MASTER EN SCIENCES INFORMATIQUES

Conception des cahiers des charges pour des systèmes distribués temps réel

Nahimana, Adolphe

Award date:
1999

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX, NAMUR

INSTITUT D'INFORMATIQUE

RUE GRANDGAGNAGE, 21, B-5000 NAMUR (BELGIUM)

Année Académique 1998-1999

Conception des cahiers des charges pour des systèmes distribués temps réel.

Adolphe Nahimana

Mémoire présenté en vue de l'obtention du grade de

Maître en Informatique

Avant-propos

Je remercie en premier lieu le professeur Pierre-Yves Schobbens pour ses conseils. Il n'a jamais laissé passer une occasion pour me guider.

Je remercie également le professeur Eric Dubois pour m'avoir proposé ce sujet.

Mes sincères remerciements à mes deux superviseurs pendant mon stage à l'ONERA-CERT à Toulouse : monsieur Jack Foisseau et monsieur Michel Lemoine. Ils ont tout fait, d'une part par leurs conseils, mais aussi pour que mon séjour à Toulouse soit le plus agréable possible.

Je tiens à remercier les autorités des Facultés Universitaires Notre Dame de la Paix de Namur pour avoir eu confiance en mes capacités. Ils m'ont soutenu dès mon arrivé à Namur, et continuent à me soutenir, au moment où j'écris ces lignes. Je pense particulièrement au professeur Raymond Paquay.

Je remercie très vivement ma famille pour tout l'amour qu'elle a toujours porté sur moi, pour ces efforts consentis pour que je puisse faire mes études.

Merci à tous les amis qui de près ou de loin ont contribué à ce que mon séjour à Namur soit agréable. ils ont été nombreux à être à mes côtés, à me soutenir. Ils n'ont pas cessé de me témoigner leur amitié. Que monsieur Mbayahaga Jean trouve ici l'expression de mes remerciements.

Résumé

Dans ce document, nous traitons de l'ingénierie des besoins dans le développement d'un système informatique. Durant cette activité l'analyste où l'informaticien cherche à obtenir une compréhension du système à mettre en place. Compréhension établie en terme de modèles. Ces modèles doivent être aussi complète et cohérente que possible. Pour des systèmes critiques, il faut que des méthodes, des outils appropriés soient utilisés. Nous présentons des outils, des méthodes qui aident à supporter rigoureusement cette phase. Nous proposons ensuite une méthodologie qui permettrait d'intégrer le langage ALBERT II et le langage UML dans un même projet de modélisation.

Abstract

In this paper, we deal with requirement engineering in the development of a software system. During this activity the analyst attempts to obtain a comprehension of the system. This comprehension is established in terms of models of the system. These models have to be as complete and coherent as possible. For critical systems, it is necessary to use methods and suitable tools. This report presents tools and methods supporting this phase rigorously. After what we propose a methodology to integrate the language ALBERT II and language UML in a same modeling project.

Introduction Générale

Il y a exactement trente ans, les informaticiens se rendaient compte que l'industrie du logiciel est en crise. Des recherches furent menées pour essayer de diminuer les coûts de développement de gros systèmes mais aussi arriver à accroître la moyenne des systèmes qui arrivent à terme, surtout pour les systèmes complexes. Mais trente ans après, même si des progrès ont été accomplis, les préoccupations courantes du génie logiciel restent : une faible productivité, un manque de fiabilité, une maintenabilité difficile et une réalisation onéreuse.

Cependant la modélisation apporte aujourd'hui une réponse efficace et rigoureuse aux problèmes de la maîtrise des problèmes du génie logiciel. Considérée hier comme une technique coûteuse et réservée seulement au domaine de la recherche, la modélisation et en particulier la modélisation formelle s'appuie sur des formalismes, et des outils éprouvés. Elle commence timidement à être employée par certaines organisations et sera pour demain une approche incontournable pour accroître la productivité. Aucun ingénieur ne peut imaginer qu'on puisse construire un pont sans plans fiables, ou pire, vérifier sa stabilité en faisant passer des camions dessus[1]. Pourtant, en informatique, la plupart des organisations sont encore à ce stade. Certaines applications sont toujours réalisées en dehors de tout modèle.

Pour des systèmes critiques où l'erreur ne se calcule pas seulement en termes de millions de dollars perdus mais aussi en nombre de vies humaines sacrifiées, il est indispensable que les besoins du système à mettre en place soient identifiés et modélisés avec précision. Cette précision sur les comportements attendus des différentes entités fonctionnelles qui composent le système ne peut être atteint qu'avec des méthodes formelles qui elles garantissent une certitude absolue.

Dans ce travail nous traitons de l'ingénierie des besoins dans le processus de développement d'une application informatique. Cette activité s'inscrit dans les premières phase de ce processus.

[1] Analogie donnée par le professeur Eric Dubois pendant un des ses cours de méthodologie de développement de logiciel

Nous avons adopté une démarche en trois étapes. Nous commençons d'abord à présenter la problématique et les concepts de bases des applications distribuées temps réel. Ces applications présentent beaucoup de contraintes dans leur réalisation. C'est pourquoi nous nous sommes intéressés dans la deuxième partie du travail à des outils théoriques éprouvés couramment utilisés dans le développement de ces applications. Les outils que nous avons choisis, à savoir la logique temporelle et les machines à états contiennent des formalismes adaptés aux différentes facettes du problème. Nous tenons dans la présentation de ces outils, à insister sur ce qu'on peut en faire.

La troisième partie du travail est consacrée à deux langages de spécification, à savoir le langage ALBERT II, ainsi que le langage UML.

Dans le chapitre sur le langage ALBERT II, nous présentons brièvement les principales caractéristiques du langage.

Dans le chapitre sur le langage UML, nous traitons des caractéristiques techniques du langage. UML est en phase de devenir un standard de la modélisation orientée objet. Qui peut donc se passer d'une «norme» ? La présence de ce chapitre dans notre travail a été dictée par notre curiosité sur une éventuelle coexistence UML-Albert II dans un même projet de modélisation. Nous sommes convaincus qu'aucune méthode de modélisation ne peut tout exprimer et qu'il faut parfois compléter les lacunes des uns par les points forts des autres. Dans cette phase de modélisation où on cherche à mieux comprendre le système, tout modèle qui peut aider à cela est le bienvenu.

Nous essayons enfin d'illuster les notions et concepts introduits dans les chapitres précedents par un exemple d'un système distribué temps réel. Nous montrons la complémantarité UML-ALBERT II. Nous partons de quelques diagrammes UML, (les diagrammes des cas d'utilisation et les diagrammes de séquence) et nous essayons de produire une spécification avec le langage ALBERT II.

Table des matières

Table des figures

Chapitre 1

Les principales caractéristiques d'un système distribué temps réel

1.1 Introduction

La véritable histoire de l'informatique commence avec le premier calculateur fonctionnel, numérique et programmable de Konrad Zuse. Après la guerre des progrès remarquables furent atteints, avec le premier ordinateur, l'ENIAC[1] mis au point dans un centre de recherche américain. Les ordinateurs à cette époque avaientt la taille d'un hangar, avec un volume de quelques centaines de mètres cubes.

Les générations de machines qui ont suivies étaient caractérisées par un modèle d'ordinateurs, dits ordinateurs centraux. Des utilisateurs assis devant leurs terminaux, devaient partager un processeur central. Avec ce type de système dit «à temps partagé», chaque utilisateur obtenait un temps pour l'exécution de ses programmes, et le processeur central devrait passer successivement d'un programme à l'autre. L'informatique à cette époque se caractérisait par une fermeture totale sur soi-même. C'est l'ère des systèmes fermés.

Dans les années quatre-vingt, une nouvelle tendance voit le jour au moment où les ordinateurs personnels apparaissent sur le marché. Ces derniers étaient tels que toute la puissance de traitement, la capacité de stockage, les programmes appartenaient à un et un seul utilisateur.

Dans le but d'aider les utilisateurs des ordinateurs personnels à accéder à des données partagées (au sein d'une organisation par exemple), les ordina-

[1]Electronic Numerical Integrator And Calculator

teurs personnels furent connectés ensemble au sein d'un réseau. Des ordinateurs spécialisés étaient utilisés comme fournisseurs de services. C'est la naissance de l'architecture client-serveur. C'est aussi l'époque de la naissance de systèmes composés de plusieurs sous systèmes reliés par l'intermédiaire d'un réseau et qui doivent coopérer pour la réalisation d'une tâche spécifique.

Dans ce chapitre nous allons aborder bièvement quelques éléments fondamentaux et théoriques sur les systèmes distribués temps réel. Que ce soit les applications distribuées ou temps réel, toutes ces applications sont constituées par plusieurs processus ou agents. Nous allons tout d'abord passer en revu les principales caractéristiques d'une application distribuée temps réel. Nous nous limiterons à énoncer leurs propriétés ainsi que les contraintes qui interviennent dans la conception de ces systèmes.

1.2 Principales caractéristiques d'un système distribuée

Un système distribué[2] est une application dont l'architecture est organisée en un ensemble de petites entités qui coopèrent par échange de messages et d'événements.

La conception de ce genre d'applications est une tâche difficile, vu les contraintes de distribution auxquelles elles doivent répondre. En effet une telle application doit supporter le parallélisme, la coopération et la synchronisation entre les différents processus qui la composent.

La distribution dans une application distribuée est matérialisée par le fait qu'elle se compose d'un certain nombre d'entités fonctionnelles, chacune d'entre elles étant spécialisée dans la réalisation d'une tâche bien précise où l'expertise est bien restreinte. Il est impératif que les différentes entités fonctionnelles coopèrent pour pouvoir arriver à la réalisation de l'objectif global de l'application.

La notion d'entité fonctionnelle peut être définie[2], par la notion d'agent, qui est une unité conceptuelle, autonome et qui interagit avec d'autres agents pour la réalisation d'un but commun : les objectifs globaux de l'application.

Un agent est donc capable d'agir sur son environnement qui est peuplé par d'autres agents. Son comportement est la conséquence de ses **observations**, de son **savoir** et de ses **interactions** avec d'autres agents. Il est donc capable d'apprendre et d'augmenter ses connaissances.

Les systèmes plus compliqués peuvent être structurés en plusieurs sous-systèmes et un certain nombre d'agents sont regroupés ensemble en une société d'agents[21](FIG. 1.1).

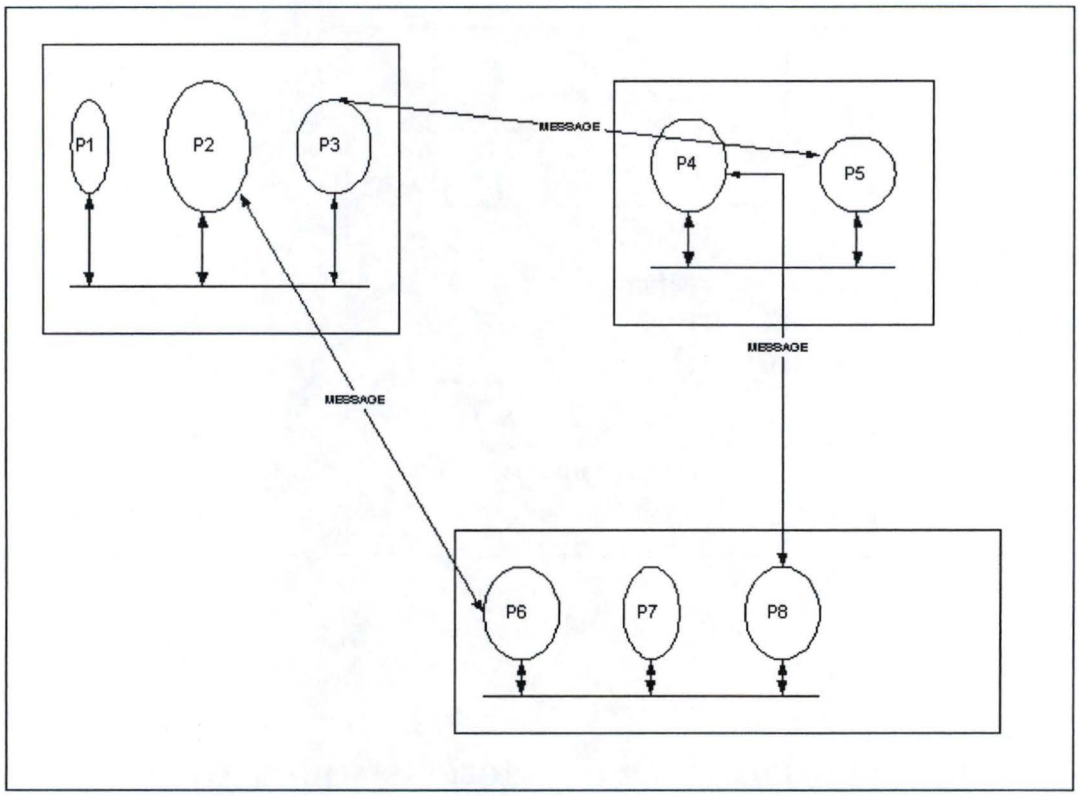


FIG. 1.1: Un exemple d'une application distribuée composée de six processus

1.2.1 Structure interne d'un agent

La figure(FIG. 1.2) représente la structure interne d'un agent. Les interfaces d'entrée et de sortie assurent la communication de l'agent avec son environnement. La partie **action** contient les traitements, c'est à dire les tâches qui sont sous la responsabilité de l'agent, tandis que la partie **composant d'état** contient les variables d'état qui caractérisent l'agent au cours de sa vie. C'est sur base des valeurs de ces variables d'état que certaines actions vont être déclenchées.

Un agent a une durée de vie pendant laquelle il a des actions sous sa responsabilité. Par exemple un agent chargé de réaliser une certaine action va accomplir une série de traitements, chacun d'entre eux ayant une durée déterminée selon la complexité de l'action à réaliser. Durant son existence, un agent

passe par plusieurs états différents dans lesquels les flux de messages qu'il enverra ou qu'il sera capable de recevoir seront différents. Le comportement d'un agent peut être déduit en examinant les flux de messages qui entrent ou qui sortent via les deux interfaces.

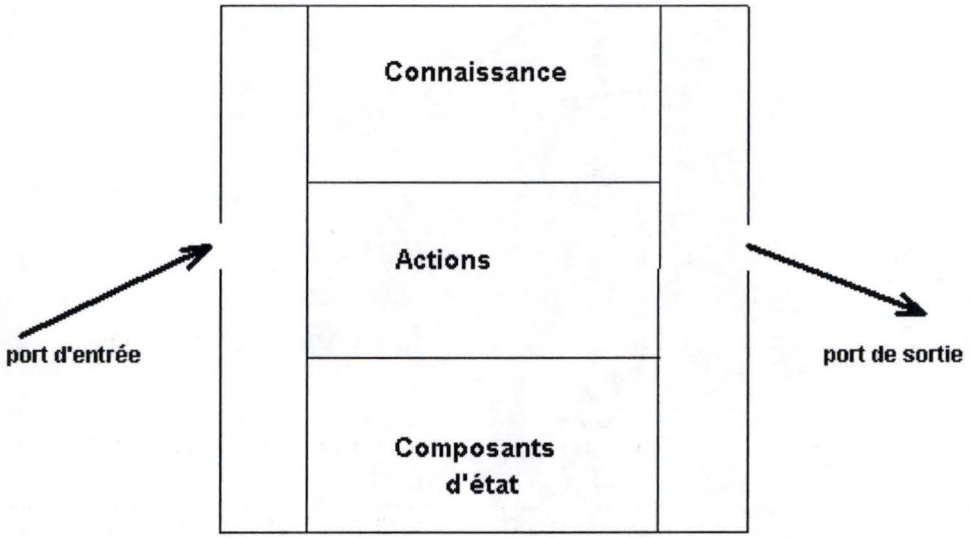


FIG. 1.2: Un modèle général de structure interne d'un agent

1.3 Les mécanismes et caractéristiques de base du temps réel

Dans les sections précédentes, nous nous sommes occupés à des applications distribuées sans toutefois aborder les aspects temps réel. Si nous considérons ces aspects, nous avons une application distribuée temps réel.

Les applications temps réel sont des applications qui par rapport aux applications classiques munissent, d'un système de dates les flux de données en entrée et en sortie. Le fonctionnement correct de l'application est fonction du flux de données en entrée ou en sortie mais aussi de la date à laquelle ce flux est arrivé ou a été produit. Le temps de réponse face à des données en entrée est une propriété importante pour de telles applications.

Les temps de réponse précisent, face à des données en entrée, les délais acceptables pour lesquels une réponse a encore une valeur. Ceci signifie qu'impérativement, l'application doit garantir le respect des échéances dans l'exécution des actions. Le temps réel regroupe sous une même appellation tout un spectre d'applications allant des applications dirigées par le temps, c'est à dire

des applications qui doivent accomplir une tâche spécifique après un certain intervalle de temps, aux applications orientées événements(réactif), c'est à dire des applications dont les actions sont déclenchées par la survenance de certains événements.

Il existe deux manières de voir le temps dans les applications temps réel : d'une part **qualitatif** et d'autre part sous l'angle **quantitatif**. Du point de vue qualitatif, c'est la problématique de synchronisation entre les différentes actions qui doit être considérée. Quantitativement, ce sont les contraintes de délai entre l'arrivée d'un événement et le début d'une action, les contraintes de fréquence des actions(pour des actions qui devront être répétées) et les contraintes de durée des actions.

On distingue dans les applications temps réel deux types de contraintes temps réel : d'une part les contraintes temps réel dites **«dures»** et d'autre part les contraintes temps réel dites **«molles»**. Les contraintes temps réel dures sont des contraintes pour lesquelles l'application doit impérativement garantir le respect des échéances. Ce type de contraintes se retrouve dans des applications à fonctionnement avec sûreté critique. Un non respect des échéances peut entraîner des conséquences catastrophiques.

Les contraintes temps réel molles sont des contraintes pour lesquelles l'application doit garantir le respect des échéances mais tolère des cas de non respect de ces échéances.

Dans le système téléphonique par exemple, il est bien spécifié qu'un certain temps après qu'un utilisateur ait composé un numéro d'un correspondant, une tonalité doit retentir. Cependant le fait que cette tonalité ne soit pas présente dans les limites de temps spécifiées n'entraîne aucune conséquence grave si ce n'est qu'une dégradation des performances du système.

Les processus temps réel peuvent aussi être caractérisés par leur **périodicité** ou **apériodicité**. Un processus périodique est un processus qui est activé à intervalle régulier, par exemple un système qui doit examiner un capteur toutes les 2 secondes et agir selon la valeur lue. Un processus apériodique est un rocessus non cyclique, il est activé seulement par l'arrivée d'un message en provenance de l'environnement.

1.4 Problématiques liées aux applications distribuées en temps réel

La conception d'une application distribuée temps réel est une tâche compliquée. Outre les problèmes liés à la distribution de l'application cette dernière doit répondre à des contraintes de temps. C'est pourquoi il est indispensable que dans sa conception, des outils appropriés puissent être utilisés.

Une autre difficulté provient du fait que dans la conception d'un tel système, il est nécessaire que l'architecture du système soit modulaire[17]. Ceci permet aux noeuds de s'adapter à certaines charges. Ainsi si un noeud est plus chargé, il pourra transférer une partie de ses traitements à un noeud moins chargé.

La prise en compte des contraintes temps réel nécessite des techniques d'ordonnancement temps réel puissantes et bien adaptées à un environnement distribué, tout en résolvant les problèmes de dates des systèmes distribués : problème de synchronisation des horloges[17]. Il faut donc dans de tels systèmes un ordonnanceur de processus.

1.5 Conclusion

Dans ce chapitre, nous avons passé en revu les principales caractéristiques des applications distribuées temps réel. Nous avons vu que la conception de ces applications posent beaucoup de difficultés et qu'il faut des outils appropriés dans leur conception. Il existe beaucoup d'outils dédiés aux systèmes temps réel. Dans le chapitre suivant, nous avons choisi de présenter la logique temporelle, qui est un outil puissant pour exprimer les contraintes de temps dans un système temps réel

Chapitre 2

La Logique temporelle

2.1 Introduction

L'on a de temps en temps besoin d'exprimer des vérités ou des contre vérités. La logique propositionnelle est un outil qui à un certain niveau peut nous aider à exprimer cela. Cependant il existe des situations où cette logique s'avère impuissante. D'autres logiques comme la logique des prédicats, permet d'étendre les formalismes de la logique propositionnelle en permettant de traiter des vérités variables ou relatives selon les valeurs de certaines variables. Une affirmation peut être vraie ou fausse selon l'individu auquel le prédicat se rapporte.

Il existe des choses qui ne sont vraies que selon le monde dans lequel on se trouve. Par exemple, certaines affirmations sont vraies aujourd'hui, mais ne le seront pas demain, peut-être parce que les concepts sur lesquels se fondent ces affirmations ont des significations différentes aujourd'hui et demain.

Cette variabilité des vérités entre mondes est exprimée par d'autres logiques comme les formalismes modales ; la logique déontique, la logique épistémique la logique temporelle etc...

Toutes ces logiques introduisent par rapport à la logique propositionnelle deux opérateurs supplémentaires, l'opérateur $\Diamond$ ainsi que l'opérateur $\Box$ qui sont appelés respectivement opérateur **modal universel** et opérateur **modal existentiel**. Ces opérateurs changent de signification selon la logique considérée.

Dans ce chapitre nous allons parler de la logique temporelle propositionnelle. Nous allons dans un premier temps considérer un domaine temporelle discret. Dans la section suivante nous parlerons de la logique temporelle avec

métrique et nous terminerons par la logique temporelle des intervalles.

2.2 Logique temporelle propositionnelle linéaire

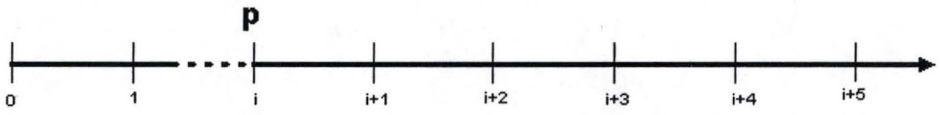
La logique temporelle propositionnelle linéaire peut être considérée comme une variante de la logique modale et permet de décrire des situations qui évoluent dans le temps. Ici, les différents mondes sont les différents instants de la ligne du temps et la relation d'accessibilité entre mondes est la relation **«est dans le futur de»** ou **«est dans le passé de»**.

La logique temporelle propositionnelle linéaire peut aussi être considérée comme une extension de la logique propositionnelle classique. Dans cette logique, on adjoint aux opérateurs classiques, des **opérateurs temporels.**

Pour chaque opérateur et chaque formule temporelle, nous allons donner une définition de l'interprétation dans un modèle donné. Le modèle que nous allons considérer est une suite de valuations[15] sur des propositions atomiques. Si une formule p est vraie à une date i, $i \geq 0$, dans la séquence σ, on dit que σ satisfait p en i. Ce qui se note

$$(\sigma, i) \models p$$

et qu'on peut schématiser par :

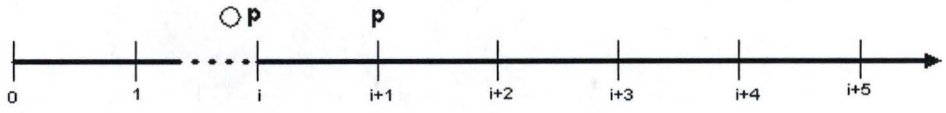


Opérateur «à l'instant prochain» ($\circ$)

La sémantique de cet opérateur est la suivante :

$$(\sigma, i) \models \circ p \Leftrightarrow (\sigma, i+1) \models p$$

La proposition $\circ p$ est vraie à la date i si et seulement si p est vraie à la date $i+1$. On peut le schématiser de la manière suivante :

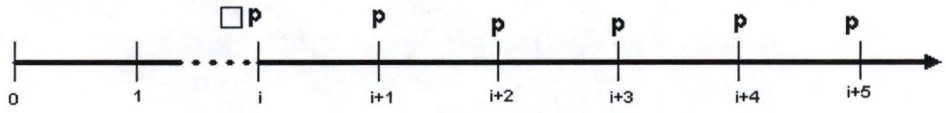


Opérateur «Toujours»(□)

La sémantique de cette opérateur est la suivante :

$$(\sigma, i) \models \Box p \Leftrightarrow \forall k \geq i : (\sigma, k) \models p$$

La proposition $\Box p$ est vraie à la date i si et seulement si p est vraie en toute date ultérieure ou égale à i. On peut schématiser cela par :

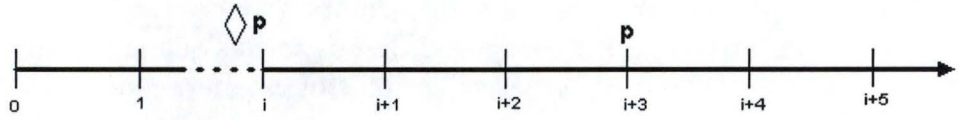


Opérateur «au moins une fois dans le futur»(◇)

La sémantique de cette opérateur est la suivante :

$$(\sigma, i) \models \Diamond p \Leftrightarrow \exists k \geq i : (\sigma, k) \models p$$

La proposition $\Diamond p$ est vraie à une date i si et seulement si la proposition p est vraie à une certaine date k ultérieure ou égale à la date i. Nous pouvons schématiser cela par :



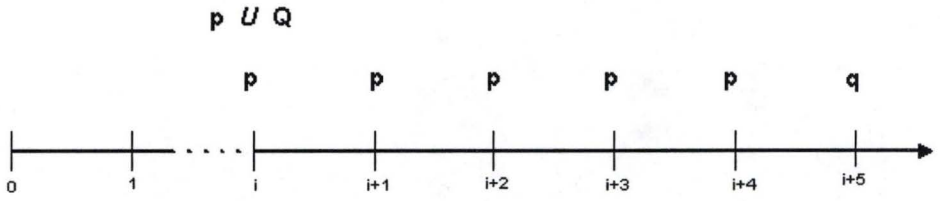
La proposition p peut être vraie à n'importe quelle date ultérieur à la date i. Cet opérateur est le dual de l'opérateur toujours.

$$(\sigma, i) \models \Diamond p \Leftrightarrow (\sigma, i) \models \neg\Box\neg p$$

Opérateur «Jusqu'à» (U)

Une proposition pUq combine à la fois un «**toujours**» et un «**au moins**», en prédisant qu'une occurrence future de q va survenir tout en exigeant que p reste vraie au moins jusqu'à la première occurrence de q.

$$(\sigma, i) \models pUq, \exists k \geq i : (\sigma, k) \models q \wedge \forall j, i \leq j < k, (\sigma, j) \models p$$



Notons en passant qu'il existe une version de cet opérateur, l'opérateur W (Until faible) qui est défini de cette façon :

$$(\sigma, i) \models pWq \Leftrightarrow (\sigma, i) \models pUq \vee (\sigma, i) \models \Box p$$

Tous ces opérateurs ont des opérateurs inverses si la relation entre les différents instants de la ligne du temps est la relation «**est dans le passé de**».

