Développer des agents

partie I: Génie logiciel orienté agent

Stéphane Airiau

Université Paris-Dauphine

Plan

- Génie logiciel orienté agent
  - méthodologie pour développer un système multiagents
- Programmation Orientée Agent
  - exemple des agents BDI (Belief Desire Intentions)

Matériel utilisé pour préparer cette leçon :

- chapître 15 de **Multiagent Systems**, édité par Gerhard Weiss, MIT Press, 2012
  « Agent Oriented Software Engineering » M. Winikoff et L. Padgham
- **Developing Intelligent Agent Systems**, M. Winikoff et L. Padgham, Wiley, 2004

Concevoir un système multiagent

- Quels agents utiliser ? Qui sont les agents ?

  joueurs, coordinateurs, managers, etc

  ex : avions ou zones géographiques ?

- Quelles informations entrent dans le système, quelles informations doivent sortir du système ?

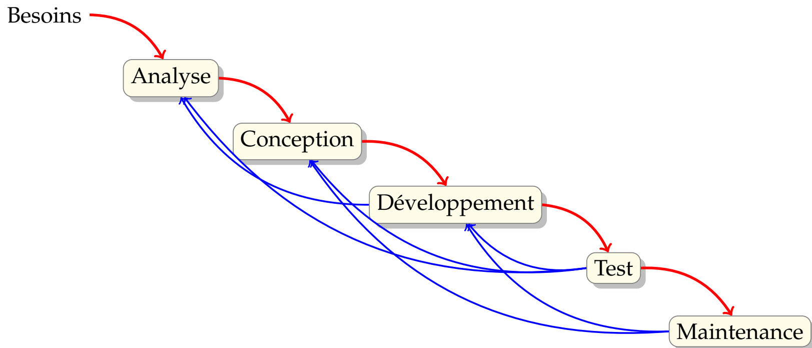
  capteurs (vision, son), message d'instructions (venant d'humains, d'autres agents) etc

- quels sont les buts ?

  se déplacer à une coordonnée, bloquer un adversaire, découvrir une zone, etc

Peut-on suivre une méthode standard pour la conception ?

Génie Logiciel



Ces étapes ne sont généralement pas suivies en cascade, mais peuvent être répétées itérativement.

Génie logiciel

Pour développer une *bonne* solution logicielle *rapidement*, on peut suivre une **méthodologie**. Cela va aider à *comprendre* le système (en modélisant une partie du système) et à concevoir le logiciel.

- elle décrit une séquence d'étapes
- elle produit des modèles
- elle définit des notation pour capturer les modèles (par exemple UML)
- elle décrit des techniques (comment faire certaines choses—des heuristiques)
- elle contient des outils pour aider à l'implémentation et à la vérification

Tension entre la vision formelle du concepteur (ou programmeur) et les besoins informels de l'utilisateur.

Une nouvelle méthodologie : pour quoi faire ?

Les SMA sont des logiciels, on peut donc utiliser des méthodes de génie logiciel, en particulier orientées objets.
MAIS

- un agent est *autonome* : il ne nécessite pas de contrôle extérieur, il décide à tout moment de ses actions
- un agent est *proactif* : il poursuit ses **buts** au cours du temps, il doit être attentif aux changements de l'environnement pour décider quels buts il peut/doit atteindre.
- un agent est *social* et communique avec d'autres agents à l'aide de **messages**.

  On peut avoir une organisation sociale (rôle des agents et obligations, normes, groupes)
- un agent est *réactif* ➾ Il faut donc porter une importance particulière aux **percepts** (forme perçue d'un stimulus externe) sinon un agent risque de manquer une opportunité.
- un agent est *situé* dans un *environnement* ➾ il faut prendre en compte les **actions** que peut exercer l'agent.

Les agents ne sont pas seulement des objets

- un objet ne contrôle pas son comportement
- le modèle objet n'a rien à dire de général à propos de la réactivité, de la pro-activité et de la communication
- les objets « classiques »peuvent utiliser le multi-threads, les SMA sont par définition multi-threads.

Cependant, les agents sont des logiciels, et souvent on peut adapter ou adopter des éléments de génie logiciel orienté objet.

ex : notation UML ⇨ notation AUML

Quand utiliser une approche agent ?

