théorie des ensembles, AMN	Modèles, transformateur de prédicats, WP
Larch, OBJ, CASL	Signature, Logique, Categories	Algebre universelle
TLA	Logique, logiques Tempor., Theo. des ens.	Modèles, Kripke
CCS, CSP, ACP	processus, events, channels	Traces, Bissimulation, Failure-divergence

Formalismes	Bases théoriques	Modèles sémantiques
π-calculus	Processus, Mobilité	Systèmes de transition, bissimulation
Actions Systems	Processus	Modèles à états, Traces
Signal	processuss, logic, adhoc	Modèles à états
HOL	logique Equationnelle	Axiomatisation
Coq	Logique, Theorie types constructive	Axiomatisation
PVS	Logique équationnelle, Logique Ordre Sup	Axiomatisation

Eléments d'analyse

Quelles méthodes pour quels systèmes ?

Le terme *méthode* désigne à la fois les formalismes ou les techniques.

Selon les systèmes à développer → **une famille de techniques, méthodes, outils**.

– C'est un fait pour les spécialistes des méthodes formelles.
– Malheureusement, la pratique est loin de cet état de fait.
– pour les non spécialistes, ce problème du choix de la (des) bonne(s) méthode(s) n'est pas évident.

→ C'est aux spécialistes de guider le développement de logiciels

Eléments d'analyse

Quels systèmes ?

Les **systèmes complexes** présentent plusieurs facettes → multifacettes.

⇒ **Irréaliste** : Développer formellement un système avec une méthode (spécification, conception, vérification, simulation, implémentation, tests).

La nature des systèmes complexes oblige le développeur à utiliser les formalismes, techniques et outils appropriés au bon moment

⇒ **Nécessité : Intégration de plusieurs approches pour le développement des systèmes informatiques : *intégration de méthodes*.**

Méthodes en génie logiciel

Méthode formelle =

- Langage de spécification
- Système formel (de raisonnement)

- Langage de spécification
  - Logique, théorie des ensembles : langage de données
  - Systèmes de transition, algèbres de processus : langage de comportement
- Système formel (de raisonnement)
  - Systèmes de type
  - Prouveur de théorèmes (*theorem proving*)
  - Vérificateur de modèles (*model checking*)

Développement formel

Développement formel de logiciel =

– Transformation systématique d'un modèle mathématique en code exécutable.

= Transformation de l'abstrait en concret

= Passage des structures mathématiques aux structures informatiques

= Raffinement jusqu'au code dans un langage de progr.

Exemple : B = Méthode formelle

+ théorie de raffinement (de machines abstraites)

⇒ méthode de développement formel

Aperçu de méthodes formelles

Réseaux de Petri

Algèbres de processus

Méthode B

Introduction aux Réseaux de Petri

Formellement, un réseau de Pétri (P-net) est un 4-uplet $(P,\ T,\ Pre,\ Post)$ où :

- P est un ens. fini de places, (avec $\mid P \mid = m$, le cardinal de P) ;
- T est un ens. fini de transitions, (avec $\mid T \mid = n$, le cardinal de T) ;
- P et T sont disjoints $(P \cap T = \{\})$;
- $Pre : P \times T \rightarrow \mathbb{N}$ est une fonction d'entrée, $Pre(p, t)$ dénote le nombre d'arcs de la place p à la transition t ;
- $Post : P \times T \rightarrow \mathbb{N}$ est une fonction de sortie, $Post(p, t)$ dénote le nombre d'arcs de la transition t la place p.

Introduction aux Réseaux de Petri

Pratiquement, un P-net est un graphe orienté bipartie dont les arcs connectent des noeuds des deux ensembles (places, transitions).

Le graphe associé a un P-net N est décrit par :

- Γ_p les transitions atteignables depuis chaque place :
  $\forall\, p \in P\,.\,\Gamma_p(p) = \{t \in T \mid Pre(p, t) > 0\}$
- Γ_t les places atteignables depuis chaque transition :
  $\forall\, t \in T\,.\,\Gamma_t(t) = \{p \in P \mid Post(p, t) > 0\}$
- W_{in} le poids de chaque arc en entrée :
  $\forall\, p \in P, \forall\, t \in T\,.\,W_{in}(p, t) = Pre(p, t)$ et
- W_{out} le poids de chaque arc en sortie :
  $\forall\, p \in P, \forall\, t \in T\,.\,W_{out}(p, t) = Post(p, t)$

Le graphe associé à un P-net est la représentation abstraite qui permet de l'analyser.

Introduction aux Réseaux de Pétri

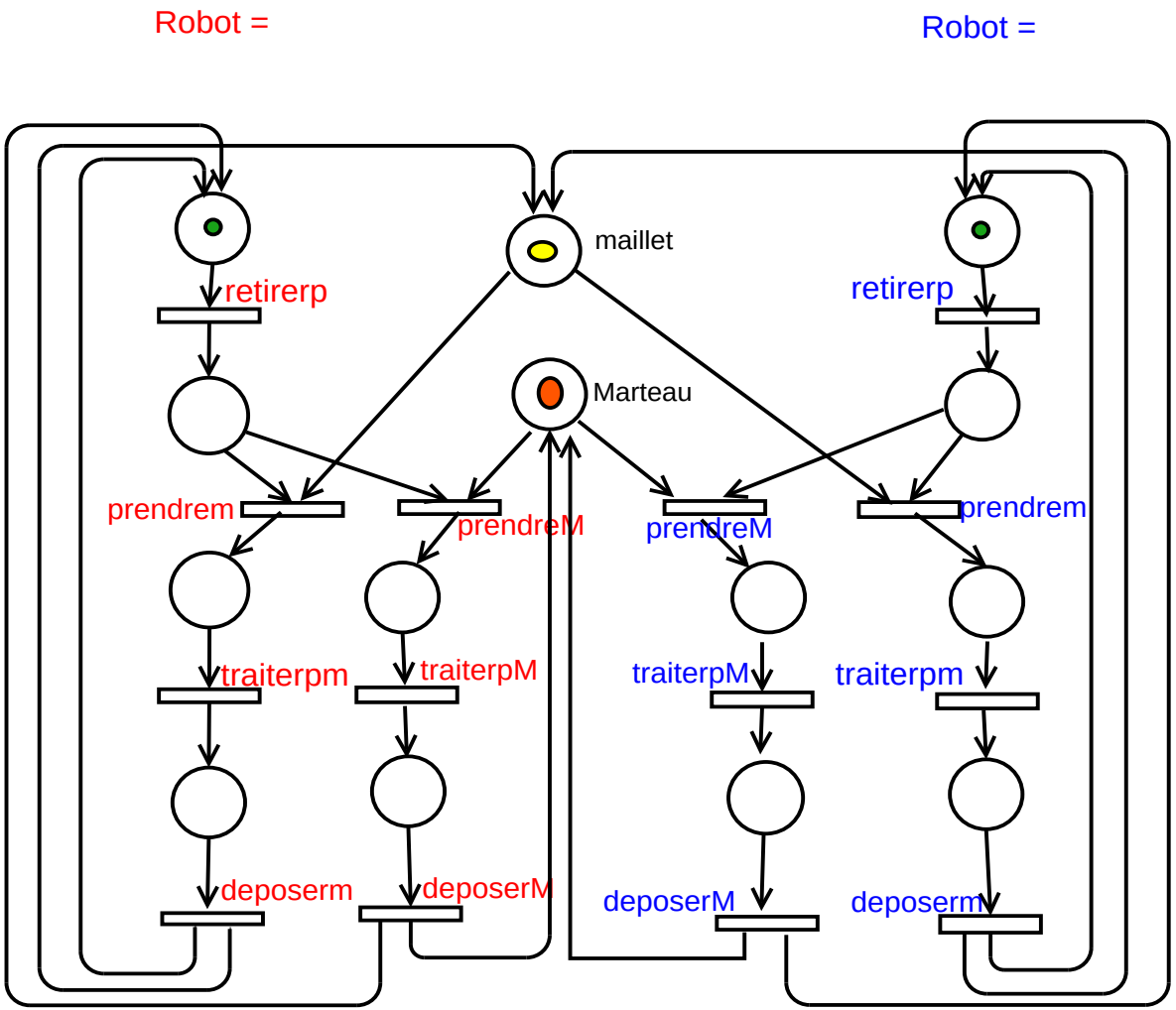


FIG. 7 – Réseau de Petri avec concurrence

Introduction aux algèbres de processus

– *Algèbres de processus* =

une catégorie de formalismes qui permettent de *décrire* et d'*analyser* les comportements de *systèmes (processus) concurrents*.

– Historique
  – Théorie des automates,
  – Théorie des réseaux de Petri,
  – Algèbres de processus :

  CSP (Communicating Sequential Processes) ; C.A.R. Hoare, 1978, `occam + Transputer`

  CCS (Calculus of Communicating Systems) ; R. Milner, 1985

  ...

Caractéristiques principales des algèbres de processus

- Spécification et étude des systèmes concurrents (communication, synchronisation)
- Abstraction sur les comportements,
- Mode de synchronisation (synchrone, RdV, actions complémentaires), etc
- Mode de composition (parallèle, entrelacement, etc),
- Modèles sémantiques (opérationnelle, traces, bissimulation, *failure-divergence*).

Concepts fondamentaux

Alphabets

L'évolution d'un processus est décrite à l'aide des noms des actions qu'il entreprend : *alphabet*.

Expressions régulières

Combinaison de noms d'action (*alphabet*) à l'aide d'opérateurs prédéfinis.

Automate à états.