2.3 Logique temporelle avec métrique

Il arrive souvent dans des spécifications qu'on ait besoin de garder des instants qui sont à une distance donnée, ou dans un intervalle donné par rapport à un instant de départ. Ce type de contrainte se retrouve généralement dans la spécification des temps de réponse, le temps de maintien d'un agent donné dans un état donné. Ces contraintes sont des contraintes où l'information temporelle est de nature quantitative. Exemple : **La détection doit commencer 1 seconde après la réception du message cds**. Nous savons que la sémantique d'une formule de la logique est un ensemble d'histoires, une séquence infinie d'états.

En suivant l'approche que présente J-P. Haton[10] on garde les mêmes opérateurs temporels, mais introduit des notations pour restreindre les états à un sous ensemble d'états, qui appartient à un intervalle, qui est identifié par rapport à l'état d'évaluation.

- $< t_0$: les états qui sont à une distance inférieure à t_0

- $= t_0$: les états qui sont à une distance égale à t_0
- $> t_0$: les états qui sont à une distance supérieure à t_0
- $\leq t_0$: les états qui sont à une distance supérieure ou égale à t_0
- $\geq t_0$: les états qui sont à une distance inférieure ou égale à t_0

Par exemple pour l'opérateur **toujours**, nous pouvons restreindre la véracité d'une assertion à certains états qui sont à une certaine distance de l'origine.

- $\Box\phi$: ϕ est toujours vérifié dans le futur
- $\Box[< t_0]\phi$: ϕ est vrai dans tous les instants futurs distants de moins de t_0 unités de temps par rapport au temps d'évaluation.
- $\Box[= t_0]\phi$: ϕ est vrai dans tous les instants futurs distants de t_0 unités de temps par rapport au temps d'évaluation.
- $\Box[> t_0]\phi$: ϕ est vrai dans tous les instants futurs distants de plus de t_0 unités de temps par rapport au temps d'évaluation.
- $\Box[\geq t_0]\phi$: ϕ est vrai dans tous les instants futurs, distants de t_0 ou plus unités de temps par rapport au temps d'évaluation.
- $\Box[\leq t_0]\phi$: ϕ est vrai dans tous les instants futurs, distants de t_0 ou moins unités de temps par au rapport temps d'évaluation.

2.4 La logique temporelle des intervalles

Dans cette approche, due principalement à Allen [1],[10] la structure du temps n'est plus fondée sur des instants, mais sur des intervalles. C'est un aspect continu temps et non discret comme en logique temporelle linéaire. Les intervalles peuvent ici se chevaucher, mais aussi comme en logique temporelle linéaire se succéder. Dans cette approche, des opérateurs entre intervalles ont été introduits pour spécifier les notions de précedence et de couverture entre intervalles. Par exemple, pour la relation e_1 **Avant(d)** e_2, le début de l'intervalle e_2 a lieu d unités de temps après la fin de l'intervalle e_1. Pour la relation, e_1 **Couvre(d)** e_2 qui est une relation de chevauchement entre intervalles, elle spécifie que les intervalles e_1 et e_2 se recouvrent pendant une période d.

2.4.1 Les différentes relations entre intervalles

- e_1 **Egale** e_2 est vrai quand e_1 coïncide avec e_2, c'est à dire même début et même fin.
- e_1 **Avant**(d) e_2 est vrai quand le début de e_2 est séparé d'un délai d de la fin de e_1.

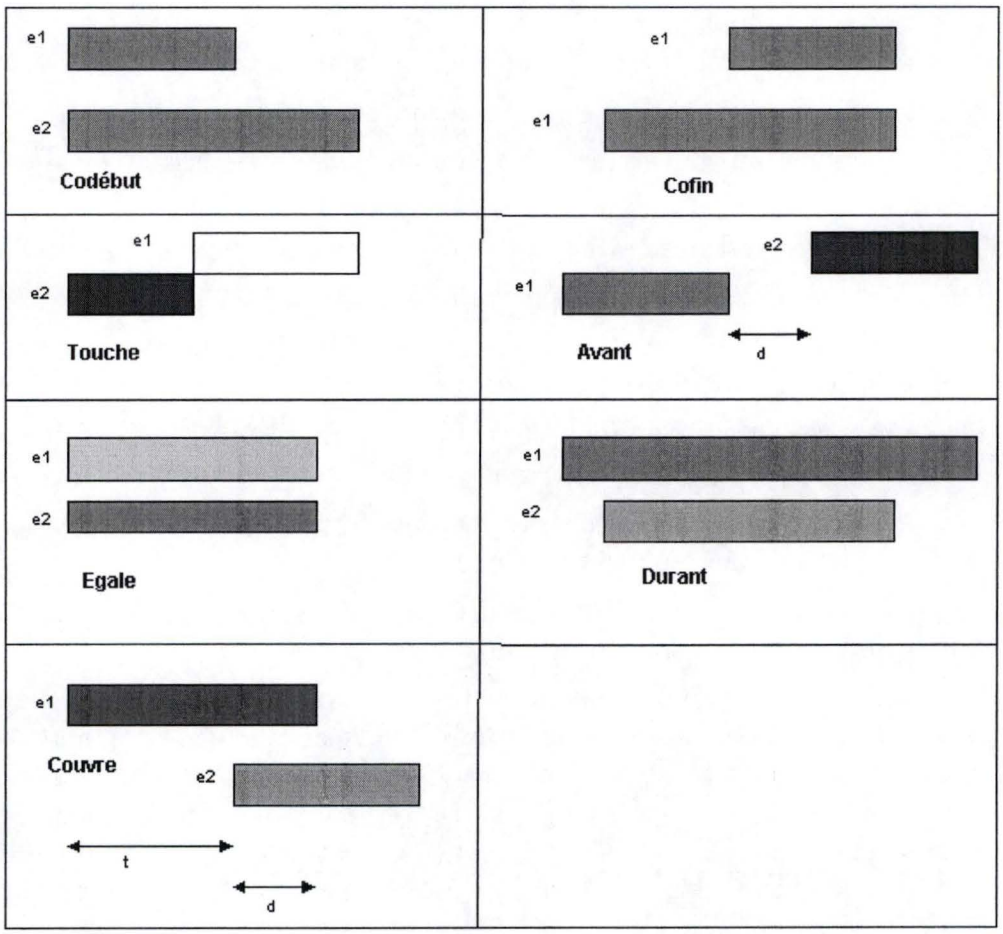


FIG. 2.1: Les différents relations entre intervalles de temps

- e_1 **Durant**(d) e_2 est vrai quand l'intervalle e_1 débute un certain moment après que l'intervalle e_2 ait débuté, et e_2 contient e_1.
- e_1 **Couvre**(d) e_2 est vérifié si l'intervalle e_1 débute un certain délai t avant le début de e_2 , mais e_1 se termine avant e_2, de telle sorte que la durée de durée (e_1) est égale à $t + d$, où d est appelé délai de recouvrement
- e_1 **touche** e_2 est vrai quand la fin de e_1 correspond exactement au début de e_2.
- e_1 **Codébut** e_2 est vrai quand e_1 et e_2 débutent au même instant et la durée de e_1 est plus petite que celle de e_2
- e_1 **Cofin** e_2 est vrai quand les deux intervalles finissent au même moment et que la durée de e_1 est plus petite que la durée de e_2

2.4.2 Propriétés

Les propriétés que nous donnons ci desous sont dues à Turner[25]. Dans son approche il considère les événements comme une structure qui est triplet $M =< E, R, \oplus >$ où

- E est un ensemble d'intervalles e_i
- R une relation binaire représentant la précédence entre les e_i, il s'agit de la relation **Avant(d)** qui est telle que $e_i R e_j \simeq \exists\ d : e_i$ **avant(d)** e_j
- $\oplus$ une relation binaire représentant le chevauchement entre les e_i, il s'agit de la relation **couvre(d)** qui est telle que $e_i \oplus e_j \simeq \exists\ d > 0 : e_i$ **couvre(d)** e_j

les propriétés sont les suivantes :

- $e_1\ R\ e_2 \Longrightarrow \neg\ (e_2\ R\ e_1)$
  relation R est **antisymétrique**
- $e_1 R e_2 \wedge e_2 R e_3 \Longrightarrow e_1 R e_3$
  R est une relation **transitive**
- $e_1 \oplus e_1$
  Tout intervalle se chevauche lui même, la relation est donc **réflexive**
- $(e_1 \oplus e_2) \Longrightarrow \neg(e_1 R e_2)$
  Les relations $\oplus$ et R sont des relations exclusives. Si deux intervalles de temps se chevauchent, alors il est impossible qu'il y ait un des deux qui soit avant l'autre.

Cette approche sert surtout à synchroniser des actions en se basant sur leurs intervalles d'exécution. On aboutit à un graphe de déclenchement des actions comme les graphes PERT.

exemple

Cette exemple est tiré du cas que nous avons modélisé avec le langage ALBERT II. Nous avons une action *«Ordre général»* qui débute en même temps que la génération du message Ogf. Quand cette action est finie, c'est l'action *«Ordreparticuliers»* qui est déclenchée et elle même touche l'action *préparation de mission*. La génération du message Ogf intervient avant l'action *ordreparticuliers*

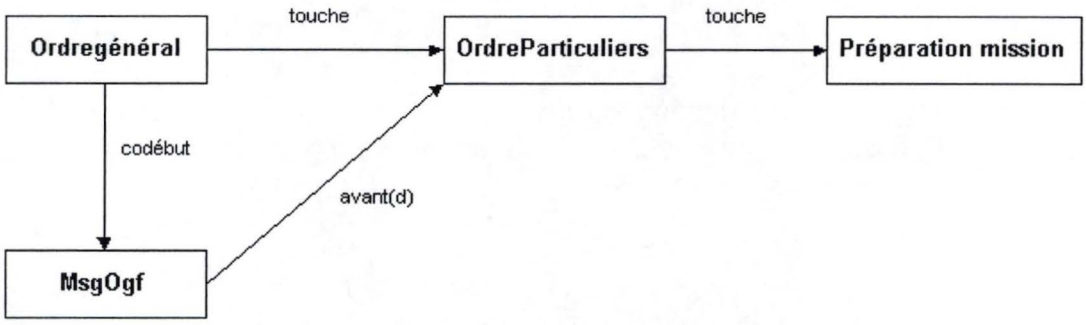


FIG. 2.2: Exemple d'un graphe temporel

2.5 conclusion

L'un des grands avantages[15] de la logique temporelle est qu'elle permet d'exprimer de manière concise les propriétés temporelles d'un système : *«La chaudière s'ouvrira toujours si on appuie sur le bouton»*, pour un système sûr ou *«l'alarme sera inévitablement déclenchée si une erreur apparaît»*.

Les logiques temporelles ont été enrichies afin de prendre en compte de manière quantitative l'écoulement du temps. Il est ainsi possible d'exprimer que certaines actions ont une durée mesurable. Plusieurs approches ont été explorées, se différenciant notamment par le fait que la nature du domaine temporel peut être discrète ou continue.

Que ce soit le domaine temporel continu ou discret, les concepts vus nous permettrons de comprendre certains formalismes présentés dans le langage ALBERT II.

Chapitre 3

Les machines à états finis

3.1 Introduction

Les machines à états forment un formalisme qui sert à modéliser des situations à plusieurs états ainsi que le passage d'un état à un autre. Prenons l'exemple d'une personne et limitons-nous à l'évolution de sa santé. Soit il est «*bien portant*», soit il est «*malade*», soit il est« *convalescent*». Il est impossible qu'il passe de l'état «*bien portant*» à «*convalescent*» sans passer par l'état «*malade*».

Nous allons dans ce chapitre présenter les machines à états finis. Nous essayerons principalement de présenter d'une façon intuitive le concept de machine à états, nous donnerons quelques propriétés y relatives ainsi que les les méthodes de représentation. Nous terminerons le chapitre avec les machines à états hiérarchiques et à états parallèles.

3.2 Modèle mathématique d'une machine à états

Une machine à états finis ou un automate fini[19] déterministe M peut être défini comme un quintuplet M=<I, F, Q, Σ , δ > où

- Q est l'ensemble de tous les états de l'automate.
- I est un état particulier qui appartient à Q et qui représente l'état initial de la machine.

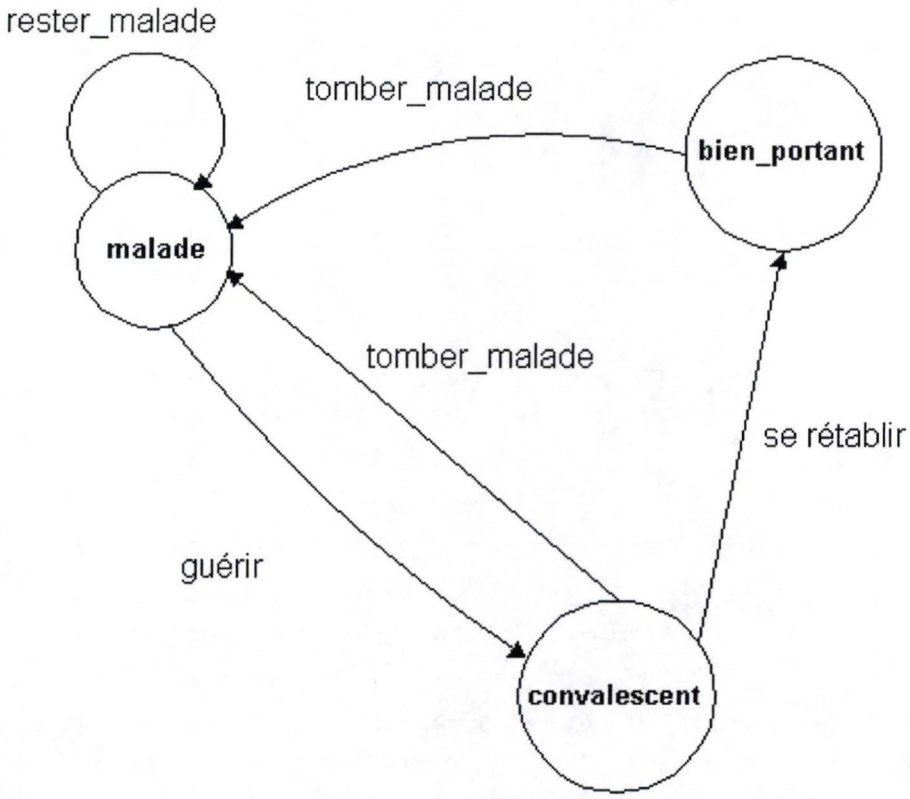


FIG. 3.1: Machine à états relative à la santé d'une personne

- F ensemble non vide d'états finaux de l'automate. F est un sous ensemble de Q.
- Σ est l'ensemble non vide de l'alphabet de l'automate.
- δ est une application qui par rapport à un état et un caractère mène vers un autre état. C'est la fonction de transition de l'automate.

Intuitivement une machine à état est un couple composé d'un ensemble d'états et de règles et ces dernières définissent les modalités de transitions entre les différents états.

Pour comprendre ce qu'est une machine à états, faisons une petite analogie avec un joueur à un jeu. Ce joueur connaît les règles du jeu et donc les coups permis et ceux qui sont interdits, il ne peut pas prévoir à l'avance ce qu'il va faire au coup suivant, cela dépendra en effet d'événements extérieurs à son influence, comme par exemple les cartes jouées par les autres joueurs, si c'est un jeu de cartes ou encore le numéro qui apparaîtra sur un dé, si le jeu consiste à lancer des dés. Par contre, à chaque fois qu'il lui faudra jouer, il saura ce qu'il a droit de faire et fera un choix. Parmi ces choix peut bien figurer celui de passer son tour, si cela est permis. Quand il ne pourra plus rien faire, il aura perdu.

Mais réciproquement, un certain nombre de situations connues à l'avance seront déclarées gagnantes pour lui.

Si ce joueur était une machine programmée, ce serait un excellent exemple d'une machine à états finis, à condition que le jeu obéisse aux contraintes suivantes :

1. les situations de jeu sont en nombre fini
2. chaque situation n'offre la possibilité que d'un nombre fini de réponses à l'événement extérieur (ce qui est toujours le cas dans la pratique, même si ces nombres peuvent être très grands),
3. le jeu se termine toujours en un nombre fini d'étapes (ce qui exclut par exemple le jeu d'échecs, où il peut y avoir des cycles.)

3.2.1 Représentation d'une machine à états

Les machines à états peuvent être représentées de trois manières différentes à savoir les tables de transitions, les diagrammes de transitions et les matrices de transitions.

Tables de transitions

Dans cette représentation tabulaire, on représente dans un tableau à deux entrées l'ensemble des événements(input) en ligne et l'ensemble des états en colonne. L'intersection de l'événement i en entrée et de l'état j correspond à l'état dans lequel on aboutit quand la machine est dans l'état j et que l'événement i survient. L'inconvénient majeur de cette représentation est qu'elle ne permet pas de donner plus de précisions sur les états, surtout l'état initial et états finaux.

Diagramme de transitions

Un diagramme de transition est un graphe orienté, dont les sommets sont les états de la machine à états et dont les arêtes sont les transitions possibles entre les états. Les états sont représentés par des ronds qui entourent le nom de l'état. Par convention on représente les états finaux par des ronds doubles et l'état initial par une flèche sans origine. Une transition entre deux états q_i et q_j est représentée par une arête orienté reliant q_i à q_j.

Matrices de transitions

La représentation par matrice de transition d'une machine à états nous amène à une matrice à n lignes et à n colonnes si la machine que nous voulons représenter a n états. L'élément M_{ij} de la matrice est l'événement qui doit survenir quand on est dans l'état q_i pour pouvoir basculer vers l'état q_j. Pour les états entre lesquels il n'y a pas de transition entre eux, on représente ce fait par un trait dans l'intersection dans la matrice de transition.

$$\mathcal{M}_{ij} = \begin{pmatrix} 1 & - & 0 & - \\ 1 & - & - & 0 \\ - & 1 & 0 & - \\ - & - & 1 & 0 \end{pmatrix}$$

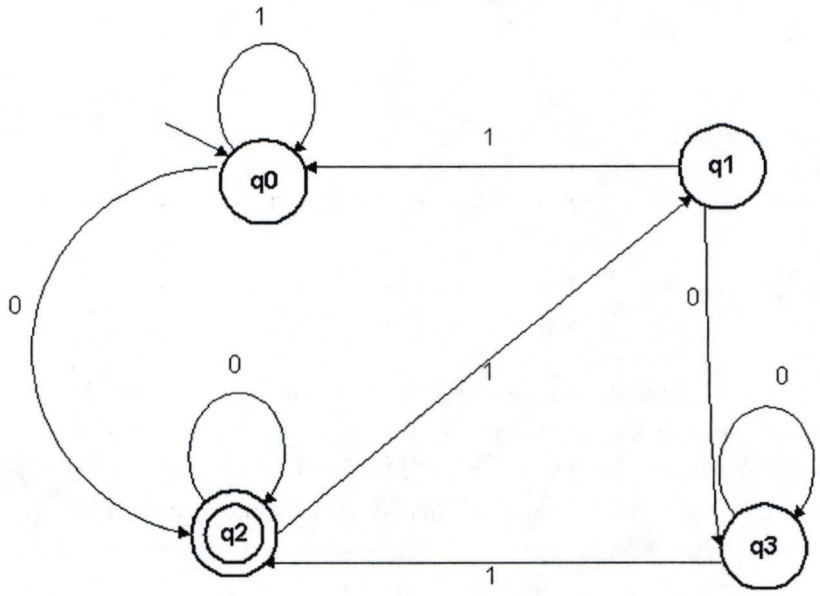


FIG. 3.2: Diagramme d'une machine à état

Pour cette machine nous avons

- $Q = \{q_0, q_1, Q_3, q_4\}$
- $\Sigma = \{0, 1\}$
- $I = \{q_0\}$
- $F = \{q_2\}$
- δ est tel que :

- $\delta(q_0, 1) = q_0$
- $\delta(q_0, 0) = q_2$
- $\delta(q_1, 1) = q_0$
- $\delta(q_2, 0) = q_2$
- $\delta(q_2, 1) = q_1$
- $\delta(q_3, 0) = q_3$
- $\delta(q_3, 1) = q_2$

	0	1
q_0	q_2	q_0
q_1	q_3	q_0
q_2	q_2	q_1
q_3	q_3	q_2

TAB. 3.1: table de transition associée au diagramme de transition de la figure(FIG. 3.2)

3.3 Machines à états hiérarchiques

Les machines à états que nous avons considérées jusqu'à présent sont des automates à états finis classiques. Les machines à états avec hiérarchie(Statecharts) est une extension des diagrammes de transition. L'extension porte sur le fait que des états peuvent être groupés dans un super-état(emboitement des états). Cette hiérarchisation des états permet à un moment donné de considérer la machine avec un niveau de détail voulu. On pourra considérer un super-état et occulter les sous-états qui composent la machine. Ceci a pour conséquence de réduire le nombre de transitions de la machine

Comme pour les diagrammes de transition, ces outils permettent de décrire des situations dont les différents états évoluent dynamiquement : une application temps réel par exemple. Cette description est réalisée en terme d'états et de transitions entre les différents états.

La principale innovation des statecharts est ce que D. Harel[11] appelle «**état-parallèle**» qui sont des états qui contiennent deux ou plusieurs machines à états, chacune ayant son propre compartiment dans la machine globale.

Considérons la machine à états de la figure(FIG.3.3). Cette machine est composée de deux machines A et B. La machine A contient un état normal,

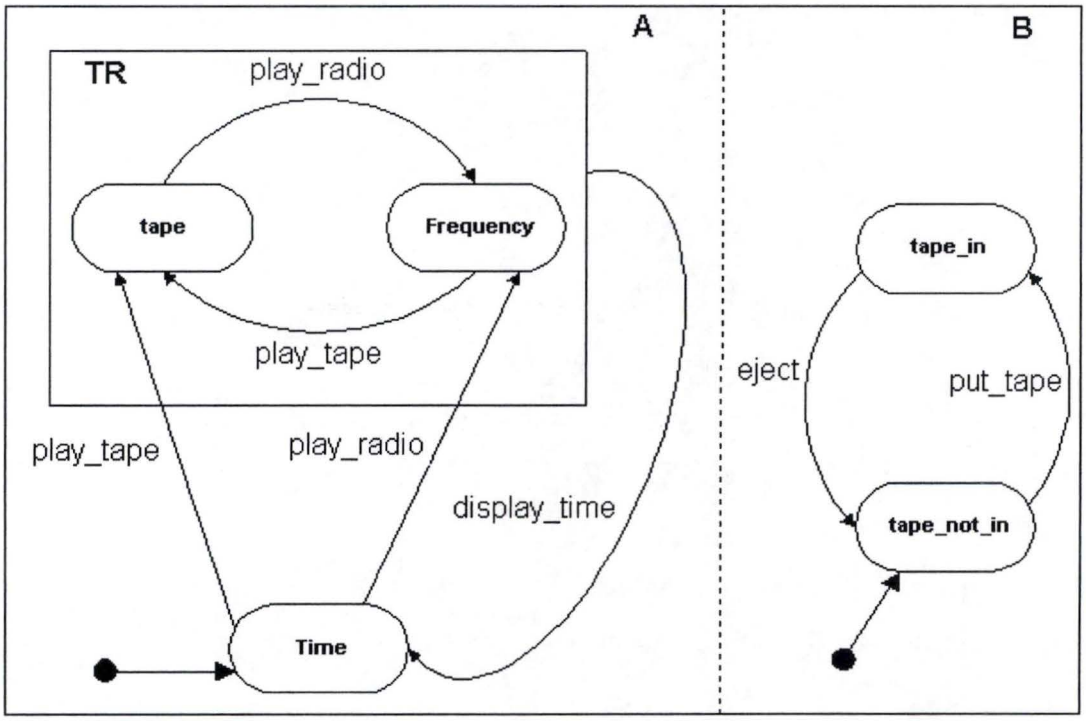


FIG. 3.3: Machine à état hiérarchique avec état-parallèle

l'état *time* et un état composite TR qui contient deux états $frequency$ et *tape* La machine B contient deux états normaux. Cette machine modélise un fonctionnement simplifié d'une radio-cassette. A tout moment l'état de la radio-cassette est un couple qui appartient dans le produit cartésien des états de A et B. La transition $display - time$ relie l'état TR et l'état *time*. Cela signifie qu'il existe des transitions entre les sous états $frequency$ et *tape* vers l'état *time*. Les sous états héritent des propriétés de leurs super-états.

3.3.1 Conclusion

Les propriétés que nous avons trouvé dans les machines à états en font un outil pour modéliser tout système qui évolue en fonction de son état. Ce qui est le cas pour les systèmes temps réel. Elles sont également très utilisées dans la description du contrôle de l'interaction dans les interfaces homme-machine. Le principal avantage est qu'elles pemettent une représentation dense : toute la dynamique du système peut être répresentée par des ronds et des flèches. Des variantes des machines à états sont incorporés dans beaucoup de langage de modélisation, surtout les langages de modélisation objet.