- système réactif ⇒ bien indiqué pour des environnements dynamiques, ouverts, incertains, complexes, etc.
- système disitribué ⇒ bien indiqué dans des environnements où les données, le contrôle et le traitement sont distribués (trafic aérien, routier, gestion d'eau, d'électricité)
- métaphores naturelles : l'environnement est une société d'agents (e-commerce, systèmes économiques, système d'informations distribuées, organisation)

- système est-il autonome ?
- a-t-il des buts ?
- vu comme un objet, est-il actif ?
- fait-il plusieurs choses en même temps ⇒ interaction
- doit-il adapter son comportement à son environnement ?

Historique du génie logiciel orienté agent

1995	DESIRE
1996	AAII, MAS-CommonKADS
1999	MaSE
2000	GAIA (v1), Tropos
2001	MESSAGE, Prometheus
2002	PASSI, INGENIAS
2003	GAIA (v2)
2005	ADEM
2007	O-MaSE

1– mileu à fin des années 90
- descriptions brèves, pas d'outils, pas toujours un cycle de vie entier

2– fin 90, début 2000
- descriptions plus détaillées, quelques outils, commence à couvrir le cycle de vie en entier

3– fin années 2000
- orienté vers les développement à l'aide de modèles (produire des modèles type UML), plus complexes

En ce moment

- moins d'efforts pour produire de nouvelles méthodologies
- moins de méthodologies « actives »
- efforts de standardisation et de consolidation

Exemple

Three parts « A » « B » and « C » (or more)
The system can be asked to assemble the parts to manufacture compounds such as « ABC », « ACB », « BA », etc..

Robot 1

- `load(part)`
- `unload()`
- `moveToFlipper()`
- `moveFromFlipper()`

Robot 2

- `join(jig)`

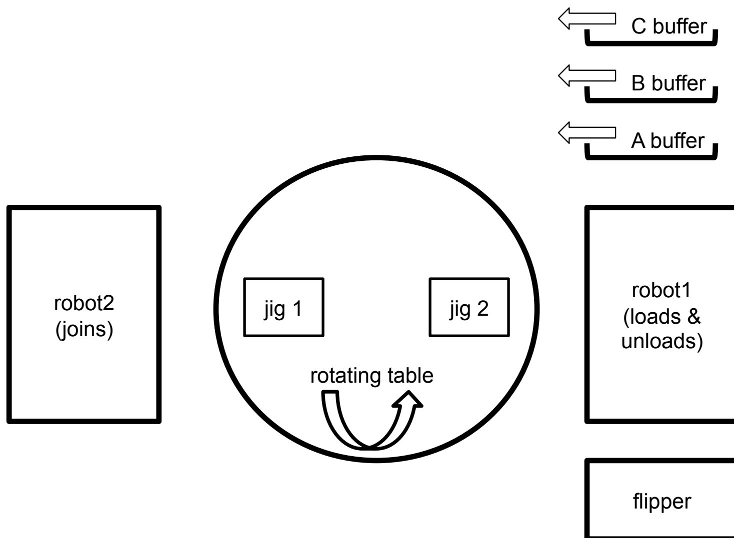
Flipper

- `flip()`

Table

- `rotateTo(jig,pos)`

Exemple



Les étapes générales communes aux méthodologies

1 – Etude des besoins
   (définitions de rôles, de scenarios, **buts**, interface avec l'environnement)
2 – Conception
   (types d'agent, structure statique, dynamique du système)
3 – Conception détaillée
   (structure interne de chaque agent)
4 – Implémentation
5 – Tests
6 – Maintenance

Besoins

Besoins

➩ définir les fonctionalités que le système doit apporter

Activités utilisées habituellement :

- donner des exemples du comportements désiré à travers des **scenarios**
- capturer les **buts** du système et leur relation
- définir les **interfaces** entre le système et son environnement

Ces activités sont généralement effectuées en parallèle et de façon itérative.