Processus élémentaire

Processus = évolution séquentielle, exprimée à l'aide de combinaison d'actions et d'opérateurs (**séquence, choix, arrêt**) : on parle de *comportement* (***behaviour***).

Concepts fondamentaux (suite)

Processus prédéfinis

– Un procesus qui ne termine pas (***deadlock***) :
il ne peut plus effectuer aucune action de son alphabet.
en CCS c'est **0**.

En CSP c'est **stop**

– Un processus qui termine mais qui ne peut effectuer aucune action
En CSP c'est `skip`.

Principaux opérateurs (illustration avec CCS)

Séquence (notée .)

Une action suivie d'un comportement (le reste des actions).

Soit un alphabet $A = \{lire, ecrire, traiter, ouvrir, piece, pOutil, dOutil, usiner\}$

```
Processus Ro =   pOutil.traiter.dOutil.0

Processus Wenix =   lire.traiter.ecrire.Wenix
```

Principaux opérateurs (illustration avec CCS)

Choix de comportements (noté +)

Le processus s'engage dans un comportement ou un autre :

non-déterminisme

```
lire.0 + ecrire.0

lire.0 + ecrire.(traiter.0 + ouvrir.0)

Processus Robot = piece.pOutil.usiner.dOutil.Robot
                  + abandonner.0
```

Communication

Communication : seul moyen de **synchroniser** et d'**échanger des valeurs** entre les processus.

La communication est effectuée via des **actions complémentaires**.

En CCS on parle de **ports** de communication.

Opérateur d'émission sur un port : $action(e)$
Opérateur de réception sur un port : $\overline{action}(x))$

les actions complémentaires $action$ et $\overline{action}$ se synchronisent et évoluent silencieusement (action τ).

Exemple

```
P1 = pa(va).pb(vb).'pc(plus(va,vb)).0
           lit deux valeurs va et vb, puis emet leur somme

P2 = 'pa(a).'pb(b).pc(ab)
           emet 2 valeurs a, b puis attent leur somme


Sys = P1|P2
```

Communication synchrone (à la CCS)

synchronisation et **communication**

– **symétrique** : bloquant pour l'émission **et** la réception

– **binaire**

Dans les algèbres de processus, il existe d'autres modes de communication : asynchrone, diffusion

Opérateurs de composition parallèle

Notation de CCS : `|`

`(P1 | P2)\ L` signifie :

P1 et P2 se déroulent en parallèle

L est une liste d'actions

P1 et P2 doivent se *synchroniser* sur les actions dans L

P1 et P2 ne doivent pas se synchroniser sur des actions n'appartenant pas à L.

Exemple de comportements en parallèle

```
P1 := a.'b.c.0
P2 := 'a.b.d.0
(P1 | P2) \ {a, b}
```

Modèles sémantiques (interpréter les comportements)

Sémantique opérationnelle

Relation de transition entre les comportements des processus.

$$Terme \rightarrow_{action} Terme$$

Sémantique axiomatique

On donne les propriétés algébriques des différents opérateurs

Conclusion à l'introduction aux alg. proc.

- De nombreuses autres algèbres processus : CSP, ACP, μCRL, LOTOS...
- Extensions avec gestion de contraintes de temps : TCSP, ...
- Peu utilisées dans les milieux industriels
- LOTOS normalisé, utilisé en industrie
- Des recherches toujours en cours...

Présentation de CCS (aspects sémantiques)

Calculus of **C**ommunicating **S**ystems, MILNER

Un **processus (ou agent)** modélise le comportement d'un système.

Elément de base de la modélisation : **actions**

→ alphabet d'actions

Action nulle : 0 (terminaison)

Un ensemble d'**opérateurs** (préfixage, composition parallèle, choix)

Modèle de communication

synchrone (entre les processus composés en parallèle)

Réalisé avec des actions de même nom : action, $\overline{\mathsf{action}}$
(complémentaires : entrée, sortie).
Evolution interne : τ (simultanément action et $\overline{\mathsf{action}}$)

Restriction dans la communication : (hiding)
Un ensemble d'actions (L) utilisé pour l'évolution interne.
$\rightarrow$ contrôler les synchronisations

Syntaxe abstraite

$$
\begin{array}{lcl}
E & ::= & 0 \\
 & | & a.E \\
 & | & E + E \\
 & | & E \mid E \\
 & | & E \setminus L \\
 & | & A \\
 & | & (E)
\end{array}
$$

CCS : Sémantique opérationnelle

Sémantique opérationnelle :

- **ensemble de règles d'inférence** définies par rapport à une syntaxe abstraite
- Chaque règle à une ou plusieurs prémisses et une seule conclusion

Règles d'inférence

$$\frac{Premisses}{Conclusion}$$

Définition de la sémantique $\Rightarrow$ un ensemble de règles pour chaque opérateur de la grammaire.

pref $\dfrac{}{\alpha.E \xrightarrow{\alpha} E}$

choixg $\dfrac{E \xrightarrow{\alpha} E'}{E+F \xrightarrow{\alpha} E'}$

choixd $\dfrac{F \xrightarrow{\alpha} F'}{E+F \xrightarrow{\alpha} F'}$

parallg $\dfrac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F}$

paralld $\dfrac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'}$

sync $\dfrac{E \xrightarrow{\alpha} E' \quad F \xrightarrow{\overline{\alpha}} F'}{E|F \xrightarrow{\tau} E'|F'}$

restr $\dfrac{E \xrightarrow{\alpha} E' \quad \alpha \notin L}{E\backslash L \xrightarrow{\alpha} E'\backslash L}$

Autres règles : parenthèsage, renommage, définition d'un agent par un autre

Spécification d'une coopération de processus

On veut spécifier la coopération entre 2 robots ouvriers pour l'usinage de pièces. Les 2 robots se servent d'un marteau ou d'un maillet. Les 2 robots ne peuvent pas se servir simultanément du même outil.

Utilisation de CCS de Milner

- Algèbre de processus avec passage de valeurs (très général).

- Nombre d'opérateurs réduit
- Sémantique opérationnelle

Spécification formelle en CCS

Quatre processus concurrents : 2 *Jobber* utilisent de *Ham* et *Mal*

$$Jobshop \widehat{=} (Jobber \mid Jobber \mid Ham \mid Mal)\backslash L$$

$$L = \{ \mathsf{geth}, \mathsf{puth}, \mathsf{getm}, \mathsf{putm}\}$$

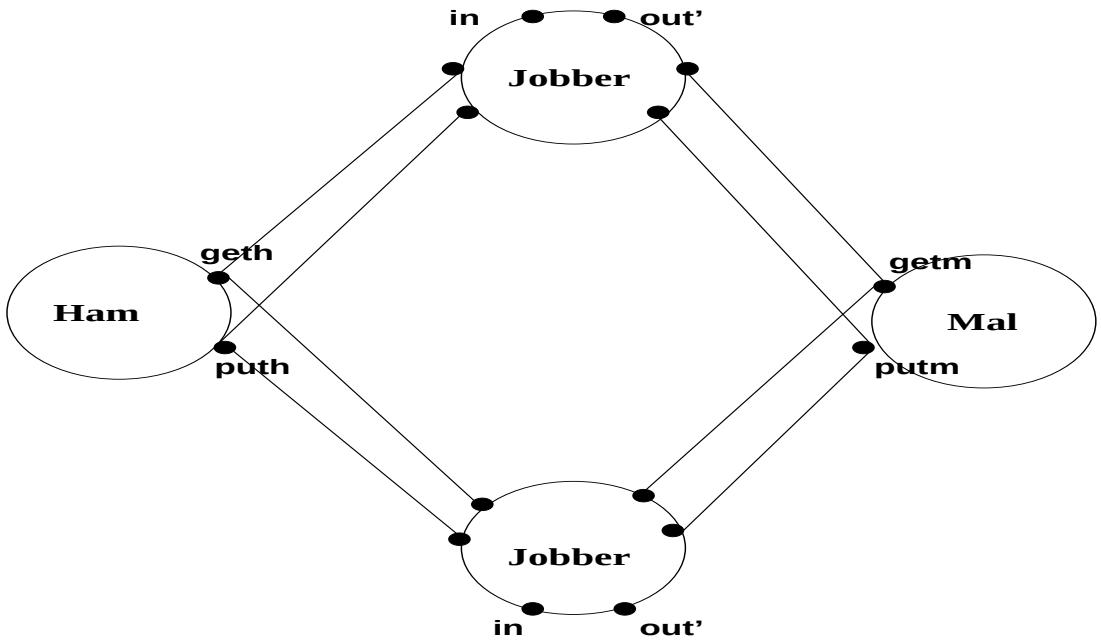


FIG. 8 – Une représentation du modèle (CCS)

Spécification formelle en CCS (suite)

$j : JOB$

$Ham \widehat{=} \mathsf{geth}.\mathsf{puth}.Ham$

$Mal \widehat{=} \mathsf{getm}.\mathsf{putm}.Mal$

$Jobber \widehat{=} \mathsf{in(j)}.Start(j)$

$Start(j) \widehat{=} \texttt{if}\ easy(j)\ \texttt{then}\ Finish(j)$
$\texttt{else if}\ hard(j)\ \texttt{then}\ Useh(j)$
$\texttt{else}\ Usetool(j)$

$Usetool(j) \widehat{=} Useh(j) + Usem(j)$

$Useh(j) \widehat{=} \mathsf{geth'}.Usingh(j)$

$Usingh(j) \widehat{=} \mathsf{puth'}.Finish(j)$

Spécification formelle en CCS (suite)

$Usem(j) \widehat{=} \mathsf{getm'}.Usingm(j)$

$Usingm(j) \widehat{=} \mathsf{putm'}.Finish(j)$

$Finish(j) \widehat{=} \mathsf{out'}(done(j)).Jobber$

$j : JOB$, $easy(j)$, $hard(j)$, $done(j)$ sont traitées comme des valeurs (exprimables dans un langage de données).

