MASTER EN SCIENCES INFORMATIQUES

Prototypage de spécifications formelles des besoins
D'ALBERT vers OBLOG

Dubru, Frédéric

Award date:
1993

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix à Namur
Institut d'Informatique

PROTOTYPAGE DE SPÉCIFICATIONS FORMELLES DES BESOINS

D'ALBERT vers OBLOG

Frédéric Dubru

Mémoire présenté en vue de l'obtention
du diplôme de Maître en Informatique

Année académique 1992-1993

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage 21, B-5000 Namur
Tél. 081/72.49.83
Fax 081/72.49.67

PROTOTYPAGE DE SPÉCIFICATIONS FORMELLES DES BESOINS

Frédéric Dubru

RÉSUMÉ • L'ingénierie des besoins (également appelée analyse des besoins) est considérée aujourd'hui comme une activité critique dans le cadre du développement de logiciel. ALBERT est un langage formel de spécification conçu pour exprimer les besoins de systèmes composites temps-réel. L'objectif de ce mémoire est de construire, à partir d'une spécification ALBERT, un prototype qui aide l'analyste à valider cette spécification. Le langage OBLOG a été choisi pour concevoir ce prototype, à la fois pour son apparente "proximité" par rapport à ALBERT et pour son caractère exécutable. Deux méthodes de prototypage ont été suivies. La première a échoué mais a permis de mettre en évidence les différences entre ALBERT et OBLOG et de préciser la nature du prototypage dans le cadre de l'ingénierie des besoins. La seconde atteint l'objectif fixé. Outre ce résultat concret, ce mémoire développe un étude de cas complète, évalue les langages utilisés et, plus généralement, permet de mieux comprendre les limitations d'un langage exécutable à des fins de spécification.

ABSTRACT • Requirements Engineering (also called Requirements Analysis) is now recognised as a critical activity in the context of software development. ALBERT is a formal specification language designed for capturing requirements of composite real-time systems. The objective of the thesis is to build, from an ALBERT specification, a prototype to help the analyst validate this specification. The OBLOG language has been chosen to design this prototype, because of its apparent "proximity" to ALBERT and its executability. Two prototyping methods have been experimented. The first failed but helped to highlight the differences between ALBERT and OBLOG and to make clear the nature of prototyping in the context of Requirements Engineering. The second attained its object. In addition to this concrete result, the thesis develops a complete case study, evaluates the languages involved and, more generally, provides a better understanding of the limitations of an executable language for specification purposes.

Mémoire de Maîtrise en Informatique
Juin 1993
Promoteur : Eric Dubois

Je tiens à remercier vivement mon promoteur, le professeur Eric Dubois. Sans ses conseils et ses recommandations, ce mémoire n'aurait jamais pu aboutir.

Merci aussi à Philippe Du Bois, qui a consacré beaucoup de temps à relire mon travail et à m'apporter ses précieuses explications, notamment sur le langage ALBERT.

Enfin, merci à Jean-Marc Zeippen. C'est au cours d'un stage au Portugal que j'ai appris OBLOG et acquis les bases nécessaires à la réalisation de ce mémoire. Il est pour beaucoup dans la réussite de ce stage.

TABLE DES MATIÈRES

INTRODUCTION

1. Introduction 2
   - 1.1. Motivation 2
   - 1.2. Objectifs 4
   - 1.3. Plan de travail 5

PARTIE I - ALBERT

2. L'ingénierie des besoins 7
   - 2.1. Terminologie employée dans ce travail 7
   - 2.2. Tâches et résultats de l'I.B. 8
   - 2.3. Les spécifications des besoins 9
   - 2.4. Les langages d'I.B. 12

3. Le langage ALBERT 16
   - 3.1. Introduction 16
   - 3.2. Exemple d'application 16
   - 3.3. Le modèle-produit 17
   - 3.4. Le modèle-procédé 26
   - 3.5. Hypothèses sur le langage 28

PARTIE II - OBLOG

4. L'approche OBLOG 33
   - 4.1. Introduction 33
   - 4.2. Le langage OBLOG-kernel 35
   - 4.3. Le langage OBLOG-light 44
   - 4.4. L'outil 51

5. Étude comparée d'ALBERT et d'OBLOG 54
   - 5.1. Comparaison de la sémantique des deux langages 54
   - 5.2. Comparaison de la logique utilisée dans les deux langages 56
   - 5.3. Comparaison au niveau syntaxique 57

PARTIE III - D'ALBERT VERS OBLOG

6. Le prototypage 59
- 6.1. Le prototypage dans le cycle de vie du logiciel 59
- 6.2. Prototypage ou animation des systèmes composites 63
- 6.3. Démarche adoptée 65

7. Première méthode 69
- 7.1. Introduction 69
- 7.2. Transformation des déclarations 71
- 7.3. Transformation des effets 73
- 7.4. Transformation des initialisations 74
- 7.5. Transformation de la responsabilité 74
- 7.6. Transformation de la perception et de la publicité 75
- 7.7. Transformation des formules 77
- 7.8. Transformation des contraintes sur l'état 80
- 7.9. Transformation des engagements 81
- 7.10. Evaluation 85

8. Deuxième méthode 87
- 8.1. Introduction 87
- 8.2. Fonctionnalités de l'animateur 88
- 8.3. Phase de préparation 91
- 8.4. Architecture du modèle générique 94
- 8.5. L'objet CLOCK_DBX 98
- 8.6. L'objet clock 98
- 8.7. L'objet u1_dbx 101
- 8.8. L'objet u1 101
- 8.9. L'objet state_dbx 107
- 8.10. La classe d'objets STATE 107
- 8.11. La classe d'objets CHANGE 109
- 8.12. La classe d'objets S1 113
- 8.13. L'objet s1_dbx 119
- 8.14. La classe d'objets R1 119
- 8.15. A propos du temps... 122
- 8.16. Evaluation 125

CONCLUSION

9. Conclusion 127
- 9.1. Évaluation des langages utilisés 127
- 9.2. Apports de ce travail 129
- 9.3. Perspectives d'avenir 131

BIBLIOGRAPHIE 133

ANNEXE - ÉTUDE DE CAS

A. Étude de cas 137
A.1. Énoncé du cas 137
A.2. Décomposition du problème 139
A.3. Types de donnée 140
A.4. Spécification de l'entreprise 140
A.5. spécification de l'atelier (premier temps) 144
A.6. spécification de l'atelier (deuxième temps) 152
A.7. spécification de l'atelier (troisième temps) 159
A.8. Prototype de la spécification du véhicule (première méthode) 164
A.9. Prototype de la spécification du véhicule (deuxième méthode) 169

INTRODUCTION

1. INTRODUCTION

1.1. MOTIVATION

a) Spécifications des besoins

Il est reconnu aujourd'hui que la phase la plus critique et la plus difficile du cycle de développement d'un logiciel est la phase d'analyse des besoins, au cours de laquelle l'informaticien décrit avec précision le système à mettre en place, ses objectifs, ses fonctions, etc...

C'est la phase la plus critique dans la mesure où c'est la première et que, à ce titre, elle conditionne toutes les phases ultérieures. Une erreur à ce niveau n'est parfois détectée que lors de l'implémentation et il est alors très difficile et très coûteux de la corriger.

C'est aussi la plus difficile car l'informaticien doit produire un dossier précis et structuré, voire formalisé, à partir d'informations collectées auprès du client et qui sont souvent imprécises, mal-structurées, incomplètes ou incohérentes. Ce travail difficile en soi, mais souvent intéressant, nécessite une étroite collaboration entre l'informaticien et le client. Or, cette collaboration n'est pas facile à mettre en oeuvre et est parfois source de conflits.

On le voit, l'analyse des besoins relève à la fois de la science du management et de la science de l'informatique. Alors que les objectifs généraux du système seront fixés par les dirigeants de l'organisation, l'informaticien est davantage concerné par les *spécifications des besoins*, qui forment une description précise du système.

L'informatique accorde un intérêt croissant à l'analyse des besoins, au point d'en faire une discipline à part entière, appelée aussi *ingénierie des besoins*. De nombreuses méthodes, parfois supportées par des outils, ont été développées pour assister les informaticiens dans leur tâche d'analyse et de rédaction des spécifications des besoins. Certaines ont eu un franc succès, comme le modèle Entité-Association, mais beaucoup ne se sont pas révélées suffisamment efficaces.

Deux phénomènes importants influencent aujourd'hui cette jeune discipline :
- d'une part, l'apparition de langages formels de spécification des besoins, pour pallier au manque de sémantique formelle des langages existants ;
- d'autre part, la prise de conscience que les spécifications des besoins ne doivent pas se limiter à décrire un système informatique mais également son environnement ([Dub93b]).

b) Formalisation

Habituellement, les spécifications des besoins sont rédigées en langage naturel tel que le français ou l'anglais. Malheureusement, dès que le système à décrire est relativement important, il devient difficile de rédiger un document parfaitement structuré et complet, exempt de toute imprécision ou incohérence. Or, les spécifications des besoins doivent servir de base au développement d'un logiciel qui ne supporte aucune interprétation.

Les langages formels (comme le langage mathématique) fournissent des règles rigoureuses d'interprétation qui garantissent la précision et l'absence d'ambiguïtés. En outre, ils supportent des raisonnements formels tels que le contrôle de cohérence ou de complétude. Evidemment, ces précieux avantages se paient par le fait que les langages formels sont incompréhensibles pour les non-initiés, en particulier pour le client.

Si les langages formels de spécification des besoins sont des outils précieux pour les informaticiens, ils contribuent à gêner la communication indispensable entre l'informaticien et le client. Dès lors, il importe de trouver les moyens de rendre les spécifications des besoins accessibles par les non-initiés. On peut penser :
- à la génération automatique, à partir de spécifications formelles, de documentation en langage naturel ([Lur92]) ;
- au prototypage des spécifications.

c) Prototypage

S'agissant d'un logiciel, un prototype est une version provisoire qui permet d'expérimenter certaines caractéristiques du logiciel avant de le développer réellement. La technique du prototypage est apparue très intéressante pour obtenir un feed-back rapide de la part des utilisateurs finaux du logiciel.

Appliqué à l'analyse des besoins, le prototypage doit permettre d'aider à vérifier l'adéquation des spécifications des besoins par rapport aux besoins réels de l'organisation. Lorsqu'elles sont rédigées en langage naturel, il est très difficile (et très coûteux) de construire un prototype. Par contre, il est possible de dériver de manière automatique, ou du moins semi-automatique, un prototype à partir de spécifications formelles.

Ainsi, on peut penser que le caractère inintelligible des langages formels est compensé par le fait que ces mêmes langages formels autorisent la génération de prototype. En effet, s'il peut produire aisément un prototype de ses spécifications des besoins, l'informaticien peut les valider, d'abord par rapport au modèle mental qu'il s'est fait du problème et ensuite par rapport aux besoins effectifs en collaboration avec le client.

1.2. OBJECTIFS

a) Objectif principal

L'objectif ultime de ce mémoire est de construire, à partir d'une spécification des besoins rédigée en ALBERT, un prototype (plus précisément un *animateur*) qui aide l'analyste à valider cette spécification. Ce prototype sera spécifié en OBLOG.

• ALBERT est un langage formel, conçu pour exprimer les besoins de systèmes composites, c'est-à-dire hétérogènes, et comportant des contraintes en temps réel. Une spécification ALBERT décrit un système comme une société d'*agents*, un agent pouvant représenter une procédure manuelle, une machine, un logiciel, etc. Chaque agent est caractérisé par ses responsabilités vis-à-vis des événements du système. Le langage s'accompagne d'une guidance méthodologique qui propose une rédaction incrémentale des spécifications.

• OBLOG n'est pas un langage de spécification des besoins ; il supporte plutôt la phase de conception des systèmes d'information classiques. OBLOG est un langage formel, orienté-objet, qui permet de modéliser un système d'information comme une collection d'objets concurrents qui interagissent entre eux. Outre un langage, l'approche OBLOG comprend également un outil (en fait, un atelier logiciel complet, comprenant la génération automatique du code des spécifications) et une méthodologie.

Pour comprendre le choix d'ALBERT et d'OBLOG, il faut savoir que ce mémoire s'inscrit dans un plan de recherche commun à ces deux langages. Ce plan comprend deux volets :
- d'une part, les concepteurs d'OBLOG ont le projet à long terme d'étendre la portée du langage afin qu'il puisse couvrir également la phase de spécification des besoins (l'utilisation d'OBLOG pour le prototypage des spécifications des besoins est une première étape en ce sens) ;
- d'autre part, les concepteurs d'ALBERT envisagent d'étudier la dérivation systématique de spécifications de conception en OBLOG, à partir de spécifications des besoins en ALBERT. Rien n'interdit de penser que notre approche de prototypage soit retenue comme point de départ.

En outre, le choix d'OBLOG comme langage cible se justifiait par son apparente "proximité" par rapport à ALBERT et par son caractère exécutable. (Pour ce qui est de la proximité entre les deux langages, elle s'est révélée, à l'expérimentation, fort trompeuse.)

b) Objectifs secondaires

A côté de l'objectif principal de prototypage, ce mémoire poursuit deux objectifs secondaires.

• Premièrement, pour bien comprendre le langage ALBERT et illustrer les méthodes de prototypage, il importait de développer une étude de cas. Nous avons choisi une application CIM concernant un atelier d'usinage. Ce cas est intéressant à spécifier en ALBERT puisqu'il s'agit d'un système composite (comprenant un robot, un logiciel, un véhicule,...) comportant

des contraintes de délai de production. Ce travail de spécification constitue une des premières applications concrètes d'ALBERT et permet dès lors de vérifier les qualités du langage.

• Deuxièmement, la confrontation entre ALBERT et OBLOG nous a permis d'évaluer certaines propositions et d'en formuler d'autres quant à l'évolution du langage OBLOG vers un langage de spécification des besoins.

1.3. PLAN DE TRAVAIL

Ce mémoire est divisé en trois parties, consacrées respectivement au langage ALBERT, au langage OBLOG et à la transformation d'une spécification de l'un vers l'autre.

a) ALBERT

Le chapitre 1 étudie de manière approfondie l'activité d'ingénierie des besoins, ainsi que les langages de spécification des besoins. Le chapitre 2 présente plus particulièrement le langage ALBERT, ses concepts et sa méthodologie, en les illustrant sur le cas de l'atelier d'usinage (un développement complet du cas étant développé en annexe).

b) OBLOG

Le chapitre 4 présente l'approche OBLOG : le langage, l'outil et, dans une moindre mesure, la méthode. Pour conclure la présentation des langages ALBERT et OBLOG, le chapitre 5 mène une étude comparée de ces deux langages.

c) D'ALBERT vers OBLOG

Le chapitre 6 est un chapitre charnière. Après avoir présenté une étude générale du prototypage, il décrit la démarche spécifique que nous avons adoptée pour prototyper une spécification ALBERT en OBLOG. Les chapitres 7 et 8, finalement, présentent les deux méthodes successives de prototypage que nous avons suivies. La première méthode est intéressante pour ses conclusions mais aboutit à un échec, tandis que la seconde tire les leçons de la première et répond à l'objectif principal que nous nous sommes fixé.

PARTIE I

ALBERT

ERRATUM • Les pages 6 et 7 sont inversées.

2. L'INGÉNIERIE DES BESOINS

Après quelques éléments terminologiques (section 2.1), la section 2.2 étudie l'ingénierie des besoins selon deux axes : les tâches et les résultats de l'ingénierie des besoins. Parmi ces résultats, les spécifications des besoins sont étudiées plus en profondeur à la section 2.3 La section 2.4, enfin, aborde les langages qui supportent l'ingénierie des besoins.

2.1. TERMINOLOGIE EMPLOYÉE DANS CE TRAVAIL

La plupart des méthodes de développement de logiciel distinguent aujourd'hui deux activités principales dans le cycle de vie d'un logiciel [Dub93b].

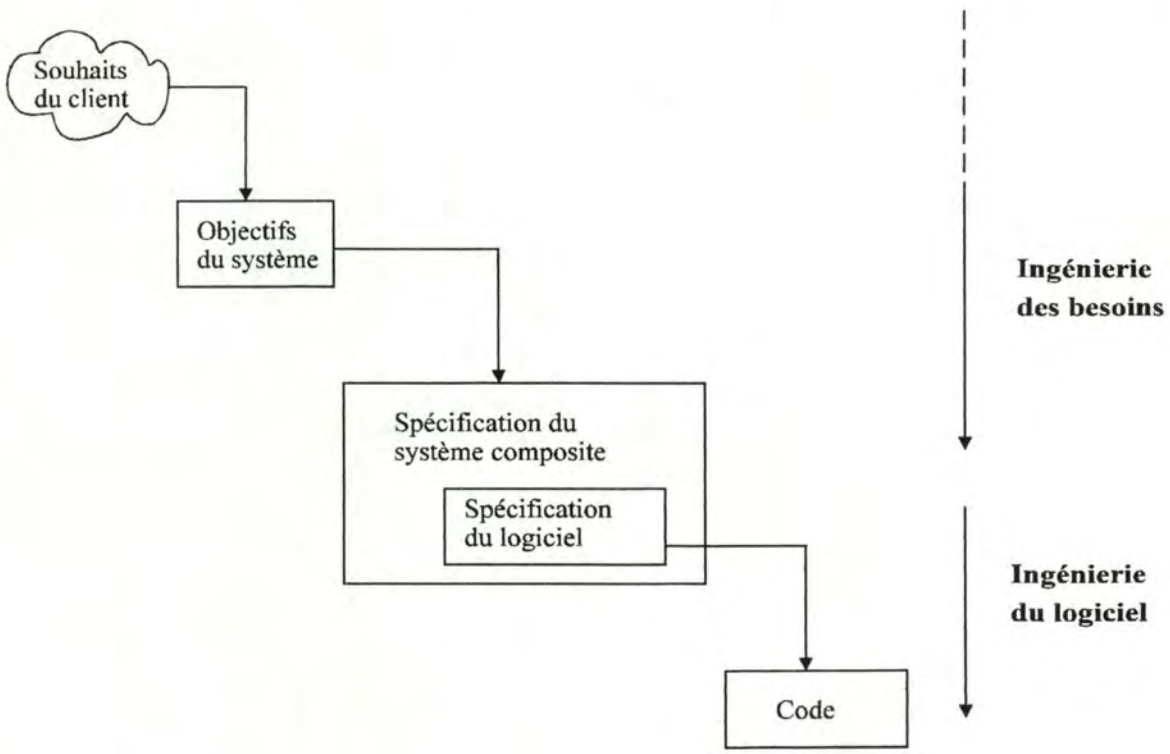


Figure 1 - Ingénierie des besoins et ingénierie du logiciel [Dub93b]

• L'*ingénierie des besoins (requirements engineering)* consiste à reformuler de manière précise et structurée les souhaits informels du client. Le produit de cette activité comprend la définition des *objectifs* du système à l'étude ainsi que sa description détaillée, appelée *spécifications des besoins*. Ces deux composantes font partie du *cahier des charges*. (L'ingénierie des besoins est également appelée *analyse des besoins*. Dans la suite, nous utiliserons l'abréviation I.B.)

• L'*ingénierie du logiciel (software engineering)* est l'activité qui développe un système informatique répondant aux exigences du cahier des charges. Cette activité se base sur des *spécifications de conception (design)*. (L'ingénierie du logiciel est également appelée *ingénierie de conception*.)

2.2. TÂCHES ET RÉSULTATS DE L'I.B.

2.2.1. LES QUATRE TÂCHES DE L'I.B.

Dubois propose un modèle de l'I.B. qui permet de mieux comprendre la nature de cette activité. Comme le suggère la figure 2, l'I.B. peut être décomposée en quatre tâches [Dub91a].

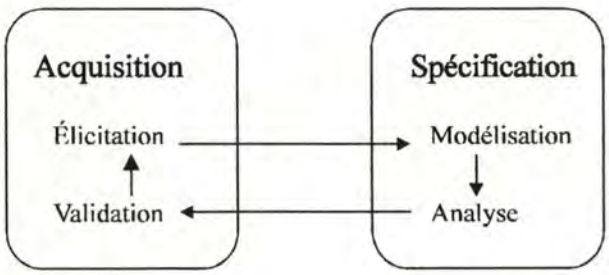


Figure 2 - Les quatre tâches de l'I.B. [Dub91a]

• Durant la phase d'*élicitation*, l'analyste collecte des informations sur le problème du client par différents moyens : interviews, discussions, observations, étude de la documentation disponible,...

• La tâche de *modélisation* consiste à traiter ces différentes informations afin d'élaborer un modèle, c'est-à-dire une représentation du problème. Ce modèle est un nouveau fragment ajouté au cahier des charges élaboré jusque là.

• Durant la tâche d'*analyse* proprement dite, l'analyste détecte les problèmes de contradiction ou d'incohérence provenant de l'incorporation de ce nouveau fragment.

• La tâche de *validation* sert à résoudre les éventuels problèmes révélés par la tâche précédente et à vérifier l'adéquation des spécifications des besoins avec les besoins effectifs du client.

Ces quatre tâches sont regroupées en deux activités : *acquisition* et *spécification*. La première met l'accent sur l'interaction nécessaire entre l'analyste et le client. Lors de la seconde activité, l'analyste travaille seul à la rédaction des spécifications. Il utilise pour cela un langage lui permettant de modéliser les informations recueillies et d'analyser cette modélisation [Pet92]. (L'étude des langages d'I.B. fait l'objet de la section 2.4).

2.2.2. RÉSULTATS DE L'I.B.

L'I.B. débouche sur l'élaboration du cahier des charges qui, une fois approuvé par les différentes parties concernées (clients, analystes et implémenteurs), constituera un contrat pouvant servir de référence en cas de litige dans les étapes ultérieures du cycle de vie [Pet92].

Ce cahier des charges peut être structuré en trois niveaux :
- la définition générale des *objectifs* du système, qui comprend notamment la place du système dans le contexte plus large de l'organisation et ses plans d'expansion. Cette définition est à resituer dans le cadre du *schéma directeur* qui reflète la stratégie du développement informatique de l'organisation ;
- les spécifications des besoins qui décrivent précisément le système à mettre en place ;
- des prescriptions qui relèvent plus de la gestion du projet que de l'I.B. : délais à respecter pour le développement, langages à utiliser, système de tarification, etc... Elles sont parfois appelées *spécifications non-fonctionnelles*.

La section suivante développe de façon plus large ce qu'il faut entendre par spécifications des besoins.

2.3. LES SPÉCIFICATIONS DES BESOINS

2.3.1. DÉFINITION

Nous retiendrons la définition de spécifications des besoins proposée par [Hab90] : « une description de ce qu'on attend que le futur système fasse, plutôt qu'une description de la manière dont il va le faire ».

En principe, l'I.B. devrait être une activité centrée sur le client. A ce stade, seules sont pertinentes « les choses à propos du système que les utilisateurs peuvent observer, et non les choses dont les programmeurs peuvent avoir besoin pour les implémenter » [Hab90]. L'informaticien doit donc s'efforcer de raisonner en termes de "choses" significatives pour le

client : objets et fonctions reconnaissables par le client, contraintes d'utilisation, événements de la vie de l'organisation...

2.3.2. SYSTÈMES COMPOSITES

Dans la définition, le mot "système" ne doit pas être pris au sens limitatif de "système informatique". L'I.B. ne doit pas seulement s'attacher à décrire un système informatique mais également son *environnement* et la nature des interactions entre les deux [Dub93b]. D'ailleurs, dans un certain nombre de cas, c'est l'analyse des besoins qui permet de décider s'il est opportun d'informatiser et, dans l'affirmative, de choisir les éléments qui feront l'objet de cette informatisation.

Un *système composite* (fait de plusieurs composants hétérogènes) peut donc donner lieu à des descriptions de procédures manuelles ou d'installations matérielles spécifiques, par exemple ([Dub93b]) :
- "ce département a la charge d'encoder les bons de commandes le jour de leur arrivée
- "tel instrument analogique mesure le pouls du patient".

2.3.3. CLASSIFICATION DES SPÉCIFICATIONS DES BESOINS

D'après [ICARUS89], on peut distinguer trois types de spécifications des besoins.

• Les *spécifications fonctionnelles* décrivent les opérations que devra effectuer le futur système ainsi que les différentes classes d'objets impliquées dans ces opérations.

• Les *spécifications comportementales (behavioural requirements)* couvrent les aspects de performance et de précision du futur système, par exemple ([Dub93b]) :
- "quand la température de l'eau dépasse 100 °C, l'alarme doit être déclenchée dans les 10 secondes" ;
- "la quantité en stock enregistrée dans le système d'information reflète la quantité réelle du stock, avec un certain écart delta".

• Les *spécifications environnementales* décrivent l'environnement dans lequel le système devra fonctionner, fixent la frontière et définissent les interactions entre la partie automatisée et la partie manuelle.

2.3.4. FRONTIÈRES DES SPÉCIFICATIONS

Il est frappant de constater que la définition proposée plus haut, comme la plupart des explications relatives aux spécifications des besoins, s'appuient sur une opposition par rapport aux programmes[1]. Comme s'il était difficile de définir le *QUOI* sans faire référence au *COMMENT*. En réalité, la distinction entre les deux n'est pas aussi facile à déterminer qu'il n'y paraît. La réflexion qui suit est empruntée à [Hab90].

Dans d'autres domaines d'ingénierie, la frontière entre la description d'un produit et le produit lui-même est bien établie ; personne ne confond, par exemple, un circuit électronique et son plan. Par contre, cette frontière semble être moins évidente dans le cas des produits logiciels. Cette confusion est essentiellement due à trois particularités du champ d'ingénierie logicielle.

• Premièrement, un programme est un ensemble d'instructions formelles exécutables sur une machine. Avec l'usage progressif des langages formels d'I.B. (dont on discutera les avantages à la section 2.4), la tentation est grande de mélanger des éléments du problème et des éléments de la solution dans un même formalisme.

• Deuxièmement, le développement de logiciels est un domaine relativement récent. La distinction entre les différents acteurs impliqués (programmeurs, ingénieurs-systèmes, informaticiens...) n'est pas aussi bien établie que celle, par exemple, qui existe dans le champ de l'ingénierie électronique (techniciens, ingénieurs-électroniciens et physiciens) [Lam88].

• Troisièmement, dans certaines approches transformationnelles de développement, une certaine partie des spécifications des besoins (les spécifications fonctionnelles associées à la partie logicielle du système) peut réellement être considérée comme une première version du futur système informatique.

On pourrait qualifier la différence entre "spécifications des besoins" et "programme" selon quatre critères [Hab90] :
- leur rôle respectif : "modèle utilisé comme base contractuelle" vs "système opérationnel exécutable sur une configuration concrète particulière" ;
- leur portée : "problème" vs "solution", "classe de comportements" vs "comportement spécifique" ;
- les responsables concernés : "analyste" vs "programmeurs" ;
- les points de vue adoptés : "vue de l'utilisateur" vs "vue de l'implémenteur".

Pour conclure, nous insisterons donc sur la nécessité de bien définir la nature des différents produits du cycle de vie afin d'éviter toute confusion. Cette préoccupation reviendra lorsque nous parlerons des prototypes, au chapitre 6.

---

[1] Le terme "programme" a dans ce mémoire un sens très large qui inclut les produits intermédiaires entre la spécification des besoins et le programme proprement dit (par exemple les spécifications de conception). Cette interprétation est conforme à la tendance actuelle d'après [Hab90].

2.3.5. QUALITÉS DÉSIRÉES DES SPÉCIFICATIONS DES BESOINS

De "bonnes" spécifications devraient réunir les caractéristiques suivantes [Dub82, Lam88] :

- *cohérence et complétude* : intuitivement, la cohérence signifie que les spécifications ne contiennent aucun aspect contradictoire et la complétude signifie que tous les éléments qu'elles contiennent sont bien définis ;
- *minimalité* : les détails d'implémentation ou les aspect d'une solution particulière, les informations redondantes ou les bruits (qui n'apportent aucune information nouvelle sur le problème) doivent être exclus ;
- *précision* et *non-ambiguïté* : la précision doit empêcher toute interprétation erronée des spécifications, tandis que la non-ambiguïté n'autorise qu'une et une seule interprétation possible ;
- *compréhensibilité* à la fois pour les utilisateurs et les implémenteurs ;
- *bonne structure* : un document bien structuré doit limiter les références en avant, introduire les concepts principaux avant les concepts secondaires et être adaptable dans le sens où des changements peuvent être facilement localisés.

2.4. LES LANGAGES D'I.B.

Très vite, les informaticiens-analystes ont ressenti le besoin de disposer d'outils et de méthodes pour les assister dans leur tâche d'analyse. Un des outils les plus précieux a été et reste encore aujourd'hui le formalisme Entité-Association, développé notamment dans [Bod89]. Cependant, rares sont les méthodes qui apportent une réelle guidance et supportent efficacement les tâches d'analyse et de validation. La plupart d'entre elles ont une syntaxe rigoureuse mais souffrent d'un manque de sémantique formelle.

2.4.1. PROPRIÉTÉS DES LANGAGES D'I.B.

Pour supporter efficacement les quatre tâches d'I.B. présentées au point 2.2.1, ainsi que les besoins d'organisation et de réutilisation des spécifications, un langage d'I.B. devrait réunir les propriétés suivantes [Dub92, Dub93b, Hab90] :

a) Expressivité

Cette qualité est sans doute la plus importante. Elle implique l'existence d'une correspondance naturelle entre les concepts du monde réel et ceux du langage. En aucun cas, la rédaction des spécifications des besoins ne doit être perçue comme une tâche fastidieuse de codage.

Rappelons que les spécifications ne décrivent pas seulement le comportement de systèmes informatiques mais également celui de systèmes composites (pouvant contenir des procédures manuelles par exemple). Le langage doit donc être suffisamment souple pour permettre à la

fois des considérations informatiques et la description "naturelle" (sans biais informatique) des entités du monde réel.

L'expressivité concerne les tâches d'élicitation et de validation et a un impact direct sur la compréhensibilité du client.

b) Structuration

Le cahier des charges est souvent volumineux et des interactions complexes existent entre les différentes parties de descriptions. Pour faciliter la tâche de modélisation, le langage doit permettre la décomposition des spécifications en unités gérables et la définition précise des relations entre ces unités.

Cette propriété est liée à la qualité de "bonne structure". Elle est essentielle pour la réutilisation et la maintenance du cahier des charges.

c) Formalisation

Le langage devrait avoir une sémantique formelle, dans le sens où des structures mathématiques sont utilisées pour représenter des aspects du monde réel [Jun91a]. Alors seulement, on dispose de règles rigoureuses d'interprétation qui garantissent la précision et l'absence d'ambiguïtés. Plus généralement, un tel langage supporte toute une série de raisonnements formels. Citons entre autres le contrôle de la cohérence et de la complétude, la dérivation automatique de nouvelles spécifications à des fins de prototypage ou de réutilisation.

Le langage doit être conçu de telle sorte que son caractère formel n'entre pas en contradiction avec les deux premières propriétés. Beaucoup de langages se basent sur la logique du premier ordre pour ses qualités expressives, et possèdent une structure algébrique pour ses possibilités de structuration (paramétrisation, héritage,...).

Il reste cependant un problème à concilier formalisation et compréhensibilité. Ceci reste un des tâches les plus difficiles dans la conception d'un langage d'I.B.

2.4.2. SÉMANTIQUES OPÉRATIONNELLE ET DÉCLARATIVE

La phase d'I.B. consiste à relever quels sont les aspects pertinents du monde réel et leurs propriétés. Ces propriétés ne sont en général pas de nature algorithmique, c'est-à-dire qu'elles ne peuvent pas être exprimées en termes de transformations appliquées sur des arguments pour produire des résultats [Dub92]. C'est pourquoi un langage formel d'I.B. doit supporter une *sémantique déclarative*, qui ne fait pas référence à un modèle informatique, afin de respecter la propriété d'expressivité.

Ceci n'empêche pas un tel langage d'avoir également une *sémantique opérationnelle* (basée sur un modèle informatique). Dans ce cas, le langage pourrait (du moins théoriquement) être

exécuté sur une machine appropriée mais cela ne signifie pas que le langage est un langage de programmation. De même, un langage qui possède une sémantique déclarative n'est pas nécessairement un langage d'I.B.

Comme nous l'avons vu plus haut (point 2.3.4), la différence entre les spécification des besoins et le programme qui réalise ces spécifications ne réside pas dans leur caractère exécutable ou non, mais repose sur d'autres critères dont leur rôle dans le cycle de vie.

2.4.3. APPROCHES ORIENTÉES-OBJETS

a) Approches fonctionnelle vs O-O

Dans les approches classiques, le cahier des charges est structuré, hiérarchisé, selon un critère fonctionnel ; cela signifie que les modules de spécifications correspondent à des fonctions identifiées dans le système. Par exemple, dans une entreprise CIM, on peut identifier les fonctions CAO, planification et contrôle de production (PCP), PAO, etc. ; cette dernière étant elle-même décomposée selon les activités assemblage, transport, stockage, emballage, etc...

Le paradigme O-O, apparu dans le champ de la programmation puis dans celui de la conception, s'étend aujourd'hui peu à peu à l'I.B. Il avance un autre critère de décomposition, réalisant une meilleure cohésion des données. Pour l'exemple ci-dessus, on peut proposer une architecture alternative, basée sur une perspective organisationnelle et non plus fonctionnelle. Cette perspective fait apparaître des unités de production à différents niveaux d'abstraction. Chacune de ces unités peut comprendre des composants CAO, PCP, PAO, etc. Ainsi, on peut identifier dans l'entreprise CIM un atelier d'usinage comprenant un contrôleur (PCP), un véhicule, une fraiseuse, un tour, etc. (PAO). A son tour, le véhicule peut être considéré comme une unité de production composée d'un ordinateur (PCP) et d'un palettiseur (PAO). La différence avec la décomposition fonctionnelle tient au fait que les fonctions ne sont plus regroupées mais au contraire distribuées entre les unités de production qui les utilisent.

(L'exemple cité est tiré de [Dub93a].)

b) Concepts O-O

En résumé, nous dirons que les approches orientées-objets organisent les spécifications d'un système comme une collection de descriptions d'objets [Jun91a]. Ces descriptions concernent à la fois les aspects statiques et les aspects dynamiques des objets.

Ce dernier point est particulièrement important. Le paradigme O-O a permis de réconcilier en un seul langage les aspects structurels et comportementaux d'un système, là où on utilisait auparavant deux formalismes différents et pas toujours faciles à intégrer (citons notamment : modèle E-A, réseaux sémantiques et *frames* pour la partie statique ; algèbre de processus, réseaux de Pétri et logique temporelle pour la partie dynamique).

Un autre principe important de l'approche O-O est l'*encapsulation*. Un objet possède un état interne qui peut être observé et changé de l'extérieur uniquement via l'*interface* de cet objet. Une bonne spécification est une spécification qui réalise des interfaces "propres" et uniformes entre les composants, notamment entre ceux faisant partie de l'environnement et ceux qui seront informatisés. Suivant cette perspective, les descriptions d'objet deviennent des unités de spécification, permettant une certaine structuration ou modularité [Jun91a].

D'autres notions O-O (attribut, action, héritage,...) apparaîtront au chapitre suivant qui introduit le langage ALBERT et dans la partie II consacrée au langage OBLOG.

3. LE LANGAGE ALBERT

La définition du langage est inspirée de [Dub93a], [Dub93b] et [Dub93c].

3.1. INTRODUCTION

ALBERT est un langage formel, conçu pour exprimer les besoins concernant des systèmes composites, y compris les systèmes en temps réel. Plus précisément, ce langage supporte l'expression :
- des propriétés à propos des entités du monde réel ;
- d'exigences de performance ;
- de propriétés de visibilité et de fiabilité.

ALBERT est un langage "orienté-agent". Ce concept d'agent, qui peut être vu comme une spécialisation du concept d'objet, sert à modéliser les différents composants du système à l'étude. Dans les systèmes composites, un agent peut représenter aussi bien une procédure manuelle qu'une installation matérielle ou un composant logiciel. L'idée est de structurer le cahier des charges selon les responsabilités contractuelles associées à chacun de ces agents.

Le langage sera illustré à l'aide d'un exemple d'application introduit à la section 3.2. La section 3.3 présente les différents concepts du langage (*modèle-produit*). A la section 3.4, nous verrons une méthode d'utilisation du langage qui favorise une élaboration incrémentale du cahier des charges (*modèle-procédé*). Enfin, la section 3.5 est consacrée aux hypothèses d'extension du langage utilisées dans le cadre de ce mémoire.

ALBERT est l'acronyme de "*Agent-oriented Language for Building and Elicitating Requirements for real-Time systems*".

3.2. EXEMPLE D'APPLICATION

L'exemple que nous allons considérer fait partie des applications CIM *(Computer Integrated Manufacturing)*, également appelées applications *productiques*. Ce domaine d'application

s'avère fort intéressant pour la rédaction de spécifications en ALBERT. En effet, il manque actuellement de méthodes adaptées à ces applications, c'est-à-dire capables de supporter leurs besoins spécifiques en informations.

Les systèmes d'information CIM se caractérisent par [Dub93a] :
- la masse d'informations à gérer et la nature souvent très structurée des informations (exemple : le *layout* d'un produit) ;
- les différents niveaux d'abstraction dans l'information (exemple : l'information nécessaire au contrôle d'une usine est l'agrégation d'informations résultant du contrôle d'équipements individuels faisant partie de l'usine) ;
- le caractère temporel de l'information (exemple : le contrôle d'une machine de production doit se baser sur des messages délivrés en temps réel).

Grâce à sa richesse d'expression et de structuration, ALBERT peut rencontrer ces exigences, comme semble l'indiquer une expérimentation en cours. Dans la suite, nous illustrerons le langage en nous inspirant d'une version simplifiée de l'application étudiée dans le cadre de cette expérimentation.

Il s'agit d'un atelier de production qui transforme des cylindres métalliques en diverses pièces : boulons, rivets, vis, etc... Un *magasin* stocke les matières premières (cylindres) et les produits finis. Un *robot* industriel et un *véhicule* assurent le transport des pièces entre le magasin et les différentes machines de l'atelier (un *poste de bridage* et deux *machines-outils*). L'ensemble de la production est automatisée à l'aide d'un *contrôleur* (c'est-à-dire un logiciel). Un *gestionnaire* approvisionne le stock, lance les ordres de production et réceptionne les produits finis. Enfin, la fabrication d'une pièce ne peut prendre plus de 20 minutes.

(L'énoncé précis du cas ainsi que sa spécification complète en ALBERT est développé en annexe.)

3.3. LE MODÈLE-PRODUIT

Le terme "modèle-produit" fait référence aux différents concepts proposés par le langage ALBERT pour rédiger la spécification d'un système.

L'agent est le concept central du langage et forme une unité de spécification. Spécifier un agent revient à définir un ensemble de *vies* modélisant tous les comportements possibles de cet agent. Concrètement, la spécification d'un agent comprend deux parties :
- les *déclarations*, qui décrivent les propriétés statiques de l'agent, introduisant ainsi le vocabulaire de l'application considérée (point 3.3.1) ;
- et les *contraintes*, qui décrivent les propriétés dynamiques de l'agent en identifiant ses comportements possibles et en excluant les autres (point 3.3.2).

Par analogie avec les concepts O-O, nous dirons que tous les agents qui ont la même spécification (soit les mêmes déclarations et les mêmes contraintes) forment une *classe d'agents*. ALBERT fournit un mécanisme d'identification interne qui permet de distinguer entre eux les différentes *instances* d'une même classe d'agents.

Même si le langage ne prévoit pas explicitement cette distinction, il est important de lire les spécifications d'un agent à deux niveaux :
- au niveau agent lorsqu'on considère l'agent pris isolément, comme un système fermé ;
- au niveau société lorsqu'on tient compte des interactions de l'agent avec les autres. La *société* désigne l'ensemble des agents du système.

3.3.1. DÉCLARATIONS

La partie déclarations d'un agent a pour but de décrire ses *attributs* (qui forment la *structure d'état* de l'agent) ainsi que les *actions* qui peuvent influencer la vie de cet agent. Les déclarations associées à l'agent `Atelier` sont présentées à la figure 3.

a) Les attributs

L'état d'un agent est structuré en un ensemble d'attributs, qui représentent des caractéristiques de l'agent. L'état d'un agent à un temps quelconque est défini par la valeur de tous ses attributs. A la figure 3, on peut lire que l'atelier est caractérisé par un statut (occupé ou non), le stock et sa capacité, le nombre de commandes traitées et le nombre de cylindres disponibles dans le stock.

Les attributs peuvent être simples (individus) ou multivalués (populations), variables ou constants. De plus, chaque attribut est typé. On distingue les *types élémentaires* des *types complexes*. Ces derniers sont construits par le spécifieur en appliquant des constructeurs de type prédéfinis (ensemble, liste, produit cartésien,...) à des types élémentaires ou non. Parmi les types élémentaires, on distingue :
- des types définis par le spécifieur. Celui-ci pourrait par exemple définir un type `ID_PIECE`, sans devoir expliciter la nature de l'identifiant d'une pièce (code numérique ou alphanumérique, libellé,...) ;
- des types prédéfinis (`STRING`, `BOOLEAN`, `INTEGER`,...) ;
- des types d'identifiant d'agent. Un type est automatiquement associé à chaque classe d'agents, permettant ainsi d'identifier les instances de la classe. Par exemple, chaque agent `Machine_outil` a un indentifiant de type `MACHINE_OUTIL`. Lorsqu'un agent est unique (comme par exemple, l'agent `Gestionnaire`), une constante est automatiquement définie pour désigner l'identifiant de cet agent (`gest` dans ce cas-ci). A l'intérieur de la spécification d'un agent, la constante `self` désigne l'identifiant de l'agent en question.

Le langage comporte une syntaxe graphique pour exprimer les déclarations. Les attributs sont représentés par une boîte comprenant le nom de l'attribut et son type encadré par un rectangle. Lorsque l'attribut est individuel, la boîte est en pointillé.

b) Les actions

Dans la terminologie ALBERT, le mot "action" est utilisé pour désigner des événements qui ont lieu dans la vie d'un agent, qu'ils aient ou non un effet sur l'état (la distinction est faite par certains langages). Les actions peuvent avoir des *arguments*. L'action `Produit` (figure 3), par exemple, a lieu chaque fois que le gestionnaire lance l'ordre de production d'une pièce. Le type de pièce (vis, boulon,...) à produire est un argument de cette action.

Les actions sont représentées par une boîte comprenant le nom de l'action et un ovale indiquant le type de l'argument éventuel.

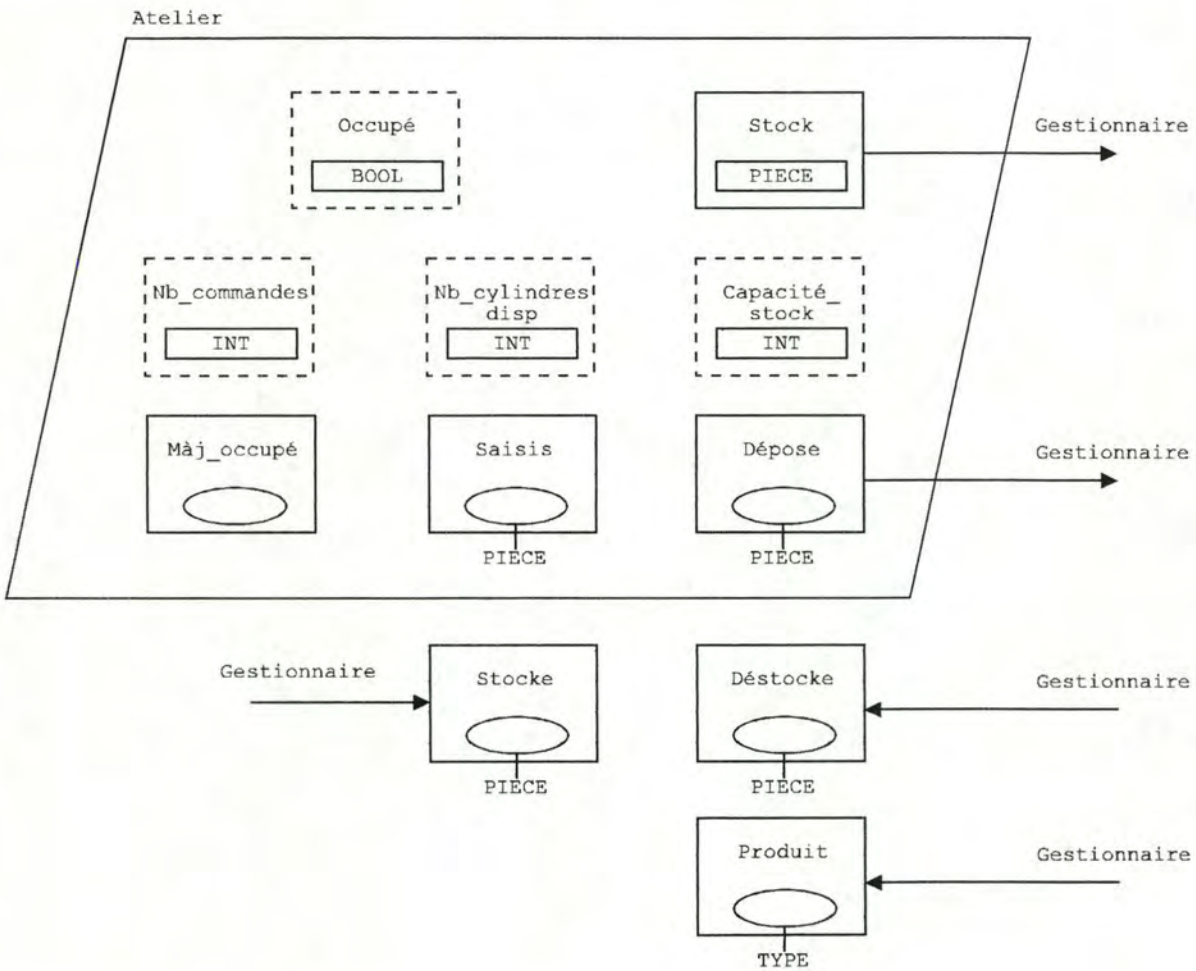


Figure 3 - Déclarations associées à l'agent Atelier

c) Visibilité

Le diagramme inclut également des notations graphiques permettant d'exprimer les relations de visibilité entre l'agent et l'extérieur (mécanismes d'*importation* et d'*exportation*).

L'information (attributs ou actions) à l'intérieur du parallélogramme est sous le contrôle de l'agent décrit (`Atelier`), tandis que l'information en-dehors du parallélogramme est importée d'autres agents de la même société. On peut lire par exemple que l'atelier a l'initiative des actions `Saisis` et `Dépose` (actions internes) tandis que c'est un agent extérieur, en l'occurrence le gestionnaire, qui a l'initiative des actions `Stocke` et `Déstocke` (actions externes). L'atelier pourrait également importer un attribut appartenant à un autre agent.

A l'intérieur du parallélogramme, les boîtes d'où part une flèche indiquent des informations (attributs ou actions) qui sont exportées, c'est-à-dire qui sont visibles de l'extérieur (ou uniquement d'un agent si la flèche est étiquetée par un nom d'agent). Par exemple, le gestionnaire peut avoir connaissance de l'attribut `Stock` et des occurrences de l'action `Dépose`.

Remarque : l'importation et l'exportation sont des propriétés statiques. Qu'une information soit potentiellement visible ne veut pas dire qu'on peut en prendre connaissance à tout moment. La *perception* et la *publicité* sont les contreparties dynamiques de l'importation et de l'exportation et permettent au spécifieur de contrôler plus finement l'échange d'information entre les agents (voir point 3.3.2).

3.3.2. CONTRAINTES

Les contraintes servent à restreindre l'ensemble généralement infini des vies possibles d'un agent, que l'on peut déduire à la lecture des déclarations.

Jusqu'à présent, le terme de "vie" d'un agent a été utilisé de manière intuitive. On définit plus précisément une *vie* comme une séquence, finie ou infinie, de *changements* et d'*états*. Chaque changement est un ensemble d'actions simultanées qui fait passer l'agent d'un état à un autre. L'absence d'action est également considérée comme un changement ; dans ce cas, le changement représente simplement le temps qui passe. Les états sont étiquetés par une valeur de temps qui évolue tout au long de la vie. La figure 4 présente une vie possible de l'agent `Atelier`.

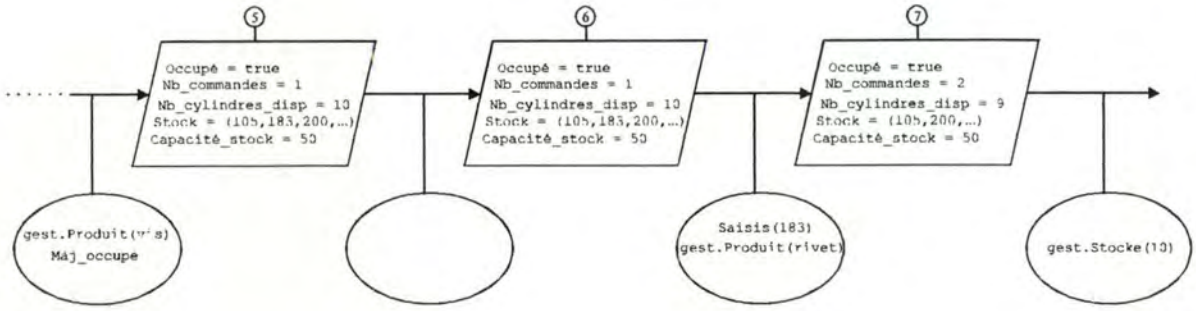


Figure 4 - Une vie possible de l'agent Atelier

Notons que la valeur d'un état à un temps donné dans une certaine vie peut toujours être dérivée à partir de l'état initial et de tous les changements apparus jusque là.

Afin d'offrir une certaine guidance méthodologique au spécifieur, les contraintes sont classées en sept catégories :
- contraintes sur l'état (**`State Behaviour`**) ;
- initialisations (**`Init`**) ;
- effets des actions (**`Effects of Actions`**) ;
- engagements (**`Commitments`**) ;
- responsabilité (**`Responsibility`**) ;
- perception (**`Perception`**) ;
- publicité (**`Publicity`**).

L'expression des contraintes est purement textuelle.

La figure 5 introduit les contraintes associées à l'agent `Atelier` et complète les déclarations de la figure 3.

Atelier

State Behaviour

```
Occupé ⇔ Nb_commandes > 0
p ∈ Stock ⇒ ◇ ¬(p ∈ Stock)
```

Init

```
Occupé = false
Stock = {}
Capacité_stock = 50
Nb_commandes = 0
Nb_cylindres_disp = 0
```

Effects of Actions

```
gest.Produit(t):
     Nb_commandes = Nb_commandes + 1
     Nb_cylindres_disp = Nb_cylindres_disp - 1
Màj_occupé: Occupé = ¬Occupé
Saisis(p): Stock = Remove(Stock, p)
Dépose(p):
     Stock = Add(Stock, p)
     Nb_commandes = Nb_commandes - 1
gest.Stocke(p):
     Stock = Add(Stock, p)
     Nb_cylindres_disp = Nb_cylindres_disp + 1
gest.Déstocke(p):
     Stock = Remove(Stock, p)
```

Commitments

```
gest.Produit(t) → ◇ ≤ 20min Saisis(p1) ; Dépose(p2)
     with p1.Type = cylindre ∧ p2.Type = t
```

Responsibility

```
F(Saisis(p) / ¬(p ∈ Stock))
```

Perception

```
X(gest.Produit(t) / Nb_cylindres_disp > 0 and t <> cylindre)
X(gest.Stocke(p) /
     ¬Occupé ∧ ¬(Card(Stock) = Capacité_stock) ∧ p.Type = cylindre)
X(gest.Déstocke(p) / ¬Occupé ∧ p ∈ Stock ∧ p.Type <> cylindre)
```

Publicity

```
X(Stock.gest / ¬Occupé)
```

Figure 5 - Contraintes associées à l'agent Atelier

a) Contraintes sur l'état

Cette rubrique regroupe des propriétés qui sont vraies dans tous les états et les propriétés reliant entre eux les états d'un agent.

