MASTER EN SCIENCES INFORMATIQUES

Etude du projet SCR et réalisation d'un module utile pour l'ensemble d'outils SCR*

Eggen, Vincent

Award date:
1998

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage, B-5000 Namur

Etude du projet SCR et Réalisation d'un module utile pour l'ensemble d'outils SCR*

par
Vincent Eggen

Mémoire soumis en vue de l'obtention du grade de Maître en informatique

Vincent Eggen

Année académique 1997 – 1998

Etude du projet SCR et réalisation d'un module utile pour l'ensemble d'outils SCR*.

Auteur : Vincent EGGEN

Directeur : Dr Eric DUBOIS, FUNDP, Belgique

Co-Directeur : Dr Constance HEITMEYER, NRL, USA

Stage : Center for High Assurance Systems, Information Technology Division
Naval Research Laboratory
Washington, DC 20375 – 5320
United States of America

Remerciements

Je tiens tout d'abord à remercier les personnes qui m'ont aidé
durant ma période de stage et durant la rédaction du présent travail.
Mais je tiens aussi à remercier tous ceux qui de près ou de loin
ont rendu notre vie à tous un peu plus douce, plus riante,
par une attention, un sourire, un petit rien de tout les jours.
J'ai passé un peu plus de trois mois au Naval Research laboratory,
Washington, District of Columbia, USA.

Merci au Dr Heitmeyer et au Dr Dubois, mes superviseurs,
qui ont eut la gentillesse de m'encadrer durant cette période.
Merci ausi à toute l'équipe du NRL, à Bruce, Ramesh, Jim pour leurs conseils,
Merci à Michael pour sa présence journalière durant ces jours de labeur,
Merci à Todd pour son aide vraiment précieuse,
Un grand grand merci à Elvinia et Angelo pour leur bonne humeur proverbiale,
Merci à Cheryl et John pour cet accueil fantastique de chaleur,
Merci à mes parents pour leur soutien de tout les instants,
Merci à toi pour ta patience et ta douceur,
toi qui me rend chaque jour un peu meilleur. Merci Nathalie.

Vincent

Table des matières

Table des illustrations

Table des tables

Introduction

Analyser un outil et une méthode, cela peut paraître un exercice assez aisé. Il suffit presque de s'assoir, de lire la documentation et de noter les points qui vous paraissent importants.
Comparer deux méthodes qui paraissent concurrentes, cela peut déjà sembler moins évident. Une méthode, c'est bien souvent une approche soutenue par un point de vue. Afin de rester objectif, il va falloir essayer d'être le moins influencé par la confrontation inévitable de son propre point de vue avec celui d'autrui.
Mais comparer objectivement deux méthodes, dont une provient d'un milieu proche et que les gens de votre entourage supportent avec verve et conviction, c'est comme faire du funambulisme quand on a le vertige: On doit garder les yeux ouverts sans tomber d'un côté comme de l'autre.
C'est pourtant l'exercice auquel va nous convier la première partie de ce travail, puisqu'il va consister à comparer la méthode d'analyse des besoins **SCR**, issue des Etats-Unis, et l'approche de l'Institut d'Informatique du même domaine appelée *Albert II*.

Afin de donner au débat une dimension plus technologique, nous verrons dans une deuxième partie une application de support à l'un des outils de **SCR***, appelé JavaSimFront. Et nous découvrirons la technologie employée par cette application.

1ère partie
Le projet SCR

Chapitre 1
Ingénierie des besoins et SCR

1. Problématique de la rédaction du cahier des charges[1]

Où tout commence par un rêve...

Imaginez... Imaginez que vous êtes membre actif, peut-être même responsable, d'un grand et beau projet d'ingénierie. Imaginez encore que ce projet se termine dans les temps escomptés. Continuez à rêver que le budget alloué n'a pas été dépassé, et que les critères de qualité, de fiabilité que l'on vous avait demandé d'atteindre le furent. Imaginez en plus que la maintenance du système est claire, facile et bon marché, tout ça à la grande satisfaction de votre client. Allez, faites-vous plaisir et imaginez aussi les éloges de ce dernier pour vous et votre admirable travail... Le rêve !

Illustrations

La vision précédente pourrait être votre réalité de vie, bien qu'il soit malheureusement nécessaire de parler au conditionnel. En effet, le quotidien est rarement aussi idyllique. Les exemples d'échecs concernant de grands projets, informatiques ou non, sont nombreux. Selon certains rapports du MIT, entre 40 et 75% des systèmes d'informations seraient rejetés à un moment donné de leur cycle de développement [DUBOIS98a]. Ainsi, a titre de bref exemple, l'aéroport international de Denver aux Etats-Unis a été construit au début des années 90, au moment où la capacité de l'aéroport tout proche de Stapleton étaient largement dépassée, soulignant l'urgence du projet. L'ouverture de l'aéroport a cependant été retardée de 25 mois, pour un coût estimé d'un demi-million de dollars par jour [SECMM95]. Autre exemple édifiant, en 1992, le General Accounting Office (GAO), toujours aux Etats-Unis, a émis un rapport dénonçant les 8 années de retard du projet *Cheyenne Mountain Upgrade*, dépassant par-là le budget de plus de 600 millions de dollars. Le GAO enfonce le clou et regrette aussi que la réalisation finale soit moins fonctionnelle que le projet initial. Dans un rapport précédent, ce même organisme laissait entendre que ce genre de problème représente la norme plutôt que l'exception, et que la situation semble être identique dans le secteur privé [GAO92]. Le rêve semble donc bien n'être qu'un rêve. Mais devons-nous nous contenter de rêver, et espérer que lorsque l'on entame un projet, celui-ci s'achève avec plus ou moins de réussite ? Certes non. Et la recherche en ce domaine est loin d'être inactive...afin que le rêve devienne réalité.

Cycle de vie des systèmes techniques

L'objectif est donc de tenter d'améliorer le processus de réalisation d'un projet d'ingénierie. Considérons pour cela un simple cycle de vie des systèmes technique, cycle que l'on peu découper en quatre phases (Figure 1). La première phase consiste en l'identification du problème du client et en la validation par ce dernier de la solution proposée. La seconde phase concerne la réalisation et la mise en place du système. La troisième phase prend en compte la maintenance et l'évolution du système. Enfin, la quatrième phase s'attache aux problèmes rencontrés lorsque l'on arrête un système donné. Si chacune des phases de ce cycle a son importance, aucune n'est aussi cruciale que la première et à plus d'un titre. De ce processus, résulte généralement un cahier des charges, que les ingénieurs vont devoir tâcher

---

[1] Cette partie, introductive et informative, est essentiellement inspirée du cours du Professeur E. Dubois, *Méthodologies de Développement Logiciel – Matière Approfondie*, année académique 97-98. Le lecteur pourra s'y référer s'il désire un complément d'information.

de respecter. La réalisation du cahier des charges, c'est tout d'abord l'occasion pour les parties en place de pouvoir se rencontrer. C'est un *intérêt social* certain pour tous les acteurs de connaître les gens avec qui ils vont travailler. Ensuite, le cahier des charges est un document ayant *force de contrat*. Ainsi, en cas de litige et après acceptation du document par les parties, il pourra être reçu comme soutien légal en vue d'appuyer les doléances de l'un ou l'autre des acteurs. D'autre part, ce document permet d'affiner considérablement les coûts estimés du projet. Dans le cas de longs projets, on va ainsi pouvoir prévoir les coûts futurs en termes de besoins techniques, humains ou temporels ; et commencer par exemple à évaluer les évolutions budgétaires. Autre aspect qui prend de plus en plus d'importance, c'est l'aspect *maturation du processus de réalisation*. Depuis quelques années maintenant, sont apparues des normes de qualités auxquels les entreprises peuvent se référer. Sur base d'un audit positif, les entreprises peuvent alors obtenir une certification qui va jouer comme autant d'assurance-qualité pour le client, et comme avantage compétitif pour les entreprises. A titre d'exemple, citons la très connue norme ISO, ou les plus évoluées SPICE et CMM. Intéressons-nous quelques instants sur la norme CMM (Capability Maturity Model). Sans entrer dans les détails, cette norme classe les entreprises selon une échelle graduée de 0 à 5, le plus haut niveau de l'échelle représentant les entreprise dans lesquelles la gestion des processus internes est la plus évoluée, le niveau zéro correspondant au néant. Quelques études montrent qu'appliquer la norme CMM réduit sensiblement le nombre de détection d'erreurs (45%), ou encore augmente le gain annuel de productivité de près de 37% [SECMM95]. Alors qu'actuellement, près des trois quarts des entreprises évaluées ne dépassent pas le niveau 1, l'intérêt pour ce genre de norme est grandissant. Or, toujours en ce qui concerne CMM, pour passer du niveau 1 au niveau 2 de l'échelle, l'entreprise doit obligatoirement intégrer un processus de création de cahier des charges, soulignant ainsi l'importance donnée à ce document.

Figure 1. Cycle de vie des systèmes techniques.

Réduction des coûts

Nous avons donc vu quelques-uns des avantages que pouvait apporter la réalisation d'un cahier des charges dans le but d'améliorer la réalisation de systèmes. Mais l'un des avantages majeurs de l'ingénierie des besoins (nom donné au domaine d'activité lié à la rédaction du cahier des charges) est avant tout de permettre, sur base de ce cahier, la **détection d'erreurs** avant qu'elles ne coûtent trop chers à corriger. En effet, à titre d'exemple, près de 40% des erreurs logicielles sont faites durant la phase d'analyse, et le coût relatif de correction est de 1 à 50 entre cette même phase et la phase de test, si l'erreur est détectée

durant cette dernière [FAULK95]. Le choix de rédaction d'un cahier des charges n'est donc plus vraiment à faire si l'on désire faire passer le rêve à la réalité.

Propriétés du cahier des charges

Ecrire un cahier des charges est donc un moment important. Mais il ne suffit pas d'écrire ce cahier, il faut aussi *bien* l'écrire. Pour cela, son contenu doit respecter quelques propriétés. A savoir, les propriétés de :

- *Complétude* : il doit définir l'ensemble des informations nécessaires à l'élaboration du système, et rien d'autre.

- *Minimalité* : il ne doit pas définir plus que ce qui est nécessaire pour comprendre le système à réaliser[2].

- *Consistance* : il ne peut y avoir de contradiction entre les informations.

- *Non-Ambiguïté* : il ne peut y avoir de risque de conflit d'interprétation des informations contenues dans le cahier (une seule interprétation possible).

- *Modifiabliblité* : le cahier des charges doit être organisé de manière à être aisément modifiable. Soulignons ici le rôle de l'analyste qui doit alors identifier et organiser les parties susceptibles de changer de manière à ce que ces derniers aient un effet minimal sur le document.

- *Traçabilité* : le document doit être organisé de manière à ce que chaque spécification soit accessible rapidement et facilement.

- *Evolutivité* : même souci que pour la propriété de modifiabilité, mais ici il faut tenir compte des aspects évolutifs des spécifications ainsi que de l'analyse des impacts d'une évolution.

- *Indépendance vis-à-vis de la solution* : le document doit être exclusivement descriptif du système à mettre en place. Les solutions aux problèmes éventuellement soulevés ne doivent pas apparaître au sein du document.

En ce qui concerne le langage de description devant normalement respecter ces propriétés, avouons-le tout de suite, la forme d'expression la plus courante est le langage naturel, et généralement l'anglais. Mais bien que ce langage soit presque universellement compris, il est ambigu et imprécis, donc source d'erreurs. La tendance est alors d'utiliser des langages formels ou semi-formels, agrémentés ou non de langage naturel, afin de tenter de respecter au mieux les précédentes propriétés. En effet, les formalismes permettent des vérifications plus ou moins poussées des spécifications tout en autorisant à la fois l'utilisation de processus d'aide à la validation de ces dernières (prototypage, simulation,...), et une gestion plus intelligente des documents. Mais l'utilisation de langages formels ne va pas non plus sans

[2] Là où la propriété de complétude définit une borne inférieure à la qualité des informations spécifiées dans le cahier des charges, celle de minimalité en définit la borne supérieure.

rencontrer quelques inconvénients. Ainsi, les techniques formelles ne seraient généralement pas vraiment accessibles aux programmeurs qui ne sont pas des chercheurs, tout en étant coûteuses; les outils supportants ces langages ne seraient pas très ergonomiques car développés par et pour des spécialistes; l'utilisation de méthodes formelles ne serait pas consistante avec les nécessités commerciales liées à la concurrence; et enfin, la plupart des efforts sont des efforts de recherche plutôt que des cas liés à la réalité des développements [NEUMANN96]. Bref, il y a sans doute un peu de vrai dans chacune de ces objections. Mais pour reprendre la position de [NEUMANN96], il y a aussi des résultats significatifs émergeants, et nous en verrons un peu plus loin.

Activités de l'ingénierie des besoins

Il nous reste à aborder le problème du processus de création du cahier des charges. Le schéma suivant (Figure 2) en fournit un bon descriptif :

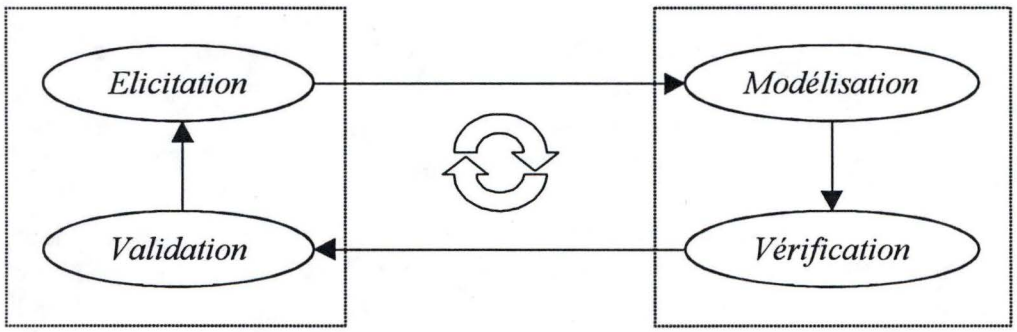


Figure 2. Activités de l'ingénierie des besoins.

On distingue quatre activités réparties entre deux domaines, celui du client et celui de l'analyste. L'activité d'*élicitation* consiste à explorer, acquérir et vérifier les besoins du client par le biais d'interview, d'introspection, d'observation du système existant et d'acquisition d'informations dans le domaine d'application. La *modélisation* consiste en l'élaboration d'une représentation alternative et conceptuelle du problème de manière à capturer au mieux la sémantique du projet, et proposer ainsi une description abstraite de celui-ci. Ensuite, l'analyste peut opérer une *vérification* du cahier des charges en terme des propriétés exposées plus haut. Généralement, cette vérification est exécutée manuellement. Mais, dans la mesure où le modèle formel du langage de spécification le permet, des techniques d'analyse automatisée peuvent être employées. Dans le cas où une ambiguïté ou une incohérence serait détectée, le document serait alors revu et corrigé. Enfin, le cahier doit être *validé* par le client [ZOWGHI95]. Mais le processus global ne s'arrête pas là. En effet, ce processus de **négociation** entre experts de domaines différents (clients et analystes) doit amener à une bonne compréhension du projet par chacune des parties. En outre, le client est versatile, son opinion peut changer à tout moment, et la vue des clients entre eux de leur problème est souvent inconsistante. De plus, l'analyste doit toujours garder en tête les objectifs stratégiques de l'entreprise pour le projet, en non pas, par exemple, proposer par principe une solution informatique là où elle ne serait pas nécessaire. Pour les raisons évoquées plus haut, le processus global est donc un processus qui se renouvelle constamment et dont le résultat évolue au fil des raffinements et des évolutions successives, afin de parvenir à un accord

commun sur le contenu du document final. On peut dès lors représenter (Figure 3) une vue globale l'évolution de l'état du processus selon trois critères : l'état des spécifications, le type de langage retenu pour exprimer le cahier des charges, et l'état des opinions des différents acteurs [POHL96]. On constate que le chemin suivi par le processus n'est pas direct, soulignant ainsi toute la difficulté de parvenir à une situation satisfaisante pour tout le monde.

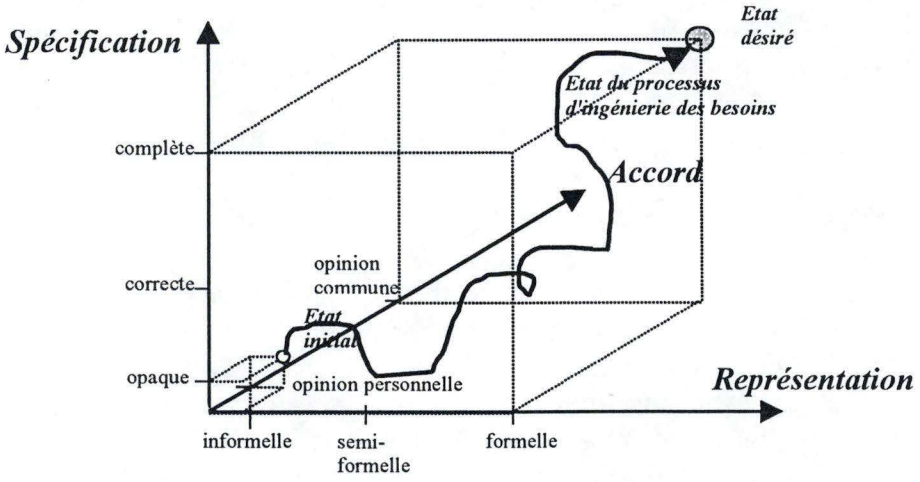


Figure 3. The *Magic Cube.*

Ingénierie des besoins vs génie logiciel

Voici donc brossé en quelques lignes une large définition de l'ingénierie des besoins. En guise de conclusion, il peut être intéressant de se pencher sur les différences qui existent entre ce domaine et le domaine du génie logiciel, tant la frontière entre les deux domaines peut sembler floue. En fait, si l'on revient sur le cycle de vie des systèmes (Figure 1), les deux activités se retrouvent dans deux phases différentes et successives (négociation – réalisation/mise en place). En outre, l'intérêt du génie logiciel se porte sur la **partie logiciel** du système à élaborer, là où l'autre domaine se penche sur **tous** les éléments

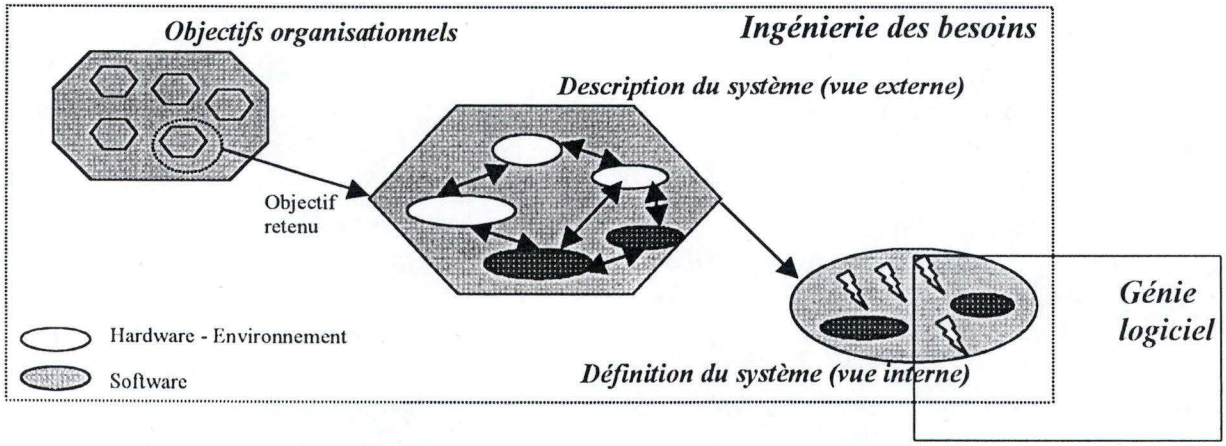


Figure 4. Ingénierie des besoins vs génie logiciel.

du système, et les relations de ces éléments avec l'**environnement**. C'est pour cette raison que l'on a introduit la notion de contraintes fonctionnelles (sur le rôle de chaques composants du système) et celle de contraintes non-fonctionnelles (qui concerne par exemple les critères de

temps de réalisation, de qualité ou de sécurité); les contraintes non-fonctionnelles étant typiques au contenu du cahier des charges. L'activité est aussi différente entre les deux domaines. Le premier adopte une approche **transformationnelle** (des spécifications vers le code), alors que le deuxième tente de créer une **modélisation** du monde tel qu'il est connu par les clients. Enfin, en génie logiciel, une **preuve formelle** de la réalisation est parfaitement possible, alors que l'attitude **descriptive** de l'ingénierie des besoins rend ce genre de preuve normalement difficile. Voyez la figure précédente (Figure 4) pour une représentation du rapport existant entre les deux domaines.

2. SCR, ou la discipline au service de l'ingénierie des besoins

Genèse du projet SCR

Un des premiers organes officiels ayant ressenti très tôt la nécessité d'introduire des formalismes au sein de l'ingénierie des besoins fut le Département de la Défense Américaine (U.S. Department of Defense – DoD). En effet, ce dernier, grand consommateur de projets généralement forts coûteux, était donc soucieux de limiter ses dépenses. Ainsi, aux milieux des années 80, le DoD dépensait notamment près de 10 milliards de dollars rien que pour des projets liés aux technologies de l'information, et le budget total devait évoluer relativement vite durant les années qui suivirent. Or, le DoD prit rapidement conscience de plusieurs lacunes dont souffraient leurs projets, et qui coûtaient relativement chers à l'organisme. En particulier :

- les projets étaient souvent démarrés sans définitions claires et non ambiguës de ce qu'ils étaient censés faire,
- de fortes lacunes dans la réalisation de la documentation étaient constatées (peu ou pas de qualité dans l'écriture, pas d'organisation des documents, imprécisions, pas d'utilité pratique),
- il n'y avait pas ou peu de réutilisations des systèmes préexistants,
- il n'y avait non plus pas ou peu d'interdépendances entre projets pouvant pourtant jouer sur les rendements d'échelles,
- ...etc. [CLEMENTS85].

Avec l'avènement de l'informatique embarquée, dans l'armement notamment, un besoin évident se faisait sentir : celui d'un projet dans lequel on choisirait un ensemble de principes et de méthodes que l'on appliquerait rigoureusement aux systèmes de développement actuels du DoD en les raffinant ou les éliminants, améliorant le système de représentation des modèles de travail des différents projets, et que les chercheurs suivrait. C'est dans ce cadre que fut créer le projet Software Cost Reduction (**SCR**) :

"*The NRL (Naval Research Laboratory) team led by David Parnas, initiated the Software Cost Reduction project in 1978 to demonstrate the feasibility and effectiveness of advanced software engineering techniques by appliyng them to a real program, the Operational Flight Program (OFP) for the A-7E aircraft. To demonstrate that (then academic) techniques like information hiding, formal specification, abstract interfaces, and cooperating sequential processes could help make software easier to undertstand, maintain, and change, the A-7E OFP was re-engineered* [FAULK95]."

Evolutions du projet

De ce projet résultèrent les bases de ce qui devait devenir la méthode **SCR**. Et depuis 20 années maintenant, le projet suit son cours. L'équipe de David Parnas a tout d'abord étendu la méthode d'analyse des besoins à un modèle mathématique standard pour les systèmes embarqués [PARNAS91]. Ensuite, le docteur Faulk et ses collègues ont intégré des techniques graphiques et orientées objets à l'approche **SCR**. Ils ont aussi défini un processus complet d'analyse des besoins [FAULK92]. Enfin, le travail de l'équipe du docteur Heitmeyer a étendu le travail du docteur Parnas à un modèle formel [HEITMEYER95a]. Ce modèle autorise une notation et une sémantique permettant l'utilisation de techniques d'analyses à des fins de vérification de propriétés de complétude et de consistance. De ce modèle formel a été développé un ensemble d'outils logiciels prototypes supportant ces techniques. Les expériences qui suivirent démontrèrent que la vérification automatique par de spécifications permet de retrouver des erreurs oubliées lors des inspections manuelles. Les outils actuels permettent aussi la création et l'édition de spécifications **SCR**. L'équipe du docteur Heitmeyer a aussi développé un simulateur, ce dernier permettant l'exécution symbolique du modèle formel sous-jacent à l'analyse, et donc permettant aussi l'analyse du comportement d'un système préalablement défini [HEITMEYER93] [HEITMEYER95b] [HEITMEYER95c] [FAULK95]. L'objectif actuel du NRL est de démontrer la capacité qu'a la technologie **SCR** d'être effective dans un milieu industriel. Le laboratoire travaille donc pour le moment en partenariat serré avec plusieurs industriels afin d'appliquer la technologie **SCR** à des problèmes réalistes et où la sécurité joue un rôle prépondérant, les résultats obtenus permettant le transfert de la technologie vers l'industrie ainsi que vers une éventuelle acquisition de cette dernière par des développeurs commerciaux.

Voici donc planté le décor du projet **SCR**. La suite du présent document vous propose de rentrer un peu plus profondément dans les arcanes du modèle. Suivez le guide...

Chapitre 2
Le modèle SCR

1. La méthodologie de développement SCR

Introduction

Le projet **SCR**, c'est d'abord une méthodologie de développement. Les bases en ayant été fondées dans la première moitié des années 80, on ressent dans cette approche l'influence des méthodologies classiques de type SA/RT (Structured Analysis/Real Time) avec un brin de décomposition fonctionnelle, ces méthodes ayant aussi vues le jour durant cette période [DUBOIS98][FAULK95]. La méthodologie a aussi connu des évolutions [FAULK92], mais afin de ne pas nous disperser, nous nous en tiendrons à la version originale de cette dernière, tel que présentée dans [CLEMENTS85].

Revue de la méthode

La méthode **SCR** commence, on oserait presque dire "évidemment", par l'**analyse des besoins**. Le document orienté-**SCR** résultant est divisé en sections biens définies, chacune d'entre-elles définissant un aspect des besoins. Une section doit par exemple définir, pour chaque sortie du système, les valeurs acceptées à tout moment par l'environnement du système. De manière à ce que l'évolution du système soit aisée, le cahier des charges doit aussi contenir une définition des éléments qui sont susceptibles de changer dans le futur. C'est le langage supporté par cette première phase de la méthodologie qui va retenir toute notre attention dans la suite de l'exposé. Signalons qu'en 1980, l'approche de ce processus était résolument orientée vers l'ingénierie logicielle.

On retrouve ensuite l'analyse de la **structure des modules**. Par "module", il faut entendre une assignation simple de travail, chaque module pouvant être décomposé en sous-modules (d'où la structure d'arbre du document résultant). La volonté de cette découpe est double. D'une part, toujours dans une optique de facilitation des évolutions futures, les modules définissent des entités simples et dont la modification ne risque pas de causer des modifications non-désirées du comportement du système. D'autre part, les modules sont construits selon le modèle de la boîte noire, avec donc une partie publique (l'interface) et une partie privée (cachée). Cette découpe facilite la gestion des processus sensibles en terme de sécurité.