Le principal inconvénient des machines à états est que le nombre d'états croit d'une façon exponentiel, et le modèle devient presque illisible.

Chapitre 4

Le langage de spécification ALBERT II

4.1 Introduction

Nous assistons dans le monde de la recherche à un certain engouement pour les langages de spécification formels. Ces langages formels sont basés sur des outils mathématiques, permettant ainsi d'exprimer les besoins d'un système avec une grande précision.

La spécification d'un système avec un langage formel permet de vérifier si les comportements du système sont bien ceux qu'on en attend. Ceci est important pour des systèmes à fonctionnement critique, car cette description rigoureuse du système permettra peut-être de mettre à l'abri des futurs utilisateurs de surprises désagréables au moment de la mise en œuvre du système.

Les statistiques[7] dans ce domaine ont montré que les erreurs commises pendant cette phase de spécification sont les plus coûteux et les plus difficile à corriger. Il est vrai que l'utilisation d'un langage de spécification formel dans la modélisation demande beaucoup d'effort, mais il est vrai aussi que cette effort sera compensée par le fait qu'on arrivera à détecter facilement des erreurs, des ambiguités qu'on aurait eu du mal à détécter en utilisant une méthode non formelle de modélisation.

En nous inspirant du manuel de référence[8], nous allons donner une description de la sémantique intuitive des principaux concepts rencontrés dans le langage.

4.2 Types de données et opérations

La première partie d'une spécification Albert est composée d'une déclaration des types de donnés et des opérations sur ces types de données.

La partie déclaration de types est composé de trois parties :

- Types de données élémentaires prédéfinis
  Les types de données prédéfinis dans le langage ALBERT II sont : INTEGER, CHAR, BOOLEAN, RATIONAL, STRING et DURATION. Un certain nombre d'opérations relatives à ces types de données est également disponible dans le manuel de référence[8].
- Types de données élémentaires définies par l'analyste
- Types de données construites sur base des types de données élémentaires en utilisant les constructeurs ci-dessous :
  1. produit cartésien :CP
  2. ensemble : SET
  3. multi-ensemble : BAG
  4. séquence : SEQ
  5. table : TABLE
  6. union : UNION
  7. enuméré : ENUM

Le langage ALBERT II permet également de définir des opérations sur les types de données définies par l'analyste. Ces opérations sont en quelque sorte des fonctions mathématiques construites sur base de la logique des prédicats du premier ordre.

4.3 Déclaration des agents et des sociétés

La déclaration d'un agent consiste à donner les composants d'états et l'ensemble des actions qui sont sous la responsabilité de l'agent. C'est au moment de la déclaration des actions et composants d'états d'un agent qu'il faut signaler les actions ou composants d'états que l'agent va exporter vers d'autres agents.

Le langage permet de déclarer des agents individuels, ou des classe d'agents, c'est à dire un ensemble d'agents avec mêmes comportements et propriétés.

Les agents sont regroupés en sociétés et les sociétés peuvent être regroupées en d'autres sociétés. Les sociétés en ALBERT II n'ont pas d'actions ou de composants d'états. Ils ne servent qu'à structurer le système en sous systèmes.

4.4 Les différents types de contraintes

Le langage ALBERT II propose une série de contraintes pour restreindre les comportements permis d'un agent. Ces contraintes portent aussi bien sur les actions sous la responsabilité d'un agent que sur les valeurs d'état permises à un moment donné du cycle de vie d'un agent.

4.4.1 Les contraintes de base (Basic constraints)

Ce type de contraintes sert à donner les relations de dépendance qui peuvent exister entre les composants d'état d'un agent, mais aussi à donner des valeurs initiales des composants d'état d'un agent.

Valeurs initiales des composants d'état

C'est une contrainte qui spécifie parmi les valeurs permises d'un composant d'état d'un agent, celle qu'a le composant au début du cycle de vie de l'agent.

Composants dérivés (Derived components)

Ce type de contraintes sert à spécifier les valeurs de composants d'état d'un agent qui peuvent être déduites des valeurs des autres composants d'état. Elle servent à montrer les relations de dépendance qui peuvent exister entre composants d'état.

4.4.2 Contraintes déclaratives (Declarative constraints)

Nous avons sous cette dénomination trois types de contraintes : les contraintes sur les états d'un agent(**state behaviour**), les contraintes de composition d'action(**action composition**) et les contraintes sur les durées d'actions (**action duration**)

State behaviour

Ce type de contrainte sert à spécifier les propriétés qui devront être vérifiées chaque fois qu'une action sera en cours d'exécution. Si aucune action n'est mentionnée, cela signifie que cette propriété sera vérifiée durant tout le cycle de vie de l'agent. L'action visée est mise entre crochets, et les propriétés qui devront être vérifiées chaque fois que l'action est en cours d'exécution sont directement mise à côté des crochets.

Action Composition

ce type de contrainte permet à l'analyste de considérer les actions à plusieurs niveaux de granularité. Les contraintes de composition d'action expriment comment des actions peuvent être définies en terme d'actions plus élémentaires. Les actions élémentaires peuvent être synchronisées de sept façons différentes.

- **Composition séquentielle**
  Nous avons un certain nombre d'actions élémentaires a_i qui composent séquentiellement une action globale A. Dans une compostion séquentielle, le début de d'une occurrence de a_1 commence en même temps que l'action globale A, et une occurrence de a_{i+1} commence t_i unités de temps après la fin de a_i avec $1 \leq i < n$. La fin de a_n doit correspondre exactement à la fin de l'occurrence de l'action globale A. Si un t_i est nul, les actions a_i et a_{i+1} se touchent.

- **Composition répétitive**
  Sous ce type de contrainte, nous avons une action globale qui est est composée de plusieurs occurrences successives, plus élémentaires d'une même action.

- **Composition parallèle**
  La composition parallèle apparait au cas nous avons une action globale A, composée d'actions élémentaires, a_1,a_1,a_2,a_3,...,a_n. Une occurrence de A commence en même temps qu'une occurrence a_i qui commence le plus tôt, et la fin d'une occurrence de A correspond à la fin de l'occurrence a_i se terminant le plus tard.

- **Composition simultanée**
  Pour chaque occurrence d'une action globale A, correspond une série d'actions plus élémentaires a_i ($1 \leq i \leq$ n). Chaque occurrence a_i commence et se termine en même temps que l'occurrence A.

- **Composition d'actions qui commencent en même temps**
  Nous avons une action globale qui est composée d'une série d'actions qui commencent en même temps que l'action globale. Cette dernière se termine en même temps que l'occurrence a_i qui se termine le plus tard.

- **Composition d'actions qui finissent en même temps**
  Nous avons une action globale A composée d'un certain nombre d'actions a_i. Ces actions se terminent en même temps que l'action globale et celle-ci commence en même temps que l'occurrence a_i qui commence le plus tôt.

- **Composition alternative**
  Chaque occurrence de l'action globale est composée d'une occurrence d'une action plus élémentaire qui commence et qui se termine en même temps que l'action globale

Nous pouvons remarquer dans les définitions données ci dessus une nette ressemblance avec les opérateurs entre intervalles dans la logique temporelle des intervalles. Les intervalles sont tout simplement les durées des différentes actions.

Action duration

Le langage ALBERT II permet à l'analyste de spécifier des contraintes sur les durées d'exécution d'une action. Ces contraintes de durée peuvent spécifier des durées exactes, après lesquelles une action devra impérativement être terminée, des durées au plus tôt, ou au plus tard.

4.4.3 Les contraintes opérationnelles(Operational constraints

Les contraintes opérationnelles sont des contraintes qui spécifient les conditions sous lesquelles les états d'un agent changent de valeur par rapport au cycle de vie d'un agent, les conditions qui doivent être vérifiées au niveau des valeurs des composants d'état pour qu'une action puisse avoir lieu. Ces contraintes sont des contraintes de **préconditions**, d'**effet d'actions** et de **déclenchements**.

Préconditions

Les préconditions définissent les conditions qui doivent être vérifiées sur les valeurs d'un ou de plusieurs composants d'état pour qu'une action puisse

être exécutée. Ces conditions doivent être vérifiées juste à l'état qui précède le l'exécution de l'action. L'action considérée doit être une action qui est sous la responsabilité de l'agent, et non une action importée.

Effet d'actions

Les effets d'actions définissent les valeurs des composants d'état d'un agent au début de l'exécution d'une action(pre-effet) ou à la fin de l'exécution d'une action (post-effet). Ces actions peuvent être des actions internes à l'agent, mais aussi des actions importées.

Déclenchement

Les contraintes de déclenchement(Triggering) servent à spécifier une condition sur un composant d'état qui permet le déclenchent d'une action. Les contraintes de déclenchement définissent une période de temps pendant laquelle un composant d'état doit rester avec une certaine valeur pour qu'action commence à être exécutée.

4.4.4 Les contraintes de coopération

Ces contraintes servent à définir les conditions pour que les agents puissent interagir. Ces conditions d'interaction sont de deux sortes :

- Conditions pour qu'un agent puisse montrer un composant d'état ou une actions à d'autres agents : contraintes de state information et action information.
- Conditions pour qu'un agent puisse percevoir un composant d'état ou une action de la part d'un autre agent : contraintes de state perception et action perception.

La percetion concerne les actions ou composants d'état externes à un agent, alors que l'information concerne les actions ou composant d'états internes. Les communications entre agents peuvent être fiables, ou non fiablse, selon les opérateurs de coopération utilisés dans les contraintes.

4.5 Conclusion

En plus des concepts permettant de modéliser les contraintes temps réel, le langage ALBERT II contient des concepts permettant de modéliser la coopération entre agents. Nous avons vu que la coopération est un concept important aussi bien pour les systèmes distribués que temps réel. C'est pourquoi en utilisant le langage ALBERT II pour spécifier de tels systèmes, on a la possibilité de définir ces contraintes pendant l'étape de spécification.

Chapitre 5

Un langage de modélisation orientée objet : UML

5.1 Introduction

Le monde informatique vit depuis un certain temps au rythme des «objets». Ceci se remarque principalement au niveau de la programmation objet avec des langages comme JAVA ou C++. Le paradigme objet a pris l'avantage sur le paradigme structuré car les applications deviennent plus facile à maintenir.

Dans ce chapitre, nous allons aborder la modélisation objet, avec le langage UML[1]. Ce langage est en train de devenir un standard de modélisation orientée objet. Nous commencerons le chapitre en abordant les concepts généraux de la plupart des méthodes de modélisation objet et en deuxième partie du chapitre nous aborderons le langage UML.

[1] Unified Modeling Language

5.2 Principaux concepts de la modélisation orientée objet

5.2.1 Notion d'objet

Une des principales tâches lorsque on modélise une application avec une approche orientée objet, est de pouvoir identifier les différents objets du domaine d'application qui peuvent présenter un intérêt pour le reste du processus.

Un objet est une entité du domaine de l'application, qui est identifiable et différente des autres objets. Dans le domaine académique par exemple, chaque **étudiant**, chaque **cours** est objet. Un objet est défini par les éléments suivants[16] :

- **état** : qui est décrit par l'ensemble de ses attributs qui forment le vecteur composants d'états de l'objet.
- **comportement** : qui est décrit par les opérations de l'objet. Ces dernières expriment le comportement qu'aura l'objet en fonction de l'état dans lequel il se trouve. Le déclenchement des opérations est rélisé grâce aux messages que l'objet reçoit de la part d'autres objets.
- **identité** : chaque objet a une identité qui le distingue des autres objets et cela indépendamment de de son état et de son comportement.

5.2.2 Notion de classe

Une classe fait référence à un ensemble, à une collection d'objets qui ont des propriétés communes, c'est à dire qui partagent une structure(attributs) et un comportement(opérations). Il serait redondant que ce soit au niveau de la spécification ou de l'implémentation de traiter ces objets individuellement. Tous les objets d'un même type, par exemple tous les étudiants d'une classe, ont une même structure (nom, prénom, date de naissance, cours suivis...). On voudrait alors pouvoir décrire un étudiant de façon général, une fois pour toutes et ensuite utiliser cette déclaration comme un type de variable. Ce type est appelé une **classe**.

Attributs

Un attribut d'une classe représente une donnée caractéristique d'une classe. Un attribut d'une classe peut seulement contenir une valeur simple (comme un

entier ou un texte), mais pas un objet au sens de la modélisation objet. Le nom d'un étudiant est un attribut légal. Le cours où il est inscrit n'est pas un attribut légal si les cours sont des objets.

Par exemple en considérant le cas de la classe étudiant, les attributs de cette classe sont par exemple le *nom*, *prénom*, *âge* etc.

Opérations

Une opération est une action qui peut être appliquée à une classe. Si on considère la classe *fenêtre*, on peut prendre comme opérations *ouvrir*, *fermer* la fenêtre.

Une opération a toujours un objet implicite en argument, c'est l'objet auquel il faut envoyer un message pour exécuter cette opération. Deux opérations de deux classes différentes peuvent avoir le même nom. Deux méthodes avec le même nom peuvent être définis de façons différentes par deux classes différentes.

Associations

Dans la section sur les classes, nous avons mentionné que les attributs ne peuvent être d'autres objets. Mais comment faire pour relier un objet à un autre ? Par exemple comment modéliser le fait que un étudiant *suit* un cours, étant donné que étudiant et cours sont tous des classes d'objets. Nous devons utiliser une **association** pour lier les deux.

Une association est un moyen pour représenter une connexion conceptuelle entre classes. On peut aussi parler de relation entre classes. Comme pour une classe ou un objet, une association possède un nom. Une association a deux sens, même si elle relie une classe à elle-même(association unaire). Un étudiant est inscrit à un ou plusieurs cours, et les cours ont des membres qui sont des étudiants.

5.2.3 Héritage

L'héritage est la relation entre un concept plus général et un concept plus particulier. L'héritage permet de faire une factorisation des attributs et opérations qui sont partagées entre classes et de particulariser les classes qui ont

d'autres propriétés additionnelles. Ainsi la relation d'héritage permet à une classe d'objets (sous-classe) de s'approprier les attributs et les opérations définis par une classe générale (super-classe). Cette façon de faire trouve toute sa puissance au niveau de l'implémentation, une méthode définie au niveau de la classe mère est automatiquement propagée vers les classes filles, comme si elle avait été déclarée localement.

L'héritage peut être *simple* ou *multiple*. L'héritage simple permet à une sous-classe d'avoir uniquement une seule super classe, alors que l'héritage multiple permet à une sous-classe d'avoir plusieurs super-classes.

Nous pouvons donner l'exemple d'une université, dans laquelle il y a des étudiants et des employés. Considérons le cas des doctorants, qui sont des étudiants, mais qui en plus dispensent des travaux pratiques ou des exercices aux autres étudiants. Nous pouvons considérer que la classe doctorant hérite à la fois de la classe étudiant mais aussi de la classe employé.

5.2.4 Agrégation

L'agrégation est un type d'association un peu particulier. L'agrégation est une relation qui permet de décrire un objet complexe en considérant les différents objets élémentaires dont il est composé. L'exemple classique qu'on donne pour illustrer l'agrégation est celui d'une voiture. L'objet voiture est composé d'une carrosserie, d'un moteur, des pneus, etc. A première vue on peut être amené à confondre l'agrégation avec le mécanisme d'héritage. Dans l'agrégation, il n'y a pas de propagation des propriétés de l'objet vers les différentes parties de l'objet composite. Ce n'est pas parce la voiture est rouge, que le moteur sera aussi rouge.

5.2.5 Quelques propriétés des associations

Cardinalité

La cardinalité décrit le nombre d'objets différents d'une classe pouvant être liés dans une association avec une autre classe d'objets. Par exemple un cours est suivi par plusieurs étudiants (1 à N) et un étudiant suit 1 à N cours (pour être considéré comme étudiant, il faut qu'il suive au moins un cours).

Attribut d'association

On a parfois besoin d'attacher de l'information à une association, par exemple la note qu'un étudiant a eu à un cours où il est inscrit. Cette information ne peut pas toujours être mise dans une des deux classes de l'association, principalement quand la cardinalité est multiple aux deux bouts de l'association, par exemple le cas d'un étudiant qui suit plusieurs cours et a donc plusieurs notes, et symétriquement un cours est suivi par plusieurs étudiants. On attache donc l'attribut à l'association elle-même.

5.3 Le langage UML

5.3.1 Introduction

Depuis une décennie, les méthodes orientées objet, que ce soit au niveau de l'analyse, de la conception ou de l'implémentation prennent de plus en plus d'importance. Les langages objets comme C++, Java, Eiffel ne sont plus à présentés. Cet engouement pour cette technologie a provoqué depuis la deuxième moitié des années 80, une multitude de langages de modélisation. Cette course vers les méthodes de modélisation était dictée par le souci, face au succès remporté par les langages de programmation orientés objet, de pouvoir conduire entièrement des projets en suivant uniquement l'orientation objet, depuis l'analyse jusqu'à l'implémentation.

C'est dans cette perspective que plusieurs langages de modélisation orientés objets sont devenus très populaires. Les langages OOD[5], OMT[18] furent parmi les premiers à être lancés, et ils ont remporté un vif succès auprès des producteurs de logiciels et de systèmes d'information.

Ces méthodes ont continué à évoluer chacune de son côté, même s'elles avaient des ressemblances les unes aux autres. Un besoin d'un standard commence à se faire sentir vers le début des années 90. C'est ainsi que l'OMG [2] est intervenu pour proposer que les auteurs puissent proposer des normes de modélisation afin de dégager un standard. Grady Booch, Yvar Jacobson, et Jim Rumbaugh, les pères des trois méthodes les plus populaires : , Booch, OOSE et OMT, ont décidé alors d'unir leurs efforts afin de créer un standard de modélisation. Le premier document en rapport avec cette méthode est sorti en octobre 1995, il s'agissait de la version 0.8, qui a été largement diffusée, pour

[2] Object Management Group

recueillir des suggestions auprès de la communauté des utilisateurs. Ces suggestions furent prises en compte dans la version 0.9 qui est sortie en Juin 1996, mais davantage dans la version 0.91 qui est disponible depuis Octobre 1996. Des géants[3] de l'industrie du logiciel apportèrent leur contribution à la définition de la version 1.0, qui fut remise à l'OMG pour standardisation. Nous en sommes actuellement à la version 1.3 et le langage connaît une popularité sans précédent dans l'histoire de la modélisation objet. Le langage permet de modéliser un système selon plusieurs points de vue et ceci à l'aide d'une série de diagrammes que nous énumérons ci dessous.

1. **diagrammes de cas d'utilisation** : qui montrent une vue de haut niveau sur les différentes façons dont le système interagit avec les utilisateurs.
2. **diagrammes de classes** : qui expriment la structure statique en terme de classes et de relations entre classes.
3. **diagrammes d'activités** : représentation du comportement d'une opération de haut niveau, en montrant la l'enchaînement des actions plus élémentaires.
4. **diagrammes d'instances d'objets** : représentation des objets et de leurs relations, correspond à un diagramme de collaboration simplifié, sans représentation des envois de messages.
5. **diagrammes de collaboration** : représentation spatiale des objets, des liens et des interactions entre les objets.
6. **diagrammes de séquences** : représentation temporelle des objets et de leurs interactions.
7. **diagrammes de déploiement** : représentation du déploiement des dispositifs matériels qui interviennent dans un système.
8. **diagrammes des composants** : représentation des composants des unités de code exécutables qui sont assemblées pour former des applications.
9. **diagrammes d'états transitions** : représentation de la dynamique du système, c'est à dire du comportement des classes en considérant l'évolution des différents états d'une classe.

Les lecteurs remarqueront que ce chapitre sur le langage UML n'est pas un pamphlet commercial, ce ne pas non plus un comparatif. C'est un exposé objectif et introductif sur UML, son histoire, ses aspects techniques, son identité de langage de modélisation orienté objet. Bref un exposé sur ce que permet ou ne permet pas le langage UML.

Le langage contient beaucoup de concepts, nous essayerons de donner les concepts qui nous semblent être les plus importants tout en insistant sur ce qu'on peut en faire.

[3]Oracle, Microsoft, Hewlett-Packard, DEC, i-Logix, Intellicorp, IBM, ICON Computing, MCI systemhouse, Rational Software et Unisys

5.3.2 Les paquetages

C'est un mécanisme pour organiser des éléments de modélisation dans des groupes. Les paquetages peuvent être emboîtés les uns dans les autres. Un système peut être vu comme un seul paquetage abstrait qui contient tout le reste. Un paquetage peut contenir d'autres sous-paquetages sans limites de niveaux.

Une classe dans un paquetage peut apparaître dans un autre sous la forme d'un élément importé au travers d'une relation de dépendance entre paquetages. Une dépendance entre deux paquetages signifie qu'au moins une classe du paquetage client utilise les services offerts par une classe du paquetage fournisseur.

L'architecture générale du système est exprimée par la hiérarchie de paquetages et par le réseau de relations de dépendance entre paquetages.

UML définit l'opérateur " : : " pour la syntaxe de la dépendance. L'expression *nompaqu* :: *nomclasse* désigne la classe *nomclasse* est définie dans le paquetage *nompaq*. Comme les classes, les paquetages possèdent une interface et une réalisation. Chaque élément contenu par un paquetage possède un paramètre qui signale si l'élément est visible ou non à l'extérieur du paquetage. Les valeurs prises par le paramètre sont publiques ou privées. Dans le cas des classes, seules celles indiquées comme publiques apparaissent dans l'interface du paquetage qui les contient ; elles peuvent alors être utilisées par les classes membres des paquetages clients.

5.3.3 Diagramme des cas d'utilisation

Les cas d'utilisations décrivent le comportement du système du point de vue utilisateur sous la forme d'actions et de réactions, c'est à dire qu'il modélise des fonctions opérationnelles du système déclenchée par un acteur externe au système. C'est une vue de haut niveau sur la façon dont le système sera utilisé. Ce genre de diagramme permet de mettre en place et comprendre les besoins du client, en terme des fonctions primaires du système à mettre en place. Les use cases ont été introduit par Ivar Jacobson[12].

Dans le diagramme, interviennent deux éléments : les acteurs et les cas d'utilisation. L'acteur représente un rôle joué par une personne ou un autre système qui interagit avec le système en cours de modélisation. Les acteurs directs sont ceux qui sont responsables de son exploitation.

Les cas d'utilisation permettent de structurer les désirs des clients. Ils se dé-

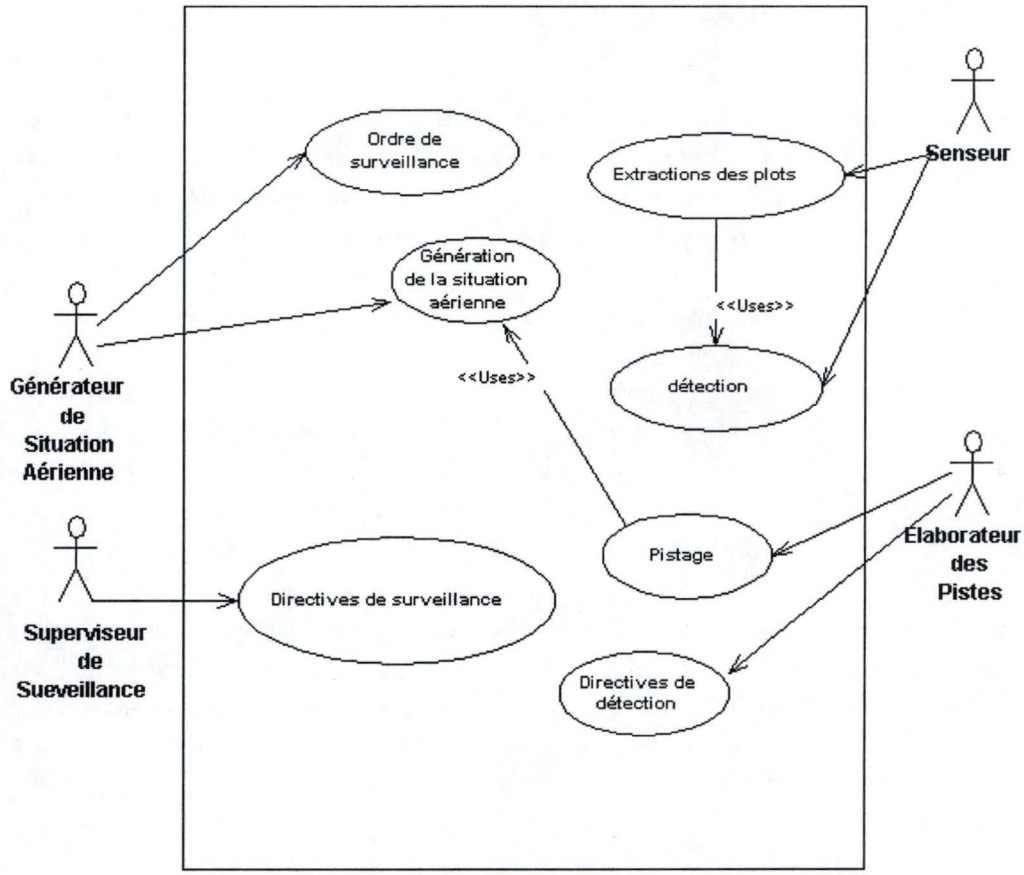


FIG. 5.1: Surveillance

terminent en prenant tous les acteurs et en essayant de déterminer les fonctions opérationnelles sous la responsabilité de chaque acteur.

Etant donné que c'est un formalisme qui permet de structurer les besoins des utilisateurs, il permet d'impliquer les futurs utilisateurs dans le processus de détermination des besoins du système.

Les cas d'utilisation peuvent présenter des relations entre eux. Il en existe principalement trois sortes.

Relation de Communication

Elle exprime la participation d'un acteur dans un cas d'utilisation, et cette dernière est signalée par une flèche entre l'acteur et le cas d'utilisation. Le sens de la flèche indique l'initiateur de l'interaction.

Relation d'utilisation

Cette relation entre cas d'utilisation indique qu'une instance du cas d'utilisation source comprend également le comportement décrit par le cas d'utilisation destination.