Rôles

Groupement d'actions, percepts et buts qui concernent la même fonctionalité.
➩ un agent pourra être une combinaison de rôles.

pour l'exemple :

- **manager :** responsible for overall management of the process. It does not perform any action
- **pickAndPlacer :** responsible for moving parts in and out of the jig when located on the east side.
  Associated actions : load, moveToFlipper, unload
- **fastener :** responsible for joining the parts together
  Associated actions : join
- **transporter :** responsible for transporting items by rotating the table.
  Associated action : rotateTo
- **flipper :** responsible for flipping the parts
  Associated action : flip

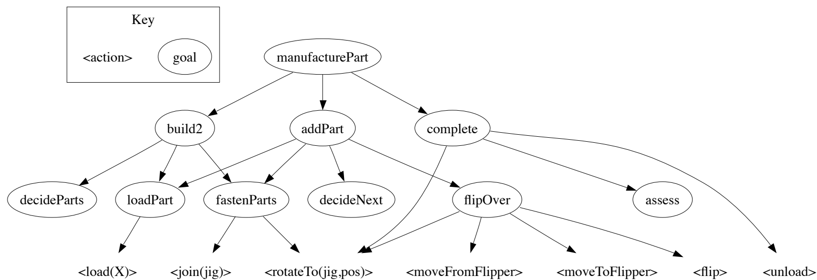
Buts

Les **buts** capturent les fonctionalités du système et leurs relations, mais ne sont pas attachées à une trace d'exécution.

Différentes techniques pour définir les buts :

- poser la question : *pourquoi* ce but doit être atteint ?
  ➩ identification des buts parents
  (approche ascendante « bottom up »)
- poser la question : *comment* ce but peut être atteint ?
  ➩ identifier les sous buts
  (approche descendante « top down »)
- autre solution : comment chaque but influence les autres buts.

Exemple



Scenarios

La définition de scenarios est courante dans les méthodologies de conception orientées objets.
⇨ définir des exemples de comportements du système en fonctionnement normal.

ex : traces d'exécution

- certaines méthodologies préconisent l'utilisation de textes pour décrire un scenario
- d'autres préconisent une description plus formelle pour automatiser certaines activités plus tard.

Exemple

Scenario: manufacturePart(ABC)

Type			**Name**	**Roles**
G			build2	manager, pickAndPlacer, fastener
	G		decideParts	manager
	G		loadPart	pickAndPlacer
		A	load(A)	pickAndPlacer
	G		loadPart	pickAndPlacer
		A	load(B)	pickAndPlacer
	G		fastenParts	fastener, transporter
		A	rotateTo(1,W)	transporter
		A	join(1)	fastener
G			addPart	manager, pickAndPlacer, fastener
	G		decideNext	manager
	G		flipOver	manager
		A	rotateTo(1,E)	transporter
		A	moveToFlipper()	pickAndPlacer
		A	flip()	flipper
	G		loadPart	pickAndPlacer
		A	load(C)	pickAndPlacer [in parallel with flip]
		A	moveFromFlipper()	pickAndPlacer
	G		fastenParts	fastener, transporter
		A	rotateTo(1,W)	transporter
		A	join(1)	fastener
G			complete	manager
	G		assess	manager
	A		rotateTo(1,E)	transporter
	A		unload()	pickAndPlacer

Interface avec l'environnement

- parfois elle est imposée par des contraintes existantes (hardware, système existant).
- parfois, le concepteur peut influencer l'interface
- parfois, le concepteur peut se placer à un plus haut niveau d'abstraction et implémente séparemment les contrôles bas niveau

Conception

Conception

- Quels sont les agents ? Quels sont leurs rôles ? Quels sont leurs buts ?
- Comment les agents communiquent-ils entre eux ?
- Comment les agents interagissent pour atteindre les buts du système ?

⇨ Production de deux modèles :

- une vue statique du système
- un modèle qui capture le comportement dynamique du système

Quels agents existent ?

comment décomposer/attribuer les rôles aux agents ?

- parfois il existe une solution naturelle
- décomposition pas trop fine ⇨ sinon couplage trop fort entre les agents (changement d'un agent va nécessiter des changements dans d'autres)
- un système est facile à comprendre si chaque type d'agent a une tâche bien précise. Si au contraire un agent se charge de beaucoup de tiaches différentes, le système peut devenir plus dur à comprendre.
- il peut exister des contraintes (hardware) qui font que certains rôles ne peuvent pas être regroupés dans le même agent.