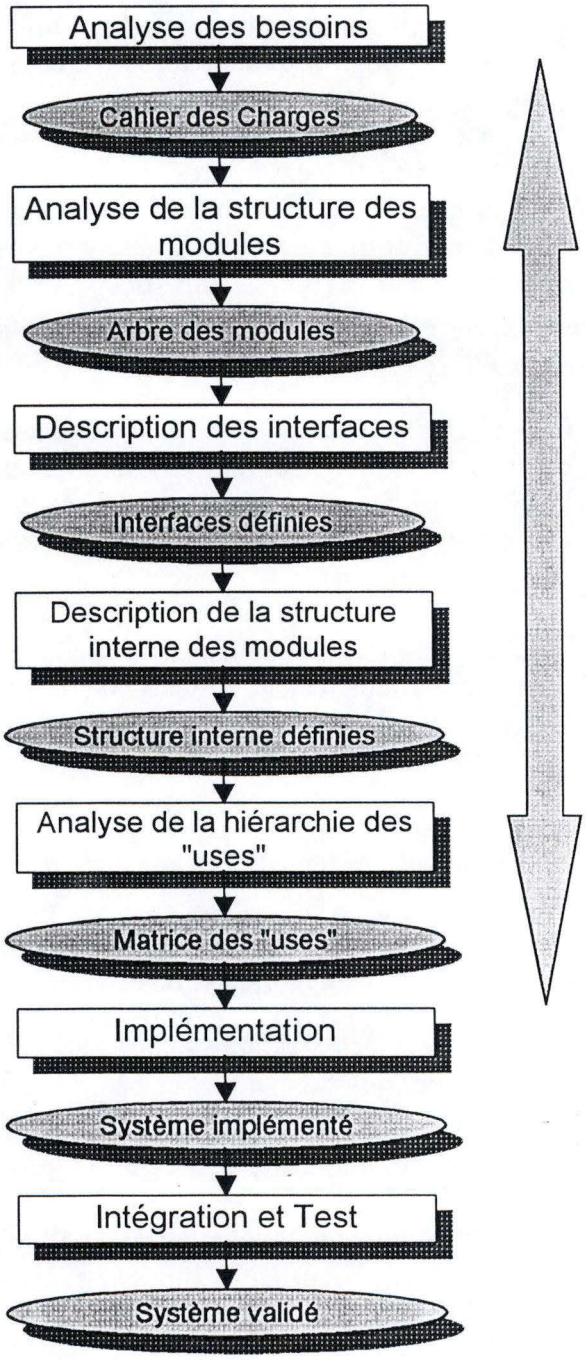


Figure 5. La méthodologie SCR

La phase suivante va donc consister à **définir les interfaces** des différents modules. C'est une phase importante, car l'interface d'un module est la seule chose que vont connaître les autres modules, d'où l'importance d'une description précise. L'essentiel du contenu de ces spécifications consiste en une liste des programmes invoquables par les programmes des autres modules, les paramètres de ces invocations, la définition des comportements attendus et non attendus de ces invocations, ainsi que des contraintes de temps et d'exactitudes.

Après les interfaces, il faut définir la **structure interne des modules**. Pour chaque programme invoquable, on va définir une fonction mathématique décrivant ses effets sur la structure des données internes. Pour chaque valeur retournée par le module, on définit une autre fonction qui assigne les valeurs de la structure de données aux valeurs retournées. Les événements non désirés sont aussi décrit ici, ainsi que la manière d'y répondre.

L'étape suivante est la description de la **hiérarchie des "uses"**, c'est-à-dire la description des modules qui requièrent la présence d'autres modules. Ceci va permettre de pouvoir créer des sous-ensembles de programmes mutuellement dépendants distincts du système global. Cette distinction est importante afin de pouvoir réaliser des tests sur les sous-systèmes sans nécessairement avoir besoin des autres parties du système. Le document final est généralement une matrice binaire dans laquelle la position (a,b) est vraie si l'exécution du programme a requiert la présence du programme b.

Les phases suivantes d'**implémentation et de test/intégration** devraient se révéler grandement simplifiées et plus rapide que par les méthodes jusqu'alors utilisées par les ingénieurs. L'application de la méthode permet en effet une détection d'erreur efficace, une génération de tests rapide et rend le projet facilement gèrable en terme de documentation et de ressources humaines.

Tout au long des processus de la méthode, une **vérification formelle** est appliquée. Les vérifications sont réalisées bien entendus sur les faits, mais aussi sur les principes. Ainsi, on vérifie que les spécifications sont conformes à une évolution future du système, que les documents respectent les règles d'organisations de **SCR**, que l'ensemble des éléments sont clairement définis, etc. C'est une des parties les plus importantes de la méthode.

Conclusion

Ceci n'est qu'une brève présentation de la méthode **SCR**. Pour une description plus détaillée ou un cas réel d'application, le lecteur peut consulter [PARNAS78]. L'engouement de l'époque pour cette méthode laisse entendre qu'elle est encore largement utilisée par l'industrie [FAULK95]. Même si en avant des nombreux avantages qu'avançait [CLEMENTS85] (réduction du temps de développement, facilité de gestion des ressources, indépendance entre les équipes de développement et le développement proprement-dit,...), on doit admettre que cette manière de procéder par décomposition fonctionnelle oblige à introduire des choix d'implémentation de manière prématurée [FAULK95]. De plus, la méthode est essentiellement orientée "logicielle" plutôt que "système". Les extensions du langage soutenu par la méthode d'analyse des besoins tentent dès lors de résoudre ces objections [HEITMEYER95c].

2. La méthodologie d'analyse des besoins SCR

Introduction

Dès 1980, un des soucis majeurs du NRL a été de mettre au point une méthodologie permettant de réaliser une analyse des besoins efficace et produisant un document sur lequel l'ensemble du développement pouvait reposer sans risque d'erreurs. C'est pourquoi l'ensemble de l'effort de recherche s'est porté sur la création d'une notation permettant au cahier des charges de remplir ces critères. En 1987, les propriétés (orientées **SCR**) des documents créés lors de l'analyse des besoins, ainsi que leurs grands principes de construction étaient énoncés. Les premières *bases formelles* de la notation **SCR** étaient également jetées [CLEMENTS87].

Propriétés et principes de construction de l'analyse des besoins SCR

Les propriétés de l'analyse des besoins **SCR** sont exprimées en fonction de leurs objectifs, l'objectif primaire étant de fournir une **spécification complète du comportement** du futur système. Le cahier des charges dans sa forme finale doit donc contenir toutes les informations nécessaires à la construction du système. On rejoint ici la notion de *complétude* énoncée au chapitre 1.1. Les objectifs additionnels (comparés aux propriétés vues précédemment) proposés par [CLEMENTS87] sont :

- *Ne spécifier que le comportement externe du système* : Aucune implémentation particulière ne doit être sous-entendue. Correspond à l'*indépendance des spécifications par rapport à la solution.*
- *Etre facile à changer* : Propriété de *modifiabilité.*
- *Servir comme outils de référence* : Propriété de *traçabilité.*
- *Prévoir les évolutions du système* : Propriété d'*évolutivité.*
- *Spécifier les réponses des événements non désirés* : Sous-ensemble de la propriété de *complétude.*
- *Spécifier les contraintes d'implémentations* : Sous-ensemble de la propriété de *complétude.*

Parallèlement, quelques grands **principes de construction** ont été mis en avant, tels que les principes de :

- *Séparation des difficultés* : Ce principe implique que l'information soit divisée en parties distinctes et indépendantes, afin de favoriser la gestion et l'évolutivité du document.
- *Etre aussi formel que possible* : La sémantique des langages formels étant normalement plus précise, concise, consistante et complète. Adéquat pour respecter aussi la propriété de *non-ambiguïté.*
- *Eviter les redondances* : Afin de satisfaire la propriété de *minimalité.*

On peut dès lors constater que l'ensemble des propriétés et principes évoqués rejoignent assez bien celles que nous avons proposées dans le premier chapitre. Soulignons toutefois que l'analyse est basée sur une étude du comportement visible et attendu du système, ce qui nous emmène au modèle sur lequel repose le rapport du système avec son environnement.

3. Le modèle des quatre variables

Introduction

Avant d'aborder la notation **SCR** et son modèle formel, abordons l'un des problèmes majeurs rencontré par l'approche **SCR** vue jusqu'à présent. En effet, jusqu'au début des années 90, l'approche du NRL consistait surtout à analyser les logiciels et non pas les systèmes dans leur globalité. Il fallut attendre la thèse de van Schouwen [HEITMEYER95c] pour que le modèle de spécification existant soit étendu au concept de spécifications orientées système. En 1991, Parnas et Madey introduisent le modèle des quatre variables, qui formalise l'approche de van Schouwen. C'est sur base de ce modèle que va se construire le modèle formel **SCR**.

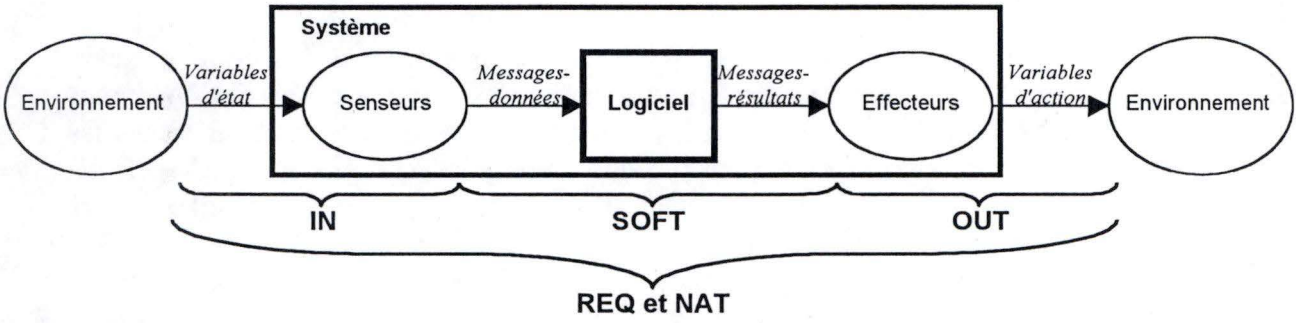


Figure 6. Le modèle des quatre variables.

Présentation du modèle

Le modèle des quatre variables représente les besoins du système par un ensemble de relations mathématiques appliquées sur quatre ensembles de variables appelées *variables d'état, variables d'action, messages-données et résultats* (Figure 6). Une *variable d'état* (ou *monitored variable*) représente une quantité environnementale qui influence le comportement du système, une *variable d'action* (ou *controlled variable*) une quantité environnementale que le système maîtrise. Les relations REQ et NAT, des valeurs d'état vers les valeurs d'actions, définissent une spécification de type boîte noire représentant le comportement désiré du système. NAT correspond à l'ensemble des valeurs possibles et capture les contraintes sur le comportement du système comme par exemple celles imposées par les lois de la physique. REQ définit des contraintes additionnelles imposées par le système en définissant les relations que le système doit maintenir entre les variables d'état et les variables d'action. Les *messages-données* (ou *input data items*) , représentants les senseurs (ou *input devices*) du système, sont des ressources mise à la disposition du système afin de reconnaître les valeurs d'état. La relation IN définit la relation d'équivalence entre ces valeurs d'états et les messages-données. On définit de manière équivalente la relation OUT, mais entre les valeurs d'action et les messages-résultats (représentants les effecteurs ou *output devices*). La relation SOFT représente l'implémentation de la partie logicielle [HEITMEYER96b].

Les constructions SCR

Pour définir les relations du modèle des quatre variables, d'autres constructions utiles sont introduites. Tout d'abord, il y a la classe des *modes* (*mode class*), qui est une machine à

états dont les états sont appelés les *modes systèmes* et dont les transitions sont déclenchées par des événements. Les systèmes complexes sont définis par plus d'une classe des modes opérant en parallèle. Est ensuite introduite le classe des *termes*. Un terme est toute fonction de variables entrantes (variable d'état ou message-donnée), de modes ou d'expression qui rend les spécifications concises. La construction suivante est la *condition* qui est un prédicat définit sur une ou plusieurs entités du système[3] à un moment donné dans le temps. Un *événement*, une autre construction, survient lorsqu'une variable entrante change de valeur. Enfin, un autre événement spécial appelé *événement de condition* (ou *conditioned event*), survient quand une condition devient vraie.

Illustration

Afin d'illustrer les précédentes constructions, nous allons en construire un exemple basé sur le problème du monte-charge (MC). Vous trouverez la définition de ce problème à l'annexe 1 du présent document.

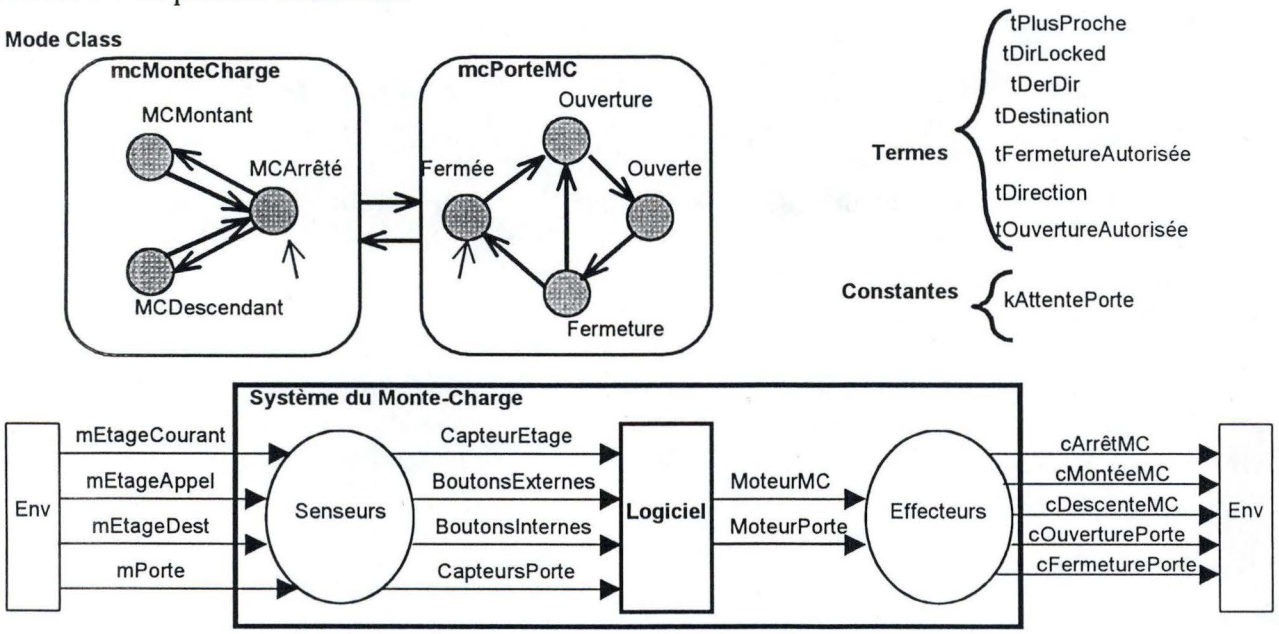


Figure 7. Les constructions SCR pour le problème du Monte-Charge.

A la lecture de la définition du problème, on se rend très vite compte que celle-ci va devoir être adaptée pour pouvoir représenter correctement le comportement du MC. A la lecture du problème, on peut déjà identifier une série de composants du système. Ainsi, le seul moyen qu'a le système de savoir où il doit se rendre, c'est soit de connaître l'étage où un utilisateur désire aller par l'appui par exemple d'un des boutons de destination, soit de connaître sur quel bouton d'appel un utilisateur a poussé. De manière moins évidente, le comportement décrit de la porte implique qu'il doit exister un senseur qui indique au système dans quel état est la porte. Identiquement, il doit y avoir une autre série de senseurs indiquant à quel étage se trouve le MC. Le système utilise donc quatre senseurs (les capteurs d'étage, les boutons internes au MC et les boutons d'appels, externes au MC, et les capteurs des portes). Afin que le MC puisse se déplacer, il faut un moteur. De même pour que les portes puissent

[3] Une entité du système est soit une variable entrante ou résultante (Message-résultat ou variable d'action), un mode ou une expression.

s'ouvrir. Ce sont les deux effecteurs. Chacun des senseurs est représenté par une variable d'état préfixée par un petit 'm', alors que les variables d'action sont préfixées par un petit 'c' (par convention). Notez que là où il a été choisi de représenter un senseur par une variable d'état, les effecteurs sont représentés par plusieurs variables d'action, représentant chacune une des actions du système sur l'environnement. Afin de rendre les spécifications plus claires, a été introduite une série de termes (**tDestination**, **tOuvertureAutorisée**, **tFermetureAutorisée**, **tDerDir**,...) dont certains sont nécessaires pour la clarté des spécifications, mais dont d'autres sont nécessaires pour la justesse des spécifications. Une constante **kAttentePorte** a aussi été introduite et représente le temps minimum d'ouverture des portes. Dans le cas de cet exemple, la distinction a été faite entre les modes du MC, et ceux de la porte. Nous avons donc deux classes de modes opérant en parallèle [HEITMEYER96b].

4. La notation SCR

Introduction

La notation **SCR** introduite avec l'analyse de l'A-7 utilise une *notation de type tabulaire*. La raison principalement invoquée pour ce choix est la facilité de compréhension des tables, et l'aisance avec laquelle elles sont utilisées en milieu industriel [HEITMEYER95a]. On peut distinguer deux types de tables, les dictionnaires et les tables fonctionnelles.

Les dictionnaires SCR

On distingue 5 sortes de dictionnaires différents : le dictionnaire des **variables**, celui des **types**, des **constantes**, des **classes des modes** et des **assertions**. L'ensemble du contenu de ces dictionnaires est décrit en annexe 2. Voici ceux correspondant à l'exemple du Monte-Charge (Tables 1-4) :

Dictionnaire des variables

Nom	Type	Valeur initiale	Précision	Commentaires
mEtageCourant	yEtage	0	N/A	Etage actuel du MC.
mEtageAppel	yEtage	0	N/A	Dernier étage où un bouton d'appel a été pressé.
mEtageDest	yEtage	0	N/A	Dernier étage dont le bouton de sélection a été pressé.
mPorte	yEtatPorte	Fermée	N/A	Etat de la porte devant laquelle se trouve le MC.
cArrêtMC	Switch	Off	N/A	Le moteur arrête le MC.
cMontéeMC	Switch	Off	N/A	Le moteur monte le MC.
cDescenteMC	Switch	Off	N/A	Le moteur descend le MC.
cOuverturePorte	Switch	Off	N/A	Le moteur ferme la porte.
cFermeturePorte	Switch	Off	N/A	Le moteur ouvre la porte.
tOuvertureAutorisée	Boolean	True	N/A	Le moteur de la porte est autorisé à ouvrir la porte quand cette variable est True
tFermetureAutorisée	Boolean	False	N/A	Le moteur de la porte est autorisé à fermer la porte quand cette variable est True
tDestination	yEtage	0	N/A	Prochain arrêt du MC.
tDirection	yDir	Stop	N/A	Direction prise par le MC.
tDerDir	yDerDir	Haut	N/A	Dernière direction prise par le MC
tDirLocked	Boolean	False	N/A	Retient si un étage destination se trouvant dans le sens précédent au MC a été proposé.
tPlusProche	Boolean	False	N/A	L'appel ou la destination soumise n'est pas plus proche que la destination du MC

Table 1. Dictionnaire des variables du MC.

Dictionnaire des types

Nom	Type de base	Unités	Valeurs légales	Commentaires
yEtage	Integer	N/A	[-1,3]	Représentation des étages du bâtiment.
yEtatPorte	Enumerated	N/A	Ouvert,Fermé,Bloquée	La porte peut être signalée Ouverte, Fermée ou Bloquée.
yDir	Enumerated	N/A	Haut,Stop,Bas	Directions possibles du MC.
yDerDir	Enumarated	N/A	Haut,Bas	Dernirères directions possible pour le MC

Table 2. Dictionnaire des types du MC.

Dictionnaire des constantes

Nom	Type	Valeur	Commentaires
kAttentePorte	Integer	5000	Le temps d'ouverture de la porte du MC est fixé à 5 secondes (5000 ms).

Table 3. Dictionnaire des constantes du MC.

Dictionnaire des modes

Nom	Modes	Mode initiale	Commentaires
mcMonteCharge	MCMontant,MCDescendant,MCArrêté.	MCArrêté	Le MC peut soit être en train de monter, de descendre ou être à l'arrêt.
mcPorteMC	Fermée,Ouverture,Ouverte,Fermeture	Fermée	La porte du MC peut soit être fermée, s'ouvrir, être ouverte ou se fermer.

Table 4. Dictionnaire des modes du MC.

Le *dictionnaire des constantes* assigne des valeurs à des constantes des spécifications, alors que le *dictionnaire des types* décrit des types définis à partir de types de base. Pour chaque classe des modes des spécifications, le *dictionnaire des modes* liste les modes de ces classes avec ses valeurs initiales. Le *dictionnaire des variables* présente l'ensemble des variables d'état, d'action ainsi que les termes des spécifications, avec pour chacune, les types de données, les valeurs initiales et la précision avec laquelle elles doivent s'exprimer.

Les tables fonctionnelles SCR

On distingue 3 sortes de tables fonctionnelles différentes, leurs objectifs communs étant de représenter au mieux le comportement du système : la table des **événements**, la table des **conditions**, et la table des **transitions.** Ces trois types de tables représentent le cœur des spécifications **SCR**. Celles-ci appliquées à notre exemple donnent les tables de l'annexe 3 dont en voici une partie significative :

Table des transitions :

mcMonteCharge

Mode Source	Evénement	Mode Destination
MCArrêté	@T(tDestination>mEtageCourant) WHEN mPorte=Fermée	MCMontant
MCArrêté	@T(tDestination<mEtageCourant) WHEN mPorte=Fermée	MCDescendant
MCMontant, MCDescendant	@T(tDestination=mEtageCourant)	MCArrêté

Table 5. Table des transitions pour mcMonteCharge.

Le événements en **SCR** sont signifié par "@T". Ainsi, on peut distinguer deux sortes d'événements majeurs, l'*événement simple* @T(tDestination = mEtageCourant) (l'étage courant du MC est devenu celui de sa destination), et l'*événement conditionné*

@T(tDestination>mEtageCourant) WHEN mPorte=Fermée (la future destination du MC est devenue plus haute que la position actuelle du MC alors que la porte est fermée).
Dans une table des modes, le système ne passe d'un mode à un autre que lorsqu'un événement spécifié survient, les autres événements étants ignorés. Le système est toujours dans un mode donné. Ainsi, dans le cas de l'exemple, l'état initial du MC est arrêté, portes fermées.

Explications de la table :
Ligne 1: Si le MC est arrêté et que l'étage de destination attribué au MC est haut dessus de l'étage actuel du MC et que la porte est fermée, alors le MC monte.
Ligne 2: Si le MC est arrêté et que l'étage de destination attribué au MC est en dessous de l'étage actuel du MC et que la porte est fermée, alors le MC descend.
Ligne 3: Si le MC monte ou descend et que l'étage de destination attribué au MC est le même que l'étage actuel du MC, alors le MC s'arrête.

Table des événements:

tDirection

Mode	Evénements		
MCArrêté	@T(mPorte=Fermé) WHEN (tDestination>mEtageCourant)	CHANGED(tDestination) WHEN NOT (mPorte=Fermé)	@T(mPorte=Fermé) WHEN (tDestination< mEtageCourant)
MCMontant	@T(mEtageCourant < tDestination)	@T(tDestination= mEtageCourant)	NEVER
MCDescendant	NEVER	@T(tDestination= mEtageCourant)	@T(mEtageCourant > tDestination)
tDirection	Haut	Stop	Bas

Table 6. Table des événements pour tDirection.

La table d'événement ci-dessus définit le terme **tDirection** en fonction de tDestination, mPorte, mEtageCourant et mPorte. Tout comme la table des transitions, les tables d'événement ne considèrent que les événements qui sont spécifiés dans celles-ci. Notez l'événement CHANGED qui signale la modification de la valeur de la variable ou du terme entre parenthèses. NEVER signifie que dans un mode donné, aucun événement ne peut être considéré pour le terme défini.

Explications de la table :
Case 1, 1: Si le MC est à l'arrêt et que la porte est fermée quand une destination assignée au MC est située plus haut que l'étage actuel du MC, alors la direction prise par le MC est le haut.
Case 1, 2: Si le MC est à l'arrêt et que la destination assignée au MC a changé alors que la porte n'est pas fermée, alors le MC ne bouge pas.
Case 1, 3: Si le MC est à l'arrêt et que la porte est fermée quand une destination assignée au MC est située plus bas que l'étage actuel du MC, alors la direction prise par le MC est le bas.
Case 2, 1: Si le MC monte et que le MC est situé plus bas que la destination du MC, alors la direction est le haut.
Case 2, 2: Si le MC monte et que le MC arrive à la destination assignée, alors le MC se stoppe.
Case 2, 3: Aucun événement ne peut donner la direction du bas au MC lorsque celui-ci monte.
Case 3, 1: Aucun événement ne peut donner la direction du haut au MC lorsque celui-ci descend.
Case 3, 2: Si le MC descend et que le MC arrive à la destination assignée, alors le MC se stoppe.
Case 3, 3: Si le MC descend et que le MC est situé plus bas que la destination du MC, alors la direction est le bas.

Table des conditions :

cDescenteMC

Mode	Conditions	
MCArrêté, MCDescendant	tDirection = Bas	NOT (tDirection =Bas)
MCMontant,	False	True
cDescenteMC	On	Off

Table 7. Tables des conditions pour cArrêtMC.

La table précédente définit la variable d'action **cDescenteMC** comme fonction de mcMonteCharge et de tDirection. La lecture en est fort simple :

Explications de la table :
Colonne 1: Si le MC est arrêté ou descend est que la direction assignée est le bas, la commande de descente passe à On.
Colonne 2: Si le MC descend ou est à l'arrêt et que le direction assignée n'est pas le bas, ou que le MC descend, la commande de montée est à Off.

Un point important est à soulever par rapport à la suite de l'exposé: les tables de conditions définissent des *fonctions totales*, alors que les tables d'événements et de transitions définissent des *fonctions partielles*. Ceci s'explique parce que certains événements ne peuvent survenir quand certaines conditions sont vraies. Ainsi, dans l'exemple précédent, l'événement @T(mcMonteCharge=Descendant) WHEN mcMonteCharge = Montant ne peut survenir, car le MC doît passer obligatoirement par l'état Arrêté avant de commencer à descendre. Dans d'autre situations, des événements peuvent survenir et ne rien changer aux valeurs des variables définies dans les tables d'événement ou de transition. Le modèle formel se base cependant sur des fonctions définissant les tables d'événements ou de transitions *complètes*, en assignant à une variable son ancienne valeur quand la table ne définit pas explicitement la nouvelle valeur de la variable.

5. Analyse de l'exemple en SCR

Afin de pouvoir évaluer la notation **SCR**, analysons comment quelques-unes des contraintes majeures de l'exemple du MC sont représentées en **SCR**.

- *"Quand le MC arrive à un étage où il doit s'arrêter, ses portes s'ouvrent pour une période de 5 secondes. Après quoi, celles-ci se referment."*

Cette contrainte implique plusieurs sous-contraintes : pour que les portes s'ouvrent, le MC doit s'arrêter; et les portes doivent être ouvertes durant 5 secondes.