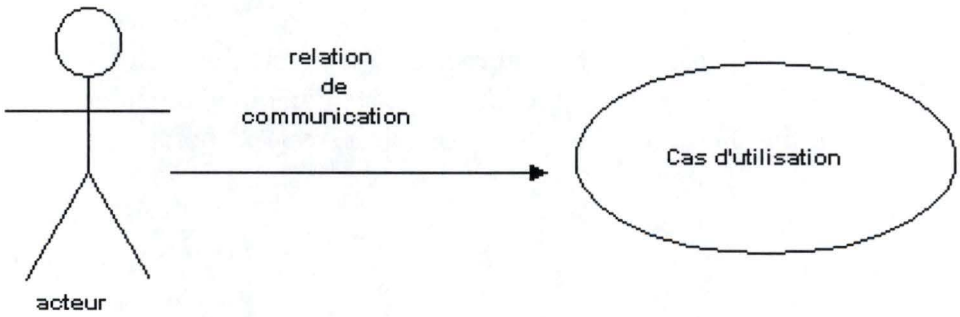


FIG. 5.2: Un diagramme de cas d'utilisation

Relation d'extension

Cette relation entre cas d'utilisation montre que le cas d'utilisation source étend le comportement du cas d'utilisation destination.

Cette description de ce que doit faire le système doit nécessairement éviter d'aller jusqu'à décrire le comportement interne du système. Il s'agit seulement de décrire l'interaction entre un acteur et le système. A ce stade on s'occupe de décrire le «quoi» mais pas le «comment».

Le fait d'avoir identifié un cas d'utilisation du système apporte une connaissance sur l'interface externe du système vis-à-vis des utilisateurs externes. Mais il est aussi nécessaire pour être plus précis de donner une description un peu plus détaillée. Cette dernière peut être donnée sous forme d'un petit texte ou sous la forme d'un diagramme de séquences.

5.3.4 Diagramme de classes

Ce type de diagrammes sert à capturer la structure statique du système, en se concentrant sur les différentes classes d'objet du domaine d'application et les relations entre elles. On trouve les éléments suivant dans ces diagrammes :

- différentes classes d'objets, leurs structures, propriétés et opérations.
- associations, agrégations, ainsi que les relations d'héritage.
- les multiplicités.
- les différents rôles.
- les contraintes sur les attributs, sur les rôles.

Une classe est représentée par un rectangle qui est divisé en trois compartiments. Le premier est celui du nom de la classe, dans le deuxième se trouvent les attributs de la classe, et dans le troisième les opérations relatives à cette classe.

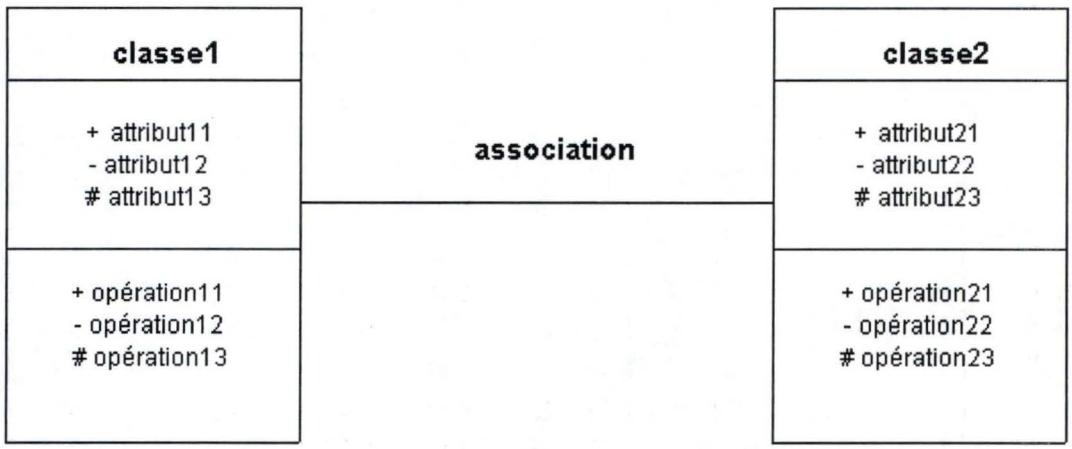


FIG. 5.3: Représentation d'une association entre deux classes en UML

Les attributs comme les opérations se divisent en trois ensembles selon le niveau de visibilité :

- les attributs et les opérations privés, c'est à dire qui sont seulement visibles à l'intérieur de la classe, c'est le signe (-) placé devant le nom de l'attribut ou de l'opération qui sert à spécifier ce niveau de visibilité.
- les attributs et les opérations publiques c'est à dire qui sont visibles à tous les clients de la classe, c'est le signe (+) placé devant le nom de l'attribut ou de l'opération qui spécifie le niveau de visibilité publique.
- les attributs ou les opérations protégés c'est à dire qui sont visible seulement par les sous classes de la classe. C'est le symbole(#) placé devant le nom de l'attribut ou de l'opération qui sert à spécifier ce niveau de visibilité.

La déclaration d'un attribut s'effectue de la manière suivante :

$$NomAttribut : TypeAttribut = ValeurInitiale$$

valeur initiale peut être absente si elle n'est pas connue.
et celle d'une opération se fait de la manière suivante

$NomOperation(NomArgument : TypeArgument, ...) : TypeRetour$

$TypeRetour$ peut être absent si l'opération ne retourne rien.

5.3.5 Diagramme d'activités

Les diagrammes d'activités servent à donner une représentation du comportement d'une fonction opérationnelle en terme d'actions plus élémentaires.

Ils représentent l'état d'exécution de cette fonction opérationnelle sous la forme d'un flot de contrôle ou d'un déroulement d'étapes. Les transitions entre activités peuvent être gardées par des conditions booléennes. Ces gardes servent à valider le déclenchement d'une activité.

Il est également possible de modéliser avec les diagrammes d'activités la notion de synchronisation entre activités. On utilise pour ce faire une barre de synchronisation.

5.3.6 Diagramme d'instances d'objets

Un diagramme d'objets est une version d'un diagramme de classes. Comme ce dernier il montre la structure statique du système mais cette fois en terme d'instances d'objets. Dans un diagramme d'objets, les associations qui servent à mettre en liaison deux ou plusieurs classes d'objets sont remplacées par des liens entre objets.

Nous pouvons aussi affirmer comme nous le verrons dans les paragraphes à venir, que les diagrammes d'objets sont une version simplifiée des diagrammes de collaboration. Dans le cas des diagrammes d'objets, les différents messages que les objets s'envoient ne sont pas représentés.

5.3.7 Diagrammes de collaboration

Les diagrammes de collaboration sont une extension des diagrammes d'objets, dans lesquels on montre l'interaction entre un groupe d'objets. Dans ce

genre de diagrammes, le temps n'est pas explicitement pris en compte dans la modélisation des interactions. Les messages entre les différents objets du diagramme portent des numéros pour exprimer comment ils vont s'enchaîner les uns des autres. Il faut que cette objet ait des opérations appropriées pour pouvoir répondre au message reçu.

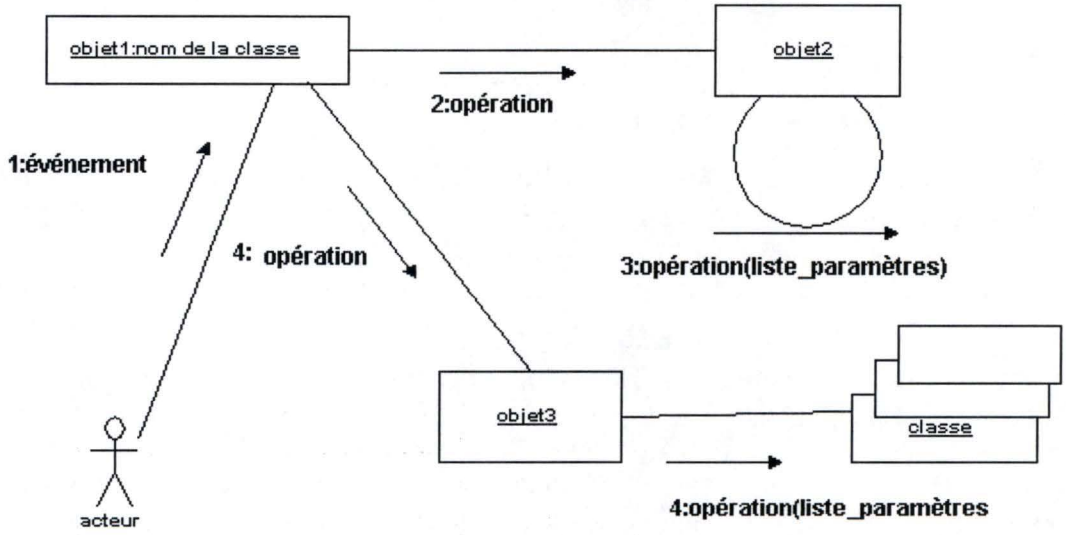


FIG. 5.4: Un diagramme de collaboration

Des contraintes peuvent apparaître sur des objets ou des liens qui sont créés ou détruits au cours d'une interaction. Ils porteront une contrainte «nouveau» ou «détruit». La contrainte «transitoire» est mis sur des objets ou des liens qui sont créés, puis détruits au sein d'une même interaction.

5.3.8 Diagramme de séquence

Dans ce type de diagramme, on modélise la dynamique du système en mettant en avant l'aspect temporel. Comme dans les diagrammes de collaboration, il s'agit de représenter les interactions entre les objets, tout en insistant sur la chronologie des envois de messages. Dans ce modèle on a des événements, qui déclenchent des actions et l'exécution de ces actions va générer d'autres événements.

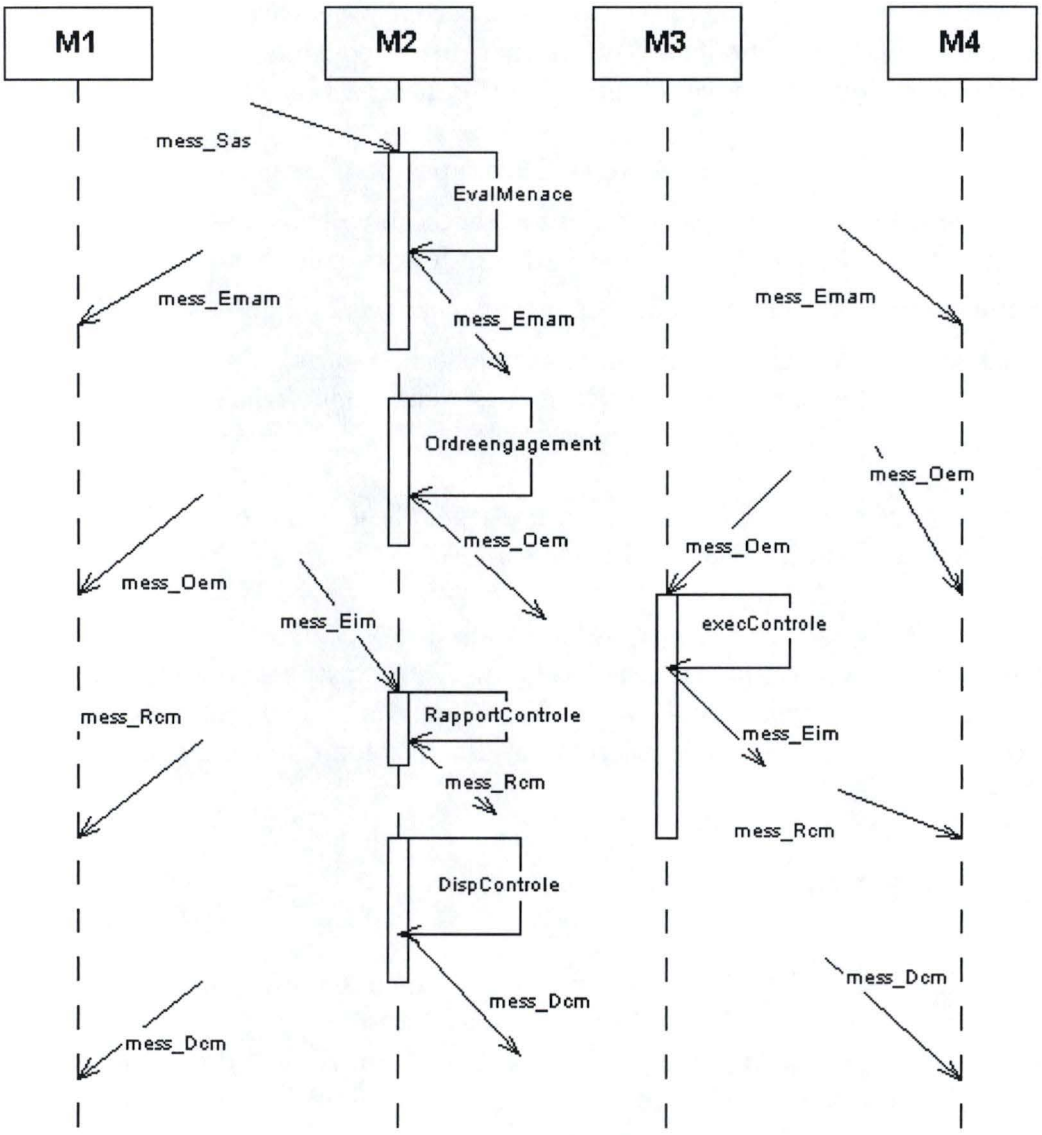


FIG. 5.5: Exemple d'un digramme de séquence

Les objets étudiés sont mis sur une seule ligne horizontale et pour chaque objet, on lui associe une barre verticale en pointillé appelée **ligne de vie** de l'objet. Cette ligne représente un axe de temps, qui est dirigé du haut vers le bas. Les messages sont représentés par des flèches horizontales orientées de l'émetteur vers le destinataire.

La représentation des périodes d'activité des objets est réalisée à l'aide de bandes rectangulaires le long des lignes de vie des objets.

Dans le diagramme, on peut indiquer les branchements conditionnels par du pseudo-code placé le long de la ligne de vie ou alors entre crochets sur le

message à conditionner. De même, on peut placer en pseudo code une boucle d'itération (while) pour modéliser une structure de contrôle permettant de répéter une opération jusqu'à ce qu'une condition ne soit plus remplie.

Le diagramme de séquence distingue deux types de messages :

- les **messages synchrones** sont des messages pour lequels l'émetteur est bloqué et attend que l'objet appelé ait fini de traiter le message.
- les**messages asynchrones** où l'emetteur n'est pas bloqué.

Un message synchrone se représente par une flèche tandis qu'un envoi asynchrone est représenté par une démi-flèche. Les deux sont dirigées de l'objet emetteur vers l'objet destinataire du message.

Un objet peut s'envoyer lui même un message. C'est le cas quand il existe une flèche d'envoi d'un message qui boucle sur la ligne de vie de l'objet.

Les diagrammes de séquence permettent aussi de spécifier les contraintes temporelles par des annotations textuelles. On indique l'instant d'émission d'un message à proximité du point de départ de la flèche et on peut par exemple spécifier le temps entre ce point et un autre point d'activation d'un autre message.

5.3.9 Diagrammes de déploiement

Les diagrammes de déploiement décrivent les éléments matériels. Ils montrent les choix de réalisation et indiquent la disposition physique des différents matériels qui entrent dans la composition d'un système, Il en résulte donc une vue globale du système implanté avec tous les éléments impliqués.

Les éléments matériels sont représentés par des cubes (noeuds). La nature du matériel peut être précisée par un stéréotype. Dans le diagramme, les noeuds sont connectés à l'aide d'un trait qui symbolise un support de communication dont la nature peut être précisée par un stéréotype.

Comme pour les classes et les objets, on différencie les noeuds et les instances de noeud par un nom souligné (pour les instances). On peut aussi préciser le nombre d'instance de noeud en mettant un nombre en bas à droite du cube. On peut indiquer la multiplicité des liaisons entre les noeuds.

5.3.10 Diagramme des composants

Un diagramme des composants montrent les unités physiques de code source et les unités exécutables qui sont assemblées pour former des applications. Les classes sont affectées à des composants fournissant des briques réutilisables pour la construction des applications. Ils montrent en quelque sorte les choix de réalisation.

Les relations de dépendance sont utilisées dans les diagrammes de composants pour indiquer qu'un composant fait référence aux services offerts par un autre composant. C'est cette dépendance qui montre les choix de réalisation. Une relation de dépendance est représentée par une flèche pointillée qui pointe du composant utilisateur vers le composant fournisseur.

5.3.11 Diagramme d'états-transitions

Un diagramme états-transitions modélise le comportement des objets d'une classe. Cette modélisation est sous forme d'un automate d'états finis. Les diagrammes états-transitions utilisés dans le langage UML s'inspirent largement des **Statecharts**[11]. Ces automates sont des automates avec hiérarchie, avec des propriétés de généralisation et d'agrégation. Cet automate sert à décrire les comportements possibles en réponse aux stimulis auxquels un objet est soumis ainsi que les actions qui en résultent.

Comme les objets qui nous entourent, un objet est à tout moment dans un certain état. L'état d'un objet à un moment donné est constitué des valeurs instantanées de ses attributs. Le passage d'un état à un autre s'effectue à l'aide des transitions qui elles mêmes sont déclenchées par des événements.

UML permet de modéliser les actions à exécuter quand on est dans un état donné, en y entrant ou en y sortant.

- *Entry* : action à exécuter dès qu'on entre dans l'état
- *Exit* : action que l'on exécute dès qu'on sort de l'état
- *On* :action interne provoquée par un événement qui ne provoque pas un un passage vers un nouvel état.
- *Do* : activité à exécuter quand un objet est dans un état donné.

  Par exemple si nous considérons notre diagramme états-transitions de la figure (FIG. 3.3) pour une radio-cassette, quand elle est dans l'état «tape», l'activité associée est de jouer une cassette si elle est aussi dans l'état «tape-in».

5.4 Conclusion

Ce chapitre sur le langage UML nous a permis de parcourir les concepts les plus importants du langage. Beaucoup de concepts n'ont pas été abordés, le langage continue d'évoluer et ne s'est pas encore stabilisé. A titre d'exemple, une version du langage appelée UMLRT : UML pour le Temps Réel vient d'être mis au point. Cette version s'inspire en grande partie des concepts du langage de modélisation ROOM.

Chapitre 6

Commentaires sur UML, ALBERT II et perspectives d'intégration des deux langages

6.1 Commentaires sur le langage UML

6.1.1 Introduction

Dans cette section nous allons présenter nos commentaires sur le langage UML. Nous donnerons notre avis sur certains concepts qui nous semblent être importants dans les différents phase de modélisation, mais en plus nous donnerons quelques exemples sur les incohérences et concepts qui ne sont pas bien définis.

Beaucoup de livres et articles sur UML ont été écrits à ce jour et deux catégories se dégagent parmi les auteurs. Les uns semblent ne rien trouver d'intéressant dans le langage, d'autres se sont fait des champions dans la publicité en faveur du langage et trouvent toutes les qualités imaginables à UML. Nous allons essayer de donner d'éventuels problèmes que nous avons pu trouver ou que nous avons trouver dans la littérature mais qui nous semblent être justifiés.

Premièrement il est difficile qu'un langage de modélisation soit appelé «standard», alors que sa sémantique est présentée en partie en langue naturelle ou avec des métamodèles. L'un des rôles d'un langage de modélisation dans un projet doit être de faciliter la communication entre les différentes équipes chargées de mener un projet à terme. Or si le langage est sans sémantique formelle,

si les concepts qu'il propose sont parfois incohérents entre eux, redondants, la communication s'en trouve profondément affectée : sans sémantique formelle il y a aucun moyen d'éviter l'ambiguïté des modèles représentés dans le langage. Les auteurs les plus critiques vont jusqu'à affirmer que UML n'est pas un langage de modélisation, mais bien une famille de langages de modélisation.

Ce manque de sémantique formelle fait qu'il est difficile d'analyser rigoureusement les résultats de l'élaboration des différents diagrammes et modèles. Les autres développeurs au sein d'une équipe, ou le client, peuvent avoir l'impression d'avoir compris ce que le concepteur d'un diagramme a voulu dire, mais peuvent en fait avoir eu une interprétation toute différente de celle souhaitée par ce dernier. Dans le meilleur des cas les divergences sont détectées assez tôt, quand les gens discutent et expriment leurs interprétations, ce qui n'est pas évident étant donné qu'il est difficile de raisonner sur le modèle[27] et déduire d'autres propriétés qui ne sont pas explicitement représentées. Remarquons que, même dans ce cas, les discussions qui s'engagent pour trouver celui qui a la bonne interprétation prennent du temps qui devrait être occupé à autre chose si le langage avait une sémantique formelle.

6.1.2 Diagrammes des cas d'utilisation

Le formalisme des Uses Case est formalisme qui facilite le dialogue avec les clients et la capture des besoins. Ils sont facilement compréhensible par les clients(au niveau des fonctionnalités du système et d'éventuelles interactions entre le système et les acteurs), Il s'agit donc d'un formalisme important. Même s'il se limite à énumérer les fonctions opérationnelles du système sans donner aucune autre information, le fait qu'il soit facilement compréhensible par toute personne sans formation particulière en informatique, en fait un outil indispensable pour pouvoir récolter et organiser les renseignements nécesssaire à la réalisation d'un système. L'autre qualité qu'on reconnaît aux cas d'utilisation est qu'ils peuvent bien servir en parallèle ou précéder des méthodes plus formelles.

Cependant des zones d'ombres subsistent quant à la sémantique de certains concepts. En effet la sémantique donnée aux deux relations qu'on rencontre dans les diagrammes des cas d'utilisation : les relations *uses* et *extends* n'est pas claire. Les deux relations sont expliquées seulement par une description informelle, respectivement par une sorte d'*«inclusion»* et d'*«extension»*, juste une traduction littérale.

D'autres questions concernant les cas d'utilisations restent toujours sans réponse. K. Berger[3] se demande par exemple s'il est possible d'avoir des cas

d'utilisation sans une relation de communication avec un acteur. Un tel cas d'utilisation contredirait en quelque sorte la philosophie qui est derrière les cas d'utilisation, comme concept servant à modéliser les interactions du système et des utilisateurs, mais pourrait être pratique pour modéliser les tâches internes au système qui peuvent être manipulées automatiquement et n'ont pas besoin d'acteur externe.

6.1.3 Diagrammes des classes

Un diagramme des classes est une notion qui est présente dans beaucoup de langages de modélisation orientés objets. L'exemple classique d'incohérence pour les diagrammes des classes est celle de la relation de composition.

D'après la documentation UML(UML Notation Guide)[24], la relation d'agrégation est définie de cette manière :

Composition : a form of aggregation with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it (i.e., they share lifetimes). Such parts can also be explicitly removed before the death of the composite. Composition may be recursive.

Cette explication contient une contradiction. D'une part elle précise qu'une fois une structure d'agrégation est créée, elle ne change pas, d'autre part elle précise une liberté d'ajouter ou d'enlever des liens.

6.1.4 Diagrammes de séquence

Des questions restent sans réponse sur la mise en oeuvre de ces diagrammes. En effet, comment peut-on représenter à l'aide de ces diagrammes deux, trois ou plusieurs messages concurrents entre deux objets ? Un diagramme de séquence exprime presque la même chose qu'un diagramme de collaboration et une transformation d'un diagramme est très facile. L'existence des deux types de diagramme n'est il pas une redondance ? Quand faut il utiliser dans un modèle les diagrammes de collaboration seuls, les diagrammes de séquences seuls ou les deux à la fois ?

Cependant malgré ces faiblesses un digramme de séquence est un formalisme important dans la modélisation d'un système étant donné qu'il constitue une façon de voir les cas d'utilisation, mais cette fois en se focalisant sur le

séquencement interne associé à un cas d'utilisation. Comme les diagrammes de séquences permettent une vision temporelle des interactions dans le système, ils peuvent servir à modéliser les contraintes temporelles. La représentation de ces dernières ne peut se faire qu'à l'aide d'annotations textuelles.

6.1.5 Diagrammes d'activités

Les diagrammes d'activités sont des diagrammes qui peuvent être utilisés à plusieurs niveaux d'abstraction : modéliser le processus entier ou pour spécifier la logique des opérations plus élémentaires. Un diagramme d'activités est important pour modéliser le flôt de contrôle relatif aux différentes fonctions opérationnelles du système.

Le formalisme en soi est un formalisme qui est proche des diagrammes états-transitions. La version 1.1[24] précise même que :

In addition the state machine formalism provides the semantic foundation for activity graphs, this mean that activity graphs are a special form of states machines.
Mais elle reste muette quant aux interactions qui doivent exister entre les deux types de diagrammes dans un modèle.

6.1.6 Diagrammes états-transitions

Les diagrammes états-transitions(statecharts) est un formalisme très connu et très important pour spécifier la dynamique d'une application. Dans le langage UML, on utilise des machines à états finis étendues et hiérarchisées, basées sur une variante des statecharts de Harel[11]. Cependant des questions méthologiques quant à leur utilisation restent ouvertes. Il faudrait que le manuel de référence[23] donne des directives méthodoloques, par exemple sur les liens sémantiques qui existent entre les diagrammes états-transitions, les diagrammes d'activités et les diagrammes de séquence. Pour le moment le «UML Notation Guide» nous dit seulement qu'à une classe correspond un diagramme états-transitions.

6.2 UML et le temps réel

Modéliser un système temps réel comme nous l'avons vu à la section 1.4, nécessite une prise en considération des contraintes de temps mais aussi la coopération entre entités du système. Dans la version actuelle du langage, il est possible d'indiquer dans un diagramme de séquence des contraintes sur les durées des actions par des annotations textuelles en se référant à des points précis sur la ligne de vie d'une entité.

Un autre type de diagramme qui est se prête bien à la modélisation des systèmes temps réel est le diagramme états-transitions à cause des ressemblences entre les formalismes que propose le modèle et le mode de fonctionnement de ces systèmes. En plus des descriptions mathématiques de ces diagrammes existent, ce qui éloigne toute ambiguité.