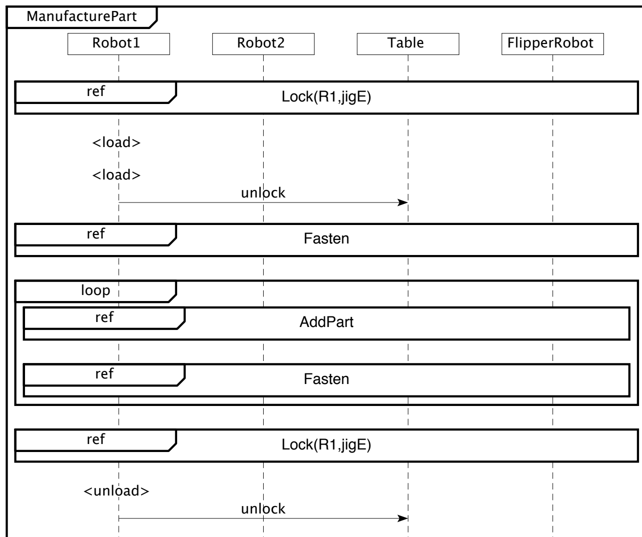
Exemple

Role	**Agent Type**	**Goals and** ***Actions***
pickAndPlacer	Robot1	loadPart, *load, unload, movetoFlipper,moveFromFlipper*
manager	Robot1	decideParts, decideNext, flipOver, assess
transporter	Table	*rotateTo*
fastener	Robot2	fastenParts, *join*
flipper	FlipperRobot	*flip*

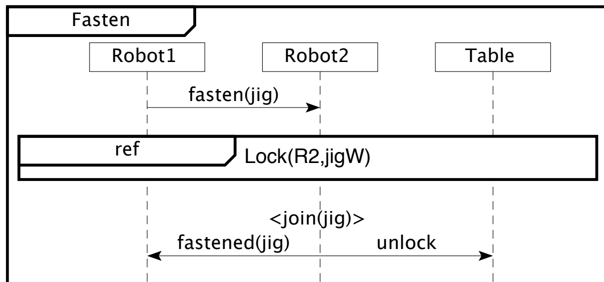
Structure de communication (messages, protocoles)

Les scénarios aident pour décider de la structure des messages.
Bien penser à tous les cas qui peuvent survenir (différents messages, erreurs, différent ordre de message)
Représentation à l'aide de diagrammes AUML.

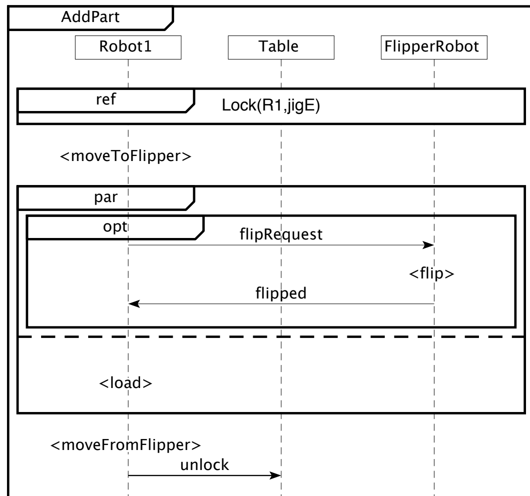
Exemple : protocole pour fabriquer une pièce composite



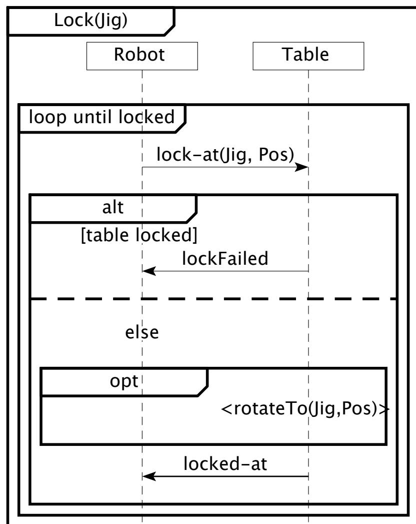
Exemple : protocole pour joindre deux composants



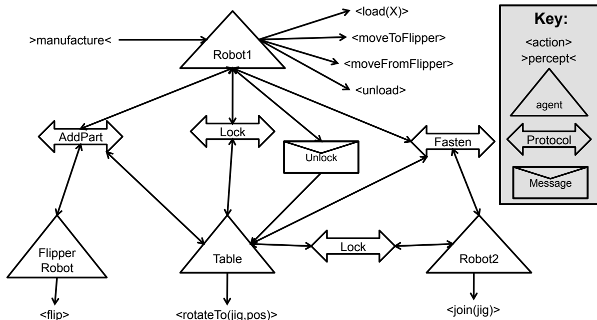
Exemple : protocole pour ajouter un composant



Exemple : protocole pour « locker »



Le système peut être capturer dans un modèle général statique.