La solution de la première sous-contrainte se trouve dans la table des événements définissant le terme **tOuvertureAutorisée**. Ce terme autorise l'ouverture de la porte lorsque le MC arrive à destination ou que le bouton d'appel est pressé à un étage lorsque le MC est à ce même étage, portes fermées. Ce dernier cas n'a pas été spécifié par le problème, et montre donc un cas où il a fallut compléter la définition en langage naturel. Le passage du mode *Fermée* vers le mode *Ouverture* (table des transitions **mcPorteMonteCharge**, annexe 3) n'est alors autorisé que lorsque **tOuvertureAutorisée** est vrai.

tOuvertureAutorisée

Mode	Evénements	
Fermée	@T (mEtageCourant=mEtageAppel) WHEN (tDirection=Stop)OR @T(tDestination=mEtageCourant)	@T(Inmode)
Ouverture	@T(Inmode)	@T(mPorte=Ouvert)
Ouverte	NEVER	@T(Inmode)
Fermeture	@T(mPorte=Bloqué)	@T(Inmode)
tOuvertureAutorisée	true	false

Table 8. Table des événements tOuvertureAutorisée.

La deuxième partie de la contrainte est reprise dans la table 9 définissant le terme **tFermetureAutorisée**. La deuxième ligne de cette table impose que la durée durant laquelle la porte n'est pas autorisée à se fermer n'excède pas **kAttentePorte**. Ce qui est intéressant, c'est la notation *Duration* qui introduit la notion de durée d'événement, c'est-à-dire la durée de temps durant laquelle le système se trouve dans un état donné [HEITMEYER97a]. Le passage du mode *Ouverte* vers le mode *Fermeture* (table des transitions **mcPorteMonteCharge**, annexe 3) n'est alors autorisé que lorsque **tFermetureAutorisée** est vrai.

tFermetureAutorisée

Mode	Evénements	
Fermée,Ouverture	NEVER	@T(Inmode)
Ouverte	@T(Duration(mPorte=Ouvert)=kAttente Porte)	@T(Duration(mPorte=Ouvert)<kAttente Porte)
Fermeture	@T(Inmode)	@T(mPorte=Bloqué) OR @T(mPorte=Fermée)
tFermetureAutorisée	true	false

Table 9. Table des événements tFermetureAutorisée.

- *"Lorsque plusieurs destinations sont choisies, le MC ira vers l'étage le plus proche dans le sens correspondant au sens du service précédent [...] Sinon, il ira à l'étage le plus proche. "*

⇒ *"Lorsque plusieurs destinations sont choisies..."*

Ici, on fait référence à l'introduction d'une séquence de choix de destination. L'attribution des destinations pour le MC se faisant quel que soit l'état du système, il n'est pas nécessaire de comparer tous les choix entre-eux. Il suffit simplement de comparer la destination actuelle du MC avec la destination proposée.

⇒ *"...le MC ira vers l'étage le plus proche dans le sens correspondant au sens du service précédent..."*

La notion de sens de service précédent justifie l'introduction du terme **tDerDir**, qui représente donc la dernière direction prise par le MC avant que celui-ci s'arrête (cnfr table **tDerDir**, annexe 3).

Afin de déterminer si la destination proposée est la plus proche par rapport à la destination actuelle, on utilise le terme **tPlusProche** (Table **tPlusProche**, annexe 3). A tout moment, ce terme indique si la dernière demande de service correspond à un étage plus proche que la destination actuelle du MC.

La véritable difficulté de cette contrainte par rapport à la notation **SCR**, c'est le fait que l'assignation de la destination proposée, qui implique une direction opposée à la précédente direction, ne se fait que s'il n'y a pas déjà eut une assignation de destination impliquant une direction équivalente à la direction précédente. Cette situation va obliger d'introduire le terme **tDirLocked** (Table **tDirLocked**, annexe 3). Celui-ci va agir comme témoin d'une situation antérieur et qui doit avoir des implications sur la situation actuelle. Enfin, l'attribution de la destination se fait dans la table suivante :

tDestination

Mode	Evénements	
MCArrêté	CHANGED(mEtageDest) WHEN ((tDirLocked AND tPlusProche) OR (NOT(tDirLocked) AND tPlusProche))	CHANGED(mEtageAppel) WHEN ((tDirLocked AND tPlusProche) OR (NOT(tDirLocked) AND tPlusProche))

MCMontant	@T(mEtageDest<tDestination) WHEN (tDerDir = Haut)	NEVER
MCDescendant	@T(mEtageDest>tDestination) WHEN (tDerDir = Bas)	@T(mEtageAppel<tDestination) WHEN (tDerDir = Bas)
tDestination	mEtageDest	mEtageAppel

Table 10. Table des événements tDestination.

La case 1-1 de la table spécifie donc cette situation où, quand le MC est à l'arrêt, et que la destination proposée change alors qu'il n'y a pas de changement de direction (avec la direction précédente) et que cette destination est la plus proche, ou bien alors qu'il y a changement de direction du MC et que la destination est la plus proche, la nouvelle destination devient la proposition de destination.

Première remarque...

Si on peut déjà faire quelques remarques concernant la notation **SCR**, c'est qu'elle exige parfois l'introduction un peu artificielle de paramètres, comme le fut **tDirLocked**. En effet, l'intérieur des tables **SCR** est évalué sur les valeurs actuelles des paramètres s'y trouvant. Dans le cas où il serait nécessaire d'évaluer une expression dont la valeur doit être évaluée précédemment, il faut nécessairement introduire un terme qui évalue l'expression (et qui donc nécessite la création d'une nouvelle table). Ceci enlève beaucoup de naturel à une écriture déjà fort rigoureuse et seule une lecture approfondie des tables permettrait dès lors de retrouver l'idée avancée par le problème initial.

6. Sémantique des spécifications SCR

Ce qui suit est une brève présentation informelle du *modèle formel* **SCR**. Le lecteur pourra trouver la définition complète de ce modèle dans [HEITMEYER95b] et dans [HEITMEYER96b]. Afin de fournir une sémantique précise et détaillée de la méthode **SCR**, le modèle représente le système par un *automate à états finis*, et exprime les constructions **SCR** en terme de cet automate. Ce modèle est un cas spécial du Modèle des Quatre Variables, décrivant les variables généralement continues d'état ou d'action comme si elles étaient discrètes. Le travail de [HEITMEYER96a] sur les systèmes hybrides (à variables discrètes et continues) étend le présent modèle aux variables continues, et propose une nouvelle définition du modèle incluant les versions continues des variables de temps et de précision.

Le modèle formel

Dans la version simple du modèle, on définit un ensemble d'entités $RF=\{r_1,r_2,...,r_n\}$ du système, ainsi qu'une fonction spéciale TY qui associe chaque entité à sa valeur légale. Définissons aussi l'état s, représenté par une fonction qui associe chaque entité de RF à sa valeur dans s. Le modèle formel est représenté par une machine à états finis $\Sigma = (S, s_0, E^m, T)$, où S est l'ensemble des états, $s_0 \in S$ l'état initial, E^m l'ensemble des événements, et T une fonction qui associe un événement au passage d'un état initial à un état résultant. Le système commence à l'état s_0. Quand un événement signale la modification d'une variable d'état, la transformation T spécifie la valeur de toutes les variables d'actions, les termes, et les classes des modes dans le nouvel état. La transformation T est la composition de fonctions plus petites appelées *fonctions tabulaires*. Ces fonctions tabulaires sont dérivées des tables **SCR** qui définissent les variables d'action, les termes, et les classes de modes. Le model formel

requiert que chaque table de conditions, d'événements, et de transitions satisfassent à certaines propriétés, celles-ci garantissant que chaque fonction tabulaire est totale.

Afin de calculer la valeur d'une entité dans un nouvel état, la transformation T peut utiliser les valeurs des entités de l'ancien et du nouvel état. On peut décrire les entités pour lesquelles une entité donnée dépend directement dans le nouvel état (signalé par un ') par les relations de dépendance D_{new},D_{old}, et D de RF×RF. Pour des entités r_i et r_j, la paire $(r_i, r_j) \in D_{new}$ ssi r'_j est un paramètre de la fonction définissant r'_i ; la paire $(r_i, r_j) \in D_{old}$ ssi r_j est un paramètre de la fonction définissant r'_i; et $D = D_{new} \cup D_{old}$. Afin d'éviter toute circularité dans les définitions, on définit un ordre partiel par l'utilisation de D^+_{new}, la fermeture transitive de D_{new}. Les faits que les fonctions tabulaires soient totales et que les entités de RF sont partiellement ordonnées assurent que la transformation T est une *fonction complète* (pour chaque événement, au moins un nouvel état du système est défini.).

Pour d'illustrer le modèle formel, considérons le système du monte-charge décrit précédemment. Dans ce système, l'ensemble des entités *RF* comprend les variables d'états (mEtageCourant, mEtageAppel, mEtageDest et mPorte), les classes des modes mcMonteCharge et mcPorteMC et les termes tDestination, tDirection, tOuvertureAutorisée, tFermetureAutorisée et tDerDir, ainsi que toutes les variables d'action. La définition des types (fonction TY) pour chacun d'eux est :

TY(mEtageCourant) = [-1,3]
TY(mEtageAppel) = [-1,3]
TY(mEtageDest) = [-1,3]
TY(mPorte) = {Ouvert, Fermé, Bloqué}
TY(mcMonteCharge) = {MCArrêté, MCMontant, MCDescendant}
TY(mcPorteMC) = {Fermée, Ouverture, Ouverte, Fermeture}
TY(tDirection) = {Haut, Stop, Bas}
TY(tDestination) = [-1,3]
TY(tOuvertureAutorisée) = {TRUE, FALSE}
...

La relation de dépendance des nouveaux états D_{new} pour le système du monte-charge est :

{(cArrêtMC, mcMonteCharge), (cArrêtMC, tDirection), (mcMonteCharge, tDestination), (mcMonteCharge, mEtageCourant), (mcMonteCharge, mPorte), (tDirection, mPorte), (tDirection, tDestination), (tDirection, mEtageCourant), (tDirection, mcMonteCharge), (tDestination, mEtageDest), (tDestination, mEtageCourant), (tDestination, tDerDir), (tDestination, mEtageAppel), (tDestination, mcMonteCharge), (tDerDir, mcMonteCharge), (cMontéeMC, mcMonteCharge), (cMontéeMC, tDirection), (cDescenteMC, tDirection), (cDescenteMC, mcMonteCharge), (cOuverturePorte, mcMonteCharge), (cOuverturePorte, tOuvertureAutorisée), ...}

Si on applique les définitions de [HEITMEYER96b] aux tables, on peut dériver les fonctions tabulaires suivantes pour les classes de modes, les termes et les variables d'action. C'est de ces fonctions que sont dérivées les différentes tables de la notation, que l'on devine aisément à la lecture de ces quelques exemples :

mcMonteCharge' =

MCMontant	*si* mcMonteCharge = MCArrêté ∧ mEtageCourant' < tDestination' ∧ mEtageCourant *not* < tDestination ∧ mPorte = Fermée
MCDescendant	*si* mcMonteCharge = MCArrêté ∧ mEtageCourant' > tDestination' ∧ mEtageCourant *not* > tDestination ∧ mPorte = Fermée
MCArrêté	*si* (mcMonteCharge = MCMontant ∧ mEtageCourant' = tDestination' ∧ mEtageCourant ≠ tDestination) ∨ (mcMonteCharge = MCDescendant ∧ mEtageCourant' = tDestination' ∧ mEtageCourant ≠ tDestination)
mcMonteCharge	*sinon.*

...

tDerDir' =

Bas	*si* (mcMonteCharge' = MCMontant ∧ mcMonteCharge ≠ MCMontant)
false	*si* (mcMonteCharge' = MCDescendant ∧ mcMonteCharge ≠ MCDescendant)
tDerDir	*sinon.*

...

cArrêtMC=

On	*si* mcMonteCharge = MCArrêté ∨ ((mcMonteCharge = MCMontant ∨ mcMonteCharge = MCDescendant) ∧ tDirection = Stop
Off	*si* (mcMonteCharge = MCMontant ∨ mcMonteCharge = MCDescendant) ∧ tDirection ≠ Stop

...

Déterminisme

Le comportement du système tel qu'il est décrit par le modèle possède une partie déterministe et une partie non déterministe. Alors que la transformation T est déterministe, les événements produits par l'environnement sont non déterministes. Les variables d'état participants à ces événements peuvent être représentées par une simple machine à états finis avec un état initial, un ensemble possible d'états et une fonction de transition entre les états. Par exemple, la variable d'état **mPorte** a comme état initial la valeur **Fermée**, l'ensemble des états possibles {**Bloquée**, **Fermée**, **Ouverte**}, et l'ensemble des transitions {(**Ouverte**, **Fermée**), (**Fermée**, **Bloquée**), (**Fermée**, **Ouverte**), (**Bloquée**, **Ouverte**)}. Pour la variable **mEtageCourant**, l'état initial est 0, les états possibles l'ensemble {-1, 0, ..., 3}, et la relation de transition qui autorise **mEtageCourant** de changer d'une unité d'un état à un autre $\{ (x,x') \mid 1=|x' - x|, -1 \leq x \leq 3, -1 \leq x' \leq 3\}$. Un des postulats important du modèle est celui que seulement une seule variable d'état change à chaque transition d'état (*The One Input Assumption*). Ainsi, si le système est à son état initial, sur un ensemble d'événements possibles, ce postulat autorise seulement la survenance d'un seul de ces événements comme transition vers le prochain état du système. Bien que les spécifications **SCR** puissent être non-déterministes, le modèle initial peut être formulé en terme de fonctions et est donc restreint à décrire des systèmes déterministes. Dans certains cas, le non-déterminisme n'est pas nécessairement une erreur, le déterminisme pouvant amener à de la sur-spécification.

[HEITMEYER96b] argumente cependant en faveur du déterminisme des spécifications, ceux-ci étant "plus facile à spécifier, corriger, et analyser que les systèmes non-déterministes."

7. Intérêt et justification du formalisme

Vérification de consistance

Un des intérêts majeurs de posséder un modèle formel est que l'on peut y appliquer une série de vérifications automatiques, notamment en ce qui concerne la consistance des spécifications. Ces vérifications déterminent alors si l'analyse est bien écrite et indépendante d'un état particulier du système [HEITMEYER96b]. Ces vérifications pouvant être réalisées sans exécution du système, on parle d'*analyse statique*. Voici les vérifications automatiques que l'on peut réaliser à partir du modèle **SCR** :

- *Justesse de la syntaxe*. Chaque composant des spécifications a une syntaxe propre.
- *Correction des types*. Chaque variable possède un type défini, et toute définition de type est satisfaite.
- *Complétude entre les définitions des variables et les classes de modes*. La valeur de chaque variable d'action, termes, et modes est définie.
- *Valeurs d'initialisation*. Toutes les valeurs initiales sont définies pour toutes les classes de mode, variables et termes. Les valeurs initiales des entités définies par les tables de condition ne sont pas nécessairement requises, car elles peuvent être dérivées des tables.
- *Joignabilité*. Chaque mode des classes de mode peut être atteint statiquement à partir du mode initial des classes.
- *Déterminisme*. Afin de rendre les spécifications déterministes, chaque condition, événement, et table de transition doit satisfaire le fait que dans un état donné, chaque variable d'action, mode ou terme soit défini de manière unique.
- *Couverture*. Chaque table de condition doit satisfaire à cette propriété, c'est-à-dire que chaque variable décrite dans une de ces tables doit être définie partout dans son domaine.
- *Pas de circularité*. Il ne doit exister aucune circularité entre les éléments de l'ensemble des dépendances. Cette propriété vérifie que les entités sont partiellement ordonnées.

L'ordre des vérifications est évidemment important. Ainsi, la vérification de la syntaxe doit précéder la vérification des types, celle-ci devant précéder la vérification de la propriété de couverture. La vérification du déterminisme et de la couverture sont les plus lourdes des vérifications. Vérifier une de ces conditions revient à évaluer si une expression logique est une tautologie. Ainsi, la vérification du déterminisme de deux entrées c_1 et c_2 d'une ligne d'une table de condition revient à vérifier que $c_1 \wedge c_2 = \mathit{faux}$. En ce qui concerne l'évaluation

de couverture d'une ligne de c_1 à c_n conditions d'une ligne de table de condition, on doit évaluer l'expression $c_1 \vee c_2 \vee ... \vee c_n = vrai$. Ce genre de méthode de vérification entraîne un temps de calcul exponentiel selon la taille des expressions à évaluer, mais il semblerait que ce problème soit éludé dans la pratique. En effet, la partition du problème en états successifs du système par l'utilisation des modes revient à décomposer en petits sous-problèmes indépendants, échappant dès lors à l'explosion exponentielle du temps de calcul. "*Our experience with consistency checking is that the number of subproblems and the size of each subproblem grow rather slowly... Thus we expect that our techniques would not suffer the state explosion problem that plagues techniques such as model checking...* [HEITMEYER96b]". Enfin, ce genre de décomposition entraîne une localisation plus aisée des erreurs dans les différentes tables, et par ce fait simplifie la correction.

Vérification de propriétés

Un autre des intérêts majeurs du modèle formel **SCR** est que l'on peut y appliquer une série de *vérifications de propriétés* de spécifications données. Ces vérifications déterminent si à un moment de la vie du système, un des états du système vérifie une certaine propriété. Ainsi, à titre d'exemple, on pourrait tâcher de vérifier si toute demande de service du MC est satisfaite à un moment donné ou endéans un certain temps. Ces vérifications peuvent être réalisées de deux manières : soit en utilisant le simulateur de spécification, soit en faisant ce que l'on appel de la vérification de modèle, ou vérification de propriétés. Dans les deux cas, comme on réalise une exécution symbolique sur le système décrit, on parle d'*analyse dynamique*. Ces deux types de vérifications seront abordés dans le chapitre suivant traitant des outils qui supportent **SCR**.

High assurance systems

L'un des domaines de prédilection d'application des spécifications **SCR** sont les systèmes qui doivent fournir des services satisfaisant certaines propriétés critiques (High Assurance Systems). Ces propriétés sont des propriétés de sécurités (pas de modifications non-autorisées du comportement, pas d'accès possible à des données critiques,...), de sûreté (où un comportement non prévu du système ne doit pas mettre la vie de gens en danger) et de temps-réel (les résultats doivent être fournis dans un temps donné). Or, cette catégorie de système peut se retrouver dans les centrales nucléaires, les systèmes médicaux, certaines catégories d'avions de ligne ou militaires, des systèmes balistiques, ou encore des satellites de télécommunication. Donc, autant de systèmes coûteux et dans lesquels les erreurs risquent de se payer au prix fort, économiquement comme humainement. D'où l'importance de détecter rapidement les erreurs, dès l'analyse des besoins. En ce qui concerne ce genre de système, l'expérience semble montrer que les méthodes formelles sont efficaces. Ainsi, du point de vue d'un des acteurs sans doute les plus expérimenté : "*[...] Since development life cycles, failure models, and verification methods that have performed well for hardware systems are not always optimal for systems that include a significant software component, the identification and evaluation of better verification techniques for such systems will be an ongoing need within the systems development discipline. This need, coupled with substantial improvement in Formal Method (FM) techniques and tools, have made FM specification and verification a technique for consideration by most projects delivering a product that includes software. FM complement inductive techniques such as testing and help projects move beyond traditional quality ceilings* [NASA95]". L'approche des gens du NRL est équivalente, et leurs choix se

sont donc naturellement portés vers une méthode formelle à haut niveau de vérifiabilité, parce que les méthodes formelles peuvent réduire les erreurs en réduisant les ambiguïtés, les imprécisions et en mettant à jour les inconsistances et les incomplétudes de manière automatique [HEITMEYER95c].

Première conclusion

La notation **SCR** bénéficie d'un haut niveau de formalisme et d'opérationnalité. La notation tabulaire, avantageuse pour l'expression de contraintes temporelles simples, devient moins évidente à l'emploi lorsque ces contraintes se complexifient et nécessite la création un peu artificielle de termes. De plus, l'analyste doit nécessairement bien comprendre le modèle sous-jacent s'il veut parvenir à écrire une spécification qui "tourne". Cela se justifie certainement par le fait que ce langage est orienté ingénieurs, c'est-à-dire qu'il s'adresse essentiellement à des gens pour qui la notation tabulaire est un moyen courant d'expression. Du point de vue du client, ce type de cahier de charge souffre malheureusement d'un manque évident d'expressivité pour qui n'est pas du tout habitué à ce type de représentation, ou pour qui n'a pas vraiment le temps de déchiffré le contenu des tables. Mais le choix des gens du NRL se justifie pleinement si l'on considère que l'ensemble des systèmes élaborés par eux sont des systèmes critiques où la sécurité joue un rôle primordial. Les compétences d'**SCR** en cette matière sont évidentes, et valide ces choix.

Chapitre 3
*L'Outil SCR**

1. Introduction

Nous avons précédemment découvert la méthode **SCR**, la notation **SCR** et le modèle formel que supporte la notation. Nous avons vu que sur base de cette méthode, il était possible de réaliser des vérifications automatiques de propriétés. Il est donc temps maintenant de découvrir l'outil, ou plutôt l'ensemble d'outils, supportant la notation **SCR** et développé depuis quelques années maintenant par les gens du NRL.

L'outil se nomme **SCR***, est codé en C++ et tourne sur station SPARC sous le système Solaris. **SCR*** contient un *éditeur de spécifications* pour construire et présenter les spécifications, un *analyseur formel* afin de tester les propriétés sur divers propriétés, et un *simulateur* pour exécuter symboliquement les spécifications. L'analyseur formel possède deux outils distincts, un *vérificateur de consistance*, qui teste les spécifications sur les propriétés du modèle formel précédemment présenté, et un *vérificateur de propriétés*, qui va permettre de vérifier la corrélation entre les spécifications et certaines propriétés critiques (Chapitre 2.7.).

Nous passerons rapidement en revue chacun de ces outils en nous attardant sur les plus intéressants, puis on abordera les raisons d'être de cet ensemble ainsi que les apports et résultats de celui-ci sur des cas réels.

2. L'éditeur de spécifications SCR

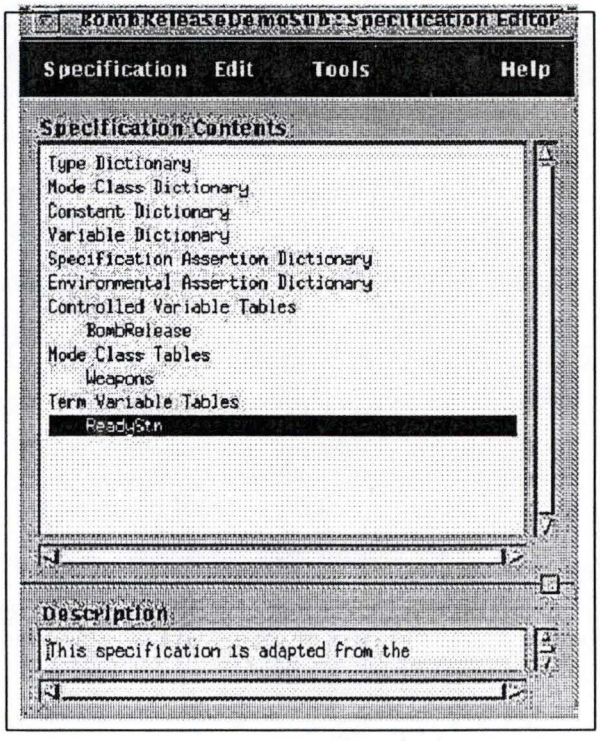


Figure 8. L'éditeur de spécifications SCR (exemple).

Dictionnaires et tables

Afin de créer, modifier ou présenter les spécifications, l'utilisateur peut utiliser l'éditeur de spécifications. Dans cet éditeur, il est possible d'éditer l'ensemble des dictionnaires et des tables (Figures 8 à 10). Rappelons qu'à l'annexe 2 se trouvent les définitions de l'ensemble des tables évoquées ici.

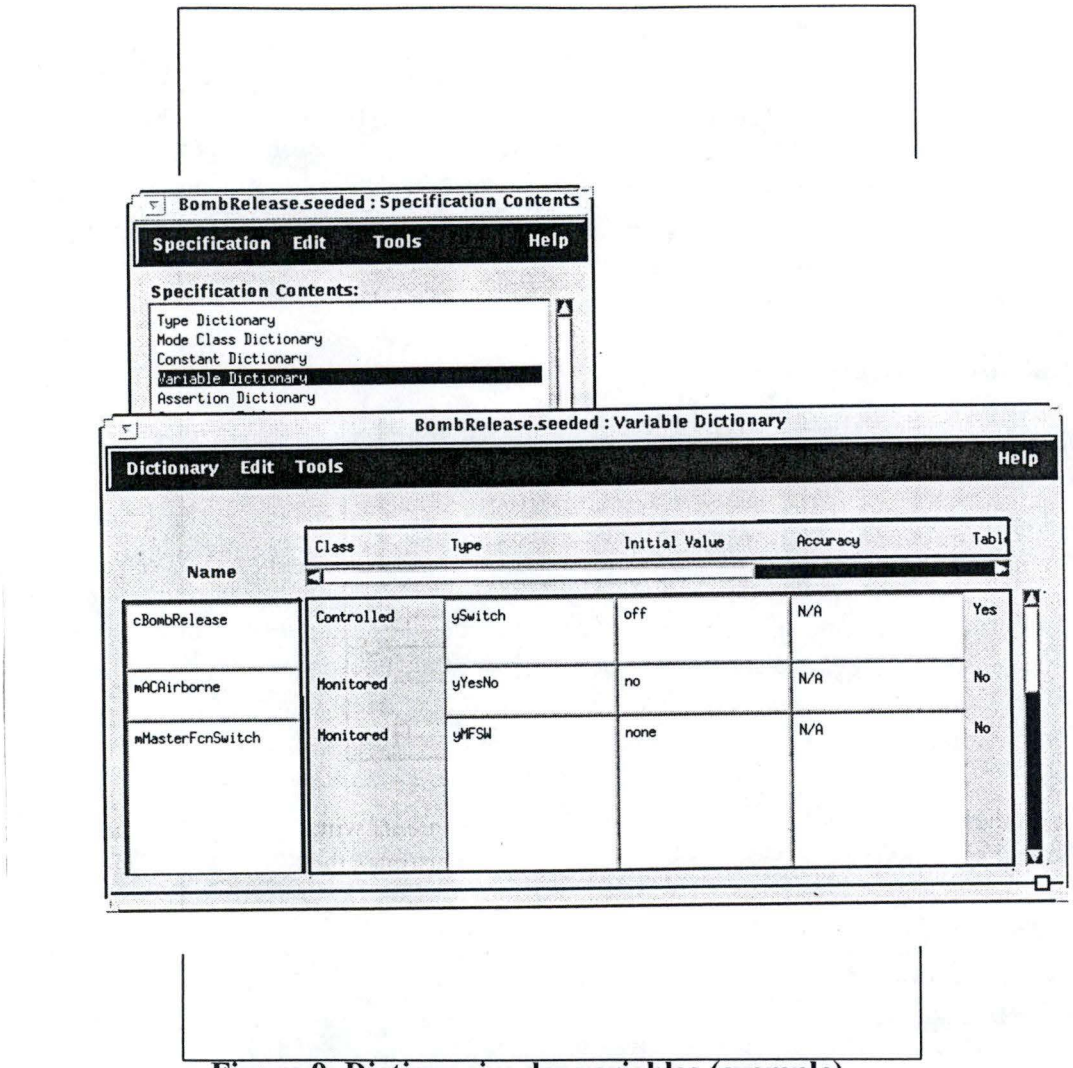


Figure 9. Dictionnaire des variables (exemple).