• Considérons d'abord les contraintes qui sont vraies dans toutes les états, habituellement désignées *invariants*. Ces contraintes sont formées selon les règles usuelles de la logique du premier ordre fortement typée, au moyen des connecteurs logiques ¬ (non), ∧ (et), ∨ (ou), ⇒ (implique), ⇔ (si et seulement si), ∀ (pour tout), ∃ (il existe). La quantification universelle globale des formules peut être omise, la règle étant qu'une variable qui n'est pas sous la portée d'un quantificateur est universellement quantifiée à l'extérieur de la formule.

• A côté des invariants, on trouve des contraintes sur l'évolution du système. Pour écrire de telles contraintes il faut être capable de faire référence, dans une formule, à plusieurs états. Ceci est rendu possible par l'utilisation de connecteurs temporels qui préfixent les propriétés. La table suivante introduit ces opérateurs (inspirés de la logique temporelle) et leur signification intuitive [Pet92] :

$\Diamond\ \alpha$	α est vrai dans l'état courant ou sera vrai dans un état futur (quel qu'il soit)
$\blacklozenge\ \alpha$	α est vrai dans l'état actuel ou était vrai dans un état passé (quel qu'il soit)
$\Box\ \alpha$	α est vrai dans l'état courant et le restera dans tous les états futurs
$\blacksquare\ \alpha$	α est vrai dans l'état actuel et l'était dans tous les états passés
$\alpha\ \mathcal{U}\ \beta$	α est vrai dans l'état actuel et sera vrai dans les états futurs jusqu'au prochain état lors duquel β sera vrai (pour toujours si β ne devient jamais vrai)
$\alpha\ \mathcal{S}\ \beta$	α est vrai dans l'état actuel et depuis le dernier état passé dans lequel β était vrai (depuis toujours si β n'a jamais été vrai)

A la figure 5, la deuxième contrainte sur l'état signifie qu'une pièce ne peut pas rester indéfiniment dans le stock. De plus, il est possible d'exprimer des délais en indiçant les connecteurs temporels d'une période de temps. Par exemple, si on modifie cette dernière contrainte de la manière suivante :

$$\texttt{p} \in \texttt{Stock} \Rightarrow \Diamond_{\leq 1\ \texttt{day}}\ \neg(\texttt{p} \in \texttt{Stock}),$$

on précise davantage la contrainte en exprimant le fait qu'une pièce ne peut pas rester plus d'un jour dans le stock.

Notons bien que les invariants et les contraintes d'évolution (parfois appelées *contraintes statiques* et *contraintes dynamiques* [Jun91a]) restreignent implicitement le comportement de l'agent en ce sens que certains changements ne sont pas permis.

b) Initialisations

Les clauses sous la rubrique **`Init`** servent à construire l'état initial en affectant des valeurs aux attributs. Par exemple, le nombre de commandes de l'atelier est initialisé à 0. Le membre de droite d'une clause d'initialisation ne peut représenter qu'une valeur constante.

c) Effets des actions

Cette rubrique décrit les effets des actions qui peuvent modifier les états (voir figure 5 comment, par exemple, une occurrence de l'action `Dépose(p)` modifie le stock).

L'effet d'une action consiste en une propriété de l'état qui suit l'occurrence de l'action. Elle décrit la valeur d'un attribut en fonction des arguments de l'action et des états précédents dans la vie (l'état précédent si ce n'est pas précisé par un opérateur temporel). La dernière clause de la rubrique **`Effects`**, exprime que l'action `Déstocke` issue par l'agent externe `Gestionnaire` est d'enlever une pièce du stock de l'atelier. Le membre de droite de l'équation fait référence à la valeur que prends l'attribut suite à l'action tandis que le membre de gauche est évalué dans l'état précédent l'occurrence de l'action.

Remarques :
- dans la description des effets des actions, on utilise une règle *(frame rule)* implicite disant que les attributs pour lesquels aucun effet n'est spécifié gardent leur valeur inchangée dans l'état suivant l'apparition d'un changement;
- un changement ne peut contenir deux occurrences d'action simultanées que si leurs effets ne portent pas sur les mêmes attributs ;
- considérons l'invariant de la figure 5 qui signifie que l'atelier est occupé dès qu'il traite au moins une commande. Etant donné les valeurs initiales des attributs `Occupé` et `Nb_commandes`, cet invariant implique que la première occurrence de `Produit` (qui incrémente le nombre de commandes) ne peut se produire que si elle est simultanée à une occurrence de `Màj_occupé` (qui change la valeur de `Occupé`).

d) Engagements

Cette rubrique concerne les relations de *causalité* qui peuvent exister entre différentes occurrences d'action.

Le symbole "→", qui ne doit pas être confondu avec le symbole logique habituel "⇒", permet d'exprimer une certaine forme d'*enchaînement (entailment)* entre les actions. Dans le cas de l'atelier, un exemple de causalité existe entre les actions `Produit`, `Saisis` et `Dépose`. Elle repose sur la nécessité d'avoir, dans l'ordre, une occurrence unique de `Saisis` et une occurrence unique de `Dépose` en réponse à chaque occurrence de `Produit` (voir figure 5).

Le symbole "→" peut être quantifié par un opérateur temporel pour exprimer des contraintes de performance. Par exemple, le symbole "$\rightarrow_{\diamond \leq 20\,min}$" dans l'engagement de la figure 5 signifie que l'occurrence de `Dépose` doit avoir lieu dans les 10 minutes après l'occurrence de `Produit`.

Le membre de droite d'un engagement ne peut contenir que des actions produites par l'agent (actions internes). Les membres de droite et de gauche peuvent contenir une ou plusieurs actions, qui peuvent être composées entre elles de trois façons :

- Act_1 ; Act_2 signifie "une occurrence de Act_1 suivie d'une occurrence de Act_2" ;
- Act_1 || Act_2 signifie "une occurrence de Act_1 et une occurrence de Act_2 (dans un ordre quelconque)" ;
- $Act_1 \oplus Act_2$ signifie "une occurrence de Act_1 ou une occurrence de Act_2 (ou exclusif)".

e) Responsabilité

Sous cette rubrique est précisé le rôle de l'agent vis-à-vis des occurrences de ses propres actions. A cette fin, on introduit une autre extension à la logique du premier ordre, inspirée de la *logique déontique* (voir [Dub91b]). Trois connecteurs spécifiques permettent d'exprimer des *obligations*, des *préventions* et des *obligations exclusives* :
- O(*Act* / *cond*) exprime que l'action *Act* doit avoir lieu lorsque, dans l'état courant, la condition *cond* est vérifiée ;
- F(*Act* / *cond*) exprime que l'action *Act* est interdite lorsque, dans l'état courant, la condition *cond* est vérifiée ;
- X(*Act* / *cond*) est un raccourci pour exprimer la combinaison des formules O(*Act* / *cond*) et F(*Act* / *cond*). Une telle combinaison est appelée *obligation exclusive*.

A titre d'exemple, considérons la clause de responsabilité de la figure 5. Elle exprime qu'il est interdit de saisir une pièce qui n'est pas dans le stock.

Signalons enfin que, par défaut, toutes les actions sont permises.

f) Perception

Ces mêmes connecteurs servent également à restreindre la visibilité de l'agent ; en d'autres termes, ils permettent d'exprimer sous quelles conditions l'agent perçoit les attributs ou les occurrences d'actions d'autres agents. Par exemple, on peut lire à la figure 5 que l'atelier tient compte des ordres de production issus du gestionnaire si et seulement si (obligation exclusive) il reste un cylindre disponible dans le stock et que l'ordre de production a un sens (vouloir produire un cylindre n'a pas de sens puisqu'il s'agit d'une matière première).

La règle par défaut est que tous les éléments importés peuvent être perçus.

g) Publicité

De même, on utilisera les connecteurs déontiques pour définir sous quelles conditions un agent offre une perception de ses propres attributs et occurrences d'action au reste de la société. Par exemple, on peut lire à la figure 5 que l'atelier laisse voir l'état du stock au gestionnaire lorsque il n'est pas occupé (et uniquement dans ce cas).

3.4. LE MODÈLE-PROCÉDÉ

Rappelons que l'I.B. n'est pas une simple activité de transcription des désirs des clients. Il s'agit de décrire de manière très précise des systèmes composites, c'est-à-dire composés de plusieurs agents, de nature informatique ou non. Le nombre d'agents et, surtout, la complexité des interactions entre ces agents font que le cahier des charges devient vite très complexe.

Pour toutes ces raisons, le langage ALBERT propose un "mode d'emploi", une méthode d'utilisation du langage. Cette méthode repose sur une élaboration incrémentale du cahier des charges. Pour mieux l'expliquer, nous avons distingué deux types de progression : l'une *verticale* et l'autre *horizontale*.

a) Progression verticale

Il s'agit d'identifier petit à petit tous les agents du problème posé. Plus précisément, la stratégie proposée dans [Dub93c] consiste à :
- exprimer les objectifs du système en le considérant comme une boîte noire ;
- identifier des sous-systèmes et attacher à chacun d'eux des responsabilités de telle sorte que ce raffinement préserve le comportement original ;
- appliquer récursivement la deuxième étape jusqu'à identifier des sous-systèmes "terminaux", c'est-à-dire des composants dont les implémenteurs sont d'accord d'implémenter les propriétés.

Cette décomposition fait apparaître une hiérarchie où on distingue les *agents individuels*, ou terminaux, et les *agents complexes*, correspondant aux noeuds non-terminaux de la hiérarchie. L'ensemble des spécifications des agents individuels doit rencontrer les objectifs du système identifiés lors de la première étape, c'est-à-dire respecter les spécifications de l'agent complexe qui se trouve au sommet de la hiérarchie.

Pour notre exemple, on peut envisager trois étapes (voir figure 6) :
- 1ère étape : l'entreprise de fabrication de pièces est considérée comme un seul agent complexe ;
- 2ème étape : l'atelier et le gestionnaire sont deux agents faisant partie de l'entreprise (les spécifications de l'atelier présentées à la section précédente se situent à cette étape) ;
- 3ème étape : si le gestionnaire est un agent individuel, l'atelier par contre est encore décomposable en agents individuels (magasin, robot, véhicule,...).

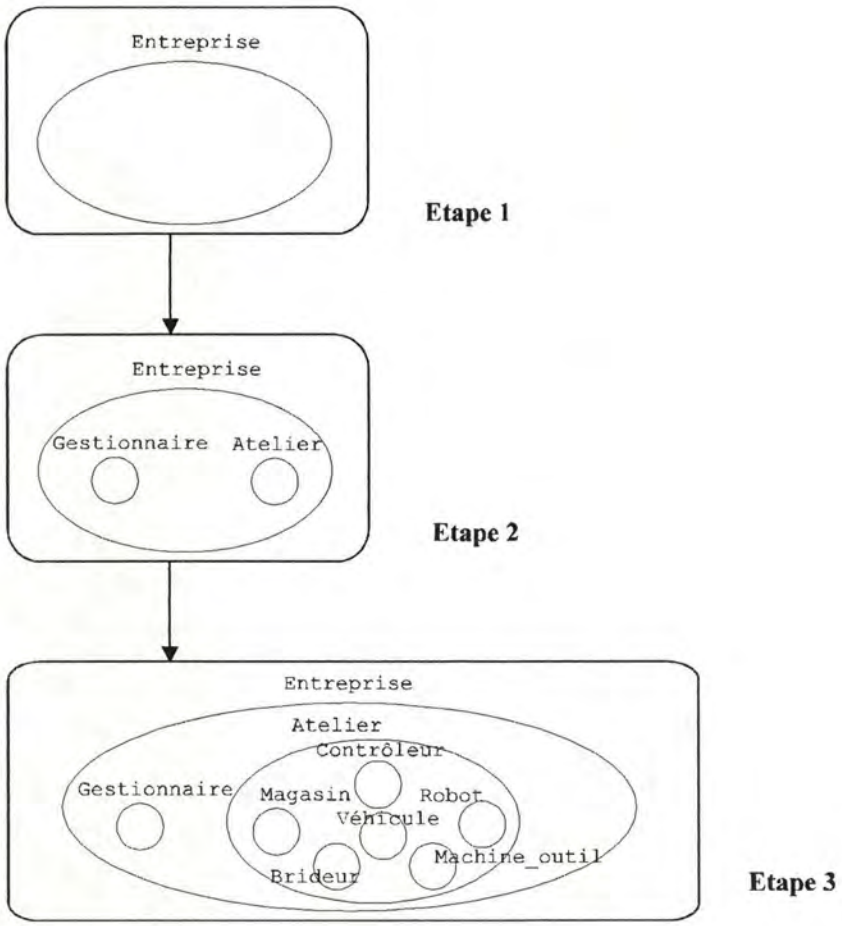


Figure 6 - Progression verticale du cahier des charges (adapté de [Dub93c])

Dans la version finale du cahier des charges, les responsabilités attachées aux différents agents (logiciels ou non) doivent apparaître clairement et précisément puisqu'elles forment une obligation contractuelle pour les implémenteurs (pas seulement les concepteurs de logiciels).

b) Progression horizontale

On se situe ici à un certain niveau de décomposition du problème, c'est-à-dire à une certaine étape de la progression verticale. Si les agents identifiés à cette étape s'avèrent déjà fort complexes, on peut envisager une spécification en trois temps.

Dans un premier temps, le spécifieur se simplifie la tâche en posant deux hypothèses :
- *omniscience :* la visibilité entre les agents est totale, ce qui veut dire qu'un agent peut prendre connaissance librement des caractéristiques ou du comportement d'un autre agent ;
- *fiabilité :* on considère que les agents se comportent de manière idéale. S'il s'agit d'une personne, on suppose qu'elle ne fait aucune erreur ; s'il s'agit d'une machine, on suppose qu'elle ne tombe jamais en panne, etc.

Dans les deux temps suivants, le spécifieur lève successivement une hypothèse et puis l'autre.

(La spécification complète de l'atelier, en annexe, est développée selon cette double progression verticale et horizontale. Dans la suite, nous ferons souvent référence à la version terminale de la spécification du véhicule, présentée à la section A.7)

3.5. HYPOTHÈSES SUR LE LANGAGE

Actuellement, le langage ALBERT est encore un langage à l'étude. Si le noyau sémantique est rigoureusement défini, par contre, la syntaxe a quelque peu été délaissée et le "langage utilisateur" reste relativement pauvre. Nous avons donc été amené à poser quelques hypothèses pour préciser, voire étendre le langage.

3.5.1. SYNTAXE

a) Philosophie ASCII

Pour des raisons évidentes de limitations typographiques, nous avons été amené à définir et à utiliser des équivalents plus "textuels" aux symboles graphiques du langage.

b) Référence ICARUS

Pour définir les différents types, prédicats et fonctions considérés comme prédéfinis dans le langage, nous nous sommes inspiré du document [ICARUS90].

3.5.2. LANGAGE UTILISATEUR

Par langage utilisateur, nous entendons tout ce qui n'ajoute rien à la sémantique du langage mais procure des facilités d'utilisation du langage au spécifieur. A cet égard, nous avons défini ce qui suit :

a) Attributs dérivés

Un attribut peut être déclaré comme *attribut dérivé*. La valeur d'un attribut dérivé est calculée, selon une formule, à partir de celles d'un ou plusieurs autres attributs. (Cette extension était déjà utilisée dans [Pet92].) Les attributs dérivés, ainsi que leur formule de dérivation, feront l'objet d'une rubrique séparée dans les déclarations (**`Derived Attributes`**).

Syntaxe :

```
Attr : Type
     = terme .
```

En outre, il est parfois intéressant d'associer plusieurs formules de dérivation à un même attribut dérivé. Le choix de la formule dépend d'une précondition.

Syntaxe :

```
Attr : Type
     (cond1) => Attr = terme1
     (cond2) => Attr = terme2 .
```

b) Préconditions d'effets

L'effet d'une action peut être soumis à une condition évaluée dans l'état précédent l'occurrence de l'action.

Syntaxe :

```
Act: (cond) => Attr = terme .
```

c) Processus

Lorsqu'un même schéma d'actions se retrouve dans plusieurs engagements, on pourra le définir comme un *processus (process)* et le réutiliser en plusieurs endroits, mais toujours dans les engagements. (Ce concept de processus est celui défini dans [Hoa85].)

Syntaxe (exemple) :

```
Act1 -> Act2 ; P
     where process P = Act3 ; Act4
```

est équivalent à

```
Act1 -> Act2 ; Act3 ; Act4.
```

d) Alternatives

Pour plus de souplesse, on pourra utiliser la structure "*if-then-else*" dans les engagements.

Exemple :

```
Act1 -> if cond then Act2 else Act3 ; Act4.
```

3.5.3. SÉMANTIQUE

Nous ne ferons que préciser un point de sémantique qui n'est pas encore fixé dans la définition actuelle du langage.

a) Actions liées par un engagement

Les actions présentes dans le membre de droite d'un ou plusieurs engagements deviennent liées à ces engagements et ne peuvent se produire en-dehors.

Exemple :

Act1 -> Act2 ; Act3 ; Act4

entraîne implicitement que les actions *Act1*, *Act2* et Act3 ne peuvent se produire que dans le cadre de l'engagement causé par *Act1*.

3.5.4. APPLICATION AU CAS

La figure 7 présente à nouveau la spécification de l'agent `Atelier`, en y intégrant cette fois nos hypothèses (remarquer en particulier comment l'invariant portant sur l'attribut `Occupé` a été remplacé par un attribut dérivé).

```
Atelier

Declarations

Attributes

Stock : SET[PIECE] -> Gestionnaire
Capacité_stock : INT
Nb_commandes, Nb_cylindres_disp : INT

Derived Attributes

Occupé : BOOL = (Nb_commandes > 0)

Actions

Saisis(PIECE)
Dépose(PIECE) -> Gestionnaire

Stocke(PIECE), Déstocke(PIECE) <- Gestionnaire
Produit(TYPE) <- Gestionnaire
```

```
Constraints

State Behaviour

In(Stock, p) => sometimef(not In(Stock, p))

Init

Stock = {}
Capacité_stock = 50
Nb_commandes = 0
Nb_cylindres_disp = 0

Effects of Actions

gest.Produit(t):
     Nb_commandes = Nb_commandes + 1
     Nb_cylindres_disp = Nb_cylindres_disp - 1
Saisis(p): Stock = Remove(Stock, p)
Dépose(p):
     Stock = Add(Stock, p)
     Nb_commandes = Nb_commandes - 1
gest.Stocke(p):
     Stock = Add(Stock, p)
     Nb_cylindres_disp = Nb_cylindres_disp + 1
gest.Déstocke(p): Stock = Remove(Stock, p)

Commitments

gest.Produit(t) -> sometimef(<= 20 min) (Saisis(p1); Dépose(p2))
     with p1.Type = cylindre and p2.Type = t

Responsibility

F(Saisis(p) / not In(Stock, p))

Perception

X(gest.Produit(t) / Nb_cylindres_disp > 0 and t <> cylindre)
X(gest.Stocke(p) /
     not Occupé and not Card(Stock) = Capacité_stock and p.Type = cylindre)
X(gest.Déstocke(p) / not Occupé and In(Stock, p) and p.Type <> cylindre)

Publicity

X(Stock.gest / not Occupé)
```

Figure 7 - Spécification de l'atelier (2ème version)

PARTIE II

OBLOG

4. L'APPROCHE OBLOG

4.1. INTRODUCTION

L'informatique est une discipline jeune et en constante évolution. La construction progressive d'un modèle du cycle de vie d'une application informatique reflète bien cette évolution.

A mesure que la taille des programmes grandissait, la nécessité est apparue de mettre en place une phase d'analyse du problème et de conception de la solution, bien distinctes de la phase de programmation proprement dite. Par la suite, c'est la maintenance qui a attiré l'attention, à un moment où on s'est rendu compte que cette activité exigeait une part disproportionnée des budgets informatiques. Enfin, on a montré au cours de la première partie qu'aujourd'hui, on accorde un intérêt croissant à la phase d'analyse des besoins. Cette phase est étendue à la notion de système composite et devient l'objet d'une discipline à part entière : l'ingénierie des besoins.

Cet élargissement du cycle de vie, coûteux et difficile à gérer, est peut-être à l'origine d'une crise du développement de logiciels [ESDI92c]. Signalons entre autres la dispersion des efforts, la distance "sémantique" entre les spécifications et l'application exécutable et le fait que les modifications sont généralement implémentées directement au niveau de la programmation sans être répercutées aux niveaux supérieurs.

L'approche OBLOG a pour ambition de rationaliser le cycle de vie :
- en intégrant les différents formalismes utilisés en un seul langage : OBLOG ;
- en intégrant les différents outils de développement selon le concept IPSE *(Integrated Project Support Environment)* ;
- en offrant un support méthodologique adéquat.

4.1.1. LE LANGAGE

OBLOG *(OBject LOGic)* est un langage destiné à supporter tout le développement d'un système d'information. En conséquence :
- il propose des concepts suffisamment de haut niveau pour pouvoir appréhender les entités du monde réel et supporte un formalisme graphique (diagrammatique) relativement simple qui peut être compris par des personnes non-initiées ;

- il fournit également des concepts de bas niveau, liés à la technologie, permettant de couvrir des détails d'implémentation d'une solution ;
- il possède une sémantique formelle rigoureuse autorisant des vérifications de cohérence et de complétude ;
- enfin, il possède une sémantique opérationnelle permettant la génération automatique du code correspondant aux spécifications (aussi bien du code C, SQL ou MOTIF).

OBLOG est un langage orienté-objet. A ce titre, il supporte les principes O-O habituels (intégration des données et du comportement, techniques d'agrégation et d'héritage,...) mais se distingue par une approche pleinement concurrente : le système d'information est une collection d'objets concurrents qui interagissent entre eux.

On le voit, les objectifs sont ambitieux. Le langage a subi plusieurs évolutions qui ont donné lieu à plusieurs versions. Concrètement, seul un sous-ensemble d'OBLOG, appelé OBLOGkernel, est aujourd'hui entièrement défini et supporté par un atelier logiciel. OBLOGkernel propose les notions de base du langage et ne peut prétendre supporter la phase d'analyse des besoins ni même le développement d'applications complexes. OBLOGkernel est décrit à la section 4.2.

Dans le cadre de ce mémoire, nous utiliserons OBLOGlight, qui sera implémenté dans la prochaine version de l'atelier logiciel. OBLOGlight est décrit à la section 4.3.

4.1.2. L'OUTIL

OBLOG-CASE est l'atelier logiciel *(workbench)* qui supporte le langage OBLOG et permet d'articuler les spécifications OBLOG avec la technologie. Concrètement, OBLOG-CASE permet aux développeurs de :
- spécifier l'application dans l'éditeur graphique ;
- concevoir l'interface utilisateur avec un éditeur de dialogues ;
- construire la base de données à l'aide d'un éditeur de schémas logiques et physiques ;
- vérifier la complétude et la cohérence de l'application ;
- générer le code de l'application ;
- etc...

OBLOG-CASE se veut un système ouvert et adopte l'approche IPSE, offrant une intégration verticale (intégration de plusieurs outils pour les différentes phases d'un projet) et une intégration horizontale (plusieurs personnes travaillant simultanément sur une vue unifiée des spécifications).

La première version, OBLOG-CASE V1.0[1], supporte le langage OBLOGkernel. Le code d'OBLOG-CASE a été spécifié et généré avec le langage OBLOGkernel, mettant ainsi en oeuvre une stratégie de *bootstrap*. OBLOG-CASE V1.0 est décrit à la section 4.4.

[1] OBLOG-CASE V1.0 est commercialisé seulement depuis avril 1993.

4.1.3. LA MÉTHODE

Après le *langage* et l'*outil*, la *méthode* constitue le troisième axe de l'approche OBLOG, qui se veut une approche "pratique". Un ensemble de règles méthodologiques sont proposées dans [ESDI93] et ont été testées durant le développement d'OBLOG-CASE V1.0. Ces règles concernent tant la gestion d'un projet que la conception de l'architecture d'une application. Leur développement dépasse le cadre de ce mémoire.

4.2. LE LANGAGE OBLOG-KERNEL

La définition du langage est inspirée de [ESDI92b, ESDI93][2] .

Pour illustrer les concepts du langage, nous réutiliserons des éléments du cas de l'atelier de production introduit au chapitre 3. (Cependant, les spécifications OBLOG qui vont suivre ne constituent pas une implémentation des spécifications des besoins de l'atelier.)

4.2.1. OBJETS, CLASSES D'OBJETS ET TYPES DE DONNÉE

a) Objets

Le cas de l'atelier fait apparaître plusieurs concepts : contrôleur, véhicule, machine, etc. Chacun de ces concepts correspond à des *objets* du monde réel. Les objets constituent les "briques" de base pour décrire un système en OBLOG.

Un objet OBLOG peut être décrit de la manière suivante :
- un objet possède une structure (voir point 4.2.2) et un comportement (voir point 4.2.3) ;
- un objet est créé et peut être détruit. Entre sa création et sa destruction, un objet peut accomplir des *actions* ;
- un objet encapsule un *état*. Cet état est représenté par les *attributs* de l'objet ;
- l'état d'un objet ne peut être changé que par les actions de cet objet ;
- les objets interagissent entre eux au sein d'une *communauté*.

Chaque objet possède un *identifiant* interne unique qui ne dépend pas de son état (par opposition aux éventuels identifiants externes correspondants à des attributs ou groupes d'attributs de l'objet).

b) Classes d'objets (et types d'objet)

Une *classe d'objets* représente un ensemble d'objets semblables. Plus précisément, tous les objets qui vérifient les mêmes propriétés structurelles et comportementales font partie d'une

---

[2] Pour des raisons pratiques, nous nous sommes permis certaines libertés au niveau de la syntaxe du langage.

même classe d'objets. Ces propriétés forment la spécification de la classe d'objets. Chaque élément de la classe est appelé objet ou *instance* de la classe.

Chaque classe d'objets définit automatiquement un type objet *(object type)*, de même nom, qui représente le type des identifiants des instances de la classe.

c) Types de donnée

Un *type de donnée (data type)* sert à représenter des entités statiques qui, contrairement aux objets, n'ont pas de comportement. Un nombre entier, par exemple, n'a pas de comportement. On distingue :
- des types de donnée prédéfinis : `int`, `nat` (pour "*natural*"), `string`, etc. ;
- des types de donnée déclarés par l'utilisateur : énumération et liste (liste de types de donnée ou liste de types objet).

d) Communauté d'objets

Une spécification OBLOG consiste en un ensemble de spécifications de classes d'objets. La communauté de toutes les classes d'objets est déclarée dans le *diagramme de communauté* (voir figure 8).

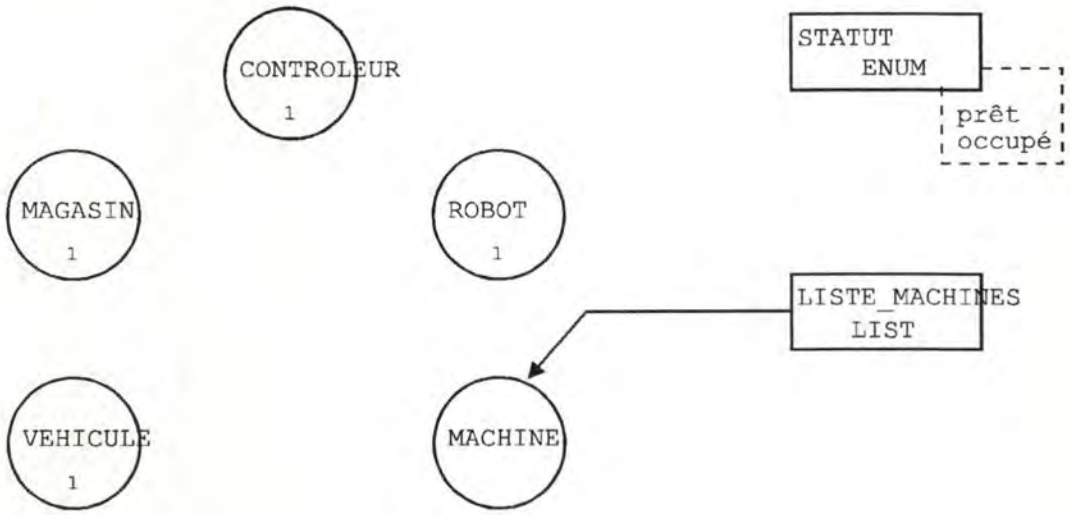


Figure 8 - Diagramme de communauté du cas de l'atelier

Chaque classe d'objets est représentée par un cercle. Lorsque le chiffre "1" apparaît sous le nom de la classe, cela signifie qu'il ne peut exister plus d'une instance dans la classe.
Le diagramme de communauté comporte également la déclaration des types de donnée définis par l'utilisateur, représentés par un rectangle.

e) Unités

Pour de larges applications, on préférera regrouper les spécifications de classes d'objets en plusieurs *unités*. A chaque unité correspond un diagramme de communauté. Dans l'exemple de l'entreprise CIM, on peut envisager de créer deux unités : une unité `GESTIONNAIRE` contenant une classe d'objets `GESTIONNAIRE` et une unité `ATELIER` correspondant au diagramme de communauté de la figure 8.

4.2.2. STRUCTURE D'UN OBJET

La structure d'un objet est définie dans le *diagramme de déclaration* de la classe d'objets. Les attributs occupent la partie supérieure du diagramme ; les actions se trouvent dans la partie inférieure (voir figure 9 les déclarations associées à la classe d'objets `VEHICULE`).

a) Attributs

Chaque attribut est déclaré par un nom et un codomaine qui définit les valeurs que l'attribut peut prendre. Un codomaine peut être :
- un type de donnée ;
- un type objet ;
- une action (voir le mécanisme de *rappel (call-back)*, plus loin dans ce chapitre).

Les attributs décrivent les propriétés des objets (par exemple le statut du véhicule) aussi bien que des relations entre les objets (l'attribut `lieu` désigne l'instance de `MACHINE` devant laquelle se trouve le véhicule).

b) Actions

Les actions sont utilisées pour décrire le comportement d'un objet. Les actions sont classifiées en *actions de naissance* (qui créent une nouvelle instance dans une classe d'objets), *actions de mort* (qui détruisent l'instance) et *actions de mise à jour* (qui ne font que modifier l'état de l'instance). Les actions de naissance et de mort sont indiquées respectivement par une astérisque ("`*`") et une croix ("`+`") en-dessous de leur nom.

De plus, on distingue les actions *actives* et *passives*. Les actions actives, comme `charge`, ont lieu de la propre initiative de l'objet. Les actions passives, au contraire, ne peuvent avoir lieu qu'à la suite d'un stimulus extérieur. L'action `lanceTransport` est un exemple d'action passive, déclenchée à partir de l'objet `contrôleur`. L'activité d'une action est indiquée par un point d'exclamation ("`!`") en-dessous de son nom. Remarquons que l'action de naissance `construit` est active : le véhicule a l'initiative pour "se créer" lui-même (au commencement de la vie de la communauté des objets).

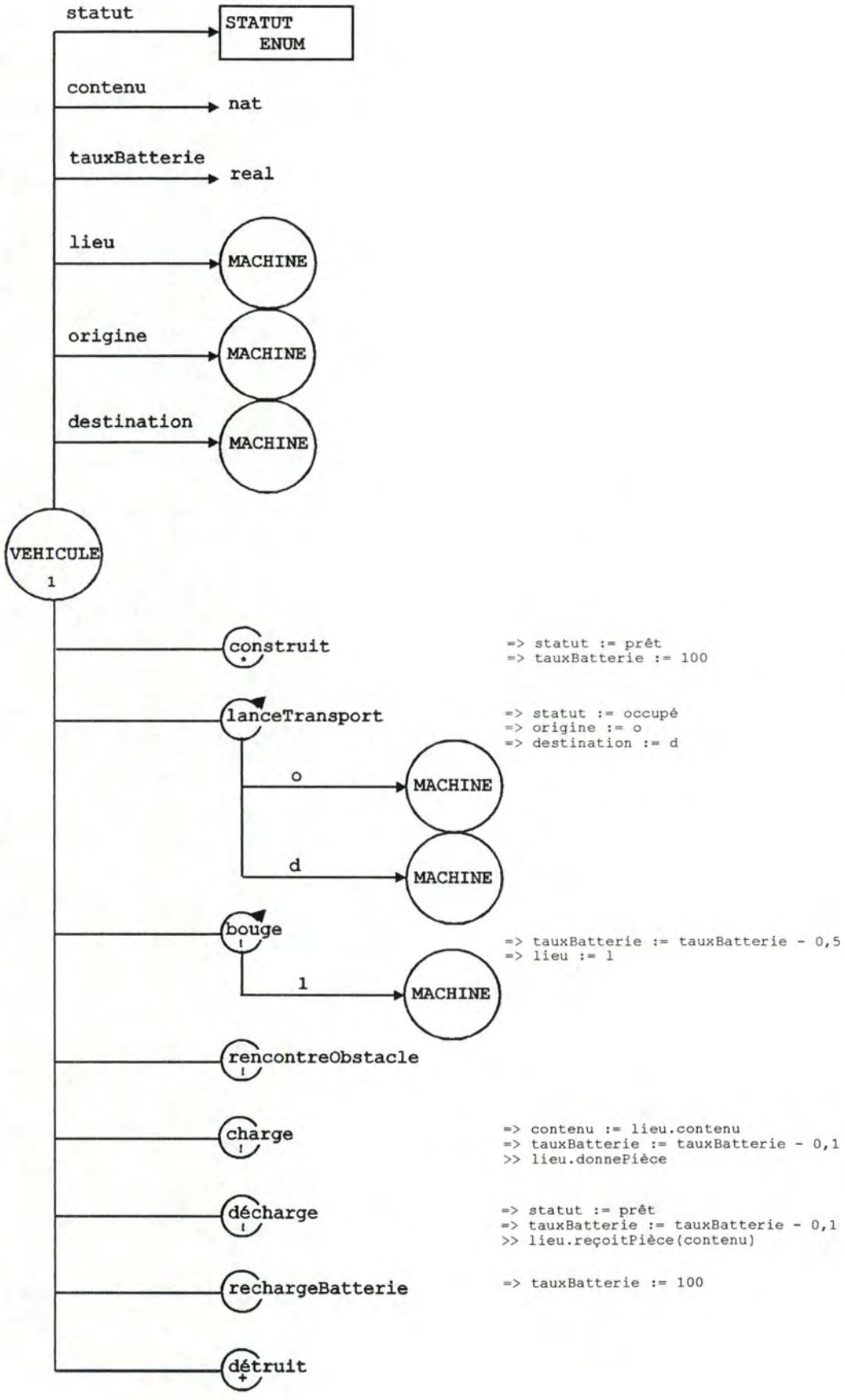


Figure 9 - Diagramme de déclaration de la classe d'objets VEHICULE

Les actions peuvent avoir des *paramètres* (que l'on déclare de la même manière que les attributs). Les paramètres sont utilisés pour définir les effets de l'action sur les attributs ou pour échanger de l'information entre objets. L'action `bouge`, par exemple, a comme paramètre la machine vers laquelle le véhicule doit se déplacer. Dans les expressions, le nom d'un paramètre est préfixé par le nom de l'action (Exemple : `bouge.l`)[3]. Les actions qui ont des paramètres sont surmontées d'un triangle noir ("▲").

Enfin, précisons que les actions n'ont pas de durée ; nous dirons qu'elles sont *atomiques*.

4.2.3. COMPORTEMENT D'UN OBJET

Le comportement d'un objet est formé de trois éléments : les *effets* des actions, le *cycle de vie* et l'*interaction* de l'objet. Effets et interactions sont spécifiés à partir du diagramme de déclaration (cf figure 9) ; le cycle de vie est représenté par le *diagramme de comportement* (voir figure 10).

a) Effets des actions

Les effets d'une action sur l'état (c'est-à-dire sur les attributs) sont décrits dans la clause *mise à jour d'attribut* associée à cette action. Cette clause comprend une liste d'affectations qui définissent les nouvelles valeurs des attributs après l'occurrence de l'action. Ces affectations sont introduites par le symbole "=>". Si la clause est vide, les attributs ne changent pas de valeur. Dans l'action `construit`, les attributs `statut` et `tauxBatterie` sont initialisés. Les actions `bouge`, `charge` et `décharge` décrémentent le taux de chargement de la batterie.

b) Cycle de vie

La *vie* d'un objet est une séquence d'actions commençant par une action de naissance et se terminant éventuellement par une action de mort. Le *diagramme de comportement* décrit les cycles de vie possibles d'un objet sous la forme d'un graphe situation-transition (voir figure 10).

Une *situation* définit un ensemble d'actions possibles. Durant sa vie, l'objet est toujours dans une situation bien déterminée et ne peut accomplir que les actions possibles dans la situation courante. Suite à l'occurrence d'un action possible, l'objet évolue d'une situation à une autre, ce qui correspond à une *transition*.

Une transition peut être inhibée par une *précondition*, en d'autres termes, une action dans une certaine situation ne peut avoir lieu que si la précondition associée est vérifiée. Enfin, le diagramme de comportement comprend également les clauses d'*instanciation* des paramètres d'action.

---

[3] Lorsqu'aucune ambiguïté n'est permise, nous n'appliquerons pas cette règle qui, selon nous, gêne inutilement la lecture.

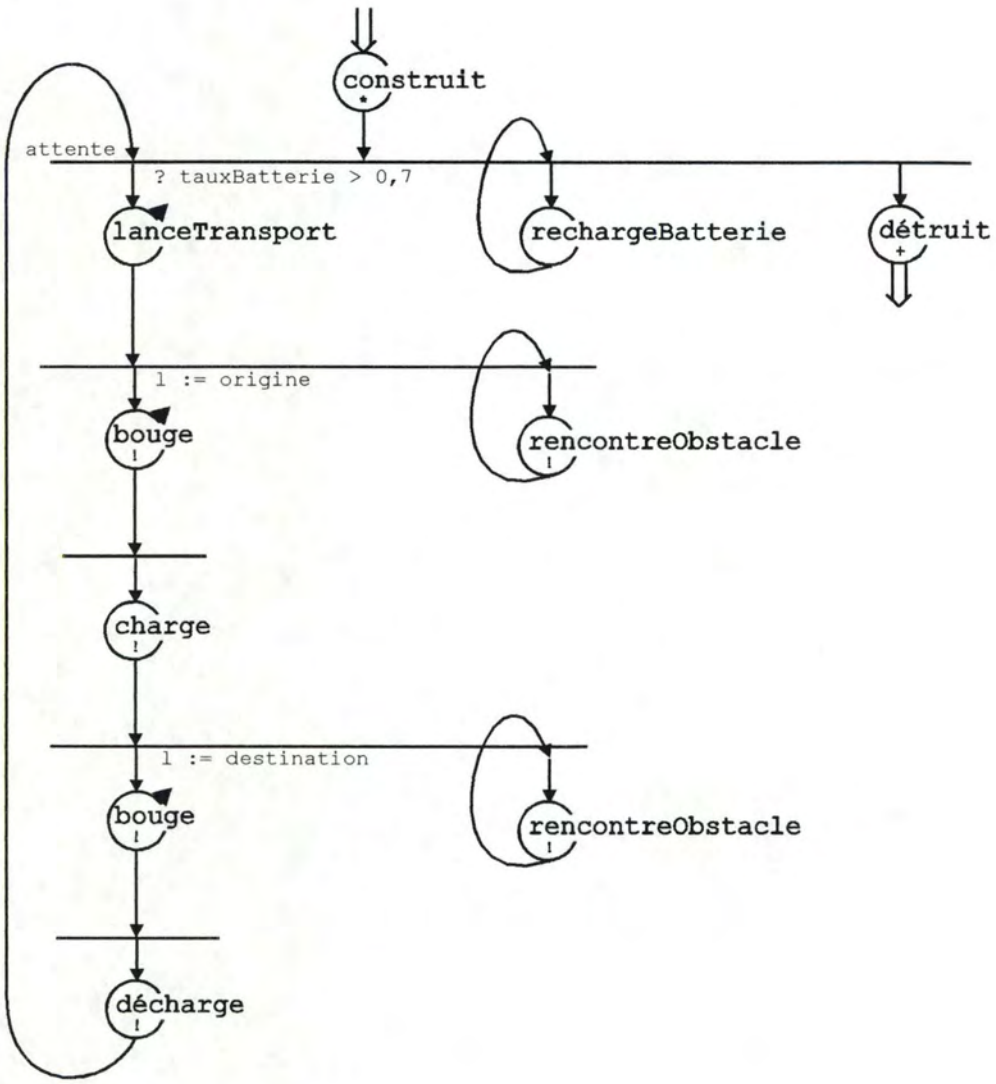


Figure 10 - Diagramme de comportement de la classe d'objets VEHICULE

A la figure 10, on peut lire que, après sa construction, le véhicule se trouve dans une situation d'attente. Les actions passives `lanceTransport`, `rechargeBatterie` et `détruit` sont "accessibles". Cependant, l'action `lanceTransport` n'est possible que si la batterie est suffisamment chargée pour mener le transport à son terme. Un transport fait intervenir deux fois l'action `bouge` avec successivement, comme paramètre, l'origine puis la destination du transport.

Remarque : un diagramme de comportement peut laisser place à de l'indéterminisme. Par exemple, à la figure 10, le véhicule a le choix entre l'action `Bouge` et l'action `rencontreObstacle` dans deux situations. Plus généralement, considérons le diagramme de la figure 11 : les actions peuvent avoir lieu dans un ordre quelconque (si ce n'est qu'aucune action ne peut avoir lieu avant la naissance ni après la mort) et les paramètres des actions prennent des valeurs arbitraires.

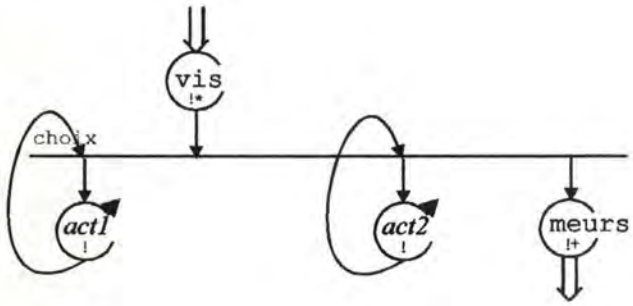


Figure 11 - Cycle de vie non déterministe

c) Interaction

Les objets interagissent entre eux via le mécanisme *d'appel d'action*. La clause *appel*, associée à une action, représente la liste des appels de cette action. Pour chaque appel (introduit par le symbole ">>"), il faut spécifier l'identifiant de l'objet appelé, l'action appelée (obligatoirement une action passive) ainsi que les éventuels paramètres réels de l'action.

L'appel synchronise la vie de l'objet appelant et de l'objet appelé. En effet, lorsqu'une action en appelle une autre, ces deux actions ont lieu simultanément, pour autant qu'elles soient toutes les deux possibles (dans le cas contraire, l'appel ne se produit pas et aucune des deux actions n'a lieu).

La clause *appel* associée à l'action `charge` signifie que cette action appelle l'action `reçoitPièce` de l'instance de `MACHINE` qui est identifiée par l'attribut `lieu`. Le paramètre échangé est la valeur de l'attribut `Contenu`.

Un cas particulier d'appel concerne les actions de naissance. Nous avons vu qu'un objet avait la capacité de se créer tout seul, comme l'objet `véhicule`. Dans d'autres cas, l'action de

naissance est passive. La figure 12, par exemple, représente le fait que le gestionnaire ajoute une nouvelle machine à l'atelier. L'action `ajoute` appelle l'action `construit` d'une nouvelle instance de la classe `MACHINE`. L'identifiant de cette nouvelle instance est fourni au paramètre `nvelleMachine`.

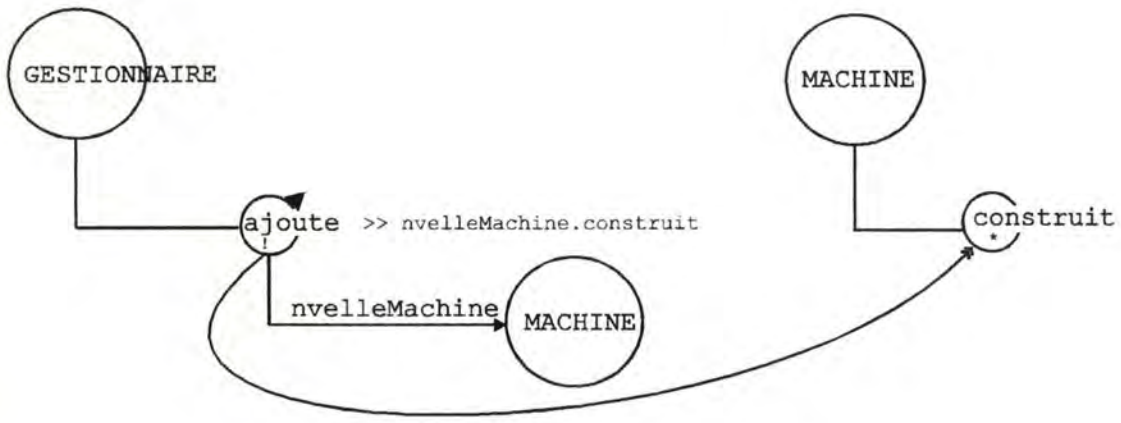


Figure 12 - Création d'une nouvelle instance de MACHINE

Signalons encore brièvement que :
- un appel peut être inhibé par une précondition ;
- si l'identifiant de l'appel est une liste d'instances, chaque instance de la liste est appelée (mécanisme de *multicast*) ;
- une action peut appeler une action d'un objet appartenant à une autre unité ;
- un appel vers ou de l'extérieur est un appel à ou de quelque chose en-dehors du monde OBLOG (par exemple une fonction C). Ce mécanisme est très intéressant puisqu'il permet d'interfacer facilement une application OBLOG avec d'autres applications ;
- une action peut appeler une action dont la référence est stockée dans un attribut *(stored action)*. Ce mécanisme permet d'implémenter une architecture client-serveur, le client passant en paramètre la référence de l'action à rappeler *(to call back)* lorsque le service est terminé.

4.2.4. LE LANGAGE D'EXPRESSION D'OBLOG-KERNEL

Dans l'exemple développé jusqu'ici, nous avons utilisé des expressions en plusieurs endroits : mise à jour d'attribut, précondition de transition, instanciation de paramètres, etc. Brièvement, signalons que les expressions sont formées au moyen :
- d'attributs et de paramètres ;
- des opérateurs classiques sur les types de donnée ;
- de la constante `SELF` qui, dans la spécification d'un objet, désigne l'objet lui-même ;
- de requêtes *(queries)*.

Il existe deux types de requêtes : les requêtes d'instance et les requêtes de classe.

• Une requête d'instance (construite avec l'opérateur ".") renvoie la valeur d'un attribut d'un objet dont on possède la référence. Par exemple, dans l'effet de l'action `charge`, "`lieu.contenu`" désigne la valeur de l'attribut `contenu` de la machine référencée par l'attribut `lieu`. Les requêtes d'instances peuvent être imbriquées (exemple : *réf1 . réf2 . réf3 . attr*).

• Les requêtes de classe sont construites à l'aide des opérateurs `ONE`, `ALL` et `EXISTS`. Exemples :
- `ONE [MACHINE | statut = prêt]` renvoie une machine prête (plus précisément, une instance de `MACHINE` dont l'attribut `statut` vaut `prêt`) ;
- `ALL [MACHINE | statut = prêt]` renvoie toutes les machines prêtes ;
- `EXISTS [MACHINE | statut = prêt]` indique s'il existe au moins une machine prête.

Comme les requêtes d'instance, les requêtes de classe peuvent être imbriquées.

4.2.5. TYPES DE CLASSE D'OBJETS

Le langage OBLOGkernel s'intègre dans un environnement de production de logiciels OBLOG-CASE V1.0 qui a pour objectif final la génération automatique du code. Pour réaliser ces logiciels, les utilisateurs peuvent faire appel à différentes technologies telles que des technologies d'interface homme-machine ou de gestion de base de données.

OBLOGkernel propose une vue unifiée de ces technologies dans le sens où les différents concepts technologiques sont reliés à des concepts OBLOGkernel : on spécifiera les concepts technologiques à l'aide de types particuliers de classe d'objets.

Les caractéristiques et les limitations d'une technologie définissent le sous-ensemble des concepts d'OBLOGkernel et la sémantique opérationnelle qui sont associés à un type d'objet. Les détails d'implémentation, tel que le *layout* d'une boîte de dialogue, ne font pas partie de la spécification du type d'objet mais sont spécifiés à l'aide d'un éditeur spécifique d'OBLOG-CASE. (On le voit, les types d'objet sont à la frontière entre le "monde" OBLOG et la technologie ; ils constituent une interface entre une spécification OBLOG et certaines technologies particulières.)

OBLOGkernel fournit les types d'objet *boîte de dialogue (kind-of DBX)* et *table (kind-of TBL)*.

a) Type boîte de dialogue

Ce type d'objet est destiné à supporter l'interface utilisateur de l'application. A un objet est associée une boîte de dialogue que l'on peut construire dans l'*éditeur de dialogues* (voir section 4.4). La relation entre la boîte de dialogue et la spécification OBLOGkernel est la suivante :

Boîte de dialogue	Spécification OBLOGkernel
Boîte de dialogue *(dialog box)*	Objet de type DBX
Etiquette	Attribut constant
Champ de saisie	Attribut variable
Champ d'affichage uniquement	Attribut variable
Bouton	Action passive, appelée de l'extérieur

b) Type table

Par défaut, les objets sont volatiles : une fois l'exécution de l'application terminée, ils disparaissent. Les objets de type table sont, eux, persistants : la population des objets de ce type est conservée dans une base de données que l'on peut créer dans l'*éditeur de schémas de base de données* (voir section 4.4). La relation entre la base de données (de type relationnel) et la spécification OBLOGkernel est la suivante :

Bases de donnée	Spécification OBLOGkernel
Table	Classe d'objets de type TBL
Enregistrement	Objet de type TBL
Champ	Attribut
Opération d'insertion	Action de naissance
Opération de mise à jour	Action de mise à jour
Opération de suppression	Action de mort

4.3. LE LANGAGE OBLOG-LIGHT

Comme son nom l'indique, OBLOGkernel constitue le noyau du langage OBLOG. La prochaine version d'OBLOG-CASE devrait fournir un langage beaucoup plus étendu, appelé OBLOGlight. Ce chapitre présente brièvement les principales extensions d'OBLOGlight que nous utiliserons au cours de la troisième partie. La définition du langage proposée ici est inspirée des documents préparatoires ([ESDI92e, ESDI92f]) disponibles au moment de la rédaction de ce mémoire ainsi que de divers exposés sur le sujet.

Les exemples utilisés pour illustrer les nouveaux concepts sont des exemples intuitifs portant sur une classe d'objets PERSONNE. Ces exemples sont présentés sous forme textuelle. Malgré les nombreux avantages de la représentation graphique, il est en effet apparu indispensable d'introduire, à l'occasion d'OBLOGlight, une représentation textuelle du langage (les deux représentations ne doivent pas nécessairement être exclusives mais peuvent coexister)[4].

[4] Comme aucune représentation d'OBLOGlight n'est encore rigoureusement définie, il nous a fallu improviser certaines notations. Selon les cas, nous aménagerons les diagrammes d'OBLOGkernel pour les adapter aux nouveaux concepts ou nous utiliserons les notations textuelles introduites dans les sections suivantes.

4.3.1. CLASSES D'OBJETS, MÉTACLASSES ET TYPES DE DONNÉE

a) Classes d'objets

• Lorsqu'une contrainte de cardinalité impose qu'une classe d'objets contienne au plus une instance *(single constraint)*, il n'est pas nécessaire d'identifier explicitement la seule instance de la classe. Pour référencer cette instance, on utilise directement le nom de la classe (en minuscules).