Une version temps réel du langage UML vient de voir le jour. La plupart des concepts qu'elle propose sont des formalismes qu'on trouve dans le langage ROOM. Cette version semble aussi baser ses modèles sur les diagrammes états-transitions. Cependant une extension de ces diagrammes serait indispensable pour pouvoir tenir compte des contraintes de temps entre les différents états et transitions d'un système. Cette extension serait par exemple une adaptation des diagrammes états-transitions en se basant notamment sur la théorie des automates temporarisés ou des réseaux de Petri temporarisés.

6.3 conclusion

UML n'est donc qu'un nouveau langage qui comporte encore assez bien d'imperfections, des erreurs de jeunesse, des oublis parfois impardonnables. La plupart des formalismes qu'on trouve dans UML existent dans d'autres langages de modélisation. Outre les langages de modélisation comme OMT, OOSE, certains aspects comme les diagrammes de classes, les diagrammes d'activités ne sont pas très éloignés respectivement du modèle entités-association ou des diagrammes de flux[4].

Le langage fournit également un grand nombre de techniques de description mais leur sémantique n'est pas définie précisement. Ceci a pour conséquence de donner l'impression que la notation unifiée est un langage facile, riche[1] alors

[1]le langage contient une multitude de concepts : le glossaire de notation fait à lui seul une soixantaine de pages.

que au fond les nombreux diagrammes qu'on dessine ne serviront peut être pas à grand chose dans les phases ultérieures de développment d'un système.

Les documents de référence, à savoir **UML Notation Guide**[24] et **UML Semantics**[23] contiennent quelques exemples isolés mais ils ne donnent pas un exemple complet d'une application complètement modélisée avec le langage et qui permettrait d'inspirer les utilisateurs du langage. Il est à remarquer que les deux documents sont difficiles à lire, surtout celui de la sémantique.

En fin de compte certains auteurs se demandent quelle importance a un lexique de représentations graphiques, en dehors de toute méthodologie. Les problèmes de positionnement des diagrammes entre eux dans un processus ne proviennent-ils pas d'une absence de méthodologie ?

6.4 Un modèle d'articulation des diagrammes UML dans un processus de modélisation

6.4.1 Introduction

Nous avons vu que la notation unifiée, n'est pas une méthodologie, mais un ensemble de notations et les manuels de référence ne précisent pas comment les agencer dans un projet. Cependant en analysant les aspects techniques de chaque diagramme, il est possible de trouver à quelle étape du processus il se prête le mieux, reste à déterminer sa relation avec les autres diagrammes dans le processus.

Nous allons dans cette section essayer de donner un modèle d'articulation des différents types de diagrammes dans un projet, c'est à dire essayer de placer chacun des diagrammes du langage UML, dans un des niveaux d'un processus de développement en cascade.

6.4.2 Un exemple d'un agencement des diagrammes UML dans un projet

Nous savons que le modèle en cascade[7] du processus de développement d'une application informatique comprends quatre étapes : analyse, conception, l'implémenation et tests.

1. Le point de départ du processus est donné par besoins exprimés par les clients. Ces besoins sont en général non structurés, ambigus, et contradictoires. La réalisation des cas d'utilisation permet alors de structurer ces besoins et de dégager les éléments essentiels pouvant servir dans les phases ultérieures. Dans un projet réel, il se peut qu'on arrive à un nombre très élevé de cas d'utilisation en raison des différentes variantes d'un même cas d'utilisation. Il faut alors structurer ces cas d'utilisation en établissant d'éventuelles relations entre eux Ce processus est à réaliser itérativement, en montrant aux clients les cas d'utilisation déjà produits et ainsi on pourra connaître s'il y a une cohérence entre les besoins non structurés et le modèle des cas d'utilisation qu'on essaie de mettre au point.

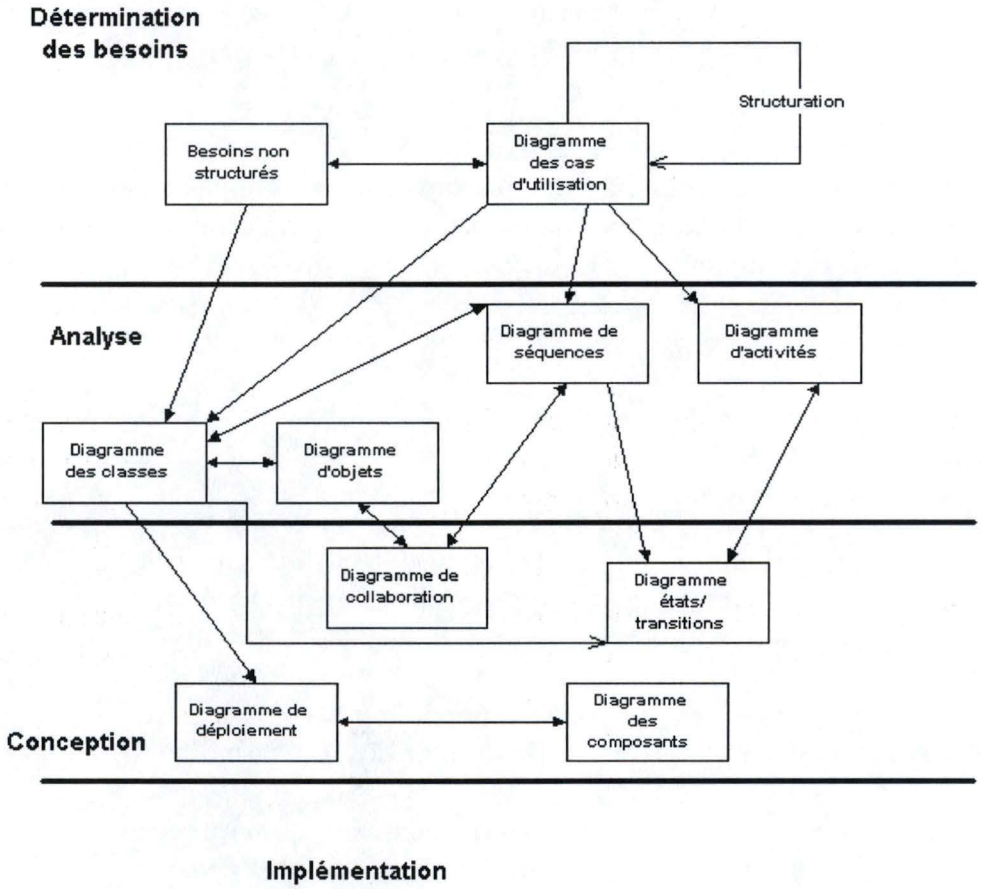


FIG. 6.1: Articulation des diagrammes UML

2. Le modèle des cas d'utilisation va alors servir de base pour la réalisation des autres diagrammes. Pour chaque cas d'utilisation, il faut réaliser **un** ou **plusieurs** diagrammes de séquence ainsi qu'un diagramme d'activité pour spécifier les activités en flots d'opérations plus élémentaires.
3. A partir des cas d'utilisation, mais aussi des objets identifiés dans la réalisation des diagrammes de séquence, on met au point les diagrammes

de classes et d'instances d'objets. Ces deux diagrammes sont au coeur de la phase d'analyse. En modélisation orienté objet, la réalisation des diagrammes de classes est une opération délicate étant donné qu'il faut déjà commencer à penser à la structure statique de l'application.

4. Les diagrammes d'instances d'objets et de séquence vont servir de base à la réalisation des diagrammes de collaboration, avec les flux de messages entre objets et les rôles des objets participants.
5. A partir des diagrammes de séquence, d'activité et de classes on construit les diagrammes d'états-transitions. Là on est déjà dans la phase de conception. La conception physique est réalisée avec les diagrammes des composants et les diagrammes de déploiement. Ils sont réalisés à partir des diagrammes des classes et permettront de montrer comment les différentes opérations vont être regroupés ensemble en composants logiciels devant être répartis sur les composants matériels.

Dans un projet réel, ces diagrammes ne sont peut-être pas tous nécessaires. C'est aux responsables du projet de déterminer les diagrammes qui donneront des modèles pertinents.

6.5 Commentaires sur le langage ALBERT II

Le langage ALBERT II, comme nous l'avons vu au chapitre 4 est un langage formel avec sémantique formelle et le langage été validé avec des cas concrets. A la différence des autres langages de spécification, il introduit le concept de coopération qui est très important pour la modélisation des applications distribuées ou temps réel. En effet, de telles applications sont implantées sous forme de processus séquentiels et qui doivent coopérer entre eux. Il est donc judicieux que cette coopération puisse être mise au jour dès la phase d'analyse.

Le langage permet aussi d'exprimer des contraintes temporelles. L'approche utilisée est celle des instants, mais derrière les contraintes de composition d'actions se cache aussi une logique temporelle des intervalles.

Le concept de base, à savoir le formalisme d'**agent** est proche du concept de **classe** ou d'objet dans les langages de modélisation orientés objet. Le langage Albert dans sa version actuelle ne permet la notion d'héritage entre agents ou société d'agents.

Dans le cas où des agents ont des comportements ou des propriétés similaires, on en fait une classe d'agents. Mais au cas où certains agents intègrent

des comportements ou des propriétés particuliers, c'est un peu lourd de refaire le travail de spécification de ces agents plusieurs fois. Il serait donc nécessaire que le langage soit doté d'un mécanisme d'héritage entre agents et entre sociétés, ceci permettait d'avoir des spécifications beaucoup plus structurées.

Dans une spécification Albert, le concepts de société n'a pas de propriétés propres. Une société est spécifiée par les spécifications des ses agents. Il serait peut être intéressant de pouvoir doter le concept de société de propriétés propres. Ceci servirait plus à accroître le dégré d'abstaction du langage. En effet une propriété définie dans une société appartiendrait à tout les agents de la société.

6.6 Est-il possible d'intégrer ALBERT II et UML

6.6.1 Introduction

Pour développer un système de façon rigoureuse, il est reconnu que l'utilisation des méthodes formelles est plus que recommandée surtout pour des systèmes critiques. L'avantage principal avec les méthodes formelles est qu'elles vous permettent de modéliser, de décrire ce que voulez avec une précision plus élevée par rapport à d'autres méthodes. Un modèle ne peut être interprété qu'une seule manière. Même si les bases mathématiques de la plupart de ces méthodes sont normalement accesibles à un analyste[2], ils ne sont pas toujours le bienvenu dans les entreprises qui leur préfèrent les méthodes semi-formelles.

C'est en partant de ce constat du manque de rigueur dû au manque formalisme des méthodes graphiques(semi-formelles) et du manque de convivialité des méthodes formelles, que des gens commencent à réfléchir à l'intégration des deux types de méthodes. Cette intégration permettrait de garder la formalité des méthodes formelles tout en accroissant l'utilisation des techniques formelles dans les entreprises.

Pour intégrer des méthodes, plusieurs approches peuvent être considérées. On peut intégrer des méthodes en considérant les concepts qu'elles ont en commun : restriction de certains concepts, renforcement, enrichissement d'une méthode par des concepts provenant de l'autre. Par exemple doter les langages formels d'un environnement graphique. On peut aussi chercher à établir les correspondances entre les deux méthodes. On peut par exemple remplacer le langage naturel utilisé pour décrire la sémantique d'un langage graphique par une notation formelle. Beaucoup de projets par exemple s'attèlent à définir une sémantique formelle du langage UML[20],[6] en utilisant le langage Z.

L'autre approche consiste à étendre une notation formelle en la dotant des caractéristiques orientés objets. L'exemple qui est le plus souvent cité est celui du langage Object-Z[13], mais l'inconvénient est que la plupart du temps on perd les outils liés à la technique formelle de départ.

Nous allons essayer de regarder dans les sections suivantes quelques voies à suivre pour doter le langage ALBERT II des caractéristiques graphiques en s'inspirant du langage UML. Dans la section suivante nous regarderons une façon de faire coexister dans un projet les langages ALBERT II et UML et ainsi profiter en même temps des avantages respectifs des deux langages.

[2]théorie des ensembles, logique des prédicats, algèbre élémentaire

6.6.2 Première approche

La première approche pour résoudre le problème de manque de formalisme des langages graphiques et le manque de convivialité des langages formels consisterait à développer des environnemnts intégrés qui permettraient à un analyste de spécifier formellement son système sans écrire de «mathématiques compliquées», mais en travaillant graphiquement. L'interface serait par exemple basée sur une notation graphique comme UML. Il faudrait alors définir des règles de dérivation permettant le passage du graphique au formel.

Cette approche présente des avantages étant donné qu'on gagne en convivialité tout en gardant l'efficacité des méthodes de spécification formelles. Ainsi il est possible de continuer à analyser la spécification avec des outils qui supportent le langage formel.

Cette voie peut être exploitée pour le langage ALBERT et l'environnement de modélisation serait basé sur une interface graphique basée sur un graphisme semblable aux digrammes états-transitions du langage UML. En effet il exite des ressemblances entre des concepts relatifs à ces diagrammes et certaines contraintes Albert telles que les contraintes de *préconditions*, *effet d'actions*, *contraintes de base* et *déclenchement*. Essayons de voir ceci plus concrétement.

Nous avons vu au chapitre 4 que le langage ALBERT II possède la notion de composant d'état (agent), ce qui est également le cas pour les digrammes états-transitions. Nous avons vu aussi que tout diagramme états-transitions possède un état initial, c'est à dire l'état dans lequel se trouve un objet quand il est crée. on peut alors déduire facilement les contraintes de base(initial valuation) à partir des états initiaux relatifs à un diagramme états-transitions.

Nous savons aussi que le passage d'un état à un autre est matérialisé dans un diagramme états-transitions par la survenance d'un événement. Les événements, les transitions, les états et les actions sont indissociables dans la description du comportement dynamique. En effet sur un diagramme états-transitions, lorsqu'on modélise la transition d'un état à un autre on représente aussi l'action à exécuter lorsque l'événement survient. L'action correspond à l'une des opérations déclarées dans la classe de l'objet destinataire de l'événement. On peut tirer de ces situations les contraintes de préconditions et de déclenchements. Pour cette dernière, il faut pour pouvoir le faire d'avoir défini, des gardes(ici elles doivent être temporelles) qui sont des conditions booléennes qui valident ou non le déclenchement d'une transition, donc l'action y associée lors de l'occurrence d'un événement.

Nous avons vu aussi que certaines actions sont exécutées lorque un objet

est dans un état donné, à l'entrée de l'état ou à la sortie de l'état. Les actions réalisées à la sortie de l'état permettent à l'objet de changer d'état, ce qui peut permettre de déduire les contraintes d'effets d'actions locales. ALBERT II va plus loin car une action sous la responsabilité d'un agent peut modifier l'état d'un autre agent.

Nous voyons donc qu'il est possible d'exploiter la ressemblance de certaines contraintes ALBERT II et des propriétés des diagrammes états-transitions pour la création d'un envirronement graphique de spécification qui générerait en partie des contraintes ALBERT II à partir d'un modèle graphique. Une extension de certains concepts dans ce type de diagrammes peut permettre de supporter d'autres types de contraintes. Je pense par exemple aux contraintes temporelles et de coopération.

Cependant, le langage ALBERT II est basé sur la logique temporelle temps réel et on trouve souvent dans une spécification des axiomes devant être vérifiés sur un état ou une action. Des problèmes sont à résoudre pour pouvoir exploiter cette voie notamment le passage d'une d'une représentation graphique vers des contraintes sous forme d'axiomes.

6.6.3 Deuxième approche

Dans un projet, souvent il n'est pas nécessaire de spécifier tout le système formellement. Il est possible de spécifier seulement les parties critiques du système avec un langage formel de spécification, et les parties non critiques par des des méthodes semi-formelles(méthodes graphiques par exemple). Cette deuxième approche consisterait alors à utiliser les deux langages en même temps dans un même projet. Il est aussi possible d'utiliser dans un projet un langage semi-formel et spécifier les parties critiques avec un outil formel.

Dans le premier cas il s'agit d'identifer tout d'abord les parties qui se prêtent bien à tel langage qu'à tel autre et et décider en conséquence le langage à utiliser.

Dans le cas des langages graphiques et formelles, les contraintes temporelles, la coopération entre différentes entités du système peuvent être spécifiées formellement. Tout ce qui est structure statique du système, les différents scénarios ne nécessitent pas une une modélisation formelle.

En prenant exemple sur les langages ALBERT II et UML, Nous aurons à faire coexister des diagrammes UML, statiques et dynamiques avec un modèle ALBERT II , mais dans ce modèle on fera plus attention aux propriétés qu'on ne

peut pas modéliser avec UML. Cette approche peut être bénéfique sur un autre plan. Plutôt que d'intégrer des techniques, il y aura aussi à faire coexister des points de vue différents et ainsi étudier les échanges possibles entre formalismes.

Le principal problème qui survient est que les deux types de modèles sont liés et cette séparation risque de faire perdre des informations si on ne fait pas attention à la traçabilité entre modèles, surtout si on a affaire à une grosse application.

Une autre idée qui n'est pas éloignée de la précédente est de partir d'une modélisation semi-formelle (graphique) pour essayer ensuite de construire une spécification formelle à l'aide des modèles semi-formels.

En effet comme en programmation où il est vivement déconseillé de passer directement à la programmation sans passer par l'analyse du problème, le passage d'exigences informelles à une spécification formelle dans un projet peut constituer un précipice dans lequel il est facile de trébucher. C'est pourquoi cette transition par une spécification sémi-formelle permettra non seulement de mettre de structurer les besoins des clients mais aussi permettra une compréhension plus poussée du système à construire. Des études sérieuses[7] ont montré que une part non négligeable des erreurs commise dans un projet sont imputables aux exigences du système pas bien comprises.

Dans ce processus, la toute première étape est de passer par le modèle des cas d'utilisation, qui rappelons le sert à donner une vue de haut niveau sur les différentes façons d'utilisation du système. Dans cette perspective les modèles ainsi obtenus permettront non seulement de comprendre rapidement les besoins du système, mais aussi permettra d'arriver rapidement à atteindre un consensus entre clients et les équipes de réalisation du système. C'est à ce moment qu'on dresse les limites du système. La détermination des acteurs et des différentes fonctions du système va alors contribuer à éclaircir progressivement ces limites du système, floues au départ, elles se précisent au fur et à mesure de l'élaboration des différents cas d'utilisation.

Cette action de délimitation est importante car elle touche aussi un autre plan : le plan contractuel. Ceci est important étant donné qu'un pourcentage non négligeable de systèmes ne sont pas utilisés parce que après leur réalisation, les clients se rendent compte qu'ils ne correspondent pas à leur besoin.

Si les cas d'utilisation servent aussi de base contractuelle, il est logique qu'elles vont servir tout au long du processus de développement du système, elles dépassent donc le seul cadre de détermination des besoins du système.

L'identification des différentes fonctions du système à l'aide des cas d'uti-

lisation permettra lors de la modélisation avec le langage ALBERT II, d'identifier sans difficultés les actions élémentaires du système ainsi que les différents agents qui composent le système. Il y a aucun problème à commencer par le modèle des cas d'utilisation étant donné que ce modèle est indépendant de toute méthode[12]. Même si il a été conçu pour les méthodes objets, il peut bien servir aussi bien dans les méthodes objets que dans d'autres types de méthodes.

Les techniques traditionnelles comme le maquettage[3] peuvent toujours servir à côte des diagrammes des cas d'utilisation. C'est un moyen en plus de valider les cas d'utilisation.

Dès que le modèle des cas d'utilisation est terminé, on peut commencer à passer à la modélisation formelle. En utilisant le langage ALBERT II, on peut commencer à identifier les actions et les agents. Dans le modèle des cas d'utilisation, un cas d'utilisation peut contenir des sous cas d'utilisation et ceci jusqu'au niveau de détails voulu par l'ingénieur des basoins. Les actions Albert vont alors se définir à partir des cas d'utilisation de bas niveau si les cas d'utilisation ont plusieurs niveaux de détails. Le concept d'agent utilisé dans le langage ALBERT II sera une abstraction des concepts d'«utilisateur» du système et d'«acteurs» définis dans le langage UML. Leur identification est assez triviale à partir des diagrammes des cas d'utilisation.

Nous avons vu que le séquencement interne des cas d'utilisation est donné par les diagrammes de séquence. Après avoir fini avec les diagrammes des cas d'utilisation, il faut continuer avec la réalisation des diagrammes de séquence.

L'objectif recherché en passant par les diagrammes de séquence pour continuer la spécification avec ALBERT II est d'une part de pouvoir identifier les différents messages et événements, d'autre part d'arriver à comprendre l'enchaînement des actions en se basant sur la chronologie de ces messages et événements. On saura ainsi quelle action est faite après telle autre et quelle propriété pour que telle action puisse être déclenchée. On peut ainsi déterminer les composants d'état des différents entités identifiées avec les diagrammes de séquence.

Cette représentation graphique des interactions entre différentes entités est d'un grand intérêt pour la suite de la modélisation. Les diagrammes de séquence donnent au concepteur une possibilité unique de voir la séquence complète des évènements qui surviennent dans un système ou sous-système avec un point de vue global. Le concepteur peut rapidement se faire une idée de la progression de la séquence d'événements sur les entités qui participent à la réalisation d'une fonction opérationnelle. On peut aussi voir la structure du du système, ce

[3]Technique consistant à construire une maquette, c'est à dire une interface utilisateur pour permettre d'obtenir les avis des futurs utilisateurs en leur montrant d'éventuelles intéractions entre eux et le système.

qui nous donne beaucoup d'information sur les caractéristiques de réalisation du système. Cette vue globale sur la dynamique du système va permettre une identification facile de certains types de contraintes. Nous pensons aux contraintes de préconditions, d'effets d'actions et de déclenchements. Le seul incovénient avec les diagrammes de séquences est qu'ils donnent seulement certains comportements mais ne donnent pas tous les comportements possibles. C'est pour cette raison que le langage UML propose un autre type de diagramme pour pouvoir visualiser tous les comportements possibles des entités composants un système.

Après avoir construit les diagrammes d'interaction, on peut se consacrer entièrement de la spécification Albert, en définissant des contraintes relatives aux différentes actions et propriétés identifiées dans les phases précédentes. La réalisation d'un diagramme états-transitions va alors intervenir pour compléter les vues dynamiques des diagrammes de séquence. Les diagrammes d'états-transitions vont aider à vérifier si dans la spécification avec le langage Albert on a tenu compte de toutes les préconditions de déclenchement des actions. Comme pour les diagrammes de séquence, ce type de diagramme va permettre d'avoir une vue globale de tous les états, les transitions entre états, ainsi que tous les événements qui surviennent au sein d'une entité. En effet bien que le langage ALBERT II possède certains types de contraintes servant à modéliser les transitions entre états, une visualisation graphique de tous les comportements possibles d'une entité du système ou de tout le système semble être un complément appréciable à un modèle Albert.

Le seul problème qui risque de survenir est celui de la maîtrise de la cohérence des documents. Au niveau des cas d'utilisation et des diagrammes de séquence ce problème ne devrait pas se poser étant donné que ces diagrammes serviraient de base pour réaliser une spécification Albert.

6.7 Amériolation du langage ALBERT II

Dans cette perspective de fournir aux langages formels des avantages graphiques des méthodes semi-formelles, une autre voie qui n'est pas très éloignée de la première approche serait d'étendre l'environnement graphique du langage ALBERT II. En effet le langage possède une contrepartie graphique en ce qui concerne la modélisation des propriétés et comportement des agents : composants d'état, actions. Pour le moment les seules contraintes qui sont supportées graphiquement par l'éditeur du langage, sont les contraintes de coopération(en partie) et les contraintes de composants dérivés. Ces diagrammes sont produits au moment où on déclare qu'une action ou un composant d'état est exporté ou

importé ou quand on définit une dépendance entre composants d'états. L'idée serait de développer la partie graphique pour que d'autres types de contraintes puissent être supportées, mais inverser la procédure pour que l'on puisse travailler graphiquement et générer automatiquement les contraintes(sous forme mathématique). On gagnerait en convivialité(du moins pour ceux qui sont allergique aux mathématiques) mais en plus cela permettrait de garder les outils qui supportent le langage.

6.8 Conclusion

En conlusion l'acceptabilité des méthodes formelles de spécification est freiné par le manque de convivialité de ces méthodes. Les liens que nous avons trouvés entre certains contraintes du langage ALBERT II et les diagrammes états-transitions du langage UML peuvent conduire à développer un environnement graphique et formel de modélisation. Cet environnement permettrait d'augmenter l'acceptabilité du langage ALBERT II dans l'industrie, Selon certains auteurs[9] des études ont montré que non seulement les méthodes graphiques permettent d'impliquer le plus possible les utilisateurs dans les projets, car elles sont compréhensibles(au moins partiellement) par les non spécialistes mais en plus elles sont d'une grande importance car la modélisation graphique est plus près des vraies applications.

Conclusion générale

La réalisation de ce travail nous a permis de nous familiariser un peu plus avec la problématique des besoins dans la réalisation des systèmes informatiques. Nous avons remarqué que face à la rigueur qui doit accompagner le développement d'applications critiques, des outils appropriés doivent être mis en oeuvre.

Le langage ALBERT II présente beaucoup d'avantages pour la modélisation des systèmes critiques, mais parallèlement il présente aussi quelques faiblesses. En effet il serait intéressant que dans l'avenir des travaux approfondis soient menés pour fournir au langage ALBERT II un environnement de modélisation graphique. Cette environnement permettrait de générer automatiquement une spécification Albert, sur laquelle portera le reste du processus, à partir d'une modélisation graphique. Ce changement permettra sans doute de faire passer le langage du monde de la recherche vers l'industrie.

Une autre extension à apporter au langage est de doter le langage de la notion d'héritage entre agents. Ceci contribuera à accroître la structuration des spécifications en Albert.