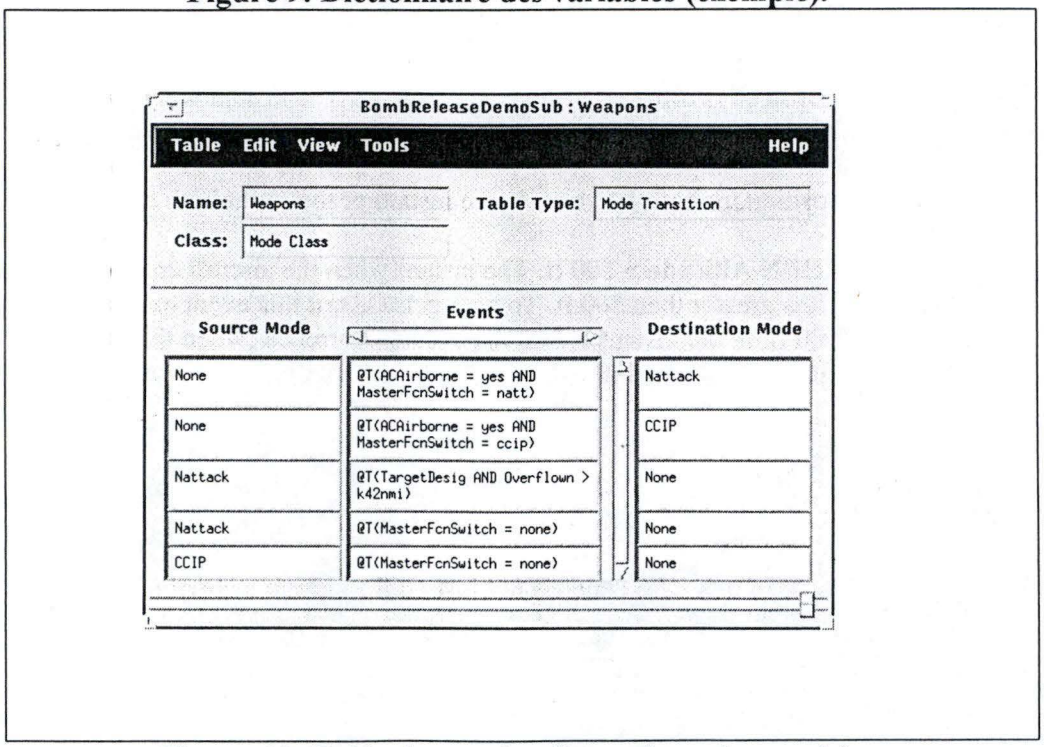
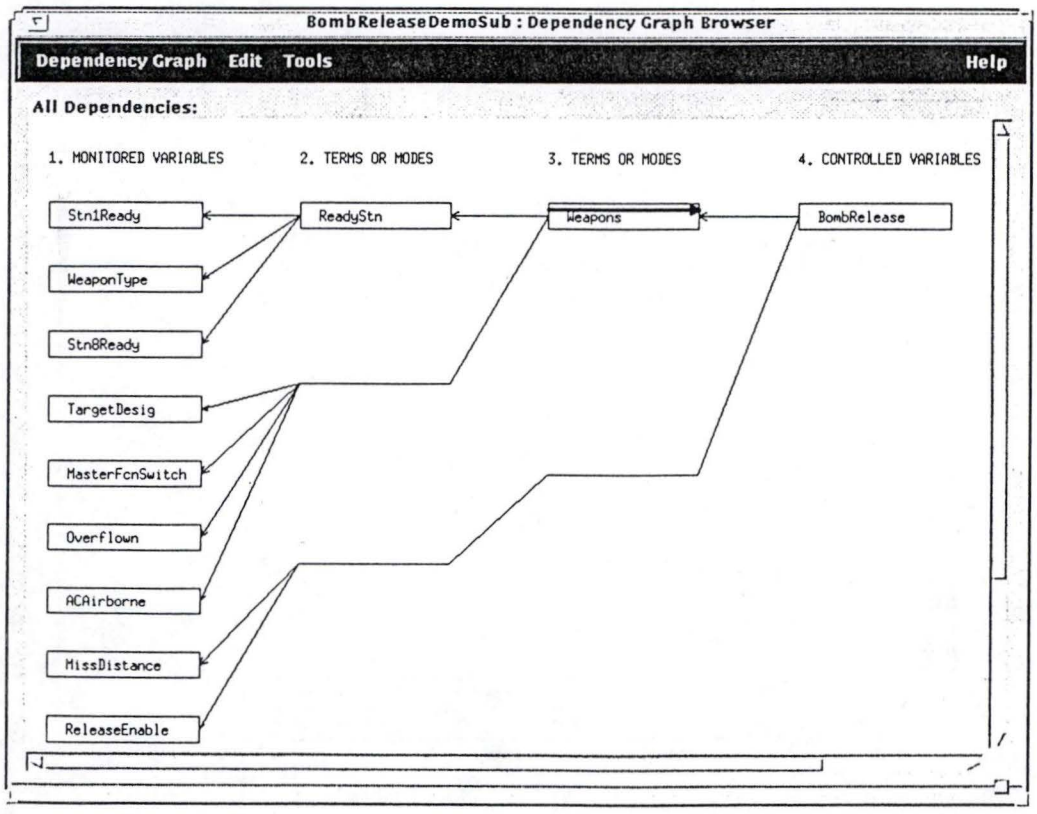


Figure 10. Table des modes d'une classe (exemple).

Examinons quelques instants le dictionnaire des assertions. Ce dictionnaire est un ajout récent à l'outil. Il permet de créer un ensemble de propriétés que le spécificateur peut tester via le simultateur, ou le vérificateur. On peut distinguer *deux sortes* de propriétés : celles qui concernent tous les *états atteignables du système*, et celles qui ne concerne qu'une paire d'*états adjacents*. Ces propriétés sont exprimées en logique temporelle classique. Sinon, nous nous trouvons en face d'un éditeur classique, chaque champ étant directement éditable d'un simple clique.

Le visualisateur des dépendances

Un des reproches majeur qui est généralement fait aux spécifications **SCR** est qu'elles donnent des informations détaillées sur le comportement du système, mais sans permettre de visualiser précisément comment des parties différentes des spécifications sont reliées entre elles, spécialement pour de grandes spécifications [HEITMEYER97b]. Afin de résoudre ce problème, un *visualisateur des dépendances* directes entre les variables d'une spécification donnée a été créé. Ce graphe (Figure 11) est construit de manière automatique par l'outil. Les variables d'état sont situées à l'extrême gauche du graphe, les variables d'action à l'extrême droite, les termes et les modes entre les deux. Ce graphe permet de déceler d'importantes informations sur les dépendances des variables, mais aussi sur les erreurs contenues dans les spécifications. Ainsi, l'existence de dépendances circulaires peut être révélée par l'apparition de flèches allant de gauche à droite entre deux nœuds différents. L'incomplétude des spécifications sera révélée par l'apparition d'orphelins, c'est-à-dire de variables non connectées à au moins une variable d'action (orphelin à droite), ou de variables qui ne sont pas des variables d'état et qui ont un ensemble de dépendance vide (orphelin à gauche).



Pour faciliter la visualisation dans les grosses spécifications, l'utilisateur peut afficher un sous-ensemble des variables. Il lui suffit simplement de sélectionner graphiquement le groupe de variable dont il désire voir apparaître le graphe de dépendance, et de refaire le même geste s'il désire raffiner le sous-graphe obtenu.

3. L'analyseur formel

Le vérificateur de consistance

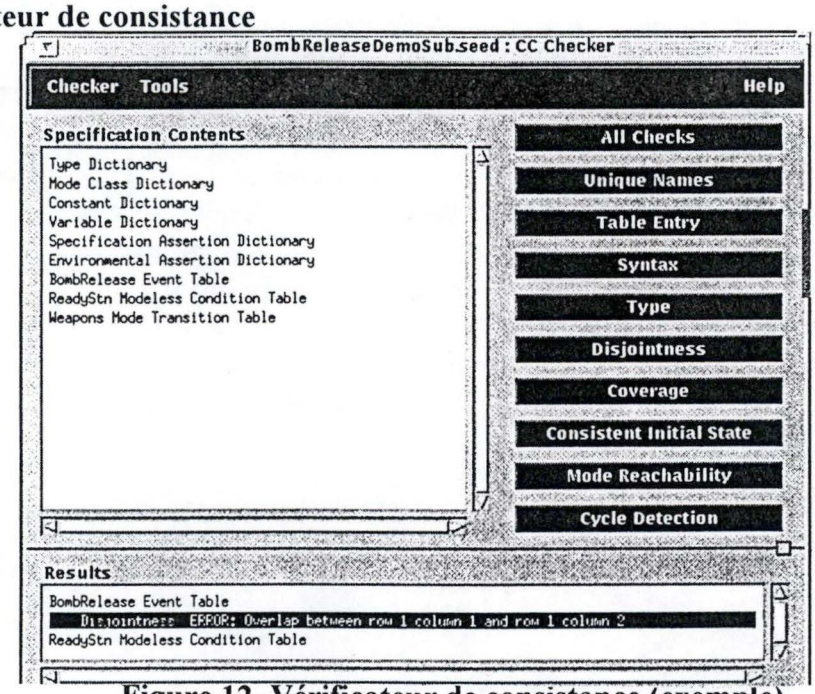


Figure 12. Vérificateur de consistance (exemple).

Nous avons déjà rencontré l'ensemble des vérifications (indépendantes du système développé) que l'on peu réaliser sur le modèle formel (Chapitre 2.6) . Ce sont les mêmes que l'on va pouvoir effectuer via l'outil. Il est possible de faire toutes les vérifications d'un seul jet, ou bien de réaliser une seule sorte de vérification à la fois (Figure 12). Dans ce cas, il faut noter que certaines vérifications doivent être réalisées avant d'autres. Pour cela, considérons le schéma suivant qui représente les dépendances entre les types de vérifications [KIRBY97] :

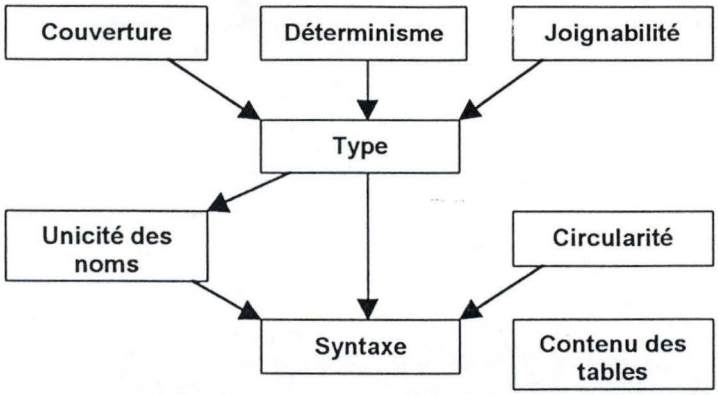


Figure 13. Dépendances des types de vérifications statiques.

Si l'utilisateur invoque une vérification qui dépend d'une autre vérification, l'outil lance automatiquement les vérifications manquantes. Lorsque des erreurs ont été trouvées, un double-clique sur l'erreur permet d'examiner la table où l'erreur trouve sa source. Il est aussi possible d'obtenir un historique des errreurs via l'édition d'un fichier continuellement utilisé par le vérificateur.

Le vérificateur de propriétés

Jusqu'il y a peu, le vérificateur de propriétés n'était pas encore implémenté dans l'outil. Récemment, un outil appelé Spin a été intégré. Celui-ci utilise l'exploration des états d'une machine à états finis à des fins de vérification de propriétés données. Comme le modèle formel **SCR** est basé sur une telle machine, la base formelle des spécifications est suffisante pour pouvoir utiliser cet outil. La vérification du modèle consiste à entrer les propriétés que l'on désire vérifier dans les dictionnaires d'assertions (par exemple, une propriété qui vérifierait que tout appel du MC est servi à un moment donné), et ensuite à invoquer Spin à partir de l'outil **SCR*** pour réaliser la vérification. A ce stade, les spécifications sont traduites en Promela, le langage de Spin, puis Spin fait évoluer les états du système en fonctions de tous les états possibles de ce système, vérifiant à chaque fois si ceux-ci vérifient ou non la propriété évoquée. Comme l'espace des états possibles[4] associés à la plupart des spécifications de type **SCR** rend la vérification inadéquate, l'utilisateur doit fournir à Spin un modèle abstrait des spécifications **SCR**. [HEITMEYER97a] montre comment l'outil traduit les spécifications **SCR** en Promela, ainsi que des techniques pour générer un modèle abstrait à partir de spécifications **SCR**. D'autres méthodes (PVS ou EVES) de vérification automatique de propriété sont à l'étude et sont principalement basées sur la preuve de théorèmes [HEITMEYER95c].

4. Le simulateur

Bases

Au démarrage du simulateur, le système se trouve dans un état de départ, qui est soit l'état initial du système, soit un état défini par l'utilisateur. Ensuite, l'utilisateur peut entrer des séquences d'événements. A chaque pression sur le bouton *Step* (Figure 14), le simulateur réalise l'exécution du prochain événement non exécuté, affiche si c'est nécessaire le nouvel état du système, et passe au prochain événement de la séquence. On peut visualiser communément le nouvel état du système soit dans la fenêtre d'affichage du simulateur, soit dans une fenêtre reprenant l'historique des variables modifiées par la séquence d'événement. Il est aussi possible de créer des scénario d'événements que l'on charge quand on le désire. Ceux-ci peuvent être exécutés entièrement en une seule passe par l'appui du bouton *Run*. Le bouton *Backup* permet de revenir à l'état précédant l'état courant par rapport à l'historique des événements, et le bouton *Restart* permet de remettre le système dans son état de départ et de revenir au premier événement en attente dans la liste des événements [KIRBY97].

---

[4] Imaginons une spécification avec un nombre restreint de variables, soit dizaine de variables entières ayant chacune une centaine de valeurs possibles (mettons de 0 à 99), ce qui donne environ 10^{20} états possibles pour le système, ce qui est déjà pas mal. Mais en plus, lors du passage vers le modèle Spin, le nombre de valeurs possibles est doublé, l'évolution du modèle SCR vers Spin obligeant à créer une variable supplémentaire pour chaque variable des spécifications afin de pouvoir représenter la valeur précédente de chacune de ces dernières.

Les événements sont aisément éditables par simple sélection de la variable sur laquelle va agir le prochain événement, puis par assignation de la nouvelle valeur de la variable via le champ d'édition qui apparaît à ce moment. Cependant, les variables prises de cette manières ne sont pas très parlante. Dans ce cadre, plusieurs expériences sont en cours pour associer des représentations graphiques aux différentes variables. Ces représentations rendraient la manipulation des spécifications beaucoup plus intuitive. Nous aborderons cet aspect du projet de manière plus complète dans la deuxième partie du présent travail.

L'intérêt majeur du simulateur est de pouvoir vérifier si le système spécifié se comporte exactement comme on le désirerait. Mais un autre attrait pour le simulateur provient de la possibilité qu'il offre de pouvoir vérifier des propriétés du système décrit.

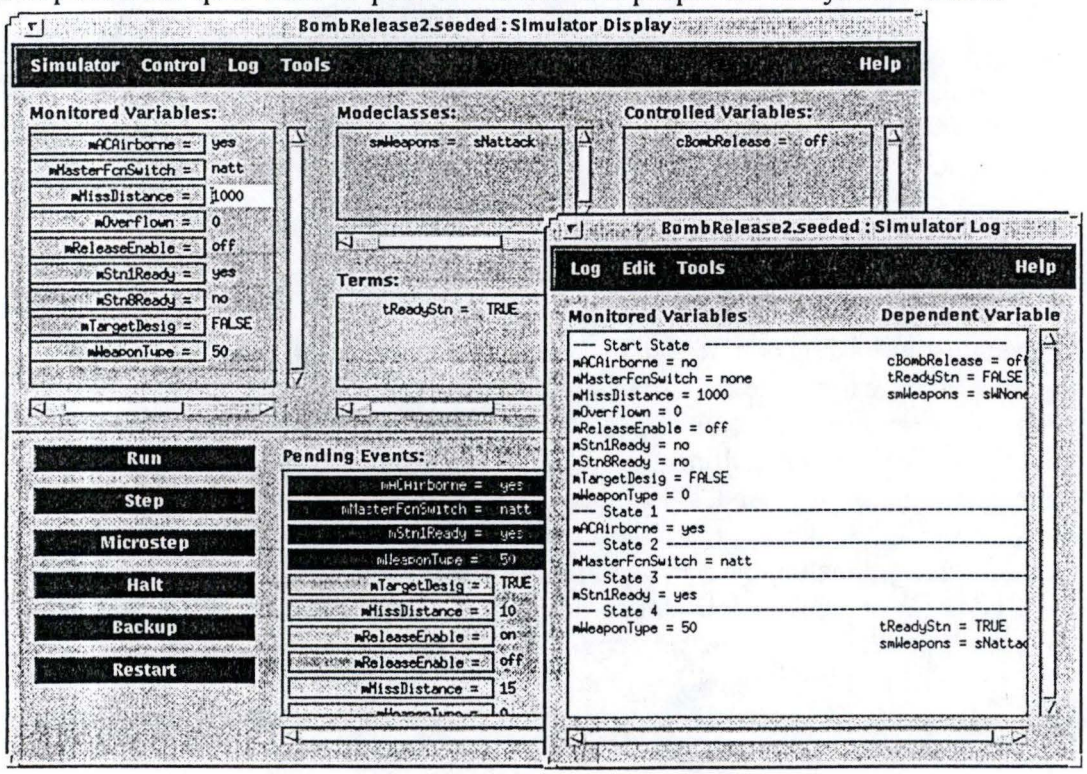


Figure 14. Le simulateur SCR (exemple).

Vérification de propriétés

Comme il a été vu précédemment, il est possible de définir des propriétés à vérifier dans les tables d'assertions. L'utilisation de scénario permet dès lors de vérifier si le système viole ces propriétés. Dès qu'une des propriétés se trouve être vérifiée ou transgressée au cours de l'exécution d'un scénario, l'historique signale le fait et renvoie éventuellement vers la variable qui en est l'origine. Cette vérification des assertions est utile pour les corrections comportementales du système. La vérification des assertions durant la simulation diffère fortement de la vérification de modèle exposée plus haut. La vérification par évaluation de tous les états du système est extrêmement gourmande en terme de ressource et de temps de calcul. On peut dès lors hésiter à développer un vérificateur de modèle, celui-ci pouvant faire double emploi avec la vérification par simulation. Mais les deux méthodes sont complémentaires, plutôt que concurrentes. [HEITMEYER97b] argumente en ce sens :

"Hence, it makes sense to check assertions via simulation early in the development of the requirements specification to weed out errors; model checking is more cost-effective when the requirements specification is more mature. Assertion cheking also require much less effort. Before model cheking can proceed, an abstract model with fewer states must be extracted from the specification. Generating such a model can be nontrivial."

5. Evaluation

Expériences probantes

Bien qu'étant encore à l'état de prototype, l'outil a déjà obtenu quelques résultats significatifs. De récentes collaborations avec le monde industriel ont démontré toute la valeur de l'outil dans sa branche d'activité. La vérification de propriétés (dépendantes ou indépendantes du système) fut particulièrement utile et appréciée par les ingénieurs de NORTEL ou de Rockwell Collins Avionics & Communications [HEITMEYER97a]. C'est lors de ces collaborations que l'intérêt pour le visualisateur de dépendances s'est révélé important.

L'A-7 revisité

De manière plus précise, l'outil a bien évidemment été utilisé afin de vérifier la première révision des spécifications de l'A-7. Déjà lors du passage de la version originale des spécifications vers sa version en notation **SCR**, il avait déjà été détecté plusieurs erreurs [PARNAS78]. Mais la vérification automatique des spécifications a été révélatrice de la puissance de cette méthode. Ainsi, sur les 36 tables de conditions des spécifications, l'outil détecta 19 erreurs et sur les tables de transitions de près de 700 lignes, 57 erreurs furent révélées. L'ensemble du processus de vérification prit un temps total de 245 secondes [HEITMEYER96b]. Avant la vérification automatique, la version révisée de l'A-7 avait été analysée manuellement par deux équipes, une composée de chercheurs, l'autre d'ingénieurs. Autant dire que la vérification manuelle montre ici ses limitations sans rien enlever évidemment aux qualités des vérificateurs manuels. Mais force est de constater que pour ce genre de système, la vérification automatique offre de nombreuses perspectives. Autre avantage de la vérification automatique : son faible coût. L'effort de vérification du système d'arrêt de la centrale nucléaire de Darlington a coûté près de 40 millions de dollars. Et pour des vérifications de propriétés jugées simple par Parnas ("*...reviewers spent too much of their time and energy checking for simple, application-independent properties* [HEITMEYER96b]."). Il ne fait pas de doute que l'utilisation d'outils pour ce genre de vérification coûte moins chers que d'utiliser des gens, pour une augmentation sensible de qualité.

Avantages

En plus de libérer le temps des gens afin qu'ils puissent réaliser des tâches plus créatives pour un résultat plus significatif, ce genre *d'outil augmente aussi considérablement la confiance* dans la correction des spécifications. Le simulateur joue un rôle important à ce niveau puisqu'il permet de vérifier le comportement du système. Enfin, la vérification de propriétés critiques renforce cette confiance, aidant ainsi considérablement les rapports entre

clients et analystes, et jouant un rôle primordial dans la validation de l'analyse par les deux parties.

6. Critique et Conclusions

Propriétés

Il est temps maintenant de comparer l'approche **SCR** avec l'ensemble des caractéristiques des cahiers des charges. Vérifions comment ils remplissent les différentes propriétés énoncées au premier chapitre :

- *Complétude et minimalité* : Dans l'ensemble, ces propriétés sont relativement bien respectées pour ce qui concerne les contraintes fonctionnelles du système. Pour le moment, les contraintes non-fonctionnelles sont encore exprimées en langage naturel et peuvent donc souffrir des lacunes propres aux langages naturels.
- *Consistance et Non-Ambiguïté* : Etant donné le niveau de formalisme de la notation **SCR** et ses capacités de vérification, les risques d'avoir des contradictions et des conflits d'interprétation sont finalement assez faible.
- *Modifiabilité et Evolutivité* : l'approche "boite noire" de la structure des spécifications **SCR** permet de modifier ou de faire évoluer les composants de manière assez aisée. Mais au sein même d'une "boite", les modifications peuvent être moins aisées. Imaginons que nous voulions maintenant introduire des modifications dans le comportement de notre monte-charge. Il faudrait procéder à quelques modifications dans certaines tables, et vérifier leurs implications sur le comportement global du monte-charge. Sur de petites spécifications cela ne pose pas vraiment de problème, mais sur de grosses spécifications, cela risque d'être moins facile. A ce titre, le simulateur pourra fournir une aide sans doute utile.
- *Traçabilité* : Les outils **SCR** permettent de gérer de manière assez efficace les documents. Peut-être que l'aspect "historique" des spécifications devrait être envisagé, mais pour un produit encore en phase d'étude, cela n'est pas ecore le point où se focalise l'intérêt.
- *Indépendance vis-à-vis de la solution* : C'est sur ce point que l'approche **SCR** pèche sans-doute le plus. En effet, les analyses exprimées en **SCR** ont tendance à exprimer un type de solution plutôt que d'être purement descriptives. Dans le cas du monte-charge par exemple, le fait d'avoir introduit deux moteurs au niveau des effecteurs induit une solution à deux moteurs, alors qu'il pourrait y avoir un moteur par variable d'action. De plus l'approche très formelle de la notation impose parfois à l'analyste d'avoir un processus de réflexion identique à celui que l'on aurait lors de la recherche d'une solution logicielle. Ainsi, c'est ce qui s'est passé lorsqu'il a fallu introduire certains des termes de l'exemple. L'effort fournit à ce moment fait donc un peu double emploi avec celui d'élaboration de la solution système.

Activités de l'ingénierie des besoins

En ce qui concerne les activités du domaine, on rejoindra les conclusions de [DUBOIS98b], à savoir qu'il est difficile de développer une notation qui puisse couvrir l'ensemble des activités d'ingénierie des besoins. A ce titre donc, la notation **SCR** convient beaucoup plus aux activités de *modélisation* et surtout de *vérification*. Même si la modélisation peut couvrir complètement le projet, la question se pose de savoir si cela est

vraiment souhaitable étant donné le manque d'expressivité de la notation, ou ne vaudrait-il pas mieux, toujours comme [DUBOIS98b] le montre, restreindre la modélisation en langage de type **SCR** aux composantes du projet qui requiert un haut niveau de vérification ? Par contre les compétences en matière de vérifications statiques ou dynamiques de propriétés d'**SCR** et de son pendant logicielle **SCR*** en font en ensemble idéal pour la vérification des composantes d'un projet. La *validation* et l'*élicitation* sont des activités qui risquent de mal supporter la notation **SCR** et son manque d'expressivité, à moins d'avoir un client pour qui ce type de notation est habituel.

SCR vs génie logiciel

Le rapport qu'entretient, ou que pourrait entretenir la notation **SCR** avec le génie logiciel est pour le moins assez évident. Le formalisme **SCR** permet de manière pas trop compliquée de réaliser de la génération de code. Ce code pouvant dès lors servir de base aux futures applications logicielles d'un projet donné. Afin de valider ce point de vue, il suffit de voir comment les spécifications **SCR** ont pu être transformées de manière efficace en Promela, le langage soutenu par le vérificateur de modèle Spin [HEITMEYER97a]. De là à générer un code applicable dans l'une ou l'autre des applications d'un projet, c'est un pas que les gens du NRL ne manquerons sans doute pas de franchir dans l'avenir.

Conclusion

La notation **SCR** est une notation basée sur un modèle mathématique stable et rigoureux qui permet un formalisme très opérationnel et offrant des avantages certains pour l'activité de vérification, le tout sur base d'outils performants. Bien que le niveau d'expressivité de la notation ne permette pas de généraliser son application au domaine d'activité, la notation et surtout l'outil le supportant peut offrir un plus majeur dans l'élaboration de projets où la sécurité joue un rôle majeur. L'expérience industrielle dont la méthode a pu bénéficier montre l'importance d'**SCR** en terme de valeur ajoutée par rapport au processus dans lequel la méthode s'insère.

Chapitre 4
Comparaison ***SCR*** *– Albert II*

[Texte à ajouter]

1. Introduction

Réaliser une étude sur un langage d'analyse des besoins *made in USA* ne pouvait réellement s'envisager sans le comparer à l'approche des chercheurs de l'Institut d'Informatique dans ce domaine. Depuis 1992 l'Institut d'Informatique, dans le cadre du projet Esprit II *Icarus*, participe à la création d'un langage de spécification nommé *Albert II*. Le langage a depuis quelque peu évolué et ses possibilités se sont étoffées.