• Une opération `CLASSOF` est définie sur toutes les classes d'objets. Cette opération accepte en argument une instance et renvoie comme résultat le nom de la classe à laquelle cette instance appartient.

b) Métaclasses

A chaque classe d'objets est associée un *objet de classe (class object)* ou *métaclasse*. Cet objet particulier (de catégorie *single*) n'a pas de représentation propre et réside "à l'intérieur" de la spécification d'une classe d'objets. Il permet de spécifier des propriétés, non pas d'une instance particulière, mais de la classe d'objets tout entière (voir les attributs de la métaclasse, plus loin dans ce chapitre).

c) Types de donnée

OBLOGlight élargit sensiblement l'ensemble des types de donnée disponibles. Notons :
- parmi les types de donnée simples : `date`, `time`, `money` ;
- parmi les constructeurs de types : `RANGE`, `STRUCT` (produit cartésien), `UNION`, `ARRAY`.

Tous les types de donnée (excepté `ARRAY`) contiennent implicitement une donnée indéfinie, appelée `NULL`. Toute opération qui devrait fournir une valeur incompatible avec le type de donnée spécifié renvoie la valeur `NULL`.

4.3.2. STRUCTURE D'UN OBJET

a) Attributs

• On distingue les attributs *variables*, que l'on peut modifier librement, des attributs *constants*, pour lesquels une valeur est donnée à la création de l'objet et ne peut jamais être modifiée.

• Un attribut peut être *dérivé*, ce qui signifie que sa valeur est calculée, selon une *règle de dérivation*, en fonction de la valeur d'autres attributs (et d'éventuelles préconditions). Exemple :

```
âge : nat
DER majeur : bool = (âge >= 18).
```

• Une *contrainte* peut être associée à un attribut afin de restreindre l'ensemble des valeurs que l'attribut peut prendre. Exemple :

```
solde : int
    {NOT majeur} ! >= 0
```

exprime que le solde d'une personne mineure ne peut être négatif (l'expression entre accolades est une précondition sur la contrainte).

• Les attributs de la métaclasse représentent des propriétés de la classe d'objets tout entière. Ces attributs sont préfixés du sigle "$". Exemple :

```
DER $nbPersonnes : nat = #(ALL[p:PERSONNE]).
```

• Les *clés (keys)* sont des attributs particuliers de la métaclasse. Une clé est une fonction qui, à partir d'un tuple de valeurs d'attributs identifiants, renvoie l'unique instance qui possède ces valeurs d'attributs. Exemple : la clé

```
$laPersonne(nom, prénom) : PERSONNE
```

renvoie une personne sur base des valeurs des attributs nom et prénom.

b) Actions

• En plus des préconditions associées à des situations particulières, on peut associer à une action une *précondition globale*.

• De même qu'il existe des attributs dérivés, une action peut être *dérivée* d'une autre action. L'action dérivée se distingue par le fait que les valeurs de ses paramètres sont données par des règles de dérivation. Exemple : étant donné une action

```
!dépense(m : nat),
```

on peut définir l'action dérivée

```
DER !dépenseTout = dépense(solde).
```

c) Interface

Le langage OBLOGkernel ne prévoit aucune restriction quant à l'utilisation des requêtes ; l'état d'un objet peut être observé librement par la communauté. D'un point de vue méthodologique, il est évidemment souhaitable de construire des interfaces "propres" entre les objets en distinguant clairement la partie *privée* et la partie *publique* d'un objet.

OBLOGlight supporte explicitement ce principe d'*interface*. La déclaration d'un objet comprend deux volets : une *interface* et un *corps*. Le corps de la déclaration comprend tous les attributs et toutes les actions de l'objet, comme dans OBLOGkernel. L'interface de la déclaration consiste en un sous-ensemble du corps rendu public, c'est-à-dire :
- une liste d'attributs observables par la communauté ;

- une liste des actions pouvant être déclenchées par la communauté (il s'agit en réalité de toutes les actions passives)[5].

Cependant, il est souvent intéressant d'offrir, non pas un sous-ensemble, mais une *vue* des attributs et des actions. C'est pourquoi l'interface d'un objet peut contenir des attributs et des actions dérivés, qui n'existent pas dans le corps mais qui sont dérivés à partir des attributs et des actions du corps.

Enfin, à chaque attribut ou action public peut être associée une précondition. L'attribut ne peut être observé, l'action ne peut être déclenchée, que si la précondition est vérifiée.

[5] Les attributs et les actions rendus publics pourraient être renommés respectivement *observations* et *événements*. Nous préférerons continuer à parler d'attributs publics et d'actions publiques.

```
PERSONNE

INTERFACE

ATTRIBUTES

nom : string
prénom : string
$nbPersonnes : nat
KEY $laPersonne(nom, prénom) : PERSONNE

ACTIONS

* nais
reçoit(m : nat)

BODY

ATTRIBUTES

CONST nom : string
CONST prénom : string
CONST dateNaiss = date
DER âge : nat
     = calendrier.dateJour - dateNaiss
DER majeur : bool
     = âge >= 18
solde : int
     {NOT majeur} ! >= 0
DER $nbPersonnes : nat
     = #(ALL[p:PERSONNE])
KEY $laPersonne(nom, prénom) : PERSONNE

ACTIONS

* nais(n, p : string)
     => nom := n
     => prénom := n
     => dateNaiss := calendrier.dateJour
reçoit(m : nat)
     => solde := solde + m
!emprunte
     >> banque.demandeCrédit(SELF)
!dépense(m : nat)
     {m <= solde} => solde := solde - m
     {m > solde} => solde := solde - (m * 1,08).
DER !dépenseTout
     = dépense(solde)
     ? solde > 0
+ !meurs
```

Figure 13 - Déclarations de la classe d'objets PERSONNE (représentation textuelle)

4.3.3. COMPORTEMENT D'UN OBJET

a) Effets des actions

La mise à jour d'un attribut par une action peut être soumise à une précondition (entre accolades). Exemple :

```
!dépense(m : nat)
     {m <= solde} => solde := solde - m
     {m > solde} => solde := solde - (m * 1,08).
```

b) Cycle de vie

• Comme la mise à jour d'un attribut, l'instanciation d'un paramètre peut être soumise à une précondition.

• Le diagramme de comportement d'un objet peut s'avérer complexe. Dans certains cas, il est préférable de définir des *patrons (patterns)*, c'est-à-dire des morceaux du diagramme de comportement qui ont une signification par eux-mêmes ou qui se répètent en plusieurs endroits. (La notion de patron correspond, dans un langage de troisième génération, à la notion de procédure.)

c) Interaction

Dans OBLOGkernel, un appel est toujours synonyme de synchronisation : l'appel ne se produit que lorsque action appelante et action appelée ont lieu simultanément. OBLOGlight fournit également un mécanisme d'*appel asynchrone* : l'action appelante peut avoir lieu indépendamment de l'action appelée.

4.3.4. LE LANGAGE D'EXPRESSION D'OBLOG-LIGHT

• Dans une expression, le nom d'une classe d'objets (en minuscules), désigne :
- soit l'unique instance de la classe s'il s'agit d'une classe de catégorie *single* ;
- soit la métaclasse, dans le cas contraire.

• Le langage d'expression des requêtes est amélioré, notamment pour pouvoir manipuler, non seulement des classes d'objets, mais également des types de donnée. L'introduction de variables facilite l'expression de requêtes imbriquées.
Exemple :
- `ALL[(p.nom, p.prénom) OF p:PERSONNE | p.âge > 60]` renvoie la liste des noms et prénoms des personnes qui ont plus de 60 ans ;
- `ALL[p:PERSONNE | EXISTS[v:VOITURE | v.propriétaire = p]]` renvoie la liste des personnes qui possèdent une voiture.

4.3.5. TYPES DE CLASSE D'OBJETS

Le langage OBLOGlight offre, en plus des types boîte de dialogue et table, un type de classe d'objets *rapport*. Les objets de types rapport *(kind-of RPT)* représentent des rapports imprimés sur tout ou partie des objets présents dans l'application au moment de l'exécution. L'éditeur associé est l'*éditeur de rapports*, qui permet de créer le format du rapport. Le langage de requête est utilisé pour définir le contenu des *champs* du rapport.

4.3.6. RELATIONS ENTRE OBJETS OU CLASSES D'OBJETS

Le langage OBLOGlight introduit, en plus de la relation d'interaction, quatre catégories de relations entre objets ou classes d'objets. Ces relations sont déclarées dans le diagramme de communauté.

a) Spécialisation / généralisation

Une classe d'objets peut être déclarée comme étant la spécialisation d'une autre classe d'objets. Par exemple, les classes d'objets `HOMME` et `FEMME` sont des spécialisations de la classe d'objets `PERSONNE`. L'objet spécialisé (aussi appelé *aspect*) hérite de la spécification de son *ancêtre* et peut la modifier plus ou moins librement.

La généralisation est la relation inverse de la spécialisation et consiste à abstraire une spécification commune à plusieurs classes d'objets.

b) Agrégation

Un objet peut être déclaré comme étant l'agrégation d'autres objets appelés *composants*. Par exemple, un objet `VEHICULE` agrège un objet `MOTEUR` et quatre objets `ROUE`. L'*agrégat* peut changer dynamiquement ses composants, ce qui permet de décrire le fait qu'un véhicule peut changer de roue pour cause de crevaison.

c) Contraintes d'intégrité

Une contrainte d'intégrité peut être définie entre différentes classes d'objets à l'aide d'une proposition booléenne portant sur les attributs publics de ces classes d'objets.

d) Association

Il s'agit d'un concept similaire à l'association du modèle Entité-Association. L'association sera toujours binaire et sans attributs ; un nom de rôle et une contrainte de cardinalité peuvent être déclarés pour chacune des deux classe d'objets membres de l'association.

4.4. L'OUTIL

L'approche OBLOG s'inscrit dans un projet à long terme destiné à offrir un atelier de conception de logiciels assistée par ordinateur, centré autour du langage OBLOG. La figure 14 présente les différents composants de ce futur atelier.

La version actuelle de l'atelier (OBLOG-CASE V1.0) supporte un sous-ensemble du langage OBLOG (OBLOGkernel). Il comprend déjà les composants suivants :
- le gestionnaire de sessions ;
- l'éditeur OBLOG ;
- l'éditeur de dialogues ;
- un éditeur de schémas de base de données (limité) ;
- un gestionnaire de *repositories* (limité) ;
- un vérificateur (limité) ;
- le générateur de code.

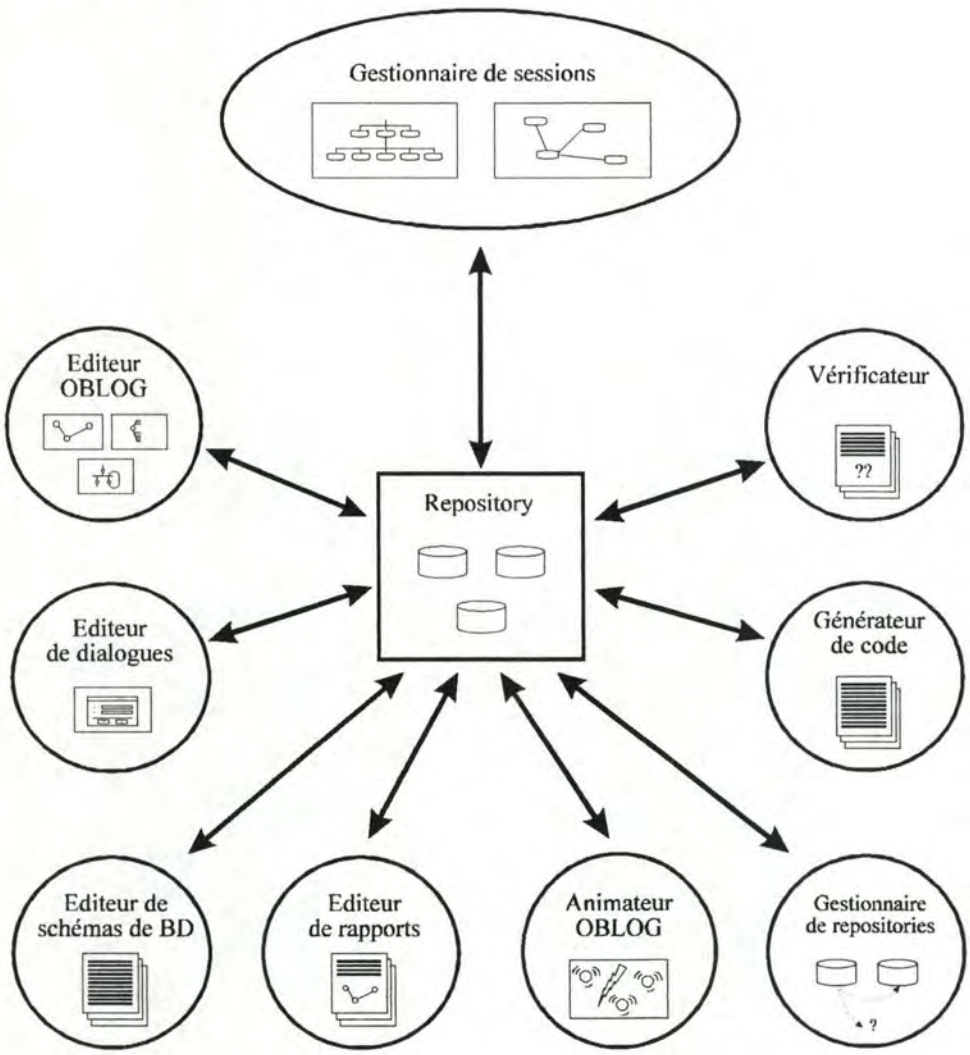


Figure 14 - Les composants du futur atelier OBLOG-CASE

a) Gestionnaire de sessions (Session Manager)

L'utilisateur, au plus haut niveau d'abstraction du projet, manipule les concepts d'*unités* ("containers" à objets), de *liens* entre unités (du type "utilise") et de *groupes* ("containers" à unités/groupes). Le gestionnaire de sessions offre deux diagrammes :
- le *diagramme de système* représente les groupes ;
- le *diagramme de projet* représente les unités et les liens entre les unités.

b) Editeur OBLOG (OBLOG Editor)

Activé dans le contexte d'une unité, l'éditeur offre les diagrammes suivants :
- le *diagramme de communauté* présente les classes d'objets et les types de donnée ainsi que les relations entre classes d'objets ;
- le *diagramme de déclaration* présente les déclarations associées à une classe d'objets ainsi que les effets des actions et les interactions de la classe d'objets avec le reste de la communauté ;
- le *diagramme de comportement* présente le cycle de vie d'un objet.

c) Editeur de dialogues (Dialog Editor)

Cet éditeur permet de créer l'interface utilisateur de l'application (menu, boîtes de dialogues, etc...). Le code MOTIF correspondant est produit par le générateur de code.

d) Editeur de schémas de base de données (Database Schema Editor)

Cet éditeur permet de gérer plus finement la partie base de données de l'application. Le générateur de code produit automatiquement le code SQL DDL (définition statique) et SQL DML (manipulation dynamique) relatif au schéma logique de la base de données. Par contre, afin d'optimiser son application ou de tirer parti de primitives puissantes offertes par le système de gestion de bases de données, l'utilisateur peut définir lui même le DDL/DML relatif au schéma physique de la base de données.

e) Editeur de rapports (Report Editor)

Cet éditeur permet de concevoir des rapports imprimés sur la population des objets de l'application. L'utilisateur "dessine" son rapport dans l'éditeur de rapport et le code PostScript correspondant est produit par le générateur de code.

f) Animateur OBLOG (OBLOG Animator)

Il s'agit d'un outil de débogage interactif semblable à ceux que proposent beaucoup d'environnements de programmation (Turbo Pascal et Turbo C++ par exemple). L'idée est de pouvoir suivre à la trace l'exécution d'une spécification OBLOG, en ayant la possibilité de consulter et de modifier la valeur des attributs ou des paramètres.

g) Gestionnaire de repositories (Repository Manager)

L'ensemble des spécifications d'un projet est stocké dans une base de données appelée *repository*. Grâce au gestionnaire de *repositories*, l'utilisateur a la possibilité de créer et gérer ses propres *repositories*. Il peut également importer et exporter des spécifications via un *flat file* sous le format ASCII.

h) Vérificateur (Checker)

Cet outil permet de vérifier la cohérence et la complétude des spécifications, tant au niveau syntaxique qu'au niveau sémantique.

i) Générateur de code (Code Generator)

Cet outil produit, sur base des différentes spécifications, du code ANSI C, MOTIF, DDL/DML SQL et PostScript, ainsi que différentes procédures de compilation et de *linkage* pour construire le programme exécutable de l'application.

Remarque : l'approche OBLOG est évidemment indépendante de tout langage cible particulier. C, SQL, MOTIF et PostScript sont choisis aujourd'hui parce qu'ils constituent des standards mais d'autres langages peuvent être envisagés.

5. ETUDE COMPARÉE D'ALBERT ET D'OBLOG

Pour conclure les deux premières parties et avant d'aborder le prototypage, il est intéressant de comparer les langages ALBERT et OBLOG dans sa version "*light*" (dans le reste de ce mémoire, le terme "OBLOG" désigne implicitement "OBLOGlight"). La section 5.1 compare la sémantique des deux langages, d'un point de vue général. La section 5.2 aborde un point sémantique particulier, à savoir la logique utilisée dans les deux langages. Les différences syntaxiques sont brièvement signalées à la section 5.3.

5.1. COMPARAISON DE LA SÉMANTIQUE DES DEUX LANGAGES

a) Ressemblances (vue superficielle)

Si nous comparions ALBERT avec un langage de troisième génération (Pascal, C,...), nous dirions certainement qu'ils n'ont rien en commun. Ce n'est certainement pas le cas entre ALBERT et OBLOG.

En effet, on retrouve de part et d'autre l'idée d'organiser un système comme une collection d'"éléments", appelée société d'agents en ALBERT et communauté d'objets en OBLOG. Chaque élément possède un état composé d'attributs. Par défaut, les attributs sont privés (l'état est interne à l'objet) mais ils peuvent être rendus publics (observables par les autres éléments). Durant la vie de l'élément, il se passe des actions qui modifient, qui ont un effet sur l'état. Les actions sont également un moyen de communication entre les différents éléments (visibilité des actions en ALBERT, appel en OBLOG), pour s'échanger de l'information (valeurs des arguments en ALBERT et des paramètres en OBLOG) ou simplement pour synchroniser les vies de ces différents éléments. Le tableau suivant synthétise les correspondances qui existent entre les concepts des deux langages lorsqu'on prend un point de vue superficiel, intuitif.

ALBERT	OBLOG
Au niveau agent	
Agent	Objet
Classe d'agents	Classe d'objets
Attribut	Attribut
Attribut dérivé	Attribut dérivé
Action	Action
Argument	Paramètre
Vie	Vie
Etat	Etat
Changement	Transition
Effet	Mise à jour
Au niveau société	
Société d'agents	Communauté d'objets
Attribut interne/exporté	Attribut privé/public
Action interne/externe	Action privée/publique
Perception des attributs	Observation des attributs
Perception des actions	Appel d'action

b) Différences (vue approfondie)

Si on poursuit la comparaison de manière plus approfondie, on constate deux différences fondamentales quant à la sémantique des deux langages.

• La première différence concerne ce qu'on pourrait appeler la "granularité" de la spécification. Une spécification ALBERT est *orientée-vie*, en ce sens qu'elle fait référence à la vie entière d'un agent. La vie d'un agent est dirigée selon les engagements ou les contraintes d'évolution de l'état. Par exemple, les actions produites au temps *t8* peuvent dépendre de celles qui ont eu lieu au temps *t5* ou même de celles qui auront lieu au temps *t12*.

Une spécification OBLOG, par contre, est *orientée-transition* : étant donné une situation quelconque, on en déduit l'ensemble des transitions possibles sans tenir compte des transitions antérieures et sans prendre conscience du but poursuivi.

Cette différence a une conséquence fondamentale. Une spécification ALBERT identifie l'ensemble des vies possibles d'un agent. Etant donné une vie quelconque, il est toujours possible de dire si cette vie est une vie valide de l'agent ou non. Mais une spécification ALBERT ne permet pas, du moins en toute généralité, de construire une vie particulière qui appartienne à l'ensemble des vies valides. En d'autres termes, elle ne donne pas les moyens d'écrire un programme qui sache, à tout moment, comment conduire la vie d'un agent de telle sorte qu'elle soit valide, puisque cette validité ne peut être évaluée qu'en prenant en compte la vie entière.

Par contre, une spécification OBLOG donne, à tout moment de la vie d'un objet, les règles de transition. Etant donné l'état initial, il suffit d'appliquer précisément ces règles pour "dérouler" la vie de l'objet, en ayant toujours la garantie que cette vie est valide.

En fait, cette différence renvoie à la distinction entre sémantique déclarative et opérationnelle. Nous avons déjà insisté sur le caractère déclaratif des langages d'I.B. (cf point 2.4.2). Une spécification ALBERT constitue une description du problème qui n'est pas basée sur un modèle informatique, c'est-à-dire qu'elle n'est pas exprimée en termes de transformations à appliquer sur des arguments pour produire des résultats.

Le langage OBLOG possède avant tout une sémantique opérationnelle et permet de décrire la solution à un problème posé. Cette description est basée sur un modèle informatique.

• Une autre différence importante dont il faudra tenir compte est facile à comprendre : deux occurrences d'action peuvent avoir lieu simultanément en ALBERT mais pas en OBLOG. En d'autres termes, un changement dans la vie d'un agent peut comprendre plusieurs actions tandis qu'une transition dans la vie d'un objet ne peut jamais comporter qu'une seule action.

5.2. COMPARAISON DE LA LOGIQUE UTILISÉE DANS LES DEUX LANGAGES

Outre les différences générales de sémantique des deux langages, nous voudrions aborder, toujours au niveau sémantique, la question particulière du langage logique utilisé, afin d'introduire certaines notions utilisées dans la suite.

• Rappelons que le langage ALBERT est basé sur la logique du calcul des prédicats du premier ordre typée (plus simplement : "logique du premier ordre" ou LPO). Cette logique est basée sur les notions de *terme*, *formule atomique* (ou *prédicat*) et *formule bien formée* (voir notamment [Pet92]). La logique ALBERT comporte en plus certaines extensions temporelles.

Remarque : on fera la supposition (réaliste) que, dans un texte ALBERT, toutes les formules sont des formules bien formées *fermées*, ce qui signifie que toute variable y apparaissant est quantifiée. (Rappelons qu'en ALBERT toute variable qui n'est pas sous la portée d'un quantificateur est implicitement quantifiée universellement en-dehors de la formule.)

Le langage OBLOG ne supporte qu'un sous-ensemble de la logique du premier ordre pour lequel nous proposons les définitions suivantes :
- *expression :* une constante est une expression ; une variable quantifiée existentiellement est une expression ; l'application d'une fonction n-aire à n expressions est une expression (un attribut ou un paramètre est considéré comme une fonction) ;
- *proposition simple :* l'application d'un prédicat n-aire à n expressions est une proposition simple ;
- *proposition composée :* composition de propositions au moyen des connecteurs `NOT`, `AND`, `OR`, `(`, `)` et du quantificateur existentiel `EXISTS[`x`]` où x représente une variable appartenant à un ensemble de données "tangible", c'est-à-dire un ensemble de données fini et présent dans la mémoire du système (par exemple la communauté d'objets ou un attribut de type `LIST`).

En synthèse, la logique OBLOG comporte deux restrictions importantes :
- d'une part, elle n'offre pas d'extensions temporelles (cette restriction est liée au caractère orienté-transition d'OBLOG) ;
- d'autre part, elle permet l'utilisation d'un quantificateur uniquement sur certains ensembles finis de données, que nous appelons ensembles de données tangibles.

(Les autres types de restriction peuvent aisément être levés à l'aide des règles de déduction logique.)

5.3. COMPARAISON AU NIVEAU SYNTAXIQUE

a) Au niveau lexical

Les différences lexicales entre les deux langages sont relativement restreintes et, de toute façon, il est impossible de les répertorier rigoureusement étant donné que la syntaxe des deux langages n'est pas clairement fixée.

b) Au niveau typographique

Les conventions typographiques ne doivent pas être négligées, si on veut comprendre plus facilement un texte ALBERT ou OBLOG.

- Exemple en ALBERT :

`Véhicule`	nom de l'agent qui représente la notion de véhicule
`VEHICULE`	type de l'identifiant de cet agent
`véhicule`	identifiant de l'agent Véhicule
`Statut`	un attribut de l'agent
`Finis_chargement`	une action de l'agent.

- Exemple en OBLOG :

`VEHICULE`	classe d'objets ou type objet
`véhicule`	une instance de la classe
`statut`	un attribut de la classe d'objets
`finisChargement`	une action de la classe d'objets
`NOT`	mot réservé du langage.

PARTIE III

D'ALBERT VERS OBLOG

6. LE PROTOTYPAGE

De plus en plus, le prototypage s'impose dans le cycle de vie d'un logiciel. On ne compte plus en effet les projets informatiques qui ont échoué parce que le produit final s'avérait inadéquat par rapport aux besoins de l'organisation. Obtenir un rapide feed-back de la part du client au cours du développement du projet apparaît aujourd'hui comme une nécessité.

Il est reconnu que la phase critique d'un projet est celle de l'élaboration du cahier des charges. Il n'est pas étonnant dès lors que beaucoup de recherches actuelles concernent le prototypage des spécifications des besoins, rendu possible par l'apparition des langages formels d'I.B.

Les deux sections suivantes aborderont successivement le prototypage dans le cycle de vie du logiciel en général (section 6.1) et, plus spécifiquement, dans le cadre des spécifications des besoins (section 6.2). La section 6.3 présente la démarche concrète que nous avons adoptée pour prototyper une spécification ALBERT en OBLOG.

6.1. LE PROTOTYPAGE DANS LE CYCLE DE VIE DU LOGICIEL

6.1.1. CLASSIFICATION DES SYSTÈMES DE PROTOTYPAGE

En informatique, la notion de prototypage recouvre une multitude de réalités, tant du point de vue des objectifs poursuivis que des approches adoptées. L'analyse de la littérature sur ce sujet montre que les termes de "prototype" et de "prototypage" concernent un large spectre de produits et d'activités. Une telle diversité rend difficile l'élaboration d'une taxonomie des systèmes de prototypage.

C'est pourtant ce que Habra a tenté de dégager en proposant une grille multidimensionnelle tenant compte de plusieurs critères de classification. Une approche de prototypage sera caractérisée par sa position relative à chacun des critères suivants [Hab90] :

- l'objectif poursuivi, par exemple tester la conformité aux besoins, montrer la faisabilité, convaincre des utilisateurs réticents,...
- le champ d'application, c'est-à-dire la partie du système concernée par le prototypage, par exemple l'interface utilisateur, les fonctionnalités, la base de données,...
- les caractéristiques du système que le prototype doit refléter, par exemple le temps de réponse, la convivialité, la robustesse,...

- la relation entre le prototype et la spécification, par exemple "correcte implémentation de", "partie de",...
- la relation entre le prototype et le produit final, par exemple "première version de", "partie de",...

Ce cadre d'analyse n'a pas qu'un intérêt théorique. Il offre en même temps des repères pour savoir ce que l'on fait. Trop souvent en effet, « le mot "prototype" désigne un produit n'ayant pas de statut clairement défini dans le cycle de vie du logiciel » [Hab90]. D'après Habra, une activité de prototypage est dite *saine* lorsqu'elle est bien définie par rapport aux critères de la grille. Nous montrerons au point 6.2.2 en quoi notre approche de prototypage peut être qualifiée de saine.

6.1.2. QUALITÉS ATTENDUES D'UN PROTOTYPE

En plus d'être saine, une approche de prototypage doit déboucher sur un prototype qui réunit les qualités suivantes [Hab90] :
- être rentable, c'est-à-dire que son coût de production doit se justifier par une réduction au moins équivalente du coût des phases suivantes ;
- être produit rapidement ;
- refléter fidèlement les caractéristiques du système pour lesquelles il a été produit ;
- être montrable, ce qui nécessite une certaine intégration du prototype dans l'environnement et une interface utilisateur ;
- être aisé à modifier suivant les observations du client, ce qui nécessite des qualités de structuration comme la modularité, afin de pouvoir localiser les changements.

6.1.3. STRATÉGIES DE PROTOTYPAGE DE SPÉCIFICATIONS FORMELLES

Un des intérêts d'utiliser un langage formel de spécification est de pouvoir dériver un prototype de manière automatique ou semi-automatique. En effet, « les langages formels permettent aussi de définir formellement ce que signifie "une implémentation correcte" et de vérifier cette correction » [Hab90]. Il ne s'agit pas d'une activité triviale pour autant puisqu'il s'agit de passer d'un langage de spécification à un langage de programmation. Pour y arriver, deux stratégies peuvent être envisagées : une *stratégie transformationnelle* ou une *stratégie d'interprétation directe* [Hab90].

a) Stratégie transformationnelle

Suivant cette stratégie, le passage d'un langage à l'autre a lieu progressivement, par applications successives de règles de transformation.

Cette production peut être en partie automatisée. En effet, on constate que « les spécifications sont souvent écrites dans un *style constructif*, c'est-à-dire [à l'aide de] propriétés qui définissent explicitement comment le résultat d'une opération est construit à partir de ses arguments ». Produire du code exécutable à partir de telles propriétés est une activité systématique qui peut être programmée. Cependant, il subsiste généralement « des propriétés écrites dans un pur

style observationnel, c'est-à-dire des propriétés ne définissant aucune relation précise entre le résultat et les arguments de l'opération ». Dans ce cas, le processus de transformation sera assisté par le spécifieur.

Comme les spécifications seront ajustées selon les commentaires exprimés par le client lors de l'exécution du prototype, il est essentiel de pouvoir établir la correspondance entre les composants de la spécification et ceux du prototype. On dira que le processus de prototypage est *traçable* lorsque « le prototype généré préserve non seulement les propriétés sémantiques de la spécification originale mais aussi la structure de la spécification et ses noms d'objet et d'opération ». Comme corollaire, le système de transformation doit « éviter, autant que possible, l'introduction d'opérations ou d'objets intermédiaires nouveaux qui n'ont pas été mentionnés explicitement par le spécifieur ».

Remarquons qu'une des applications intéressantes du l'atelier OBLOG-CASE consiste à rédiger les spécifications de conception d'un logiciel en OBLOG et d'utiliser le générateur de code pour dériver automatiquement un prototype. Ce cas illustre une stratégie transformationnelle du langage OBLOG vers le langage C.

b) Stratégie d'interprétation directe

« L'idée est de construire une machine abstraite qui admette le langage de spécification comme langage source, au lieu de traduire le langage de spécification dans un langage de programmation pour lequel une machine abstraite existe déjà. »

Le problème revient à associer au langage de spécification une *sémantique opérationnelle*, dont on aura prouvé l'équivalence avec la sémantique déclarative. N'importe quel langage approprié (tel le C) peut être utilisé pour implémenter l'interpréteur. Si c'est un langage de bas niveau, manipulant des concepts comme les registres, on peut même obtenir une interprétation efficace.

Cette stratégie apparaît beaucoup plus simple que la première puisqu'elle demande la maintenance d'un seul produit, qui joue à la fois le rôle de spécification et de prototype, au lieu de deux produits différents comme dans le cas de la stratégie transformationnelle. On évite notamment les problèmes de traçabilité.

Cependant, certains auteurs restent d'avis de conserver une nette séparation entre la spécification d'une part et les différents produits exécutables du cycle de développement d'autre part. L'utilisation de deux langages distincts contribue
- à empêcher la confusion des tâches de spécification et de programmation ;
- surtout à prévenir la tentation, pour son concepteur, d'adapter le langage de spécification pour le rendre directement opérationnel, en n'autorisant par exemple que le style constructif.

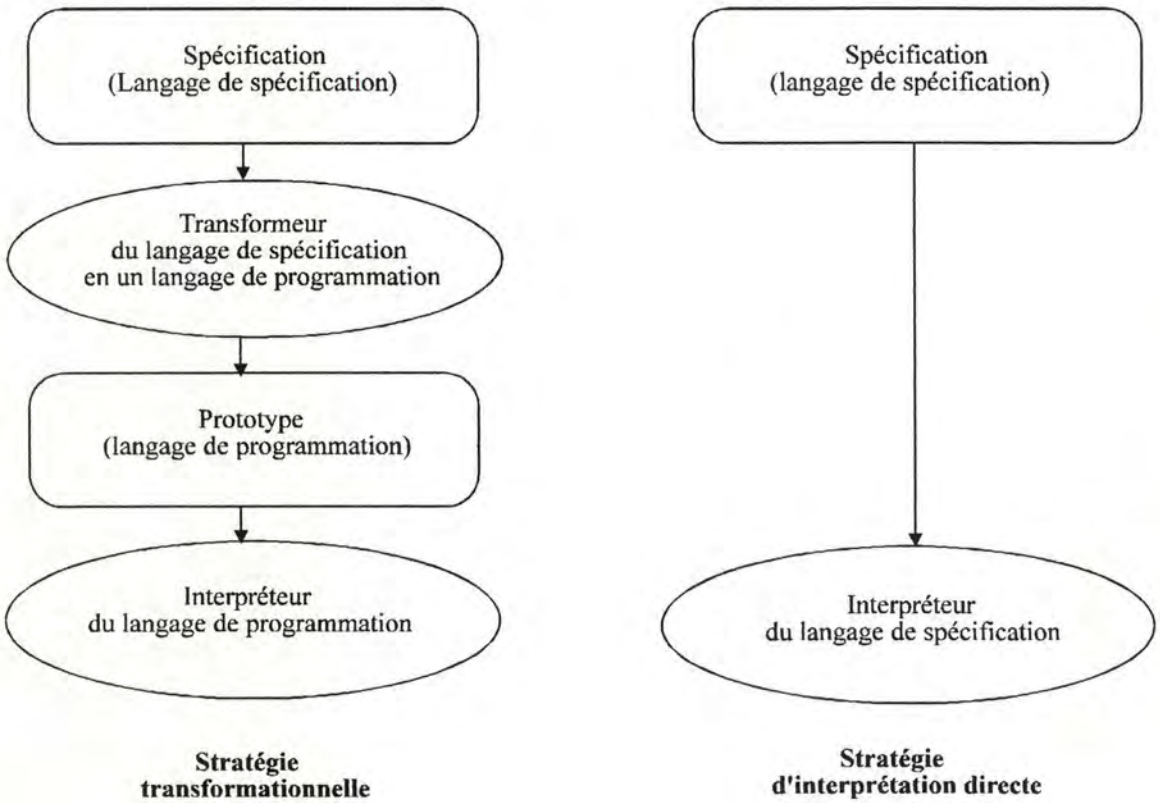


Figure 15 : Stratégies de prototypage [Hab90]

6.1.4. ENVIRONNEMENT DE PROTOTYPAGE

Il ne suffit pas d'obtenir un bon prototype, encore faut-il un bon environnement pour pouvoir l'exploiter.

Pour cela, il est capital que l'outil de prototypage présente une interface utilisateur suffisamment riche. L'utilisateur doit pouvoir :

- assister l'outil lors de la génération du prototype (lorsque cette génération ne peut être entièrement automatique) ;
- orienter l'exécution du prototype pour pouvoir tester un maximum de cas en un minimum de temps. L'outil doit donc permettre d'introduire facilement les échantillons de données ;
- adapter les spécifications selon les commentaires du client lors de l'exécution. Ceci nécessite de disposer d'une *trace* de la transformation, c'est-à-dire d'une information qui préserve les connexions entre les composants du prototype et ceux de la spécification.

En outre, le prototypage sera d'autant plus efficace que les feed-backs seront présentés de manière précise et complète, tant à la génération (trace de la transformation pour la stratégie transformationnelle) qu'à l'exécution (affichage des résultats et des erreurs).

6.2. PROTOTYPAGE OU ANIMATION DES SYSTÈMES COMPOSITES

Lorsque Habra a réalisé son étude du prototypage (dans [Hab90]), l'I.B. n'était pas encore telle qu'elle est aujourd'hui et il n'a considéré que le prototypage des spécifications de logiciel. Aujourd'hui, on admet de plus en plus que l'I.B. ne concerne pas seulement la partie informatique d'un système, ce qui a donné lieu à la notion de système composite. Pour rappel, un système composite est un système hétérogène comprenant à la fois des composants logiciels, matériels, humains, etc... (cf point 2.3.2). La notion de système composite est d'ailleurs présente dans le langage ALBERT.

Cette section a pour but d'étendre l'étude du prototypage aux spécifications de ces systèmes composites rédigées en ALBERT.

6.2.1. SYSTÈMES OUVERTS ET SYSTÈMES FERMÉS

D'une certaine manière, un logiciel est un système *ouvert* qui reçoit en entrée des arguments et fournit en sortie des résultats. Les spécifications des besoins définissent la relation entre les arguments et les résultats et tout le travail de prototypage consiste à expliciter cette relation de manière à pouvoir construire concrètement les résultats à partir des arguments fournis par l'utilisateur. Lors de l'exécution du prototype, l'analyste joue le rôle de l'utilisateur final en introduisant des échantillons de données bien choisis et teste la validité des résultats.

Par contre, un système composite est un système *fermé*. En d'autres termes, un système composite comprend à la fois le logiciel et l'utilisateur. Il n'est donc plus question d'arguments ni de résultats, encore moins de construire des résultats à partir d'arguments. La question de construire un programme exécutable qui soit une "implémentation correcte" d'une spécification des besoins n'a plus de sens quand on parle de systèmes composites. En effet, comme une telle spécification décrit un monde qui évolue de manière indéterministe et que cet indéterminisme n'est pas levé par des arguments, il est impossible de programmer le comportement du système.

Cette distinction entre système ouvert et fermé est fondamentale et amène à repenser la notion de prototype. Notons que, étant donné la spécification ALBERT d'un agent, il est possible de dériver l'ensemble (généralement infini) de toutes les vies valides de cet agent. Cependant, un tel résultat est, en pratique, impossible à exploiter par l'analyste. Par contre, il serait extrêmement intéressant de construire un programme exécutable qui simule le comportement du système composite en interaction avec l'analyste. Ainsi, l'analyste peut lever les indéterminismes et "diriger" le système dans une vie particulière parmi l'ensemble des vies possibles.

Par opposition avec le prototypage classique de logiciels, un tel type de prototypage est parfois qualifié d'*animation* (le prototype étant appelé *animateur*).

6.2.2. CLASSIFICATION DE L'APPROCHE

Nous avons vu que l'activité d'I.B. comportait quatre tâches : *élicitation*, *modélisation*, *analyse* et *validation* (cf figure 2). Le prototypage des spécifications des besoins permet de mener la tâche de validation en interaction avec le client, en lui donnant la possibilité de valider l'adéquation des spécifications par rapport à ses besoins effectifs. Cette validation est incrémentale : l'analyste exploite le prototype obtenu et le client suggère des modifications ; la version correspondante des spécifications est alors mise à jour et un nouveau prototype est généré. Le cycle "spécification-prototypage" se poursuit jusqu'à ce qu'un accord se dégage entre les deux parties. Le cahier des charges est finalisé et la phase de développement suivante peut commencer.

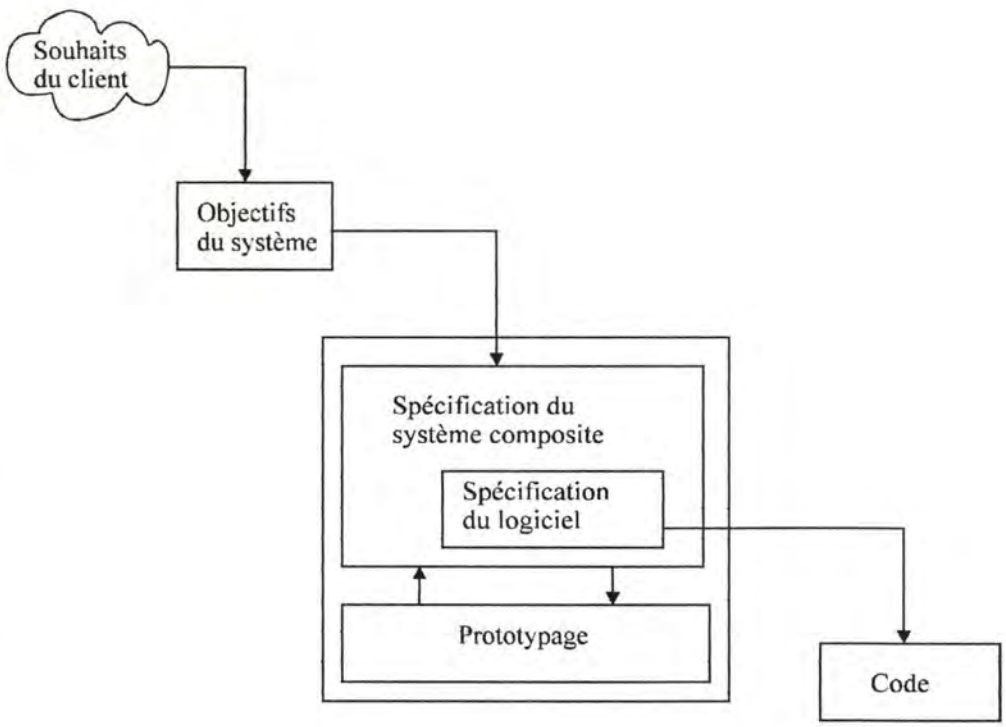


Figure 16 - Prototypage des spécifications des besoins

La grille proposée au point 6.1.1 continue à s'appliquer à notre notion étendue de prototypage. Notre approche de prototypage peut se situer dans la classification proposée au point 6.1.1 comme suit :

- l'objectif du prototypage est de valider l'adéquation des spécifications des besoins par rapport aux besoins réels du client ;
- son champ d'application est les spécifications fonctionnelles qui, pour rappel, décrivent les opérations que devra effectuer le futur système ainsi que les différentes classes d'objets impliquées dans ces opérations (cf point 2.3.3) ;
- les caractéristiques de la spécification que le prototype doit refléter sont la fonctionnalité, la cohérence et le respect des contraintes temporelles ;

- le prototype représente une "animation" correcte des spécifications des besoins ;
- il n'y a aucune relation entre le prototype et le système futur.

6.2.3. QUALITÉS ATTENDUES DU PROTOTYPE

Les mêmes qualités que celles répertoriées au point 6.1.2 seront attendues d'un prototype des spécifications des besoins.

6.2.4. STRATÉGIE DE PROTOTYPAGE

Etant donné que le langage ALBERT décrit des systèmes fermés, l'idée même de lui associer une sémantique opérationnelle n'a pas de sens. Il est dès lors impossible de construire une machine abstraite qui interprète le langage ALBERT. La stratégie d'interprétation directe doit donc être rejetée.

Dans le reste de ce mémoire, nous tentons de dégager une stratégie transformationnelle dans le but de transformer une spécification ALBERT en une spécification OBLOG équivalente. Cette stratégie doit tenir compte de la nécessité d'interagir avec l'utilisateur pour diriger l'exécution du prototype.

6.2.5. ENVIRONNEMENT DE PROTOTYPAGE

Puisque l'utilisateur est amené à diriger l'exécution du prototype, il est indispensable que l'environnement de prototypage fournisse une interface utilisateur adéquate. Cette interface doit fournir en permanence des informations à l'utilisateur sur l'état du système et sur ses diverses possibilités d'évolution.

6.3. DÉMARCHE ADOPTÉE

Pour construire le système de prototypage d'ALBERT en OBLOG, nous avons adopté successivement deux méthodes.

6.3.1. PREMIÈRE MÉTHODE

a) Principe

La première méthode que nous avons adoptée peut être qualifiée d'intuitive. Sur base de l'apparente proximité des concepts d'ALBERT et d'OBLOG (cf section 5.1), nous avons développé un ensemble de règles systématiques pour transformer une spécification ALBERT

en une spécification OBLOG. Typiquement, chaque agent donne lieu à un objet, chaque attribut ou action en ALBERT donne lieu à un attribut ou une action en OBLOG, les clauses d'effet deviennent des clauses de mise à jour, etc... Chaque élément d'une spécification ALBERT peut être transformé, de manière plus ou moins satisfaisante.

Cependant, il nous faut reconnaître que cette méthode est un échec et ne satisfait pas à l'objectif de prototypage d'une spécification ALBERT. En réalité, ce n'est qu'en expérimentant cette méthode que nous avons pris conscience des différences profondes entre ALBERT et OBLOG, d'une part, et de la nature particulière du prototypage des spécifications des besoins d'un système composite, d'autre part.

b) Première cause d'échec

Les profondes différences sémantiques entre ALBERT et OBLOG ont fait l'objet de la section 5.1.

• La première différence réside dans le fait qu'ALBERT est orienté-vie alors qu'OBLOG est orienté-transition. Dans les cas simples, nous proposons des règles de transformation qui visent à enregistrer dans l'état courant les informations sur le passé et le futur, nécessaires pour pouvoir orienter la vie de l'agent. Cependant, cette démarche ne réussit pas à traiter les engagements ou les contraintes d'évolution de l'état qui contiennent des connecteurs futurs bornés, par exemple :

```
R1 -> Sometimef(<= 10 sec) (S1 ; S2).
```

La seule solution consiste à mettre en place des mécanismes pour surveiller l'état de l'agent afin d'être capable, a posteriori, de déclarer que la vie menée est invalide.

• De toute façon, les transformations proposées occultent la deuxième différence fondamentale : la simultanéité possible des actions en ALBERT. La première méthode ne permet pas de traduire cette simultanéité en OBLOG. Or, elle fait partie intégrante de la sémantique du langage ALBERT ; si on la supprime, la spécification d'un agent n'a plus la même signification et, en particulier, ne fournit pas le même ensemble de vies valides.

c) Deuxième cause d'échec

En outre, puisque une spécification ALBERT est indéterministe par nature, cette méthode de transformation fournit une spécification OBLOG indéterministe également. Cela se traduit, dans le diagramme de comportement des objets, par le fait qu'une situation peut donner lieu à plusieurs transitions possibles et que les paramètres des actions ne sont pas instanciés (cf figure 11). Or, l'indéterminisme, en OBLOG, est levé arbitrairement à la génération du code, ce qui veut dire que le prototype exécutera une vie possible de l'agent, choisie arbitrairement parmi l'ensemble des vies possibles et donc statistiquement non-significative. Un tel prototype n'est évidemment d'aucune utilité.

Ce problème est celui que nous évoquions au point 6.2.1. S'agissant de systèmes composites, le prototypage doit prévoir une interaction avec l'utilisateur lui permettant de lever les indéterminismes et de conduire une vie particulière du système.

d) Une proposition d'évolution pour OBLOG

L'approche OBLOG a pour ambition, à long terme, de supporter toutes les phases du cycle de vie. Or il est évident qu'elle ne peut pas encore prétendre supporter l'activité d'I.B. A partir des leçons de la méthode intuitive, nous pouvons formuler une proposition d'évolution du langage et de l'outil OBLOG afin qu'ils puissent supporter l'activité d'I.B.

• Nous avons vu (cf point 2.4.1) qu'un langage d'I.B. devait avoir des qualités d'expressivité, de structuration et de formalisation. OBLOG possède certainement les deux dernières et on pourrait imaginer d'enrichir l'expressivité d'OBLOG avec les notions d'obligation et d'engagement, etc., qui se sont avérées utiles en I.B. Quant à la simultanéité des actions, signalons qu'une des premières versions d'OBLOG proposait la notion d'actions concurrentes ou parallèles (voir [Ser91a]).

• Cependant, nous savons à présent que le caractère exécutable d'une spécification OBLOG n'est d'aucune utilité si cette spécification présente de l'indéterminisme. Pourtant, il serait possible de construire un outil d'animation qui exécute la spécification en interaction avec l'utilisateur, de telle sorte que celui-ci soit interrogé chaque fois que la spécification comporte un choix non-déterministe. Un tel outil permettrait réellement le prototypage de spécifications des besoins en OBLOG.

e) Intérêt de l'expérience

Malgré que cette méthode ait échoué, nous avons choisi d'en exposer les grandes lignes, au chapitre 7, pour trois raisons importantes :
- premièrement, elle nous a permis de mieux comprendre, par l'expérimentation, la différence entre ALBERT et OBLOG ;
- deuxièmement, elle nous a permis d'émettre une proposition quant à l'évolution du langage et de l'environnement OBLOG ;
- troisièmement, cette expérience, et ses conclusions, nous ont été d'une grande utilité pour mettre en oeuvre une méthode alternative. (Certaines transformations de constructions ALBERT en OBLOG sont d'ailleurs réutilisées dans le cadre de cette deuxième méthode.)

6.3.2. DEUXIÈME MÉTHODE

Puisque l'approche OBLOG ne convient pas pour animer une spécification ALBERT, il reste à construire nous-même un animateur ALBERT. Cet animateur pourrait être implémenté en C ou en Pascal ; nous avons choisi de l'implémenter... en OBLOG. Mais, à la différence de la première méthode, le choix d'OBLOG n'est plus justifié par son apparente proximité avec ALBERT, mais uniquement parce qu'OBLOG est un langage de haut niveau qui permet de spécifier et d'implémenter facilement une application.

Cette deuxième méthode consiste à construire, en OBLOG, un modèle générique d'une spécification ALBERT, aussi bien dans ses aspects statiques que dynamiques. (Il s'agit d'un méta-modèle dans la mesure où une spécification est déjà un modèle du monde réel.) Cette démarche répond aux deux causes d'échec de la première méthode :

- le modèle enregistre tous les états et tous les changements d'un agent (il est "orienté-vie") et gère la simultanéité possible des actions ;
- le modèle comprend explicitement une interaction avec l'utilisateur qui contrôle le temps et lève les indéterminismes.

Etant donné une spécification ALBERT particulière, il suffit d'instancier le modèle générique selon cette spécification pour en obtenir un animateur. Plus précisément, on obtient la spécification OBLOG de l'animateur. Il reste ensuite à produire le code exécutable correspondant (y compris le code de l'interface utilisateur) à l'aide du générateur de code intégré dans l'environnement OBLOG.

Remarque : le lecteur intéressé davantage par cette deuxième méthode de prototypage peut, sans difficulté, passer directement au chapitre 8 quitte à revenir par la suite au chapitre 7.

7. PREMIÈRE MÉTHODE

7.1. INTRODUCTION

Ce chapitre présente la première méthode, intuitive, que nous avons suivie pour prototyper ou animer une spécification ALBERT en OBLOG[1].

7.1.1. EXPLICATION DE LA MÉTHODE

Concrètement, le processus de transformation d'ALBERT vers OBLOG doit comprendre deux phases : une phase d'*opérationalisation* et une phase de *traduction*.

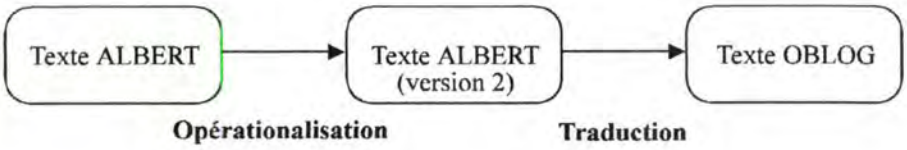


Figure 17

a) Phase d'opérationalisation

Cette phase consiste à dégrader la spécification ALBERT initiale de façon à obtenir une spécification équivalente, toujours exprimée en ALBERT mais n'utilisant plus que des concepts ayant leur équivalent en OBLOG. On dira d'une telle spécification qu'elle est *opérationnelle* d'un point de vue OBLOG.

Deux cas doivent être envisagés :
- soit une propriété écrite en style observationnel doit être exprimée de manière constructive. C'est typiquement le cas des engagements qui n'ont pas d'équivalent en OBLOG et pour lesquels il faudra mettre en place des mécanismes pour surveiller l'évolution de la vie de l'agent ;

[1] Rappelons que nous utilisons la version OBLOGlight.

- soit une propriété écrite en style constructif doit être réécrite dans un style toujours constructif mais moins riche. Par exemple, certaines formules de la logique du premier ordre devront être reformulées à l'aide du sous-ensemble de cette logique supporté par OBLOG.