Le langage UML suscite beaucoup d'espoir dans le monde de la modélisation. Bien que le langage possède des aspects puissants pouvant aider dans la modélisation des systèmes critiques, le langage présentent beaucoup de lacunes. Beaucoup de concepts ne sont pas bien définis, et peuvent conduire à divers interprétation. Il est nécessaire que des travaux visant à doter le langage d'une sémantique formelle soient menés pour pouvoir mieux éclaircir les concepts qu'il nous propose.

Si dans le passé les analystes se fixaient seulement sur l'utilisation d'une méthode particulière, ils commencent à se rendre compte qu'aucun langage ne peut tout exprimer. La tendance est donc de plus en plus à l'intégration des méthodes afin d'enrichir progressivement une méthode par des éléments en provenance des autres. Cette intégration de méthodes passe par plusieurs voies. La plus accessible est l'utilisation conjointe des méthodes différentes dans un même

projet, et la plus délicate étant la définition d'environnements de modélisation qui intègrent les avantages de plusieurs méthodes.

Bibliographie

[1] Allen J.F, *Towars general theory of action and time*Artificial Intelligence vol.23, 2, pp 123-154.

[2] Attoui A.,*Les systèmes multi-agents et le temps réel*, Eyrolles, 1997.

[3] K. Berger , *Using UML for Modeling a Ditributed Java Application*, Munich, July 1997.

[4] F. Bodart , Y. Pigneur , *Conception Assistée des Systèmes d'Information*, Masson, 1989.

[5] G.Booch ,*Object Oriented Design with application*, The Benjamin/Cummings Publishing Company, 1991.

[6] J.M. Bruel ,*Sémantique formelle pour UML. Pourquoi ? Comment ?*, Pau, 1999.

[7] E. Dubois ,*Méthodologie de développement de logiciel*, cours de l'Institut d'Informatique, Namur, 1998.

[8] P. Du Bois , *The ALBERT II Manual Reference, Language Constructs and Informal Semantics, Version 2.0*, Namur, 1997.

[9] H. Ehrig , *Graphical support and integration of formal and semi-formal methods for software specification and development*, Berlin, 1994.

[10] J.P Haton et al, *Raisonnement en intelligence artificielle, modèles, techniques et architectures pour les systèmes de connaissances* , InterEditions, 1991.

[11] D. Harel ,*StateCharts : A Visual Formalism for Complex Systems*, Science of Computer Programming, Vol. 8, 1987.

[12] I. Jacobson , *Object-Oriented Software Engineering, A Use Case Driven Approach*,Addison-Wesley, 1992.

[13] C. Kevin , *An object-orientated extension to Z. In John E. Nicholls, editor, Z User Workshop, Oxford 1990, Workshops in Computing, pages 151-172*,Springer-Verlag, 1991.

[14] M. Lai , *UML, la notation unifiée de modélisation objet-applications en Java*, InterEditions, 1997.

[15] Z. MANNA and al., *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[16] P-A. Muller, *La modélisation objet avec UML*, Donod, 1986.

[17] J. Ramaekers ., *Systèmes d'exploitation, matières approfondies*, cours de l'Institut d'Informatique, Namur 1998.

[18] J. Rumbaugh and al, *Object Oriented Modeling and Design*, Prentice Hall International, 1991.

[19] P-Y Schobbens, *syntaxe et sémantique des langages de programmation*, cours de l'Institut d'Informatique, Namur, 1997.

[20] W-A. Scott, *The Unified Modeling Language V1.1 and beyond : The techniques of object-Oriented Modeling*, An Ambysoft Inc. White Paper, SIGS Books/Cambridge University Press, 1998.

[21] A. UMAR *Distributed computing and client server systems*, Prentice Hall, 1993.

[22] UML Group, *UML summary, version 1.1*, Rational Software Corporation, Santa Clara, CA-95051, USA, 1 September 1997, URL : http ://bardolfo.ce.unipr.it/documentation/uml/97-08-03.pdf.

[23] UML Group, *UML Semantics. Version 1.1*, Rational Software Corporation, Santa Clara, CA-95051, USA, 1 September 1997, URL :http ://bardolfo.ce.unipr.it/documentation/uml/97-08-04.pdf.

[24] UML Group, *UML Notation Guide, version 1.1*, Rational Software Corporation, Santa Clara, CA-95051, USA, 1 September 1997, URL : http ://bardolfo.ce.unipr.it/documentation/uml/97-08-05.pdf.

[25] R. Turner , *Logics for artificial intelligence*, Halsted Press, 1984.

[26] THOMSON-CSF, *CIS-CAT, outil d'architecture logique de SIC pour le commandement, suivant la méthode CUBE TOOL*, version 2.3, THOMSON-CSF, 1991.

[27] J.M Zeippen and al, *the Analyst Reasoning on requirement Specifications for Real-Time and Distributed systems*, Namur 1997.

Annexes

Annexe A

Exemple d'utilisation conjointe d'ALBERT II et UML dans une spécification

A.1 Introduction

L'exemple que nous allons modéliser avec les langages Albert et UML est un hypothétique protocole de communication entre des personnes chargées de surveiller un territoire. L'exemple paraît plus simple par rapport à la réalité sur le terrain, mais il nous a permis d'illustrer quelques une de nos idées sur une éventuelle complémantarité ALBERT II-UML dans un projet. Nous avons dans un premier temps modélisé l'exemple avec le langage UML, notamment avec les diagrammes des cas d'utilisation et les diagrammes d'interaction et dans une deuxième phase nous nous sommes basés sur ces diagrammes pour spécifier l'exemple avec le langage ALBERT II.

A.2 Bref présentation de l'exemple

Le cas que nous présentons est un exemple qui a été utilisé pour valider l'outil CIS-CAT[26] qui est un outil de conception et de dimensionnement d'architecture logique de Système d'Information et de Communication(SIC) pour le commendement et qui a été développé chez THOMSON-CSF. L'exemple contient trois fonctions opérationnelles. La description de ces fonctions opérationnelles est repris ci-dessous.

A.2.1 Fonction opérationnelle surveillance

Le superviseur de la surveillance (S1) définit les directives de surveillance qu'il envoie au générateur de situation aérienne (S2), à l'élaborateur de pistes (S3) et à l'utilisateur de la surveillance (S5). Après avoir reçu ces directives, S2 définit les ordres de surveillance qu'il envoie à S3 et à S1. Cette définition dure au plus 7 secondes. De ces directives et en fonction de la situation aérienne courante, S3 envoie des directives de détection aux Senseurs (S4). Ce message est également envoyé à S2. S4 détecte(pendant au plus 3 secondes) les objets volants dans sa zone de détection et effectue l'extraction des plots(2 secondes). Ces plots sont envoyés à S3 qui est en charge de réaliser le pistage. Les pistes primaires sont générées par S3, ensuite envoyées à S2 qui met à jour la situation aérienne(5secondes) et cette dernière est envoyée à S1, S3 et S5. C'est S5 qui est chargé d'informer le Commandant du théâtre de la situation aérienne.

En mode de fonctionnement secours vertical, c'est l'utilisateur de la surveillance qui définit les ordre de surveillance.

A.2.2 Fonction opérationnelle emploi des forces

Le Commandant de théâtre (F1) génère l'ordre général en tenant compte de la situation aérienne. (si F5 est en mode de fonctionnement secours vertical, c'est lui qui est prends en charge cette action. Cet ordre est envoyé au Commandant d'opérations (F2), au Commandant élémentaire (F3) et aux responsables interface contrôle de mission et surveillance (F5 et F6). En fonction de cet ordre général et de la situation courante F2 (F5 si il est en mode de fonctionnement secours vertical) effectue la planification d'opérations et génère des ordres particuliers(15 secondes) qui sont envoyés à F1, F3, F5, F6 et à l'exécutant F4. La préparation de mission est réalisée au niveau de l'unité par F3. Les ordres de mission sont envoyés à F2, F4, F5 et F6. Les missions sont exécutées par F4 qui donne rapport à F3 et informe F5 et F6. Les comptes rendu de mission sont générés par F3 et envoyés à F2 en charge de produire le compte rendu d'opérations qui sera transmis à F1. C'est également F3 qui est chargé du compte rendu sur la disponibilité des forces qu'il commande(6 secondes). Il remet ce rapport à F2 qui en fait une synthèse pour produire l'état des forces(15 secondes) destiné à F1 et F5.

En mode de fonctionnement secours vertical, c'est le commandant des opérations qui définit l'ordre général.

A.2.3 Fonction opérationnelle contrôle mission

Le Contrôleur (M2) est chargé d'effectuer la plupart des services de cette fonctionnalité. C'est en effet le Contrôleur qui fait l'évaluation de la menace courante(12 secondes), qu'il envoie au Commanditaire (M1) et au niveau Associé (M4) A partir de cette évaluation, les ordres d'engagement sont générés et transmis au niveau Assujetti(M3) et à M1 et M4. En fonction des ordres de contrôle qu'il a reçu, M3 ajuste sa mission et rend compte à M2. Ce dernier génère et envoie un rapport de contrôle(4 secondes) à M1 et également à M4. La disponibilité de contrôle est également effectuée par M2 et le message correspondant est envoyé à M1 et à M4.

A.2.4 Mode de fonctionnement

Nous avons 2 modes de fonctionnement :

- fonctionnement normal : chacun s'acquitte de ses responsabilités habituelles.
- Le fonctionnement secours vertical : dans ce mode de fonctionnement un acteur se voit dévolu occasionnellement des responsabilités qu'il ne dispose pas en fonctionnement normal, mais à ce moment il est déchargé des ses activités habituelles.

A.3 Les diagrammes des cas d'utilisation et de séquence

A.3.1 Introduction

Dans cette exemple, nous nous sommes intéressés à deux types de diagrammes, à savoir les diagrammes des cas d'utilisation et les diagrammes d'interaction. Bien que les digrammes des classes soient au coeur de la phase d'analyse nous n'avons pas trouvé d'intérêt à exploiter ce diagramme pour cette exemple.

A.3.2 Les cas d'utilisation

Nous avons trouver trois cas d'utilisation au niveau le plus élevé : la surveillance, l'emploi forces et le contrôle de mission. Ces cas d'utilisation correspondent à 5 scénarios comme nous le verrons avec les diagrammes de séquence. Les acteurs sont ici les personnes qui doivent communiquer et les senseurs. Les diagrammes des cas d'utilisation nous a servi de points de départ pour atteindre le niveau de description des scénarios. Malheusement les informations à notre disposition ne nous ont pas permis de décrire d'une façon détaillée les sous cas d'utilisation.

A.3.3 diagrammes de séquence

Les diagrammes de séquence se construisent très facilement à partir de la description des flux de messages. Un scénario est associé chaque fois à un cas d'utilisation. Ici les scénarios changent selon le mode de fonctionnement d'un acteur particulier. Nous avons décidé de décrire deux scénarios pour la surveillance, deux scénarions pour l'emploi des force et un scénario pour le contrôle de mission. Dans cette dernière fonction opérationelle, tous les acteurs n'ont que comme mode de fonctionnement, le mode normal.