Ce qui est intéressant dans la comparaison qui va suivre, c'est tout d'abord le fait que 15 années séparent la genèse des deux méthodes. Or comme chacun sait, 15 années en informatique équivalent à plusieurs révolutions techniques et logicielles qui ne manqueront pas d'influencer le rapport de l'un par rapport à l'autre. De plus, le milieu de développement du projet **SCR** est relativement différent de celui d'*Albert II*. En effet, comment comparer l'effort de recherche de l'Institut d'Informatique à la puissante machine de recherche américaine qu'est le Department of Defense ? Pourtant la comparaison en vaut la peine, chacun y allant de ses petits succès, et chaque méthode trouvant sa place dans le long chemin conduisant à l'élaboration de nouveaux systèmes.

Dans ce chapitre, nous verrons tout d'abord une brève présentation du langage *Albert II*, de son modèle ainsi que de sa notation, puis nous l'appliquerons à l'exercice maintenant connu du Monte-Charge, exercice que nous critiquerons, et enfin nous ferons une comparaison des approches **SCR**-*Albert II*. Cette comparaison sera suivie de la conclusion de ce chapitre qui sera aussi celle de la première partie de ce travail.

2. Le langage *Albert II*

Généralité

Parmi les spécificités du langage *Albert II*, on retrouve un haut niveau d'*expressivité* et de *formalité*. Le langage est basé sur une ontologie de concepts qui fut prouvée utile pour la capture de contraintes fonctionnelles propres aux systèmes temps-réels, distribuées et composites. Basé sur une variante orientée-objet d'une logique temporelle temp-réel appelée *Albert*-Kernel, le langage se caractérise aussi par une *syntaxe naturelle* qui permet par exemple d'exprimer dans un langage formel les desiderata généralement peu formels des clients. Le langage *Albert II* est agencé de manière à saisir les *contraintes fonctionnelles*. En ce qui concerne les contraintes non-fonctionnelles, une intégration de l'environnement **Kaos** est proposée [DU_BOIS97][DUBOIS98b].

Le modèle conceptuel *Albert II*

Le contenu des spécifications *Albert II* décrit généralement le *comportement* de systèmes *composites*. Les spécifications sont structurées en *agents*, les agents étant des entités du monde réel qui ont une certaine autonomie comportementale (une personne, un sous-système,...)[HEYMANS98a]. La sémantique d'une spécification *Albert II* est donnée par sa correspondance en *Albert*-Kernel. L'ensemble des axiomes dérivés de la translation de spécifications *Albert II* en *Albert*-Kernel définit un ensemble de modèles de la spécification.

Chacun de ces modèles est fait de la somme des vies des instances des agents qu'ils contiennent [HEYMANS98b].

La vie d'un agent correspond à une séquence alternée d'*états* et de *modifications* (Figure 15). Chaque modification est marquée par une valeur temps-réelle toujours grandissante, mais pas nécessairement équidistante de la précédente. Le marquage temporel représente l'instant où quelque chose arrive au système. L'état d'une instance d'agent représente la valeur de tous ses *composants d'état* dans l'intervalle de temps durant lequel ils sont inchangés. Une modification est composée d'*événements* simultanés, un événement correspondant à :

- soit la survenance d'une action immédiate ("<action>" dans la Figure 15);
- soit le début de la survenance d'une action ("<action" dans la Figure 15);
- soit la fin de la survenance d'une action ("action>" dans la Figure 15).

Seul la survenance d'un événement peut induire un changement d'état, la valeur d'un état à un moment donné pouvant donc toujours être dérivée de manière déterministe à partir de l'état initial et de l'historique des événements.

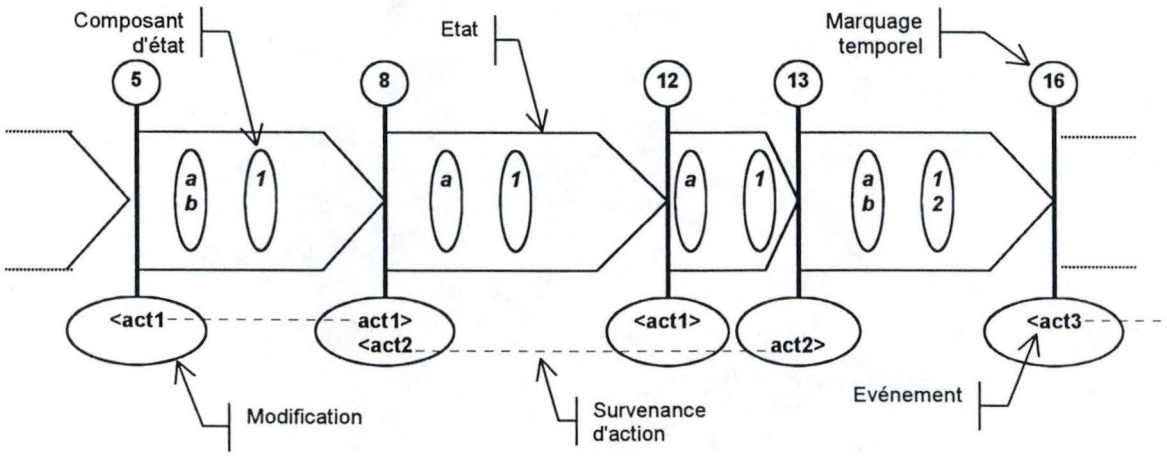


Figure 15. Comportement d'une instance d'agent (détails).

Comme l'objectif des spécifications *Albert II* est de définir le comportement d'un système, sa structure implique que cela revient à décrire le comportement des agents composants le système. Le rôle de l'analyste est double :

- Définir le vocabulaire du système (Décrire les agents, la structure de leurs états, leurs actions, ...) dans les *déclarations* du document. Celles-ci peuvent être textuelles ou graphiques.
- Ensuite, restreindre l'ensemble des comportements possibles par l'élaboration de *contraintes*. Pour ce faire, il utilise des *gabarits textuels* [DU_BOIS97].

La syntaxe *Albert II*

Les spécifications *Albert II* comprennent deux parties :

- la partie 'statique', faite de définitions de *types* et d'*opérations*;
- et la partie 'dynamique', liée à la spécification des agents.

La partie est écrite au début des spécifications, car elle définit les types et les opérations qui seront nécessaires à plusieurs places dans les spécifications. L'exemple suivant définit le type *TEtatPorte* qui peut prendre quatre valeurs énumérées :

TEtatPorte = ENUM[*Ouvert*, *se_Ferme*, *Fermee*, *s_Ouvre*, *Bloquee*].

L'analyste peut trouver un ensemble de types prédéfinis pour l'aider ainsi que les opérations associées. Il pourra trouver aussi une série de constructeurs de type ainsi que des opérateurs de types paramètres. Tout cela ainsi que les définitions qui vont suivre peut être examiné dans [DU_BOIS97]ou dans [HEYMANS97b].

La partie 'dynamique' représente l'essentiel des spécifications. La décomposition des spécifications en plusieurs agents permet de réduire les dépendances entre les différentes parties, les agents étant regroupés en *sociétés*. Ainsi, dans l'exemple du MC exposé ci-après, on définit la société *Système du Monte Charge* comme ceci :

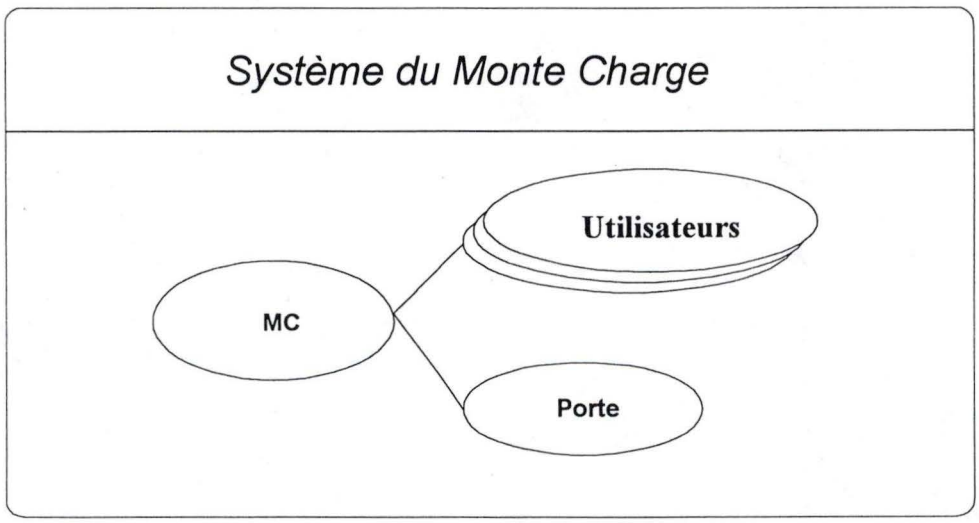


Figure 16. Structure des agents du Système du MC.

Société : **Système du Monte Charge**

(*MC*)
(*Portes*)
(*Utilisateurs*)

Dans ce cas ci, la société est composée de trois agents : le monte-charge, les portes du monte-charge, et les utilisateurs de ce dernier. La définition de la société, tant textuelle que graphique est très simple, la représentation graphique étant un rien plus explicite puisqu'elle permet d'extraire la structure des agents.

Les agents sont généralement responsables des données qu'ils connaissent. Dans le cas du MC par exemple, les utilisateurs connaissent l'étage où ils se trouvent et où ils désirent aller. Donc l'agent *Utilisateurs* possède un *composant d'état* appelé *Destination* et *Origine*. La définition se fait de cette manière :

STATE COMPONENTS

Destination instance-of *Integer*
Origine instance-of *Integer*

Dans ces deux cas le type de ces composants est très simple. Mais ils auraient pu être typés de manière plus complexe. De manière graphique, les composants d'états sont représentés par des rectangles reprenant le nom du composant et le type de valeur qu'il accepte; les agents par un parallélogramme (Figure 17). Mais la représentation graphique comprend un peu plus que cela. En effet, les actions possibles des agents apparaissent au sein de l'agent sous forme de rectangles aux bords arrondis dans lesquels figure le nom de l'action. Les flèches à l'intérieur d'une action ou d'un composant d'état signifie que cette action ou ce composant est "vu" par l'agent signalé après la flèche. Les actions ou composants d'état se trouvant hors de l'agent sont ceux qui sont "vu" par cet agent, l'agent figurant à proximité de la flèche représentant dans ce cas l'agent qui "montre" cette action ou ce composant.

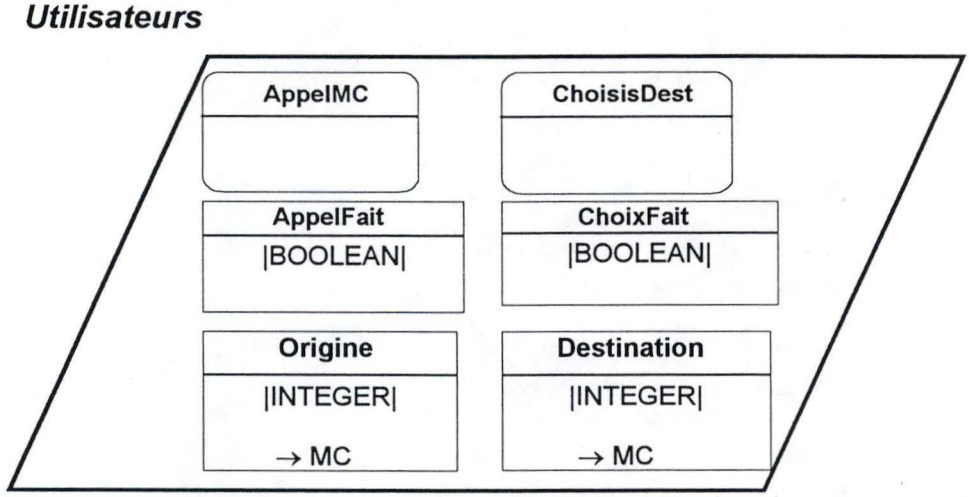


Figure 17. L'agent *Utilisateurs*.

La partie restante de la spécification consiste en une série de *contraintes* restreignant l'ensemble des comportements possibles des agents. Il y a en tout quatre types de contraintes, toutes associées à des gabarits agissants comme autant de guides méthodologiques. Les types de contraintes sont :

- les contraintes de *base*;
- les contraintes *déclaratives*;
- les contraintes *opérationnelles*;
- et les contraintes de *coopération*.

Les contraintes de *base* sont destinées à décrire l'*état initial* d'un agent et à donner des règles de dérivation pour les composants d'état dérivés. En voici un petit exemple :

BASIC CONSTRAINTS

DERIVED COMPONENTS

```
Le temps d'attente de l'utilisateur est le temps d'arrivée du MC
additionné au temps d'ouverture des portes du MC.
```

TmpAttente = *TmpOuverturePorte* + *TmpArrivéeMC*

INITIAL VALUE

```
Au début l'utilisateur n'a pas encore fait son appel.
```

AppelFait = FALSE

Les contraintes *déclaratives* permettent d'exprimer de manière déclarative des contraintes sur l'ensemble du comportement d'un agent. Le *comportement d'état* exprime les conditions qui doivent être satisfaites à tout moment de la vie de l'agent. Les *compositions d'actions* restreignent la survenance des actions à certaines conditions de séquencement, parallélisme, répétition,...La *durée d'action* limite la durée des actions à une limite maximum, minimum ou précise. Exemples :

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

```
| Quand la porte est ouverte, elle l'est pendant 5 secondes, après elle se
| ferme.
```

$[]Statut = Ouverte \Rightarrow (\text{Futr}_{5''}\ Statut = se\_Ferme)$

ACTION COMPOSITION

```
| Quand le MC ouvre la porte, l'action ouvre débute.
```

$OuvrePorte \leftrightarrow Ouvre \mid\Rightarrow MC.OuvrePorte$

ACTION DURATION

```
| Ouvrir la porte doit prendre moins de 5 secondes.
```

$Ouvre < 5\ sec$

Les contraintes *opérationnelles* sont au nombre de trois: les *préconditions,* les *effets d'action,* et les *enclenchements.* Les préconditions sont les conditions que doivent remplir les actions avant de débuter, les enclenchements sont les conditions sous lesquelles les actions sont effectivement démarrées. Les effets d'action peuvent avoir des effets pré et post. Syntaxiquement, les effets sont séparés par une paire de crochets représentant l'action et déterminant ainsi si la condition est pré ou post :

OPERATIONAL CONSTRAINTS

PRECONDITIONS

```
| On ne peut fermer les portes que quand elles sont ouvertes depuis 5
| secondes.
```

$Ferme :\ \text{Lasted}_{5''}\ (Statut = Ouvert)$

EFFECTS OF ACTIONS

```
| Ouvrir la porte fait passer le Statut de la porte à s_ouvre, puis quand
| l'action se termine, il passe à Ouvert.
```

Ouvre : *Statut* := *s_Ouvre*
[]
Statut := *Ouvert*

TRIGGERINGS

| L'action Ferme est enclenchée si la porte est ouverte depuis au moins 5 secondes.

Lasted$_{5''}$ *(Statut = Ouvert)*/TRUE ~> *Ferme*

Enfin, les contraintes de coopération expliquent comment les agents interagissent avec leur environnement, c'est-à-dire quelles parties des composants et des actions d'un agent donné, les autres agents peuvent "voir". Identiquement, on définit aussi les actions et les composants qu'un agent donné peut "voir" chez les autres agents. Tous cela est repris sous les termes d'*information d'action* et *d'état*, et de *perception d'action* et *d'état* :

COOPERATION CONSTRAINTS

ACTION PERCEPTION

| L'agent Porte perçoit l'action du MC OuvrePorte lorsque le Statut de la porte est *Fermée*.

XX(MC.OuvrePorte / Statut = Fermee)

STATE INFORMATION

| A tout moment, le MC connaît le Statut de la porte.

XX (Statut.MC / *TRUE*)

Voilà en ce qui concerne l'essentiel de la syntaxe des spécifications *Albert II*, d'autres aspects plus pointilleux pouvant être lus dans [DU_BOIS97]. L'ensemble de l'exemple du MC appliqué à *Albert II* peut être consulté à l'annexe 4.

3. Analyse de l'exemple en *Albert II*

Tout comme pour la notation **SCR**, nous allons analyser quelque peu ce que donne notre exemple sous *Albert II*.

- "*Quand le MC arrive à un étage où il doit s'arrêter, ses portes s'ouvrent pour un période de 5 secondes. Après quoi, celles-ci se referment.*"

Reprenons-en les deux sous-contraintes : pour que les portes s'ouvrent, le MC doit s'arrêter; et les portes doivent être ouvertes durant 5 secondes.

Considérons la séquence de contrainte suivante :

Porte.Statut = Fermee $\wedge$ *Destination = PosAct* $\wedge$ *Statut* $\neq$ *Arrete*/TRUE~>*Stoppe* *(3.21)*
Stoppe : *Statut := Arrete*
[] *(3.13)*
MC.Statut := Arrete $\wedge$ *(Statut = Ferme* $\vee$ *Statut = Bloquee)*/TRUE ~> *Ouvre* *(2.9)*
Ouvre : *Statut := s_Ouvre*
[] *(2.7)*
Statut := Ouvert
Lasted$_{5''}$ *(Statut = Ouvert)*/TRUE ~> *Ferme* *(2.10)*

La contrainte (3.21) donne les conditions permettant au MC de se stopper, c'est-à-dire portes fermées et à destination. Ce qui à pour effet (3.13) de faire passer le statut du MC à *Arrete,* validant par cela le déclenchement de (2.9). La fin de l'événement d'ouverture fait passer le statut de la porte à *Ouvert* (2.7), ce qui après 5 secondes de vie permet à l'action *Ferme* de l'agent Porte (2.10) de se déclencher.
Si on considère que les autres contraintes des spécifications valident celles exposées ici, on voit qu'un arrêt du MC doit nécessairement amener à une ouverture des portes de 5 secondes, puis à sa fermeture. On peut aussi constater le degré d'expressivité des contraintes exprimées, et avec quelle facilité la séquence d'événements peut se lire.

- "*Lorsque plusieurs destinations sont choisies, le MC ira vers l'étage le plus proche dans le sens correspondant au sens du service précédent [...] Sinon, il ira à l'étage le plus proche.* "

⇒ "*Lorsque plusieurs destinations sont choisies...*"

Ici, on fait référence à l'introduction d'une séquence de choix de destination. L'attribution des destinations pour le MC se faisant quel que soit l'état du système, il n'est pas nécessaire de comparer tous les choix entre-eux. Ceci s'exprime en *Albert II* par l'absence de contraintes de déclenchement sur les actions *ChoisiDest* et *AppelMC*. Il suffit simplement de comparer la destination actuelle du MC avec la destination proposée.

⇒ "*...le MC ira vers l'étage le plus proche dans le sens correspondant au sens du service précédent...*"

Utilisateurs.ChoisiDest: *Destination := Utilisateur.Destination*
[] *(3.15)*
...
[] *(3.6)*
Destination = Utilisateurs.Destination Since *((Destination = d' ∧ (PosAct ≤ Utilisateurs.Destination ≤ d' ∧ DerDir ≠ Haut) ∨ (d' ≥ Utilisateurs.Destination ≥ PosAct ∧ DerDir ≠ Bas))* Until!
(Destination = d'' ∧ (PosAct ≤ Utilisateurs.Destination ≤ d'' ∧ DerDir = Haut) ∨ (d'' ≥ Utilisateurs.Destination ≥ PosAct ∧ DerDir = Bas)))

Il y a tout d'abord l'assignation de la destination du MC (3.15). Mais l'essentiel de la contrainte s'exprime dans la contrainte (3.6). Celle-ci nécessite quelques explications. Tout d'abord, la sémantique des opérateurs temporels:

α Since β (α est vrai depuis que β était vrai)
α Until! β (α est vrai jusqu'à de que β devienne vrai et à ce moment α devient faux)

La forme générale de l'expression est donc :

α Since (β_1 Until! β_2) avec :

α = *Destination = Utilisateurs.Destination*
β_1 = *Destination = d' ∧ (PosAct ≤ Utilisateurs.Destination ≤ d' ∧ DerDir ≠ Haut) ∨*

$$\beta_2 = Destination = d'' \wedge (PosAct \leq Utilisateurs.Destination \leq d'' \wedge DerDir = Haut) \vee (d'' \geq Utilisateurs.Destination \geq PosAct \wedge DerDir = Bas)$$

β_1 exprime la situation où une proposition de destination est plus proche que la destination actuelle du MC, mais dans le sens contraire du sens prit précédemment par le MC.
β_2 exprime la situation où une proposition de destination est plus proche que la destination actuelle du MC, et allant dans le même sens que le sens prit précédemment par le MC.

La clause β_1 Until! β_2 exprime donc que le MC préférera toujours prendre la même direction que celle du sens précédent, sauf quand on ne propose que des directions allants dans le sens contraire du sens précédent.

La clause α Since β avec $\beta = \beta_1$ Until! β_2, valide donc le fait que Destination = Utilisateurs.Destination ne puisse se faire que dans le cas où β_1 serait vrai jusqu'à ce que l'on rencontre un cas β_2 vrai.

Première petite comparaison

Une première remarque peut être avancée. On constate ici toute la force des opérateurs temporels appliqués à *Albert II*. En effet, en une seule ligne, on parvient à exprimer une contrainte complexe, là où son équivalent **SCR** a exigé trois tables. Bien évidemment cela fut contraire à la lisibilité de la contrainte, d'où la nécessité de la décomposition qui précède. *Albert II* permet de réaliser ce genre de décomposition, et cela aurait permis de rendre la contrainte plus lisible. Mais l'objectif ici était de pouvoir comparer des contraintes *Albert II* avec leur équivalent **SCR** (cnfr Chapitre 2.5). En effet, la création de la table **SCR tDirLocked** fut obligatoire si on voulait exprimer correctement la contrainte car **SCR** imposait que ce terme soit évalué dans sa propre table d'événement avant d'être évalué dans la table d'événement définissant **tDestination**. Dans la notation *Albert II*, les expressions entretiennent des relations bien plus aisées avec les états passés ou futurs des composants, puisque dans une même expression, les états non-courants d'un composant côtoient sans difficulté ses états courants. Cet aspect renforce l'expressivité de la notation *Albert II*, et montre un peu combien parfois la notation **SCR** peut être peu naturelle à l'emploi.

La notation *Albert II* n'est cependant pas exsangue de constructions un peu artificielles. Si on considère la contrainte (2.4), on peut constater que pour réaliser la composition co-débutante de *Ouvre* avec *MC.OuvrePorte*, il faut nécessairement introduire une nouvelle action *OuvrePorte*, jumelle d'*Ouvre*, mais réceptacle de la composition des actions. Bien que cela soit imposé, cette contrainte alourdit la représentation de l'agent qui se voit inclure une nouvelle action ressemblant très fortement à une autre.

Enfin, il est intéressant d'aborder le problème du *déterminisme* en *Albert II*. En effet, ce langage n'est pas déterministe. Par exemple, il n'est pas nécessaire de spécifier une valeur initiale pour tous les composants. Ou encore, si la survenance d'une action n'est pas restreinte par une précondition, cette action peut survenir à n'importe quel moment. Bien que cela ne soit pas un vrai problème, les spécifications *Albert II* gagnants en souplesse de ce manque de

déterminisme, ce problème de déterminisme risque être un frein à l'animation des spécifications, comme nous le verrons ci-après.

4. Evolutions et perspectives d'*Albert II*

Expérience préalable

Le langage *Albert II* ne peut encore prétendre à une aussi grande expérience en milieu industriel qu'**SCR**. Mais une des expériences majeure fut le projet 2RARE (2Real Applications for Requirements Engineering), où deux grands systèmes distribués et hétérogènes et dont les spécifications avaient une grande importance, furent envisagés. Le premier projet était la réalisation d'un réseau de satellites et de stations terriennes dont la partie logicielle représente une importance toute particulière. Le second projet est une application multimédia offrant un service de vidéo à la demande au travers du réseau téléphonique, une extension de ce projet devant intégrer par la suite des services plus complexes (téléshopping,...). L'utilisation d'*Albert II* pour ce projet permit d'une part à ce dernier d'évoluer vers la version actuelle, mais permit aussi l'élaboration d'outils facilitant l'édition et la correction des spécifications [2RARE95].

L'animateur

Bien qu'il existe déjà des outils supportants l'édition de spécifications *Albert II*, l'une des évolutions majeure de cet ensemble d'outil est le projet de réalisation d'un *animateur*. L'animation va consister pour un utilisateur ou à un ensemble d'utilisateurs de construire de manière dynamique un comportement global du système défini basé sur le comportement individuel de ses composants, composants ayants une représentation graphique. Ce comportement sera progressivement construit par interaction avec l'animateur, qui vérifiera si le comportement des composants respecte les contraintes telles qu'elles ont été définies dans l'outil. Le problème du déterminisme sera dès lors résolu par demande dynamique à l'utilisateur de tâcher de résoudre le manque de déterminisme rencontré. L'ensemble des tests peut dès lors prendre la forme de *scénario*, représentant les situations dans lesquelles on désire vérifier que les spécifications se comportent comme les clients le désirent.

Comme on peut déjà le constater, l'idée générale d'animateur *Albert II* rejoint celle du simulateur **SCR** et de ses extensions. On y retrouve l'ensemble des caractéristiques du produit du NRL, à savoir : une base formelle, l'animation symbolique, la création dynamique de scénario et leur gestion, ainsi que la possibilité de réaliser des retours en arrière durant une exécution. Un point reste cependant pour le moment différent. En effet, l'animateur *Albert II* est un outil distribué, ce en quoi, il va permettre à des utilisateurs distants de pouvoir "jouer" avec l'outil sans que ces derniers ne se côtoient directement [HEYMANS97a]. Par contre, il ne semble pas que dans un avenir proche, l'animateur puisse réaliser de la vérification dynamique de propriétés par vérification de modèle comme le réalise **SCR***, pour trois raisons majeures :

- la transformation du langage *Albert II* vers une extension plus opérationnelle ne va pas rendre les spécifications *Albert II* reconnaissables par les gens qui ont écrit ces dernières;

- la perte d'expressivité due à la transformation de l'extension du langage vers l'automate d'exécution;
- l'apparition du problème déjà évoqué plus haut d'explosion des états de l'automate.

Mais l'approche *Albert II* en cette matière possède quand même deux avantages [HEYMANS97b] :

- avec cette technique, on peut toujours fournir une liste des obligations qu'un état donné du système doit réaliser dans le futur. De cette manière, les utilisateurs vont être capables d'estimer raisonnablement quelles sont les obligations qui seront satisfaites et celles qui ne le seront pas dans le futur;
- le fait de ne pas pouvoir prouver une propriété n'est pas aussi grave que de ne pas être capable de trouver des erreurs dans les spécifications, d'autant plus que le feedback donné sur les erreurs trouvées est exprimé dans les termes de ceux qui ont écrit les spécifications.

Gageons que cela joue en faveur de la notation.