Exemple de spécifications en Z

Sort

$$in, out : \text{seq}\ Record$$

$$(\forall\, i, j : \text{dom}(out) \bullet i \leqslant j \Rightarrow out(i).key \leqslant out(j).key) \land items(in) = items(out)$$

avec *Record* un autre schéma offrant *key*,
items une fonction définie sur les séquences.

Record

$$key : \ldots$$
$$data : \ldots$$

$$(predicat)$$

Deuxième partie

Problématique de l'intégration de méthodes formelles

Motivation des recherches en intégration de méthodes formelles

☞ **Comment spécifier/développer** des systèmes complexes ?
(formellement sinon rigoureusement)

☞ **Trouver des méthodes formelles efficaces** pour spécifier et développer des systèmes complexes.

– **Système complexe** ⇒ pas seulement la taille, dévéloppement non trivial, plusieurs caractéristiques,

☛ Systèmes réels → plusieurs caractéristiques ;
Outils d'aujourdhui focalisent sur des caractéristiques précises

⇒ **inadéquation**. Recourir à l'intégration des méthodes.

Motivations (suite)

Caractéristiques ⇒ données, contrôle, réaction, concurrence, contraintes de temps, etc

- **Spécifier** ⇒ construire des modèles formels (mathématiques)
- **Développer** ⇒ construire correctement des systèmes à partir des modèles
  (besoin d'outils appropriés, ou réutilisation de l'existant)

Coin du voile sur les problèmes

Multiples concepts et modèles utilisés par les langages et méthodes de spécification

- Logique classique (tiers-exclu)
- Logique intuitionniste
- Logique modale
- Logique d'ordre supérieur
- Théorie des ensembles (avec ou sans axiomatisation)
- Théorie des types (+ logique classique ou intuitionniste)
- Algèbre universelle
- Théorie des catégories
- Théorie des automates (systèmes de transition, réseaux de Petri)
- ...

Coin du voile ...

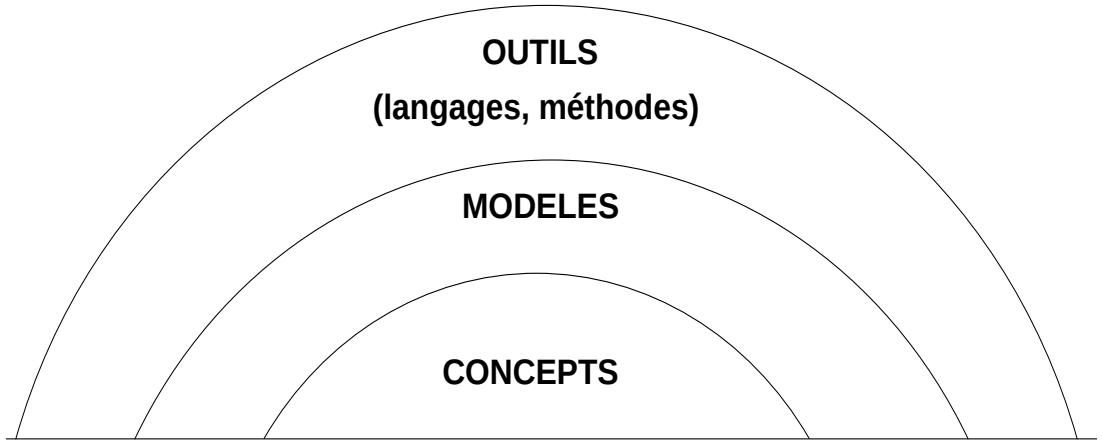


FIG. 9 – Concepts, Modèles, Outils

→ *Transfer Problem*

Objectifs de ce cours

☞ Sensibilisation aux problèmes posés par l'intégration de méthodes formelles

☞ Présentation des pistes de recherches

☞ Présentation de travaux et résultats actuels,

☞ Initiation à ce domaine de recherche

Un petit tour de l'état de l'art

- Approches hybrides
  - RAISE (Projet Esprit, 198X)
  - LOTOS (1987), μ-CRL, PSF
- Extensions d'une méthode par d'autres aspects
  - Z+CSP (1990), TLA+Z(1994), Z+CCS(1996), Timed-CSP, etc
  - TAA+RdP, CASL+CCS(2000)
- Intégration forte
  - Zave et Jackson, 1990+, multiformalisme, multiparadigme
  - Event Calculus (1992+)
  - Machines à configuration (1998+)

☞ Nouvelle conférence internationale, IFM (première en 1999)

Exemple de recherche

Approches courantes :

- Considérer une méthode/langage,
- Etant donné un cahier de charges,
- Spécifier et développer.

Approche en vue :

- Du cahier des charges sortir les caractéristiques du système,
- Elaborer l'environnement de développement approprié,
- Spécifier, développer.

Autres réponses ... systèmes complexes

- **Tests** (spécification, programmation puis mise à l'épreuve),
- **Programmation, vérification à postériori** (assistants de preuve)
- **Synthèse de programmes** à partir des spécifications,
- **Architecture de logiciels** (composants et composition),
- **Composition de modules** (spécification ou programmes),
- **AltaRica** (composition de systèmes de transition),
- **B Système** (développement correct jusqu'au code)
- ...

Troisième partie

Problèmes d'interaction sémantiques(logiques)

Rappel : caractéristiques des méthodes

Formalismes	Langage de description	Modèles sémantiques
VDM, Z	logique 1er ordre, Théorie des ensembles	Modèles à états
B	logique 1er ordre, Théorie des ensembles, AMN	Modèles, transformateur de prédicats, WP
Larch, OBJ, CASL	Signature, Logique, Catégories	Algebre universelle
TLA	Logique, logiques Tempor., Theo. des ens.	Modèles, Kripke
CCS, CSP, ACP	processus, events, channels	Traces, Bissimulation, Failure-divergence
Réseaux de Pétri	graphe	Systèmes de transition

→ **hétérogénéité des formalismes, théories et modèles.**

Quelques verrous à l'intégration de méthodes formelles

Le développement formel par l'approche intégrée n'est pas trivial.

→ problèmes :

- hétérogénéité syntaxique,
- hétérogénéité sémantique et,
- outillage (problèmes de la combinaison de plusieurs outils formels).

Ces problèmes doivent être :

→ clairement posés et

→ **résolus partiellement sinon totalement dans les environnements d'intégration de méthodes.**

Compatibilité entre méthodes

Lorsqu'on intègre deux ou plusieurs méthodes, **chacune implique ses caractéristiques, sa syntaxe, sa sémantique, ses outils.**

Les caractéristiques d'une méthode peuvent être soulignées par les facettes du système qu'elle permet de traiter.

La syntaxe fournit un support de description de modèles, tel un langage formel.
(*logiques, théorie des ensembles et types constituent les principales bases des formalismes*).

La sémantique permet de décrire **les modèles induits par les spécifications construites**.

Les modèles sémantiques courants :

- **modèles logiques (interprétations, kripke)**,
- **modèles à états**,
  systèmes de transition,
- **traces**,
- **algèbre**,
- etc

Intégration de méthodes → **compatibilité entre ces trois aspects** (syntaxe, sémantique, outils).

Quelques verrous

(Les paradigmes *données*, *processus*, *fonctions* sont transversaux)

- Considérant les **fondements théoriques des méthodes** → incompatibilités possibles !
  (Il peut y avoir des choix incompatibles dans leurs fondements.)
- Les bases des méthodes peuvent être si éloignées qu'elles ne soient pas *productives* ensemble.
- **Aucun point commun entre les fondements théoriques, pas de divergences (syntaxe et sémantique).**

- **Aucun point commun entre les fondements théoriques, avec des divergences (syntaxe et sémantique).**

- Plus généralement, leurs caractéristiques sont-elles compatibles ?

Examen des cas d'intégration

Considérons les principaux contextes d'intégration et utilisons la classification en :
Donnée (par Etat),
Donnée (par Opération) et
Processus (inclut Systèmes de Transition).

– Méthodes *model-oriented, property-oriented* → **Donnée**

– Méthodes *process-oriented* → **Processus**.

Nous n'incluons pas la caractéristique **Temps** mais les principes sont les mêmes.

Toutes les autres considérations peuvent être réorganisées autour de *Donnée, Operation et Process.*

3 Cas possibles (de combinaisons)

Donnée	Donnée
Process	Donnée
Process	Donnée

Ici intégration des méthodes 2 à 2, mais généralisable à plusieurs, selon le même principe.

Cas 1 - Méthodes Donnée vs Donnée

Des exemples pour illustrer chaque cas puis examen des problèmes syntaxiques et sémantiques possibles.

Etat vs Operation	:	Z ou B et Algébrique(ASL)
Etat vs Etat	:	Z et B
Operation vs Operation	:	Algébrique et Algébrique

Hétérogénéité Syntaxique (propositions)

Plusieurs possibilités :

– Une **méthode** peut être la **principale** et l'autre la **secondaire**.

structures du langage secondaire utilisées dans le premier.
structures du langage secondaire traduites dans le premier.

Verrous possibles : *compatibilité* entre les structures des deux langages.

– Aucune des deux méthodes n'est la principale
Une tierce méthode est nécessaire pour intégrer les deux.

Cette troisième méthode doit être une méthode cible pour les traductions des structures des deux autres.