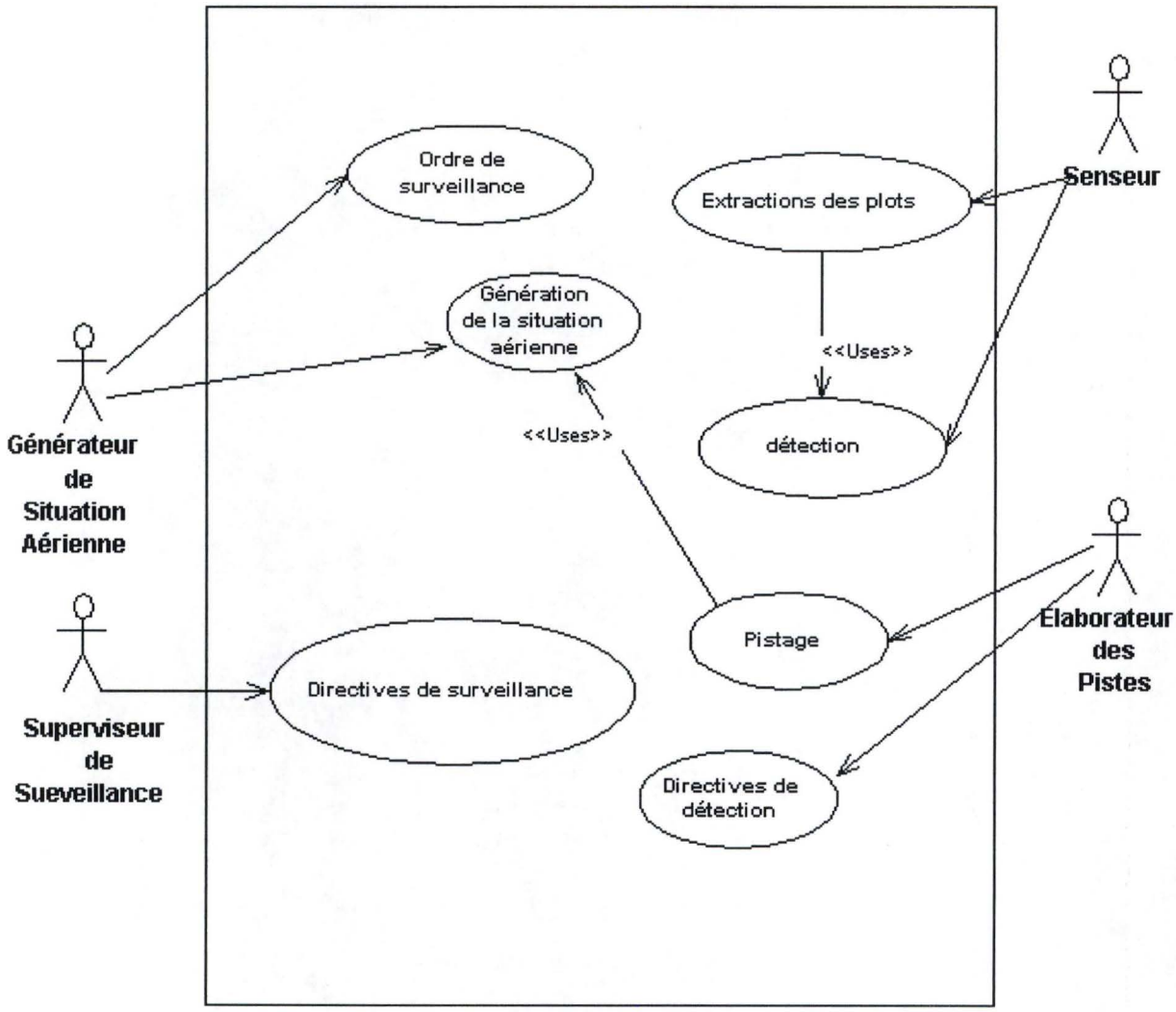


FIG. A.1: Surveillance

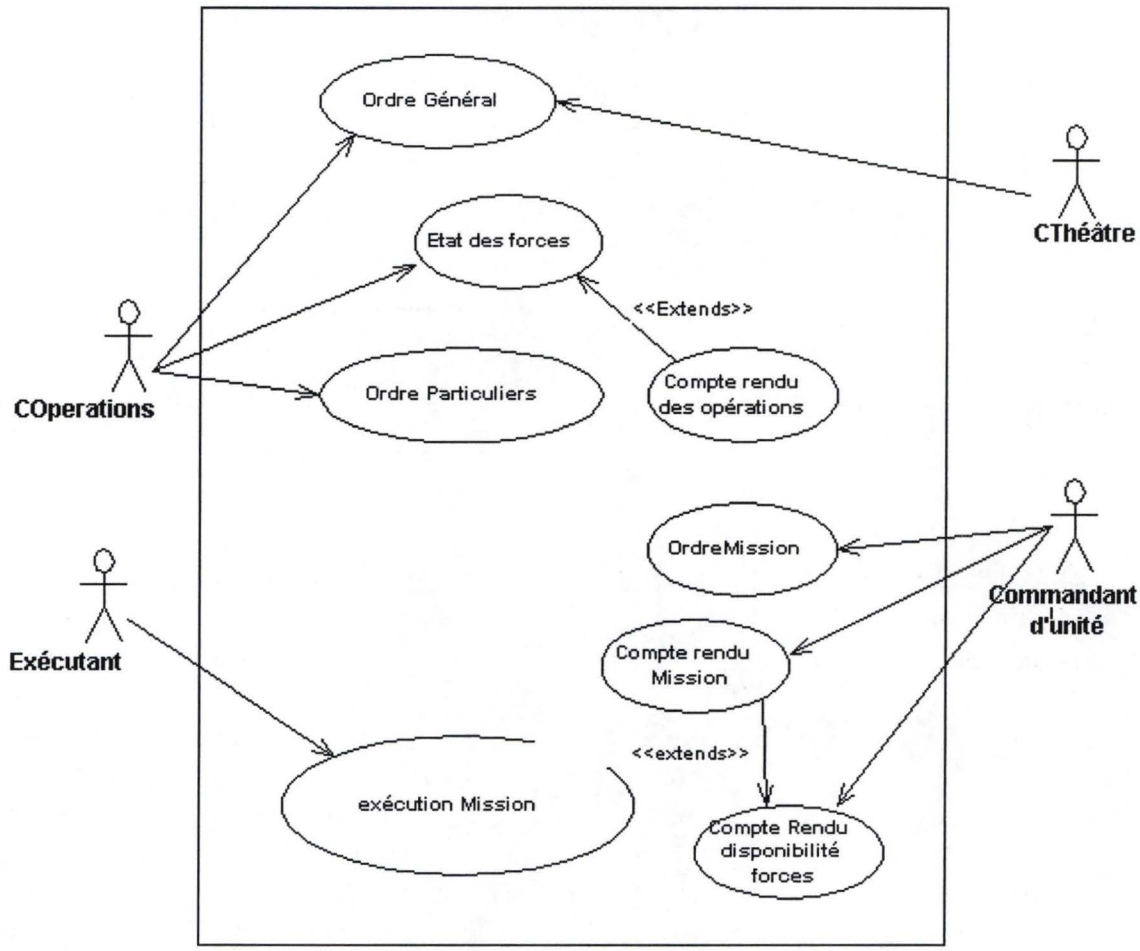


FIG. A.2: Emploi forces

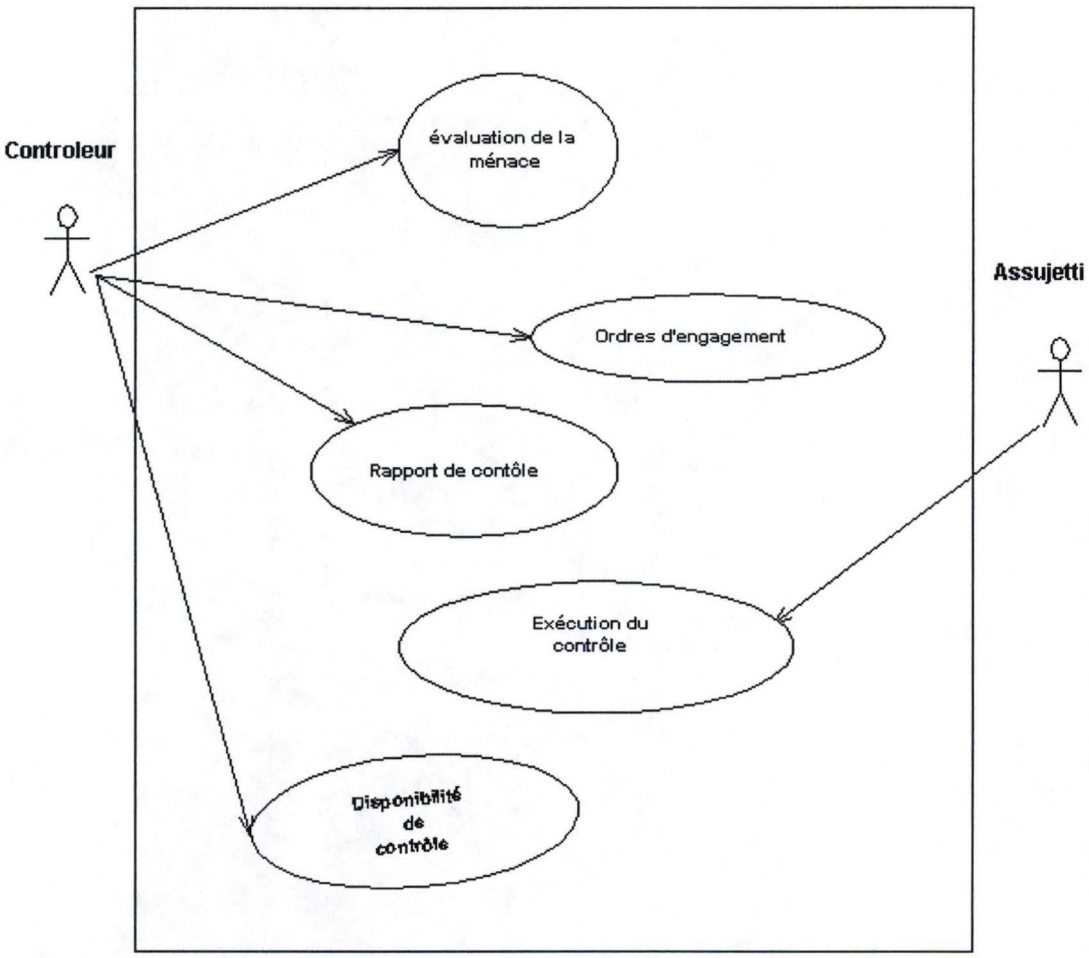


FIG. A.3: Contrôle Mission

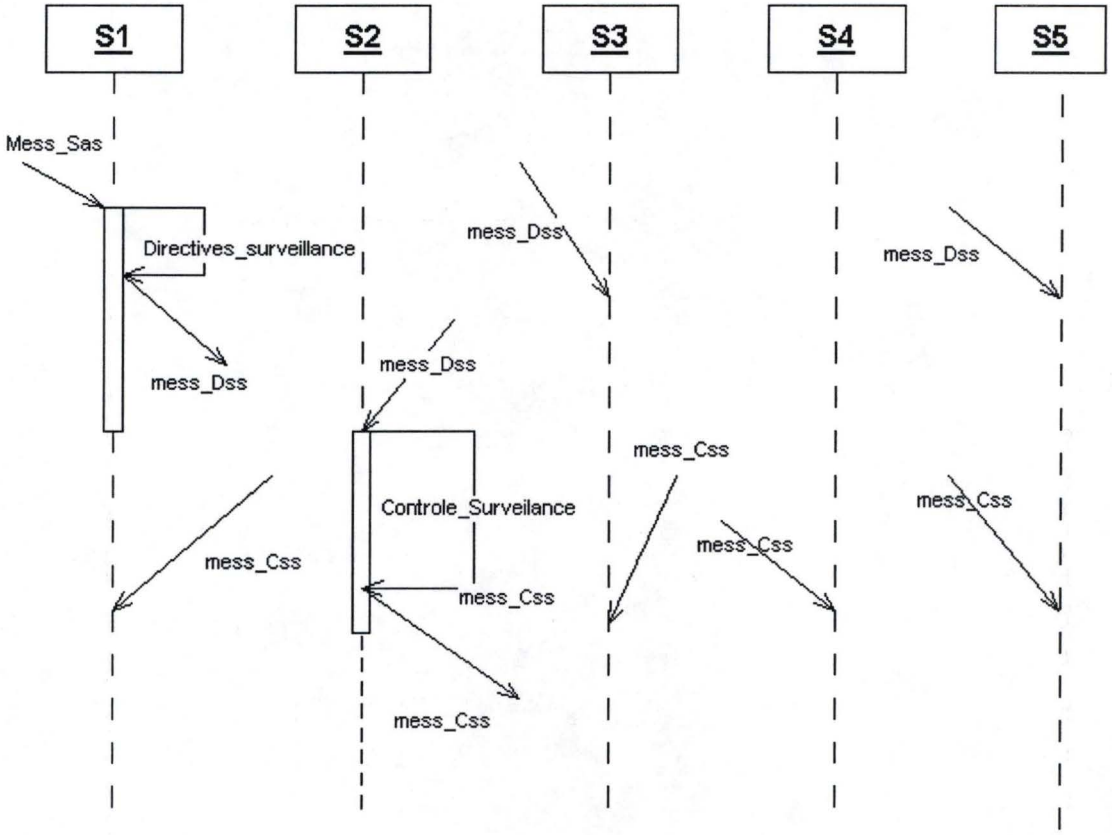


FIG. A.4: Surveillance 1.1

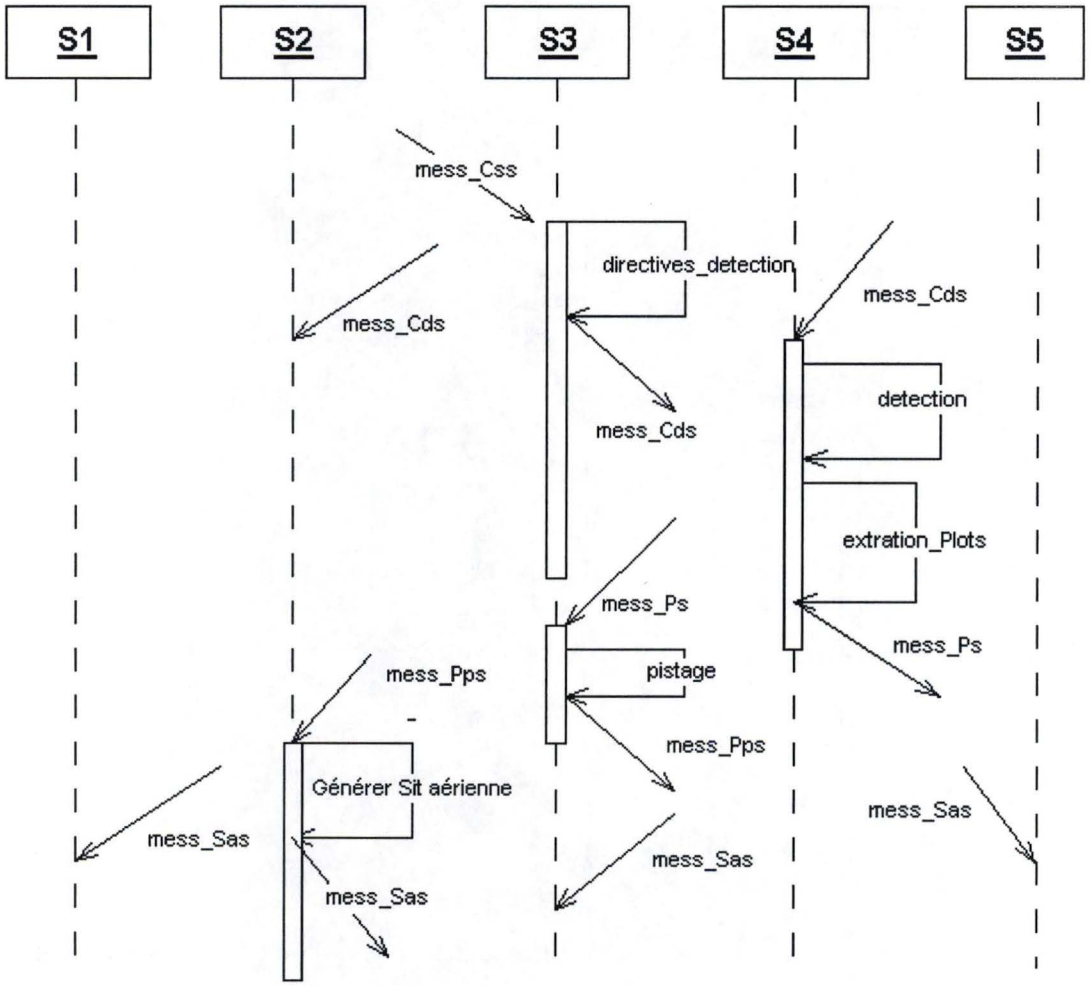


FIG. A.5: Surveillance 1.2

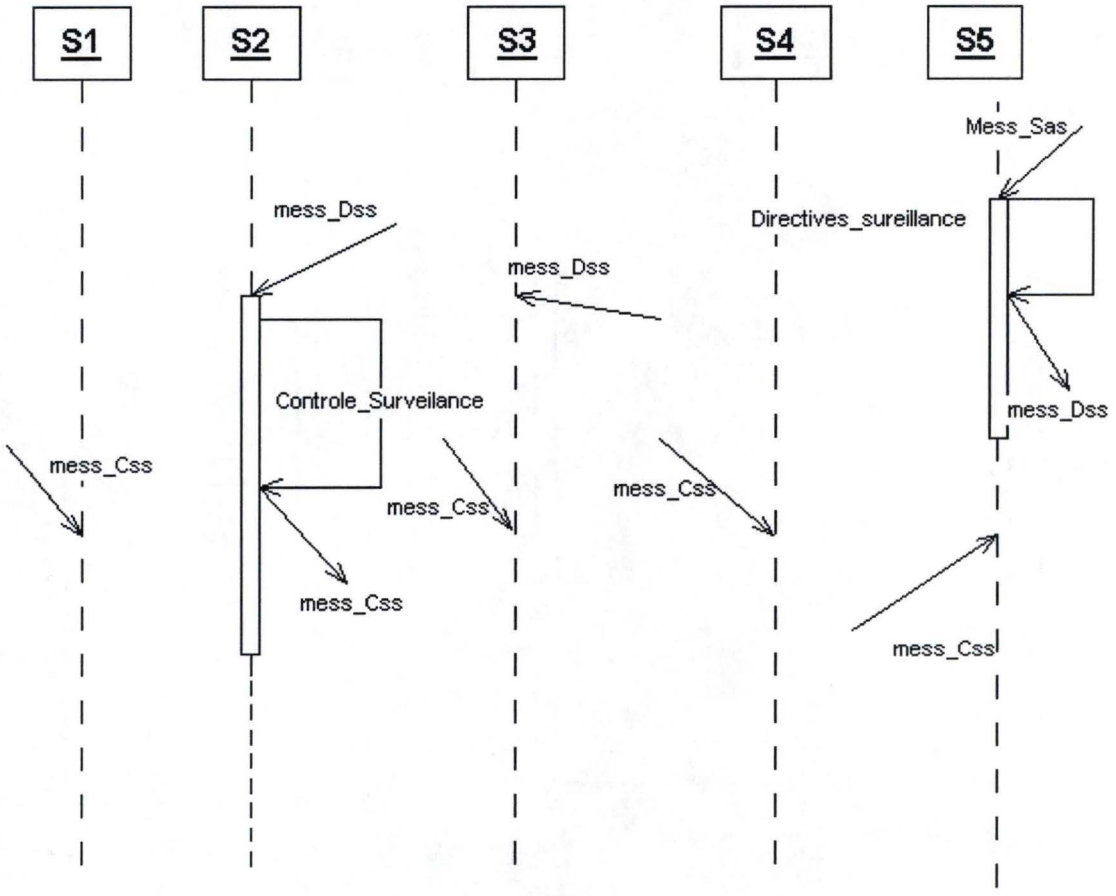


FIG. A.6: Surveillance 2.1

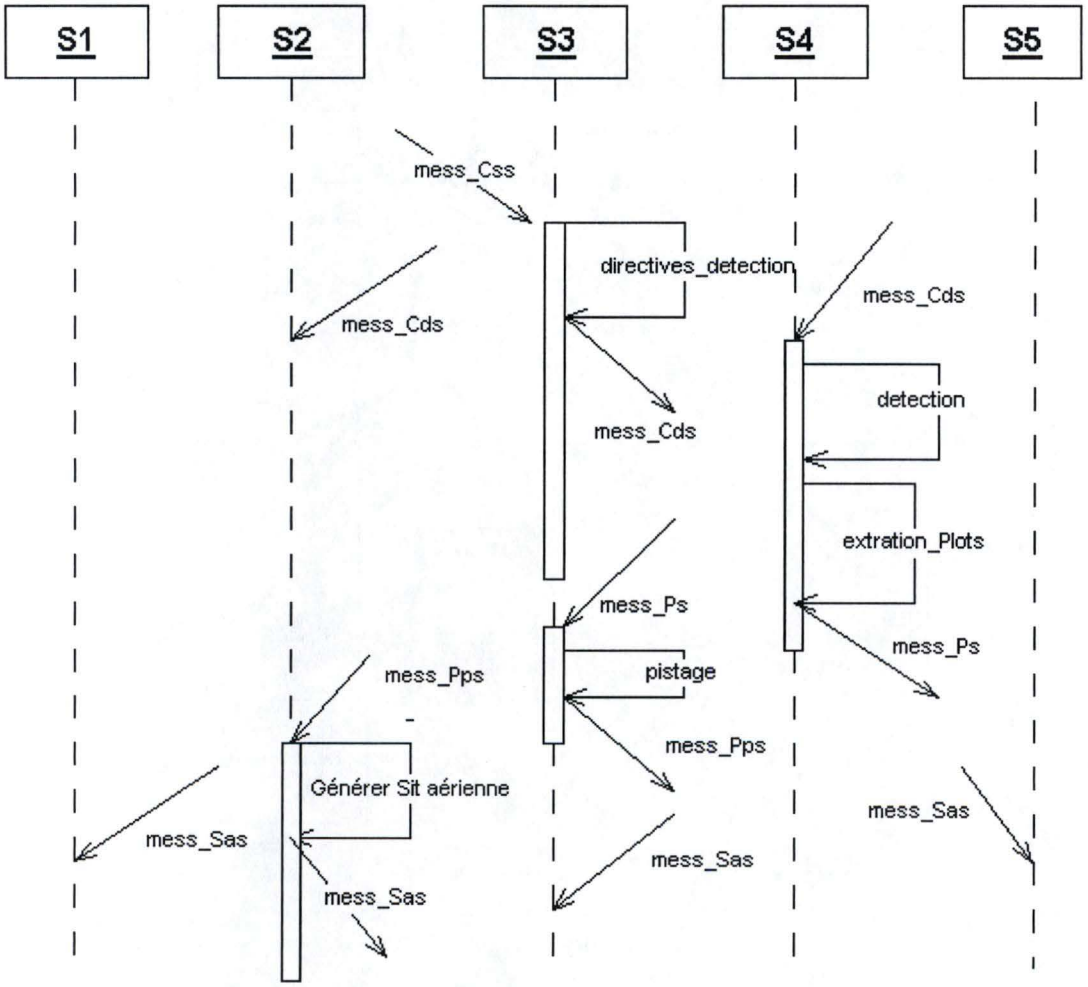


FIG. A.7: Surveillance 2.2

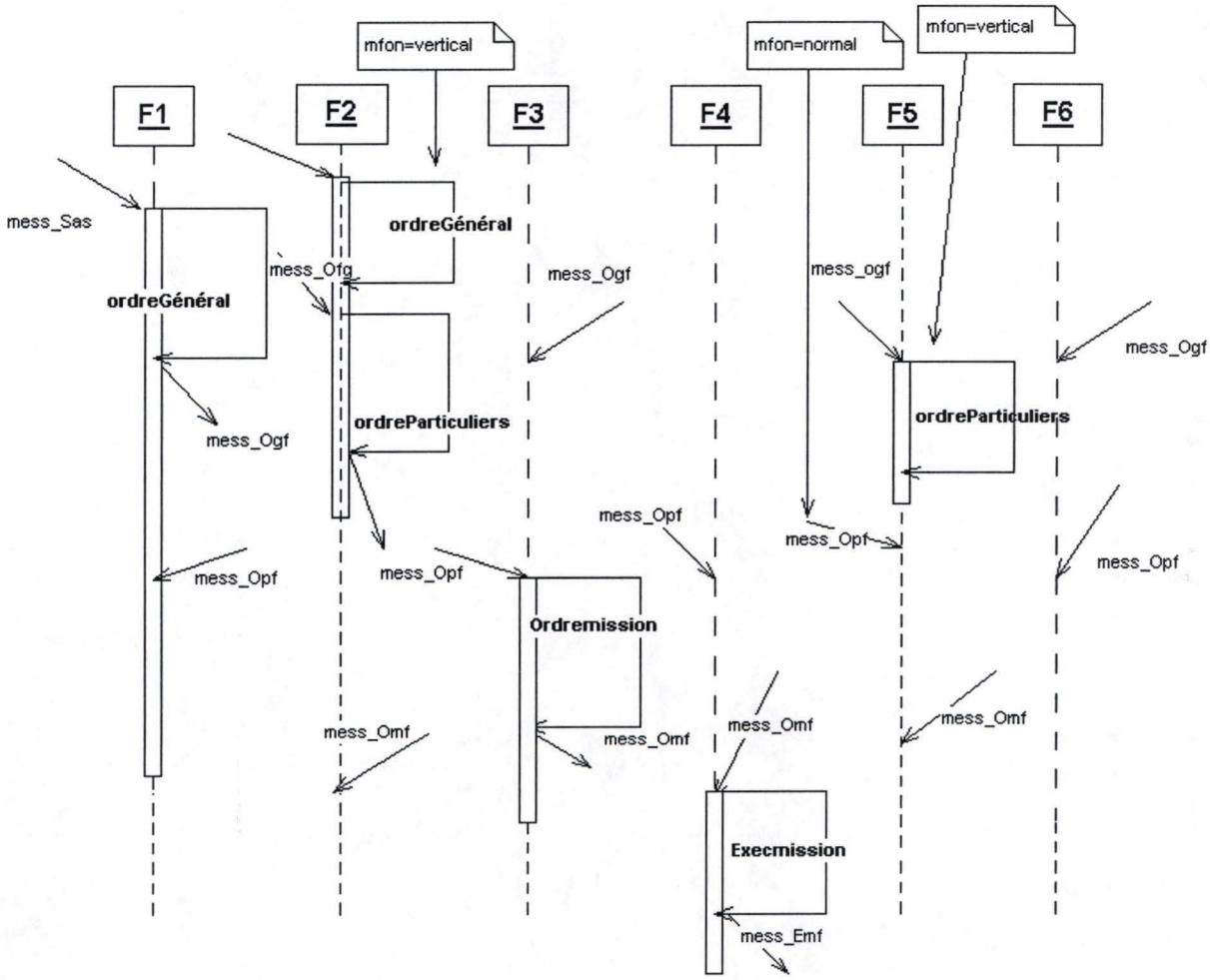


FIG. A.8: Emploi forces 2.1

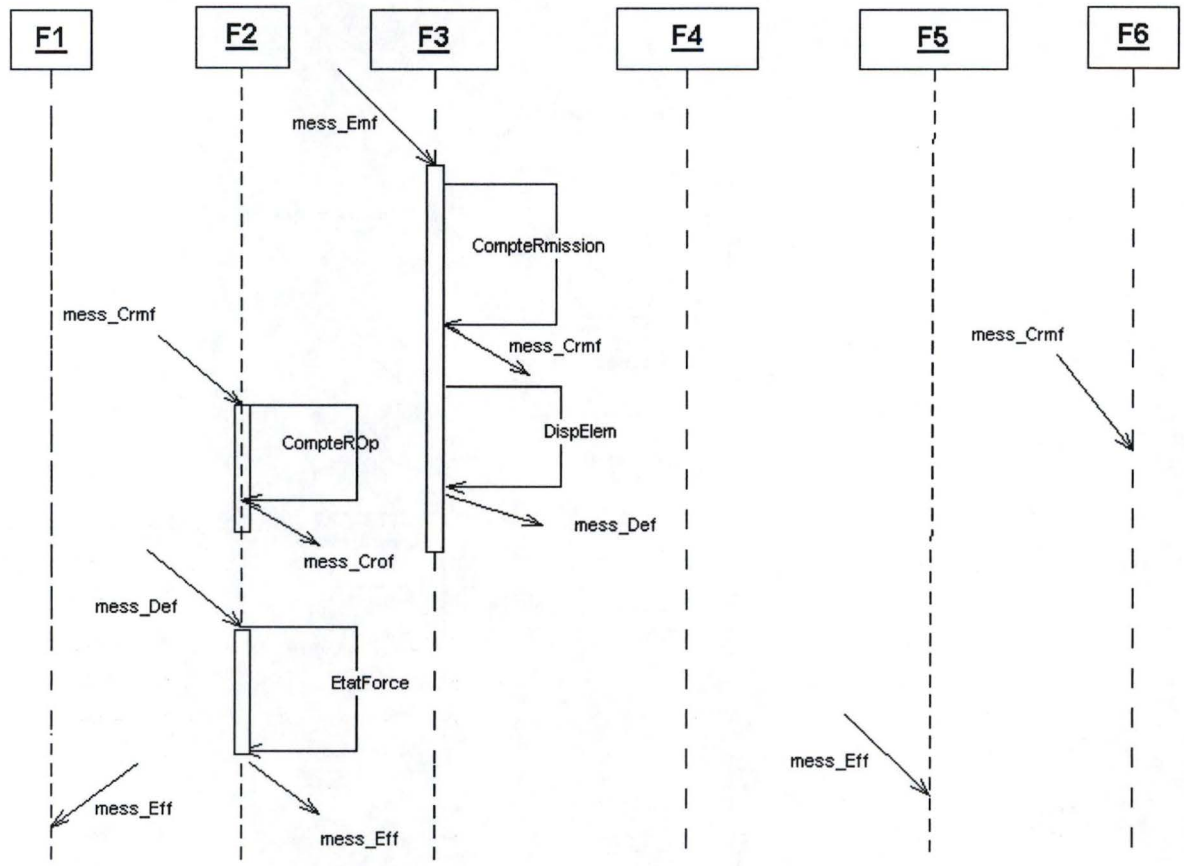


FIG. A.9: Emploi forces 2.2

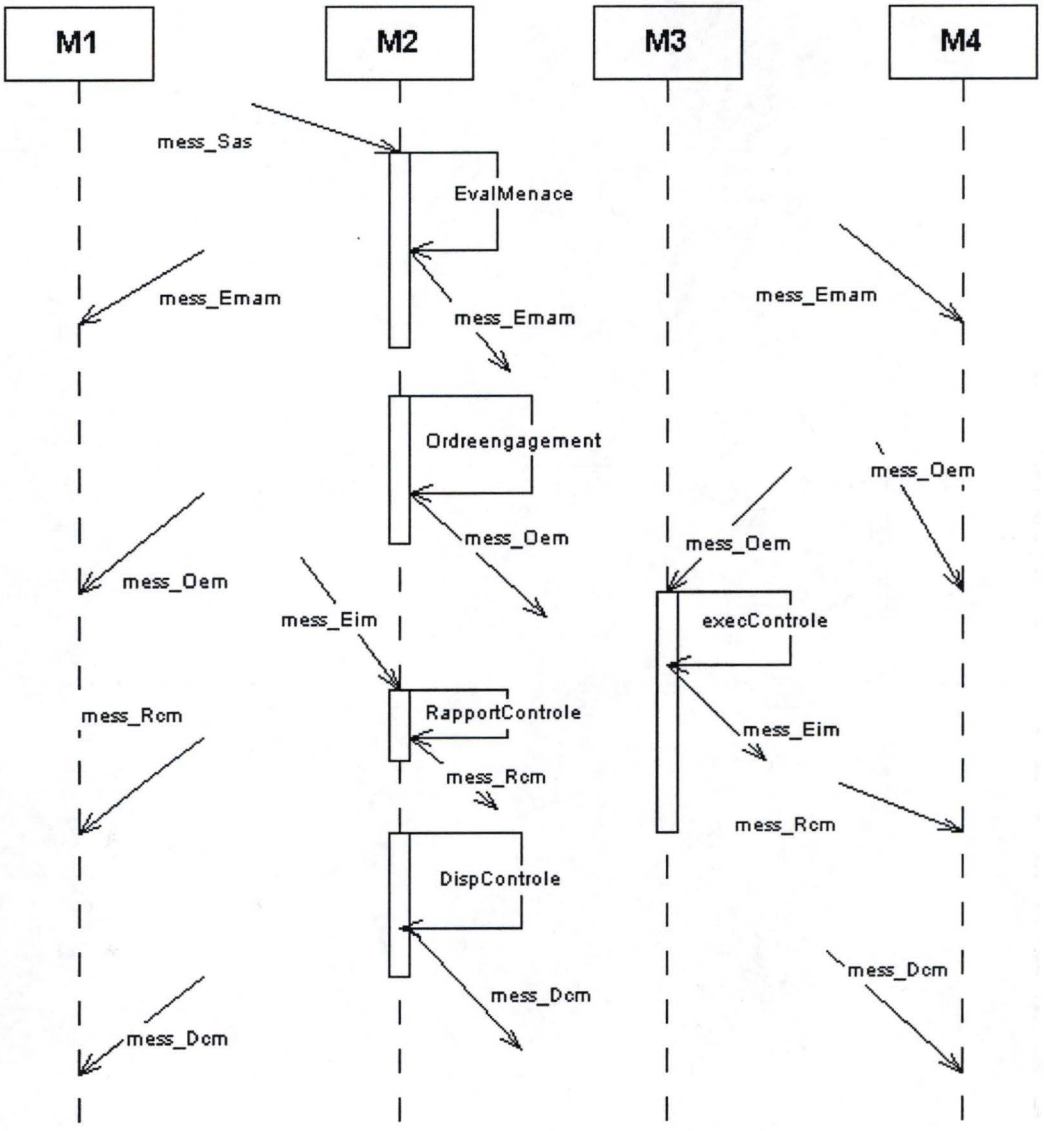


FIG. A.10: Contrôle Mission

Comme on le remarque dans cet exemple, un scénario d'un système événementiel est une séquence d'actions et d'événements permettant d'assurer une fonction de plus haut niveau. Dans les diagrammes présentés ci haut, les actions sont déclenchées par l'arrivée des événements déclencheurs(préconditions) qui sont ici des messages entrants vers l'entité qui réalise l'action. Les messages résultants de l'exécution des actions peuvent mener à un ou plusieurs événements déclencheurs vers d'autres entités.

La construction d'un tel diagramme nous a permis de faire un pont entre les exigences informelles et la spécification formelle avec le langage ALBERT II. Il est à remarquer que ces diagrammes se prêtent bien aux systèmes événementiels comme dans notre exemple. Une version de ces diagrammes appelée «Message Sequence Chart» est d'ailleurs très utilisée en télécommunication.

Nous pouvons aussi remarquer sur ces diagrammes que la structure de ces derniers révèle des informations sur la structure de l'architecture du système. En effet les diagrammes d'interactions permettent de mesurer la décentralisation du système. Distinguons les deux cas extrêmes de la structure des diagrammes d'interaction.

Le premier extrême est un diagramme en peigne. Ce type de diagramme est caractérisé par une entité qui agit sur toutes les autres et contrôle toutes les autres entités. Une grande partie des comportements est placé dans l'entité centrale qui doit connaître tous les autres et les utilise pour des commandes ou des questions directes. Le diagramme en peigne indique une structure centralisée. Nous pouvons voir que dans nos diagrammes nous avons un contrôle centralisé. Pour la clarté des diagrammes nous n'avons pas représenté l'entité centrale qui assure le contrôle.

L'autre extrême est le diagramme en escalier. Ce type de diagramme est caractérisé par une forte délégation des responsabilités. Chaque objet ne connaît qu'un petit nombre d'entités et ceux qui peuvent l'aider pour un comportement spécifique. Nous n'avons pas d'entité centrale. Les escaliers indiquent souvent une structure décentralisée. Chaque entité a une tâche séparée et seul les objets avoisinants sont nécessairement en interaction. Nous n'avons pas de diagrammes de ce genre dans notre cas.

A.4 Modélisation avec le langage ALBERT II

Après la construction des diagrammes d'interaction, nous avons modélisé l'exemple à l'aide du langage ALBERT II. L'identification des agents était assez

trivial après la construction de ces diagrammes. Pour l'identification des sociétés nous nous sommes basés sur la façon de regrouper les acteurs dans l'outil CISCAT. Dans cet outil, il existe ce le concept de «centre type». En effet cette notion permet de regrouper de manière convenable différents acteurs relatifs à des fonctions opérationnelles différentes afin de permettre un exercice efficace du commandement. Le regroupement dans les centres types respecte en général le rang hiérarchique des différents acteurs.

Les composants d'état que nous avons identifiés sont les messages courants des agents ainsi que leurs modes de fonctionnement.

Pour ce qui est des actions, les informations à notre disposition nous permettaient d'identifier facilement les actions en rapport à chacune des trois fonctions opérationnelles identifiées.

Les contraintes qu'on a eu à faire face sont principalemnt des contraintes de préconditions pour l'exécution des action : message courant à la disposition d'un agent et son mode de fonctionnement pour pouvoir accomplir une action. Les contraintes de coopération ont été beaucoup exploitées. En effet nous avons modélisé le fait qu'un agent envoie un message à un autre par une percetion du composant d'état message courant. Certaines actions ont des délais d'exécution, qui sont en général des durées au plus tard.

Nous regrettons que les seules contraintes temporelles que nous avons trouvé dans l'exemple soient seulement des contraintes de durée d'actions. Cette situation ne nous a pas permis d'exploiter d'autres concepts temps réel du langage alors que ces derniers constituent la principale force du langage ALBERT II.

A.5 Spécification Albert

Data Types and Operations

CONSTRUCTED TYPES

```
| Message représente tous les types de messages qui sont
| échangés entre agents
```

MESSAGE =
ENUM [Msg_Dss, Msg_Css, Msg_Cds, Msg_Ps, Msg_pps, Msg_Sas, Msg_Ogf, Msg_Opf, Msg_Omf, M

```
| Type pour exprimer les deux modes de fonctionnement que les agents
| peuvent avoir
```

FONMODE = ENUM [Norm, Sev]

```
| Type booléen pour exprimer le status des senseurs
```

STATUS = ENUM [running, stopped]

Society: **sda**

```
| Nous avons sept sociétés : centre du théâtre des opérations
| centre opérationnelle des combats, commandement d'unité
| centre de surveillance, centre de détection, centre de contrôle
| centre vecteur
```

(*CTO*)
(*COC*)
(*CU*)))
(*CSU*)))
(*CDE*)))
(*CC*)))
(*VEC*)))

Society: **sda.CTO**

```
| dans cette société nous avons S5 et le commandant du théâtre des
| opérations
```

(*UtilSurv*)
(*CTheatre*)

Agent: **sda.CTO.UtilSurv**

| c'est l'agent chargé d'assurer la liaison avec les agents
| qui utilisent la surveillance

DECLARATIONS

State Components

| Cette composante sert à donner le message courant
mCour instance-of *MESSAGE**
$\longrightarrow$ *sda.CTO.CTheatre*
| Composant d'état qui nous renseigne sur le mode de fonctionnement.
Mfon instance-of *FONMODE**
$\longrightarrow$ *sda.COC.SupSurveillance*

Actions

DirectSurveillance

BASIC CONSTRAINTS

Initial Valuation

| Le message courant au début est le message situation aérienne
$mCour = Msg\_Sas$
| Mfon a comme valeur Undef au début
$Mfon =$ UNDEF

DECLARATIVE CONSTRAINTS

State Behaviour

[*DirectSurveillance*] $Mfon = Sev$

OPERATIONAL CONSTRAINTS

Preconditions

| Cette action a eu lieu seulement si le message courant est le
| message situation aérienne et si le mode de fonctionnement
| est le mode secours vertical
$DirectSurveillance : (Mfon = Sev) \wedge (mCour = Msg\_Sas)$

Effects of Actions

COC.SupSurveillance.DirectSurveillance : [ ]
mCour := *Msg_Dss*
CSU.GenSituationAerienne.MAJSituationAerienne : [ ]
mCour := *Msg_Sas*
DirectSurveillance : [ ]
Mfon := *Norm*
mCour := *Msg_Dss*

COOPERATION CONSTRAINTS

State Information

| Le message courant est toujours montré à CTheatre
$\mathcal{K}$ (*mCour.CTheatre* / TRUE)
| Mode de fonctionnement est toujours montré
| au superviseur de la surveillance
$\mathcal{K}$ (*Mfon.COC.SupSurveillance* / TRUE)

Agent: **sda.CTO.CTheatre**

DECLARATIONS

State Components

| état qui nous renseigne sur le mode de fonctionnement.
**Mfon* instance-of *FONMODE*
| état qui nous renseigne sur le message courant
mCour instance-of *MESSAGE**

Actions

| Il s'agit de l'ordre général généré par le commandant du théâtre des
| opérations
**OrdreGeneral* $\longrightarrow$
sda.CC.InterControleMission, sda.COC.COperations, sda.CSU.InterSurveillance, sda.CU.CommElem

BASIC CONSTRAINTS

Initial Valuation

| Au début, CTheatre est en mode de fonctionnement normal
Mfon = *Norm*
| Au début le message courant est le message situation aérienne.