5. Critique et conclusions sur *Albert II*

Propriétés

Comme nous l'avons fait précédemment, comparons l'approche ***Albert II*** avec l'ensemble des caractéristiques des cahiers des charges. Vérifions d'abord comment ils remplissent les différentes propriétés énoncées au premier chapitre :

- *Complétude et minimalité* : Dans l'ensemble, le formalisme de la notation *Albert II* permet de respecter correctement ces propriétés, surtout grâce à l'aide que procurent les gabarits de rédaction des contraintes. Cependant, la notation est résolument orientée vers la description des contraintes fonctionnelles, préférant laisser la description de contraintes non-fonctionnelles à des notations plus appropriées comme *Kaos* [DUBOIS98b].
- *Consistance et Non-Ambiguïté* : La syntaxe et le vérificateur syntaxique *Albert II* permettent de respecter ces propriétés avec efficacité.
- *Modifiabilité et Evolutivité* : L'approche orientée 'agents' de la notation *Albert II* est définitivement un gage de facilité d'évolution des spécifications. Tout comportement devant évoluer dans un système sera aisément modifiable par modification des comportements des agents participant à ce comportement. Par contre, l'analyse d'impact d'une évolution sera un peu moins facile à juger, mais fort heureusement l'animateur, ainsi qu'un repository Telos des agents, viennent à point pour supporter cet aspect [DUBOIS98b].
- *Traçabilité* : Les outils de gestion des spécifications *Albert II* sont efficaces dans ce domaine, et permettent d'avoir de manière assez rapide un plan d'ensemble des parties d'un projet, comme d'un point particulier. Sur ce point, la base de donnée Telos évoquée plus haut permet la traçabilité des méta-modèles *Albert II* avec ceux d'autres langages de description de cahier de charge.
- *Indépendance vis-à-vis de la solution* : Le point fort d'*Albert II*. L'exemple du monte-charge est significatif. La description du problème a volontairement été vague sur certains points du système de manière à examiner les apports des langages par rapport à l'énoncé. Comme on peut le constater, les spécifications *Albert II* qui en résultèrent sont restées

strictement descriptives du système, sans introduire d'idée de la solution comme cela fut le cas en **SCR**.

Activités de l'ingénierie des besoins

[DUBOIS98b] classe résolument *Albert II* dans l'activité de *modélisation*, ce qui est parfaitement cohérent avec les objectifs opérationnels du langage. Cependant, il n'est pas exclu que les activités plus orientées 'client' d'*élicitation* et de *validation* puissent partiellement supporter la notation *Albert II*, le niveau d'expressivité et de naturel du langage permettant relativement facilement au client d'en saisir la sémantique. La partie *vérification* peut être supportée par l'animateur pour une part, mais pour une vérification plus poussée de propriétés, les modifications qu'il faudrait apporter au langage le dénatureraient trop que pour envisager cette solution via le langage tel qu'il existe actuellement [HEYMANS98b].

Albert II vs génie logiciel

Les rapports d'*Albert II* avec le génie logiciel semblent surtout intéressant au niveau de l'analyse logicielle, surtout dans le cas de développement en langages orientes-objets. En effet, la notion d'agent est assez proche de celle d'objet, les agents *Albert II* pouvant dès lors servir de base à la création d'objets génériques desquels on pourra dériver une architecture plus complexe. Par contre, étant donné le niveau d'abstraction d'*Albert II*, on peut actuellement difficilement envisager de faire de la génération de code sans dénaturer à nouveau le langage.

Conclusion

La notation *Albert II* est une notation basée sur un modèle à haut niveau d'expressivité et de naturel. Son aisance en logique temporelle en fait un bon moyen d'exprimer des contraintes temps-réels complexes. Sa découpe orientée 'agents' et la manière dont ces agents 'communiquent' permettent quant à eux de représenter les systèmes distribués sans grandes difficultés. La présence de l'animateur et sa facilité d'adaptation aux référentiels des clients permettent enfin à *Albert II* d'avoir un rôle non-négligeable au sein des activités de l'ingénierie des besoins.

6. *Albert II* versus SCR

Relation Système-Environnement

Une des différences majeures entre *Albert II* et **SCR** est la manière dont l'un et l'autre envisagent le rapport entre le système à élaborer et l'environnement. Le modèle des Quatre Variables de **SCR** montre de manière assez évidente une vision centrée sur le système, où l'environnement n'est perçu que par la 'sensibilité' des variables d'état ou d'action. Le comportement de l'analyste est de se placer à l'intérieur du système et de se poser la question "que me faut-il pour que je puisse 'voir' ou 'agir' sur mon environnement ?". Bref, un petit peu comme si on regardait par le petit bout de la lorgnette. Par contre, en *Albert II*, on retourne la lorgnette, on prend une vision globale du monde dans lequel le futur système côtoie un environnement. Cette différence se manifeste notamment par l'apparition de l'agent

'Utilisateurs' dans les spécifications *Albert II*, alors qu'aucune référence à l'entité susceptible d'utiliser le système décrit en **SCR**, n'est exprimée.

Cade historique de recherche

Comme nous l'avons vu précédemment, près de 15 années séparent la genèse du projet **SCR** au projet *Albert II*. Or, il y a 15 ans, les grands principes de l'orienté-objet n'étaient pas encore vraiment en application, et cela se traduit inévitablement par des différences dans les orientations 'stratégiques' d'**SCR** par rapport à *Albert II*. L'effectivité d'**SCR** sera optimale dans les projets où les méthodes de développement sont essentiellement orientées SA/RT, comme elles le sont encore pour la majorité des projets du DoD. Par contre, l'approche *Albert II* s'adapte beaucoup mieux à la pensée 'orientée-objet' pour les raisons évoquées au point précédent, et bénéficie du fait d'être 'dans le mouvement', et sans doute mieux adapté aux futures évolutions du domaine.

Domaines d'activité

En ce qui concerne le point de vue du domaine d'activité des deux langages au sein de l'ingénierie des besoins, la question se pose de savoir dans quelle mesure ils sont *concurrents*. En effet, nous avons vu qu'*Albert II* se prêtait bien à l'activité de *modélisation* tout en pouvant balayer un peu plus, et couvrir certains aspects de la *validation* et de l'*élicitation*. Par contre, **SCR**, tout en pouvant couvrir l'activité de *modélisation* (malgré quelques restrictions quant à l'intérêt de modéliser des systèmes dans leur entièreté), se révèle très efficace pour ce qui est de la *vérification*. Il existe donc une certaine concurrence entre les deux langages, concurrence que l'on va quand même tâcher de résoudre.

En fait, plutôt que de parler de concurrence, il est préférable de considérer une complémentarité entre les deux approches, comme le propose [DUBOIS98b]. Ainsi, il serait évident de laisser la modélisation à *Albert II*, et la vérification des parties critiques des spécifications à **SCR**. Cette approche possède quelques avantages : tout d'abord, elle permet de conserver l'expressivité des spécifications tout en apportant les avantages de la vérification à des parties du système où la sécurité est primordiale. Ensuite, elle dégage le client de devoir s'habituer à une notation un peu technique, tout en permettant aux analystes de se repartir les spécialisations, les uns couvrant les aspects *Albert II*, les autres en vérifiant les parties critiques. Enfin, c'est une aide évidente à la validation des spécifications, les résultats de la vérification **SCR** appuyant l'analyse *Albert II*.

On peut dès lors enrichir le modèle des activités de l'ingénierie des besoins à ce point de vue :

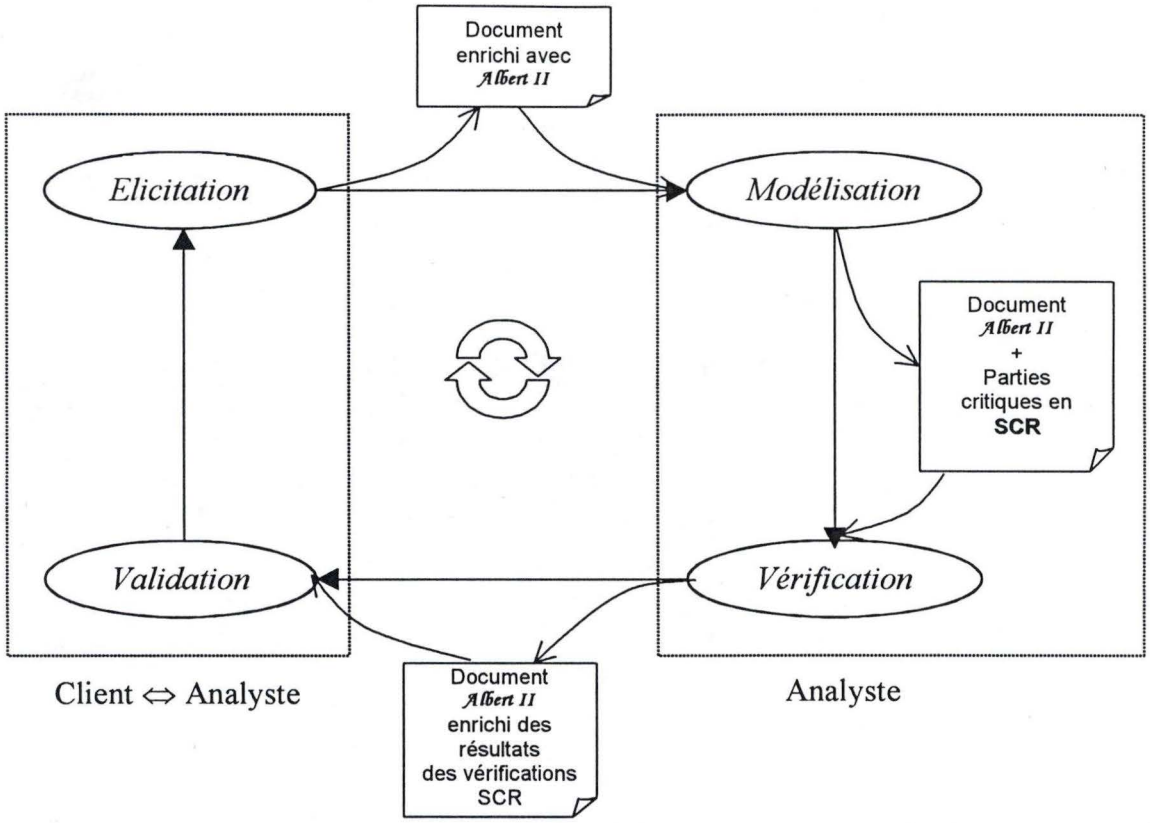


Figure 18. Schéma d'activité de l'ingénierie des besoins enrichi.

7. Conclusion

Tout au long de ces chapitres, nous avons découvert deux manières fort différentes de réaliser l'analyse des besoins : la méthodologie **SCR**, issue du NRL; et le langage *Albert II*, de l'Institut d'Informatique.

La première, basée sur une représentation tabulaire très opérationnelle, permet de réaliser des analyses de contraintes fonctionnelles vérifiables à l'aide d'outils de systèmes ou l'aspect de sécurité joue un rôle primordial. En plus des outils traditionnels de gestion d'analyse, les outils de vérification comme un simulateur de spécification et un vérificateur de modèle, apportent un plus indéniable à l'activité de l'analyste.

La suivante se veut expressive, déclarative et naturelle. Définitivement orientée client, elle tire avantage d'une aisance particulière à représenter les systèmes distribués où les aspects temps réels peuvent être importants. L'ajout d'un animateur aux outils de support permet d'en renforcer l'expressivité durant la validation des spécifications.

Enfin, nous avons comparé ces méthodes. De cette comparaison est née l'idée d'une complémentarité de ces dernières au sein du domaine d'activité, la coopération résultante augmentant l'efficacité du processus global de réalisation des cahiers de charge.

2ème partie
Le module JavaSimFront

Chapitre 5
De la simulation à l'animation

1. Introduction

SCR* et son simulateur

Lors de la première partie de l'exposé, nous avons découvert l'ensemble **SCR*** de support à la notation **SCR**. Nous avons pu apprécier l'apport de son simulateur en matière de vérification et d'outil de validation. Durant la présentation, nous avons abordé le désire des membres du NRL de réaliser des extensions du simulateur permettant l'animation de variables, c'est-à-dire, l'association à chaque variable d'un composant graphique interactif à partir duquel un utilisateur donné pourrait manipuler le simulateur sans jamais avoir à s'occuper des variables sous-jacentes.

Tcl/Tk

Quelques expériences de ce type ont été réalisées par les associés du Docteur Heitmeyer. Elles furent basées sur des interfaces construites à partir d'une version améliorée de l'ensemble Tcl/Tk appelée TAE+. Cet ensemble comprend deux choses : d'une part le langage de scripte Tcl (Tool command language); et d'autre part sa librairie graphique Tk (ToolKit). Cet ensemble a fourni des résultats appréciables dans le cadre d'animation de spécifications **SCR**. [HEITMEYER98] nous fait part des impressions obtenues lors d'un test réalisé en 1997 : "*Both the Navy manager and the safety engineers were very positive about our simulation and animation of the SRS (Software Requirements Specification). They viewed this as an effective means for validating the SRS*". Le processus d'animation est relativement simple. Le schéma suivant en donne un aperçu :

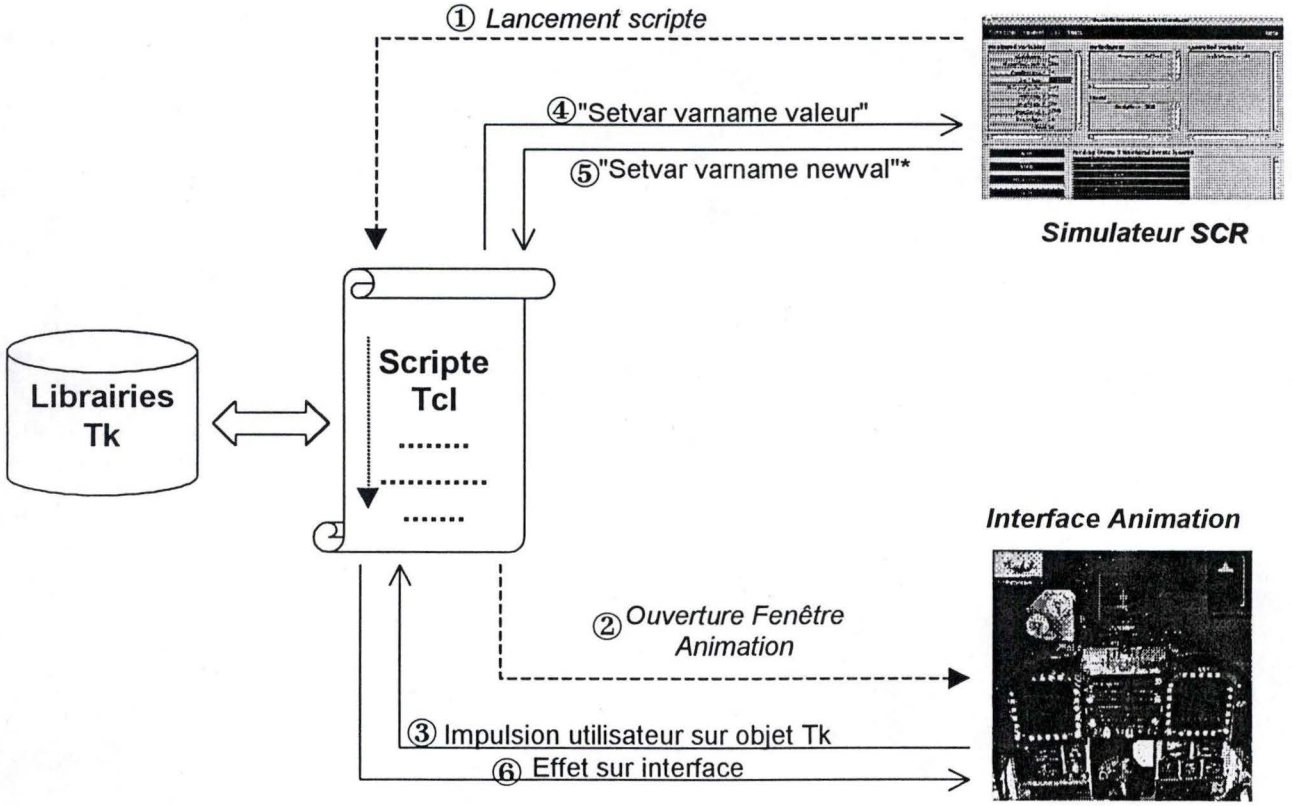


Figure 19. Animation SCR sous Tcl/Tk.

Tout d'abord, à partir du simulateur, l'utilisateur presse le bouton "Open/Close Simulator Front-end", et choisit un scripte existant pour la simulation qui s'exécute (①). La validation du choix lance l'exécution du scripte. Celui-ci "pioche" dans les librairies Tk les objets graphiques nécessaires et les affiches dans une fenêtre d'animation(②). A partir de ce moment, il est possible d'animer les spécifications. L'interface d'animation saisit les interactions de l'utilisateur sur les objets Tk de la fenêtre (③). Le scripte Tcl interprète ces interactions et communique au simulateur le nom de la variable correspondant à l'objet graphique, ainsi que la nouvelle valuation de cette variable (④). La communication se fait selon un protocole prédéfini et compris par les deux processus. Le simulateur calcule le nouvel état des spécifications sur base de la valeur de la variable passée par le scripte, et renvoie la liste des variables modifiées ainsi que leurs nouvelles valeurs (⑤). Enfin, Tcl, sur base de ces variables et du script, fait évoluer la représentation des objets Tk (⑥). Le processus continue tant que l'une des deux fenêtres (simulation – animation) reste ouverte.

Ce procédé permet de faire fonctionner parallèlement le simulateur et la fenêtre d'animation. Cela autorise dès lors de modifier la valeur d'une variable via le simulateur et d'en voir les résultats dans la fenêtre d'animation, l'inverse étant également possible. Par contre, si cette manière de faire procure de bons résultats, elle reste néanmoins tributaire d'un désagréable inconvénient : celui de devoir *programmer* chacune des interfaces. Or cet effort n'est pas semble-t-il pas évident et tributaire de spécialisations qu'il est difficile d'exiger de la part d'analystes, même spécialisés. Et puis surtout, c'est consommateur de temps. Donc l'idée est venue de réaliser un éditeur d'interface pour le simulateur. Cet éditeur utiliserait des composants graphiques interactifs prédéfinis, permettrait de graphiquement "relier" ceux-ci aux variables d'une spécification donnée, et autoriserait la réutilisation d'ensembles de composants adéquats à une spécification donnée. Or, depuis quelques mois maintenant, l'intérêt pour la vague Java et son modèle d'objets distribués JavaBeans a submergé le marché. Le concept de JavaBeans a séduit les gens du NRL et est alors né le projet JavaSimFront ou **JSF**, pour Java Simulator Front-End Editor.

2. Le projet JavaSimFront

Objectif

L'objectif du projet **JSF** était donc la réalisation d'un éditeur d'interfaces basé sur des composants graphiques JavaBeans, chacun des composants devant pouvoir être liés à une ou plusieurs variables du simulateur de manière à ce que les comportements des variables soient dépendants de leurs pendants graphiques, et inversement. Les interfaces ainsi créées doivent alors être capables d'interagir à distance avec une spécification donnée du simulateur, tout comme le faisaient les réalisations précédantes sous Tcl/Tk.

Intérêts des JavaBeans dans ce cadre

Les avantages procurés par les JavaBeans pour ce genre d'application sont multiples. Comme nous le verrons un peu plus loin, les JavaBeans sont avant tout des "*composants logiciels distribués et réutilisables que l'on peut manipuler graphiquement* [SUN97]". De cette définition, on peut retirer les avantages suivants pour le projet **JSF** :

- Ce sont des composants logiciels. Cela signifie que l'on peut potentiellement *tout* réaliser sous forme de Beans à partir du moment où on désire se mettre à la

programmation Java. Dans le cas du projet, comme il est difficile de prévoir à l'avance quels vont être les représentations dont on va avoir besoin d'une spécification à une autre, cette souplesse ouvre toute grande les portes de l'imagination des analystes.

- Ce sont des composants distribués et réutilisables. Cela sous-entend deux choses. D'une part, que les JavaBeans ne doivent pas nécessairement se trouver sur la station où se trouve l'éditeur pour pouvoir être utiliser, permettant donc la mise en place d'une base de données de JavaBeans commune, et dans laquelle tous les analystes pourraient venir "piocher" ce dont ils ont besoin. D'autre part, la réutilisabilité permet de séparer le Beans proprement-dit de ce que l'on en fait; la définition des Beans étant toujours disponible pour de multiples instanciations. Ces aspects vont permettre de séparer complètement d'éditeur de ses composants, permettant ainsi, par exemple, de pouvoir faire une animation chez le client sur base d'une simulation distante, et avec une version "light" de l'éditeur, les JavaBeans non-utilisés ne devant pas être localisés au même endroit que l'éditeur.
- Ce sont des composants manipulables graphiquement. Cela signifie simplement que les JavaBeans offrent des facilités qui les rendent exploitables via des environnements graphiques. Généralement, les JavaBeans ont leur propre représentation, mais selon l'environnement d'édition, les JavaBeans n'ayant pas de représentations peuvent se voir attribuer une représentation par défaut. Les JavaBeans peuvent subir de multiples manipulations, l'étendue de celles-ci dépendant directement de leurs outils d'édition. Mais à titre d'exemple, citons simplement la création de liens dynamiques entre JavaBeans ou encore la personnalisation des propriétés des Beans. Cet aspect permettra de simplifier considérablement le travail des analystes.

En somme, l'utilisation de JavaBeans dans ce cadre permet de bénéficier de tous leurs avantages. Le principal objectif de ce projet est donc de réaliser un environnement permettant de manipuler les JavaBeans, exercice intéressant d'autant plus qu'à l'époque, peu d'exemples de tels environnements existaient, si ce n'est la BeanBox de Sun et quelques environnements de développement comme le JavaWorkshop, toujours de Sun.

Le projet

Le projet fut présenté fin novembre 1997, et devait comporter 3 phases. La première phase d'une durée d'environ 2 semaines, était d'étudier comment faire pour qu'un outil Java puisse *communiquer avec le simulateur*, sans changer quoique ce soit aux outils existants. Cette phase devait aussi permettre à votre serviteur d'apprendre Java, les concepts liés à JavaBeans, et de pouvoir évaluer la faisabilité du projet selon les objectifs fixés. La deuxième phase comprenait la réalisation d'un *environnement simplifié* d'édition des Beans, avec en sus la proposition d'une *interface* de manipulation des liens entre Beans et variables **SCR**. Cette phase devait se terminer début avril 1998 par la réalisation d'un outil permettant d'évaluer les potentialités d'une telle approche et pouvant servir d'outil de démonstration lors d'une présentation ayant lieux courant avril. Enfin, la dernière phase consistait à *finaliser l'éditeur* et intégrant toutes les fonctionnalités utiles à ce genre d'outil.

La première phase du projet a abouti à la réalisation d'un petit serveur d'informations à partir duquel plusieurs perspectives intéressantes peuvent être envisagées. La deuxième phase

a résulté sur un environnement de manipulation d'objets graphiques que l'on peut directement relier à des variables **SCR** et tester avec le simulateur, la troisième phase étant toujours en cours de réalisation.

3. Petit exemple introductif

Avant d'aborder les parties suivantes, il serait interressant de donner un petit exemple de manipulation de JavaBeans de manière à clarifier la vision du lecteur sur les utilisations potentielles et l'intérêt des Beans dans le contexte de l'éditeur.

Thermostat et Lumière

Imaginez que vous ayez à animer une petite spécification tournant sur le simulateur **SCR**. Cette spécification comprend une variable d'état *thermostat* et une variable d'action *lumière*, qui représentent respectivement un thermostat de maison à partir duquel on peut faire varier la puissance des radiateurs de l'habitation, et une lampe qui s'allume lorsque la température effective de la maison descend ou augmente au-delà d'une valeur donnée. Un système donc très simple.

Historiette

Vous ouvrez votre outil d'édition d'interfaces et vous examinez la série de Beans mis à votre disposition. Parmis ceux-ci vous choisissez le Bean "thermostat" que d'un simple drag&drop, vous placez judicieusement dans la fenêtre d'édition. Vous faites de même avec le Bean nommé "voyant lumineux". Ensuite, vous allez dans la fenêtre reprenant les variables que le simulateur utilise, et d'un nouveau drag&drop, vous placez la représentation de la variable *lumière* sur le Bean "voyant lumineux". Comme c'est une variable d'action, l'éditeur vous demande quelle propriété du Bean doit être influencée par la valeur de la variable *lumière*. Parmi les propriétés du Beans que l'éditeur vous propose, vous lui indiquez la propriété "state" qui indique si le voyant est allumé ou éteint. Ensuite, vous faites glisser la variable *thermostat* sur l'autre Bean, et là, l'éditeur vous demande à quel événement le Beans doit réagir pour faire changer la valeur de la variable. Vous lui indiquez l'événement "MouseClicked". Ensuite, il vous demande quelle propriété du Bean doit être passée à la variable *thermostat*. Vous lui indiquez la propriété "value" représentant la valeur actuelle du thermostat. Vous en profitez pour donner aux propriétés "maxval" et "minval" du Beans, les valeurs maximum et minimum acceptables par la variable *thermostat*. Il vous reste à tester votre nouvelle interface. Vous pressez le "slider" du Bean "thermostat", et vous constatez qu'en dessous d'une certaine valeur, le Bean "voyant lumineux" change de couleur. Votre animation fonctionne.

Constatation

Au-delà du caractère trivial et un peu déterministe de ce petit exemple, on peut constater que le processus à l'air plutôt simple. Il l'est dans la manipulation. Mais il l'est beaucoup moins en coulisses. Mais pour l'utilisateur, cette simplicité est la force des JavaBeans. Construire un environnement d'accueil et de manipulation de Beans nécessite une bonne maîtrise des certains aspects de la programmation Java autour des JavaBeans. Le prochain chapitre va nous introduire à une partie de ces aspects.

Chapitre 6
Prérequis : le modèle JavaBeans

1. Introduction

L'objectif du présent chapitre n'est pas de donner "les milles et unes astuces pour écrire un Bean", mais d'introduire les concepts essentiels qui vont nous permettre d'envisager un outil de manipulation de Beans. Dans le petit exemple du précédent chapitre, nous avons parlés d'"événements" et de "propriétés". Ce sont deux des aspects des Beans. Un troisième aspect est celui de "méthodes".

→ les méthodes des JavaBeans

La seule manière d'interagir avec un objet donné est d'invoquer ses méthodes. En effet, les objets, à l'instar de toutes classes instanciées écrites en respect de la philosophie orientée-objet, ne rendent aucun de leurs champs accessibles de l'extérieur à l'exception des champs déclarés `public`. Cependant, contrairement aux classes normales, le mécanisme de bas niveau d'invocation "classique" de méthode n'est pas la seule manière de manipuler les Beans. Les méthodes publiques sont reléguées à un rôle secondaire dû à la présence de deux mécanismes de plus hauts niveaux préférés : les propriétés et les événements. Il est toujours possible d'interagir avec les méthodes publiques pour manipuler les Beans, mais cela n'est pas la manière idéale.

→ les propriétés des JavaBeans

Les propriétés ne sont rien de moins, conceptuellement parlant, que des attributs classiques d'objets mais qui sont supportés par un API spécifique destiné à la lecture et l'écriture de ceux-ci. De simples conventions permettent aux outils d'identifier automatiquement les méthodes éditants les propriétés d'un Bean donné. Dès qu'un outil connaît ces méthodes, il peut permettre l'édition des propriétés de celui-ci et dynamiquement changer son comportement.

→ les événements des JavaBeans

La meilleure manière pour un Bean de communiquer des informations avec d'autres composants est par l'envoi et la réception d'événements. De cette manière, les Beans peuvent se connaître entre-eux, et créer ainsi des "réseaux" de Beans communiquants. Le prochain point introduit le modèle événementiel des Beans.