Nous avons tenté de dégager un ensemble de règles d'opérationalisation. Ces règles ne comportent que des transformations *sémantiques* et idéalement, elles comportent toutes les transformations sémantiques, laissant les transformations syntaxiques pour la phase suivante. Comme la version opérationalisée du texte ALBERT est une version dégradée, il se peut que sa sémantique soit moins riche que celle de la version initiale.

b) Phase de traduction

Une fois opérationalisée, la spécification est prête à être traduite en OBLOG, chaque concept ayant un équivalent en OBLOG. Nous avons dégagé l'ensemble des règles de traduction à appliquer. Ces règles comportent toutes les transformations *syntaxiques* et uniquement celles-là.

c) Justification de la démarche

Ce souci de décomposition se retrouve dans tout problème informatique et se justifie ici pour deux raisons. Le fait de distinguer les transformations sémantiques et syntaxiques permet, d'une part, de mieux maîtriser le problème de la transformation d'ALBERT vers OBLOG et offre, d'autre part, de meilleures garanties de traçabilité en rendant plus compréhensible le processus de transformation.

7.1.2. NOTATIONS

Dans les règles de transformations, nous utiliserons les notations suivantes :

ALBERT	OBLOG	Désignation
U, V	U, V	Agent/objet
A, B	a, b	Attribut
R, S	r, s	Action
T	T	Type de données
i, j, m, n	i, j, m, n	Indice
c	c	Constante
x, y	x, y	Argument/paramètre
e	e	Expression
t	t	Terme
p, q	p, q	Proposition
α, β	α, β	Formule

Pour mieux les distinguer, les mots créés par le processus de transformation sont en anglais.

7.1.3. PLAN DE RECHERCHE

Nous rechercherons les règles de transformation en procédant du plus simple vers le plus compliqué, de manière incrémentale. La transformation des déclarations ira sans peine (section 7.2). Quant aux contraintes (section 7.3 et suivantes), il sera possible, à partir d'un noyau de règles de traduction, principalement celles concernant les effets (section 7.3) et les préventions (point 7.5.1), d'opérationaliser les autres cas de figures de manière à se ramener aux cas de base pour lesquels on dispose de règles de traduction. Finalement, la méthode sera évaluée à la section 7.10.

Nous avons délibérément choisi cette approche intuitive, pour faciliter la compréhension. De plus, faute de place, nous ne développerons pas en détail toutes les transformations. On trouvera, en annexe (section A.8.), une illustration de la méthode sur les spécifications des besoins associées au véhicule de l'atelier.

7.2. TRANSFORMATION DES DÉCLARATIONS

7.2.1. ATTRIBUTS

La transformation des attributs serait immédiate s'il n'y avait le problème de la visibilité. En effet, en OBLOG, il n'y a qu'une alternative :
- soit l'attribut est privé et ne peut être manipulé que par l'objet lui-même ;
- soit il est public auquel cas la communauté entière peut y accéder (même si cet accès peut être soumis à une condition).

Autrement dit, il n'y a pas moyen, pour un objet, d'exporter un attribut vers certains objets et pas vers d'autres. Pour cette raison, nous avons été amené à poser l'hypothèse de transparence, c'est-à-dire à rendre publics tous les attributs d'un agent, et à gérer explicitement les autorisations liées à la perception et la publicité des attributs (voir les transformations de la perception et de la publicité, plus loin dans ce chapitre).

7.2.2. ATTRIBUTS DÉRIVÉS

Un attribut dérivé en ALBERT devient naturellement un attribut dérivé en OBLOG. Nous verrons plus loin dans ce chapitre comment transformer les règles de dérivation.

7.2.3. ACTIONS

a) Principe général

Chaque action interne ou externe en ALBERT donne lieu à une action respectivement privée ou publique en OBLOG. Chaque argument d'une action en ALBERT donne lieu à un

paramètre d'action en OBLOG, auquel on donne artificiellement un nom (vu que les arguments n'ont pas de nom en ALBERT).

b) Visibilité des actions

En ALBERT, la perception par un agent des actions des autres agents de la société permet de synchroniser les vies des différents agents. En OBLOG, cette synchronisation est assurée par le mécanisme d'appel.

Exemple : dans l'agent *U*, la clause

```
S(x) -> V
```

qui rend l'action *S* visible pour l'agent *V*, est traduite en

```
s(x:T_x)
    >> v.s(x),
```

où la variable *x* a été remplacée par le paramètre *x*. Cet appel est synchrone, ce qui signifie que *s* et *v.s* ont lieu simultanément, reproduisant ainsi fidèlement le mécanisme de la perception en ALBERT.

c) Actions de vie, de mort et d'échec

En OBLOG, toute vie d'un objet doit explicitement commencer par une action de naissance et peut se terminer par une action de mort. En ALBERT, aucune marque syntaxique ne permet de repérer une éventuelle action initiale ou finale. Rien ne nous empêche cependant d'ajouter artificiellement, dans la version opérationalisée, deux actions `Live` et `Die`, qui seront traduites en OBLOG respectivement en une action de naissance et une action de mort. Nous verrons, au moment de transformer les contraintes, les avantages liés à cet artifice.

De plus, il ne sera pas possible de construire une version opérationnelle de la spécification OBLOG qui ait une sémantique équivalente à la version originale. C'est pourquoi nous introduisons, dans la version opérationalisée, une action d'échec `Fail`. Toute occurrence de l'action `Fail` signifie que la vie de l'agent n'est pas une vie valide par rapport à la spécification originale.

d) Comportement

En ALBERT, les déclarations décrivent un ensemble de vies possibles dans lesquelles les actions peuvent avoir lieu sans aucune contrainte. De manière équivalente le langage OBLOG associe à chaque objet un comportement par défaut rendant toutes les actions possibles (figure 18).

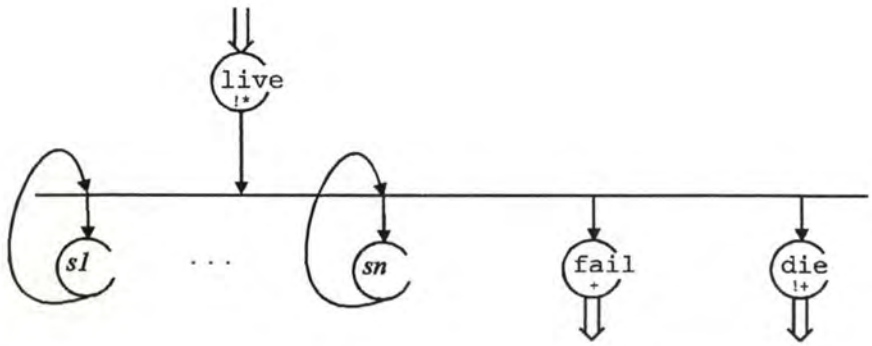


Figure 18 - Comportement par défaut d'un objet

7.3. TRANSFORMATION DES EFFETS

Les concepts d'"effets" et de "mise à jour" étant synonymes, la traduction des effets est généralement facile :

```
S(x) : (p) => A = t
```

devient

```
s(x:T_x)
      {p} => A := t,
```

où la variable *x* a été remplacée, dans *p* et dans *t*, par le paramètre *x*.

Dans l'exemple ci-dessus, *x* est une variable implicitement quantifiée universellement. Mais dans certains cas, on désire faire dépendre l'effet de l'action de certaines valeurs particulières des arguments de cette action. Cependant, il y a toujours moyen de se ramener au cas général en exprimant la contrainte sur les arguments à l'intérieur de la précondition sur l'effet.

Exemple :

```
S(5) : (p) => A = t
```

devient

```
S(x)
      {p AND x = 5} => A := t.
```

Nous appellerons *normalisation* d'une clause le processus qui consiste à généraliser la clause de telle sorte que plus aucun argument ne soit instancié.

7.4. TRANSFORMATION DES INITIALISATIONS

Etant donné qu'on a introduit l'action `Live`, il suffit de déplacer les clauses d'initialisation de la rubrique **`Init`** à la rubrique **`Effects`**, comme effets de l'action `Live`, et de les transformer comme les autres effets.

7.5. TRANSFORMATION DE LA RESPONSABILITÉ

7.5.1. PRÉVENTIONS

Une prévention sera traduite naturellement en utilisant une précondition de transition :

F(*S*(*x*) / *p*)

est traduit en

s(*x*:*T_x*)
? NOT *p*

où la variable *x* a été remplacée, dans *p*, par le paramètre *x*.

Certaines préventions doivent être normalisées avant d'être traduites.

7.5.2. OBLIGATIONS

OBLOG n'offre pas le concept d'obligation. Cependant, il est possible de simuler le caractère obligatoire d'une action en interdisant toutes les autres.

a) Principe

Considérons un exemple simple : l'agent *U* possède deux actions *S* et *R* telles que :

Responsibility
O(*S*(*e*) / *o*)
F(*S*(*x*) / *p*)
F(*R*(*y*) / *q*).

Ces trois clauses sont opérationalisées comme suit :

State Behaviour
not *o* and *p'*

Responsibility
F(*S*(*x*) / *o* and *x*<>*e*)
F(*S*(*x*) / *p*)
F(*R*(*y*) / *q* or *o*),

avec $p' = p$ où toute occurrence de *x* est remplacée par *e*.

Lorsque la condition d'obligation *o* est vérifiée, l'action *R* est interdite, de même que l'action *S* avec d'autres arguments que ceux imposés par la clause d'obligation. La prochaine action ne peut donc être que *S*(*e*), à moins que *S* ne soit interdite par *p* ; c'est pourquoi on ajoute une contrainte sur l'état afin d'empêcher d'avoir *o* et *p'* en même temps (dans la spécification originale, *S* ne peut jamais être à la fois interdite et obligatoire). (Nous verrons, plus loin dans ce chapitre, comment transformer les contraintes sur l'état.)

En agissant de la sorte, on peut dire que, selon la spécification opérationalisée, *S*(*e*) n'est pas vraiment obligatoire mais du moins "inévitable" lorsque *o* est vraie.

b) Généralisation

Ce principe d'opérationalisation peut aisément être généralisé quel que soit le nombres de clauses "O". Cependant, il n'est pas possible d'exprimer la contrainte sur l'état lorsque la clause d'obligation n'instancie pas les arguments (dans l'exemple, *x* était instancié à *e*).

7.5.3. OBLIGATIONS EXCLUSIVES

Par définition, une clause

X(*S*(*x*) / *p*)

correspond aux clauses

O(*S*(*x*) / *p*)
F(*S*(*x*) / not *p*)

pour lesquelles nous venons de proposer des transformations.

7.6. TRANSFORMATION DE LA PERCEPTION ET DE LA PUBLICITÉ

7.6.1. ATTRIBUTS

Comme nous l'avons déjà mentionné, il n'est pas possible en OBLOG d'offrir la visibilité d'un attribut à certains objets et pas à d'autres, encore moins de restreindre cette visibilité en fonction de l'objet concerné.

La solution que nous avons adoptée consiste
- d'une part à transformer toute clause de publicité d'un attribut en une clause de perception correspondante, chez les agents importateurs ;
- à introduire, dans la déclaration d'un agent, un attribut dérivé pour chaque attribut importé par l'agent, dont la règle de dérivation évalue l'autorisation d'importer l'attribut ;
- à remplacer, dans la spécification, toute les références à un attribut importé par le nom de l'attribut dérivé correspondant.

Exemple : dans U, la clause d'exportation

```
X(A.v / p)
```

est remplacée par une clause d'importation équivalente dans V

```
X(u.A / p).
```

Cette dernière est remplacée à son tour par un attribut dérivé :

```
u_A : T_u_A
      (p) => u_A = u.A
      (NOT p) => u_A = null.
```

Enfin, toute occurrence de "*u.A*" est remplacée par "*u_A*".

7.6.2. ACTIONS

a) Principe

Rappelons que la visibilité des actions correspond en OBLOG au mécanisme d'appel. Exemple : dans U, la clause

```
S(x) -> V
```

qui rend l'action S visible pour l'agent V, est traduite en

```
s(x:T_x) >> v.s(x).
```

Pour U, restreindre la visibilité de l'action S revient à conditionner l'appel :

```
X(S.v / p)
```

est traduit en

```
s(x:T_x)
      {p} >> v.s(x).
```

Etant donné l'hypothèse de transparence, on peut remplacer les clauses de perception d'action par des clauses de publicité correspondantes. Supposons que la spécification de V contient une clause de perception relative à S :

```
F(u.S / q).
```

Il suffit d'adapter, dans U, la clause de publicité correspondante :

```
X(S.v / p and not q),
```

ce qui traduit bien le fait que la visibilité de *u.S* par V n'est effective que lorsque U offre cette visibilité et que V la prend en compte.

b) Généralisation

Ces principes peuvent aisément être adaptés à d'autres types de clause, en particulier au cas où la publicité est offerte à plusieurs agents (ou à plusieurs instances d'une classe d'agents), grâce au mécanisme de *multicast* (appel simultané de plusieurs objets).

Exemple : la clause

$S(x)$ -> $\{V1, \ldots, Vn\}$

est traduit en

$s(x{:}T\_x)$ >> ALL[v:UNION($V1, \ldots, Vn$)].$s(x)$.

7.7. TRANSFORMATION DES FORMULES

Jusqu'à présent, nous avons été capable de traduire en OBLOG des contraintes faisant intervenir une condition p, où p est une proposition OBLOG. Cette section est consacrée à la transformation d'une formule ALBERT quelconque en une proposition OBLOG. Ce type d'opérationalisation nous permettra de généraliser les transformations vues jusqu'ici.

En synthèse, la logique OBLOG est limitée pour les raisons suivantes :
- pas de connecteurs d'implication ni d'équivalence ;
- pas de variables si ce n'est à l'intérieur d'une requête ;
- pas de quantificateur universel ;
- quantification existentielle uniquement sur un ensemble de données tangible (par exemple la communauté d'objets ou un attribut de type LIST) ;
- types de donnée finis ;
- pas d'extensions temporelles.

En pratique, la transformation des formules de la logique du premier ordre (LPO) ne pose pas de problèmes. Il en va tout autrement par contre de la logique temporelle.

Les règles de déduction classiques permettent de se débarrasser de certaines restrictions relatives aux formules de la logique du premier ordre (LPO). Par contre, les formules temporelles sont beaucoup plus difficiles à manipuler et il existe peu de règles de déduction qui nous soient utiles. Bien sûr, on peut toujours exiger au spécifieur d'opérationaliser lui-même les formules temporelles. Pour notre part, nous proposons des heuristiques de transformations mais qui ne sont valables que si on accepte de distinguer une logique temporelles dans le passé (LTP) et une logique temporelle dans le futur (LTF), autrement dit, de ne pas mélanger dans une formule des connecteurs du passé avec des connecteurs du futur.

7.7.1. FORMULES LPO

L'absence de connecteurs d'implication et d'équivalence peut aisément être surmontée en vertu des règles logiques de déduction (reprises telles quelles en tant que règles d'opérationalisation) :

α => β ≡ not α or β
α <=> β ≡ (α and β) or (not α and not β).

De même, la règle suivante permet de se passer de quantificateur universel :

```
(forall x:T_x) (α)    ≡ not (exists x:T_x) (not α).
```

Les variables utilisées en ALBERT comme arguments des actions deviennent, en OBLOG, les paramètres des actions correspondantes.

Le fait de disposer, en OBLOG, de types de donnée uniquement finis est une restriction insurmontable mais qui, en pratique, ne devrait pas poser de problèmes (en effet, la plupart des applications se contentent de types de donnée finis).

Finalement, il reste un cas non-traité : la quantification d'une variable qui n'est pas un argument d'action et qui n'appartient pas à un ensemble de données tangible. Dans ce cas, une intervention du spécifieur sera nécessaire pour opérationaliser la formule.

7.7.2. FORMULES LTP

a) Principe

Puisque OBLOG n'offre pas de connecteurs temporels, il s'agit de mettre en place, lors de la phase d'opérationalisation, des mécanismes pour suivre l'évolution de l'état de l'agent. L'idée est de remplacer, si c'est possible, une formule temporelle par un attribut booléen qui contient à tout moment la valeur de vérité de la formule.

Considérons un exemple simple : toute occurrence, dans le texte ALBERT, de "`sometimep` α", où α est une formule LPO, est remplacé par "`Sometimep_`α". La spécification est complétée par la déclaration :

```
Attributes
Sometimep_α : BOOL
```

et les contraintes (pour toute action R qui a un effet sur un attribut de α) :

```
Init
Sometimep_α = α_I
Effects
R: (α_R) => Sometimep_α = true
```

où α_I représente la valeur initiale de α et α_R la valeur de α après une occurrence de R.

Cette démarche d'opérationalisation vaut pour toute formule LPO α telle qu'il soit possible d'en contrôler les changements de valeur, en d'autres termes, lorsque la valeur de α n'est modifiée qu'à la suite d'un effet de l'agent. La formule α ne peut donc pas contenir d'attributs importés (d'ailleurs, le langage ALBERT ne permet pas de consulter la vie antérieure d'un autre agent).

b) LTP bornée

On tient un raisonnement similaire pour les connecteurs bornés, en introduisant artificiellement un agent `Clock` qui offre un attribut `Time` à tous les agents de la société.

Exemple : "`sometimep(<= d)`" est remplacé par "`Sometimepb_α`" tel que :

```
Attributes
LTF_α : TIME
Time : TIME <- Clock
Derived Attributes
Sometimepb_α : BOOL = α or (clock.Time - LTF_α <= d)
Init
LTF_α = -∞
Effects of Actions
R:(α and not pR) => LTF_α = clock.Time
```

où `LTF_α` représente le dernier temps où la formule est devenue fausse (*Last Time False*) et *pR* la valeur de α après une occurrence de *R*.

c) Généralisation

Comme une formule LTP est transformée en un attribut, rien n'empêche de composer ces formules comme on le ferait avec des attributs, au moyen des connecteurs `not`, `and`, `or`, `=>` et `<=>`. Exemple :

`sometimep` *p1* `and not always` *p2* `=> sometimep` *p3*

devient

`Sometimep_`*p1* `and not Always_`*p2* `=> Sometimep_`*p3*.

De même, on peut utiliser un quantificateur si ce quantificateur peut être traduit en OBLOG :

`(exists` *x*`:`*T_x*`) (In(`*X_set*`,` *x*`) and sometimep` *α(x)*`)`

devient

`(exists` *x*`:`*T_x*`) (In(`*X_set*`, <`*x*`, Sometimep_α>))`

après avoir modifié le type de *X_set* de façon à enregistrer la valeur de `sometimep` α pour chaque élément de l'ensemble.

Les connecteurs temporels LTP peuvent être combinés entre eux s'ils ne sont pas bornés. Exemple :

`sometimep(alwaysp_α),`

qui exprime que α a été vrai jusqu'à un certain moment du passé, peut être transformé successivement en

`sometimep(Alwaysp_α)`

comme d'habitude, puis, étant donné que `Alwaysp_`α est un attribut au même titre que les autres, en

`Sometimep_always_`α.

Par contre, il n'est pas possible de transformer

`sometimep(alwaysp(<=`d`)` α`)`

car la transformation d'un connecteur temporel borné fait intervenir l'attribut `clock.Time` et on a vu que notre démarche de transformation ne s'applique pas à des formules contenant des attributs importés.

Remarque : certaines formules temporelles peuvent être simplifiées en appliquant les règles de la logique temporelle. Kröger en propose un certain nombre dans [Krö87]. La logique temporelle classique ne considère que le temps futur mais il est raisonnable de penser que ces règles peuvent être adaptées si on ne fait que renverser l'échelle de temps.

Quoi qu'il en soit, la transformation des formules temporelles ne peut être entièrement automatisée et doit être guidée par l'utilisateur.

7.7.3. FORMULES LTF

Si on fait l'hypothèse de ne pas combiner dans une même formule des connecteurs LTP et LTF, on ne peut rencontrer une formule LTF que comme contrainte sur l'état. En effet, il n'y a pas de sens à faire référence au futur dans une clause d'effet ou de responsabilité. Les contraintes sur l'état font l'objet de la prochaine section.

7.8. TRANSFORMATION DES CONTRAINTES SUR L'ÉTAT

Les contraintes sur l'état d'un agent (reprises sous la rubrique **`State Behaviour`**), sont de deux types :
- les contraintes statiques, ou invariants, qui sont vérifiées dans tous les états possibles d'un agent ;
- les contraintes dynamiques, qui portent sur l'évolution du système.

Ces contraintes restreignent implicitement le comportement admissible d'un agent en ce sens que certains changements, donc certaines actions, sont interdits. Il ne sera pas toujours possible, en OBLOG, de prévenir ces actions. Une solution de compromis consiste à surveiller la valeur d'une contrainte pour être capable, au besoin, de détecter a posteriori qu'elle n'est pas vérifiée. C'est dans ce but que nous avons introduit, dans la version opérationalisée du texte ALBERT, une action `Fail`. Toute occurrence de `Fail` signifie que la vie de l'agent n'est pas une vie valide par rapport à la spécification originale.

7.8.1. CONTRAINTES STATIQUES

OBLOG permet d'exprimer des contraintes sur les attributs, mais ces contraintes ne peuvent pas faire référence à des observations. En effet, considérons la contrainte

a > 5.

D'un point de vue opérationnel, cette contrainte crée une précondition sur toutes les actions qui mettent à jour l'attribut a, empêchant qu'une action donne à a une valeur interdite. Considérons à présent la contrainte

$a + v.b > 5$.

Il est impossible de contraindre l'attribut b, puisque b appartient à un autre objet, et il est également impossible de surveiller constamment la valeur de b pour adapter la valeur de a afin de respecter la contrainte.

Concrètement, il nous reste à remplacer une telle contrainte, dans la spécification opérationalisée, par une obligation relative à l'action `Fail` ; pour l'exemple ci-dessus :

`X(Fail / not (`$A + v.B > 5$`))`.

7.8.2. CONTRAINTES DYNAMIQUES

Ces contraintes sont exprimées au moyen de formules LTP ou LTF. Nous avons déjà vu (point 7.7.2) quelques heuristiques pour se débarrasser des formules LTP.

Certaines contraintes LTP peuvent avantageusement être remplacées par une contrainte LTF équivalente. Exemple :

α `=> alwaysf` β ≡ `sometimep` α `=>` β.

C'est le cas pour toutes les connecteurs `always` et `until`, bornés ou non.

Pour les connecteurs `sometimef` et `sometimef(<=`d`)`, il convient d'introduire des préventions sur l'action `Die`, de façon à empêcher que la vie de l'agent ne se termine avant d'avoir respecté la contrainte. Un connecteur `sometimef(>`d`)` donnera lieu à un invariant, fonction de `clock.time`, qui interdit de dépasser le temps limite d avant d'avoir respecté la contrainte.

7.9. TRANSFORMATION DES ENGAGEMENTS

Les *engagements* permettent d'exprimer un lien de *causalité* entre certaines occurrences d'actions. *R1* -> *S1* ; *S2* se lit "*R1* cause *S1*, puis *S2*". On appellera *R1* la cause et *S1* et *S2* les conséquences de l'engagement.

7.9.1. IDÉE INTUITIVE

Jusqu'à présent, nous n'avons eu affaire qu'à des diagrammes de comportement plats. Intuitivement, on pourrait penser que les engagements vont rompre cette monotonie et exploiter la puissance d'expression d'OBLOG pour représenter les enchaînements d'actions.

Considérons l'engagement *R1 -> S1 ; S2* et sa traduction intuitive (où *R2* est une action quelconque en-dehors de l'engagement) :

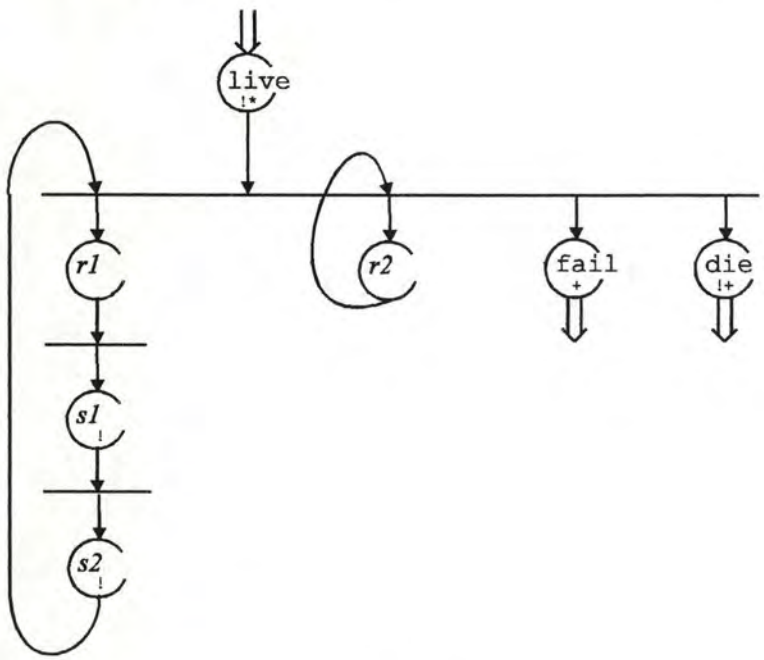


Figure 19

Malheureusement, cette traduction ne respecte pas la spécification originale puisque, après avoir commencé un engagement, l'agent ne pourrait plus accomplir que les actions liées à cet engagement (ici, *S1* puis *S2*), au détriment des autres actions (ici, *R2*) ou d'une autre "instance" de la même "classe d'engagements". En d'autres termes, la vie de l'agent ne pourrait accepter que des engagements sous forme de séquences non-entrecoupées d'actions, comme illustré à la figure 20 :

Figure 20 - Séquences non-interrompues

Remarque : bien qu'insatisfaisante en toute généralité, cette solution peut être envisagée pour décrire le comportement d'un agent non-partageable (qui ne peut mener qu'une opération à la fois). Il est interdit au véhicule, par exemple, d'engager simultanément plusieurs mouvements ou un mouvement et un déchargement, etc.

7.9.2. LA SOLUTION "PON-COT"

En réalité, les engagements ont été introduits dans le langage pour une raison évidente d'expressivité. Il est fastidieux mais néanmoins possible de les exprimer uniquement en termes de préventions/obligations, pour lesquelles nous disposons de méthodes de traduction.

a) Principe

Considérons à nouveau un engagement simple *R1 -> S1 ; S2*, tel que les actions *R1*, *S1* et *S2* sont distinctes et ne possèdent pas d'arguments. En voici une transformation possible :

```
Attributes
PON : TAB([1..2], INT)
Init
PON[1] = 0, PON[2] = 0
Effects of Actions
R1: PON[1] = PON[1] + 1
S1: PON[1] = PON[1] - 1
    PON[2] = PON[2] + 1
S2: PON[2] = PON[2] - 1
Agents Responsabilities
F(S1 / PON[1] = 0)
F(S2 / PON[2] = 0)
F(Die / PON[1] > 0 or PON[2] > 0).
```

Le tableau PON reprend, pour chacune des conséquences, le nombre d'occurrences possible *(Possible Occurrences Number)*, c'est-à-dire le nombre de fois que l'action peut se produire à partir du temps présent. Une prévention empêche une action d'avoir lieu si son PON vaut 0 (en vertu de l'hypothèse de la liaison des conséquences à leur engagement, cf point 3.5.3). De plus, une prévention empêche l'agent de mourir avant d'avoir honoré tous ses engagements.

La figure 21 montre une vie possible de l'agent :

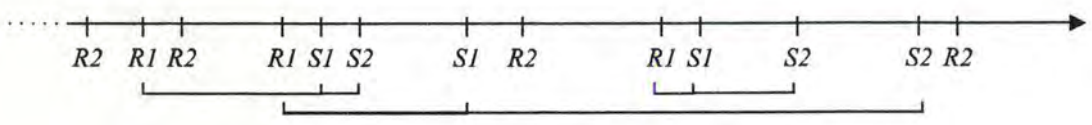


Figure 21 - Chevauchement et emboîtement possibles

b) Prise en compte du temps

L'introduction d'une contrainte de performance

R1 `-> sometimef(<=` *d*`)` *S1* `;` *S2*

induit les ajouts suivants :

Attributes
`COT : SEQ[TIME]`

Init
`COT = []`

Effects of Actions
R1`: COT = Append(COT, clock.Time)`
S2`: COT = Remove(COT, First(COT))`

Responsibility
`O(`*S2* `/ not Empty(COT) and clock.Time - First(COT) =` *d*`).`

La suite COT enregistre au fur et à mesure les temps des occurrences des causes *(Causes Occurrences Times)*, c'est-à-dire les différents moments où un engagement a été amorcé. L'obligation contraint l'agent à honorer l'engagement dans le temps imparti. Une fois l'engagement honoré, le temps correspondant est supprimée de la suite.

Remarque : lorsqu'un agent mène de front plusieurs instances d'une classe d'engagements, il n'est pas toujours possible de rattacher telle occurrence d'action à l'une ou l'autre instance. Nous avons opté pour une stratégie FIFO (*First In First Out)*, celle qui nous semblait la plus naturelle. La figure 22 illustre cette stratégie appliquée à la vie de la figure 21.

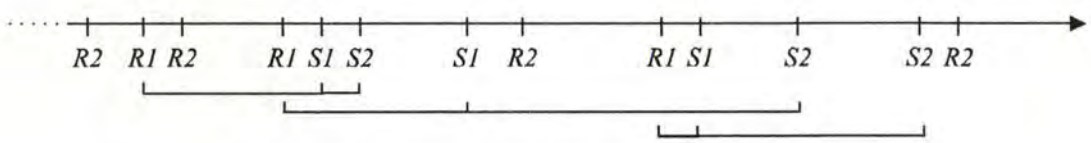


Figure 22 - Chevauchement mais non emboîtement

c) Généralisation

En adaptant les effets et les préventions relatives au tableau PON, on peut facilement prendre en compte :
- les répétitions :

R1 `->` *S1* `;` *S1*

- les alternatives :

R1 `->` *S1* `xor` *S2*

- le parallélisme :

R1 `->` *S1* `||` *S2.*

Par contre, la transformation d'engagements avec arguments n'est pas toujours immédiate. Considérons trois exemples d'engagements :

```
R1 -> S1(x) ; S2(y)          (E1)
R1 -> S1(e1) ; S2(e2)        (E2)
R1 -> S1(z) ; S2(z).         (E3)
```

L'engagement (E1) n'apporte aucun difficulté supplémentaire et peut être transformé comme vu précédemment. Il en est de même pour (E2) si on normalise les clause d'effets et de prévention créées par la transformation. Un cas pose problème, c'est celui où une même variable est utilisée comme argument de plusieurs actions, comme dans l'engagement (E3). Cela signifie qu'il faut mémoriser dans un attribut la valeur de ce paramètre durant toute la durée de l'engagement.

Rappelons que nous avons considéré deux types d'extension du langage en ce qui concerne les engagements : les processus et la structure "*if-then-else*". Tout processus sera remplacé par sa définition dans laquelle on aura remplacé les paramètres formels par les paramètres actuels. Un engagement de la forme

```
R1 -> if α then S1 else S2
```

peut être transformé de manière analogue à un engagement de la forme

```
R1 -> S1 xor S2
```

en adaptant les effets et préventions de *S1* et *S2* à la condition α. En l'absence de clause `else`, il suffit de poser *S2* = `Nil`, où `Nil` est une action "bidon".

Enfin, tous ces principes s'appliquent quel que soit le nombre de conséquences. De même, en introduisant un deuxième tableau pour enregistrer la survenances des causes, on peut généraliser le nombre de causes. Considérons un exemple simple :

```
R1 ; R2 -> S1 ; S2.
```

Il suffit de conditionner l'effet

```
R2: PON[1] = PON[1] + 1
```

de manière à n'incrémenter `PON[1]` que si l'occurrence de *R2* a bien été précédée d'une occurrence de *R1*.

7.10. EVALUATION

Le but de ce chapitre était de montrer, dans les grandes lignes, la méthode transformationnelle que nous avons choisie de suivre dans un premier temps. Nous avons déjà expliqué pourquoi cette méthode ne convient pas pour animer une spécification qui décrit un système fermé et qu'il fallait nécessairement prévoir une interaction avec l'utilisateur pour lever les indéterminismes. Indépendamment de ce problème, nous pouvons évaluer la valeur du processus de transformation.

Si on considère séparément chaque fragment d'une spécification ALBERT, nous sommes effectivement parvenu à proposer une suite de transformations pour aboutir à une traduction similaire en OBLOG, en distinguant bien les transformations sémantiques et syntaxiques.

Du point de vue de la syntaxe et de la structure, la spécification OBLOG globale qui résulte de ce processus de transformation est proche de la spécification originale, sauf en ce qui concerne les formules temporelles et les engagements, qui introduisent de nombreux éléments supplémentaires. Ces deux cas mis à part, la traçabilité du processus est assurée.

D'un point de vue sémantique cependant, plusieurs types de limitation doivent être considérés.

• Premièrement, certaines transformations ne peuvent être automatisées et demandent l'intervention de l'analyste. C'est le cas :
- des formules qui comportent une quantification sur un ensemble de données non tangible (c'est-à-dire qui n'appartient pas à la mémoire du système) ;
- des formules temporelles mélangeant des connecteurs LTP et LTF ;
- de certaines formules temporelles complexes.

• Deuxièmement, certaines transformations affaiblissent le sens de la spécification de départ. Ainsi, les engagements comportant une contrainte de performance sont traités selon une stratégie FIFO, ce qui n'est pas imposé par la sémantique d'ALBERT.

• Troisièmement, la spécification opérationalisée autorise des vies qui ne sont pas valides par rapport à la spécification originale. (Il est vrai que ces vies se distinguent par le fait, que tôt ou tard, elles contiennent une occurrence de l'action `Fail`.) Ce problème est dû au fait qu'il est impossible de transformer une spécification orientée-vie en une spécification orientée-transition.

• Enfin, et surtout, la spécification OBLOG ne permet pas que deux actions aient lieu simultanément, ce qui dénature sérieusement la spécification de départ.

En conclusion, il faut rappeler que le processus de transformation directe d'ALBERT vers OBLOG est voué à l'échec, compte tenu des différences profondes entre ces deux langages. Cependant, certaines règles de transformation, prises isolément, s'avèrent intéressantes et nous les retrouverons au chapitre suivant, consacré à la deuxième méthode de transformation.

8. DEUXIÈME MÉTHODE

8.1. INTRODUCTION

Ce chapitre aborde le prototypage d'ALBERT en OBLOG[1] d'une toute autre manière. L'idée est la suivante : puisque l'approche OBLOG ne convient pas pour animer une spécification ALBERT, nous devons construire nous-même un animateur de cette spécification. Cette deuxième approche consiste à construire un modèle générique d'une spécification ALBERT qui, une fois instancié à une spécification particulière, devient un animateur de celle-ci. En fait, il s'agit de construire le système d'information ou le *repository* d'une spécification ALBERT, en y intégrant une interaction avec l'utilisateur pour lever les indéterminismes.

L'utilisation d'OBLOG pour spécifier cet animateur se justifie au moins pour trois raisons :
- le langage OBLOG se prête bien à la modélisation d'un système d'information, en prenant en compte non seulement ses aspects structurels mais également ses aspects comportementaux ;
- l'éditeur de dialogue permet de concevoir facilement l'interface utilisateur de l'animateur et le langage permet de gérer cette interface à l'aide des objets de type boîte de dialogue ;
- le générateur de code permet d'obtenir directement une version exécutable de l'animateur.

Concrètement, le processus de transformation qui permet d'obtenir un animateur en OBLOG à partir d'une spécification en ALBERT est décomposé en deux phases :
- une phase de *préparation* du texte ALBERT ;
- une phase d'*instanciation*, qui consiste à instancier la spécification OBLOG du modèle générique en fonction de la spécification ALBERT ainsi préparée.

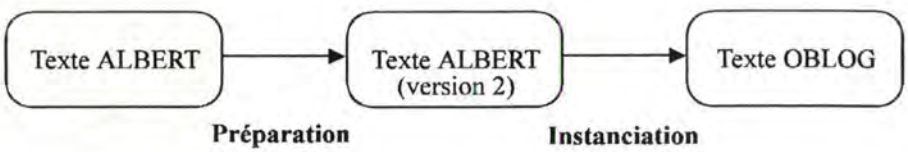


Figure 23

La section 8.2 présente les fonctionnalités attendues de l'animateur, en fonction desquelles nous intégrerons l'interaction avec l'utilisateur dans le modèle générique. La section 8.3 décrit

[1] Rappelons que nous utilisons la version OBLOGlight.

la phase de préparation du texte ALBERT. Les sections 8.4 à 8.14 décrivent le modèle générique. Finalement, la section 8.15 approfondit les problèmes liés au temps et la section 8.16 évalue la méthode de prototypage.

On trouvera, en annexe (section A.9), une illustration de la méthode sur les spécifications des besoins associées au véhicule de l'atelier.

8.2. FONCTIONNALITÉS DE L'ANIMATEUR

Ces fonctionnalités peuvent être considérées à deux niveaux :
- au niveau société, l'utilisateur contrôle le déroulement du temps. Ceci comporte non seulement la possibilité de faire avancer le temps mais également de "faire marche arrière" *(backtracking)* ;
- au niveau agent, il contrôle la vie de l'agent en initialisant l'état et en choisissant, parmi des suggestions, les occurrences d'action de chaque changement.

Pour mieux se représenter ces diverses fonctionnalités, nous suggérons, figure 24, une interface utilisateur possible (Dans les boîtes de dialogue, les exemples concernent les agents `Véhicule` et `Gestionnaire`.)

a) Niveau société

Les fonctionnalités du niveau société sont regroupées dans une boîte de dialogue appelée *Clock*, qui affiche le temps courant (soit t) :
- *Tick :* avancer au temps $t+1$;
- *Reset :* annuler toutes les modifications intervenues au temps t sur l'état et sur le changement en construction de chaque agent ;
- *Reverse :* annuler toutes les modifications intervenues au temps t et revenir au temps $t-1$.
- *Kill :* arrêter définitivement le temps, c'est-à-dire faire mourir tous les agents ;

Lorsque l'utilisateur sélectionne la fonction *Tick* ou *Kill*, le coordinateur (*Clock*) en évalue la faisabilité. La fonction *Tick* ne peut être réalisée que lorsque tous les agents respectent leurs contraintes sur l'état et leurs obligations relatives au temps t. La fonction *Kill* exige en plus que les agents n'aient pas de contraintes sur l'état ou d'engagements à respecter dans le futur. Un message d'erreur apparaît si nécessaire, dans la boîte de dialogue intitulée *Error*.

b) Niveau agent

A chaque agent est associée une boîte de dialogue qui joue le rôle de panneau de contrôle de l'agent *(Agent Control Panel)*. Etant donné un temps t quelconque (pouvant être différent du temps courant), cette boîte présente, sur trois colonnes :

- l'état de l'agent, c'est-à-dire les valeurs des différents attributs, au temps *t* ;
- une liste d'actions *candidates* à figurer dans le changement en construction au temps *t*, ainsi que la classe des arguments valides pour ces actions ;
- le changement en construction, c'est-à-dire la liste des occurrences d'action déjà présentes dans le changement au temps *t*, ainsi que leurs arguments.

Cette énumération comprend les attributs importés et les actions externes.

Une action est dite *candidate* au changement si elle n'est pas interdite par une clause de prévention et qu'elle n'entre pas en concurrence avec les occurrences déjà présentes dans le changement en construction (c'est-à-dire que ses effets ne portent pas sur les mêmes attributs). Comme ces deux conditions peuvent être fonction des arguments de l'action, une action peut être candidate avec tels arguments et ne pas l'être avec d'autres. Or, il est impossible, pratiquement, de présenter la liste de tous les arguments valides (même si, en réalité, cette liste est finie puisqu'un ordinateur ne propose que des types de donnée finis). C'est pourquoi on s'efforcera de présenter un état résumé des arguments valides.

Un message d'erreur apparaît dans la boîte *Error* lorsque une action est à la fois interdite et obligatoire. (Dans ce cas, l'utilisateur n'a plus d'autre choix que la fonction *Reset* ou *Reverse* du coordinateur *Clock*.)

Enfin, le panneau de contrôle de l'agent propose les fonctionnalités suivantes :
- *Back :* consulter l'état et le changement au temps *t-1* ;
- *Next :* consulter l'état et le changement au temps *t+1* (fonction inaccessible au temps courant) ;
- *Modify :* modifier les valeurs initiales de l'état à l'aide de la boîte de dialogue intitulée *State Initialization* (fonction accessible uniquement lorsque le temps courant vaut *t0*) ;
- *Add :* ajouter une occurrence d'action dans le changement (fonction accessible uniquement au temps courant). La boîte de dialogue *Action Instanciation* est prévue pour donner des valeurs aux arguments éventuels de l'action.

Remarque : la fonction *Back* ne doit pas être confondue avec la fonction *Reverse* ; *Back* permet de consulter le passé, mais tout en restant au temps courant affiché par le coordinateur.

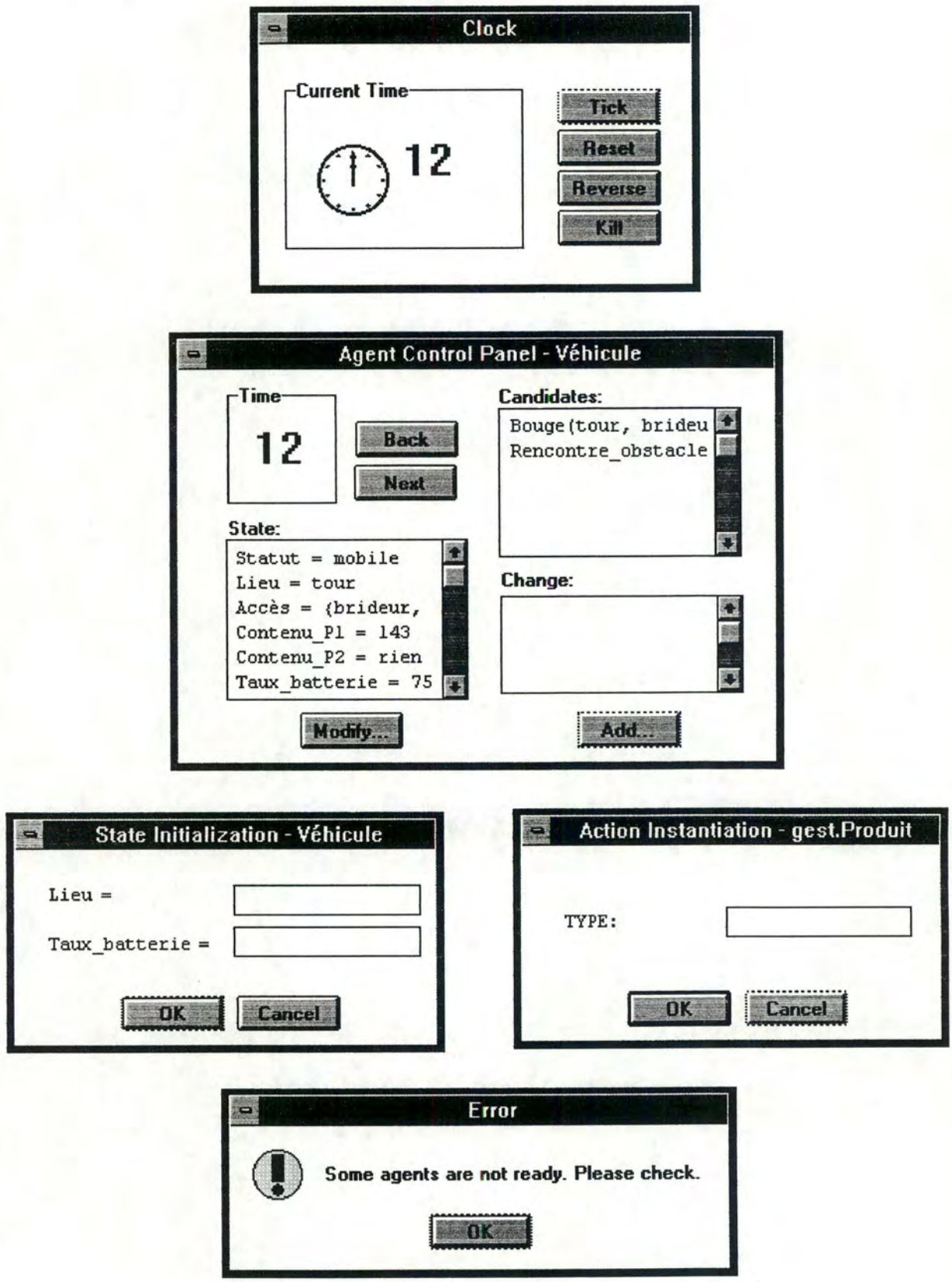


Figure 24 - Proposition d'interface utilisateur

8.3. PHASE DE PRÉPARATION

La phase de préparation comporte elle-même deux étapes : une étape d'*opérationalisation* et une étape de "*formatage*".

a) Etape d'opérationalisation

Le fait de récupérer des fragments de la spécification ALBERT, et en particulier des formules, a pour conséquence que l'on reste partiellement dépendant de l'expressivité d'OBLOG. Une phase d'opérationalisation, comme celle utilisée au chapitre précédent mais beaucoup plus limitée, reste nécessaire. Rappelons qu'elle consiste à appliquer des transformations sémantiques au texte ALBERT afin de le rendre opérationnel d'un point de vue OBLOG.

Dans un premier temps, nous nous simplifierons la tâche en appliquant les règles de transformations vues au chapitre précédent concernant :
- le remplacement des clauses de publicité en clauses de perception équivalentes (cf point 7.6.1) ;
- les formules LPO (logique du premier ordre, cf point 7.7.1) ;
- les formules LTP et LTF (logiques temporelles dans le passé et dans le futur, cf points 7.7.2 et 7.7.3) ;
- les engagements (cf point 7.9.2).

Nous verrons dans un deuxième temps (voir section 8.15) que l'approche de transformation que nous avons adoptée permet de traiter directement les formules temporelles et les engagements.

b) Etape de "formatage"

L'étape de formatage consiste à reformuler le texte ALBERT (à nouveau par des transformations sémantiques) obtenu après opérationalisation afin qu'il respecte le format type présenté à la figure 25. (On peut montrer que toute spécification opérationalisée peut être exprimée sous ce format.)

La figure 25 introduit la spécification type d'un agent ayant les caractéristiques suivantes :
- *na* attributs internes, dont *ni* sont initialisés (sans nuire à la généralité, supposons que ces *ni* attributs initialisés sont les *ni* premiers) ;
- *nb* attributs importés ;
- *nc* attributs dérivés ;
- *ns* actions internes ;
- *nr* actions externes.

Nous définissons une clause globalisée comme la composition de différentes clauses de la spécification originale, à l'aide des connecteurs `and` ou `or` selon les cas (`and` pour les contraintes sur l'état, `or` pour les clauses "F" et "O"). La normalisation d'un clause a été définie à la section 7.3.

Les contraintes sur l'état sont globalisées en deux groupes : les contraintes relatives au temps courant et les contraintes relatives à la mort de l'agent. Les contraintes relatives à la mort correspondent à toutes les conditions préventives associées à l'action `Die`, créées par l'opérationalisation des formules LTF et des engagements. (Mis à part l'utilisation de ses préventions, l'action `Die` n'est plus reprise dans le modèle générique, pas plus d'ailleurs que l'action `Live` ou `Fail`).

L'action *S1* a *ne* effets sur *ne* attributs différents, une clause globalisée de prévention et *no* clauses d'obligation. Ces clauses sont toutes normalisées.

A chaque attribut importé sont associées des clauses de perception. De manière similaire à l'action *S1*, l'action externe *u_R1.R1* a *ne* effets, une clause globalisée de prévention et une clause globalisée d'obligation relatives à sa perception. Ces clauses relatives à *u_R1.R1* sont normalisées. (Pour alléger les notations, nous conviendrons que les notations concernant une action sont "locales" à celle-ci. Par exemple, *p_F* représentera, suivant les cas, la prévention associée à *S1* ou à *R1*.)

Enfin, à chaque action exportée sont associées des clauses normalisées de publicité.

U1

Declarations

Attributes

A1 : T_A1
...
Ana : T_Ana

B1 : T_B1 <- U_B1
...
Bnb : T_Bnb <- U_Bnb

Derived Attributes

C1 : T_C1 = e_C1
...
Cnc : T_Cnc = e_Cnc

Actions

S1 (T_x1, ..., T_xnx) -> {U_V1, ..., U_Vnv}
...
Sns (T_x1, ..., T_xnx) -> {U_V1, ..., U_Vnv}

R1 (T_x1, ..., T_xnx) <- U_R1
...
Rnr (T_x1, ..., T_xnx) <- U_Rnr

Constraints

State Behaviour

p_C_cur

Init

A1 = c_I1
...
Ani = c_Ini

Effects of Actions

S1 (x1, ..., xnx) :
    (p_E1) => A_E1 = e_E1
    ...
    (p_Ene) => A_Ene = e_Ene
...

Responsibility

F(*S1 (x1, ..., xnx) / p_F*)
O(*S1 (x1, ..., xnx)* with *p_O1_arg / p_O1*)
...
O(*S1 (x1, ..., xnx)* with *p_Ono_arg / p_Ono*)
...
F(Die / *p_C_fut*)

Perception

X(*u_B1.B1 / p_X_B1*)
...
X(*u_Bnb.Bnb / p_X_Bnb*)

F(*u_R1.R1 (x1, ..., xnx) / p_F*)
O(*u_R1.R1 (x1, ..., xnx) / p_O*)
...

Publicity

X(*S1 (x1, ..., xnx) .{u_V1} / p_X_V1*)
...
X(*S1 (x1, ..., xnx) .{u_Vnv} / p_X_Vnv*)

Figure 25 - Spécification ALBERT type

8.4. ARCHITECTURE DU MODÈLE GÉNÉRIQUE

L'architecture proposée se conforme à la philosophie "orientée-agent". Les objets du niveau société sont regroupés au sein d'une unité `MAIN` (par analogie avec la fonction *main* en C), tandis que chaque agent fait l'objet d'une unité séparée. On supposera enfin que l'objet `error_dbx` fait partie d'une unité-librairie prédéfinie `LIB`. Le diagramme de projet correspondant à cette architecture est présenté à la figure 26.

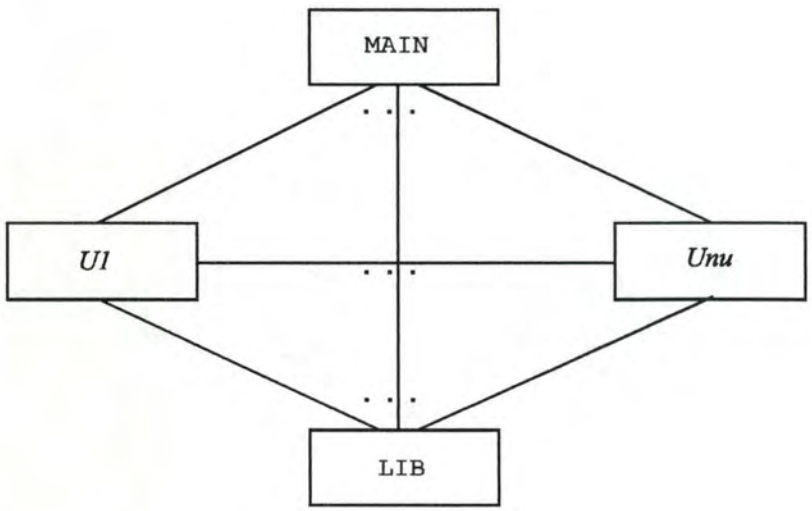


Figure 26 - Diagramme de projet

a) Niveau société

La figure 27 représente le diagramme de communauté de l'unité `MAIN`. L'objet coordinateur est l'objet `clock`. L'objet `clock_dbx` représente la boîte de dialogue associée. La classe d'objets `AGENT` est la généralisation de tous les objets *u1,...,unu* des différentes unités *U1,...,Unu*.