Verrous : *problèmes de compatibilité* ;
Existe-t-il (toujours) cette troisième méthode ?
Est-il possible de traduire (sans perte) les structures des différentes méthodes ?

Compatibilité entre les constructions des méthodes(languages)

Pour préciser le problème, simplifions le en considérant les langages formels définis par leurs grammaires.

Soient deux languages L_1 et L_2 tels que :

- $L_1 = L(G_1)$ est le langage engendré par la grammaire G_1, et
- $L_2 = L(G_2)$ est le langage engendré par G_2,
- et soit $iL = L_1 \cap L_2$

Contraintes sur L_1, L_2	Observations	Pistes pour l'integration
$L_1 \cap L_2 = \emptyset$	pas d'ambiguités	augmentation, $L_1 \cup L_2$
$L_1 \cap L_2 \neq \emptyset$	Ambiguités Possibles	résoudre la comatibilité sur $L_1 \cap L_2$
$L_1 \cap L_2 = L_1$ *or* $L_1 \cap L_2 = L_2$	sous-ens., inclusion syntaxiq.	Pas de pbm d'integration sauf si des parties du sous-ens. ont des sémantiq. diff.
$\exists\ T \mid L_1 = T(L_2)$	T peut être des transformations syntaxiques (ou semantiq)	Définitions *Passerelle* (syntaxiq. ou semantiq.)

$L_1 \cap L_2 = \emptyset$ $\exists\ sL_1 \subseteq L_1, \exists\ sL_2 \subseteq L_2$ $\exists\ T \mid T(sL_1) = sL_2$	T peut être des transformations syntaxiques (ou sémantiq)	Definition *Passerelle* (syntaxiq. ou semantiq.)

TAB. 1 – Synthèse sur l'hétérogénéité syntaxique

$L_1 = T(L_2)$ représente le cas où un des deux langages est une transformation de l'autre.

Par rapport à cette transformation, il est possible de définir une passerelle entre les deux languages.

Autre cas : **sous-langages identiques à un renommage près.**

Hétérogénéité sémantique

Par rapport à la section précédente, il y aura des problèmes de *compatibilité sémantique.*

Les sémantiques ou généralement les fondements théoriques des méthodes peuvent être

– les **mêmes**,

– **complètement différentes mais cohérentes**,

– complètement différentes et **pas cohérentes**,

– **différentes mais avec des points communs.**

Contraintes sur $L1$, $L2$	contraintes sémantiques	Pistes de solutions
$iL = \emptyset$ Pas d'ambiguités	Semantiques incompatibles selon les modèles utilisés	OK si sémantiques compatibles. A étudier si modèles incompatibles
$iL \neq \emptyset$ Ambiguités Possibles	$Sem_{L_1}(iL) = Sem_{L_2}(iL)$	OK. Ils sont compatibles
$iL \neq \emptyset$ Ambiguités Possibles	$Sem_{L_1}(iL) \neq Sem_{L_2}(iL)$	Incompatibles. Ambiguités
$iL = L_1 = L_2$		Pas de problèmes
$L_1 = T(L_2)$		Compatible. Définition de 'Semantic bridge'

Sem_{L_i} : sémantique par rapport au langage L_i.

Aspects *outillage (raisonnement formel)*

La compatibilité peut être **étendue aux outils de raisonnement formel et de développement (vérification, simulation, raffinement, synthèse, etc)**

L'état de l'art montre que pour certains besoins les **logiques temporelles sont très utilisées et assez efficaces. (vérification de propriétés,** *liveness*)
Elles sont intégrées à plusieurs approches.

Les outils fondés sur **les logiques d'ordre supérieur** promettent aussi de bons résultats (HOL, Coq, PVS, etc).
(vérification de propriétés)

→ Le *Transfer Problem*

Le raisonnement formel dans un cadre théorique n'est pas toujours valable dans un autre cadre théorique.

Une méthode et ses outils sont basés sur un système de raisonnement.

Une autre méthode et ses outils sont basés sur un autre système de raisonnement.

Cas 2 - Donnée vs Process

Etat vs Process	:	Z ou B **et** CCS ou CSP ou RdP
	:	Z ou B **et** Systèmes de trans.
Operation vs Process	:	Algébriq. **et** CSP ou CCS ou RdP
	:	Algébriq. **et** Systèmes de trans.

Syntaxe intégrée

Une méthode principale, une méthode secondaire.

Par exemple une algèbre de processus comme principale et une méthode orientée donnée comme secondaire.
Approche efficace pour le passage de valeurs aux algèbres de processus.

La méthode **orientée donnée** (*la principale*) et une **algèbre de processus** (*la secondaire*)

Par exemple, on décrit un type de données et on utilise ces opérations comme alphabet des processus.

Le problème de l'hétérogénéité syntaxique n'est pas très difficile ici.
Le problème de l'expressivité démeure.

Sémantique intégrée

En général **un modèle sémantique est fixé** et des traductions effectuées. La compatibilité sémantique reste à étudier en détails comme problème.

– Sémantique algèbre de processus + . . .

– Sémantique langage de données + . . .

→ Technique de plongement **Embedding**

Cas 3 - Process vs Process

L'approche orientée processus est plus 'homogène' que les autres.

En effet, CSP (Hoare), CCS (Milner) et ACP (Bergstra) sont les principales méthodes sur lesquelles s'appuient toutes les autres.

Les bases théoriques sont déjà les mêmes.

Les différences entre algèbres de processus sont le plus souvent liées :

- **à la puissance d'expression**,
- aux **modèles de communication** utilisés,
- au **traitement du temps**,
- au **traitement des données**.

Process vs Process	:	CSP and CCS or RdP
	:	CCS, CSP and Trans. systems

Exemple de LOTOS (combine CCS, CSP)

Syntaxe et Sémantique

Du point de vue syntaxique, les algèbres de processus ne sont pas très différentes : elles proposent les mêmes opérateurs.

$\rightarrow$ moins de problèmes de compatibilité

Du point de vue sémantique, **la sémantique des traces** semble être un bon liant des différentes algèbres.

Utilisation de lois algébriques (CSP) et règles sémantiques (CCS)

Autres modèles sémantiques :

– Failure-divergence (CSP),
– bissimulation, equivalence observationnelle (CCS),
– π-calcul.

Outillage

Utilisation du modèle sémantique commun : systèmes de transition

Techniques d'intégration de méthodes

– Plongement syntaxique ou sémantique (Embedding Techniques)
– Combinaison de systèmes (de preuve)

Spécification dans différents formalismes,
plongement dans une logique pour assurer une homogénéité dans l'analyse

Compatibilité syntaxique

→ L'intégration de méthodes nécessite la définition de domaines de compatibilité syntaxique

Exemple : tous les formalismes utilisant la même logique pour décrire leurs constructions sont dans le même domaine syntaxique.

Logique 1er ordre, Théorie des ensembles, etc

Compatibilité sémantique

→ Définition de domaines de compatibilité sémantique par rapport aux modèles sémantiques.

Conséquence : Deux ou plusieurs méthodes considérées dans un domaine, sont compatibles et peuvent être intégrées simplement.

Exemple : Deux méthodes utilisant les systèmes de transition.

Compatibilité entre méthodes (outils)

Idées : théorie du raffinement (données, opérations).

→ **refinement-compatibility**

→ **Génération d'environnements de développement**

Passerelles entre spécifications

→ Mise en œuvre de l'intégration

Quand deux méthodes sont compatibles, on peut définir une passerelle entre elles.

Le concept

Méthode formelle → langage de spécification.

Intégration de méthodes → passerelle entre langages de spécification formelle

A partir d'une spécification dans un langage, on peut obtenir une spécification équivalente dans l'autre langage (deep/shallow embedding).

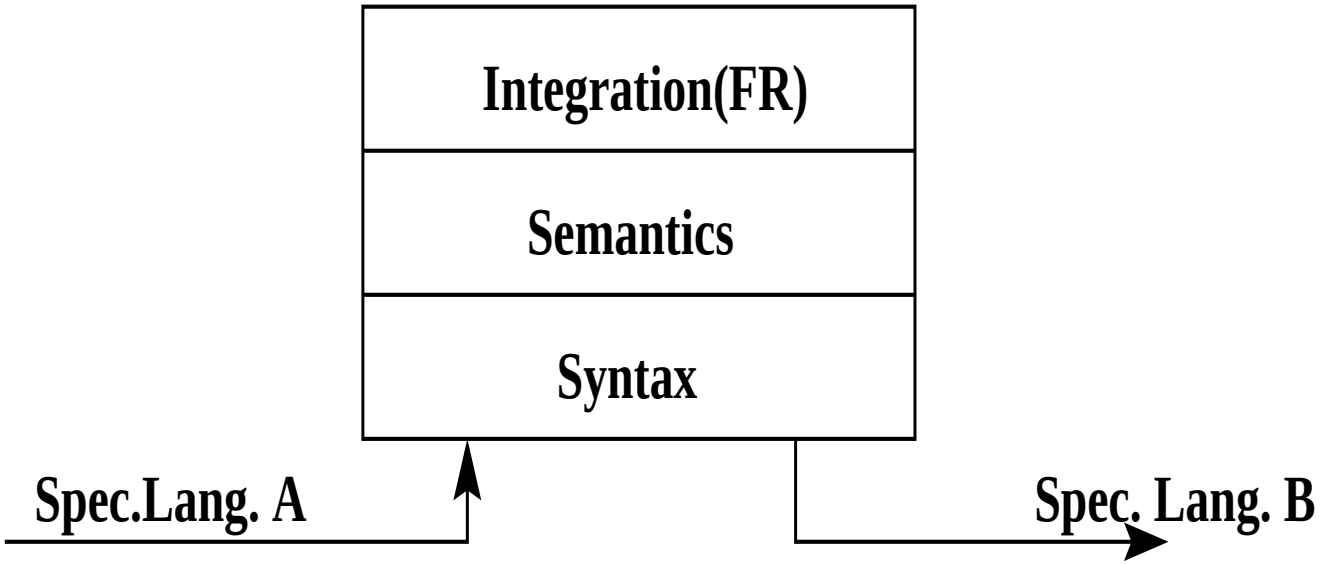


FIG. 10 – Passerelle entre spécifications

Niveau syntaxique

Une spécification dans un language donné peut être :

– **traduite en une spécification** dans l'autre langage,

– **encapsulée par une spécification** dans l'autre langage.