En résumé, on peut voir les JavaBeans comme des boites noires, dont une parties des composantes, en fonction de leurs types, seraient "visibles" et accessibles de l'extérieur (Figure 20).

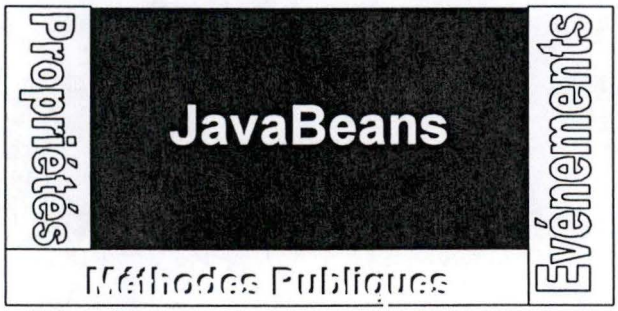


Figure 20. Métaphore de la boîte noire pour JavaBeans.

2. Le modèle événementiel de délégation

Le modèle énévementiel supporté par Java est basé sur le concept d'"auditeurs d'événements". Un objet intéressé par la réception d'événements est déclaré auditeur d'événement (*Event listener*). Un objet qui génère des événements (*Event source*) maintient une liste des auditeurs qui sont intéressés par la survenance d'un événement, et fournit des méthodes qui permettent aux auditeurs de s'ajouter ou de se retirer eux-mêmes de cette liste. Quand une source génère un événement, elle signale à tous ses auditeurs que l'événement est survenu. Il fait cela en invoquant une méthode de l'auditeur et en passant un objet de type *événement*. Ce procédé implique que tous les auditeurs intéressés doivent implémenter les méthodes adéquates. Ceci est assuré par l'obligation pour les auditeurs d'implémenter l'interface correspondant au type d'événement qu'il désire "écouter" (Figure 21).

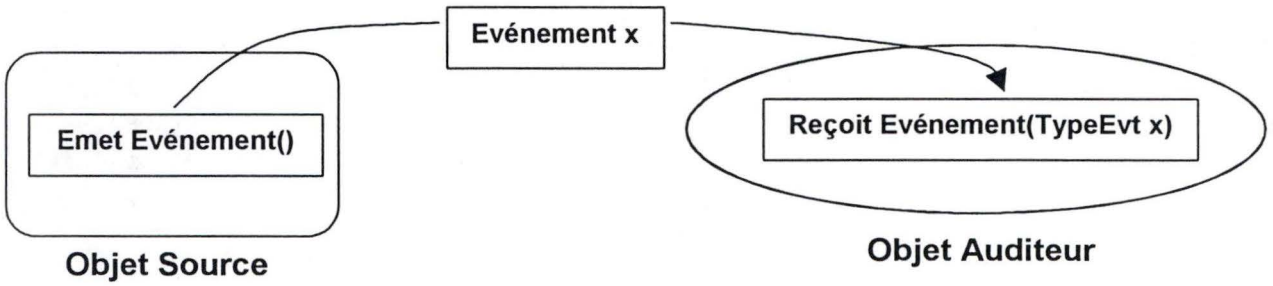


Figure 21. Modèle de délégation des événements.

Ainsi, à titre d'exemple, les objets désirants réagir aux événements de la souris doivent implémenter l'interface `MouseListener`, ainsi que les méthodes requises par cette interface. En voici la définition :

```
public interface MouseListener extends EventListener
{       public void mouseClicked(MouseEvent e);
        public void mouseEntered(MouseEvent e);
        public void mouseExited(MouseEvent e);
        public void mousePressed(MouseEvent e);
        public void mouseReleased(MouseEvent e);
}
```

Ensuite, une fois qu'une classe a implémenté une interface qui peut réagir à un type d'événement donné, on doit enregistrer cette classe auprès d'une source d'événement instanciée. Cet enregistrement suit des conventions d'écriture de telle manière que si une source d'événement génère des événements du type *x*, il y a une méthode appelée `addxListener()` qui ajoute un auditeur à cette source. Si on désire par exemple que la classe `ABC` suivante soit auditeur de l'objet `source` pour les événements souris, cela donne :

```
class ABC implements MouseListener //ABC doit implémenter l'interface MouseListener
{...
        Object source = new Object();     //On instancie l'objet source
        source.addMouseListener(this);    //On enregistre ABC(this)auprès de la source
...
        public void mouseClicked(MouseEvent e)//Lorsque l'on va cliquer sur la source
        {       //on peut modifier une valeur, appeler une méthode,...
        }
...
}
```

On voit donc comment on peut associer objets et réactions aux actions sur ces objets. Le principe est bien entendu équivalent pour les Beans. Si on veut qu'un Bean réagisse à un type événement donné, on peut dès lors le signaler comme source, et tous les objets auditeurs intéressés n'auront qu'à s'enregistrer auprès de lui pour réagir. Cela nous sera utile par la suite.

3. Les propriétés des JavaBeans

L'aspect le plus aisé d'emploi et le plus visible des JavaBeans est son ensemble de propriétés. Les propriétés permettent de personnaliser les JavaBeans de manière dynamique, à travers une fenêtre d'édition de l'éditeur ou de manière automatique. Tout changement fait sur une propriété d'une Bean va se refléter normalement directement sur le Bean concerné. Afin de distinguer les propriétés au sein des Beans, des conventions d'écriture ont été imposées. Tout ce qui ne respecte pas ces conventions ne sera pas reconnu comme propriété d'un Bean, et cours donc le risque d'être inutile.

Getters et setters

Tout d'abord, les propriétés sont définies complètement par la présence ou l'absence de méthodes d'accès. Ces méthodes d'accès peuvent être classées en deux catégories, les méthodes de lecture ou "getters", et les méthodes d'écriture ou "setters". Une propriété de lecture est définie par la présence d'une méthode dont la signature est la suivante :

```
public <typePropriété> get<nomPropriété>()
```

La propriété d'écriture prend cette forme :

```
public void set<nomPropriété>(<typePropriété> nomParamFormel)
```

Toutes les méthodes d'accès doivent être déclarées publiques, et pour une propriété *p* donnée, par convention de représentation son nom commence toujours par une minuscule, sauf quand utilisée dans le nom d'une méthode, où la première lettre est alors en majuscule. Ces conventions doivent être respectées car les JavaBeans reposent entièrement sur elles.

De plus toute propriété doit être déclarée `protected`.

A titre d'exemple, pour notre petit Bean "thermostat" du chapitre 5, nous aurions eut :

```
...
protected int temperature;
...
public int getTemperature() {...}
public void setTemperature(int newTemp) {...}
```

Ensuite, il existe 5 catégories de propriétés : les propriétés simples (celles que nous venons de voir), les propriétés booléennes, les propriétés indexées, les propriétés limitées, et les propriétés contraintes. Examinons-les rapidement :

Propriétés booléennes

Une propriété très simple utilisées comme alternative à la méthode de lecture classique pour les propriétés booléennes :

```
public boolean is<nomPropriété>()
```

Le "setter" quant à lui ne bouge pas.

Propriétés indexées

Ces propriétés permettent de supporter les tableaux et/ou leurs éléments individuels. De manière stricte, les tableaux non pas vraiment besoins de mécanismes supplémentaires que les conventions de représentations existantes :

```
public <typePropriété>[] get<nomPropriété>()
public void set<nomPropriété>(<typePropriété>[] nomParamFormel)
```

Pour accéder à des éléments individuels de tableaux, les conventions deviennent :

```
public <typePropriété>[] get<nomPropriété>(int index)
public void set<nomPropriété>(int index, <typePropriété> valeur)
```

Propriétés limitées

Ce type de propriétés (*bound properties*) permet aux Beans de générer des événements lorsque leurs valeurs changent. Il faut tout d'abord pour cela que la classe utilisant le Bean s'enregistre comme auditeur de celui-ci par (comme vu au point précédent) :

```
public void addPropertyChangeListener(PropertyChangeListener x);
```

puis il faut que dans le Bean, on déclare les propriétés limitées du Bean. Cela se fait par création d'une instance de `PropertyChangeSupport`, défini comme suit :

```
public class PropertyChangeSupport extends Object implements Sreializable
{       public PropertyChangeSupport(Object sourceBean);
        public synchronized void addPropertyChangeListener(PropertyChangeListener auditeur);
        public synchronized void removePropertyChangeListener(PropertyChangeListener auditeur);
        public void firePropertyChange(String nomPropriete, Object oldValue, Object newValue);
}
```

Propriétés contraintes

Tout comme les propriétés limitées, les propriétés contraintes (*constrained properties*) envoient un événement lorsqu'une de leurs propriétés est modifiée, mais permettent aussi à ce changement d'être interdit par l'auditeur. Ce cas est intéressant, mais afin de ne pas nous perdre dans des détails techniques, nous ne l'aborderons pas ici. Le lecteur peut en trouver une bonne explication dans [Vanhelsuwe97].

Nous venons donc de faire le tour des types de propriétés qu'un Bean peut avoir. On le constate, potentiellement tout dans un Bean peut être paramètré : sa forme, sa couleur, le fait qu'il puisse être re-dimensionné ou pas, le texte qui l'accompagne, ... tant que ses propriétés

sont accessibles grâce à des conventions de représentation. Mais cela n'est pas le seul problème, encore faut-il que l'outil qui manipule les Beans puissent faire une sorte de bilan des méthodes, propriétés ou événements que la personne qui a créer le Bean a voulu mettre à disposition de l'outil. C'est le sujet du prochain point.

4. Introspection et BeanInfo

java.beans.Introspector

Un des aspects qui fait le succès de Java, c'est la possibilité de charger dynamiquement des classes. Dans le cas qui nous concerne, cela signifie que l'éditeur de Beans ne connaît pas à l'avance les Beans qu'il va devoir manipuler. Il faut donc que de manière dynamique, l'éditeur puisse déterminer les propriétés, événements et méthodes qu'un Bean donné supporte. Cela se réalise par l'utilisation de mécanismes d'*introspections*. Ce mécanisme est basé sur le mécanisme de *réflexion* permettant d'obtenir des informations concernant les membres d'une classe (contenu dans le *package* `java.lang.reflect`). C'est un mécanisme très puissant, mais relativement complexe, car applicable à toutes les classes, Beans compris. Cependant il serait fastidieux, et contraire à la philosophie des JavaBeans, de devoir écrire un processus complet d'analyse de contenu de Bean. Il y aurait des problèmes de standards entre les outils de manipulation, sans parler du risque d'erreur. Donc, l'introspection vient simplifier fortement le processus, et justifie les règles d'écritures exposées précédemment. L'introspection particularise le mécanisme de réflexion aux seuls Beans, un simple appel à l'*"introspecteur"* permettant dès lors de recevoir un objet `BeanInfo` contenant l'ensemble des informations désirées. A l'annexe 5, le lecteur peut trouver la définition de ces classes ainsi qu'un exemple des classes les plus utiles pour le problème qui nous concerne. Si vous y jetez un œil, vous constaterez que malgré la difficulté supposée du processus, la classe `Introspector` est particulièrement simple.

java.beans.BeanInfo

A partir de l'objet `BeanInfo` obtenu en résultat de l'introspection, il est possible de connaître tous les composants publics d'un Bean donné : méthodes, événements, propriétés. L'attitude par défaut de l' *"introspecteur"* est de repérer au sein d'un Bean tous les composants susceptibles d'être utiles sans distinction les uns des autres. Mais, il est possible d'attribuer au mécanisme d'introspection une attitude plus déterministe, et de lui fournir uniquement ce que le programmeur du Bean a précisément voulu montrer. Il faut pour cela que lors de l'écriture d'un Bean, appelé `abc` par exemple, le programmeur ait créer une classe `abcBeanInfo` implémentant l'interface `BeanInfo`. A partir de cette classe, il est possible de raffiner l'interface du Bean et d'enrichir sa personnalisation. Ce sont des aspects importants pour celui qui désire programmer des Beans. Ils le sont moins de notre point de vue, le processus d'introspection étant le même, que cette classe soit présente ou non. A titre d'information, le contenu de l'objet `BeanInfo` permet par exemple d'ajouter la présence d'une icône de représentation du Bean, ou d'un *Wizard* (appelé *Customizer*) afin simplifier le processus de personnalisation des propriétés d'un Bean donné.

La figure suivante expose de manière très simplifiée le processus global d'introspection :

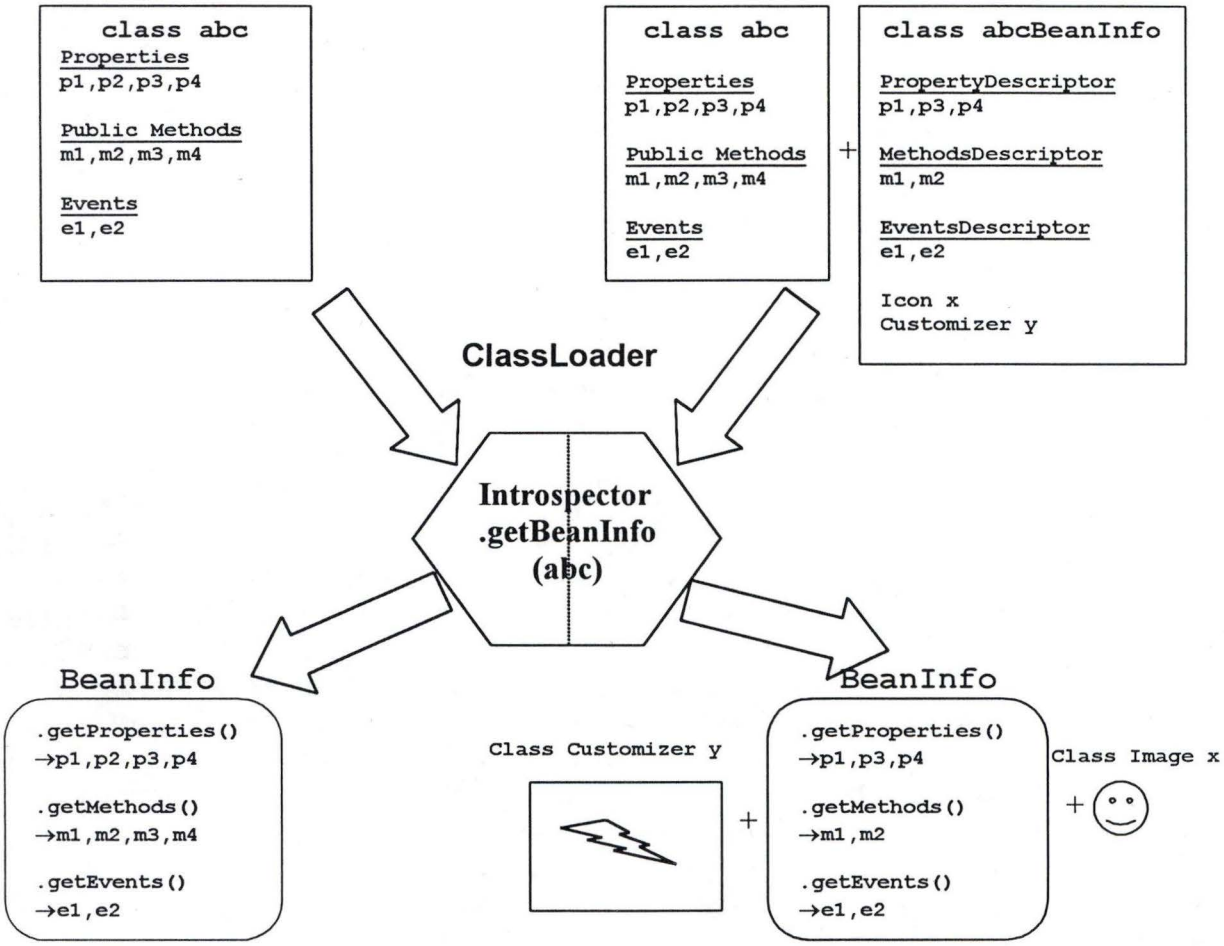


Figure 22. Vue simplifiée du processus d'introspection.

5. Format JAR et *Serialization*

Le format JAR

Jusqu'à présent, nous avons vu découvert le modèle événementiel sous-jacent à Java, et donc aux JavaBeans. Nous avons rencontré les différents types propriétés propres aux Beans, et vu comment, à partir du processus d'introspection, il est possible d'obtenir une image précise (via BeanInfo) des composantes visibles des Beans. Dans le point précédent, nous avons vu qu'un simple Bean peut être composé de multiple fichier :

- sa classe principale;
- une classe `BeanInfo` éventuellement associée;
- et une représentation sous forme d'icône (un fichier GIF).

Des Beans plus complexes peuvent requièrent quelques fichiers supplémentaires :

- une classe pour le `Customizer`;
- une classe pour un éditeur de propriété additionnel;
- une collection de classes liées aux classes de support du Bean.

Or, les Beans sont supposés être complets, composés de fichiers indissociables les uns des autres. Mais le risque de perdre tout ou une partie d'un fichier durant la distribution des Beans existe. Sun a donc introduit l'outil de compression JAR (Java ARchive), qui permet de créer des fichiers compressés Java diminuants les risques de distribution.

Le format JAR est entièrement compatible avec le format ZIP. Ce qui distingue l'un de l'autre, c'est que le format JAR impose que le premier fichier soit un fichier de description du contenu du fichier archive. Ce fichier doit obligatoirement avoir le *path* suivant :

```
META-INF/MANIFEST.MF
```

Le fichier porte donc le nom de `MANIFEST.MF`, et doit être situé dans le sous-répertoire `META-INF` de la racine de l'archive. Le format général de ce fichier est :

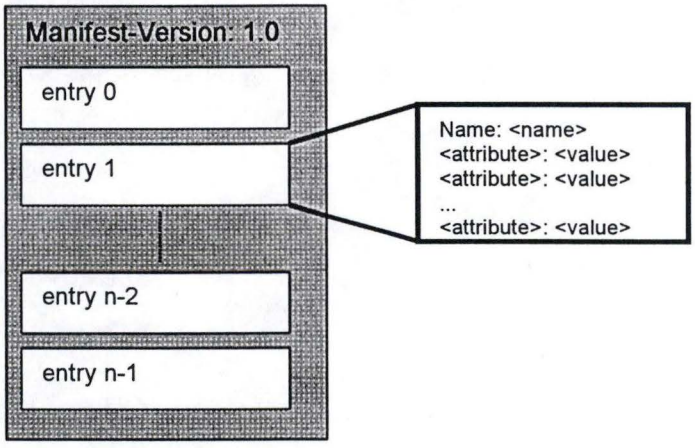


Figure 23. Format général du MANIFEST.MF.

Sans entrer dans les détails de ce fichier, signalons qu'il reprend : le nom des fichiers de l'archive, des valeurs de vérification (*checksum*) et l'identification des Beans contenus dans l'archive.

Ce qui est important, c'est que l'éditeur doit donc être capable de lire les Beans au sein même des fichiers archives. Le *package* `java.io` fournit les outils nécessaires à ce processus. Voir [VANHELSUWE97] ou un équivalent pour plus de détails concernant ce point.

Le problème de la *sérialisation*

Il nous reste un point à aborder afin de couvrir plus ou moins complètement cette brève introduction :le problème de la sauvegarde des interfaces que l'on construirait à base de Beans. Pour réaliser cette sauvegarde, il va falloir construire une structure d'objets qui va conserver l'état des Beans et des leurs relations à un moment donné. Cette structure permettra plus tard de restaurer l'état de l'interface. Ce problème est lié au problème de la *sérialisation* d'objets. Un objet est *sérialisable* si une fois instancié, son état peut être transformé en flux de bits pour être stocké sur une mémoire non-volatile. Pour cela, l'objet doit implémenter l'interface `Serializable` et remplir certaines propriétés (voir [MAX1997] pour plus de détails). La *sérialisation* est un exercice compliqué qui nécessiterait un peu plus de développement, mais nous ne nous pencherons pas plus en avant sur ce point dans le cadre du présent travail.

6. Conclusion

A la lecture de ce chapitre, on peut se rendre compte de la complexité intrinseque des Beans. Ce sont des composants d'une grande maniabilité et d'une grande souplesse. Mais cette souplesse aurait pu se faire au prix de la robustesse. Sun a cependant visiblement mis suffisament de moyens en oeuvre pour que l'architecture des Beans bénéficie d'une robustesse correcte. L'application JavaSimFront pourra être juge d'une partie de cette robustesse, tout comme de la souplesse des Beans. En effet, l'un des questions majeures que l'on peut se poser concernant cette application, c'est de savoir comment on va pouvoir relier le comportement des variables **SCR** à des Beans. C'est le sujet du prochain chapitre.

Chapitre 7
Problématique d'intégration Bean-SCR

1. Introduction

Nous savons maintenant à peu près tout ce dont nous avons besoin de savoir pour avoir une idée de la manière dont on peut manipuler les Beans et créer des associations entre-eux et les méthodes, événements ou propriétés d'autres objets ou Beans. Cependant, comme vous pouvez déjà l'imaginer, créer un éditeur de JavaBeans n'est pas trivial. Toute une série de questions doivent être résolues, certaines seront liées à l'environnement de manipulation, d'autres, comme nous l'avons vu précédemment, à la *sérialisation* des objets ou encore liées aux problèmes de sécurisation dans un environnement distribué. Mais la majorité de ces considérations peuvent trouver des réponses en parcourant l'abondante littérature à propos des Beans ou de la programmation Java. Le présent chapitre concerne plutôt l'évaluation d'un problème propre à la situation qui nous concerne. Il s'agit de répondre à la question : "J'ai une variable **SCR**, je désire lier son comportement à un Beans : que se passe-t-il ? Comment faire la liaison ? A quelles contraintes suis-je astreint ?" Tâchons de répondre à ces quelques questions.

Considérations préalables

Dans l'exposé qui va suivre, nous allons faire une série de suppositions sur l'outil, de manière à nous dégager de considérations un peu secondaires par rapport à la question soulevée :

- L'outil d'édition est capable de communiquer avec le simulateur **SCR** selon son protocole.
- L'outil possède une interface d'édition des variables actuellement utilisées par la simulation **SCR**.
- L'outil est capable de lire un fichier contenant des Beans, et de les instancier. Tout les Beans mis à sa disposition sont représentés dans une fenêtre.
- Il existe une fenêtre d'édition dans laquelle les utilisateurs peuvent venir "installer" des Beans et les relier à des variables **SCR**.
- La méthode de manipulation des Beans, comme des variables, est le drag&drop; tout Bean déposé dans la fenêtre d'édition s'y instancie; toute variable déposée sur un Bean instancié se lie à lui.
- Les propriétés, événements et méthodes d'un Bean instancié sont parfaitement éditables par l'outil via le processus d'introspection, et le travail réalisé par l'outil autour de l'objet de type `BeanInfo` est parfaitement transparent pour l'utilisateur.

En résumé, nous nous trouvons dans un environnement idéal et notre seule considération va être le processus de liaison d'un Bean et d'une variable **SCR**. Nous ne considérerons pas les liaisons de Bean à Bean. Mais cela pourraît figurer dans une extension de l'outil.

2. Analyse du processus de liaison variables-Beans

Pour rappel, les types de données **SCR** sont au nombre de quatre : entiers, réels, booléens et énumérés. Ils y a aussi 3 types de variables : les variables d'état (*monitored variable*), d'action (*controlled variable*), et les termes. Ces dernières ne seront pas prises en compte, n'ayant pas nécessairement besoin d'être représentées par un Beans (bien qu'elles le

pourraient). Nous devons donc considérer deux types de création de lien : le lien d'une variable d'état avec un Bean; et celui d'une variable d'action. Analysons ce dernier processus.

Liaison variables d'action – Beans

Action utilisateur n°1 : Il "dépose" une variable d'action sur un Bean instancié dans la fenêtre d'édition.

Les variables d'action représentent des effecteurs agissant sur l'environnement du système. Dans ce cas précis, un effecteur va être représenté par un Bean dont le comportement est déterminé par les variations des valeurs de ses propriétés. Il faut donc lier une des propriétés d'un Bean donné à une variable d'action pour que le changement de valeur de la variable d'action modifie le comportement du Bean. Un Bean pouvant avoir plusieurs propriétés, l'outil doit permettre à l'utilisateur d'en choisir une.

Processus résutlants
⇒ Recherche des propriétés du Bean concerné par analyse de l'objet BeanInfo de ce Bean.
⇒ L'outil propose à l'utilisateur de choisir une des propriétés du Bean.

Action utilisateur n°2 : Il choisit la propriété correspondant au comportement que la variable doit induire au Bean.

Ici se pose la question de la correspondance des types entre celui de la variable et celui acceptable par la propriété. En effet, si par exemple la variable est un booléen, et que la propriété attend un paramètre de type {On, Off}, il va falloir résoudre ce conflit. Cette question sera traitée dans le prochain point. Pour le moment, contentons-nous de considérer que la correspondance des types est établie. On va donc créer le lien entre la variable et la propriété d'un Bean.

Processus résutlant
⇒ Création du lien entre la variable et le Bean

Pour réaliser ce lien, on peut envisager plusieurs solutions. Une des plus évidentes est de réaliser une table de correspondance entre le nom de la variable **SCR**, le nom du ou des Beans ayant un lien avec cette variable, et par Bean concerné, les noms des propriétés liées à la variable. Lorsque la valeur de la variable change, il suffit alors de parcourir la table et pour chaque Bean correspondant, on value les propriétés indiquées de la nouvelle valeur de la variable.

Le processus que l'on vient d'analyser soulève donc une question : "Comment réaliser l'association si la variable n'a pas le même type que celui du paramètre de la propriété associée ?". Passons à l'analyse du processus de lien entre variables d'état et Beans.

Liaison variables d'état – Beans

Action utilisateur n°1 : Il "dépose" une variable d'état sur un Bean instancié dans la fenêtre d'édition.

Les variables d'état représentent des senseurs captants des impulsions de l'environnement sur le système. Dans ce cas précis, un senseur va être représenté par un Bean dont le comportement est déterminé par les actions qui vont être faites sur lui. Il faut donc lier un Bean donné à un type événement. Un Bean peut réagir à plusieurs types d'événement, l'outil doit permettre à l'utilisateur d'en choisir un.

Processus résutlants

⇒ Recherche des événements auquel le Bean concerné répond par analyse de l'objet BeanInfo de ce Bean.

⇒ L'outil propose à l'utilisateur de choisir un des événements proposés.

Action utilisateur n°2 : Il choisit un des événements correspondant à celui auquel le Bean doit réagir pour induire un changement de valeur de la variable d'état.

Ici, il faut que l'événement choisi corresponde à l'événement interne qui modifie la propriété cible. La figure suivante (Figure 24) explique cette restriction. Le Bean répond en interne à 2 types d'événements, respectivement X et Y. Lorsqu'un événement X survient, le Bean modifie la propriété A, et quand un événement Y arrive, il modifie la propriété B. Imaginons maintenant que l'on désire lier une variable d'état à un Bean que l'on voudrait voir réagir à l'événement Y (le Bean devenant *source* d'événement pour son "environnement"). Si lors de la survenance de cet événement, on examine la valeur de la propriété A pour la donner comme nouvelle valeur de la variable d'état, ça ne sert à rien, la propriété A n'ayant pas changée d'un iota. Il aurait fallu faire l'association sur base de l'événement X.