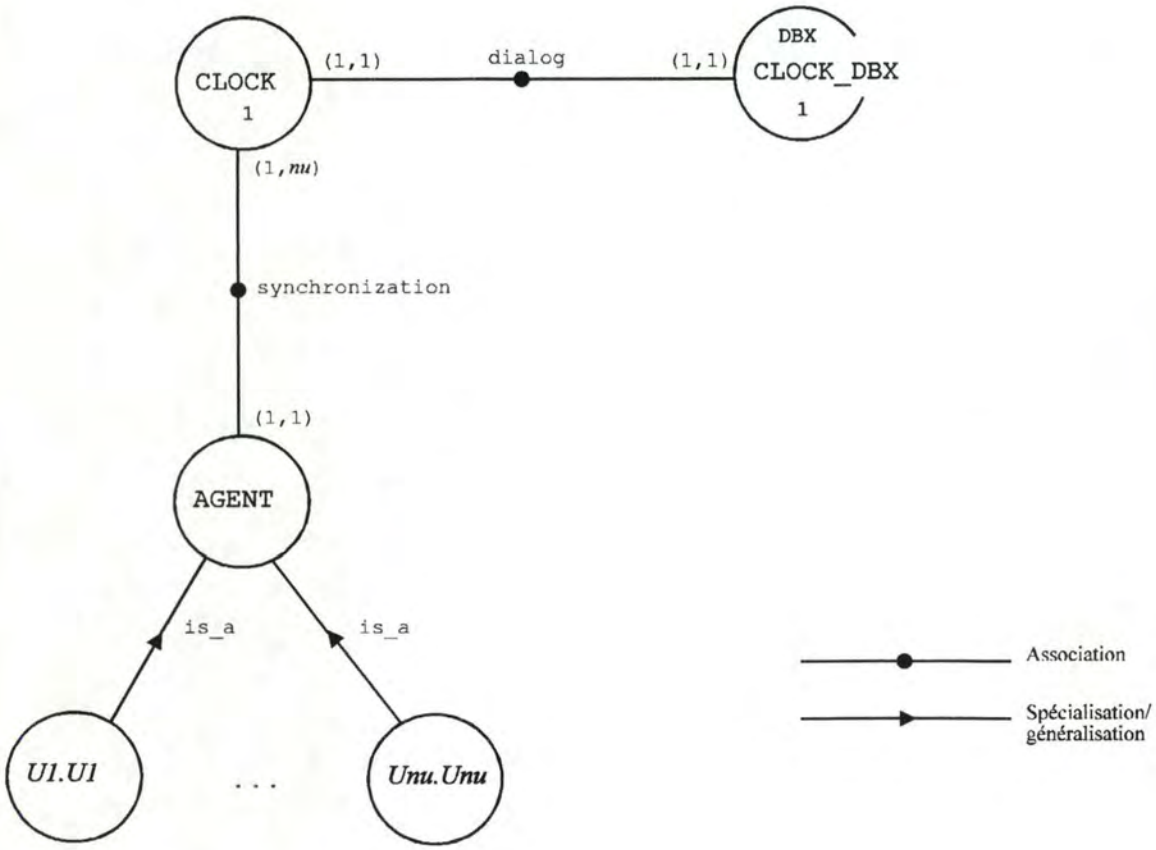


Figure 27 - L'unité MAIN

b) Niveau agent

L'agent *U1*, par exemple, est entièrement représenté par la communauté d'objets de l'unité *U1*, représentée à la figure 28.

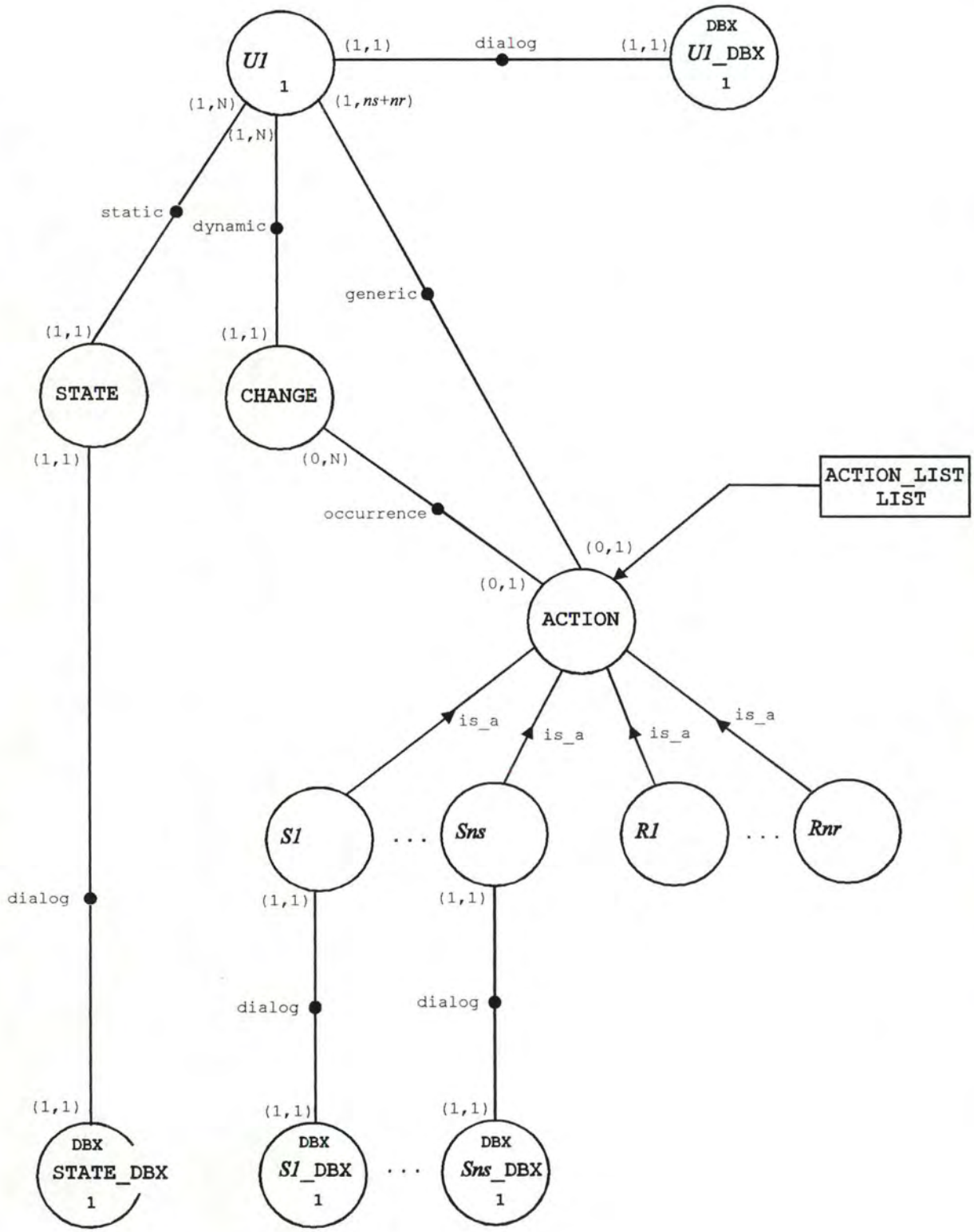


Figure 28 - L'unité U1

L'objet central, qui porte également le nom de l'agent (*u1*), est en quelque sorte un *manager* qui contrôle et dirige le comportement de l'agent. C'est le seul interlocuteur visible du point de vue du coordinateur *Clock*. Cet objet *u1* est entouré de classes d'objets satellites qui prennent chacun en charge un aspect de l'agent :
- `STATE` : les états de l'agent jusque et y compris le temps courant ;
- `CHANGE` : les changements correspondants à chaque état ;
- *S1*,...,*Sns* : les *ns* actions internes de l'agent ;
- *R1*,...,*Rnr* : les *nr* actions externes de l'agent.

Plus précisément, un objet *s1* ou *r1* (c'est-à-dire une instance de la classe d'objets *S1* ou *R1*) représente, à sa naissance, la classe des occurrences possibles de l'action *S1* (*R1*) à un certain moment de la vie de l'agent : à chaque tuple d'arguments possible correspond une occurrence possible. Nous dirons que l'objet *s1* (*r1*) représente l'*action générique S1* (*R1*). Ensuite, lorsqu'une occurrence particulière de l'action *S1* (*R1*) est choisie pour figurer dans un changement, l'objet *s1* (*r1*) représente cette occurrence.

Toujours à propos des actions, la classe d'objets `ACTION` est la généralisation de toutes les classes *S1*,...,*Sns*,*R1*,...,*Rnr*. Une instance de cette classe ne peut pas participer en même temps à une association `generic` et à une association `occurrence`. Le type de données `ACTION_LIST` représente, comme son nom l'indique, une liste d'instances de `la classe ACTION`.

Enfin, il reste à parler du rôle de trois objets de type boîte de dialogue (à mettre en relation avec leur *layout* qui a été présenté à la figure 24) :
- *u1*`_dbx` : panneau de contrôle de l'agent *U1* ;
- `state_dbx` : boîte d'initialisation de l'état initial de l'agent *U1* ;
- *s1*`_dbx` : boîte d'instanciation des arguments de l'action *S1*.

Les sections suivantes décrivent en détail la spécification OBLOG de tous ces objets ou classes d'objets, en commençant par l'unité `MAIN` (sections 8.5 et 8.6) et en poursuivant par l'unité *U1* (sections 8.7 à 8.14). Les conventions utilisées dans ces spécifications sont définies ci-après.

• Comme certains fragments de l'architecture doivent être instanciés en fonction de la spécification ALBERT, il convient de les distinguer :

`state.reset`	doit apparaître tel quel dans le texte OBLOG
p_O1_arg	doit être remplacé par l'élément de la spécification ALBERT correspondant (cf figure 25).

• Par souci d'économie ou pour faciliter la lecture, il arrive souvent qu'on "aménage" les diagrammes OBLOG. Ici, nous présenterons, pour chaque classe d'objets, deux diagrammes :
- un diagramme de déclaration des attributs (aspects statiques de l'objet ; la déclaration des actions peut aisément être reconstituée à partir du second diagramme) ;
- un diagramme qui superpose les diagrammes de comportement, de mise à jour et d'interaction (aspects dynamiques de l'objet).

8.5. L'OBJET CLOCK_DBX

Cet objet est responsable de la représentation à l'écran de l'objet `clock`. Il s'agit d'une boîte de dialogue qui affiche le temps courant, reçoit et transmet les choix de l'utilisateur à l'objet `clock` (fonctions *Tick*, *Reset*, *Reverse* et *Kill*).

L'action `refresh` met à jour l'affichage du temps courant.

8.6. L'OBJET CLOCK

Le comportement de l'objet `clock` se borne à créer tous les agents (action `create`), à ouvrir la boîte de dialogue `clock_dbx` (action `open`), puis à attendre les stimuli de l'utilisateur, via la boîte de dialogue.

Lorsque une des actions `tick`, `kill`, `reset` ou `reverse` a été sélectionnée, elle appelle à son tour l'action correspondante auprès de tous les objets. Il s'agit d'un appel synchrone, ce qui signifie que les vies de tous les objets évoluent en parallèle. Dans le cas des fonctions *Tick* et *Kill*, l'appel ne réussit (actions `doTick` ou `doKill`) que si tous les agents sont, respectivement, prêts ou prêts à mourir (voir les attributs `ready` et `readyToDie` dans l'objet *ui*, plus loin dans ce chapitre). Dans le cas contraire, l'action `fail` commande l'ouverture de la boîte de messages d'erreur (l'objet `error_dbx` rappelle l'objet `clock` via l'action `ok`).

Le seul attribut de l'objet est le temps courant, `curTime`. Après une mise à jour de cet attribut (par les actions `doTick` ou `reverse`), l'action `refresh` commande le rafraîchissement de la boîte de dialogue `clock_dbx`.

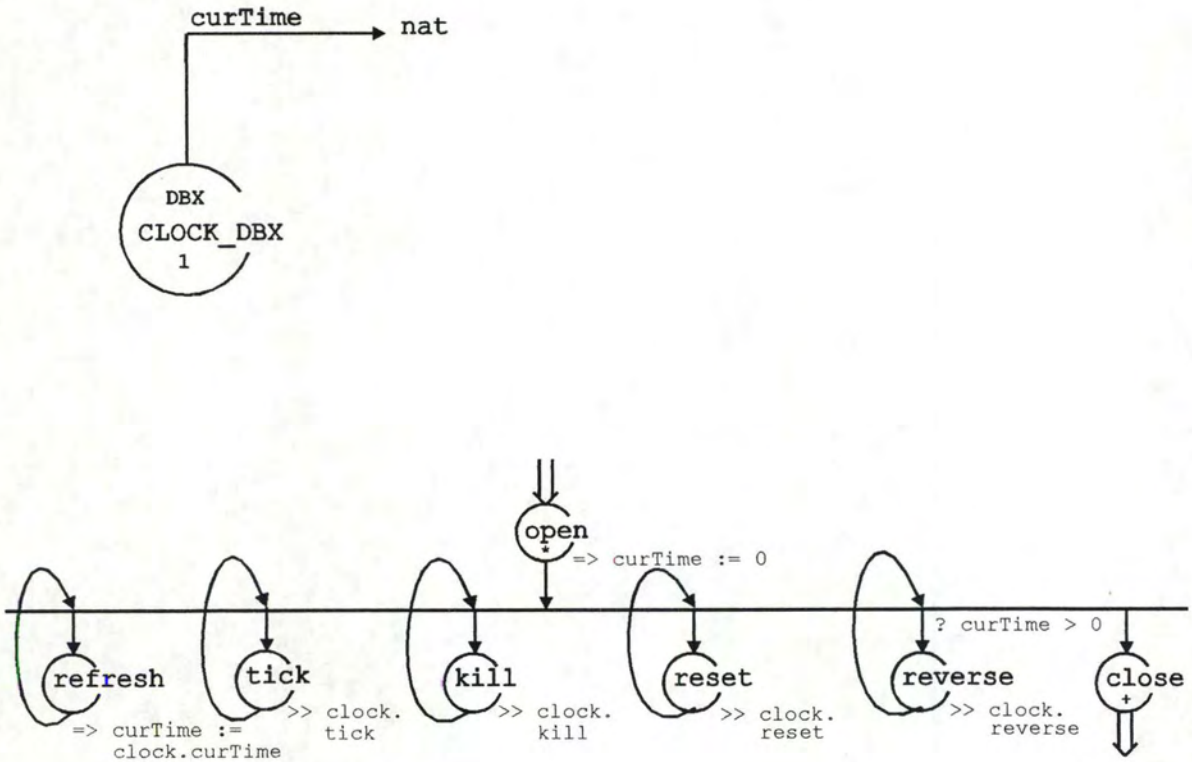


Figure 29 - L'objet clock_dbx

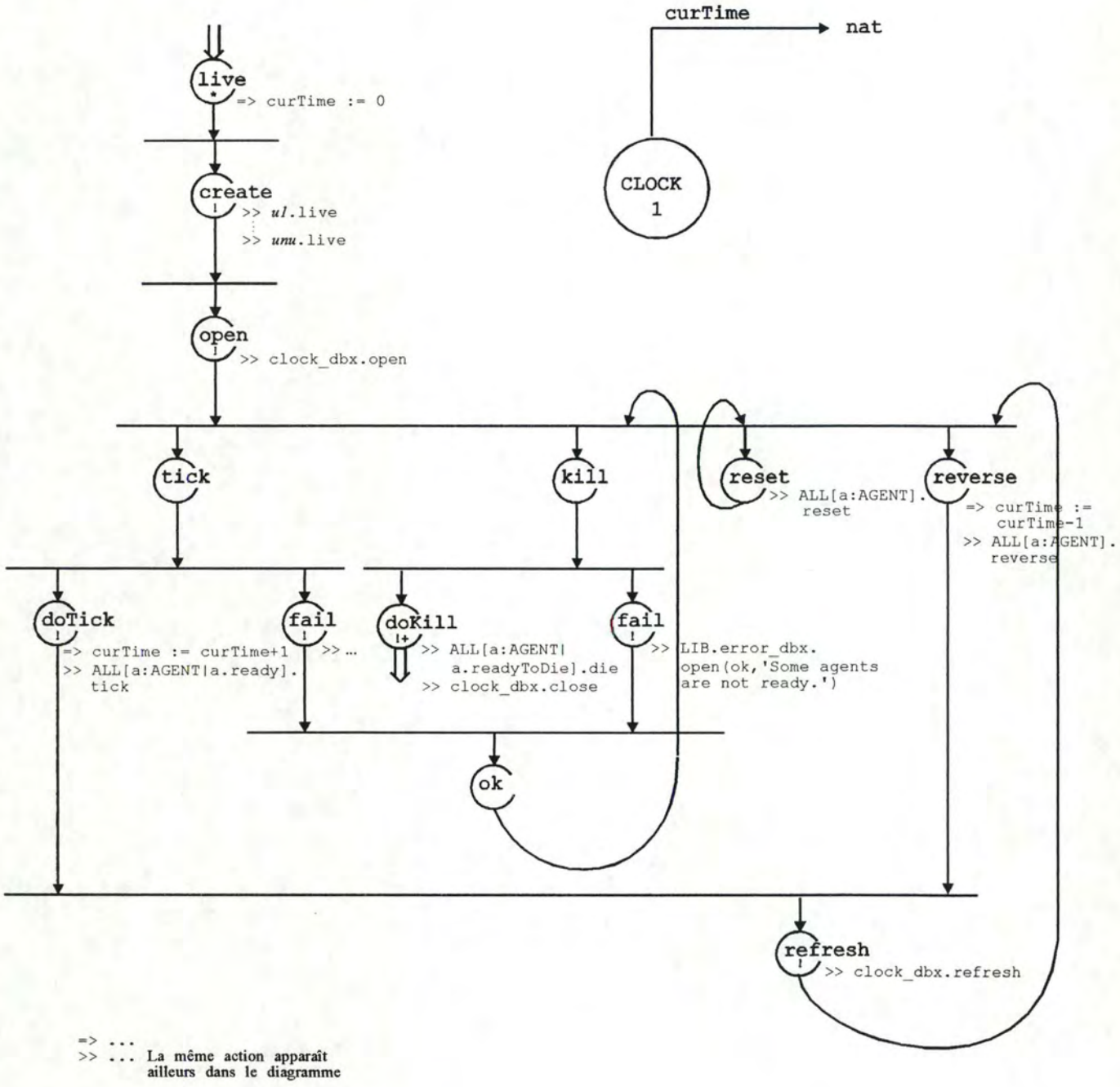


Figure 30 - L'objet clock

8.7. L'OBJET U1_DBX

L'objet `u1_dbx` représente le panneau de contrôle qui permet à l'utilisateur de consulter et d'influencer la vie de l'agent *U1*. Il s'agit d'une boîte de dialogue qui affiche, pour un temps donné (attribut `oneTime`), l'état et le changement de l'agent (`oneState` et `oneChange`), ainsi qu'une liste d'actions encore candidates au changement (`allCand`) s'il s'agit du temps courant, c'est-à-dire le temps de l'objet `clock`. Cette boîte reçoit et transmet les choix de l'utilisateur à l'objet *u1* (fonctions *Back*, *Next*, *Modify* et *Add*).

L'action `refresh` met à jour l'affichage.

Remarque : il n'a pas été tenu compte du type de donnée à afficher. Il est certain que toutes les données ne peuvent être affichées telles quelles sur un écran et qu'il faudrait, par exemple, une fonction qui donne le nom d'une action (en caractères imprimables) à partir de la référence de l'objet correspondant. (Cette remarque vaut pour tous les objets de type boîte de dialogue *(kind-of DBX)*.)

8.8. L'OBJET U1

Remarque : la classe d'objets *U1* ne possède qu'une seule instance, *u1*, qui représente l'unique agent *U1*. Il est aisé d'adapter les spécifications OBLOG si l'on veut représenter une classe d'agents *U1*.

8.8.1. ATTRIBUTS

L'attribut `curTime` est une simple copie de l'attribut de l'objet `clock`, mise à la disposition de toute l'unité *U1*. Les attributs `curState` et `curChange` contiennent les références de l'état et du changement courants. L'attribut `prevState` calcule la référence de l'état précédent ; `failed` indique si l'agent est dans une vie valide ou non ; `inited` indique si l'état initial est entièrement initialisé, c'est-à-dire s'il existe, dans la spécification ALBERT, une clause d'initialisation pour chaque attribut.

Les attributs suivants font référence aux actions génériques :
- `cand` indique s'il existe une action candidate, c'est-à-dire qui ne soit pas interdite (par une clause "F") ou incompatible avec les occurrences d'action du changement en construction (car ayant un effet sur les mêmes attributs) ;
- `oblig` indique s'il existe une action obligatoire (par une clause "O") ;
- `contrad` indique s'il existe une action contradictoire, c'est-à-dire à la fois obligatoire et interdite ou à la fois obligatoire et incompatible.

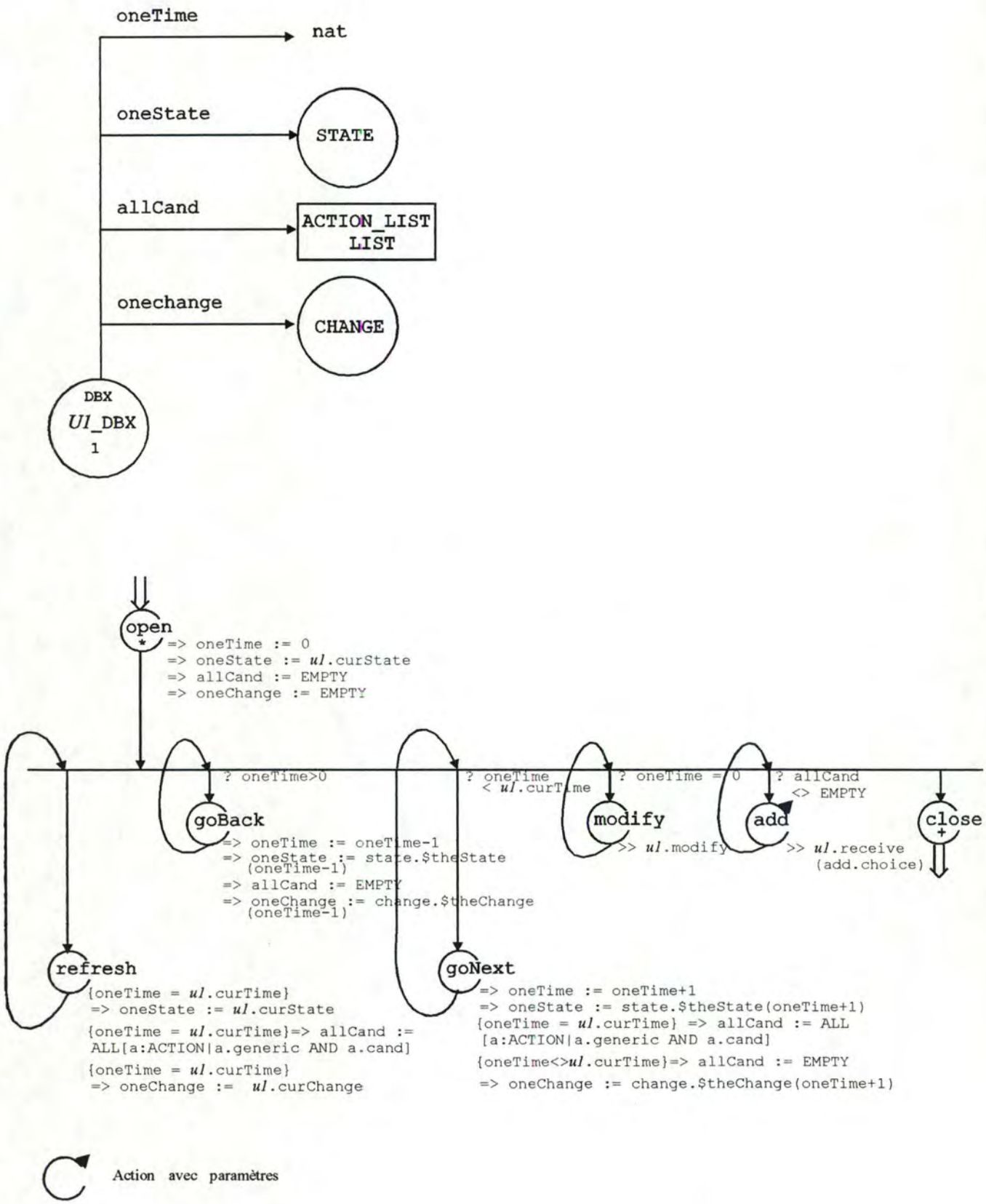


Figure 31 - L'objet ul_dbx

Enfin `ready` et `readyToDie` évaluent les conditions d'évolution de l'agent :
- `ready` indique si l'agent est prêt à passer au temps suivant, ce qui est vrai, au temps t0, lorsque l'état est initialisé et respecte les contraintes et, aux autres temps, lorsque les obligations et les contraintes sur l'état relatives au temps courant sont respectées.
- `readyToDie` indique si l'agent est prêt à mourir, ce qui est vrai quand il est prêt à passer au temps suivant et qu'il n'a pas de contraintes à respecter dans le futur.

8.8.2. ACTIONS

Le comportement de l'objet *u1* comporte trois phases : *big-bang*, temps initial et temps ultérieurs.

a) Big-bang

La première phase est une phase de création : créations de l'état initial et du changement initial, création de toutes les actions et ouverture de la boîte de dialogue associée à l'agent (actions `live`, `createAll` et `open`).

b) Temps initial

La deuxième phase est consacrée au temps initial et commence, dans le diagramme de comportement, par la situation `t0`.

S'il existe des attributs non-initialisés, l'action `init` a lieu sur initiative de l'agent lui-même. Cette action fait appel à l'état pour qu'il ouvre la boîte de dialogue d'initialisation de l'état. L'utilisateur est alors invité à introduire les valeurs. Selon la manière dont le dialogue se termine, c'est l'action `ok` ou `cancel` qui est rappelée.

L'action `ok` est suivie de l'action `refreshAll` qui rafraîchit la boîte de dialogue de tous les agents pour mettre à jour l'affichage des valeurs des attributs.

L'action `cancel` indique que l'utilisateur ne réussit pas à affecter des valeurs initiales valides aux attributs (voir l'objet `state_dbx` plus loin dans ce chapitre). Etant donné que l'état doit absolument être initialisé, l'action `fail` fait ouvrir la boîte de messages d'erreur (l'action suivante, `ok`, est rappelée lorsque l'utilisateur a pris connaissance du message en cliquant sur le bouton *OK*).

Par la suite (mais toujours au temps *t0*), l'utilisateur peut encore modifier les valeurs initiales des attributs en cliquant sur le bouton *Modify* dans la boîte de dialogue de l'agent, ce qui déclenche l'action `modify` de *u1*. Le scénario est le même que pour l'action `init` si ce n'est que l'action `cancel` ne donne pas lieu à un échec. En effet, si l'utilisateur ne réussit à affecter des valeurs initiales valides aux attributs (ou tout simplement s'il annule le choix de la fonction *Modify*), l'état initial n'est pas modifié et l'action `continue` a lieu.

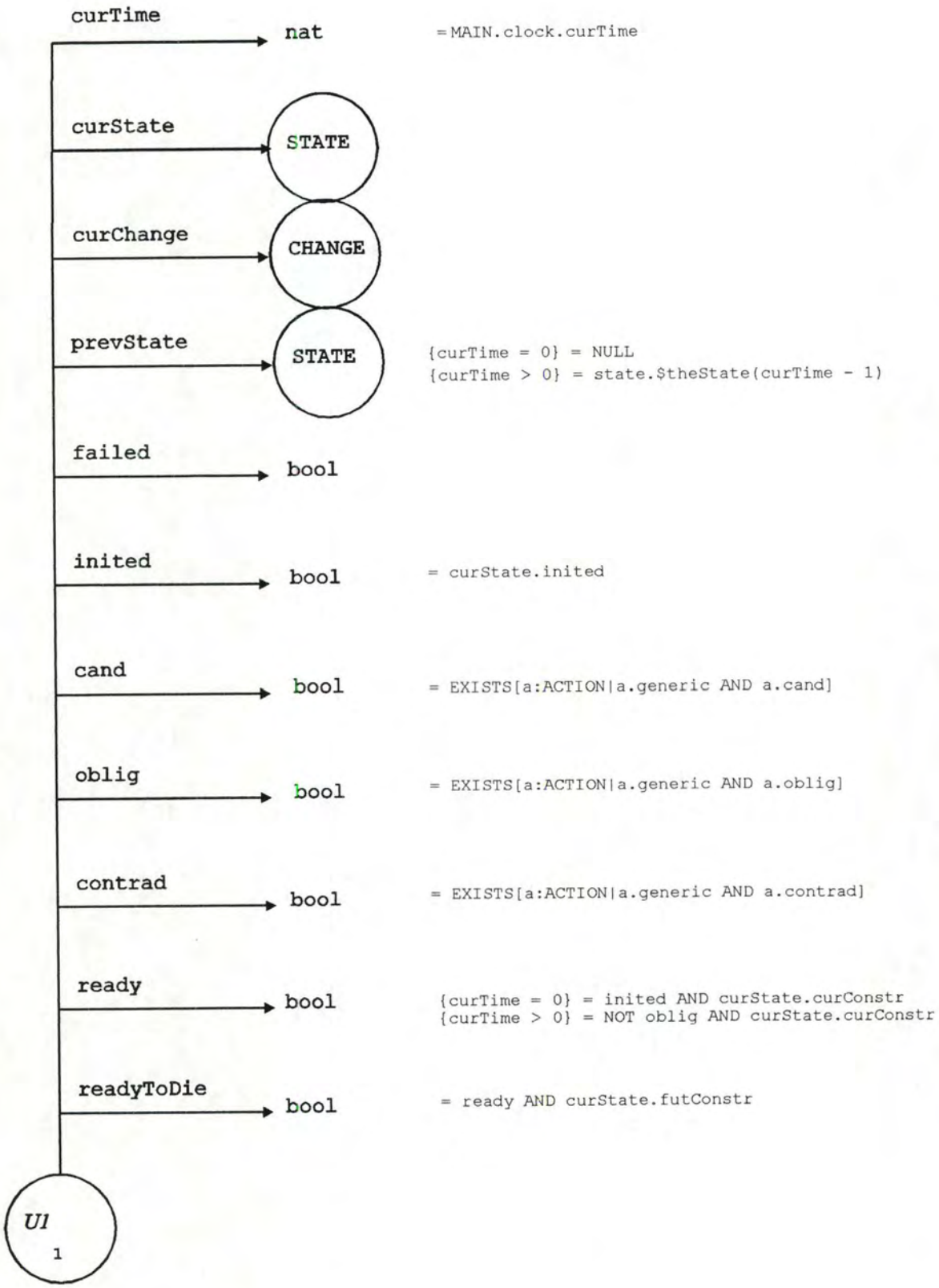


Figure 32a - L'objet u1

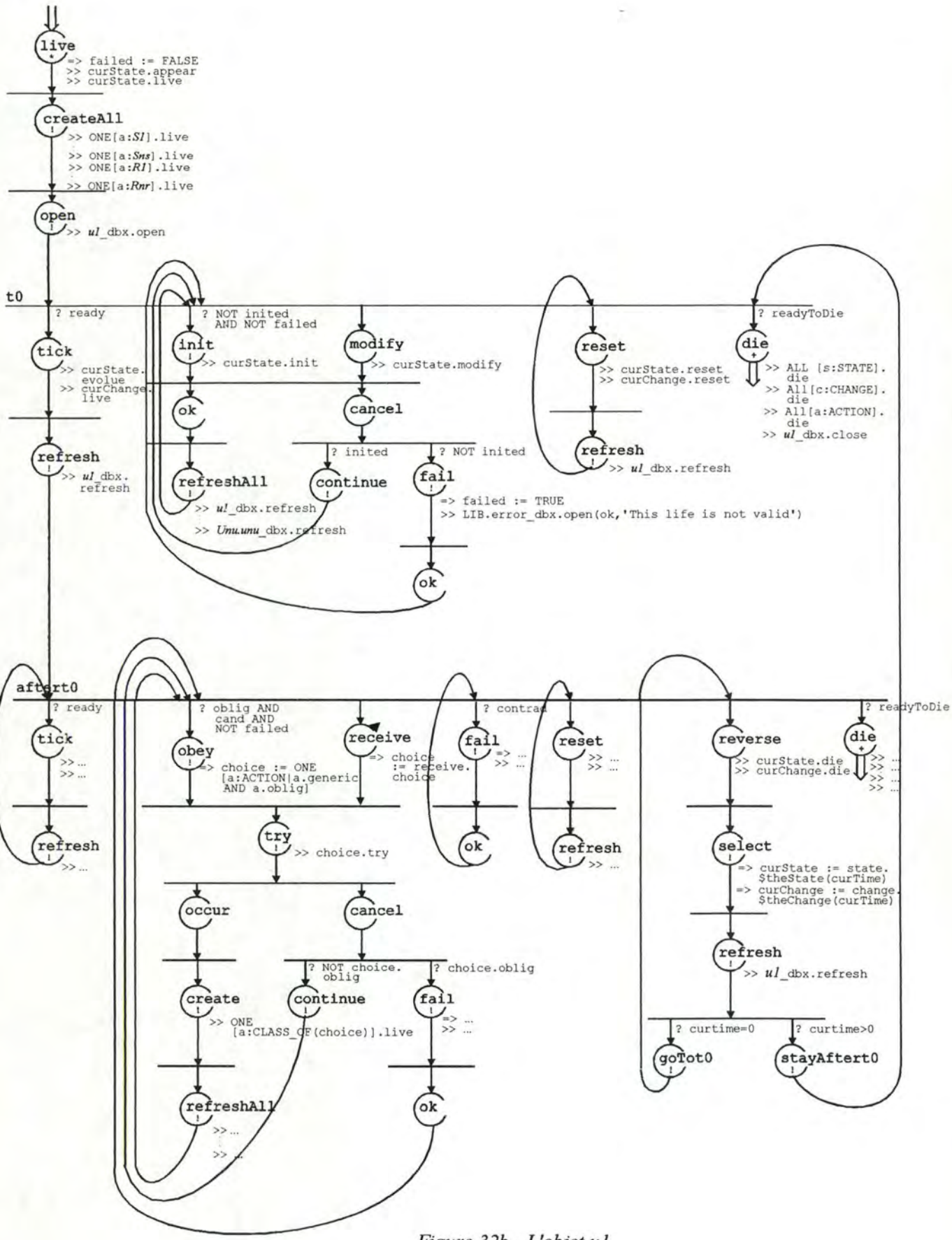


Figure 32b - L'objet ul

Les actions `tick`, `reset` et `die` sont déclenchées par l'objet `clock`. L'action `tick` crée un nouvel état et un nouveau changement, l'action `reset` "remet à zéro" l'état et le changement courants, l'action `die` tue tous les objets de l'unité. Enfin, l'action `refresh` "rafraîchit" la boîte de dialogue de l'agent *U1* après une action `tick` ou `reset`.

c) Temps ultérieurs

La troisième phase est consacrée aux temps ultérieurs à *t0* et commence, dans le diagramme de comportement, par la situation `aftert0`. Imaginons que l'agent vient de "passer" au temps *t1* : suite à l'action `tick`, l'état courant est une copie de l'état initial et le changement est vide. L'agent doit "évaluer la situation" pour connaître la conduite à suivre.

S'il existe une action générique obligatoire, l'agent peut la prendre en charge d'initiative, sans qu'une confirmation de l'utilisateur ne soit nécessaire. L'action `obey` choisit une action obligatoire parmi toutes, disons *S1*. l'action `try` a pour effet d'"essayer" de produire une occurrence de *S1*. Encore faut-il, pour que *S1* se produise effectivement, que l'utilisateur introduise des valeurs d'arguments qui soient valides.

L'action `occur` exprime le fait que *S1* a eu lieu avec succès, ce qui veut dire que, d'action générique, elle est devenue occurrence. L'agent crée donc une nouvelle instance de la classe d'objets *S1* pour préparer la prochaine occurrence de *S1* (action `create`) et demande à mettre à jour l'affichage des boîtes de dialogue de tous les agents afin de prendre en compte cette nouvelle occurrence (action `refreshAll`).

L'action `cancel`, par contre, indique que l'utilisateur ne réussit pas à affecter des valeurs valides aux arguments de *S1* (voir l'objet `s1_dbx` plus loin dans ce chapitre). Etant donné que *S1* est obligatoire, l'action `fail` fait ouvrir la boîte de messages d'erreur (l'action suivante, `ok`, est rappelée lorsque l'utilisateur a pris connaissance du message en cliquant sur le bouton *OK*).

Lorsque plus aucune action générique n'est obligatoire, l'utilisateur peut en sélectionner une explicitement, disons encore *S1*, parmi la liste des candidats affichée dans la boîte de dialogue (fonction *Add*), ce qui se traduit par une action `receive`. Le scénario est le même que pour les actions obligatoires si ce n'est que l'action `cancel` ne donne pas lieu à un échec. En effet, si l'utilisateur n'arrive pas à affecter des valeurs valides aux arguments de *S1* (ou tout simplement s'il annule le choix de la fonction *Add*), *S1* ne devient pas une occurrence du changement en construction et l'action `continue` a lieu.

Il existe cependant une deuxième situation de blocage qui provoque une action `fail`, outre le cas d'une action obligatoire pour laquelle il n'existe pas de valeurs d'arguments valides : celle où une action générique est contradictoire. A noter que, dans ces deux cas d'échec, l'utilisateur n'a plus d'autre choix que de sélectionner, au niveau de la boîte `clock_dbx`, la fonction *Reset* (inutile si le changement est vide) ou *Reverse*.

On retrouve à nouveau les actions `tick`, `reset` et `die`, déclenchées par l'objet `clock` et auxquelles il faut ajouter, pour les temps ultérieurs à *t0*, l'action `reverse`. L'action `reverse` tue l'état et le changement courants ; l'action `select` rend courants l'état et le changement précédents. Suivant que le nouveau temps courant est égal ou supérieur à 0, l'objet retourne

dans la situation `t0` ou `aftert0` (actions `goTot0` ou `stayAftert0`). Enfin, l'action `refresh` "rafraîchit" la boîte de dialogue de l'agent *UI* après une action `tick`, `reset` ou `select`.

8.9. L'OBJET STATE_DBX

La boîte de dialogue `state_dbx` a pour but de saisir des valeurs valides pour initialiser les *na-ni* attributs qui ne possèdent pas de clause **Init** dans le texte ALBERT. Ces valeurs sont transmises à l'état initial.

Par valeurs valides (voir l'attribut `valid`), on entend des valeurs non-nulles qui vérifient les contraintes sur l'état :
- qui s'appliquent au temps initial, c'est-à-dire les invariants, une partie des contraintes de LTP et de LTF ;
- et qui ne concernent que des attributs internes et les attributs dérivés à partir d'autres attributs internes uniquement (numérotés ici, sans nuire à la généralité, de 1 à *k*), ce que l'on pourrait qualifier de contraintes "intra-agent". En effet, il est difficile pour l'utilisateur de respecter les contraintes faisant référence à des attributs importés tant que les états de tous les agents ne sont pas encore initialisés. Au besoin, la fonction *Modify* permettra de modifier par la suite les valeurs en fonction des contraintes "inter-agent".

Ces contraintes sont globalisées par *p_C_init* (aisé à déduire de la spécification ALBERT, à partir de *p_C_cur*).

8.10. LA CLASSE D'OBJETS STATE

8.10.1. ATTRIBUTS

L'attribut `$theState(time)` est une clé (appartenant à la métaclasse), qui renvoie la référence d'un état à partir de sa valeur de temps. L'attribut `time` enregistre le temps de l'état et constitue un identifiant externe de la classe.

L'attribut `curConstr` indique si les attributs respectent les contraintes relatives à l'état courant ; `futConstr` indique si les attributs respectent les contraintes relatives à la mort de l'agent ; `inited` indique si tous les attributs internes sont initialisés.

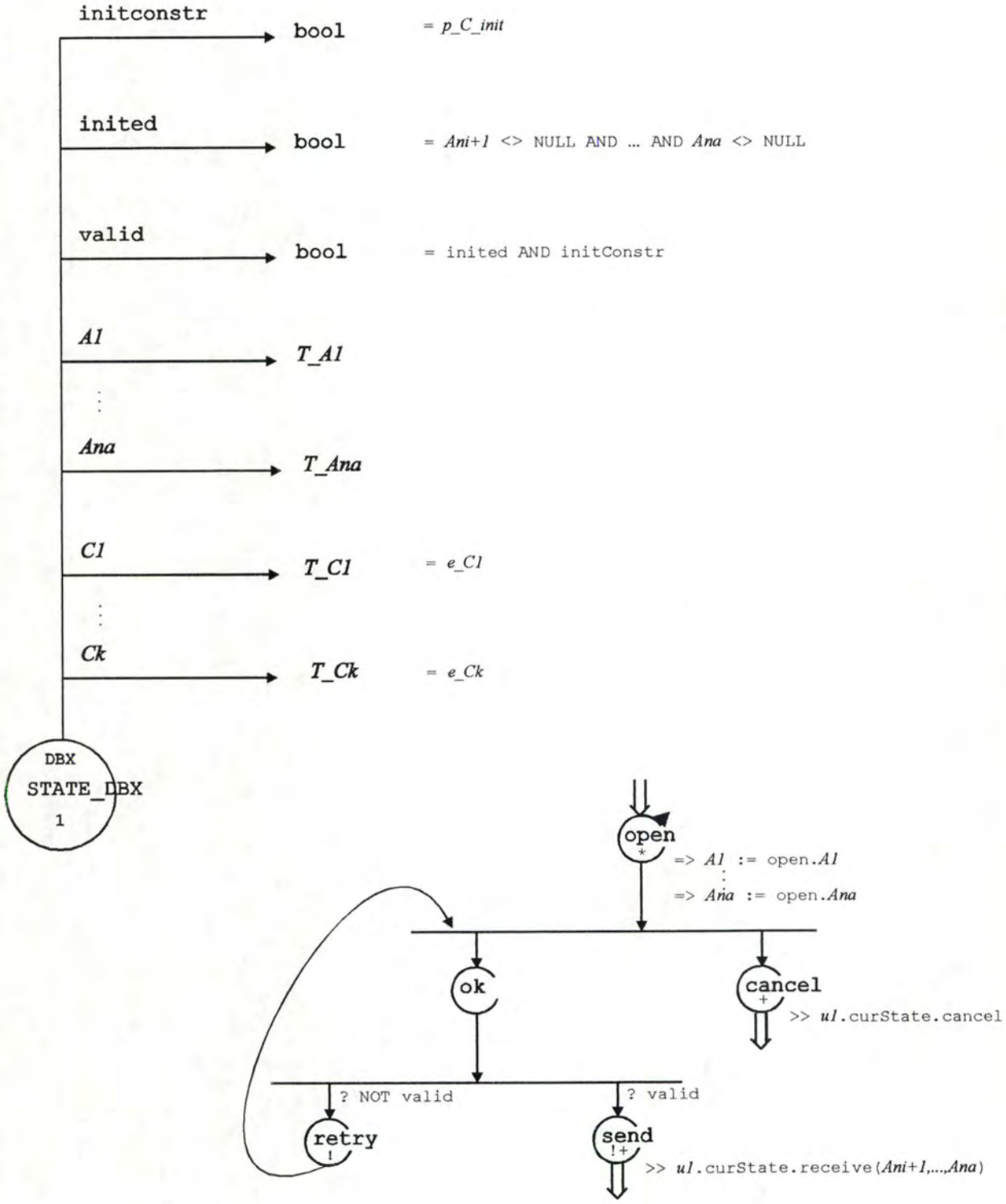


Figure 33 - L'objet state_dbx

Les attributs *A1*,...,*Ana*, *B1*,...,*Bnb* et *C1*,...,*Cnc* correspondent aux attributs réels de l'agent, c'est-à-dire ceux de la spécification ALBERT originale. Ils se décomposent en trois groupes :
- les attributs internes ;
- les attributs importés, dont les valeurs sont évaluées par des règles de dérivation en fonction des clauses de perception ;
- les attributs dérivés, dont les valeurs sont calculées par une règle de dérivation en fonction des valeurs des autres attributs.

Enfin, les attributs *A1*`Mod`,...,*Ana*`Mod`, à valeur booléenne, indiquent si les attributs internes correspondant *A1*,...,*Ana* ont fait l'objet d'un effet d'une occurrence d'action du changement courant.

8.10.2. ACTIONS

Deux actions de naissance sont associées à la classe d'objets `STATE`, pour distinguer l'état initial des autres états.

a) Etat initial

L'action `appear` prépare le premier état en initialisant les attributs pour lesquels il existe une clause **`Init`**. L'action `ask`, appelée par l'objet *u1*, fait ouvrir la boîte d'initialisation de l'état afin que l'utilisateur affecte des valeurs initiales aux autres attributs. Suivant la manière dont se termine le dialogue, c'est l'action `receive` ou `cancel` qui est rappelée. L'action `reset` annule toutes les modifications faites par l'utilisateur dans le but de recommencer la phase d'initialisation.

b) Etats ultérieurs

L'action `evolve` prépare un nouvel état à partir du précédent. Dans un premier stade, en effet, le nouvel état constitue une simple copie de l'état précédent. Lorsqu'une occurrence d'action apparaît dans le changement en construction, elle produit ses effets sur l'état en appelant les actions `set`*A1*,...,`set`*Ana* qui permettent de modifier les valeurs des attributs internes. A nouveau, l'action `reset` permet d'annuler toutes ces modifications et de recommencer à partir d'une nouvelle copie de l'état précédent.

8.11. LA CLASSE D'OBJETS CHANGE

Chaque instance de la classe `CHANGE` représente un changement de la vie de l'agent et est identifiée par une valeur de temps (attribut `time`). L'attribut `occs` contient la liste des occurrences (avec leurs arguments) figurant dans le changement. Cette liste est nécessairement vide au temps *t0*. L'action `append` permet d'ajouter une occurrence à la liste.

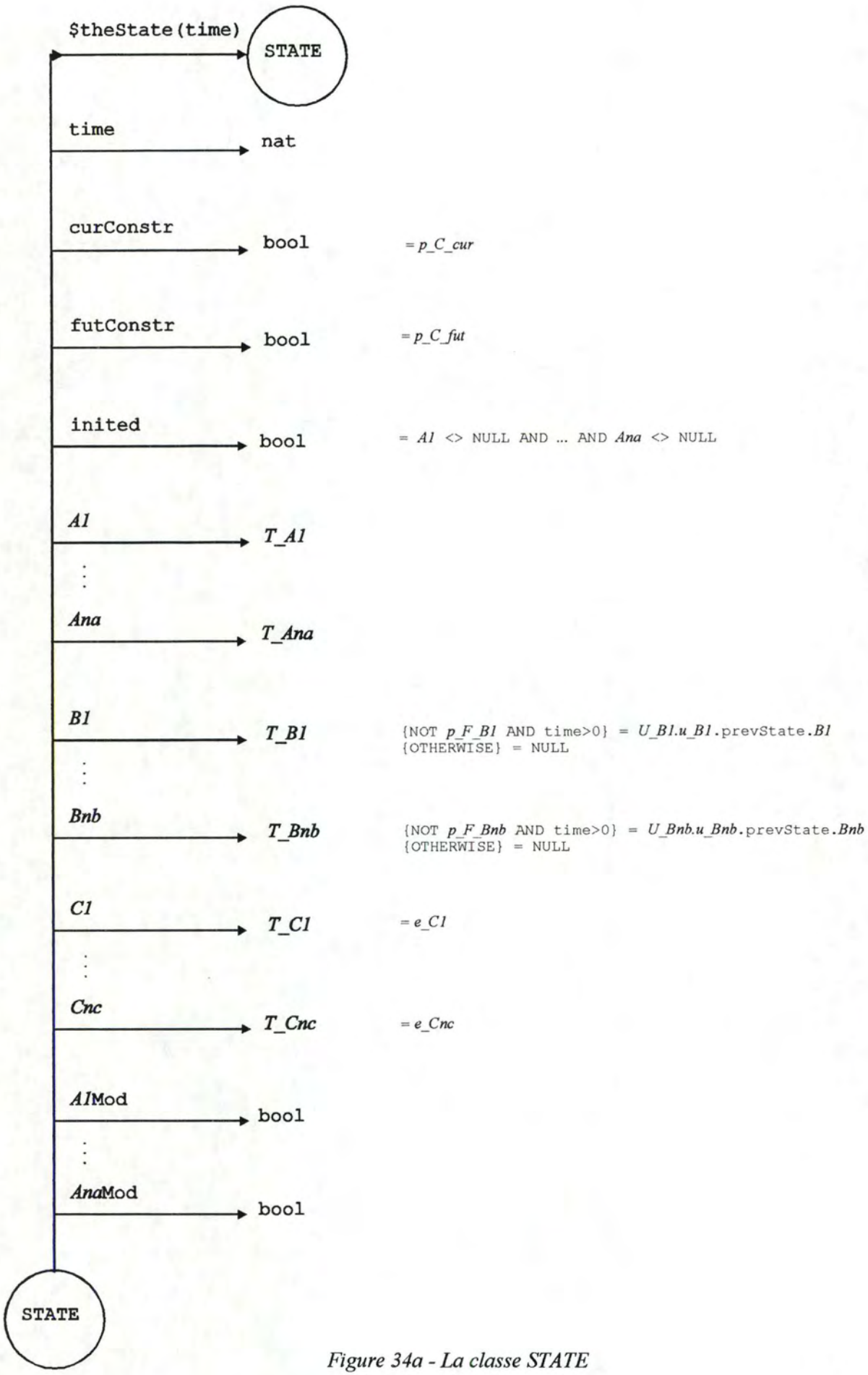


Figure 34a - La classe STATE

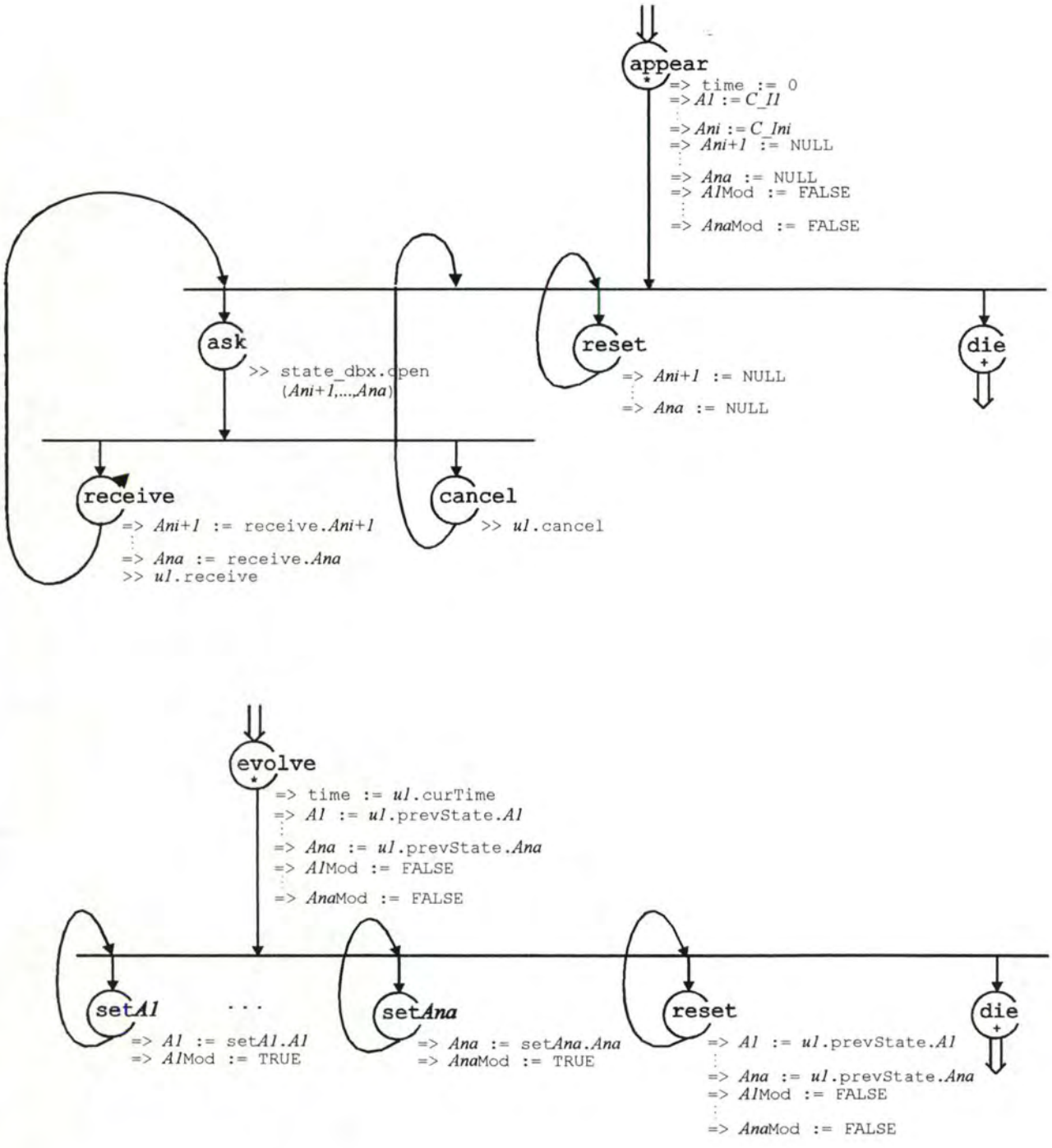


Figure 34b - La classe STATE

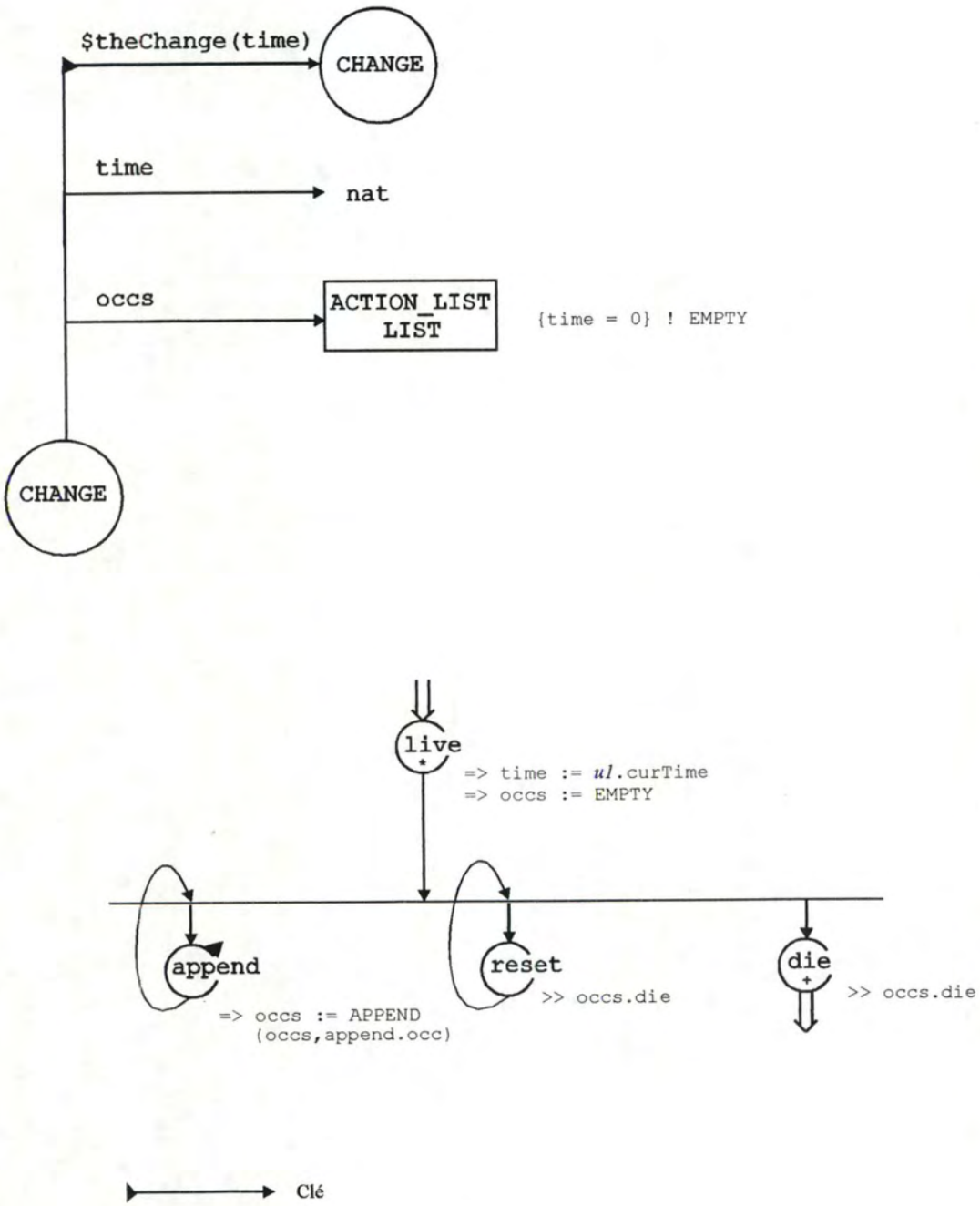


Figure 35 - La classe CHANGE

8.12. LA CLASSE D'OBJETS S1

Remarque : la classe d'objets *S1* représente, au sens ALBERT, une action interne avec des arguments. Il est très facile de déduire les spécifications OBLOG nécessaires pour représenter une action interne sans arguments.