Conception détaillée

Conception détaillée

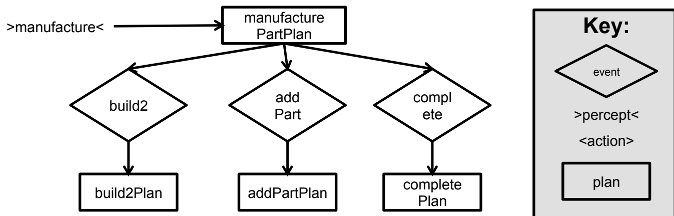
Décrire la structure interne des agents.

- comment les agents se comportent pour atteindre leurs buts ?
- comment répondre aux messages reçus ou aux stimuli extérieurs ?

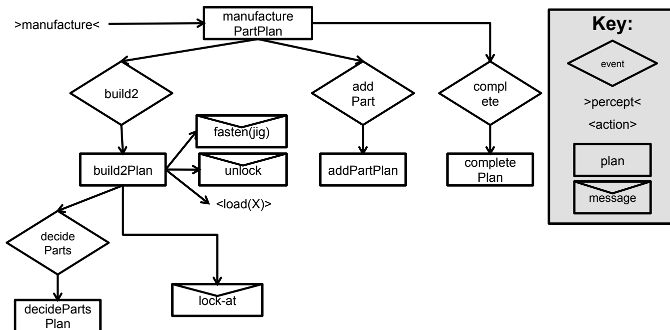
But : atteindre une précision suffisante pour pouvoir effecture la phase d'implémentation.

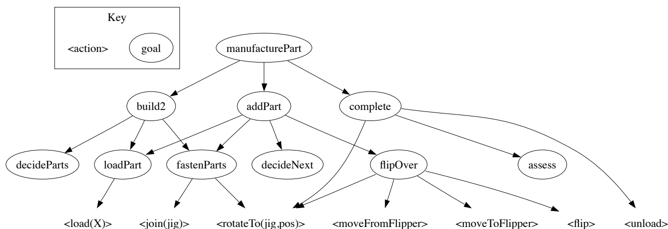
- conception de l'interface (actions et communication)
- les modèles dépendent de la plateforme de programmation utilisée (ex : JADE, BDI agents)

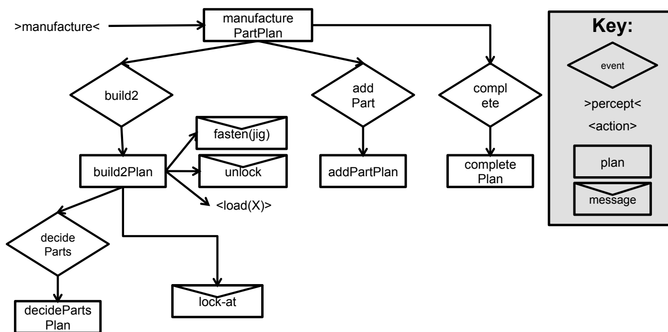
Exemple : pour BDI agents, étape initiale, plan pour Robot 1