Exemple : une spécification Z traduite en une spécification dans un langage imaginaire HL qui encapsule les constructions de Z.

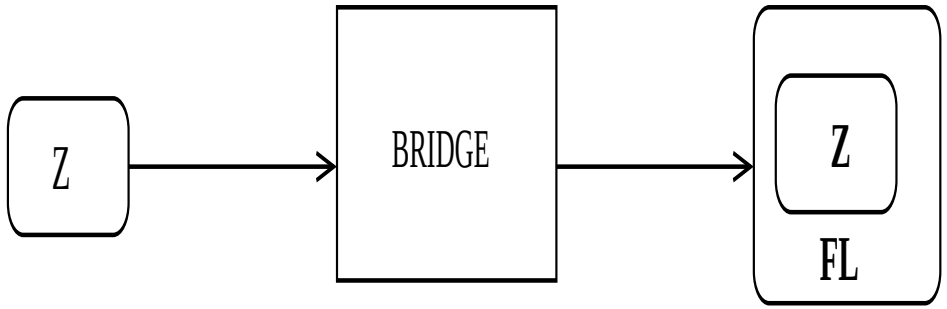


FIG. 11 – Z et HL

Niveau sémantique

Un des langages est utilisé pour exprimer la sémantique de l'autre.

Exemple : *HOL*(Gordon) pour exprimer la sémantique de *Z* ou *HOL* pour exprimer la sémantique de *CCS*. *B* en PVS ou Coq

La passerelle a la forme générale suivante :

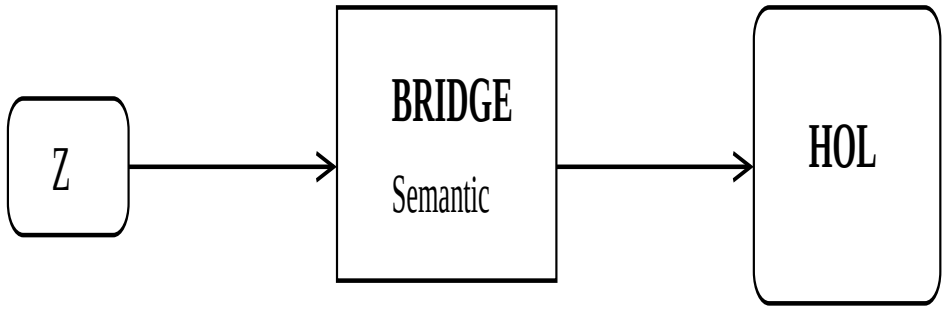


FIG. 12 – Z en HOL

Généralisation pour l'approche d'intégration

Intégration de méthodes → logiques et théories d'ordre supérieur, dans les passerelles entre méthodes.

Au niveau syntaxique, HOL ou PVS peuvent être utilisés comme langage de haut niveau pour encapsuler (selon les contraintes de compatibilité) un ou plusieurs langages pris comme langage de bas niveau.

Dans cette approche, les constructions des langages de bas niveau doivent être traduites/encapsulées dans le langage de haut niveau.

Au niveau sémantique, l'approche peut être :

- choix de langages ou de théories (higher order logics, type theory, category theory, etc) comme langages de haut-niveau,
- description de chaque spécification de bas niveau dans le langage de haut niveau,
- raisonnement dans le langage de haut niveau

Synthèse

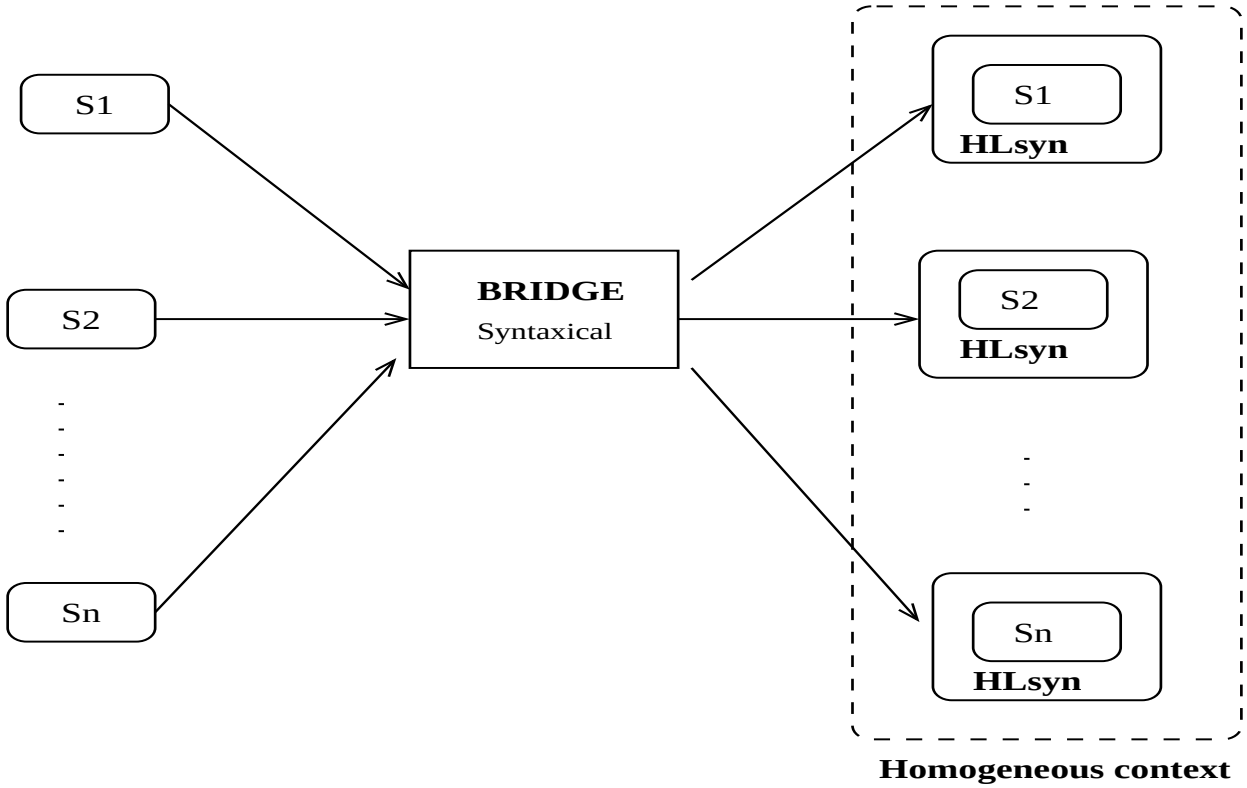


FIG. 13 – Généralisation : niveau syntaxique

L'intégration et le raisonnement formel peuvent être faits dans un tiers contexte.

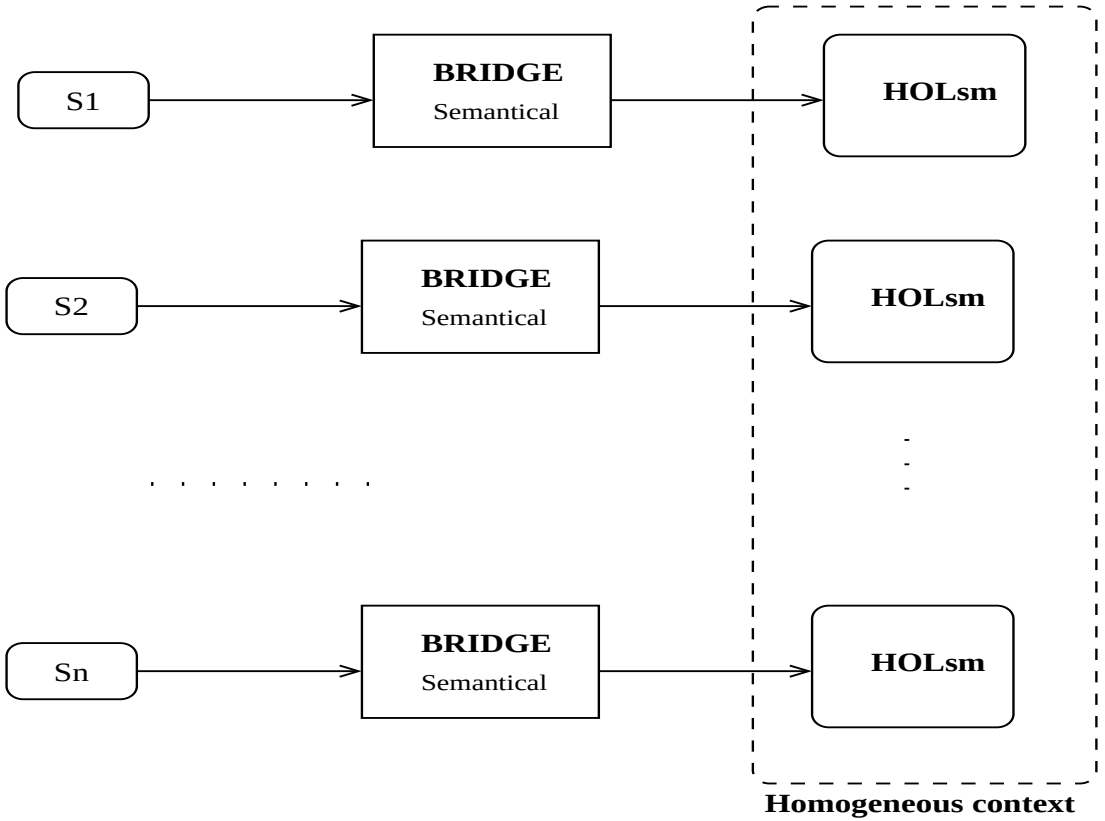


FIG. 14 – Généralisation : niveau sémantique

Un exemple de base d'intégration peut être :