8.12.1. LE PROBLÈME DE LA CANDIDATURE

a) Présentation du problème

Le problème de la candidature consiste à évaluer si oui ou non l'action *S1* est candidate au changement courant, en d'autres termes, si elle peut figurer, dans la boîte de dialogue de l'agent, parmi la liste des actions candidates que l'utilisateur peut ajouter dans le changement en construction. Cette décision repose sur deux questions :

- *S1* n'est-elle pas interdite par la clause de prévention qui lui est associée ?
- les effets de *S1* sont-ils compatibles avec ceux des occurrences d'actions du changement en construction ?

La difficulté d'évaluer la candidature réside dans le fait que la réponse à ces deux questions est fonction de la valeur des arguments de *S1*. Or, ceux-ci ne sont pas nécessairement connus au moment de l'évaluation.

b) Solution triviale

La solution triviale à ce problème consiste à raisonner, non pas en termes d'actions mais en termes d'occurrences d'action. Puisque les types de données sont finis, il est possible de passer en revue toutes les occurrences d'action possibles, c'est-à-dire tous les tuples d'arguments possibles, et de ne retenir que ceux qui sont acceptables. Cette solution est, naturellement, impraticable.

c) Solution idéale

La solution idéale consiste à identifier toutes les classes d'occurrences de *S1* qui ont le même "comportement" vis-à-vis de préventions, des obligations et des effets. Supposons par exemple que *S1* n'a qu'un argument, de type entier, et possède les contraintes d'effets et de responsabilité suivantes :

```
Effects of Actions
S1(n) : A1 = n
S1(5) : A2 = false
Responsibility
F(S1(n) with n<5 or n>21)
F(S1(14) / cond).
```

Ce petit exemple permet d'identifier facilement quatre classes d'occurrences disjointes qui résument à elles seules tous les cas possibles :

S1(5)
S1(*n*) with 6<=*n*<=13
S1(14)
S1(*n*) with 15<=*n*<=21.

Supposons que *cond* est vrai : *S1*(14) doit être enlevé de la liste des candidats. Si, de plus, *A2* a déjà été mis à jour dans l'état courant, *S1*(5) doit également être enlevé, etc.

Ce problèmes de décomposition en classes se complique lorsqu'on multiplie le nombre d'arguments et de contraintes. Cependant, ces deux nombres sont finis et connus ; il semble donc possible de trouver un algorithme qui résolve le problème. Il faut cependant remarquer que la question de la candidature doit être réévaluée dans chaque état puisque, dans les contraintes, les arguments peuvent dépendre de l'état. Exemple :

F(*S1*(*n*) with *n*>*A3*).

Remarque : de tels raisonnements relèvent en fait de l'*analyse statique* d'une spécification ALBERT (voir [Cou77]).

d) Solution pragmatique

Le problème de la candidature ne se poserait pas si on connaissait à l'avance les arguments de l'action *S1*. Or, on constate en pratique que, dans bien des cas, un argument ne peut prendre qu'une seule valeur (même si cette valeur évolue au cours du temps). C'est le cas par exemple pour l'action Décharge(PIECE) dans l'agent Véhicule : l'argument de Décharge ne peut prendre que la valeur de l'attribut qui référence la pièce actuellement sur le palettiseur :

F(Décharge(p) with p <> Contenu_P1).

C'est encore le cas lorsque une action est obligatoire et que la valeur d'un argument est imposée explicitement par la clause d'obligation :

O(Bouge(o, garage) / not contr.Atelier_occupé).

La solution que nous avons retenue consiste donc à proposer, quand elle existe, la seule valeur acceptable d'un argument. On dégage ainsi un premier sous-ensemble des occurrences possibles. Il se peut que ce sous-ensemble candidat contienne des occurrences qui soient inacceptables (voire qu'elles le soient toutes). Ce n'est qu'au moment où l'utilisateur introduira les valeurs des autres arguments que l'on pourra évaluer si l'occurrence est acceptable ou non.

Concrètement, imaginons que l'analyse de la spécification relative à *S1* se prête bien à identifier une valeur unique pour le premier argument *x1*. Supposons donc qu'il est possible de déduire de la clause de prévention globale de *S1*, une ou plusieurs clauses particulières :

F(*S1*(*x1*,...,*xnx*) / *p_F*) =>
F(*S1*(*x1*,...,*xnx*) with *x1* <> *e_F1_arg1* / *p_F1_arg1*)
...
F(*S1*(*x1*,...,*xnx*) with *x1* <> *e_Fj_arg1* / *p_Fj_arg1*)

où *e_F1_arg1,...,e_Fj_arg1* et *p_F1_arg1,...,p_Fj_arg1* ne dépendent pas des arguments. De même, supposons que l'analyse des clauses d'obligation permette de déduire :

p_O1_arg => x1 = e_O1_arg1
...
p_Ono_arg => x1 = e_Ono_arg1

où *e_O1_arg1,...,e_Ono_arg1* ne dépendent pas des arguments.

De cette façon, on peut construire une fonction qui essaie d'identifier une valeur unique pour *x1* (voir l'attribut `arg1Sugg`, plus loin dans cette section).

8.12.2. LE PROBLÈME DE LA COMPATIBILITÉ

Ce problème consiste à décider si une occurrence d'action est compatible avec les occurrences d'action présentes dans le changement en construction. Deux occurrences d'action sont dites incompatibles si leurs effets portent sur un même attribut. Comme l'effet d'une action peut dépendre de la valeur de ses arguments ou de l'état, la question de la compatibilité doit être évaluée dynamiquement, au cours de la vie de l'agent.

Ce problème peut être résolu relativement facilement. Etant donné une occurrence de *S1*, on peut déduire, sur base de la spécification, sur quels attributs porteront les effets de cette occurrence. De même, il est possible de retrouver tous les attributs mis à jour par toutes les occurrences du changement. Si aucun de ceux-là ne se retrouve parmi ceux-ci, l'occurrence n'est pas incompatible.

Nous avons préféré utiliser une technique plus simple, qui consiste à associer, dans l'état, un drapeau *(flag)* à chaque attribut interne (seul un attribut interne peut faire l'objet d'un effet). Chaque drapeau signale si son attribut a été mis à jour dans l'état courant. Il suffit donc de vérifier qu'aucun des attributs concernés par les effets d'une occurrence de *S1* n'a déjà été mis à jour.

8.12.3. ATTRIBUTS

L'attribut `generic` indique si l'objet représente l'action générique *S1* ou une occurrence de *S1*, autrement dit, si l'objet ne fait pas encore partie ou fait déjà partie d'un changement.

Les attributs `arg1Sugg`,...,`arg`*nx*`Sugg` calculent, pour chaque argument, la seule valeur possible si elle existe, suggérant ainsi une classe d'occurrences, que nous avons appelée action générique.

Les attributs suivants évaluent une série de propriétés à propos de cette action générique (dont certaines ont déjà été introduites dans l'objet *u1*) :
- `prev` indique si l'action est interdite ;
- `oblig` indique si l'action est obligatoire, c'est-à-dire si une des *no* obligations doit être honorée ;

- `oblig1` évalue si la première obligation doit encore être honorée, c'est-à-dire si, d'une part, sa condition d'obligation *p_O1* est vraie et, d'autre part, s'il n'existe pas déjà dans le changement une occurrence de *S1* qui honore cette obligation. Il en va de même pour `oblig2`,...,`oblig`*no* (cependant, une obligation n'est pas prise en compte tant que les obligations précédentes ne sont pas honorées) ;
- `incomp` indique si l'action est incompatible avec les occurrences d'action déjà présentes dans le changement en construction ;
- `contrad` indique si l'action est contradictoire ;
- `cand` indique si l'action est candidate ;
- `insted` indique si tous les arguments font l'objet d'une suggestion.

Enfin, les attributs `arg1`,...,`arg`*nx* enregistrent les *nx* arguments réels de l'action (une fois qu'ils sont connus).

Dans les profils *p_F*, *p_O1*, *p_E1*, *e_O1_arg1*, *p_F1_arg1*, *e_F1_arg1*, ou équivalents aux indices près, toute référence à un attribut doit être préfixée par "*u1*.`prevState`." et toute variable-argument doit être remplacée par "`arg1Sugg`",..., ou "`arg`*nx*`Sugg`", selon l'ordre de l'argument. Dans les profils *p_O1_arg*,...,*p_Ono_arg* (dans la définition de `oblig1`,...,`oblig`*no*), il faut procéder de même pour les attributs et remplacer toute variable-argument par "*s1*.`arg1`",..., ou "*s1*.`arg`*nx*", selon l'ordre de l'argument.

8.12.4. ACTIONS

Une fois que l'action générique candidate a été sélectionnée (action `try`), que ce soit par l'agent lui-même ou par l'utilisateur, son comportement est relativement simple.

Dans le cas où tous les arguments font l'objet d'une suggestion, il n'y a plus qu'à les accepter et à recopier les valeurs de `arg1Sugg`,...,`arg`*nx*`Sugg` dans `arg1`,...,`arg`*nx* (action `accept`). Dans le cas contraire, `ask` fait ouvrir la boîte d'instanciation afin de saisir les valeurs manquantes (suivant la terminaison du dialogue, c'est l'action `cancel` ou `receive` qui est rappelée). Ensuite, l'action ainsi instanciée produit ses effets sur les attributs (actions `haveEffect1`,...,`haveEffect`*ne*), informe l'agent concerné si *S1* est une action exportée (action `show`) et s'ajoute au changement (action `occur`). A ce stade, l'objet n'a plus qu'à attendre la mort, qui viendra à la suite d'une fonction *Reset*, *Reverse* ou *Kill*.

Dans les profils *p_E1*,...,*p_Ene*, *e_E1*,...,*e_Ene*, *p_X_V1*,...,*p_X_Vnv*, toute référence à un attribut doit être préfixée par "*u1*.`prevState`." et toute variable-argument doit être remplacée par "`arg1`",..., ou "`arg`*nx*", selon l'ordre de l'argument.

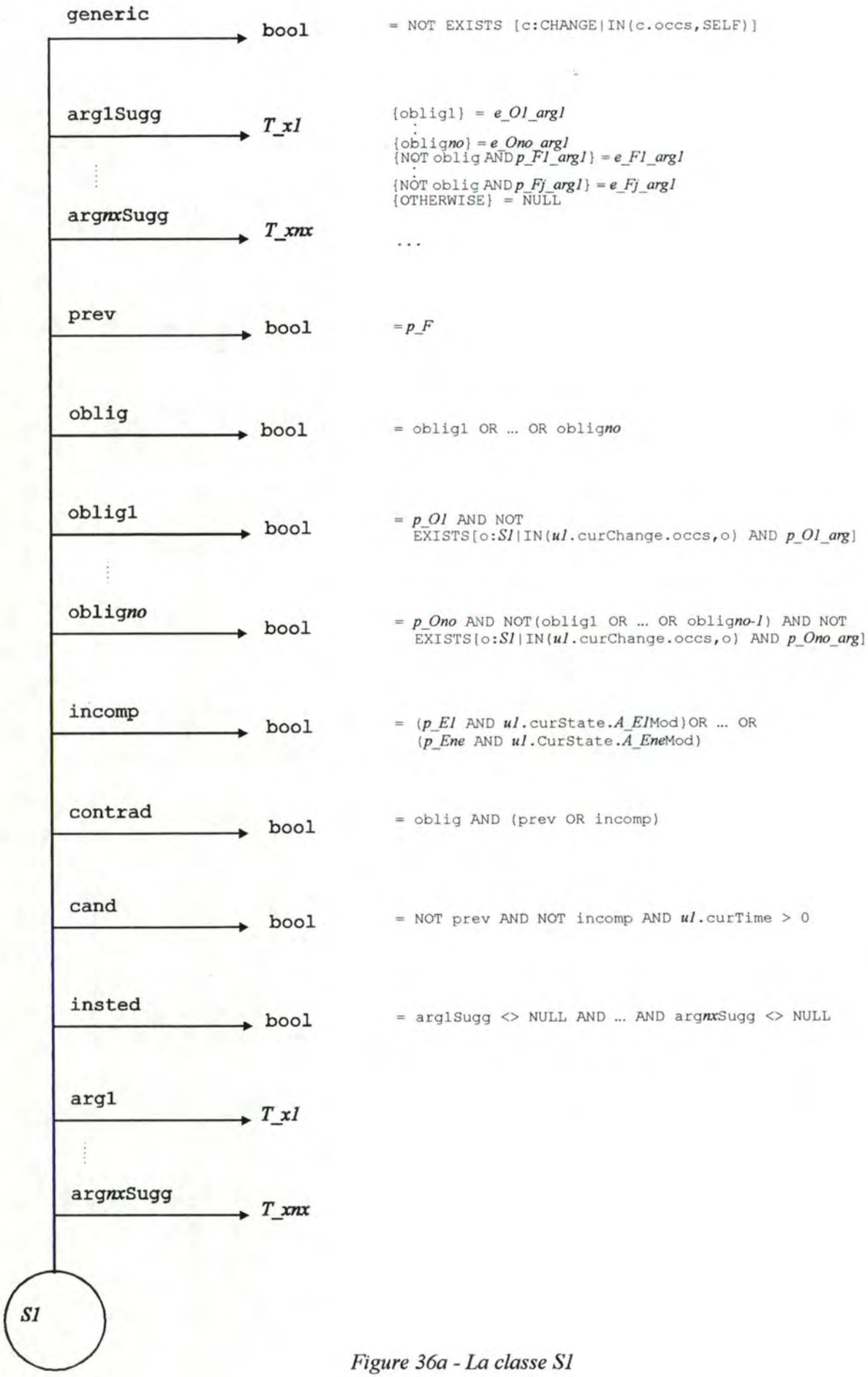


Figure 36a - La classe S1

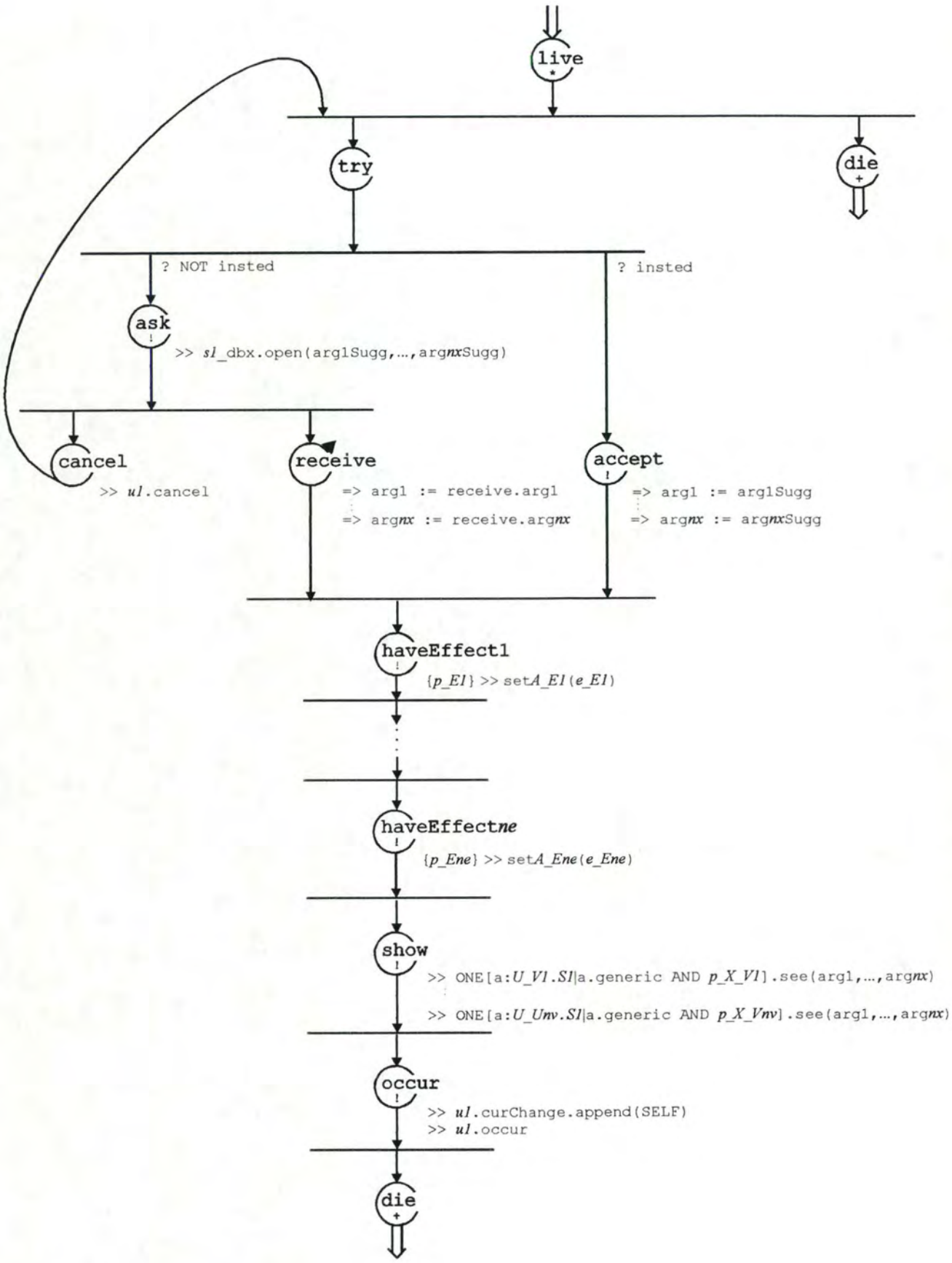


Figure 36b - La classe S1

8.13. L'OBJET S1_DBX

La boîte de dialogue `s1_dbx` a pour but de saisir des valeurs valides pour initialiser les *nx* arguments d'une action générique *S1*, candidate au changement. Ces valeurs sont transmises à l'objet *S1* correspondant.

Par valeurs valides (voir l'attribut `valid`), on entend des valeurs non-nulles telles que :
- elles ne sont pas interdites par la clause de prévention associée à *S1*, ou plus particulièrement par la partie de cette clause qui fait référence aux arguments ;
- si l'action *S1* est soumise à une obligation, elles constituent un tuple de valeurs correspondant à cette obligation ;
- elles ne provoquent pas d'incompatibilité avec les occurrences déjà présentes dans le changement.

Les contraintes *p_F_arg*, *p_E1_arg*,...,*p_Ene_arg* sont déduites respectivement de *p_F*, *p_E1*,...,*p_Ene* en ne retenant que la partie de la clause qui fait référence aux arguments. Dans tous les profils, toute référence à un attribut doit être préfixée par "*u1*`.prevState.`" et toute variable-argument doit être remplacée par "`arg1`",..., ou "`arg`*nx*", selon l'ordre de l'argument.

8.14. LA CLASSE D'OBJETS R1

La description d'une action externe *R1* est similaire à celle de *S1* mais beaucoup plus simple étant donné qu'on en connaît les arguments éventuels. Il n'y a donc pas de suggestions à faire, ni de valeurs d'arguments à saisir.

R1 se distingue de *S1* par l'action `see` et l'attribut `seen`. L'action `see` est déclenchée lorsque, dans l'agent *U_R1*, l'action *R1* a lieu (autrement dit, lorsque l'action externe *u_R1.R1* a lieu). L'attribut booléen `seen` est alors mis à vrai. Ce n'est qu'à partir de ce moment que *R1* peut prétendre au changement, du point de vue de *U1*. En effet, l'action *R1* est interdite si `seen` est faux et ne peut être obligatoire que quand `seen` est vrai.

Dans tous les profils, toute référence à un attribut doit être préfixée par "*u1*`.prevState.`" et toute variable-argument doit être remplacée par "`arg1`",..., ou "`arg`*nx*", selon l'ordre de l'argument.

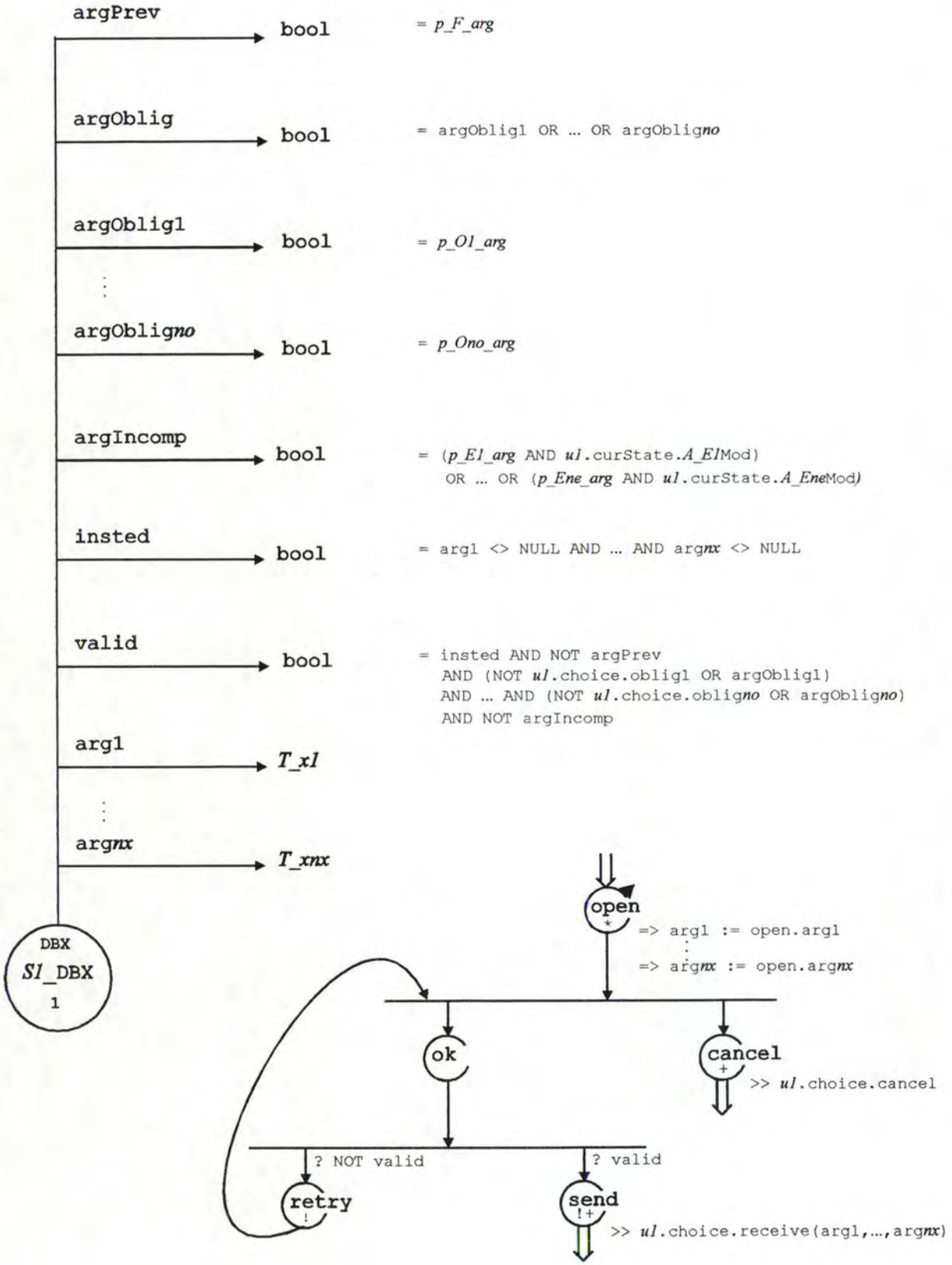


Figure 37 - L'objet s1_dbx

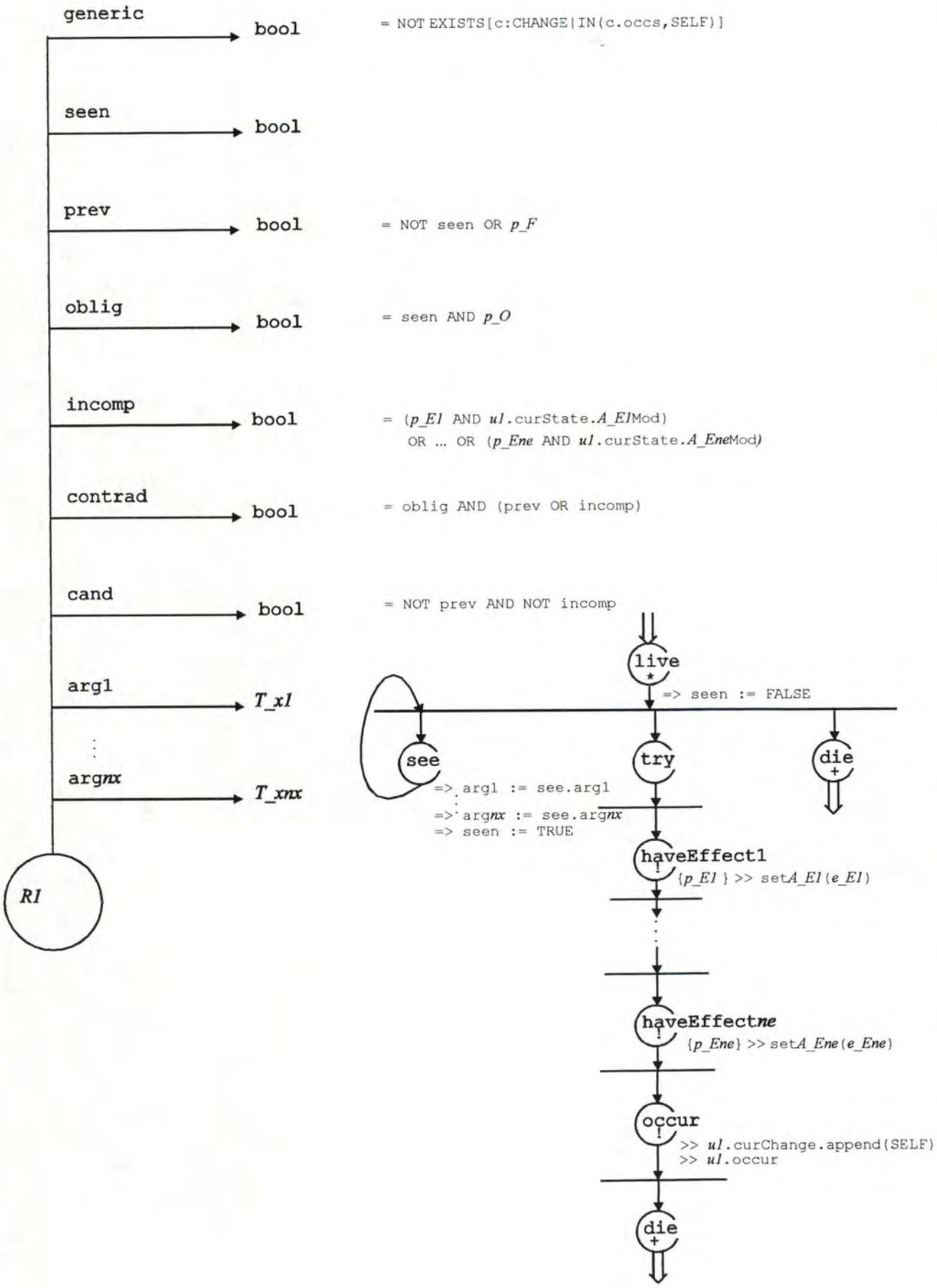


Figure 38 - La classe R1

8.15. A PROPOS DU TEMPS...

Vu l'importance des aspects temporels en ALBERT, nous avons préféré leur consacrer une section à part.

La méthode que nous avons présentée dans ce chapitre implémente fidèlement la notion de vie d'un agent en ALBERT, à savoir une séquence discontinue de changements et d'états. Le fait de conserver en permanence les informations sur ces changements et ces états permet de consulter le passé et, surtout, de revenir en arrière quand on s'aperçoit que la sous-vie menée jusqu'ici ne fait pas partie d'une vie valide.

En outre, ces informations sur le passé nous ouvrent de puissantes possibilités pour traiter plus efficacement les formules temporelles et, dans une moindre mesure, les engagements.

8.15.1. FORMULES TEMPORELLES

a) Formules LTP

Les formules LTP sont très faciles à transformer, dans notre méthode, puisqu'il suffit d'"implémenter" leur définition théorique.

Considérons par exemple le connecteur `sometimep`. Etant donné une suite d'états σ, la validité de la formule `sometimep` p, à un moment de référence i, sous une substitution globale v (notée $(\sigma,i,v) \models p$) est donnée par la définition suivante [Jun91a] :

$(\sigma,i,v) \models$ `sometimep` p ssi il existe j, $0 <= j <= i$ tel que $(\sigma,j,v) \models p$,

ce que l'on peut implémenter fidèlement en OBLOG comme suit :

```
sometimep p  ≡  EXISTS[s:STATE | s.time <= i AND p(s)],
```

où *p(s)* représente la proposition p évaluée à l'état `s`.

On procède de même pour les autres connecteurs LTP. Quant aux connecteurs bornés, il suffit d'adapter les formules. Exemple :

```
sometimep(<= d)  ≡  EXISTS[s:STATE | s.time <= i AND s.time >= i-d
    AND p(s)].
```

Cette puissance d'expressivité vaut aussi pour la composition des connecteurs :

```
sometimep(alwaysp p)  ≡  EXISTS[s1:STATE | s1.time <= i AND
    NOT EXISTS[s2:STATE | s2.time <= s1.time AND NOT p(s2)]].
```

Il suffit donc d'appliquer ces formules dans *p_C_cur*, avec i = `SELF.time`.

b) Formules LTF

Il n'est évidemment pas possible de consulter les états futurs de l'agent puisqu'on ne dispose que d'une sous-vie de l'agent. Cependant, on peut montrer qu'il est possible d'exprimer toutes les contraintes LTF par des contraintes sur l'état courant ou relatives à la mort de l'agent.

Exemple :

```
p => sometimef q
```

peut être transformé en la contrainte suivante, à vérifier avant la mort de l'agent :

```
NOT EXISTS[s1:STATE | p(s1) AND NOT EXISTS[s2:STATE | s2.time >= s1.time
    AND q(s2)]] .
```

A nouveau, il suffit d'intégrer ces définitions dans *p_C_cur* et dans *p_C_fut*. Il convient également, dans la classe d'objets STATE, de préfixer l'attribut `futConstr` du sigle "$" afin d'en faire un attribut de la métaclasse. En effet, *p_C_fut* ne fait plus référence à des attributs du dernier état qui enregistrent des informations sur le passé, mais consulte directement la classe des états.

Remarque : faute de fondement théorique qui nous permette de mélanger LTP et LTF, nous continuons à envisager séparément les deux logiques.

8.15.2. ENGAGEMENTS

a) Engagements sans contrainte de performance

Rappelons qu'au chapitre précédent, nous avions associé des préventions :
- aux actions-conséquences de l'engagement afin de garantir qu'elles aient lieu dans le bon ordre et uniquement après une occurrence de l'action-cause ;
- à l'action de mort pour empêcher l'agent de mourir avant d'avoir terminé ses engagements.

Le tableau PON était chargé de mémoriser les informations sur le passé nécessaires pour exprimer ces préventions.

Dans la deuxième méthode, ce tableau peut être supprimé puisqu'il est possible de consulter les changements passés. Nous proposons de ne plus transformer les engagements mais de procéder comme suit :
- dans la classe d'objets *S1*, modifier la règle de dérivation de l'attribut `prev` comme suit :

```
prev : bool = p_F OR p_M
```

où *p_M* est une contrainte relative aux engagements ;
- dans l'objet *u1*, modifier l'attribut `readyToDie` comme suit

```
readyToDie : bool = ready AND state.$futConstr AND change.$futConstr
```

après avoir ajouté, dans la classe d'objets CHANGE, l'attribut

```
$futConstr : bool = p_M_fut
```

où *p_M_fut* est une contrainte sur les changements qui doit être respectée à la mort de l'agent.

Exemple : soit l'engagement simple *R1* -> *S1* ; *S2*. Pour *S1*, on a :

p_M ≡ #(ALL[a:*S1* | NOT a.generic]) = #(ALL[a:*R1* | NOT a.generic]).

Le fait que *p_M* intervient dans prev entraîne que le nombre d'occurrences de *S1* ne peut pas excéder le nombre d'occurrences de *R1*. De même, pour *S2*, on a :

p_M ≡ #(ALL[a:*S2* | NOT a.generic]) = #(ALL[a:*R1* | NOT a.generic]).

Enfin, dans CHANGE, on a :

p_M_fut ≡ #(ALL[a:*S2* | NOT a.generic]) = #(ALL[a:*R1* | NOT a.generic])

qui entraîne qu'à la mort, il faut autant d'occurrences de *S2* que d'occurrences de *R1*.

b) Engagements avec contrainte de performance

Le traitement d'un engagement avec contrainte de performance est plus difficile. Au chapitre précédent, nous avions introduit la suite COT pour enregistrer les temps des causes des engagements tant qu'ils n'étaient pas honorés. La différence entre le temps courant et le premier temps de la suite ne pouvait pas dépasser le délai imposé par la contrainte de performance.

A présent, nous proposons de modifier, dans l'objet *u1*, l'attribut ready comme suit

```
ready : bool
     {curTime = 0} = inited AND curState.curConstr
     {curTime > 0} = NOT oblig AND curState.curConstr AND
     curChange.curConstr
```

après avoir ajouté, dans la classe d'objets CHANGE, l'attribut

curConstr : bool = *p_M_cur*

où *p_M_cur* est une contrainte sur le changement courant ;

Exemple : soit l'engagement simple *R1* -> sometimef(<= *d*) *S1* ; *S2*. Dans CHANGE, on a :

p_M_cur ≡ curTime - *t* < d

où *t* doit être remplacé par le temps du premier engagement non encore honoré, ce qui empêche l'agent de passer au temps suivant s'il est impossible d'honorer cet engagement (et a fortiori ceux qui suivent) dans les délais. La quantité *t* peut être calculée par une requête complexe sur la classe des changements.

8.16. EVALUATION

Dans ce chapitre, nous avons fait l'ébauche d'une méthode de transformation qui semble prometteuse.

• Tout d'abord, elle répond aux deux causes d'échec de la méthode précédente :
- premièrement, le modèle générique est construit de telle manière que l'utilisateur dirige l'animation de la spécification : il contrôle le déroulement du temps, initialise l'état et choisit les occurrences d'action. De ce fait, il peut réellement sélectionner, parmi l'ensemble des vies possibles des agents, celles qui sont pertinentes pour l'aider à valider la spécification ;
- deuxièmement, le modèle générique respecte fidèlement la sémantique d'ALBERT. Si l'animateur n'empêche pas de suivre des vies non valides, il le signale tôt ou tard à l'utilisateur et lui offre la possibilité de revenir en arrière *(backtracking)* afin de "corriger le tir". Contrairement à la première méthode, celle-ci tient compte de la simultanéité possible des actions en ALBERT.

• Ensuite, cette méthode est relativement simple à implémenter. En particulier, elle permet d'opérationaliser facilement les aspects temporels du langage (extensions de la logique temporelle et engagements) qui constituaient les principales difficultés de la méthode précédente. Pour être complet, il faut cependant signaler deux cas qui doivent encore être opérationalisés par l'analyste ("à la main") :
- les formules qui comportent une quantification sur un ensemble non tangible (c'est-à-dire qui n'appartient pas à la mémoire du système) ;
- les formules temporelles qui mélangent des connecteurs LTP et LTF.

• Enfin, rappelons que certaines parties de la spécification doivent être précisées ou généralisées. Il faut :
- généraliser la spécification d'un agent à une classe d'agent ;
- implémenter les objets de type boîte de dialogue en veillant à ce que les informations à afficher soient mémorisées sous un format "affichable" ;
- développer le traitement des formules temporelles et des engagements, qui n'a été qu'esquissé à la section précédente.

CONCLUSION

9. CONCLUSION

En conclusion, nous nous proposons d'évaluer les langages utilisés (section 9.1) avant d'aborder l'apport de ce travail (section 9.2). Pour terminer, la section 9.3 présente quelques perspectives d'avenir.

9.1. ÉVALUATION DES LANGAGES UTILISÉS

Une des difficultés de ce mémoire a été la découverte et la compréhension de deux langages de très haut niveau : ALBERT et OBLOG. Cette difficulté a été renforcée par le fait qu'il s'agit de langages encore en développement pour lesquels il n'existe pas de document de référence[1]. D'un autre côté, découvrir de nouveaux langages est toujours une expérience passionnante, que cette section se propose d'évaluer.

9.1.1. LE LANGAGE ALBERT

Le langage ALBERT est un langage de spécification des besoins de haut niveau. Il est conçu pour exprimer les besoins de systèmes composites comportant des contraintes de temps réel. Nous avons vu qu'un langage d'ingénierie des besoins (I.B.) devait réunir trois qualités ; il serait dès lors intéressant d'évaluer ALBERT selon ces trois critères.

a) Expressivité

L'expressivité est peut-être la qualité principale d'un langage d'I.B. Elle implique une correspondance naturelle entre les concepts du monde réel et ceux du langage. Les concepts d'agent, de vie, d'attributs et d'actions sont autant de concepts naturels qui permettent d'appréhender le monde réel. L'expressivité d'ALBERT tient surtout dans la grande diversité des contraintes qu'il permet d'exprimer : invariants, engagements, obligations, etc. En outre, il est possible, dans l'expression de ces contraintes, de faire référence au temps, que ce soit la vie passée, présente ou même future de l'agent.

[1] Le seul document de référence disponible concerne OBLOGkernel uniquement ([ESDI93])

ALBERT est un langage "orienté-vie". Cette caractéristique a l'énorme avantage de permettre l'expression de contraintes naturelles puissantes comme les engagements mais demande de la part du spécifieur de grandes qualités d'"imagination". En effet, pour comprendre la spécification d'un agent, il faut pouvoir se représenter la vie entière de cet agent. En particulier, il n'est pas facile de manipuler les connecteurs temporels. En fait, cette difficulté à penser "orienté-vie" vient sans doute du fait que les langages de programmation nous ont inculqué le réflexe de raisonner "orienté-transition".

ALBERT contient tous les "ingrédients" de l'expressivité. Cependant, il manque toute une série de "macros", de raccourcis de langage, qui rendraient l'utilisation d'ALBERT plus "confortable". La syntaxe d'un engagement, par exemple, est relativement rigide et ne permet pas d'exprimer aisément des engagements complexes. Les versions ultérieures du langage enrichiront probablement ce que nous avons appelé le "langage utilisateur". Nous avons, pour notre part, proposé et utilisé un certain nombre d'extensions concernant ce langage utilisateur.

b) Structuration

La décomposition d'un système en agents, la distinction entre déclarations et contraintes et la classification de ces dernières en rubriques, et surtout la méthode incrémentale de spécification sont autant d'éléments qui aident à structurer les spécifications. Dans le cadre de ce mémoire, nous n'avons pas abordé les mécanismes de paramétrisation et d'héritage qui seront supportés par les versions ultérieures d'ALBERT (voir à ce propos [Dub92]).

c) Formalisation

Enfin, ALBERT possède une sémantique formelle rigoureuse, basée sur la logique du premier ordre et ses variantes temporelle et déontique. Cette base formelle était un préalable nécessaire à notre démarche de prototypage.

9.1.2. LE LANGAGE OBLOG

OBLOG, plus précisément OBLOGlight, est un langage de spécification de conception de haut niveau. Il permet de modéliser de manière souple et relativement intuitive des systèmes d'information composés d'objets concurrents. De plus, ce langage s'inscrit dans une approche cohérente comprenant également un outil et une méthode.

La version actuelle du langage, appelée OBLOGkernel, est relativement pauvre. Dans le cadre de ce mémoire, nous avons utilisé OBLOGlight, la version ultérieure en cours de développement, qui a pour ambition de supporter l'analyse des besoins. Nous avons utilisé OBLOGlight dans deux contextes différents : comme langage de spécification des besoins (au chapitre 8) et comme langage de spécification de conception (au chapitre 9).

a) OBLOGlight comme langage de spécification des besoins

Par rapport à OBLOGkernel, OBLOGlight offre des extensions qui en font un langage très expressif : concepts de métaclasse, attributs et actions dérivés, interface, langage d'expression enrichi, etc... OBLOGlight offre également de nombreux mécanismes de structuration, tels que les unités, l'héritage et l'agrégation, et supporte une sémantique formelle.

Pour toutes ces qualités et par la nature même du langage, il nous semble qu'OBLOGlight peut servir à décrire les besoins d'un système, autrement dit à supporter l'analyse des besoins.

Cependant, il est certain qu'OBLOGlight est beaucoup moins riche qu'ALBERT. En particulier, OBLOGlight est un langage "orienté-transition" et ne permet pas d'exprimer des contraintes temporelles. Nous avons montré qu'il n'était pas possible de transformer une spécification ALBERT en une spécification OBLOG équivalente.

b) OBLOGlight comme langage de spécification de conception

Si OBLOGlight est un langage d'I.B. pauvre, il est par contre très bien conçu pour spécifier une application. Ainsi, nous avons pu concevoir l'architecture d'un animateur beaucoup plus rapidement et facilement que nous n'aurions pu le faire avec un langage classique de troisième génération.

Parmi les avantages d'OBLOGlight à ce niveau, citons la concurrence des objets, les objets de type boîte de dialogue, l'héritage, le langage de requête, etc... La plupart des extensions d'OBLOGlight s'avèrent très utiles.

9.2. APPORTS DE CE TRAVAIL

a) Apport principal

Les langages formels d'I.B., tels qu'ALBERT, fournissent des règles rigoureuses d'interprétation qui garantissent la précision et l'absence d'ambiguïtés. En outre, ils supportent des raisonnements formels tels que le contrôle de cohérence ou de complétude. Cependant, ces avantages se paient par le fait que les langages formels sont difficiles à comprendre et à utiliser. Il est certain que le client n'a pas la possibilité matérielle de comprendre le cahier des charges du système à développer s'il est rédigé dans un langage formel. De plus, nous avons fait l'expérience que même pour un informaticien habitué à la formalisation, il n'est pas facile de manipuler un langage tel qu'ALBERT.

Dans ce contexte, le prototypage des spécifications des besoins apparaît particulièrement intéressant pour permettre à l'informaticien de valider leur l'adéquation :
- dans un premier temps, par rapport au modèle mental qu'il s'est fait du problème ;
- dans un second temps, par rapport aux besoins effectifs du client, ce qui permet de réimpliquer celui-ci dans le développement du projet.

Dans ce mémoire, nous avons jeté les bases d'un système de prototypage de spécifications ALBERT. Plus précisément, nous avons spécifié, en OBLOGlight, l'architecture générique d'un animateur de spécifications ALBERT. Cette architecture doit être instanciée en fonction d'une spécification particulière. Le code exécutable de l'animateur peut ensuite être généré automatiquement grâce au générateur de code de l'environnement OBLOG.

Pour ce qui est des fonctionnalités du système de prototypage, signalons que l'utilisateur contrôle entièrement le déroulement de la vie des agents. Il contrôle la marche du temps dans le système, en avant comme en arrière (possibilité de *backtracking*), influe sur le comportement des agents et peut consulter leur passé.

La nécessité d'une telle interaction avec l'utilisateur ne nous était pas apparue clairement au départ. En effet, habituellement, le prototypage porte sur des logiciels, c'est-à-dire des systèmes ouverts. Le prototypage consiste à tester si l'exécution du programme-prototype fournit les résultats attendus en fonction des arguments donnés par le testeur qui, en l'occurrence, simule le comportement de l'utilisateur final du logiciel.

A l'inverse, ALBERT décrit des systèmes composites, soit des systèmes fermés, comprenant à la fois les composants logiciels et leurs utilisateurs. Le prototypage de tels systèmes n'a de sens que si le testeur intervient, à l'intérieur même du système, pour poser des choix et lever des indéterminismes. Cette réflexion nous a amené à adapter les notions de prototype et de prototypage dans le cadre des spécifications des besoins de systèmes composites (et à utiliser de préférence les termes *animateur* et *animation*).

b) Apports secondaires

Comme nous l'avions annoncé, ce mémoire a apporté également certains résultats en ce qui concerne les deux langages utilisés.

• La spécification de l'atelier d'usinage (un cas réel) constitue une des premières applications du langage ALBERT et apporte une matière intéressante pour ces concepteurs. En outre, cette tâche nous a permis de formuler certaines propositions d'extension du langage et de faire la critique de ses qualités d'expressivité. Par nos remarques également, nous avons pu attirer l'attention des concepteurs sur certains aspects du langage qui ne sont pas clairement expliqués.

• L'utilisation d'OBLOG dans le cadre de ce mémoire confirme, si besoin est, l'utilité des extensions d'OBLOGlight par rapport à OBLOGkernel. La confrontation entre ALBERT et OBLOG permet de formuler d'autres propositions d'extension afin qu'OBLOG puisse couvrir la phase d'I.B. Cette évolution devrait aller de pair, selon nous, avec le développement d'un outil d'animation au sein de l'environnement OBLOG.

• Enfin, l'expérience de la transformation de spécifications ALBERT en spécifications OBLOG nous a permis de mieux comprendre la spécificité d'un langage de spécification des besoins par rapport à d'autres langages utilisés dans le cycle de vie d'un logiciel et, en particulier, les limitations d'un langage de programmation à des fins de spécification.

9.3. PERSPECTIVES D'AVENIR

Nous sommes conscient que ce travail n'est pas achevé. Nous avons investi beaucoup d'efforts et de temps pour comprendre en profondeur la sémantique d'ALBERT et d'OBLOG, préciser ou poser les hypothèses d'extension de ces langages sans lesquelles il était impossible de travailler, inventer une syntaxe adéquate et créer entièrement une étude de cas. Ensuite, nous avons développé une première méthode de prototypage qui, bien qu'intéressante pour ses conclusions, a débouché sur un échec. Pour toutes ces raisons, nous n'avons pas pu mener la réalisation de notre système de prototypage aussi loin que nous ne l'aurions souhaité.

• D'abord, comme nous l'avons déjà mentionné, certaines parties de la spécification du modèle générique doivent être précisées ou généralisées.

• Ensuite, il reste à implémenter le système de prototypage correspondant à la deuxième méthode en construisant un programme qui, étant donné une spécification ALBERT, est capable de produire la spécification OBLOG de son animateur, c'est-à-dire capable d'instancier le modèle générique à une spécification donnée. (Ce programme peut-être écrit dans n'importe quel langage de programmation.) Le code exécutable de l'animateur peut ensuite être généré automatiquement à partir de la spécification OBLOG ainsi obtenue. (Cependant, comme l'atelier logiciel qui supportera OBLOGlight n'est pas encore disponible, on peut prendre la décision de redéfinir le modèle générique en OBLOGkernel ou dans un langage de programmation de haut niveau.)