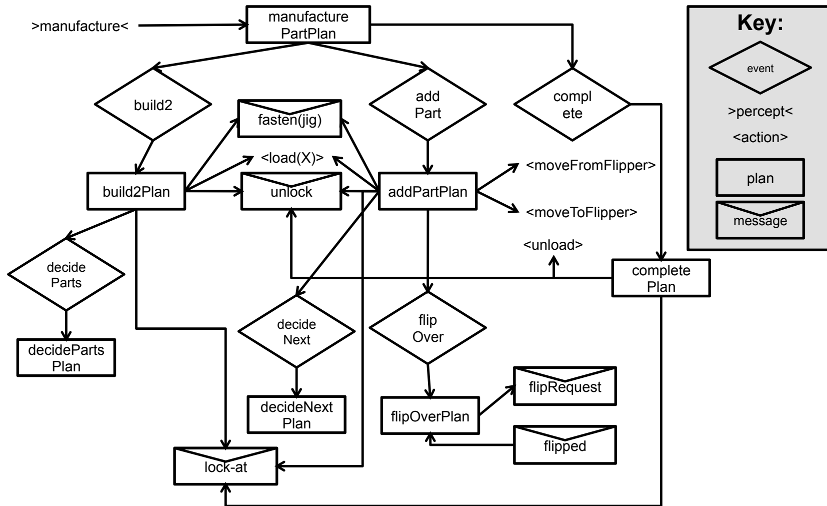
Exemple : étape 2, plan pour Robot 1



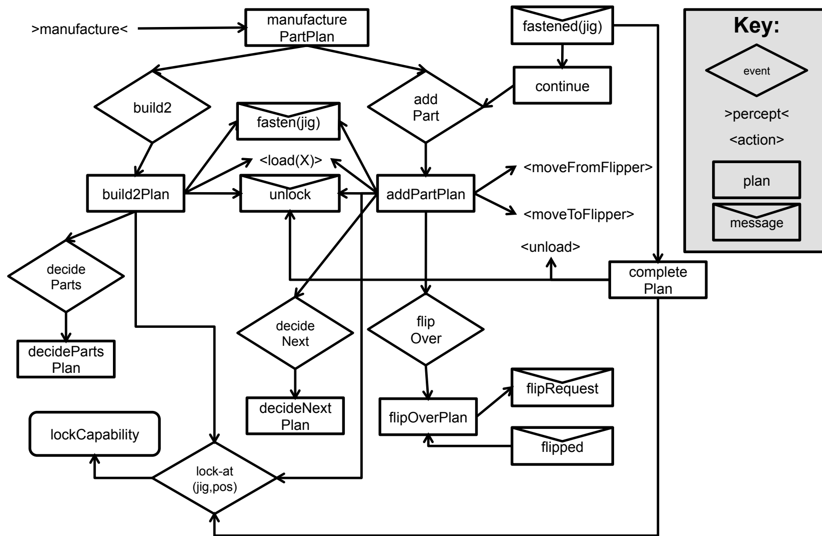

Key
<action>
goal
manufacturePart
build2
addPart
complete
decideParts
loadPart
fastenParts
decideNext
flipOver
assess
<load(X)>
<join(jig)>
<rotateTo(jig,pos)>
<moveFromFlipper>
<moveToFlipper>
<flip>
<unload>


>manufacture<
manufacture PartPlan
build2
add Part
compl ete
fasten(jig)
unlock
<load(X)>
build2Plan
addPartPlan
complete Plan
decide Parts
decideParts Plan
lock-at
Key:
event
>percept<
<action>
plan
message

Exemple : étape 3, plan pour Robot 1



Exemple : étape 4, plan pour Robot 1

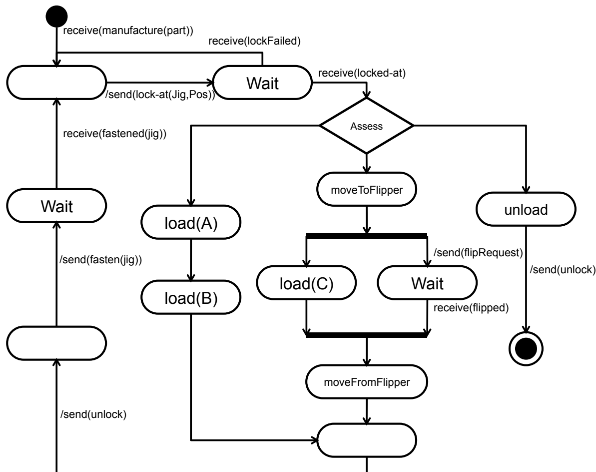


Finalisation

- vérifier que les messages envoyés sont bien reçus.
- ajouter des plans : SimpleAdd, FlipAnAdd
  ⇨ permet des solutions alternative en cas d'échec.