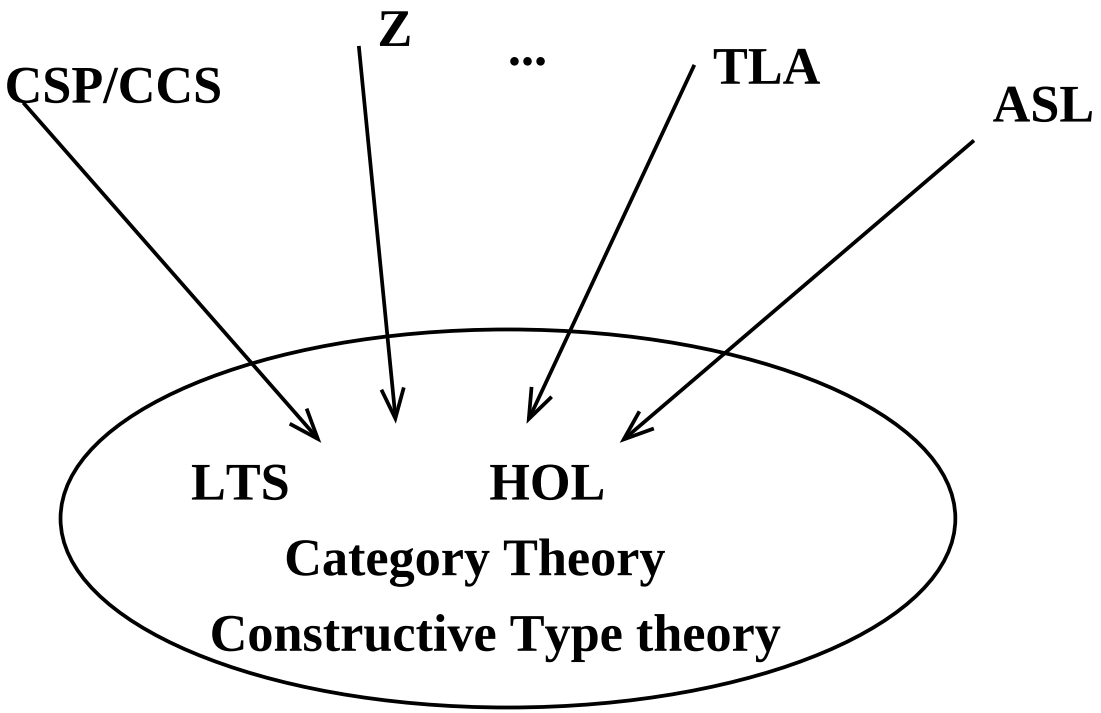


FIG. 15 – Base d'intégration

Plongement sémantique (Semantic Embedding)

– Illustration de la technique de plongement

– A partir d'un cours de
M. Gordon, *Teaching Hardware and Software Verification in a Uniform Framework*

Logique d'ordre supérieur (H.O.L.)

– Extension de la logique du premier ordre (F.O.L.) en autorisant des **variables qui sont des relations et des fonctions**

– Utilisation de **types** pour préserver la cohérence (consistency) comme dans leur utilisation première (Russel)

Notation HOL

notation du calcul des prédicats	
Notation	Signification
`T`	vérité (Truth)
`F`	fausseté (Falsity)
$P(x)$ (ou $P\,x$)	x a la propriété P
$\neg\ t$	$non\ t$
$t_1 \vee t_2$	t_1 ou t_2
$t_1 \wedge t_2$	t_1 et t_2

Notation HOL (suite)

notation du calcul des prédicats	
Notation	Signification
$t_1 \Rightarrow t_2$	t_1 implique t_2
$t_1 \sim t_2$	t_1 si et seulement si t_2
$t_1 = t_2$	t_1 égale t_2
$\forall\, x.t[x]$	quel que soit x, $t[x]$
$\exists\, x.t[x]$	il existe un x pour lequel $t[x]$
$(t \rightarrow t_1 \mid t_2)$	si t alors t_1 sinon t_2

Notation HOL (suite)

Types

- Les types sont **atomiques** ou **composés**.

- **Exemples de types atomiques :** *bool*, *int*, *num*, *real*

- Les types composés sont construits à partir de types atomiques ou des types composés en utilisant des opérateurs de types.
- **Exemples de types composés :**
  - Si σ, σ_1 et σ_2 sont des types
  - alors σ *list*, $\sigma_1 \rightarrow \sigma_2$, $\sigma_1 \times \sigma_2$, sont des types
  - *list* est un opérateur unaire, $\rightarrow$ et $\times$ sont binaires infixés.
  - $t : \sigma$ signifie t a le type σ

Plongement de L_{prop} dans HOL

Pour illustrer le plongement sémantique, considérons le langage propositionnel L_{prop} :

$$
\begin{array}{lcl}
\text{wff} & ::= & \text{True} \\
& | & \text{N wff} \\
& | & \text{C wff wff} \\
& | & \text{D wff wff}
\end{array}
$$

Approche 1 : *deep embedding (plongement profond)*

Elle consiste à représenter les wff dans la logique hôte (ici HOL) par des valeurs d'un certain type, soit *wff*, et puis à définir dans la logique hôte une fonction sémantique, soit $\mathcal{M}$, récursivement :

$$\begin{aligned}
\mathcal{M}(\mathit{True}) &= \mathtt{T} \\
\mathcal{M}(\mathit{Nw}) &= \neg\, \mathcal{M}(w) \\
\mathcal{M}(C\ w_1\ w_2) &= \mathcal{M}(w_1)\ \wedge\ \mathcal{M}(w_2) \\
\mathcal{M}(D\ w_1\ w_2) &= \mathcal{M}(w_1)\ \vee\ \mathcal{M}(w_2)
\end{aligned}$$

avec $\mathcal{M}$ une constante de type d'ordre supérieur $\mathit{wff} \longrightarrow \mathit{bool}$

Approche 2 : *shallow embedding (plongement superficiel)*

consiste à fixer des notations conventionnelles pour traduire les wffs en des termes de la logique hôte.

Supposons que $[\![w]\!]$ est la traduction de w dans la HOL.

L'opération $w \mapsto [\![w]\!]$ n'est pas définie dans la logique hôte, mais correspond à un sucre syntaxique de niveau méta (tel un macro).

La traduction de w en logique est définie récursivement par :

$$[\![True]\!] \equiv T$$
$$[\![N\ w]\!] \equiv \text{"}\neg\text{ "} \frown [\![w]\!]$$
$$[\![C\ w_1\ w_2]\!] \equiv [\![w_1]\!] \frown \text{ " } \wedge \text{ " } \frown [\![w_2]\!]$$
$$[\![D\ w_1\ w_2]\!] \equiv [\![w_1]\!] \frown \text{ " } \vee \text{ " } \frown [\![w_2]\!]$$

- Avec le *shallow embedding*, seuls les théorèmes exprimés dans le langage plongé sont prouvables.
  Dans l'exemple, la quantification sur les wff n'est pas exprimable.
  Il y a moins de chose dans la logique et le plongement demande moins de travail sur la logique, il est donc plus souvent facile de traiter des notations complexes.

- L'avantage du *plongement profond* est que les théorèmes concernant le langage plongé peuvent être prouvés.

  Par exemple

$$\mathcal{M}(C\ w_1\ w_2) = \mathcal{M}(N\ D\ N\ w_1\ N\ w_2)$$

Shallow embeddind de la logique de Floyd Hoare

Soit le langage des commandes engendré par la grammaire suivante :

$$
\begin{array}{lcl}
E & ::= & N \mid E_1 + E_2 \mid E_1 - E_2 \mid E_1 \times E_2 \mid \cdots \\
B & ::= & E_1 = E_2 \mid E_1 \leqslant E_2 \mid \cdots \\
C & ::= & \texttt{SKIP} \\
 & \mid & V := E \\
 & \mid & C_1 \,;\, C_2 \\
 & \mid & \texttt{IF } B \texttt{ THEN } C_1 \texttt{ ELSE } C_2 \\
 & \mid & \texttt{WHILE } B \texttt{ DO } C
\end{array}
$$

Exercice

On veut plonger ce langage dans une logique d'ordre supérieur. On veut également plonger la logique de Hoare (règles de spécification de correction partielle) dans une logique dórdre supérieur.

Démarche

Pour le langage, plonger toutes les constructions dans les constructions de la logique hôte avec leur sémantique. (donc il faudra exprimer la sémantique des commandes du langage).

Rappels de logique (de Hoare)

(triplet de Hoare exprime la spécification de correction partielle)

Si P est une formule de la logique des prédicats, $\vdash P$ signifie que P peut être déduit des lois de la logique et de l'arithmétique.