• Enfin, si l'architecture proposée peut servir de point de départ à l'implémentation d'une première version du système de prototypage, de nombreuses améliorations peuvent être envisagées, notamment :
- en recherchant un algorithme plus perfectionné de sélection des actions candidates. Il est possible, par exemple, d'énumérer toutes les occurrences d'action possibles lorsque le domaine de l'argument est restreint (c'est le cas pour les types booléens et les types d'identifiant d'agent) ;
- en affinant la fonction de *backtracking* de façon à pouvoir annuler une occurrence parmi les occurrences d'action du changement en construction ;
- et surtout en perfectionnant l'interface utilisateur. Il serait intéressant de voir à l'écran certains fragments de la spécification comme les contraintes sur l'état ou, lors de la saisie des arguments, les contraintes associées à ces arguments. Une autre idée consisterait à afficher la progression des engagements.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

OUVRAGES ALBERT

[Dub93a] Eric Dubois, Philippe Du Bois and Michaël Petit, "Elicitating and Formalising Requirements for C.I.M. Information systems", submitted to the 5th Conference on Advanced Information systems engineering - CAISE'93, Paris, June 1993

[Dub93b] Eric Dubois, Philippe Du Bois and Michaël Petit, "O-O Requirements Analysis : an Agent Perspective", *Proc. of the 7th European Conference on Object-Oriented Programming - ECOOP'93* (to appear), Kaiserslautern (Germany), July 1993, LNCS, Springer-Verlag

[Dub93c] Eric Dubois, Philippe Du Bois and Michaël Petit, "ALBERT : an Agent-oriented Language for Building and Elicitating Requirements for real-Time systems", submitted to the 4th European Software Engineering Conference - ESEC'93 , Garmisch (Germany), September 1993

[Pet92] M. Petit, *Construction et formalisation de spécifications conceptuelles pour les systèmes productiques*, mémoire, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur (Belgium), septembre 1992

OUVRAGES OBLOG

[ESDI92a] ESDI SA, *Pragmatic introduction to the OBLOG approach in system design*, Notes du séminaire présenté dans le cadre du cours de Méthodologie de développement de logiciels, m.a., FUNDP Namur, Institut d'informatique, Lisbon, April 92

[ESDI92b] ESDI SA, "OBLOGkernel - Language Summary and Workbench Presentation", Document, Lisbon, September 92

[ESDI92c] ESDI SA, "OBLOG", Folder, Lisbon, September 92

[ESDI92d] ESDI SA, "OBLOG-CASE V1.0 - Software Product Specification", Lisbon, October 92

[ESDI92e] ESDI SA, "Towards OBLOGlight - Draft Language Proposal", Lisbon, October 92

[ESDI92f] ESDI SA, "Towards OBLOGlight - Draft Language Reference", Lisbon, November 92

[ESDI93] ESDI SA, *OBLOG-CASE V1.0 - User's Guide*, Lisbon, April 93

[Ser91a] Amílcar Sernadas, C. Sernadas, P. Gouveia, P. Resende and J. Gouveia, "OBLOG - An informal introduction", Computer Science Group INESC, Lisbon, January 91

[Ser91b] A. Sernadas, H.-D. Ehrich, "What is an object after all ?", *Object-oriented databases : analysis, design and construction*, R. Meersman, W. Kent, S. Khosla (eds), North Holland, 1991, pp. 36-69

AUTRES OUVRAGES

[Bod89] François Bodard et Yves Pigneur, *Conception assistée des systèmes d'information*, coll. MIPS, Masson, Paris, 1989

[Coa90] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1990

[Cou77] P. Cousot and R. Cousot, "Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximative of fixpoints", *Proc. Fourth ACM Symps. on Principles of Programming Languages*, pp. 238-252, 1977

[deC92] Dennis de Champeaux and Penelope Faure, "A comparative study of object-oriented analysis methods", *Journal of Object-Oriented Programming*, March-April 1992

[Dub82] Eric Dubois, J.-P. Finance and A. van Lamsweerde, "Towards a Deductive Approach to Information System Specification and Design", *Requirements Engineering Environment*, Y. Ohno (ed), North-Holland, 1982

[Dub91a] Eric Dubois, J. Hagelstein and A. Rifaut, *From Natural Language Processing to Logic for Expert Systems*, Chapter 6 : "a Formal Language for the Requirements Engineering of Composite Systems", A. Thayse (editor), Wiley, 1991

[Dub91b] Eric Dubois, "Use of deontic logic in the requirements engineering of composite systems", *Proc. of the first international workshop on deontic logic in computer science*, J.J. Meyer and R.J. Wieringa (eds), Amsterdam (The Netherlands), December 1991

[Dub92] Eric Dubois, Philippe Du Bois and André Rifaut, "Elaborating, Structuring and Expressing Formal Requirements of Composite Systems", *Proc. of the 4th Conference on Advanced Information systems engineering - CAISE'92*, Manchester (UK), May 1992, LNCS 593, Springer-Verlag, 1992

[Hab90] Naji Habra, *A Transformational Method for Functional Prototyping*, Phd thesis, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur (Belgium), September 1990

[Hoa85] C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985

[ICARUS89] Eric Dubois, J.-P. Finance and A. van Lamsweerde, "First Description of the ICARUS Language Kernel for the Product Level", *Report*, ICARUS ESPRIT project 2537, June 1989

[ICARUS90] "First Version of the ICARUS Product and Process Language", *Report*, ICARUS ESPRIT project 2537, March 1990

[Jun91a] Ralf Jungclaus, Gunter Saake, Thorsten Hartmann and Cristina Sernadas, *Object-Oriented Specification of Information Systems : The TROLL Language*, Technische Universität Braunschweig, Informatik-Berichte, Braunschweig, 1991

[Jun91b] Ralf Jungclaus, Gunter Saake and Cristina Sernadas, "Formal Specification of object systems", *Proc. of TAPSOFT'91 Vol. 2*, S. Abramsky and T. Maibaum (eds), pp. 60-82, Brighton (UK), 1991, LNCS 494, Springer-Verlag

[Krö87] Fred Kröger, *Temporal Logic of Programs*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin Heidelberg, 1987

[Lam88] D. A. Lamb, *Software Engineering : Planning for Change*, Prentice-Hall, 1988

[Lur92] Jesus Lurdes and Rogério Carapuça, "Automatic Generation of Documentation for Information Systems", INESC, Lisbon, 1992

ANNEXE
ÉTUDE DE CAS

A. ÉTUDE DE CAS

Le cas présenté ici a déjà été évoqué à plusieurs reprises au cours de ce mémoire. Il s'agit d'une version simplifiée de l'atelier de production installé au centre de ressources CIM du Centre de Recherche Public Henri Tudor à Luxembourg. Ce cas nous a servi pour illustrer à la fois la spécification des besoins d'un système composite et le prototypage de cette spécification.

La section A.1 définit l'énoncé précis du cas. La section A.2 décompose le problème conformément au modèle-procédé. Les sections A.3 à A.7 présentent les types de donnée utilisés et les versions successives de la spécification du cas. Enfin, les sections A.8 et A.9 illustrent successivement la première et la deuxième méthode de prototypage développées dans le corps de ce mémoire, en utilisant la spécification du véhicule.

A.1. ÉNONCÉ DU CAS

L'atelier de production transforme des cylindres métalliques en diverses pièces : boulons, rivets, vis, etc... L'atelier comprend les éléments suivants (voir figure 39) :
- un *magasin* approvisionné en cylindres et recevant les produits finis ;
- un *robot* industriel, équipé d'une main, qui assure le transport des pièces entre le magasin et l'atelier proprement dit ;
- un poste de bridage automatique (ou *brideur*) chargé de brider, c'est-à-dire placer dans un étau, les pièces à destination de la fraiseuse ;
- un *véhicule* optoguidé assurant le transport des pièces à usiner entre les machines-outils et le brideur ;
- deux machines-outils : une *fraiseuse* et un *tour* ;
- un *contrôleur* qui dirige l'ensemble de la production, en assurant le séquencement des différentes opérations de production.

Le cycle de production, par exemple d'une vis, comprend les étapes suivantes :
- le robot choisit un cylindre disponible dans le stock du magasin ;
- après pivotage, le robot dépose le cylindre sur une palette du brideur ;
- le véhicule charge la palette et son contenu, la transporte et la décharge sur l'entrée-sortie du tour ;
- le tour usine le cylindre, d'abord en l'élimant pour faire apparaître la tête puis en plaçant un pas de vis ;
- le véhicule transporte la pièce (et sa palette) au brideur, qui bride la pièce, puis à la fraiseuse ;

- la fraiseuse produit la rainure de la tête de la vis ;
- le véhicule ramène vis et palette au brideur, qui débride la pièce ;
- le robot range la vis dans le magasin.

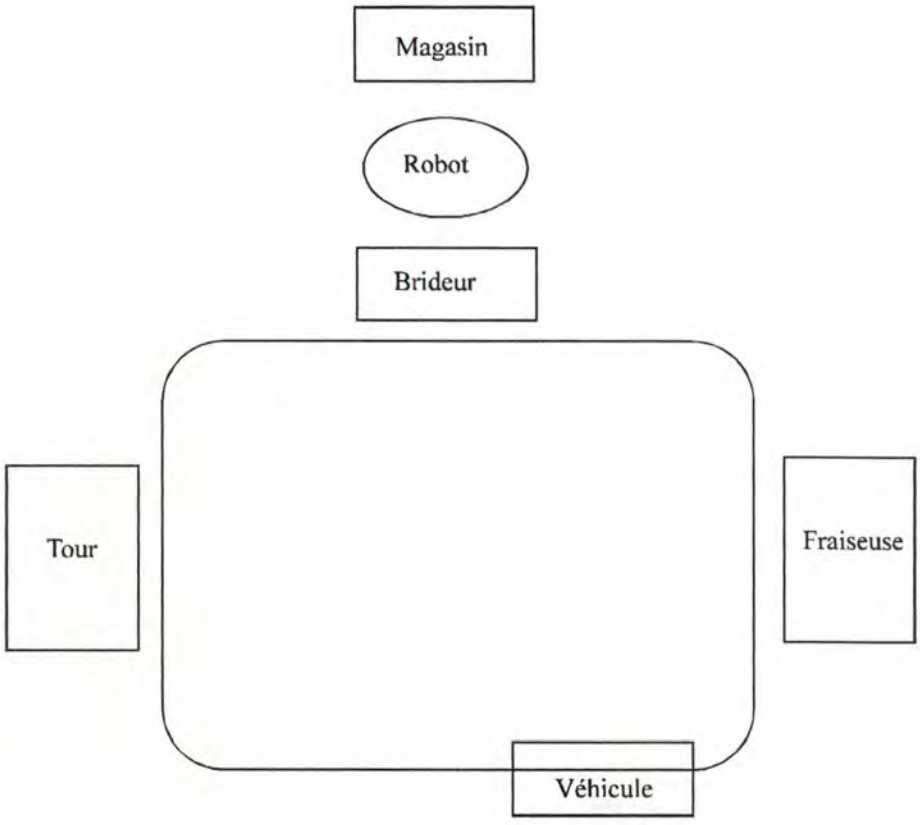


Figure 39 - Schéma de l'atelier

Remarques :
- le contrôleur peut gérer plusieurs commandes en même temps ;
- pour simplifier, les machines-outils et le poste de bridage ne possèdent chacun qu'une entrée-sortie ;
- le véhicule est surmonté d'un palettiseur à deux places, ce qui signifie qu'il est capable d'échanger son contenu contre celui d'une machine. Pour simplifier, on fera l'hypothèse que le véhicule ne peut jamais porter deux pièces simultanément que dans le cas d'un échange, ceci pour éviter des situations éventuelles d'interblocage.

Enfin, il reste à parler du rôle d'un dernier agent, extérieur à l'atelier : le *gestionnaire*. C'est lui qui émet les ordres de production, approvisionne le stock en matières premières ou retire les produits finis.

A.2. DÉCOMPOSITION DU PROBLÈME

Cette décomposition a déjà été suggérée lors de la présentation du modèle-procédé (section 3.4). Elle prend deux formes : une progression verticale et une progression horizontale.

• Une progression verticale possible est la suivant :
- 1ère étape : l'entreprise est considérée comme un seul agent ;
- 2ème étape : l'entreprise est composée des agents `Atelier` et `Gestionnaire` ;
- 3ème étape : l'atelier est composé des agents `Contrôleur`, `Magasin`, `Robot`, `Brideur`, `Véhicule` et `Machine_outil`.

La figure 40 présente la hiérarchie des agents qui résultent de cette décomposition.

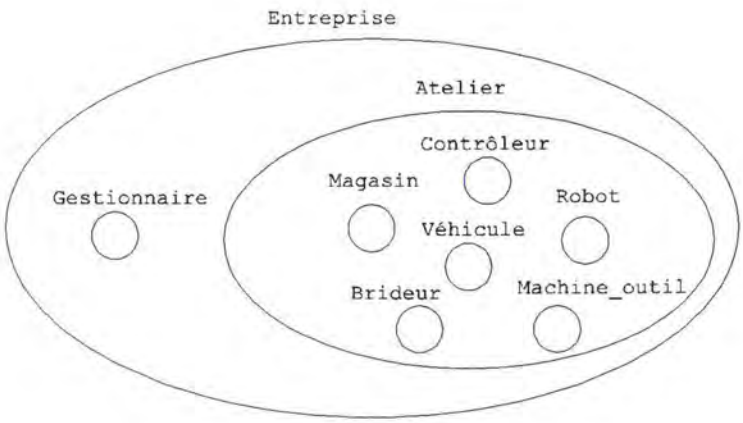


Figure 40 - Hiérarchie des agents

• Vu la complexité qui apparaît déjà lors de la troisième étape, celle-ci fera l'objet d'une progression horizontale, en jouant sur les hypothèses d'omniscience et de fiabilité.

Le plan des sections A.4 à A.7 est calqué sur cette double décomposition (la section A.3 introduit le types de données utilisés) :
- l'entreprise est un agent virtuel, abstrait. Il n'est pas nécessaire de spécifier cet agent explicitement en tant qu'agent individuel (1ère étape) ;
- la section A.4 présente la spécification de l'entreprise en tant qu'agent complexe (2ème étape) ;
- la section A.5 présente la spécification de l'atelier en tant qu'agent complexe, en posant les hypothèses d'omniscience et de fiabilité (3ème étape, 1er temps) ;
- la section A.6 lève l'hypothèse d'omniscience (3ème étape, 2ème temps) ;
- la section A.7 lève l'hypothèse de fiabilité (3ème étape, 3ème temps).

A.3. TYPES DE DONNÉE

La spécification suivante présente les types de données *(data types)* qui seront utilisés dans toutes les spécifications ultérieures.

A.4. SPÉCIFICATION DE L'ENTREPRISE

Le spécifications qui suivent présentent la spécification de l'entreprise considérée comme un agent complexe, c'est-à-dire composée d'un gestionnaire et d'un atelier.

Data types

```
GESTIONNAIRE = {gest}
CONTROLEUR = {contr}
VEHICULE = {véhic}
MACHINE_OUTIL = {fraiseuse, tour}

TYPE = {cylindre, boulon, rivet, vis, vis_1, vis_2,...}
PIECE = UNION[CP[Type : TYPE, Numéro : NAT, En_étau : BOOL,...], rien]
PROGRAMME = TYPE
STATUT = {indisponible, vide, réservé, prêt, occupé, actif, mobile, fini}
LIEU = UNION[MAGASIN, ROBOT, BRIDEUR, VEHICULE, MACHINE_OUTIL]
OPTION = {en_étau, hors_étau}
```

Atelier

Declarations

Attributes

```
Stock : SET[PIECE] -> Gestionnaire
Capacité_stock : INT
Nb_commandes, Nb_cylindres_disp : INT
```

Derived Attributes

```
Occupé : BOOL = (Nb_commandes > 0)
```

Actions

```
Saisis(PIECE)
Dépose(PIECE) -> Gestionnaire

Stocke(PIECE), Déstocke(PIECE) <- Gestionnaire
Produit(TYPE) <- Gestionnaire
```

Constraints

State Behaviour

```
In(Stock, p) => sometimef(not In(Stock, p))
```

Init

```
Stock = {}
Capacité_stock = 50
Nb_commandes = 0
Nb_cylindres_disp = 0
```

Effects of Actions

```
gest.Produit(t):
      Nb_commandes = Nb_commandes + 1
      Nb_cylindres_disp = Nb_cylindres_disp - 1
Saisis(p): Stock = Remove(Stock, p)
Dépose(p):
      Stock = Add(Stock, p)
      Nb_commandes = Nb_commandes - 1
gest.Stocke(p):
      Stock = Add(Stock, p)
      Nb_cylindres_disp = Nb_cylindres_disp + 1
gest.Déstocke(p): Stock = Remove(Stock, p)
```

Commitments

```
gest.Produit(t) -> sometimef(<= 20 min) (Saisis(p1); Dépose(p2))
      with p1.Type = cylindre and p2.Type = t
```

Responsibility

```
F(Saisis(p) / not In(Stock, p))
```

Perception

```
X(gest.Produit(t) / Nb_cylindres_disp > 0 and t <> cylindre)
X(gest.Stocke(p) /
      not Occupé and not Card(Stock) = Capacité_stock and p.Type =
      cylindre)
X(gest.Déstocke(p) / not Occupé and In(Stock, p) and p.Type <> cylindre
```

Publicity

```
X(Stock.gest / not Occupé)
```

Gestionnaire

Declarations

Attributes

Nom : STRING

Stock : SET[PIECE] <- Atelier

Actions

Stocke(PIECE), Déstocke(PIECE) -> Atelier
Produit(TYPE) -> Atelier

Dépose(PIECE) <- Atelier

Constraints

Commitments

atelier.Dépose(p) -> sometimef(<= 1 day) Déstocke(p)

Responsibility

F(Produit(t) / atelier.occupé)

A.5. SPÉCIFICATION DE L'ATELIER (PREMIER TEMPS)

Les spécifications présentées dans les trois sections qui suivent forment, par rapport à la spécification précédente, une étape de décomposition supplémentaire. En effet, l'atelier n'y est plus considéré comme une boîte noire mais comme un agent complexe composée des agents `Contrôleur`, `Magasin`, `Robot`, `Brideur`, `Véhicule` et `Machine-outil`.

A.5.1. HYPOTHÈSES

Rappelons que, dans un premier temps, le spécifieur se simplifie la tâche en posant les hypothèses d'omniscience et de fiabilité.

Il s'agit de deux hypothèses fortes. L'hypothèse d'omniscience nécessiterait, par exemple, d'équiper le magasin de capteurs afin de "voir" le robot, ce qui paraît fort peu réaliste. Quant à l'hypothèse de fiabilité, elle signifie, par exemple, qu'on ne tient pas compte du cas où le véhicule rencontre un obstacle sur sa route.

Ces deux hypothèses seront levées successivement aux sections A.6 et A.7.

A.5.2. CLÉS DE LECTURE

Deux mots d'explication pour faciliter la lecture des spécifications :

a) Organisation générale de l'atelier

L'atelier fonctionne de manière centralisée puisque c'est le contrôleur qui coordonne les différents postes de production. Il séquence les différentes opérations d'une commande de production et envoie les ordres aux agents concernés. (Un ordre est une action de la forme `Lance_`*opération* ; quand l'agent concerné a achevé l'opération, il produit l'action `Finis_`*opération* correspondante.)

Toute commande commence par une préparation (transport d'un cylindre jusqu'au brideur, bridage si nécessaire) et se termine par une terminaison (même opération en sens inverse).

Pour gérer le séquencement des opérations, le contrôleur doit tenir compte du statut des différents agents (prêt, occupé, actif,...).

b) Statuts des agents

Chaque agent (excepté le magasin, qui a un rôle passif) possède un attribut `Statut` :
- le contrôleur est occupé entre le moment où il reçoit une commande et le moment où il réserve un cylindre auprès du magasin ; il est prêt sinon ;

- le robot est occupé lorsqu'il prépare ou termine une commande (transport d'une pièce du magasin vers le brideur ou vice-versa) ; il est occupé sinon ;
- le brideur est dit "vide" lorsque son E/S est vide et n'est pas réservée pour le robot, auquel cas son statut vaut "réservé" ; il est prêt lorsque son E/S contient une pièce, occupé pendant le (dé)bridage ; enfin, son statut vaut "fini" lorsque la pièce sur son E/S peut être enlevée ;
- le véhicule est actif lors d'un (dé)chargement et mobile lors d'un déplacement ; il est prêt sinon ;
- une machine-outil se comporte comme le brideur (sauf que son statut ne prend jamais la valeur "réservé").

En outre, le contrôleur gère le statut de l'atelier dans son ensemble. L'atelier est prêt dès qu'il reste un cylindre disponible (c'est-à-dire non réservé) dans le stock. Il est occupé dès qu'il traite au moins une commande. (Ces deux états "prêt" et "occupé" ne sont pas exclusifs.)

Contrôleur

Declarations

Attributes

```
Nb_commandes : INT
Statut : STATUT

Nb_cylindres_disp : INT <- Magasin
Statut : STATUT <- {Robot, Brideur, Véhicule, Machine_outil}
Lieu : LIEU, Nb_pièces : INT <- Véhicule
```

Derived Attributes

```
Atelier_prêt : BOOL -> Gestionnaire
      = (magasin.Nb_cylindres_disp > 0)
Atelier_occupé : BOOL -> Gestionnaire
      = (Nb_commandes > 0)
```

Actions

```
Lance_préparation(OPTION) -> {Robot, Brideur}
Lance_chargement(LIEU), Lance_mouvement(LIEU, LIEU),
      Lance_déchargement(LIEU)
      -> Véhicule
Lance_usinage(MACHINE_OUTIL, PROGRAMME) -> Machine_outil
Lance_bridage, Lance_débridage -> Brideur
Lance_terminaison -> {Robot, Brideur}

Produit(TYPE) <- Gestionnaire
Réserve_cylindre -> Magasin
Finis_terminaison <- Robot
```

Constraints

Init

```
Nb_commandes = 0
Statut = prêt
```

Effects of Actions

```
gest.Produit(t):
      Nb_commandes = Nb_commandes + 1
      Statut = occupé
Réserve_cylindre: Statut = prêt
robot.Finis_terminaison: Nb_commandes = Nb_commandes - 1
```

Commitments

```
gest.Produit(boulon) ->
      (Lance_préparation(hors_étau);
      Lance_transport(brideur, tour);
      Lance_usinage(tour, boulon);
      Lance_transport(tour, brideur);
      Lance_terminaison)
gest.Produit(rivet) ->
      (Lance_préparation(hors_étau);
      Lance_transport(brideur, tour);
      Lance_usinage(tour, rivet);
      Lance_transport(tour, brideur);
      Lance_terminaison)
gest.Produit(vis) ->
      (Lance_préparation(hors_étau);
      Lance_transport(brideur, tour);
      Lance_usinage(tour, vis_1);
      Lance_transport(tour, brideur);
      Lance_bridage;
      Lance_transport(brideur, fraiseuse);
      Lance_usinage(fraiseuse, vis_2);
      Lance_transport(fraiseuse, brideur);
      Lance_terminaison)
...

where process Lance_transport(o,d) =
      if d <> véhic.Lieu then Lance_mouvement(o1,o) endif;
      Lance_chargement(o);
      Lance_mouvement(o,d);
      Lance_déchargement(d)
```

Responsibility

```
O(Réserve_cylindre / Statut = occupé)
F(Lance_préparation(opt) /
      robot.Statut <> prêt or brideur.Statut <> vide)
F(Lance_bridage / brideur.Statut <> prêt)
F(Lance_chargement(l) /
      véhic.Lieu <> l or véhic.Statut <> prêt or l.Statut <> fini)
F(Lance_mouvement(o,d) / véhic.Lieu <> o or véhic.Statut <> prêt
      or véhic.Statut = prêt since véhic.Statut = mobile)
F(Lance_déchargement(l) /
      véhic.Lieu <> l or véhic.Statut <> prêt or l.Statut <> vide)
O(Lance_déchargement(véhic.Lieu) /
      véhic.Statut = prêt and véhic.Nb_pièces = 2)
```

```
F(Lance_usinage(m, pgm) / m.Statut <> prêt)
F(Lance_débridage / brideur.Statut <> prêt)
F(Lance_terminaison /
      robot.Statut <> prêt or brideur.Statut <> prêt)
```

Perception

```
X(gest.Produit(t) / Atelier_prêt and t <> cylindre)
```

Magasin

Declarations

Attributes

```
Stock : SET[PIECE] -> {Robot, Gestionnaire}
Capacité : INT
Nb_cylindres_disp : INT -> Contrôleur

Atelier_occupé : BOOL <- Contrôleur
```

Derived Attributes

```
Quantité : INT = Card(Stock)
Vide : BOOL -> Gestionnaire
      = (Quantité = 0)
Plein : BOOL -> {Gestionnaire, Robot}
      = (Quantité = Capacité)
```

Actions

```
Réserve_cylindre <- Contrôleur
Saisis_MP(PIECE), Dépose_PF(LIEU) <- Robot
Stocke(PIECE), Déstocke(PIECE) <- Gestionnaire
```

Constraints

State Behaviour

```
In(Stock, p) => sometimef(not In(Stock, p))
```

Init

```
Stock = {}
Capacité = 50
Nb_cylindres_disp = 0
```

Effects of Actions

```
contr.Réserve_cylindre: Nb_cylindres_disp = Nb_cylindres_disp - 1
robot.Saisis_MP(p): Stock = Remove(Stock, p)
robot.Dépose_PF(p): Stock = Add(Stock, p)
gest.Stocke(p):
      Stock = Add(Stock, p)
      Nb_cylindres_disp = Nb_cylindres_disp + 1
gest.Déstocke(p): Stock = Remove(Stock, p)
```

Perception

```
X(robot.Saisis_MP(p) / In(Stock, p))
X(robot.Dépose_PF(p) / not Plein)
X(gest.Stocke(p) /
      not contr.Atelier_occupé and not Plein and p <> rien and p.Type =
      cylindre)
X(gest.Déstocke(p) /
      not contr.Atelier_occupé and In(Stock, p) and p.Type <> cylindre)
```

Publicity

```
X(Stock.gest / not contr.Atelier_occupé )
```

Robot

Declarations

Attributes

```
Statut : STATUT -> Contrôleur
Situation : LIEU

Stock : SET[PIECE] <- Magasin
Plein : BOOL <- Magasin
Statut : STATUT, Contenu_ES : PIECE <- Brideur
```

Actions

```
Déplace(LIEU, LIEU)
Saisis_MP(PIECE), Dépose_PF(PIECE) -> Magasin
Dépose_MP(PIECE), Saisis_PF(PIECE) -> Brideur
```

```
Finis_préparation
Finis_terminaison -> Contrôleur

Lance_préparation(OPTION), Lance_terminaison <- Contrôleur
```

Constraints

Init

```
Statut = prêt
Situation = magasin
```

Effects of Actions

```
Déplace(o,d): Situation = d
contr.Lance_préparation(opt): Statut = occupé
Finis_préparation: Statut = prêt
contr.Lance_terminaison: Statut = occupé
Finis_terminaison: Statut = prêt
```

Commitments

```
contr.Lance_préparation(opt) -> sometimef(<= 2 min)
      (if Situation <> magasin then Déplace(o, magasin) endif;
      Saisis_MP(p);
      Déplace(magasin, brideur);
      Dépose_MP(p);
      Finis_préparation)
Lance_terminaison -> sometimef(<= 3 min)
      (if Situation <> brideur then Déplace(o, brideur) endif;
      Saisis_PF(p);
      Déplace(brideur, magasin);
      Dépose_PF(p);
      Finis_terminaison)
      with p = brideur.Contenu_ES
```

Responsibility

```
F(Saisis_MP(p) /
      Situation <> magasin or not In(magasin.Stock, p) or p.Type <>
      cylindre)
F(Dépose_MP(p) /
      Situation <> brideur or brideur.Statut <> réservé)
F(Saisis_PF(p) / Situation <> brideur or brideur.Statut <> fini)
F(Dépose_PF(p) / Situation <> magasin or magasin.Plein)
```

Perception

```
X(contr.Lance_préparation(opt) / Statut = prêt)
X(contr.Lance_terminaison / Statut = prêt)
```

Brideur

Declarations

Attributes

```
Statut : STATUT -> {Contrôleur, Robot}
Contenu_ES : PIECE -> {Robot, Véhicule}
```

Actions

```
Permet_bridage, Serre, Désserre
Finis_bridage, Finis_débridage
Finis_préparation, Finis_terminaison

Lance_préparation(OPTION), Lance_terminaison <- Contrôleur
Dépose_MP(PIECE), Saisis_PF(PIECE) <- Robot
Charge(PIECE), Décharge(PIECE) <- Véhicule
Lance_bridage, Lance_débridage <- Contrôleur
```

Constraints

Init

```
Statut = vide
Contenu_ES = rien
```

Effects of Actions

```
contr.Lance_préparation(opt): Statut = réservé
robot.Dépose_MP(p):
      Contenu_ES = p
      Statut = prêt
Permet_bridage: Statut = occupé
Finis_préparation: Statut = fini
véhic.Charge(p):
      Contenu_ES = rien
      Statut = vide
véhic.Décharge(p):
      Contenu_ES = p
```

```
      Statut = prêt

contr.Lance_bridage: Statut = occupé
Serre: Contenu_ES.En_étau = true
Finis_bridage: Statut = fini
contr.Lance_débridage: Statut = occupé
Déserre: Contenu_ES.En_étau = false
Finis_débridage: Statut = fini
contr.Lance_terminaison:
      (Contenu_ES.En_étau) => Statut = occupé
      (not Contenu_ES.En_étau) => Statut = fini
robot.Saisis_PF(p):
      Contenu_ES = rien
      Statut = vide
```

Commitments

```
contr.Lance_préparation(opt) -> sometimef(<= 3 min)
      (if opt = en_étau then Permet_bridage; Serre endif;
      Finis_préparation)
contr.Lance_bridage -> sometimef(<= 1 min)
      (Serre;
      Finis_bridage)
contr.Lance_débridage -> sometimef(<= 1 min)
      (Désserre;
      Finis_débridage)
contr.Lance_terminaison -> sometimef(<= 1 min)
      (if Contenu_ES.En_étau then Désserre; Finis_débridage endif;
      Finis_terminaison)
```

Responsibility

```
F(Permet_bridage/ Statut <> prêt)
F(Serre / Statut <> occupé)
F(Désserre / Statut <> occupé)
```

Perception

```
X(robot.Dépose_MP(p) / Statut = réservé)
X(véhic.Charge(p) / Statut = fini)
X(véhic.Décharge(p) / Statut = vide)
X(contr.Lance_bridage / Statut = prêt)
X(contr.Lance_débridage / Statut = prêt)
X(robot.Saisis_PF(p) / Statut = fini)
```

Véhicule

Declarations

Attributes

```
Statut : STATUT -> Contrôleur
Lieu : LIEU -> Contrôleur
Accès : SET[LIEU]
Contenu_P1, Contenu_P2 : PIECE

Contenu_ES : PIECE <- {Brideur, Machine_outil}
```

Derived Attributes

```
Nb_pièces : INT -> Contrôleur
      (Contenu_P1 = rien and Contenu_P2 = rien) => Nb_pièces = 0
      (Contenu_P1 = rien xor Contenu_P2 = rien) => Nb_pièces = 1
      (Contenu_P1 <> rien and Contenu_P2 <> rien) => Nb_pièces = 2
```

Actions

```
Charge(PIECE) -> {Brideur, Machine_outil}
Bouge(LIEU, LIEU)
Décharge(PIECE) -> {Brideur, Machine_outil}
Finis_chargement(LIEU), Finis_mouvement(LIEU, LIEU),
      Finis_déchargement(LIEU)

Lance_chargement(LIEU), Lance_mouvement(LIEU, LIEU),
      Lance_déchargement(LIEU)
      <- Contrôleur
```

Constraints

Init

```
Statut = prêt
Accès = {brideur, fraiseuse, tour}
Contenu_P1 = rien, Contenu_P2 = rien
```

Effects of Actions

```
contr.Lance_chargement(l): Statut = actif
Charge(p):
      (Nb_pièces = 0) => Contenu_P1 = p
      (Nb_pièces = 1) => Contenu_P2 = p
Finis_chargement(l): Statut = prêt
contr.Lance_mouvement(o,d): Statut = mobile
```

```
Bouge(o,d): Lieu = d
Finis_mouvement(o,d): Statut = prêt
contr.Lance_déchargement(l): Statut = actif
Décharge(l):
      Contenu_P2 = rien
      Contenu_P1 = Contenu_P2
Finis_déchargement(l): Satut = prêt
```

Commitments

```
contr.Lance_chargement(l) -> sometimef(<= 10 sec)
      (Charge(p);
      Finis_chargement(l))
contr.Lance_mouvement(o,d) -> sometimef(<= 40 sec)
      (Bouge(o,d);
      Finis_mouvement(o,d))
contr.Lance_déchargement(l) -> sometimef(<= 10 sec)
      (Décharge(p);
      Finis_déchargement(l))
```

Responsibility

```
F(Charge(p) / Lieu.Contenu_ES = rien or p <> Lieu.Contenu_ES)
F(Décharge(p) / p <> Contenu_P1)
```

Perception

```
X(contr.Lance_chargement(l) / Statut = prêt and Lieu = l and Nb_pièces <>
      2)
X(contr.Lance_mouvement(o,d) / Statut = prêt and Lieu = o and In(Accès, d))
X(contr.Lance_déchargement(l) /
      Statut = prêt and Lieu = l and l.Contenu_ES = rien)
```

Publicity

```
X(Charge(p).l / Lieu = l)
X(Décharge(p).l / Lieu = l)
```

Machine_outil

Declarations

Attributes

```
Statut : STATUT -> Contrôleur
Programmes : SET[PROGRAMME]
Contenu_ES : PIECE -> Véhicule
```

Actions

```
Produit_rivet, Produit_filet, Produit_pas, Produit_rainure
Finis_usinage(MACHINE_OUTIL, PROGRAMME)

Charge(PIECE), Décharge(PIECE) <- Véhicule
Lance_usinage(MACHINE_OUTIL, PROGRAMME) <- Contrôleur
```

Constraints

Init

```
Statut = vide
(self = fraiseuse) => Programmes = {vis_2}
(self = tour) => Programmes = {boulon, rivet, vis_1}
Contenu_ES = rien
```

Effects of Actions

```
véhic.Décharge(p):
      Contenu_ES = p
      Statut = prêt
contr.Lance_usinage(m, pgm): Statut = occupé
Finis_usinage(m, pgm):
      Statut = fini
      Contenu_ES.Type = pgm
véhic.Charge(p):
      Contenu_ES = rien
      Statut = vide
```

Commitments

```
contr.Lance_usinage(tour, boulon) -> sometimef(<= 4 min)
      (Produit_rivet;
      Produit_filet;
      Finis_usinage(tour, boulon))
contr.Lance_usinage(tour, rivet) -> sometimef(<= 2 min)
      (Produit_rivet;
      Finis_usinage(tour, rivet))
contr.Lance_usinage(tour, vis_1) -> sometimef(<= 2 min)
      (Produit_pas;
      Finis_usinage(tour, vis_1))
```

```
contr.Lance_usinage(fraiseuse, vis_2) -> sometimef(<= 2 min)
      (Produit_rainure;
      Finis_usinage(fraiseuse, vis_2))
```

Perception

```
X(véhic.Décharge(p) / Statut = vide)
X(contr.Lance_usinage(m, pgm) /
      Statut = prêt and m = self and In(Programmes, pgm))
X(véhic.Charge(p) / Statut = fini)
```

A.6. SPÉCIFICATION DE L'ATELIER (DEUXIÈME TEMPS)

A.6.1. HYPOTHÈSES

Les spécifications qui suivent lèvent l'hypothèse d'omniscience des agents. La visibilité n'est plus que partielle et reflète la configuration physique de l'atelier (capteurs, senseurs, câbles,...).

Il semble que l'hypothèse d'omniscience doive être levée avant celle de fiabilité. En effet, certains aspects de fiabilité sont liés à la représentation qu'un agent se fait d'un autre agent qu'il ne voit pas ; en d'autres termes, la manière de traiter la visibilité a des incidences sur la fiabilité.

A.6.2. CLÉS DE LECTURE

• Le contrôleur n'a plus aucune visibilité de l'état des autres agents. Les seules échanges d'information prennent la forme d'actions (qui matérialisent, disons, des signaux électriques à travers un réseau de câbles entre le contrôleur et les agents).

Grâce à ces échanges d'information et à sa propre connaissance du système, le contrôleur est capable de se représenter une "image" des autres agents. (Voir l'ajout des attributs représentant le nombre de cylindres disponibles (non-réservés) dans le stock, le statut de chaque agent, le lieu du véhicule et le nombre de pièces qu'il porte.)

• La visibilité doit refléter les situations réelles sur le terrain. Prenons l'exemple du véhicule. Il semble logique qu'il ait une certaine perception des machines auprès desquelles il effectue ses chargements et déchargements. Ainsi, la clause de perception

```
X(l.Contenu_ES / Lieu = l)
```

exprime le fait que le véhicule ne perçoit le contenu de l'entrée-sortie d'une machine que s'il se trouve en face de cette machine. La contrainte

```
F(Charge(p) / Lieu.Contenu_ES = null or p <> Lieu.Contenu_ES)
```

signifie en outre que le véhicule est doué d'une certaine "intelligence" lui permettant de vérifier l'absence ou la présence d'une pièce sur une entrée-sortie.

• Par contre, nous avons jugé que le brideur n'était pas capable, lors d'une opération de terminaison, de tester la présence ou non d'un étau autour de la pièce qui se trouve sur son entrée-sortie (dans un cas, il doit débrider la pièce, dans la cas contraire, il peut la laisser immédiatement au robot). C'est désormais le contrôleur qui doit donner cette information, au moment de lancer l'ordre de terminaison, en ajoutant un argument de type `OPTION` à l'action `Lance_terminaison`.

Contrôleur

Declarations

Attributes

```
Nb_commandes : INT
Nb_cylindres_disp : INT
Statuts : TAB(LIEU, STATUT)
Lieu-véhic : LIEU, Nb_pièces_véhic : INT
```

Derived Attributes

```
Atelier_prêt : BOOL -> Gestionnaire
      = (magasin.Nb_cylindres_disp > 0)
Atelier_occupé : BOOL -> Gestionnaire
      = (Nb_commandes > 0)
```

Actions

```
Lance_préparation(OPTION) -> {Robot, Brideur}
Lance_chargement(LIEU), Lance_mouvement(LIEU, LIEU),
      Lance_déchargement(LIEU)
      -> Véhicule
Lance_usinage(MACHINE_OUTIL, PROGRAMME) -> Machine_outil
Lance_bridage, Lance_débridage -> Brideur
Lance_terminaison(OPTION) -> {Robot, Brideur}
Permet_saisie_PF -> Robot

Stocke(PIECE) <- Gestionnaire
Produit(TYPE) <- Gestionnaire
Finis_préparation <- {Robot, Brideur}
Finis_chargement(LIEU), Finis_mouvement(LIEU, LIEU),
      Finis_déchargement(LIEU)
      <- Véhicule
Finis_usinage(MACHINE_OUTIL, PROGRAMME) <- Machine_outil
Finis_bridage <- Brideur
Finis_débridage <- Brideur
Finis_terminaison <- {Brideur, Robot}
```

Constraints

Init

```
Nb_commandes = 0
Statuts[robot] = prêt
Statuts[brideur] = vide
Statuts[véhic] = prêt
Statuts[fraiseuse] = vide
Statuts[tour] = vide
Lieu_véhic = brideur
Nb_pièces_véhic = 0
Nb_cylindres_disp = 0
```

Effects of Actions

```
gest.Stocke(p): Nb_cylindres_disp = Nb_cylindres_disp + 1
gest.Produit(t): Nb_commandes = Nb_commandes + 1
Lance_préparation(opt):
      Statuts[robot] = occupé
      Statuts[brideur] = réservé
robot.Finis_préparation: Statuts[robot] = occupé
brideur.Finis_préparation: Statuts[brideur] = fini
Lance_chargement(l): Statuts[robot] = actif
véhic.Finis_chargement(l):
      Statuts[véhic] = prêt
      Statuts[l] = vide
      Nb_pièces_véhic = Nb_pièces_véhic + 1
Lance_mouvement(o,d): Statuts[véhic] = mobile
véhic.Finis_mouvement(o,d):
      Statuts[véhic] = prêt
      Lieu_véhic = d
Lance_déchargement(l): Statuts[véhic] = actif
véhic.Finis_déchargement(l):
      Statut[véhic] = prêt
      Statuts[l] = prêt
      Nb_pièces_véhic = Nb_pièces_véhic - 1
Lance_usinage(m, pgm): Statuts[m] = occupé
m.Finis_usinage(m,pgm): Statuts[fraiseuse] = fini
Lance_bridage: Statuts[brideur] = occupé
brideur.Finis_bridage: Statuts[brideur] = fini
Lance_débridage: Statuts[brideur] = occupé
brideur.Finis_débridage: Statuts[brideur] = fini
Lance_terminaison(opt):
      (opt = en_étau) => Statuts[brideur] = occupé
      (opt = hors_étau) => Statuts[brideur] = fini
      Statuts[robot] = occupé
brideur.Finis_terminaison: Statuts[brideur] = vide
robot.Finis_terminaison:
      Statuts[robot] = prêt
      Nb_commandes = Nb_commandes - 1
```

Commitments

```
gest.Produit(boulon) ->
      (Lance_préparation(hors_étau);
      Lance_transport(brideur, tour);
      Lance_usinage(tour, boulon);
      Lance_transport(tour, brideur);
      Lance_terminaison(hors_étau))
gest.Produit(rivet) ->
      (Lance_préparation(hors_étau);
      Lance_transport(brideur, tour);
      Lance_usinage(tour, rivet);
      Lance_transport(tour, brideur);
      Lance_terminaison(hors_étau))
gest.Produit(vis) ->
      (Lance_préparation(hors_étau);
      Lance_transport(brideur, tour);
      Lance_usinage(tour, vis_1);
      Lance_transport(tour, brideur);
      Lance_bridage;
      Lance_transport(brideur, fraiseuse);
      Lance_usinage(fraiseuse, vis_2);
      Lance_transport(fraiseuse, brideur);
      Lance_terminaison(hors_étau))
...

where process Lance_transport(o,d) =
      if d <> véhic.Lieu then Lance_mouvement(o1,o) endif;
      Lance_chargement(o);
      Lance_mouvement(o,d);
      Lance_déchargement(d)
```

Responsibility

```
F(Lance_préparation(opt) /
      Statuts[robot] <> prêt or Statuts[brideur] <> vide)
F(Lance_bridage / Statuts[brideur] <> prêt)
F(Lance_chargement(l) /
      Lieu-véhic <> l or Statuts[véhic] <> prêt
      or Statuts[l] <> fini)
F(Lance_mouvement(o,d) / Lieu_véhic <> o or Statuts[véhic] <> prêt
      or Statuts[véhic] = prêt since Statuts[véhic] = mobile)
F(Lance_déchargement(l) /
      Lieu_véhic <> l or Statut[véhic] <> prêt
      or Statuts[l] <> vide)
O(Lance_déchargement(Lieu_véhic) / Statuts[véhic] = prêt and
      Nb_pièces[véhic] = 2)
F(Lance_usinage(m, pgm) / Statuts[m] <> prêt)
F(Lance_débridage / Statut[brideur] <> prêt)
F(Lance_terminaison(opt) /
      Statuts[robot] <> prêt or Statuts[brideur] <> prêt)
F(Permet_saisie_PF / Statuts[brideur] <> fini)
```

Perception

```
X(gest.Stocke(p) / not Atelier_occupé)
X(gest.Produit(t) / Atelier_prêt and t <> cylindre)
```

Magasin

Declarations

Attributes

```
Stock : SET[PIECE] -> {Robot, Gestionnaire}
Capacité : INT
Nb_cylindres : INT
```

Derived Attributes

```
Quantité : INT = Card(Stock)
Vide : BOOL -> Gestionnaire
      = (Quantité = 0)
Plein : BOOL -> {Gestionnaire, Robot}
      = (Quantité = Capacité)
```

Actions

```
Saisis_MP(PIECE), Dépose_PF(LIEU) <- Robot
Stocke(PIECE), Déstocke(PIECE) <- Gestionnaire
```

Constraints

State Behaviour

```
In(Stock, p) => sometimef(not In(Stock, p))
```

Init

```
Stock = {}
Capacité = 50
Nb_cylindres = 0
```

Effects of Actions

```
robot.Saisis_MP(p):
      Stock = Remove(Stock, p)
      Nb_cylindres = Nb_cylindres - 1
robot.Dépose_PF(p): Stock = Add(Stock, p)
gest.Stocke(p):
      Stock = Add(Stock, p)
      Nb_cylindres = Nb_cylindres + 1
gest.Déstocke(p): Stock = Remove(Stock, p)
```

Perception

```
X(robot.Saisis_MP(p) / In(Stock, p))
X(robot.Dépose_PF(p) / not Plein)
X(gest.Stocke(p) / not Plein and p <> rien and p.Type = cylindre)
X(gest.Déstocke(p) / In(Stock, p) and p.Type <> cylindre)
```

Robot

Declarations

Attributes

```
Statut : STATUT
Situation : LIEU
Saisie_PF_permise : BOOL

Stock : SET[PIECE] <- Magasin
Plein : BOOL <- Magasin
Contenu_ES : PIECE <- Brideur
```

Actions

```
Déplace(LIEU, LIEU)
Saisis_MP(PIECE), Dépose_PF(PIECE) -> Magasin
Dépose_MP(PIECE), Saisis_PF(PIECE) -> Brideur

Finis_préparation, Finis_terminaison -> Contrôleur

Lance_préparation(OPTION), Lance_terminaison(OPT) <- Contrôleur
Permet_saisie_PF <- Contrôleur
```

Constraints

Init

```
Statut = prêt
Situation = magasin
```

Effects of Actions

```
Déplace(o,d): Situation = d
contr.Lance_préparation(opt): Statut = occupé
Finis_préparation: Statut = prêt
contr.Lance_terminaison(opt): Statut = occupé
contr.Permet_saisie_PF: Saisie_PF_permise = true
Saisis_PF(p): Saisie_PF_permise = false
Finis_terminaison: Statut = prêt
```

Commitments

```
contr.Lance_préparation(opt) -> sometimef(<= 2 min)
      (if Situation <> magasin then Déplace(o, magasin) endif;
      Saisis_MP(p);
      Déplace(magasin, brideur);
      Dépose_MP(p);
      Finis_préparation)
Lance_terminaison(opt) -> sometimef(<= 3 min)
      (if Situation <> brideur then Déplace(o, brideur) endif;
      Saisis_PF(p);
      Déplace(brideur, magasin);
      Dépose_PF(p);
      Finis_terminaison)
      with p = brideur.Contenu_ES
```

Responsibility

```
F(Saisis_MP(p) /
      Situation <> magasin or not In(magasin.Stock, p) or p.Type <>
      cylindre)
F(Dépose_MP(p) /
      Situation <> brideur or brideur.Contenu_ES = rien)
F(Saisis_PF(p) /
      Situation <> brideur or Saisie_PF_permise)
F(Dépose_PF(p) / Situation <> magasin or magasin.Plein)
```

Perception

```
X(magasin.Stock / Situation = magasin)
X(brideur.Contenu_ES / Situation = brideur)
X(contr.Lance_préparation(opt) / Statut = prêt)
```

```
X(contr.Lance_terminaison(opt) / Statut = prêt)
X(contr.Permet_saisie_PF / Statut = occupé)
```

Brideur

Declarations

Attributes

```
Statut : STATUT
Contenu_ES : PIECE -> {Robot, Véhicule}
```

Actions

```
Permet_bridage, Serre, Désserre
Finis_bridage, Finis_débridage -> Contrôleur
Finis_préparation, Finis_terminaison -> Contrôleur

Lance_préparation(OPTION), Lance_terminaison(OPTION) <- Contrôleur
Dépose_MP(PIECE), Saisis_PF(PIECE) <- Robot
Charge(PIECE), Décharge(PIECE) <- Véhicule
Lance_bridage, Lance_débridage <- Contrôleur
```

Constraints

Init

```
Statut = vide
Contenu_ES = rien
```

Effects of Actions

```
contr.Lance_préparation(opt): Statut = réservé
robot.Dépose_MP(p):
      Contenu_ES = p
      Statut = prêt
Permet_bridage: Statut = occupé
Finis_préparation: Statut = fini
véhic.Charge(p):
      Contenu_ES = rien
      Statut = vide
véhic.Décharge(p):
      Contenu_ES = p
      Statut = prêt

contr.Lance_bridage: Statut = occupé
Serre: Contenu_ES.En_étau = true
Finis_bridage: Statut = fini
contr.Lance_débridage: Statut = occupé
Déserre: Contenu_ES.En_étau = false
Finis_débridage: Statut = fini
contr.Lance_terminaison(opt):
      (opt = en_étau) => Statut = occupé
      (opt = hors_étau) => Statut = fini
robot.Saisis_PF(p):
      Contenu_ES = rien
      Statut = vide
```

Commitments

```
contr.Lance_préparation(opt) -> sometimef(<= 3 min)
      (if opt = en_étau then Permet_bridage; Serre endif;
      Finis_préparation)
contr.Lance_bridage -> sometimef(<= 1 min)
      (Serre;
      Finis_bridage)
contr.Lance_débridage -> sometimef(<= 1 min)
      (Désserre;
      Finis_débridage)
contr.Lance_terminaison(opt) -> sometimef(<= 1 min)
      (if opt = en_étau then Désserre; Finis_débridage endif;
      Finis_terminaison)
```

Responsibility

```
F(Permet_bridage/ Statut <> prêt)
F(Serre / Statut <> occupé)
F(Désserre / Statut <> occupé)
```

Perception

```
X(robot.Dépose_MP(p) / Statut = réservé)
X(véhic.Charge(p) / Statut = fini)
X(véhic.Décharge(p) / Statut = vide)
X(contr.Lance_bridage / Statut = prêt)
X(contr.Lance_débridage / Statut = prêt)
X(robot.Saisis_PF(p) / Statut = fini)
```

Véhicule

Declarations

Attributes

```
Statut : STATUT
Lieu : LIEU
Accès : SET[LIEU]
Contenu_P1, Contenu_P2 : PIECE

Contenu_ES : PIECE <- {Brideur, Machine_outil}
```

Derived Attributes

```
Nb_pièces : INT
      (Contenu_P1 = rien and Contenu_P2 = rien) => Nb_pièces = 0
      (Contenu_P1 = rien xor Contenu_P2 = rien) => Nb_pièces = 1
      (Contenu_P1 <> rien and Contenu_P2 <> rien) => Nb_pièces = 2
```

Actions

```
Charge(PIECE) -> {Brideur, Machine_outil}
Bouge(LIEU, LIEU)
Décharge(PIECE) -> {Brideur, Machine_outil}
Finis_chargement(LIEU), Finis_mouvement(LIEU, LIEU),
      Finis_déchargement(LIEU)
      -> Contrôleur

Lance_chargement(LIEU), Lance_mouvement(LIEU, LIEU),
      Lance_déchargement(LIEU)
      <- Contrôleur
```

Constraints

Init

```
Statut = prêt
Accès = {brideur, fraiseuse, tour}
Contenu_P1 = rien, Contenu_P2 = rien
```

Effects of Actions

```
contr.Lance_chargement(l): Statut = actif
Charge(p):
      (Nb_pièces = 0) => Contenu_P1 = p
      (Nb_pièces = 1) => Contenu_P2 = p
Finis_chargement(l): Statut = prêt
contr.Lance_mouvement(o,d): Statut = mobile
Bouge(o,d): Lieu = d
Finis_mouvement(o,d): Statut = prêt
contr.Lance_déchargement(l): Statut = actif
Décharge(l):
      Contenu_P2 = rien
      Contenu_P1 = Contenu_P2
Finis_déchargement(l): Satut = prêt
```

Commitments

```
contr.Lance_chargement(l) -> sometimef(<= 10 sec)
      (Charge(p);
      Finis_chargement(l))
contr.Lance_mouvement(o,d) -> sometimef(<= 40 sec)
      (Bouge(o,d);
      Finis_mouvement(o,d))
contr.Lance_déchargement(l) -> sometimef(<= 10 sec)
      (Décharge(p);
      Finis_déchargement(l))
```

Responsibility

```
F(Charge(p) / Lieu.Contenu_ES = rien or p <> Lieu.Contenu_ES)
F(Décharge(p) / p <> Contenu_P1)
```

Perception

```
X(l.Contenu_ES / Lieu = l)

X(contr.Lance_chargement(l) / Statut = prêt and Lieu = l and Nb_pièces
      2)
X(contr.Lance_mouvement(o,d) / Statut = prêt and Lieu = o and In(Accès,
X(contr.Lance_déchargement(l) /
      Statut = prêt and Lieu = l and l.Contenu_ES = rien)
```

Publicity

```
X(Charge(p).l / Lieu = l)
X(Décharge(p).l / Lieu = l)
```

Machine_outil

Declarations

Attributes

```
Statut : STATUT
Programmes : SET[PROGRAMME]
Contenu_ES : PIECE -> Véhicule
```

Actions

```
Produit_rivet, Produit_filet, Produit_pas, Produit_rainure
Finis_usinage(MACHINE_OUTIL, PROGRAMME) -> Contrôleur

Charge(PIECE), Décharge(PIECE) <- Véhicule
Lance_usinage(MACHINE_OUTIL, PROGRAMME) <- Contrôleur
```

Constraints

Init

```
Statut = vide
(self = fraiseuse) => Programmes = {vis_2}
(self = tour) => Programmes = {boulon, rivet, vis_1}
Contenu_ES = rien
```

Effects of Actions

```
véhic.Décharge(p):
      Contenu_ES = p
      Statut = prêt
contr.Lance_usinage(m, pgm): Statut = occupé
Finis_usinage(m, pgm):
      Statut = fini
      Contenu_ES.Type = pgm
véhic.Charge(p):
      Contenu_ES = rien
      Statut = vide
```

Commitments

```
contr.Lance_usinage(tour, boulon) -> sometimef(<= 4 min)
      (Produit_rivet;
      Produit_filet;
      Finis_usinage(tour, boulon))
contr.Lance_usinage(tour, rivet) -> sometimef(<= 2 min)
      (Produit_rivet;
      Finis_usinage(tour, rivet))
contr.Lance_usinage(tour, vis_1) -> sometimef(<= 2 min)
      (Produit_pas;
      Finis_usinage(tour, vis_1))
contr.Lance_usinage(fraiseuse, vis_2) -> sometimef(<= 2 min)
      (Produit_rainure;
      Finis_usinage(fraiseuse, vis_2))
```

Perception

```
X(véhic.Décharge(p) / Statut = vide)
X(contr.Lance_usinage(m, pgm) /
      Statut = prêt and m = self and In(Programmes, pgm))
X(véhic.Charge(p) / Statut = fini)
```

A.7. SPÉCIFICATION DE L'ATELIER (TROISIÈME TEMPS)

A.7.1. HYPOTHÈSES

Les spécifications qui suivent lèvent l'hypothèse de fiabilité des agents. Elles constituent la dernière version des spécifications de l'atelier (du moins dans le cadre de notre étude de cas ; de façon plus réaliste, on pourrait pousser la décomposition plus loin et identifier, dans le véhicule par exemple, des agents comme la batterie ou le palettiseur).