$mCour = Msg\_Sas$

OPERATIONAL CONSTRAINTS

PRECONDITIONS

| Cette action a eu lieu lorque le message courant est
| message situation aérienne et que F2 est en mode de fonctionnement
| normal.

OrdreGeneral : $(mCour = Msg\_Sas) \wedge (COC.COperations.Mfon \neq Sev)$

EFFECTS OF ACTIONS

| à la fin de l'action OrdreParticuliers de COperations, le message
| courant est le
| ordre particulier

COC.COperations.OrdreParticuliers : [ ]
$mCour := Msg\_Opf$

| à la fin de l'action CompteRenduOp de COperations le message courant est
| le
| compte rendu d'opération.

COC.COperations.CompteRenduOp : [ ]
$mCour := Msg\_Crof$

| à la fin de l'action EtatForces de COperations, le message courant est
| le
| message état des forces.

COC.COperations.EtatForces : [ ]
$mCour := Msg\_Eff$

COOPERATION CONSTRAINTS

STATE PERCEPTION

| mCour est perçu par le CTheatre si sa valeur est Msg_Sas

$\mathcal{XK}$ (*UtilSurv.mCour* / $mCour = Msg\_Sas$)

| CTheatre perçoit toujours le mode de fonctionnement
| de COperations

$\mathcal{K}$ (*COC.COperations.Mfon* / TRUE)

Society: **sda.COC**

| Société a comme agents le Superviseur de la Surveillance,
| le COperations et le Commanditaire.

(*SupSurveillance*)
(*COperations*)
(*Commanditaire*)

Agent: **sda.COC.SupSurveillance**

| Il s'agit de l'agent superviseur de surveillance

DECLARATIONS

State Components

| L'état qui nous renseigne sur le mode de fonctionnement
**Mfon* instance-of *FONMODE*
| L'état qui nous renseigne sur le message courant
mCour instance-of *MESSAGE**
$\longrightarrow$ *sda.COC.Commanditaire, sda.COC.COperations*

Actions

| L'action d'élaboration des directives de surveillance
**DirectSurveillance* $\longrightarrow$
sda.COC.COperations, sda.COC.Commanditaire, sda.CSU.GenSituationAerienne, sda.CDE.ElabPistes, s

BASIC CONSTRAINTS

Initial Valuation

| Au début le mode de fonctionnement est normal
$Mfon = Norm$
| Le message courant au début est le message situation aérienne
$mCour = Msg\_Sas$

OPERATIONAL CONSTRAINTS

Preconditions

| Cette action a eu lieu si le message courant
| est le message situation aérienne
| le mode de fonctionnement de S5 est
| le mode normal
DirectSurveillance : $(mCour = Msg\_Sas) \wedge (CTO.UtilSurv.Mfon \neq Sev)$

Effects of Actions

| après cette action le message courant est le

| message situation aérienne
CSU.GenSituationAerienne.MAJSituationAerienne : [ ]
mCour := *Msg_Sas*
| après cette action, le message courant est le
| message Css
CSU.GenSituationAerienne.OrdreSurveillance : [ ]
mCour := *Msg_Css*

COOPERATION CONSTRAINTS

STATE INFORMATION

| mCour est toujours montré au commanditaire
$\mathcal{K}$ (*mCour.Commanditaire* / TRUE)
| mCour est toujours montré au COperations
$\mathcal{K}$ (*mCour.COperations* / TRUE)

Agent: **sda.COC.COperations**

DECLARATIONS

STATE COMPONENTS

| L'état mode de fonctionnement
Mfon instance-of *FONMODE**
⟶ *sda.CTO.CTheatre*
mCour instance-of *MESSAGE**
⟶ *sda.CC.InterControleMission*

ACTIONS

| L'action d'élaboration des ordres particuliers
OrdreParticuliers ⟶
sda.CC.InterControleMission, sda.VEC.Executant, sda.CTO.CTheatre, sda.CSU.InterSurveillance, sda.C
| L'action d'élaboration des comptes rendu d'opérations
CompteRenduOp ⟶ *sda.CTO.CTheatre*
| L'action état des forces
EtatForces ⟶ *sda.CC.InterControleMission, sda.CTO.CTheatre*
| action ordre général des forces
OrdreGeneral

BASIC CONSTRAINTS

INITIAL VALUATION

| le mode de fonctionnement au début est le mode normal
Mfon = *Norm*
| Le message courant au début est le message situation aérienne
mCour = *Msg_Sas*

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

[*OrdreGeneral*] *Mfon* = *Sev*
[*OrdreParticuliers*] *Mfon* = *Norm*

ACTION DURATION

| L'action OrdreParticuliers dure au plus 15 secondes
| *OrdreParticuliers* | $\leq$ *15"*
| L'action état des forces dure au plus 15 secondes
| *EtatForces* | $\leq$ *15"*

OPERATIONAL CONSTRAINTS

PRECONDITIONS

| L'action Ordre particuliers a eu lieu si l'agent est en mode de
| fonctionnement normal et que le message courant est le message ordre
| général
| des forces.
OrdreParticuliers : *(mCour = Msg_Ogf)* $\wedge$ *(Mfon = Norm)*

| L'action Ordre général a eu lieu si l'agent est en mode de
| fonctionnement secours vertical et que le message courant est le message
| situation aérienne
OrdreGeneral : *(Mfon = Sev)*$\wedge$ *(mCour = Msg_Sas)*
| cette action a eu lieu si le message courant
| est le message compte rendu de mission de forces
CompteRenduOp : *mCour = Msg_crmf*
| cette action a eu lieu si le message courant
| est le message disponibilité élémentaire des forces
EtatForces : *mCour = Msg_def*

EFFECTS OF ACTIONS

| La fin de cette action, le message courant est le message ordre général
| et le mode de fonctionnement est le mode normal

CTO.CTheatre.OrdreGeneral : [ ]
mCour := *Msg_Ogf*
Mfon := *Norm*

| La fin de cette action, le message courant est le message ordre de mission

CU.CommElem.OrdreMission : [ ]
mCour := *Msg_Omf*

| La fin de cette action, le message courant est le message compte rendu de mission

CU.CommElem.CompteRenduMission : [ ]
mCour := *Msg_Crmf*

| La fin de cette action, le message courant est le message disponibilité élémentaire

CU.CommElem.DispElem : [ ]
mCour := *Msg_Def*

OrdreGeneral : [ ]
Mfon := *Norm*
mCour := *Msg_Ogf*

COOPERATION CONSTRAINTS

Action Perception

| l'action directives de surveillance est toujours perçu par F2

$\mathcal{K}$ (*SupSurveillance.DirectSurveillance* / TRUE)

State Perception

| mCour est toujours perçu par F2

$\mathcal{K}$ (*SupSurveillance.mCour* / TRUE)

State Information

$\mathcal{K}$ (*mCour.CC.InterControleMission* / TRUE)
$\mathcal{K}$ (*Mfon.CTO.CTheatre* / TRUE)

Agent: **sda.COC.Commanditaire**

| agent commanditaire

DECLARATIONS

State Components

mCour instance-of *MESSAGE**
$\longrightarrow$ *sda.CC.Controleur*

BASIC CONSTRAINTS

Initial Valuation

| le message courant au début est le message situation aérienne
mCour = *Msg_Sas*

OPERATIONAL CONSTRAINTS

Effects of Actions

| à la fin de cette action, le message courant est le message
| évaluation de la ménace
CC.Controleur.EvalMenace : [ ]
mCour := *Msg_Emam*

| à la fin de cette action, le message courant est le message
| ordre d'engagement
CC.Controleur.OrdreEng : [ ]
mCour := *Msg_Oem*

| à la fin de cette action, le message courant est le message Rcm
CC.Controleur.RapportControle : [ ]
mCour := *Msg_Rcm*

| à la fin de cette action, le message courant est le message
| disponibilité élémentaire
CC.Controleur.DisponControle : [ ]
mCour := *Msg_Dcm*

COOPERATION CONSTRAINTS

Action Perception

| Le commanditaire perçoit toujours l'action directives
| de surveillance de la par du superviseur de la surveillance
$\mathcal{K}$ (*SupSurveillance.DirectSurveillance* / TRUE)

State Perception

| Le commanditaire perçoit toujours les valeurs du composant
| d'état mCour de la part du superviseur de la surveillance
$\mathcal{K}$ (*SupSurveillance.mCour* / TRUE)

State Information

| Le commanditaire montre toujours le message courant

| au contrôleur
$\mathcal{K}$ (*mCour.Controleur* / TRUE)

Society: **sda.CU**

| Dans cette société, nous avons
| le commandant élémentaire et l'associé
(*CommElem*)
(*Associe*)

Agent: **sda.CU.CommElem**

DECLARATIONS

State Components

mCour instance-of *MESSAGE**
⟶ *sda.CC.InterControleMission*
Mfon* instance-of *FONMODE

Actions

OrdreMission ⟶
sda.COC.COperations, sda.VEC.Executant, sda.CC.InterControleMission, sda.CSU.InterSurveillance
CompteRenduMission ⟶ *sda.COC.COperations*
DispElem ⟶ *sda.COC.COperations*

BASIC CONSTRAINTS

Initial Valuation

| le message courant a pour valeur Undef
mCour = UNDEF
| le mode de fonctionnement est le mode normal
Mfon = *Norm*

DECLARATIVE CONSTRAINTS

Action Duration

| L'action dure au plus 6 secondes
$| DispElem | \leq 6''$

OPERATIONAL CONSTRAINTS

PRECONDITIONS

| Le compte rendu de mission n'est réalisé qu'après reception
| du message exécution de mission
CompteRenduMission : $(mCour = Msg\_Emf)$
| Le Commanadant élémentaire déclenche l'action OrdreMission après
| avoir reçu le message Msg-Opf.
OrdreMission : $(mCour = Msg\_Opf)$

EFFECTS OF ACTIONS

| á la fin de cette action, le message courant de CommElem est le message
| ordre général des forces
CTO.CTheatre.OrdreGeneral : [ ]
$mCour := Msg\_Ogf$
| á la fin de cette action, le message courant de CommElem est le message
| ordre particulier des forces
COC.COperations.OrdreParticuliers : [ ]
$mCour := Msg\_Opf$
| á la fin de cette action, le message courant de CommElem est le message
| exécution de mission des forces
VEC.Executant.ExecutionMission : [ ]
$mCour := Msg\_Emf$

COOPERATION CONSTRAINTS

STATE PERCEPTION

| Le commandant élémentaire perçoit toujours
| le message courant de l'associé
$\mathcal{K}$ (*Associe.mCour* / TRUE)

STATE INFORMATION

| F3 montre toujours son message courant au
| responsable de l'interface avec contrôle de mission
$\mathcal{K}$ (*mCour.InterControleMission* / TRUE)

Agent: **sda.CU.Associe**

DECLARATIONS

State Components

mCour instance-of *MESSAGE**
⟶ *sda.CU.CommElem*

BASIC CONSTRAINTS

Initial Valuation

| Le message courant a pour valeur Undef au début
mCour = UNDEF

OPERATIONAL CONSTRAINTS

Effects of Actions

| à la fin de cette action, le message courant est le message
| évaluation de la ménace
CC.Controleur.EvalMenace : [ ]
mCour := *Msg_Emam*

| à la fin de cette action, le message courant est le message
| ordre d'engagement
CC.Controleur.OrdreEng : [ ]
mCour := *Msg_Oem*

| à la fin de cette action, le message courant est le message
| rapport de contrôle de mission
CC.Controleur.RapportControle : [ ]
mCour := *Msg_Rcm*

| à la fin de cette action, le message courant est le message
| disponibilité de contrôle
CC.Controleur.DisponControle : [ ]
mCour := *Msg_Dcm*

COOPERATION CONSTRAINTS

State Information

$\mathcal{K}$ (*mCour.CommElem* / TRUE)

Society: **sda.CSU**

| Dans cette société, nous avons deux agents :le générateur de la situation aérienne et
| le responsable de l'interface avec la surveillance.

(*GenSituationAerienne*)
(*InterSurveillance*)

Agent: **sda.CSU.GenSituationAerienne**

DECLARATIONS

State Components

Mfon* instance-of *FONMODE
mCour instance-of *MESSAGE**
⟶ *sda.CSU.InterSurveillance*

Actions

| action de génération de situation aérienne

MAJSituationAerienne ⟶
sda.CSU.InterSurveillance, sda.COC.SupSurveillance, sda.CDE.ElabPistes, sda.CTO.UtilSurv

| Il s'agit de la définition des ordres de surveillance à envoyer à l'Elaborateur des pistes

OrdreSurveillance ⟶
sda.CSU.InterSurveillance, sda.COC.SupSurveillance, sda.CDE.ElabPistes

BASIC CONSTRAINTS

Initial Valuation

| Au début le message courant a pour valeur Undef

mCour = UNDEF

| Le mode de fonctionnement est le mode normal

Mfon = *Norm*

DECLARATIVE CONSTRAINTS

Action Duration

| cette action dure au plus 7 secondes

| *OrdreSurveillance* | ≤ 7″

| cette action dure au plus 5 secondes

| *MAJSituationAerienne* | $\leq 5''$

OPERATIONAL CONSTRAINTS

PRECONDITIONS

OrdreSurveillance : *(mCour = Msg_Dss)*
MAJSituationAerienne : *(mCour = Msg_Pps)*

EFFECTS OF ACTIONS

COC.SupSurveillance.DirectSurveillance : [ ]
mCour := Msg_Dss

COOPERATION CONSTRAINTS

ACTION INFORMATION

```
L'action génération de la situation aérienne est toujours montrée au
responsable de l'interface avec la surveillance
```

$\mathcal{K}$ (*MAJSituationAerienne.InterSurveillance* / TRUE)

STATE INFORMATION

```
S2 montre toujours son message courant au responable
de l'interface avec la surveillance
```

$\mathcal{K}$ (*mCour.InteSurveillance* / TRUE)

Agent: **sda.CSU.InterSurveillance**

DECLARATIONS

STATE COMPONENTS

mCour instance-of *MESSAGE**

BASIC CONSTRAINTS

INITIAL VALUATION

```
Le message courant a comme valuer Undef au début
```

mCour = UNDEF

OPERATIONAL CONSTRAINTS

EFFECTS OF ACTIONS

CTO.CTheatre.OrdreGeneral : [ ]
mCour := *Msg_Ogf*
COC.COperations.OrdreParticuliers : [ ]
mCour := *Msg_Opf*
CU.CommElem.OrdreMission : [ ]
mCour := *Msg_Omf*
VEC.Executant.ExecutionMission : [ ]
mCour := *Msg_Emf*

COOPERATION CONSTRAINTS

ACTION PERCEPTION

| L'action Ordre de surveillance du générateur de la situation aérienne est toujours montrée au responsanle de l'interface avec la surveillance
$\mathcal{K}$ (*GenSituationAerienne.MAJSituationAerienne* / TRUE)
| L'action Ordre de surveillance est toujours montré au responsable de l'interface avec la surveillance
$\mathcal{K}$ (*GenSituationAerienne.OrdreSurveillance* / TRUE)

STATE PERCEPTION

| Cet état est toujours montré au responsable de l'interface avec la surveillance
$\mathcal{K}$ (*GenSituationAerienne.mCour* / TRUE)

Society: **sda.CDE**

| les agents de cette société sont les senseurs ainsi que l'élaborateur des pistes.
(*Senseur*)))
(*ElabPistes*)

Agent: **sda.CDE.Senseur**

| c'est l'agent Senseur
DECLARATIONS

STATE COMPONENTS

Status instance-of *STATUS**
$\longrightarrow$ *sda.CDE.ElabPistes*

ACTIONS

ExtractionPlots $\longrightarrow$ *sda.CDE.ElabPistes*
▌Il s'agit de l'action de détection
Detection $\longrightarrow$ *sda.CDE.ElabPistes*

BASIC CONSTRAINTS

INITIAL VALUATION

Status = *running*

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

[ ] *Status* = *Running*

ACTION DURATION

▌L'action d'extraction des plots dure au plus deux secondes
| *ExtractionPlots* | $\leq 2''$
▌L'action de détection dure au plus 3 secondes
| *Detection* | $\leq 3''$

OPERATIONAL CONSTRAINTS

PRECONDITIONS

Detection : *(mCour* = *Msg_Cds)*

COOPERATION CONSTRAINTS

ACTION INFORMATION

$\mathcal{K}$ (*ExtractionPlots.ElabPistes* / TRUE)
$\mathcal{K}$ (*Detection.ElabPistes* / TRUE)

STATE INFORMATION

$\mathcal{K}$ (*Status.ElabPistes* / TRUE)

Agent: **sda.CDE.ElabPistes**

DECLARATIONS

State Components

Mfon* instance-of *FONMODE
mCour instance-of *MESSAGE**

Actions

| Action directives de détection
DirectivesDetection
| action de pistage
Pistage

BASIC CONSTRAINTS

Initial Valuation

mCour = UNDEF
Mfon = *Norm*

OPERATIONAL CONSTRAINTS

Preconditions

| cette action a eu lieu si le message courant
| est le message contrôle de détection
DirectivesDetection : *mCour* = *Msg_Css*
| cette action a eu lieu si le message courant
| est le message Ps
Pistage : *mCour* = *Msg_Ps*

Effects of Actions

| à la fin de cette action, le message courant
| est le message Ps
Senseur.ExtractionPlots : [ ]
mCour := *Msg_Ps*
| à la fin de cette action, le message courant
| est le message contrôle de détection
DirectivesDetection : [ ]
mCour := *Msg_Cds*

| à la fin de cette action, le message courant
| est le message directive de surveillance
COC.SupSurveillance.DirectSurveillance : [ ]
mCour := *Msg_Dss*

| à la fin de cette action, le message courant
| est le message situation aérienne
CSU.GenSituationAerienne.MAJSituationAerienne : [ ]
mCour := *Msg_Sas*

| à la fin de cette action, le message courant
| est le message contrôle de surveillance
CSU.GenSituationAerienne.OrdreSurveillance : [ ]
mCour := *Msg_Css*

COOPERATION CONSTRAINTS

Action Perception

$\mathcal{K}$ (*Senseur.ExtractionPlots* / TRUE)
$\mathcal{K}$ (*Senseur.Detection* / TRUE)

State Perception

$\mathcal{K}$ (*Senseur.Status* / TRUE)

Society: **sda.CC**

| C'est le centre de contrôle, avec comme agent
| le Contrôleur, le responsable de l'Interface contrôle mission
(*Controleur*)
(*InterControleMission*)

Agent: **sda.CC.Controleur**

DECLARATIONS

State Components

mCour instance-of *MESSAGE**
⟶ *sda.VEC.Assujetti, sda.CU.Associe, sda.COC.Commanditaire*

Actions

| action d'évaluation de la ménace
EvalMenace ⟶ *sda.COC.Commanditaire, sda.CU.Associe*
| action rapport de contrôle
RapportControle ⟶ *sda.COC.Commanditaire, sda.CU.Associe*
| action de disponibilité de contrôle
DisponControle ⟶ *sda.COC.Commanditaire, sda.CU.Associe*
| action d'ordre d'engagement
OrdreEng ⟶ *sda.COC.Commanditaire, sda.CU.Associe, sda.VEC.Assujetti*

BASIC CONSTRAINTS

Initial Valuation

| Au début, le message courant est le message
| situation aérienne
$mCour = Msg\_Sas$

DECLARATIVE CONSTRAINTS

Action Duration

| Cette action dure 12 secondes
$| EvalMenace | \leq 12''$
| Cette action dure 4 secondes
$| RapportControle | \leq 4''$

OPERATIONAL CONSTRAINTS

Preconditions

| L'action évaluation de la méace a eu lieu
| si le message courant est le message situation aérienne
EvalMenace : $mCour = Msg\_Sas$

Effects of Actions

| à la fin de cette action, le message courant est le message Eim
VEC.Assujetti.ExecutionControle : [ ]
$mCour := Msg\_Eim$
| à la fin de cette action, le message courant est le message Emam
EvalMenace : [ ]
$mCour := Msg\_Emam$
| à la fin de cette action, le message courant est le message rapport de contrôle
RapportControle : [ ]
$mCour := Msg\_Rcm$

| à la fin de cette action, le message courant est le message disponibilité de contrôle

DisponControle : [ ]
mCour := *Msg_Dcm*

| à la fin de cette action, le message courant est le message ordre d'engagement

OrdreEng : [ ]
mCour := *Msg_Oem*

COOPERATION CONSTRAINTS

State Perception

$\mathcal{XK}$ (*Assujetti.mCour* / *mCour* = *Msg_Eim*)
$\mathcal{K}$ (*Commanditaire.mCour* / TRUE)

State Information

$\mathcal{K}$ (*mCour.Assujetti* / TRUE)
$\mathcal{K}$ (*mCour.Commanditaire* / TRUE)
$\mathcal{K}$ (*mCour.Associe* / TRUE)

Agent: **sda.CC.InterControleMission**

DECLARATIONS

State Components

Mfon instance-of *FONMODE**
mCour instance-of *MESSAGE**

Actions

OrdreParticuliers

BASIC CONSTRAINTS

Initial Valuation

Mfon = *Norm*
mCour = *Msg_Sas*

OPERATIONAL CONSTRAINTS

PRECONDITIONS

| L'action Ordre particuliers intervient si l'agent est en mode de
| fonctionnement secours vertical et que le message courant est le
| message ordre général des forces.

OrdreParticuliers : $(mCour = Msg\_Ogf) \wedge (Mfon = Sev)$

EFFECTS OF ACTIONS

| à la fin de cette action, le message courant est le message ordre
| général

CTO.CTheatre.OrdreGeneral : [ ]
$mCour := Msg\_Ogf$

| La fin de cette action, le message courant est le message ordre
| particulier

COC.COperations.OrdreParticuliers : [ ]
$mCour := Msg\_Opf$

| à la fin de cette action, le message courant est le message état des
| forces

COC.COperations.EtatForces : [ ]
$mCour := Msg\_Eff$

| La fin de cette action, le message courant est le message ordre de
| mission

CU.CommElem.OrdreMission : [ ]
$mCour := Msg\_Omf$

| à la fin de cette action, le message courant est le message exécution de
| mission

VEC.Executant.ExecutionMission : [ ]
$mCour := Msg\_Emf$

| à la fin de cette action, le message courant est le message ordre
| particulier
| et le mode de fonctionnement est le mode normal

OrdreParticuliers : [ ]
$Mfon := Norm$
$mCour := Msg\_Opf$

Society: **sda.VEC**

| Dans cette société nous avons l'assujetti, l'exécutant et

(*Assujetti*)
(*Executant*)

Agent: **sda.VEC.Assujetti**

DECLARATIONS

State Components

mCour instance-of *MESSAGE**
$\longrightarrow$ *sda.CC.Controleur*

Actions

ExecutionControle $\longrightarrow$ *sda.CC.Controleur*

BASIC CONSTRAINTS

Initial Valuation

mCour = UNDEF

OPERATIONAL CONSTRAINTS

Preconditions

| cette action a eu lieu si le message courant est le message
| ordre d'engagement
ExecutionControle : *mCour* = *Msg_Oem*

Effects of Actions

| à la fin de cette action, le message courant est le message Eim
ExecutionControle : [ ]
mCour := *Msg_Eim*
| à la fin de cette action, le message courant est le message ordre
| d'exécution de mission
CC.Controleur.OrdreEng : [ ]
mCour := *Msg_Oem*

COOPERATION CONSTRAINTS

State Perception

$\mathcal{XK}$ (*CC.Controleur.mCour* / *mCour* = *Msg_Oem*)

State Information

| Le message courant est toujours montré au contrôleur
$\mathcal{K}$ (*mCour.CC.Controleur* / TRUE)

Agent: **sda.VEC.Executant**

DECLARATIONS

State Components

mCour instance-of *MESSAGE**
$\longrightarrow$ *sda.CC.InterControleMission*

Actions

ExecutionMission $\longrightarrow$
sda.CU.CommElem, sda.CSU.InterSurveillance, sda.CC.InterControleMission

BASIC CONSTRAINTS

Initial Valuation

```
| mCour a comme valeur Undef au début
```
mCour = UNDEF

OPERATIONAL CONSTRAINTS

Preconditions

```
| L'exécution d'une mission n'intervient que si le message courant
| est le message ordre de mission des forces
```
ExecutionMission : *mCour* = *Msg_Omf*

Effects of Actions

```
| à la fin de cette action, le message courant
| est le message ordre particulier des forces
```
COC.COperations.OrdreParticuliers : [ ]
mCour := *Msg_Opf*

```
| à la fin de cette action, le message courant
| est le message ordre de mission des forces
```
CU.CommElem.OrdreMission : [ ]
mCour := *Msg_Omf*

COOPERATION CONSTRAINTS

Action Information

| cette action est toujours montrée au commandant élémentaire

$\mathcal{K}$ (*ExecutionMission.CU.CommElem* / TRUE)

| cette action est toujours montrée au responsable de l'interface avec la surveillance

$\mathcal{K}$ (*ExecutionMission.CSU.InterSurveillance* / TRUE)

| cette action est toujours montrée responsable de l'interface avec contrôle de mission

$\mathcal{K}$ (*ExecutionMission.CC.InterControleMission* / TRUE)

STATE INFORMATION

| L'exécutant montre toujours son message courant au responsable de l'interface contrôle mission

$\mathcal{K}$ (*mCour.InterControleMission* / TRUE)