$\vdash \{P\}C\{Q\}$ signifie que $\{P\}C\{Q\}$ est

- soit une instance des schémas d'axiomes A1, A2 (suivants)
- soit déductible par une séquence d'application des règles R_i de telles instances.

Axiomes et règles de la logique de Hoare

A1 : Axiome du `SKIP`. Pour toute formule P

$$\vdash \{P\}\,\texttt{SKIP}\,\{P\}$$

A1 : Axiome de l'affectation. P formule, V variable de programme, E expression (entière)

$$\vdash \{P[E/V]\}\ V := E\ \{P\}$$

($P[E/V]$ dénote le résultat de la substitution de E aux occurrences libres de V dans P.)

R1 : règle de la précondition (renforcement)

$$\frac{\vdash P' \Rightarrow P \quad \vdash \{P\}\, C\, \{Q\}}{\vdash \{P'\}\, C\, \{Q\}}$$

R2 : règle de la postcondition (affaiblissement)

$$\frac{\vdash \{P\}\, C\, \{Q\} \quad \vdash Q \Rightarrow Q'}{\vdash \{P\}\, C\, \{Q'\}}$$

R3 : règle de la séquence

$$\frac{\vdash \{P\}\, C_1\, \{Q\} \quad \vdash \{Q\}\, C_2\, \{R\}}{\vdash \{P\}\, C_1 ; C_2\, \{R\}}$$

R4 : règle du `IF`

$$\frac{\vdash \{P \wedge B\}\, C_1\, \{Q\} \quad \vdash \{P \wedge \neg\, B\}\, C_2\, \{Q\}}{\vdash \{P\}\ \texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2\ \{Q\}}$$

R5 : règle du `WHILE`

$$\frac{\vdash \{P \wedge B\}\, C_1\, \{Q\}}{\vdash \{P\}\ \texttt{WHILE}\ B\ \texttt{DO}\ C_1\ \{Q \wedge \neg\, B\}}$$

Sémantique des commandes

→ *Une commande dénote une fonction.*

La dénotation traditionnelle d'une commande C est une fonction, $\mathsf{Meaning}(C)$, sur les états de machine. L'idée est :

$\mathsf{Meaning}(C)(s)$ ="l'état résultant de l'exécution de C dans l'état s"

Note : Attention, les fonctions dénotées par les commandes sont partielles ; par exemple une boucle peut ne pas se terminer. Puisque les fonctions en calcul des prédicats sont totales, elles ne peuvent être utilisées comme dénotation des commandes. On utilisera plutôt, pour la sémantique des commandes, des prédicats sur des paires d'états $(s_1,\ s_2)$;

L'idée est que si C(programme) dénote c (logique) alors :

$$c(s_1, s_2) \equiv (\mathsf{Meaning}(C)(s_1) = s_2)$$

$$c(s_1, s_2) \equiv (\mathsf{Meaning}(C)(s_1) = s_2)$$

veut dire donc

$$c(s_1, s_2) = \begin{cases} \mathtt{T} & \text{si l'exécution de } C \text{ dans l'état } s_1 \text{ donne } s_2 \\ \mathtt{F} & \text{autrement} \end{cases}$$

Exemple :

Si c_{while} est le prédicat dénoté par `WHILE T DO` C, alors :

$$\forall\, s_1\ s_2 . \mathsf{c}_{while}(s_1, s_2) = \mathtt{F}$$

(Pour bien manipuler les états, on va les définir proprement)

Le type des états : *state*

$state = string \rightarrow num$

XYZ représente la chaîne constituée des caractères X, Y, et Z.
XYZ a donc le type *string*.

Un état s dans lequel les chaînes X, Y, Z sont liées respectivement à 1, 2, 3 et les autres chaînes à 0, est défini par :

$$s = \lambda\, x.(x =' X' \rightarrow 1 \mid (x =' Y' \rightarrow 2 \mid (x =' Z' \rightarrow 3 \mid 0)))$$

Si e, b et c sont les dénotations des commandes E, B, et C, alors

$$e : state \rightarrow num$$

$$b : state \rightarrow bool$$

$$c : state \times state \rightarrow bool$$

Exemples :

La dénotation de X+1 est $\lambda\, s.s'X' + 1$

La dénotation de (X+Y)>10 est $\lambda\, s.(s'X' + s'Y') > 10$

On introduit (pour convenance) les notations $[\![E]\!]$ et $[\![B]\!]$ pour les termes logiques représentant les dénotations de E et B.

Par exemple :

$$\begin{aligned} [\![X + 1]\!] &= \lambda\, s.s'X' + 1 \\ [\![(X + Y) > 10]\!] &= \lambda\, s.(s'X' + s'Y') > 10 \end{aligned}$$

Plongement des commandes

Les prédicats en HOL qui correspondent aux 5 types de commandes sont définis. Pour chaque commande C, un terme $[\![C]\!]$ de type $state \times state \rightarrow bool$ est défini comme suit :

1. $[\![\texttt{SKIP}]\!] = \mathsf{Skip}$
   avec la constante Skip définie par : $\mathsf{Skip}(s_1, s_2) = (s_1 = s_2)$

2. $[\![V := E]\!] = \mathsf{Assign}(\text{'V'}, [\![E]\!])$
   avec la constante Assign définie par :
   $\mathsf{Assign}(v, e)(s_1, s_2) = (s_2 = \mathsf{Bnd}(e, v, s_1))$
   où $\mathsf{Bnd}(e, v, s) = \lambda\, x.(x = v \rightarrow e\ s \mid s\ x)$

3. $[\![C_1\,;C_2]\!] = \mathsf{Seq}([\![C_1]\!],[\![C_2]\!])$
   avec la constante Seq définie par :
   $\mathsf{Seq}(c_1,c_2)(s_1,s_2) = \exists\, s.c_1(s_1,s) \wedge c_2(s,s_2)$

4. $[\![\texttt{IF}\ B\ \texttt{THEN}\ C_1\ \texttt{ELSE}\ C_2]\!] = \mathsf{If}([\![B]\!],[\![C_1]\!],[\![C_2]\!])$
   avec la constante If définie par :
   $\mathsf{If}(b,c_1,c_2)(s_1,s_2) = (b\ s_1 \rightarrow c_1(s_1,s_2) \mid c_2(s_1,s_2))$

5. $[\![\texttt{WHILE}\ B\ \texttt{DO}\ C]\!] = \mathsf{While}([\![B]\!],[\![C]\!])$
   avec la constante While définie par :
   $\mathsf{While}(b,c)(s_1,s_2) = \exists\, n.\mathsf{Iter}(n)(b,c)(s_1,s_2)$
   où Iter est définie par :

$$\begin{aligned}
\mathsf{Iter}(0)(b,c)(s_1,s_2) &= \texttt{F}\\
\mathsf{Iter}(n+1)(b,c)(s_1,s_2) &= \mathsf{If}(b,\mathsf{Seq}(c,\mathsf{Iter}(n)(b,c)),\mathsf{Skip})(s_1,s_2)
\end{aligned}$$

Plongement des spécifications de correction partielle

Une spécification de correction partielle $\{\mathrm{P}\}\quad \mathrm{C}\quad \{\mathrm{Q}\}$ dénote :

$$\forall\, s_1\ s_2.[\![P]\!]s_1 \wedge [\![C]\!](s_1, s_2) \Rightarrow [\![Q]\!]s_2$$

On peut définir une constante Spec pour abréger cette formule :

$\mathsf{Spec}(p, c, q) = \forall\, s_1\ s_2.p\ s_1 \wedge c(s_1, s_2) \Rightarrow q\ s_2$

Exemple :

$$\{X = 1\}\ \ X := X + 1\ \ \{X = 2\}$$

dénote $\mathsf{Spec}([\![X = 1]\!], \mathsf{Assign}('X', [\![X + 1]\!]), [\![X = 2]\!])$

Exercice

Exprimer par plongement, la sémantique de la spécification suivante (où x et y sont des variables logiques, et X, Y des variables de programme) :

$$\{X = x \wedge Y = y\}\ Z := X\,;X := Y\,;Y := Z\ \{X = y \wedge Y = x\}$$

Généralisation : intégration de méthodes

– Choisir/définir une logique hôte,

– Plonger les différentes méthodes/formalismes dans cette logique,

– Raisonner (développement formel) dans le cadre de cette logique.

Solution

$$\begin{array}{l}\mathsf{Spec}([\![X = x \wedge Y = y]\!], \\ \quad \mathsf{Seq}(\mathsf{Assign}('Z', [\![X]\!]), \\ \quad\quad \mathsf{Seq}(\mathsf{Assign}('X', [\![Y]\!]), \mathsf{Assign}('Y', [\![Z]\!]))), \\ \quad [\![X = y \wedge Y = x]\!])\end{array}$$

Quatrième partie

Application aux composants

Modélisation de composants logiciels

Modèle Kmelia

Un composant est caractérisé par :

son nom,

son espace d'état,

son interface (une abstraction de ses services et de son comportement),

une initialisation,

ses services (offerts et requis),

ses propriétés (invariant établissant les conditions requises pour une utilisation et un fonctionnement corrects du composant) et

son comportement (conditions d'utilisation et d'enchaînement de ses services).