A.7.2. CLÉS DE LECTURE

Deux types de problème concernant le véhicule ont été envisagés :
- le véhicule est alimentée par une batterie, qui peut éventuellement se décharger complètement ;
- un obstacle peut survenir sur la route du véhicule.

Dans les deux cas, le véhicule n'est plus en mesure de remplir normalement ses engagements ; une occurrence de l'action `Echoue` a lieu et est exportée vers le contrôleur, qui interrompt la production, et vers le gestionnaire.

Pour simplifier, on suppose que, à ce moment, le gestionnaire intervient. Il résout le problème (en rechargeant la batterie ou en éliminant l'obstacle), termine manuellement l'opération commencée par le véhicule et, lorsque tout est en ordre, en avertit le contrôleur. Celui-ci reprend la production comme si l'opération s'était déroulée normalement.

Contrôleur

Declarations

Attributes

```
Atelier_arrêté : BOOL -> Gestionnaire
Nb_commandes : INT
Nb_cylindres_disp : INT
Statuts : TAB(LIEU, STATUT)
Lieu-véhic : LIEU, Nb_pièces_véhic : INT
```

Derived Attributes

```
Atelier_prêt : BOOL -> Gestionnaire
      = (magasin.Nb_cylindres_disp > 0 and not Atelier_arrêté)
Atelier_occupé : BOOL -> Gestionnaire
      = (Nb_commandes > 0)
```

Actions

```
Lance_préparation(OPTION) -> {Robot, Brideur}
Lance_chargement(LIEU), Lance_mouvement(LIEU, LIEU),
      Lance_déchargement(LIEU)
      -> Véhicule
Lance_usinage(MACHINE_OUTIL, PROGRAMME) -> Machine_outil
Lance_bridage, Lance_débridage -> Brideur
Lance_terminaison(OPTION) -> {Robot, Brideur}
Permet_saisie_PF -> Robot

Stocke(PIECE) <- Gestionnaire
Produit(TYPE) <- Gestionnaire
Finis_préparation <- {Robot, Brideur}
Finis_chargement(LIEU), Finis_mouvement(LIEU, LIEU),
      Finis_déchargement(LIEU)
      <- Véhicule
Finis_usinage(MACHINE_OUTIL, PROGRAMME) <- Machine_outil
Finis_bridage <- Brideur
Finis_débridage <- Brideur
Echoue <- {Robot, Brideur, Véhicule, Machine_outil}
Règle_probl_véhic <-Gestionnaire
Reprends_production <- Gestionnaire
Finis_terminaison <- {Brideur, Robot}
```

Constraints

Init

```
Atelier_arrêté = false
Nb_commandes = 0
Statuts[robot] = prêt
Statuts[brideur] = vide
Statuts[véhic] = prêt
Statuts[fraiseuse] = vide
Statuts[tour] = vide
Lieu_véhic = brideur
Nb_pièces_véhic = 0
Nb_cylindres_disp = 0
```

Effects of Actions

```
gest.Stocke(p): Nb_cylindres_disp = Nb_cylindres_disp + 1
gest.Produit(t): Nb_commandes = Nb_commandes + 1
Lance_préparation(opt):
      Statuts[robot] = occupé
      Statuts[brideur] = réservé
robot.Finis_préparation: Statuts[robot] = occupé
brideur.Finis_préparation: Statuts[brideur] = fini
Lance_chargement(l): Statuts[véhic] = actif
véhic.Finis_chargement(l):
      Statuts[véhic] = prêt
      Statuts[l] = vide
      Nb_pièces_véhic = Nb_pièces_véhic + 1
Lance_mouvement(o,d): Statuts[véhic] = mobile
véhic.Finis_mouvement(o,d):
      Statuts[véhic] = prêt
      Lieu_véhic = d
Lance_déchargement(l): Statuts[véhic] = actif
véhic.Finis_déchargement(l):
      Statuts[véhic] = prêt
      Statuts[l] = prêt
      Nb_pièces_véhic = Nb_pièces_véhic - 1
Lance_usinage(m, pgm): Statuts[m] = occupé
m.Finis_usinage(m,pgm): Statuts[fraiseuse] = fini
Lance_bridage: Statuts[brideur] = occupé
brideur.Finis_bridage: Statuts[brideur] = fini
Lance_débridage: Statuts[brideur] = occupé
brideur.Finis_débridage: Statuts[brideur] = fini
l.Echoue:
      Statuts[l] = indisponible
      Atelier_arrêté = true
gest.Règle_probl_véhic(l, p1, p2):
      Statuts[véhic] = prêt
      Lieu_véhic = l
      (p1 = rien and p2 = rien) => Nb_pièces_véhic = 0
```

```
      (p1 = rien and p2 <> rien) => Nb_pièces_véhic = 1
      (p1 <> rien and p2 = rien) => Nb_pièces_véhic = 1
      (p1 <> rien and p2 <> rien) => Nb_pièces_véhic = 2
gest.Reprends_production: Atelier_arrêté = false
Lance_terminaison(opt):
      (opt = en_étau) => Statuts[brideur] = occupé
      (opt = hors_étau) => Statuts[brideur] = fini
      Statuts[robot] = occupé
brideur.Finis_terminaison: Statuts[brideur] = vide
robot.Finis_terminaison:
      Statuts[robot] = prêt
      Nb_commandes = Nb_commandes - 1
```

Commitments

```
gest.Produit(boulon) ->
      (Lance_préparation(hors_étau);
      Lance_transport(brideur, tour);
      Lance_usinage(tour, boulon);
      Lance_transport(tour, brideur);
      Lance_terminaison(hors_étau))
gest.Produit(rivet) ->
      (Lance_préparation(hors_étau);
      Lance_transport(brideur, tour);
      Lance_usinage(tour, rivet);
      Lance_transport(tour, brideur);
      Lance_terminaison(hors_étau))
gest.Produit(vis) ->
      (Lance_préparation(hors_étau);
      Lance_transport(brideur, tour);
      Lance_usinage(tour, vis_1);
      Lance_transport(tour, brideur);
      Lance_bridage;
      Lance_transport(brideur, fraiseuse);
      Lance_usinage(fraiseuse, vis_2);
      Lance_transport(fraiseuse, brideur);
      Lance_terminaison(hors_étau))
...

where process Lance_transport(o,d) =
      if d <> véhic.Lieu then Lance_mouvement(o1,o) endif;
      Lance_chargement(o);
      Lance_mouvement(o,d);
      Lance_déchargement(d)
```

Responsibility

```
F(Lance_préparation(opt) /
      Statuts[robot] <> prêt or Statuts[brideur] <> vide)
F(Lance_bridage / Statuts[brideur] <> prêt)
F(Lance_chargement(l) /
      Lieu-véhic <> l or Statuts[véhic] <> prêt
      or Statuts[l] <> fini)
F(Lance_mouvement(o,d) / Lieu_véhic <> o or Statuts[véhic] <> prêt
      or Statuts[véhic] = prêt since Statuts[véhic] = mobile)
F(Lance_déchargement(l) /
      Lieu_véhic <> l or Statut[véhic] <> prêt
      or Statuts[l] <> vide)
O(Lance_déchargement(Lieu_véhic) / Statuts[véhic] = prêt and
      Nb_pièces[véhic] = 2)
F(Lance_usinage(m, pgm) / Statuts[m] <> prêt)
F(Lance_débridage / Statut[brideur] <> prêt)
F(Lance_terminaison(opt) /
      Statuts[robot] <> prêt or Statuts[brideur] <> prêt)
F(Permet_saisie_PF / Statuts[brideur] <> fini)
```

Perception

```
X(gest.Stocke(p) / not Atelier_occupé)
X(gest.Produit(t) / Atelier_prêt and t <> cylindre)

...
```

Véhicule

Declarations

Attributes

```
Statut : STATUT
Lieu : LIEU
Accès : SET[LIEU]
Contenu_P1, Contenu_P2 : PIECE
Taux_batterie : REAL

Contenu_ES : PIECE <- {Brideur, Machine_outil}
```

Derived Attributes

```
Nb_pièces : INT
      (Contenu_P1 = rien and Contenu_P2 = rien) => Nb_pièces = 0
      (Contenu_P1 = rien xor Contenu_P2 = rien) => Nb_pièces = 1
      (Contenu_P1 <> rien and Contenu_P2 <> rien) => Nb_pièces = 2
```

Actions

```
Charge(PIECE) -> {Brideur, Machine_outil}
Bouge(LIEU, LIEU)
Rencontre_obstacle
Décharge(PIECE) -> {Brideur, Machine_outil}
Echoue -> {Contrôleur, Gestionnaire}
Finis_chargement(LIEU), Finis_mouvement(LIEU, LIEU),
      Finis_déchargement(LIEU)
      -> Contrôleur
Regénère_batterie
Recharge_batterie
Finis_rech_batterie -> Gestionnaire

Lance_chargement(LIEU), Lance_mouvement(LIEU, LIEU),
      Lance_déchargement(LIEU)
      <- Contrôleur
Lance_rech_batterie <- Gestionnaire
Règle_probl_véhic(LIEU, PIECE, PIECE) <- Gestionnaire
```

Constraints

State Behaviour

```
Taux_batterie > 0
always_p(<= 10 min) Statut = prêt => Taux_batterie = 100
```

Init

```
Lieu = brideur
Accès = {brideur, fraiseuse, tour}
Contenu_P1 = rien, Contenu_P2 = rien
```

Effects of Actions

```
contr.Lance_chargement(l): Statut = actif
Charge(p):
      (Nb_pièces = 0) => Contenu_P1 = p
      (Nb_pièces = 1) => Contenu_P2 = p
      Taux_batterie = Taux_batterie - 0,1
Finis_chargement(l): Statut = prêt
contr.Lance_mouvement(o,d): Statut = mobile
Bouge(o,d):
      Lieu = d
      Taux_batterie = Taux_batterie - 0,5
Rencontre_obstacle: Statut = indisponible
Finis_mouvement(o,d): Statut = prêt
contr.Lance_déchargement(l): Statut = actif
Décharge(l):
      Contenu_P2 = rien
      Contenu_P1 = Contenu_P2
      Taux_batterie = Taux_batterie - 0,1
Finis_déchargement(l): Statut = prêt
Regénère_batterie: Taux_batterie = Max(100, Taux_batterie + 10)
gest.Lance_rech_batterie: Statut = indisponible
Recharge_batterie: Taux_batterie = 100
Finis_rech_batterie: Statut = prêt
Echoue: Statut = indisponible
gest.Règle_probl_véhic(l, p1, p2):
      Statut = prêt
      Lieu = l
      Contenu_P1 = p1
      Contenu_P2 = p2
```

Commitments

```
contr.Lance_chargement(l) -> sometime_f(<= 10 sec)
      ((Charge(p);
      Finis_chargement(l))
      xor Echoue)
contr.Lance_mouvement(o,d) -> sometime_f(<= 40 sec)
      ((Bouge(o,d);
      Finis_mouvement(o,d))
      xor Echoue)
contr.Lance_déchargement(l) -> sometime_f(<= 10 sec)
      ((Décharge(p);
      Finis_déchargement(l))
      xor Echoue)

gest.Lance_rech_batterie -> sometime_f
      (Recharge_batterie;
      Finis_rech_batterie)
```

Responsibility

```
F(Charge(p) / Lieu.Contenu_ES = rien or p <> Lieu.Contenu_ES or Statut
      indisponible)
F(Bouge(o,d) / Statut = indisponible)
F(Rencontre_obstacle / Statut <> mobile)
F(Décharge(p) / p <> Contenu_P1 or Statut = indisponible)
F(Regénère_batterie / Taux_batterie = 100)
```

Perception

```
X(l.Contenu_ES / Lieu = l)
X(contr.Lance_chargement(l) / Statut = prêt and Lieu = l and Nb_pièces
      2)
X(contr.Lance_mouvement(o,d) / Statut = prêt and Lieu = o and In(Accès,
```

```
X(contr.Lance_déchargement(l) /
      Statut = prêt and Lieu = l and l.Contenu_ES = rien)
X(gest.Lance_rech_batterie / Statut <> actif and Statut <> mobile)
X(gest.Règle_probl_véhic(l, p1, p2) / Statut = indisponible)
```

Publicity

```
X(Charge(p).l / Lieu = l)
X(Décharge(p).l / Lieu = l)
```

A.8. PROTOTYPE DE LA SPÉCIFICATION DU VÉHICULE (PREMIÈRE MÉTHODE)

Cette section illustre la première méthode de prototypage sur la dernière version de la spécification associée à l'agent `Véhicule`.

A.8.1. PHASE D'OPÉRATIONALISATION

Voir les spécifications ALBERT qui suivent.

A.8.2. PHASE DE TRADUCTION

Voir les spécifications OBLOG qui suivent.

Véhicule

Declarations

Attributes

```
Statut : STATUT -> all
Lieu : LIEU -> all
Accès : SET[LIEU] -> all
Contenu_P1, Contenu_P2 : PIECE -> all
Taux_batterie : REAL -> all
LTT_prêt : TIME
      /* Enregistre le dernier temps où Statut est devenu prêt */
PON_ch : TAB([1..3], INT), COT_ch : SEQ[TIME]
PON_mv : TAB([1..3], INT), COT_mv : SEQ[TIME]
PON_dé : TAB([1..3], INT), COT_dé : SEQ[TIME]
PON_re : TAB([1..2], INT)

Contenu_ES : PIECE <- {Brideur, Machine_outil}
Time : TIME <- Clock
```

Derived Attributes

```
Nb_pièces : INT -> all
      (Contenu_P1 = rien and Contenu_P2 = rien) => Nb_pièces = 0
      (Contenu_P1 = rien xor Contenu_P2 = rien) => Nb_pièces = 1
      (Contenu_P1 <> rien and Contenu_P2 <> rien) => Nb_pièces = 2
Alwayspb_prêt : BOOL -> all
      = (Statut = prêt) and (clock.Time - LTT_prêt >= 10 min)
Failed : BOOL -> all
      = not (not Alwayspb_prêt or Taux_batterie = 100)
      or (not Empty(COT_ch) and clock.Time - First(COT_ch) > 10 sec))
      or (not Empty(COT_mv) and clock.Time - First(COT_mv) > 40 sec))
      or (not Empty(COT_dé) and clock.Time - First(COT_dé) > 10 sec)
```

Actions

```
Live
Charge(PIECE) -> {Brideur, Machine_outil}
Bouge(LIEU, LIEU)
Rencontre_obstacle
Décharge(PIECE) -> {Brideur, Machine_outil}
Echoue -> {Contrôleur, Gestionnaire}
Finis_chargement(LIEU), Finis_mouvement(LIEU, LIEU),
      Finis_déchargement(LIEU)
      -> Contrôleur
Regénère_batterie
Recharge_batterie
Finis_rech_batterie -> Gestionnaire
Fail
Die

Lance_chargement(LIEU), Lance_mouvement(LIEU, LIEU),
      Lance_déchargement(LIEU)
      <- Contrôleur
Lance_rech_batterie <- Gestionnaire
Règle_probl_véhic(LIEU, PIECE, PIECE) <- Gestionnaire
```

Constraints

State Behaviour

```
Taux_batterie > 0
```

Effects of Actions

```
Live:
      Statut = prêt
      Accès = {brideur, fraiseuse, tour}
      Contenu_P1 = rien, Contenu_P2 = rien
      LTT_prêt = 0 min
      PON_ch[1] = 0, PON_ch[2] = 0, PON_ch[3] = 0, COT_ch = []
      PON_mv[1] = 0, PON_mv[2] = 0, PON_mv[3] = 0, COT_mv = []
      PON_dé[1] = 0, PON_dé[2] = 0, PON_dé[3] = 0, COT_dé = []
      PON_re[1] = 0, PON_re[2] = 0
contr.Lance_chargement(l):
      Statut = actif
      PON_ch[1] = PON_ch[1] + 1
      COT_ch = Append(COT_ch, clock.Time)
Charge(p):
      (Nb_pièces = 0) => Contenu_P1 = p
      (Nb_pièces = 1) => Contenu_P2 = p
      Taux_batterie = Taux_batterie - 0,1
      PON_ch[1] = PON_ch[1] - 1
      PON_ch[2] = PON_ch[2] + 1
      PON_ch[3] = PON_ch[3] + 1
Finis_chargement(l):
      Statut = prêt
      LTT_prêt = clock.Time
      PON_ch[2] = PON_ch[2] - 1
      PON_ch[3] = PON_ch[3] - 1
      COT_ch = Remove(COT_ch, First(COT_ch))
contr.Lance_mouvement(o,d):
      Statut = mobile
      PON_mv[1] = PON_mv[1] + 1
      COT_mv = Append(COT_mv, clock.Time)
```

```
Bouge(o,d):
      Lieu = d
      Taux_batterie = Taux_batterie - 0,5
      PON_mv[1] = PON_mv[1] - 1
      PON_mv[2] = PON_mv[2] + 1
      PON_mv[3] = PON_mv[3] + 1
Rencontre_obstacle: Statut = indisponible
Finis_mouvement(o,d):
      Statut = prêt
      LTT_prêt = clock.Time
      PON_mv[2] = PON_mv[2] - 1
      PON_mv[3] = PON_mv[3] - 1
      COT_mv = Remove(COT_mv, First(COT_mv))
contr.Lance_déchargement(l):
      Statut = actif
      PON_dé[1] = PON_dé[1] + 1
      COT_dé = Append(COT_dé, clock.Time)
Décharge(l):
      Contenu_P2 = rien
      Contenu_P1 = Contenu_P2
      Taux_batterie = Taux_batterie - 0,1
      PON_dé[1] = PON_dé[1] - 1
      PON_dé[2] = PON_dé[2] + 1
      PON_dé[3] = PON_dé[3] + 1
Finis_déchargement(l):
      Satut = prêt
      LTT_prêt = clock.Time
      PON_dé[2] = PON_dé[2] - 1
      PON_dé[3] = PON_dé[3] - 1
      COT_dé = Remove(COT_dé, First(COT_dé))
Regénère_batterie: Taux_batterie = Max(100, Taux_batterie + 10)
gest.Lance_rech_batterie:
      Statut = indisponible
      PON_re[1] = PON_re[1] + 1
Recharge_batterie:
      Taux_batterie = 100
      PON_re[1] = PON_re[1] - 1
      PON_re[2] = PON_re[2] + 1
Finis_rech_batterie:
      Statut = prêt
      LTT_prêt = clock.Time
      PON_re[2] = PON_re[2] - 1
Echoue:
      Statut = indisponible
      (PON_ch[3] > 0) => PON_ch[2] = PON_ch[2] - 1, PON_ch[3] = PON_ch[3] -
      1
      (PON_mv[3] > 0) => PON_mv[2] = PON_mv[2] - 1, PON_mv[3] = PON_mv[3] -
      1
      (PON_mv[3] > 0) => PON_dé[2] = PON_dé[2] - 1, PON_dé[3] = PON_dé[3] -
      1
gest.Règle_probl_véhic(l, p1, p2):
      Statut = prêt
      LTT_prêt = clock.Time
      Lieu = l
      Contenu_P1 = p1
      Contenu_P2 = p2
Fail
Die
```

Responsibility

```
F(Charge(p) / Lieu.Contenu_ES = rien or p <> Lieu.Contenu_ES or Statut
      indisponible or PON_ch[1] = 0 or Failed)
F(Finis_chargement(l) / PON_ch[2] = 0 or Failed)
F(Bouge(o,d) /
      Statut = indisponible or PON_mv[1] = 0 or Failed)
F(Finis_mouvement(o,d) / PON_mv[2] = 0 or Failed)
F(Rencontre_obstacle / Statut <> mobile or Failed)
F(Décharge(p) / p <> Contenu_P1 or Statut = indisponible or PON_dé[1] =
      or Failed)
F(Finis_déchargement(l) / PON_dé[2] = 0 or Failed)
F(Regénère_batterie / Taux_batterie = 100 or Failed)
F(Recharge_batterie / PON_re[1] = 0 or Failed)
F(Finis_rech_batterie / PON_re[2] = 0 or Failed)
F(Echoue / PON_ch[3] = 0 and PON_mv[3] = 0 and PON_dé[3] = 0 or Failed)
F(Fail / not (Failed))
F(Die / Empty(COT_ch) or Empty(COT_mv) or Empty(COT_dé)
      or (PON_re[1] > 0 or PON_re[2] > 0))
```

Perception

```
X(l.Contenu_ES / Lieu = l)
```

Publicity

```
X(Charge(p).l / Lieu = l and l.Statut = fini)
X(Décharge(p).l / Lieu = l and l.Statut = vide)
```

VEHICULE

INTERFACE

ATTRIBUTES

```
statut : STATUT
lieu : LIEU
accès : SET(LIEU)
contenuP1, contenuP2 : PIECE
tauxBatterie : real
nbPièces : int
alwayspb_prêt : bool
failed : bool
```

ACTIONS

```
lanceChargement(l : LIEU)
lanceMouvement(o,d : LIEU)
lanceDéchargement(o,d : LIEU)
lanceRechBatterie
règleProblVéhic(l : LIEU, p1 : PIECE, p2 : PIECE)
```

BODY

ATTRIBUTES

```
statut : STATUT
lieu : LIEU
accès : SET(LIEU)
contenuP1, contenuP2 : PIECE
tauxBatterie : real
      ! > 0
DER nbPièces : int
      {contenuP1 = rien AND contenuP2 = rien} = 0
      {contenuP1 = rien XOR contenuP2 = rien} = 1
      {contenuP1 <> rien AND contenuP2 <> rien} = 2
LTT_prêt : time
PON_ch : ARRAY([1..3], int), COT_ch : LIST[time]
PON_mv : ARRAY([1..3], int), COT_mv : LIST[time]
PON_dé : ARRAY([1..3], int), COT_dé : LIST[time]
PON_re : ARRAY([1..2], int)
DER alwayspb_prêt : bool
      = (statut = prêt) AND (clock.time - LTT_prêt >= 10 MIN)
DER failed : bool
      = NOT (NOT alwayspb_prêt OR tauxBatterie = 100)
      OR (NOT EMPTY(COT_ch) AND clock.time - FIRST(COT_ch) > 10 SEC))
      OR (NOT EMPTY(COT_mv) AND clock.time - FIRST(COT_mv) > 40 SEC))
      OR (NOT EMPTY(COT_dé) AND clock.time - FIRST(COT_dé) > 10 SEC)
```

ACTIONS

```
*! live
      => statut := prêt
      => accès := (brideur, fraiseuse, tour)
      => contenuP1 := rien
      => contenuP2 := rien
      => LTT_prêt := 0 MIN
      => PON_ch[1] := 0, PON_ch[2] := 0, PON_ch[3] := 0, COT_ch := ()
      => PON_mv[1] := 0, PON_mv[2] := 0, PON_mv[3] := 0, COT_mv := ()
      => PON_dé[1] := 0, PON_dé[2] := 0, PON_dé[3] := 0, COT_dé := ()
      => PON_re[1] := 0, PON_re[2] := 0
!charge(p : PIECE)
      {nbPièces = 0} => contenuP1 := p
      {nbPièces = 1} => contenuP2 := p
      => tauxBatterie := tauxBatterie - 0,1
      => PON_ch[1] := PON_ch[1] - 1
      => PON_ch[2] := PON_ch[2] + 1
      => PON_ch[3] := PON_ch[3] + 1
      >> ALL[l : UNION(BRIDEUR, MACHINE_OUTIL)| lieu = l AND l.statut =
      fini].charge(p)
!bouge(o,d : LIEU)
      => lieu := d
      => tauxbatterie := tauxBatterie - 0,5
      => PON_mv[1] := PON_mv[1] - 1
      => PON_mv[2] := PON_mv[2] + 1
      => PON_mv[3] := PON_mv[3] + 1
!rencontreObstacle
      => statut := indisponible
!décharge(p : PIECE)
      => contenuP2 := rien
      => contenuP1 := contenuP2
      => tauxBatterie := tauxBatterie - 0,1
      => PON_dé[1] := PON_dé[1] - 1
      => PON_dé[2] := PON_dé[2] + 1
      => PON_dé[3] := PON_dé[3] + 1
      >> ALL[l : UNION(BRIDEUR, MACHINE_OUTIL)| lieu = l AND l.statut =
      vide].décharge(p)
!échoue
      => statut := indisponible
      {PON_ch[3] > 0} => PON_ch[2] := PON_ch[2] - 1, PON_ch[3] := PON_c
      - 1
      {PON_mv[3] > 0} => PON_mv[2] := PON_mv[2] - 1, PON_mv[3] := PON_m
      - 1
      {PON_mv[3] > 0} => PON_dé[2] := PON_dé[2] - 1, PON_dé[3] := PON_d
```

```
      - 1
      >> contr.échoue
      >> gest.échoue
!finisChargement(l : LIEU)
      => statut := prêt
      => LTT_prêt := clock.time
      => PON_ch[2] := PON_ch[2] - 1
      => PON_ch[3] := PON_ch[3] - 1
      => COT_ch := REMOVE(COT_ch, FIRST(COT_ch))
      >> contr.finisChargement(l)
!finisMouvement(o,d : LIEU)
      => statut := prêt
      => LTT_prêt := clock.time
      => PON_mv[2] := PON_mv[2] - 1
      => PON_mv[3] := PON_mv[3] - 1
      => COT_mv := REMOVE(COT_mv, FIRST(COT_mv))
      >> contr.finisMouvement(o,d)
!finisDéchargement(l : LIEU)
      => statut := prêt
      => LTT_prêt := clock.time
      => PON_dé[2] := PON_dé[2] - 1
      => PON_dé[3] := PON_dé[3] - 1
      => COT_dé := REMOVE(COT_dé, FIRST(COT_dé))
      >> contr.finisDéchargement(l)
!regénèreBatterie
      => tauxBatterie := MAX(100, tauxbatterie + 10)
!rechargeBatterie
      => tauxBatterie := 100
      => PON_re[1] := PON_re[1] - 1
      => PON_re[2] := PON_re[2] + 1
!finisRechBatterie
      => statut := prêt
      => LTT_prêt := clock.time
      => PON_re[2] := PON_re[2] - 1
      >> gest.finisRechBatterie
lanceChargement(l : LIEU)
      => statut := actif
      => PON_ch[1] := PON_ch[1] + 1
      => COT_ch := APPEND(COT_ch, clock.time)
lanceMouvement(o,d : LIEU)
      => statut := mobile
      => PON_mv[1] := PON_mv[1] + 1
      => COT_mv := APPEND(COT_mv, clock.time)
lanceDéchargement(o,d : LIEU)
      => statut := actif
      => PON_dé[1] := PON_dé[1] + 1
      => COT_dé := APPEND(COT_dé, clock.time)
lanceRechBatterie
      => statut := indisponible
      => PON_re[1] := PON_re[1] + 1
règleProblVéhic(l : LIEU, p1 : PIECE, p2 : PIECE)
      => statut := prêt
      => LTT_prêt := clock.time
      => lieu := l
      => contenuP1 := p1
      => contenuP2 := p2
!fail
+ !die
```

BEHAVIOUR

```
*! live
!charge(p : PIECE)
      ? NOT (lieu.contenuES = rien OR p <> lieu.contenuES OR statut =
      indisponible OR PON_ch[1] = 0 OR failed)
!bouge(o,d : LIEU)
      ? NOT (statut = indisponible OR PON_mv[1] = 0 OR failed)
!rencontreObstacle
      ? NOT (statut <> mobile OR failed)
!décharge(p : PIECE)
      ? NOT (p <> contenuP1 OR statut = indisponible OR PON_dé[1] = 0 C
      failed)
!échoue
      ? NOT (PON_ch[3] = 0 AND PON_mv[3] = 0 AND PON_dé[3] = 0 OR faile
!finisChargement(l : LIEU)
      ? NOT (PON_ch[2] = 0 OR failed)
!finisMouvement(o,d : LIEU)
      ? NOT (PON_mv[2] = 0 OR failed)
!finisDéchargement(l : LIEU)
      ? NOT (PON_dé[2] = 0 OR failed)
!regénèreBatterie
      ? NOT (tauxBatterie = 100 OR failed)
!rechargeBatterie
      ? NOT (PON_re[1] = 0 OR failed)
!finisRechBatterie
      ? NOT (PON_re[2] = 0 OR failed)
lanceChargement(l : LIEU)
lanceMouvement(o,d : LIEU)
lanceDéchargement(o,d : LIEU)
lanceRechBatterie
règleProblVéhic(l : LIEU, p1 : PIECE, p2 : PIECE)
!fail
      ? NOT (NOT failed))
+ !die
      ? NOT (EMPTY(COT_ch) OR EMPTY(COT_mv) OR EMPTY(COT_dé)
      OR (PON_re[1] > 0 OR PON_re[2] > 0))
```

A.9. PROTOTYPE DE LA SPÉCIFICATION DU VÉHICULE (DEUXIÈME MÉTHODE)

Cette section illustre la deuxième méthode de prototypage sur la dernière version de la spécification associée à l'agent `Véhicule`.

Remarque : on a utilisé les transformations des formules temporelles et des engagements abordées à la section 8.15. L'action `Echoue` est décomposée en `Echoue_ch`, `Echoue_mv` et `Echoue_dé` selon l'engagement où elle apparaît. (Cet artifice devrait être évité en améliorant la transformation des engagements.)

A.9.1. PHASE DE PRÉPARATION

Voir les spécifications ALBERT qui suivent.

A.9.2. PHASE D'INSTANCIATION

Voir les spécifications OBLOG qui suivent (extraits).

Véhicule

Declarations

Attributes

```
Statut : STATUT
Lieu : LIEU
Accès : SET[LIEU]
Contenu_P1 : PIECE
Contenu_P2 : PIECE
Taux_batterie : REAL

Contenu_ES : PIECE <- {Brideur, Machine_outil}
```

Derived Attributes

```
Nb_pièces : INT
      (Contenu_P1 = rien and Contenu_P2 = rien) => Nb_pièces = 0
      (Contenu_P1 = rien xor Contenu_P2 = rien) => Nb_pièces = 1
      (Contenu_P1 <> rien and Contenu_P2 <> rien) => Nb_pièces = 2
```

Actions

```
Charge(PIECE) -> {Brideur, Machine_outil}
Bouge(LIEU, LIEU)
Rencontre_obstacle
Décharge(PIECE) -> {Brideur, Machine_outil}
Echoue_ch -> {Contrôleur, Gestionnaire}
Echoue_mv -> {Contrôleur, Gestionnaire}
Echoue_dé -> {Contrôleur, Gestionnaire}
Finis_chargement(LIEU) -> Contrôleur
Finis_mouvement(LIEU, LIEU) -> Contrôleur
Finis_déchargement(LIEU) -> Contrôleur
Regénère_batterie
Recharge_batterie
Finis_rech_batterie -> Gestionnaire

Lance_chargement(LIEU) <- Contrôleur
Lance_mouvement(LIEU, LIEU) <- Contrôleur
Lance_déchargement(LIEU) <- Contrôleur
Lance_rech_batterie <- Gestionnaire
Règle_probl_véhic(LIEU, PIECE, PIECE) <- Gestionnaire
```

Constraints

State Behaviour

```
Taux_batterie > 0 and (sometimep(<= 600) Statut <> prêt or Taux_batteri
      100)
```

Init

```
Statut = prêt
Accès = {brideur, fraiseuse, tour}
Contenu_P1 = rien
Contenu_P2 = rien
```

Effects of Actions

```
contr.Lance_chargement(l): Statut = actif
Charge(p):
      (Nb_pièces = 0) => Contenu_P1 = p
      (Nb_pièces = 1) => Contenu_P2 = p
      Taux_batterie = Taux_batterie - 0,1
Finis_chargement(l): Statut = prêt
contr.Lance_mouvement(o,d): Statut = mobile
Bouge(o,d):
      Lieu = d
      Taux_batterie = Taux_batterie - 0,5
Rencontre_obstacle: Statut = indisponible
Finis_mouvement(o,d): Statut = prêt
contr.Lance_déchargement(l): Statut = actif
Décharge(l):
      Contenu_P2 = rien
      Contenu_P1 = Contenu_P2
      Taux_batterie = Taux_batterie - 0,1
Finis_déchargement(l): Satut = prêt
Regénère_batterie: Taux_batterie = Max(100, Taux_batterie + 10)
gest.Lance_rech_batterie: Statut = indisponible
Recharge_batterie: Taux_batterie = 100
Finis_rech_batterie: Statut = prêt
Echoue_ch: Statut = indisponible
Echoue_mv: Statut = indisponible
Echoue_dé: Statut = indisponible
gest.Règle_probl_véhic(l, p1, p2):
      Statut = prêt
      Lieu = l
      Contenu_P1 = p1
      Contenu_P2 = p2
```

Commitments

```
contr.Lance_chargement(l) -> sometimef(<= 10)
      ((Charge(p);
```

```
      Finis_chargement(l))
      xor Echoue_ch)
contr.Lance_mouvement(o,d) -> sometimef(<= 40)
      ((Bouge(o,d);
      Finis_mouvement(o,d))
      xor Echoue_mv)
contr.Lance_déchargement(l) -> sometimef(<= 10)
      ((Décharge(p);
      Finis_déchargement(l))
      xor Echoue_dé)

gest.Lance_rech_batterie -> sometimef
      (Recharge_batterie;
      Finis_rech_batterie)
```

Responsibility

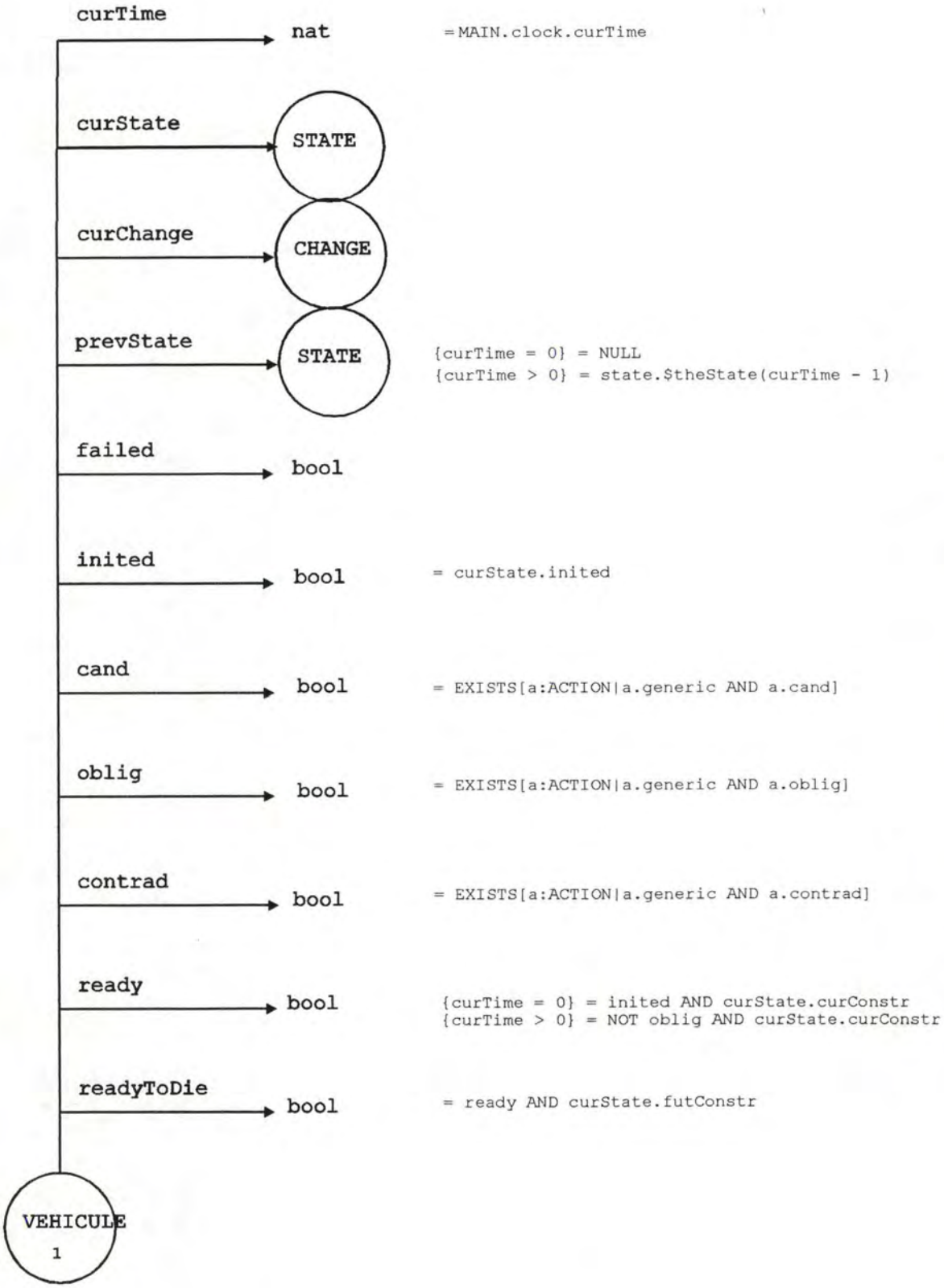
```
F(Charge(p) / Lieu.Contenu_ES = rien or p <> Lieu.Contenu_ES or Statut =
      indisponible)
F(Bouge(o,d) / Statut = indisponible)
F(Rencontre_obstacle / Statut <> mobile)
F(Décharge(p) / p <> Contenu_P1 or Statut = indisponible)
F(Regénère_batterie / Taux_batterie = 100)
```

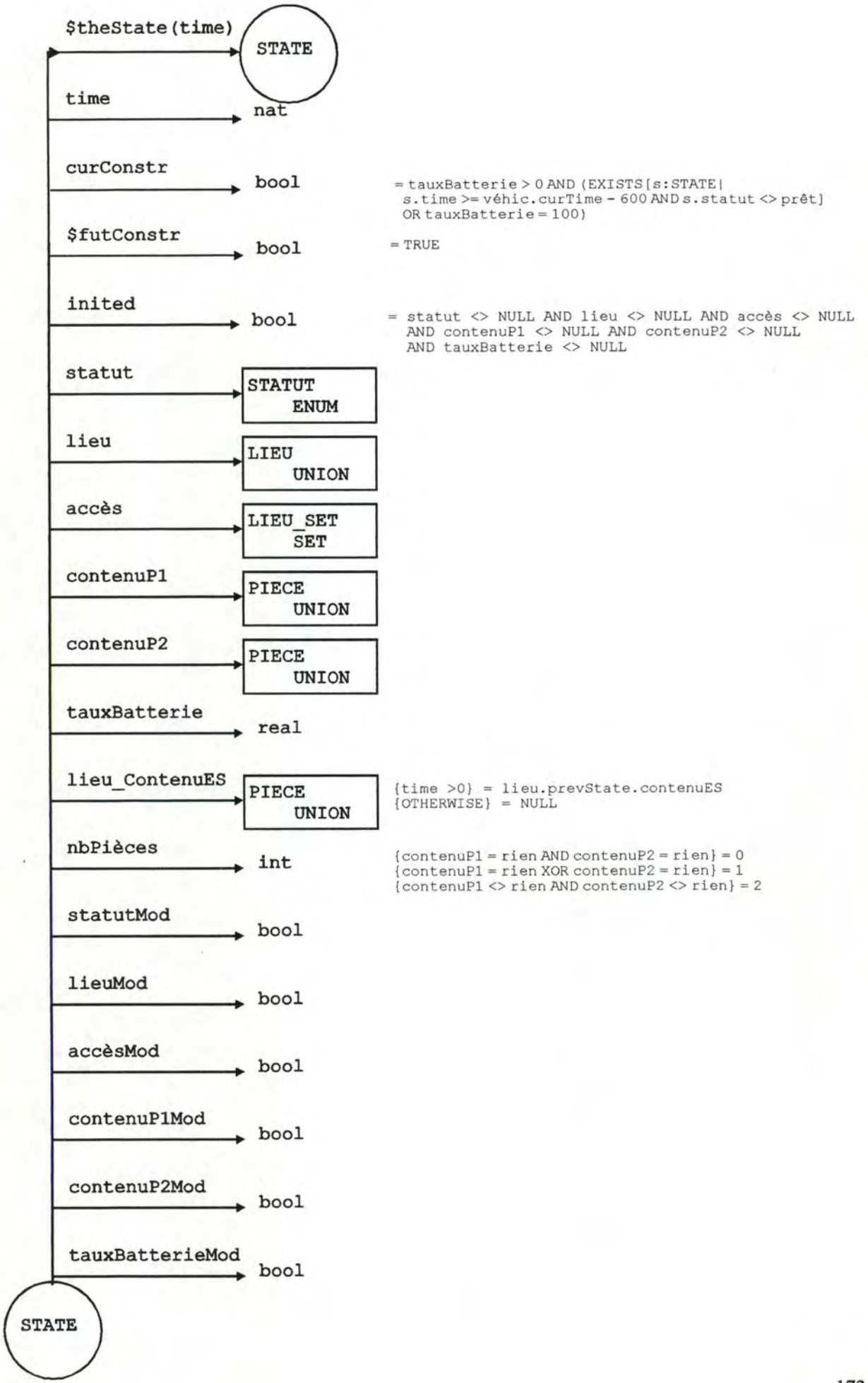
Perception

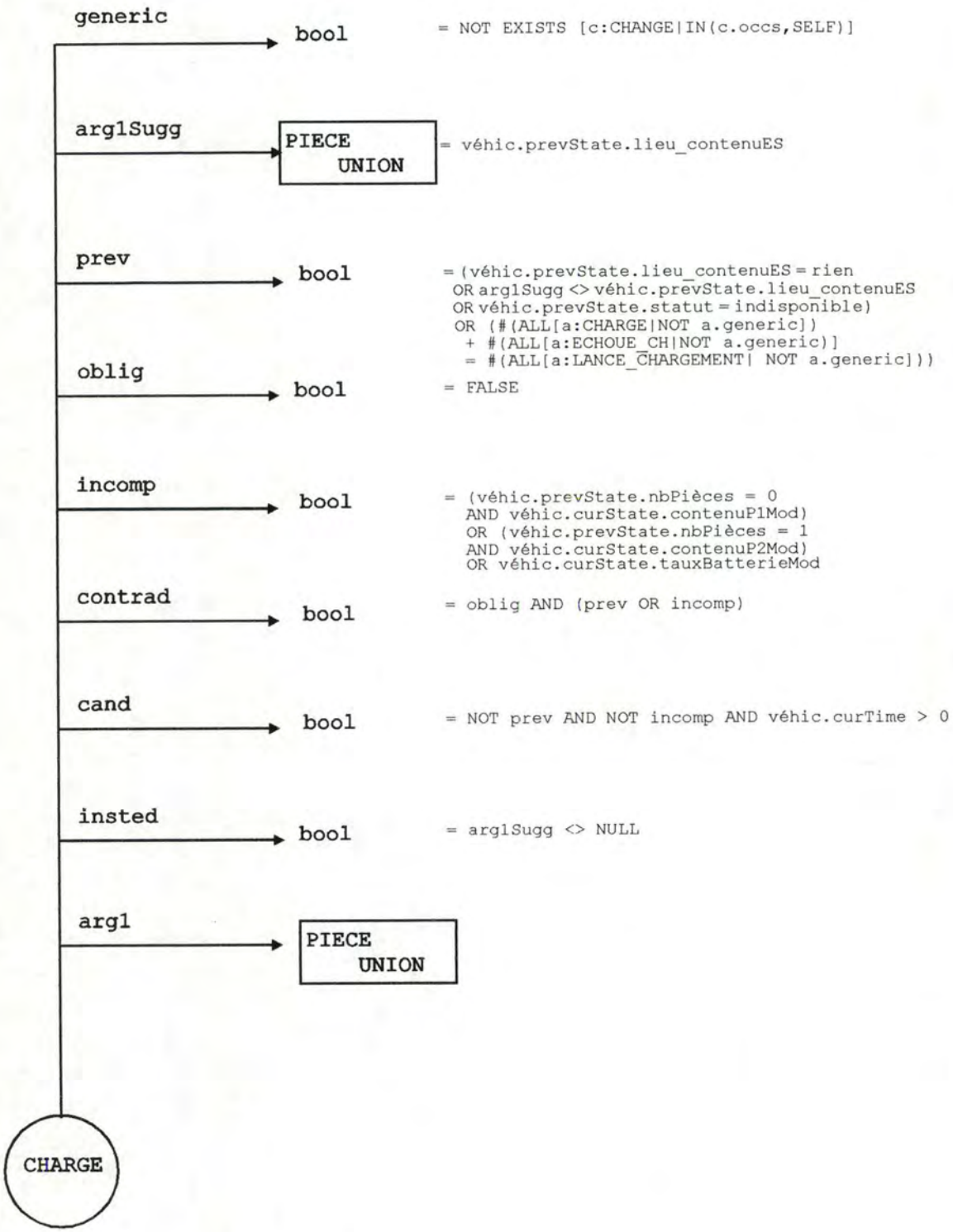
```
X(l.Contenu_ES / Lieu = l)

O(contr.Lance_chargement(l) / Statut = prêt and Lieu = l and Nb_pièces <>
      2)
F(contr.Lance_chargement(l) / not (Statut = prêt and Lieu = l and Nb_pièces
      <> 2))
O(contr.Lance_mouvement(o,d) / Statut = prêt and Lieu = o and In(Accès, d))
F(contr.Lance_mouvement(o,d) / not (Statut = prêt and Lieu = o and
      In(Accès, d)))
O(contr.Lance_déchargement(l) /
      Statut = prêt and Lieu = l and Lieu.Contenu_ES = rien)
F(contr.Lance_déchargement(l) /
      not (Statut = prêt and Lieu = l and Lieu.Contenu_ES = rien))
O(gest.Lance_rech_batterie / Statut <> actif and Statut <> mobile)
F(gest.Lance_rech_batterie / not (Statut <> actif and Statut <> mobile))
O(gest.Règle_probl_véhic(l, p1, p2) / Statut = indisponible)
F(gest.Règle_probl_véhic(l, p1, p2) / not (Statut = indisponible))
```

Publicity

```
X(Charge(p).l / Lieu = l)
X(Décharge(p).l / Lieu = l)
```


curTime
nat
=MAIN.clock.curTime
curState
STATE
curChange
CHANGE
prevState
STATE
{curTime = 0} = NULL
{curTime > 0} = state.$theState(curTime - 1)
failed
bool
inited
bool
= curState.inited
cand
bool
= EXISTS[a:ACTION|a.generic AND a.cand]
oblig
bool
= EXISTS[a:ACTION|a.generic AND a.oblig]
contrad
bool
= EXISTS[a:ACTION|a.generic AND a.contrad]
ready
bool
{curTime = 0} = inited AND curState.curConstr
{curTime > 0} = NOT oblig AND curState.curConstr
readyToDie
bool
= ready AND curState.futConstr
VEHICULE
1


$theState(time)
STATE
time
nat
curConstr
bool
= tauxBatterie > 0 AND (EXISTS[s:STATE|
s.time >= véhic.curTime - 600 AND s.statut <> prêt]
OR tauxBatterie = 100)
$futConstr
bool
= TRUE
inited
bool
= statut <> NULL AND lieu <> NULL AND accès <> NULL
AND contenuP1 <> NULL AND contenuP2 <> NULL
AND tauxBatterie <> NULL
statut
STATUT
ENUM
lieu
LIEU
UNION
accès
LIEU_SET
SET
contenuP1
PIECE
UNION
contenuP2
PIECE
UNION
tauxBatterie
real
lieu_ContenuES
PIECE
UNION
{time >0} = lieu.prevState.contenuES
{OTHERWISE} = NULL
nbPièces
int
{contenuP1 = rien AND contenuP2 = rien} = 0
{contenuP1 = rien XOR contenuP2 = rien} = 1
{contenuP1 <> rien AND contenuP2 <> rien} = 2
statutMod
bool
lieuMod
bool
accèsMod
bool
contenuP1Mod
bool
contenuP2Mod
bool
tauxBatterieMod
bool
STATE


generic
bool
= NOT EXISTS [c:CHANGE|IN(c.occs,SELF)]
arg1Sugg
PIECE
UNION
= véhic.prevState.lieu_contenuES
prev
bool
= (véhic.prevState.lieu_contenuES = rien
OR arg1Sugg <> véhic.prevState.lieu_contenuES
OR véhic.prevState.statut = indisponible)
OR (#(ALL[a:CHARGE|NOT a.generic])
+ #(ALL[a:ECHOUE_CH|NOT a.generic)]
= #(ALL[a:LANCE_CHARGEMENT| NOT a.generic]))
oblig
bool
= FALSE
incomp
bool
= (véhic.prevState.nbPièces = 0
AND véhic.curState.contenuP1Mod)
OR (véhic.prevState.nbPièces = 1
AND véhic.curState.contenuP2Mod)
OR véhic.curState.tauxBatterieMod
contrad
bool
= oblig AND (prev OR incomp)
cand
bool
= NOT prev AND NOT incomp AND véhic.curTime > 0
insted
bool
= arg1Sugg <> NULL
arg1
PIECE
UNION
CHARGE