Exemple : à l'aide d'automates finis

Notation de O-MaSE



Implémentation

- PDT (Prometheus) ➾ produit du code en JACK
- TAOM4E (Tropos) ➾ produit du code en Jadex
- IDK (INGENIAS) ➾ produit du code en JADE
- agentTool III (O-MaSE) ➾ produit du code en JADE
- PTK (PASSI) ➾ produit du code en JADE avec AgentFactory

peu proposent un « allé-retour » où des changements d'implémentation peuvent modifier les modèles de conception (Prometheus fait un peu).

Tests

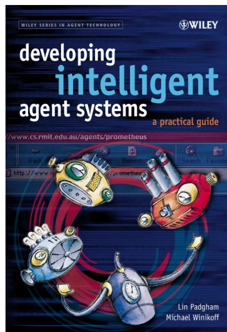
- « unit-testing » (test de modules)
- « interaction testing » (tests entre modules)
- « sytem testing », « acceptance testing » (comportement du système global)

- test cases

Certains outils dédiés (eCATpour Tropos, surtout pour vérifier envois et reception de messages)
Certains génèrent des « test cases » automatiquement

Développement d'outils pour la vérification formelle.
ex : le système est décrit à l'aide d'un automate, des propriétés sont décrites dans une logique temporelle, et on utilise « model checking » pour vérifier le système
ex : ConGolog programme $\rightarrow$ traduit dans une logique de grand ordre $\rightarrow$ vérification à l'aide de « theorem proving »

Exemple avec Prometheus

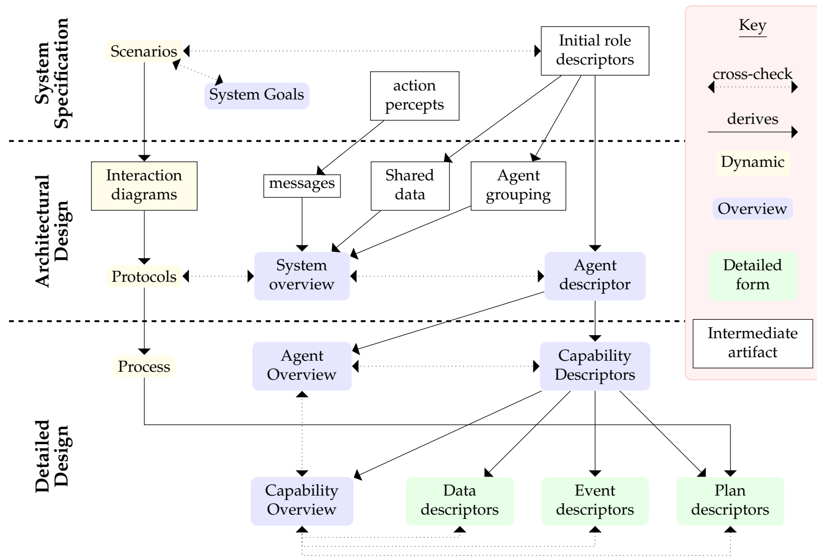


Prometheus Design Tool (PDT)

Outil de conception

- Aide à la conception : trois phases reliées
- Aide à la vérification : aide à la cohérence des modèles
- Aide à l'implémentation : génère un squelette de code (dans le language JACK)

1– Utilisation de PDT pour concevoir et analyser le système, génération du squelette du code
2– Utilisation de Eclipse pour l'implémentation à partir du squelette du code
3– Utilisation de JACK pour compiler et exécuter le système


System Specification
Architectural Design
Detailed Design
Scenarios
System Goals
action percepts
Initial role descriptors
Interaction diagrams
messages
Shared data
Agent grouping
Protocols
System overview
Agent descriptor
Process
Agent Overview
Capability Descriptors
Capability Overview
Data descriptors
Event descriptors
Plan descriptors
Key
cross-check
derives
Dynamic
Overview
Detailed form
Intermediate artifact

Prometheus Design Tool (PDT)

Conclusion

- choix de méthodologies pour développer des systèmes multiagents.
- choix en fonction du language agent utilisé
- il existe aussi « Organisation Oriented Programming »

⇒ l'organisation est l'objet principal

  - méchanismes de coordination
  - méchanismes de régulation
  - agents : entrée/sortie dans l'organisation, changer l'organisation, suivre/violer une norme, sanctionner/encourager d'autres agents
  - etc...