Application aux composants

Un composant C par un 6-uplet $\langle \mathcal{E}, Init, \mathcal{A}, \mathcal{D}_S, I, \mathcal{C}_S \rangle$ avec :

- $\mathcal{E} = \langle T, V, V_T, Inv \rangle$ l'espace d'états où T est un ensemble de types prédéfinis, V un ensemble de variables, V_T un ensemble de variables typées, $V_T \subseteq V \times T$ et Inv l'invariant d'état ;
- $Init$ l'initialisation des variables de V_T ;
- $\mathcal{A}$ un ensemble fini d'actions internes ;
- I l'interface du composant : un ensemble fini de noms de services, offerts ou requis, visibles par l'environnement du composant. I_p et I_r forment une partition : $((I = I_p \cup I_r) \wedge (I_p \cap I_r = \emptyset))$;

– $\mathcal{D}_S$ est une relation de description des services ;
le domaine de la relation peut contenir, en plus des noms de services (I_p et I_r) de l'interface I, des noms de services offerts sous des conditions particulières définies plus loin.
Les descriptions des services offerts ($\mathcal{D}_{S_p}$) et des services requis ($\mathcal{D}_{S_r}$) forment une partition de $\mathcal{D}_S$ c'est à dire :
$\mathcal{D}_S = (\mathcal{D}_{S_p} \cup \mathcal{D}_{S_r}) \wedge (\mathcal{D}_{S_p} \cap \mathcal{D}_{S_r}) = \emptyset$.
De plus il y a une projection de la partition de I sur le domaine de $\mathcal{D}_S$:
$I_p \subseteq \mathrm{dom}(\mathcal{D}_{S_p}) \wedge I_r \subseteq \mathrm{dom}(\mathcal{D}_{S_r}) \wedge dom(\mathcal{D}_{S_p}) \cap dom(\mathcal{D}_{S_r}) = \emptyset$

– $\mathcal{C}_S$ un ensemble de contraintes portant sur les services offerts du composant ($\mathcal{C}_{S_p}$) et sur les services requis de l'environnement du composant ($\mathcal{C}_{S_r}$). Ces contraintes portent sur la composabilité des composants.

Application aux composants

Service

Un service s est défini par un couple $(I_s,\ \mathcal{B}_s)$ où :
I_s est l'interface et $\mathcal{B}_s$ est un éventuel comportement dynamique (les services requis n'en n'ont pas).

$\mathcal{B}_s$, le comportement d'un service s est un système de transitions étiquetées (ou LTS) spécifié par un 6-uplet $\langle S, L, \delta, \Phi, S_0, S_F \rangle$ où
S est l'ensemble des états de $\mathcal{B}_s$,
$S_0 \in S$ est l'état initial,
$S_F \subset S$ est l'ensemble non vide des états finaux,
L est l'ensemble des étiquettes des transitions.
$\delta : S * L \rightarrow S$ est la relation de transition.
$\Phi : S \leftrightarrow sub_s$ est la relation d'étiquetage des états.

Application aux composants

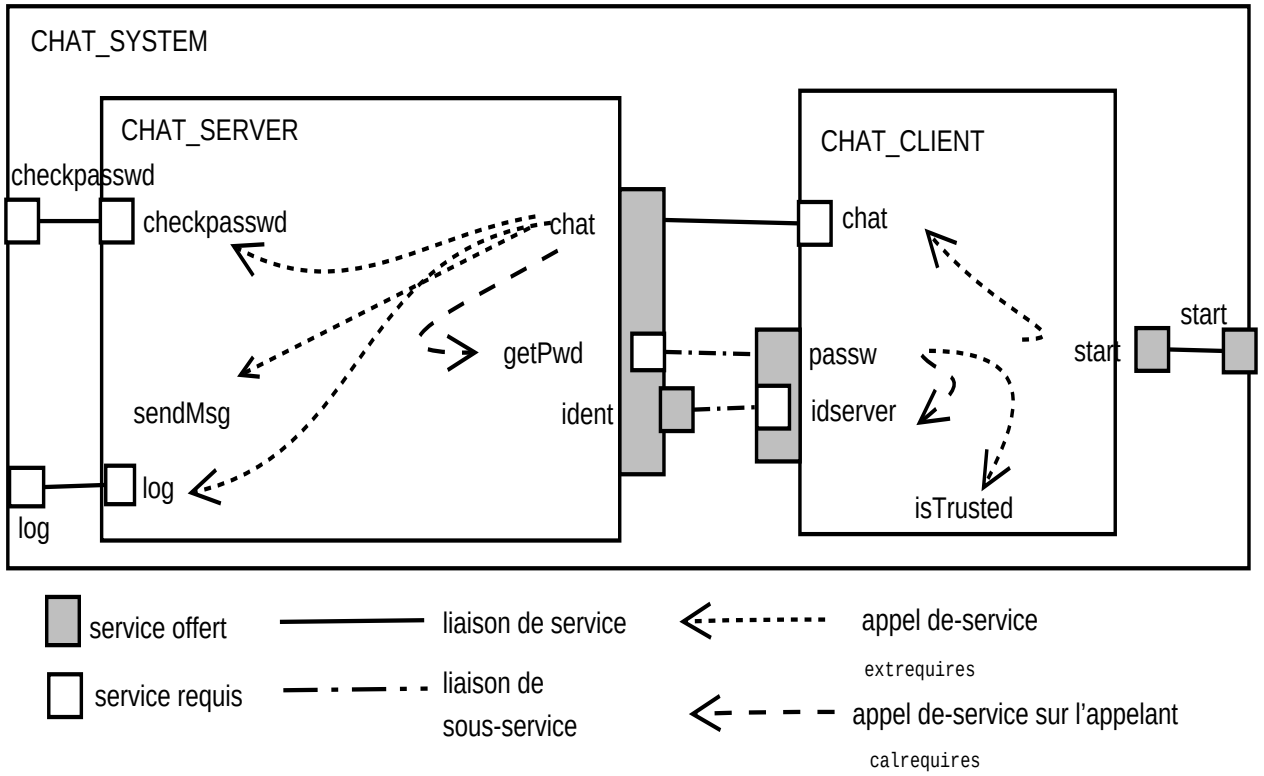
Assemblage de composants

Un assemblage est un ensemble de composants liés par leurs services, qui coopèrent pour la construction d'une architecture logicielle.

Les *liaisons* entre composants abstraits sont les liens entre les services requis par les uns et les services offerts par les autres.

Un *lien* établit un canal implicite entre un service requis et un service offert.

Application aux composants



Application aux composants

Un assemblage de composants est une relation *assemble* définie sur un ensemble de composants.

Elle est définie à partir des relations *link* et *hiddenlink*.

Soit $\mathcal{C}$ un ensemble de composants C_k avec $k \in 1..n$;

un assemblage de composants est une relation $assemble : \mathcal{C} \times \mathcal{C}$ telle que

pour deux composants C_i et C_j avec $i \in 1..n \wedge j \in 1..n \wedge i \neq j$ et avec

$C_i = \langle\langle T_i, V_i, V_{T\,i}, Inv_i\rangle, Init_i, \mathcal{A}_i, \langle I_{p\,i}, I_{r\,i}\rangle, \langle \mathcal{D}_{S_p\,i} \cup \mathcal{D}_{S_r\,i}\rangle, \mathcal{C}_{S\,i}\rangle$ et

$C_j = \langle\langle T_j, V_j, V_{T\,j}, Inv_j\rangle, Init_j, \mathcal{A}_j, \langle I_{p_j}, I_{r\,k}\rangle, \langle \mathcal{D}_{S_p\,j} \cup \mathcal{D}_{S_r\,j}\rangle, \mathcal{C}_{S\,j}\rangle$

a) les services requis de C_i sont liés à des services offerts de C_j et vice-versa :

$$\forall (C_i, C_j) \in assemble.$$
$$\{(s_i, s_j) \in link \mid ((s_i \in I_{p\,i} \wedge s_j \in I_{r\,j}) \vee (s_i \in I_{r\,i} \wedge s_j \in I_{p\,j}))\} \neq \emptyset$$

Dans la suite $\mathsf{compLink}(assemble, Ci, Cj)$ dénote une fonction auxiliaire qui, étant donné deux composants C_i et C_j d'un assemblage, fournit les liens entre C_i et C_j :
$\mathsf{compLink}(assemble, Ci, Cj) = \{(s_i, s_j) \in link \mid ((s_i \in I_{p\,i} \wedge s_j \in I_{r\,j}) \vee (s_i \in I_{r\,i} \wedge s_j \in I_{p\,j}))\}$ avec $(C_i, C_j) \in assemble$.

b) de plus certains liens entre les services s_i et s_j (des composants C_i et C_j) peuvent être caractérisés par des sous-liens (définis avec la relation *hiddenlink*) :

$$sublink : link \leftrightarrow hiddenlink$$

avec les contraintes *c1*, *c2* suivantes :

c1) Les sous-liens ne sont définis que pour des liens de services issus de composants assemblés.

$$\forall (s_i, s_j) \in \mathsf{dom}(sublink) \,.\; \exists (C_i, C_j) \in \mathsf{dom}(assemble) \mid (s_i, s_j) \in \mathsf{compLink}(assemble, Ci, Cj)$$

c2) Les sous-liens d'un lien (s_i, s_j) sont définis entre :

les requis de s_i $(cals_i)$ et n'importe quel offert de C_j (I_{pj}),
les offerts de s_i $(subs_i)$ et les requis de s_j $(cals_j)$,
les requis de s_i $(cals_i)$ et les offerts de s_j $(subs_j)$,
les offerts de s_i $(subs_i)$ et les requis de C_k (I_{rk}) pour un k quelconque.

On a ainsi la contrainte :

$$sublink(s_i \mapsto s_j) \subseteq (cal_{si} \times I_{pj}) \cup (sub_{si} \times cal_{sj}) \cup (cal_{si} \times sub_{sj}) \cup (sub_s \times I_{rk})$$

Conclusion

Il y a plein de choses intéressantes à faire.

Les outils fondamentaux sont là.

Il faut y aller !