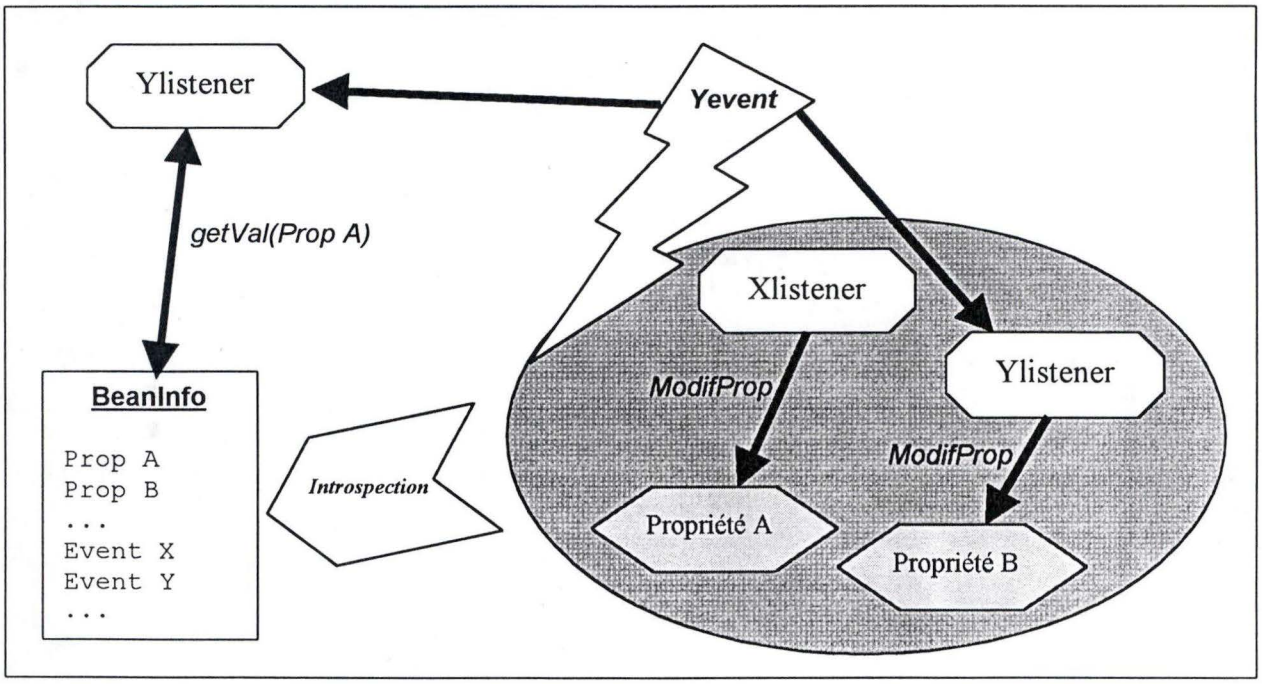


Figure 24. Problème des correspondances d'événements internes et externes à un Bean.

Un corollaire à la contrainte précédente est que pour qu'un Bean puisse être utilisé comme source d'événement provoquant la modification de la valeur d'une variable, est qu'il faut nécessairement que le Bean modifie en interne la valeur de la propriété susceptible de représenter la variable. Il est courant de programmer des Beans dont l'apparence se modifie uniquement lorsque leurs propriétés changent. Mais les Beans qui nous intéressent doivent

aussi changer leurs propriétés en réaction à des événements. Ce problème est dû au fait que les Beans sont des sortes de boîtes noires, dans lesquelles on ne connaît que les noms des composants publics. Mais il est par contre impossible de savoir quelles méthodes (autre que les 'getters' et 'setters') modifient des propriétés[5], ni sous quelles conditions elles sont modifiées.

Processus résutlants

⇒ Notification de l'événement choisi comme source d'événements pour les "oreilles" de l'entourage du Bean concerné.

⇒ L'outil propose à l'utilisateur de choisir une des propriétés du Bean, afin que celle-ci puisse passer sa valeur à la variable d'état lorsque l'événement choisi surviendra.

Action utilisateur n°3 : Il choisit la propriété correspondant à la variable d'état concernée.

Ici, encore survient le problème de correspondance des types entre les variables et les propriétés.

Processus résutlant

⇒ Création du lien entre la variable et le Bean

Ce processus été abordé dans un point précédent. Considérons ce lien comme établi.

On peut cependant donner une approximation en pseudo-code Java de la manière d'implémenter ce processus du point de vue événements :

```
Class BeanEnvironnement implements XListener; // la classe est auditeur
{       ...
        public void NotifieBean()
        {       ...
                BeanConcerne.addXListener(this);// le bean concerné est source
                ...
        }
        public void eventfired(XEvent e) // quand un événement de type X survient...
        {       ...                      // on va chercher la valeur de la propriete
                VarSCR.setVal(BeanConcerne.getVal(ProprieteConcernee);// concernee du
                                         //bean, et on l'assigne à la variable SCR
                ...
        }
}
```

Le problème de liaison des variables d'état avec des Beans n'est pas encore terminé. Il faut aussi que le comportement d'une variable d'état concernée influence celui du Bean concerné. En effet, il est aussi possible de manipuler les variables d'état à partir du simulateur **SCR**. Dès lors, on doit appliquer le même processus que pour les variables d'action. Mais comme on connaît déjà la propriété associée à la variable, l'opération est automatique et donc transparente pour l'utilisateur.

Evaluation

[5] Les propriétés contraintes et limitées ne subissent pas cette limitation. On aurait pu, avec ce genre de propriétés, lier directement une contrainte à la modification d'une variable. Mais afin de rester général, cette hypothèse a été écartée, bien qu'un 'bon' outil doit en tenir compte.

Nous avons distingué une série de processus important. Un semblant de solution se profile à l'horizon. Mais un problème reste encore à résoudre : comment faire correspondre les types des variables avec ceux requis par les propriétés des Beans ? Evaluons ce point.

3. Correspondance des types et des valeurs

1ère approche

Le problème de correspondance des types peut s'envisager selon le type des variables **SCR** que l'on désire lier. On peut clairement remarquer chez les variables entières et réelles des différences avec les variables booléennes et énumérées. Le fait de distinguer dès à présent ces différences va nous aider par la suite :

Variables entières et réelles :

- Ensemble des valeurs acceptables connu, très grand.
- Une valeur initiale.
- Parfois bornées supérieurement et/ou inférieurement.
- Un indice de précision généralement unitaire.

Variables énumérées :

- Ensemble des valeurs acceptables non-connu, généralement petit.
- Une valeur initiale.

Variables booléennes :

- Ensemble des valeurs acceptables connu et petit.
- Une valeur initiale.

Le processus global de vérification de correspondance va être de :

1 – comparer les types respectifs de la variable SCR et de la propriété du Bean concerné;

2 – si la correspondance est possible, lier variable et propriété;

3 – si la correspondance n'est pas possible, refuser le lien.

Comparaison des types respectifs

Tout d'abord, pour une variable **SCR** donnée, nous connaissons son type, celui-ci étant fournit par le simulateur lors de l'établissement de la connexion avec l'outil.
Il nous reste donc à découvrir le type de la propriété, connaissant son nom. Pour cela, l'introspection va nous venir en aide. Si on examine à nouveau la définition de l'interface **BeanInfo** (annexe 5), on constate la déclaration de la méthode **getPropertyDescriptor** retournant un ensemble de **PropertyDescriptor**. Via la méthode **getPropertyType** retournant un objet de type **Class**, on peut obtenir le type de la propriété concernée. A titre d'information, voici un morceau de pseudo-code Java permettant ce processus :

```
...
BeanInfo bi = Introspector.getBeanInfo(bean);
...
protected String getPropertyType(BeanInfo beanInfo, String propertyName)
{     PropertyDescriptor[]    props;
      PropertyDescriptor      propertyDescriptor;
```

```
        String                  propType = null;
        ...
        props = beanInfo.getPropertyDescriptors();
        ...
        for (int i=0; i < props.length; i++)//on parcours la liste des props
        {     propertyDescriptor = props[i];
              ...
              if (propertyDescriptor.getName()==propertyName)
              {     propType = propertyDescriptor.getPropertytype().getName();
              }
        }
        return propType;
}
```

Maintenant que nous pouvons comparer le type de la variable avec le type de la propriété, la correspondance est aisée. Pour la suite de l'exposé, nous supposerons que cette correspondance est établie.

Types SCR	Types Java
Integer	int
Float	float
Boolean	boolean
Enumerated	java.lang.String

Table 11. Comparaison des types SCR et Java.

Attribution des valeurs

La phase suivante consiste à attribuer les valeurs de la variable aux propriétés du Bean. Pour cela, nous distinguerons les variables selon leurs types de données.

⇒ *variables entières et réelles*

Tout d'abord, on attribue la valeur actuelle de la variable **SCR** à la propriété concernée. Si la variable **SCR** est bornée, deux possibilités s'offrent : soit le Bean permet de spécifier une valeur maximum et/ou minimum pour cette propriété via un outil de personnalisation ; soit, on réalise un contrôle dynamique sur la valeur de la variable avant que celle-ci soit communiquée au simulateur. C'est sans doute cette dernière méthode la plus adéquate, la première dépendant du type de Bean auquel on a affaire. Si une variable se voit attribuer une valeur illégale, il suffit de corriger automatiquement la valeur de la variable et de modifier la valeur de la propriété. Ce procédé peut être appliqué aussi dans le cas où un indicateur de précision serait spécifié.

⇒ *variables booléennes*

Le cas des variables booléennes est trivial. On attribue simplement la valeur actuelle de la variable à la propriété concernée.

⇒ *variables énumérées*

Le cas des variables énumérées est plus délicat car, si on connaît l'ensemble des valeurs acceptées par la variable **SCR** ainsi que leur nombre, ça n'est normalement pas le cas pour la propriété qui nous intéresse. Plusieurs cas peuvent se présenter :

- Ou bien, le Bean a été construit expressément pour être manuellement personnalisable en fonction d'une variable énumérée donnée, et dans ce cas, il suffira de configurer le Bean. C'est certainement la solution la plus élégante, mais qui nécessite un petit effort de programmation.
- Ou bien le Bean est quelconque. Dans ce cas, à moins de connaître à priori l'ensemble de ses valeurs acceptables, il n'est pas possible de connaître de manière automatique cet ensemble. Si on le connaît, on peut alors passer par une table de correspondance (Figure 12) qui sera utilisée lors de chaque valuation de la propriété ou de la variable concernée.

Val acceptées var SCR x	Val acceptées propriété A
BOcc	val1
Start	val2
Stop	val3
Undef	val4

Table 12 . Exemple de table de correspondance des valeurs.

4. Conclusion

Ce chapitre nous a permis d'évaluer la faisabilité du projet. Nous nous sommes rendu compte que d'une part, il était parfaitement possible de relier les variables **SCR** aux composants Beans; d'autre part qu'il était quand même nécessaire d'introduire quelques restrictions quant à :

- l'utilisation des variables d'état avec les Beans. Ces derniers doivent respecter quelques contraintes pour pouvoir être exploitables avec ce genre de variable (point 7.2.).
- l'utilisation de variables de type énuméré. A moins de connaître par avance les valeurs acceptables par les propriétés d'un Bean quelconque, il vaut mieux faire appel aux fonctionnalités de *personnalisation* de certains Beans écrits pour ces cas précis. Sinon, l'ensemble des autres variables peut être assez facilement "lié" à un Bean donné.

Bien évidemment, ce chapitre ne répond pas à toutes les questions. Ainsi, on peut se demander s'il vaut mieux écrire des Beans "généralistes", permettant autant de connecter des variables entières qu'énumérées (à l'aide des outils de personnalisation); ou vaut-il mieux écrire des Beans très spécialisés, justes valables pour un type de valeur ou même un nombre restreint de valeur ? Est-il possible de créer une concurrence entre les Beans associés à des variables ? Que cela risque-t-il d'entraîner ? On peut encore en ajouter, mais examinons maintenant l'état du projet du point de vue programme.

Chapitre 8
Le serveur proxy JSF

1. Introduction

Le petit serveur proxy JSF est essentiellement le résultat d'une application test qui devait me permettre de découvrir la programmation Java, et de considérer comment on pouvait faire communiquer des applications Java avec le simulateur. Au fur et à mesure que se construisait cette petite application, on pouvait deviner l'intérêt qu'elle pourrait susciter si on pouvait passer un peu de temps à la faire évoluer.

Dans un premier temps, le serveur proxy était un simple application permettant de tester les communications par sockets, et de découvrir le protocole utilisé par le simulateur pour communiquer avec les scripts TAE+. Le serveur possède une partie "client" connectée au simulateur, et une partie "serveur" où l'application JSF vient se connecter (Figure 25).

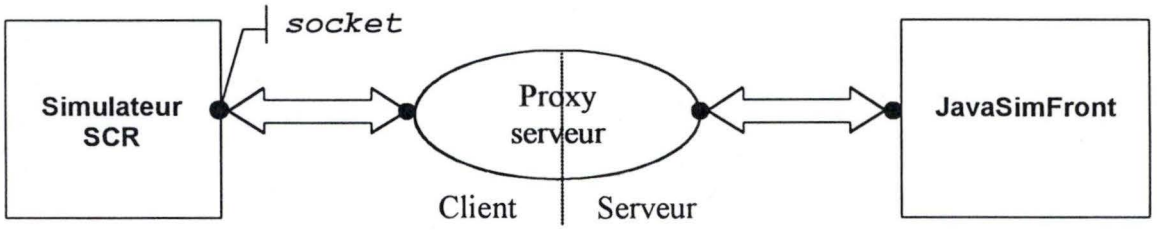


Figure 25. Structure d'ensemble projet JSF.

Bien entendu, c'est avant tout un petit outil de test et de correction d'erreur. Mais comme nous le verrons un peu plus loin, il est aussi plein d'espoir.

2. Le protocole d'échanges

Ceci est une brêve présentation du protocole utilisé par les applications du projet. Ce protocole, n'est pas très évolué, le simulateur étant toujours un outils prototype. Mais il est suffisant pour les objectifs du projet JSF.
Chaque message commence avec un code numérique identifiant le type du message qui suit. Certains messages contiennent des commentaires qui se terminent par un caractère de retour à la ligne et dans lesquels les sauts de lignes litéraux sont symbolisés par le caractère '\'. Le protocole fournit les types messages suivants:

Nom de code	Origine	Description	Implémenté
kSF_ErrorSym	Simulateur - JSF	Erreurs	non
kSF_Comment	Simulateur – JSF	Le commentaire qui suit est ignoré	oui
kSF_TypeDict	SImulateur	spécifie le dico des type	non
kSF_VarDict	Simulateur	Specfie le dico des variables	non
kSF_VarChange	Simulateur – JSF	Changement de valeur de variable	oui
kSF_Restart	Simulateur – JSF	Redémarrage de la simulation	oui
kSF_Shutdown	Simulateur – JSF	Ferme la fenêtre du simulateur	oui
kSF_Freeze	Simulateur	Gèle le Simulateur	oui
kSF_Unfreeze	Simulateur	Dégèle le Simulateur	oui
kSF_ModalMsg	Simulateur	Affiche une Fenêtre Modale	oui
kSF_ModalMsgDismiss	JSF	Efface la fenêtre modale	oui

Table 13 . Définition des types des messages du protocole SCR*.

Dans la définition complète du protocole dont on fait grâce ici, les noms de code sont représentés par des chiffres allant de 0 à 10. Dans la pluspart des cas, c'est la commande *4* d'assignation de variable qui sera la plus utilisée. Ainsi, "*4 ACAirborne no*" assignera la valeur *no* à la variable *ACAirborne*.

3. Ouverture de l'application

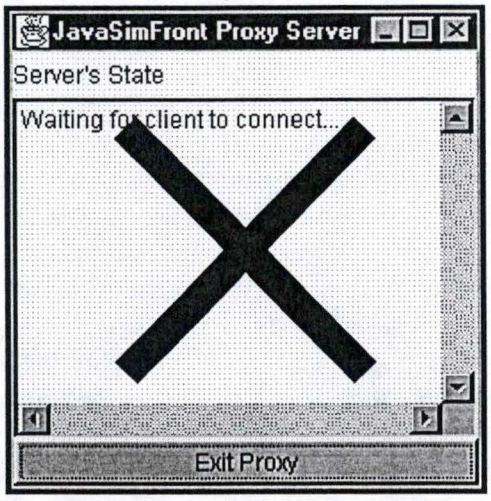


Figure 26. Le JavaSimFrontProxy Server.

Pour ouvrir le serveur proxy, il suffit de faire exactement la même opération que pour l'ouverture d'une animation TAE+. Mais au lieu d'ouvrir un script Tcl, l'application lance un script Unix définissant les variables d'environnement et mettant en route le serveur. Par convention pour les tests, dès que le simulateur détecte la connexion de la partie client du serveur, il envoie le contenu complet des dictionnaires **SCR** des spécifications en cours d'exécution, suivi des valuations des valeurs actuelles des variables de ces spécifications. Pour exemple, voici un échantillon d'un de ces envois:

```
BombReleaseDemoSub
Table  Type Dictionary Type Dictionary 5     5
Name   Base Type      Units  Legal Values   Comment
Distance       Integer feet   [0 , k50nmi]   (6072 ft per nmi)
MFSW   Enumerated     N/A    natt, boc, ccip, tf, nattoff, bocoff, none   Positions of the Master Function Selector Switch.
Switch  Enumerated     N/A    on, off
WeapTyp Integer N/A    [0 , 99]
YesNo  Enumerated     N/A    yes, no Legacy of A-7 spec
Table  Constant Dictionary    Constant Dictionary    2     5
Name   Class  Type   Value  Comment
k42nmi  Constant       Distance       255024  42 nmi X 6072 ft/nmi
k50nmi  Constant       Integer 303600  50 nmi X 6072 ft/nmi
....
4 ACAirborne no
4 BombRelease off
4 MasterFcnSwitch none
4 MissDistance 1000
4 Overflown 0
4 ReleaseEnable off
4 Stn1Ready no
4 Stn8Ready no
4 TargetDesig FALSE
4 WeaponType 0
4 time 0
```

L'exemple provient d'une simplification des spécifications de l'A-7. Cette exemple sera repris sur l'ensemble des photos d'écran qui vont agrémenter cette présentation.

4. Intérêts

Tout d'abord, d'un point de vue personnel, ce petit serveur fut très utile durant la phase de développement de l'application JSF. L'une des raisons principales fut qu'il me permit de continuer à développer chez moi le programme JSF "on-line", c'est-à-dire comme si JSF pouvait toujours se connecter au simulateur, le serveur "simulant" le comportement du simulateur.

Mais là n'est pas son principal intérêt. Tout d'abord, il peut servir de **serveur unique à de multiples clients**, chacun ayant accès à la même simulation. Ensuite, c'est un endroit intéressant pour créer un accès à une **base de données de JavaBeans**. En effet, sa position "médiane" lui permet de participer à la gestion du trafic au cours de l'utilisation du simulateur, et de pourquoi pas, de gérer les accès au JavaBeans de manière à ce qu'un client civil ne puisse pas avoir accès à des Beans comprenants des informations d'ordre militaire. Dans le même ordre d'idée, un base commune va permettre à tous les clients de pouvoir profiter des créations des autres clients. De plus, cela permet de créer des clients "light", donc d'augmenter la portabilité de JSF. Enfin, il peut aisément être inséré dans **un environnement sécurisé** de type *firewall*. Comme les clients du serveur pourraient communiquer via Internet, la nécessité d'une telle particularité pourrait se faire sentir.

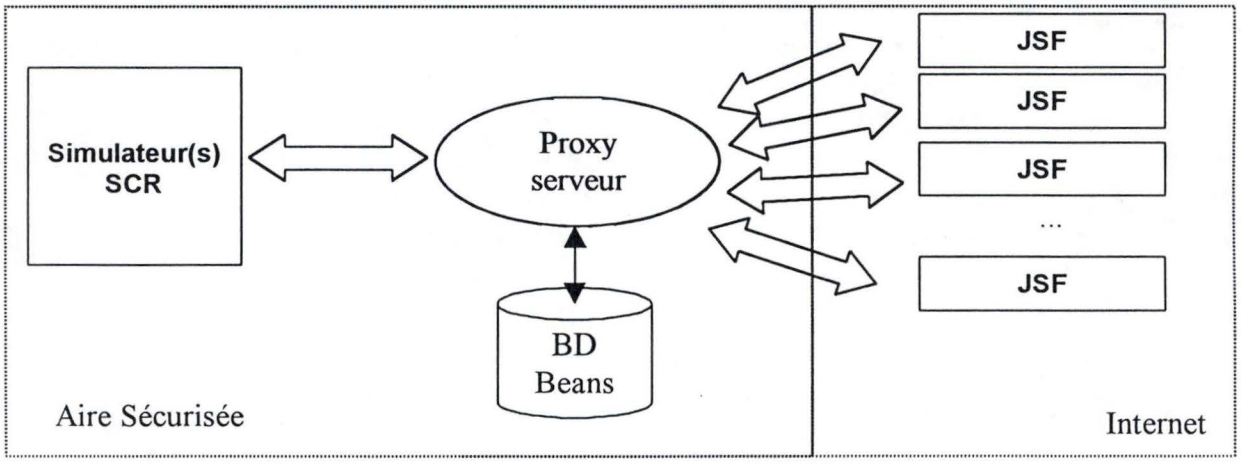


Figure 27. Evolutions autour du serveur proxy.

5. Conclusion

L'évolution autour du serveur proxy n'apporte rien de vraiment neuf, ce genre de structure étant de plus en plus utilisées, surtout avec l'avènement des Intranets [IM96]. Pour citer un exemple vu précédemment, [HEYMANS97b] nous expose une approche similaire avec l'animateur *Albert II*, le simulateur de l'un comme l'animateur de l'autre ne pouvant que bénéficier d'une telle approche.

Chapitre 9
L'éditeur JSF

1. Introduction

Comme évoqué dans le premier chapitre de cette partie, le projet JavaSimFront était initialement divisé en 3 phases. Le présent chapitre à pour objectif de présenter les résultats de la deuxième phase du projet, c'est-à-dire la réalisation d'un *environnement simplifié* d'édition des Beans, avec en sus la proposition d'une *interface* de manipulation des liens entre Beans et variables **SCR**. Cette phase devait se terminer début avril 1998 par la réalisation d'un outil permettant d'évaluer les potentialités d'une telle approche et pouvant servir d'outil de démonstration lors d'une présentation formelle ayant lieux mi-avril.

La deuxième phase comportait en fait deux échéances. La première correspondait à la mi-février 98, et je devais y présenter une proposition d'interface pour l'éditeur, ainsi qu'une proposition des fonctionnalités que l'outil devait être capable de supporter. La deuxième échéance survenait donc début avril avec une présentation de l'outil supportant partiellement les JavaBeans. On peut dors et déjà dire que les objectifs furent *partiellement* atteints.

En fait, si l'interface semblait inspirer l'équipe du Docteur Heitmeyer, l'échéance du 1er avril a poussé l'orientation de l'outil vers une voie légèrement différente que ce qui était initialement attendu. Après la mi-mars, comme l'échéance approchait et que l'outil ne manipulait pas encore les JavaBeans de manière suffisamment stable, je fus contraint de fournir une version de l'outil proposant les mêmes fonctionnalités que l'outil initialement prévu mais sans les fonctionnalités JavaBeans. En lieu et place, j'ai donc rapidement créer des composants graphiques qui avaient le comportement approximatif de Beans afin que l'équipe du Docteur Heitmeyer ait une maquette fonctionnelle à présenter à la mi-avril.

C'est cette maquette que je vous propose de découvrir maintenant.

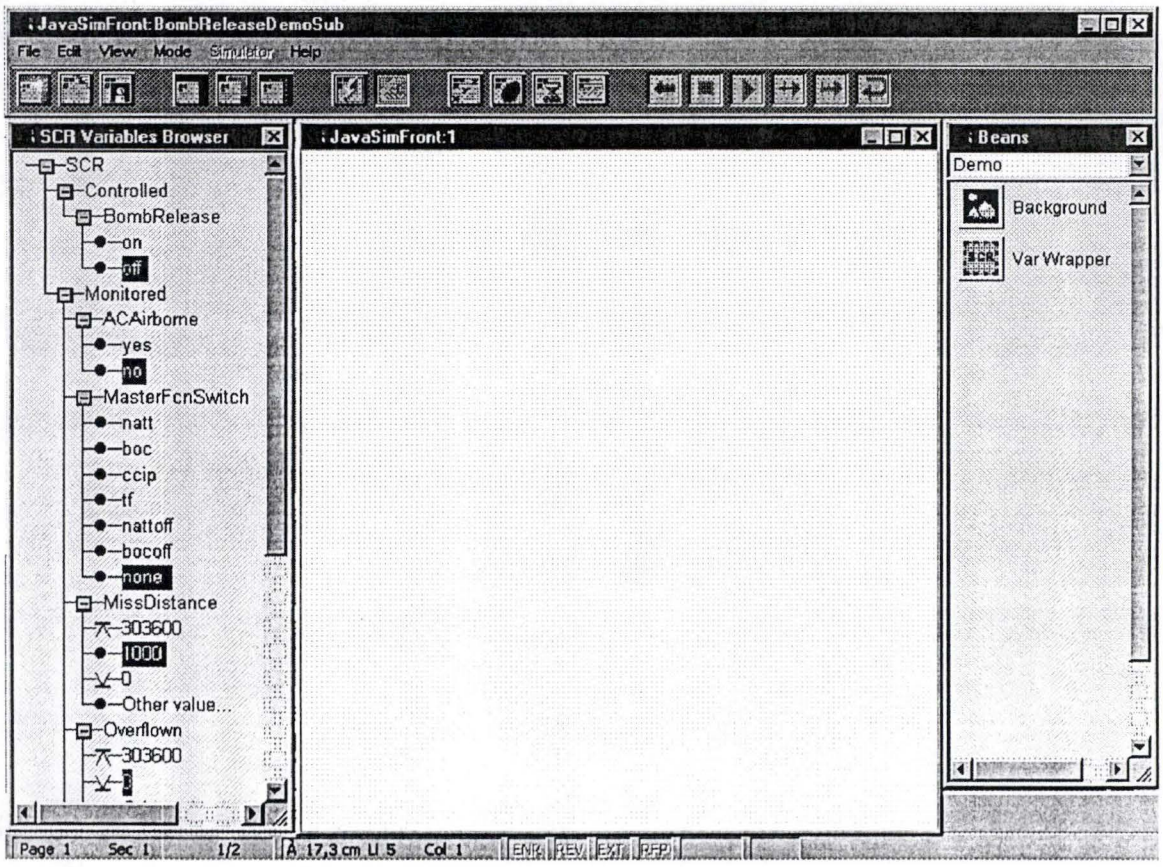


Figure 28. JavaSimFront.

2. L'application principale JSF

Une des premières fonctionnalités de l'application principale est sa connexion avec le simulateur, généralement via le serveur proxy. Afin d'établir la connexion, cette dernière demande à l'utilisateur sur quelle station le serveur tourne (Figure 29). La première station proposée est toujours la station locale. Les autres stations dépendent de l'utilisation habituelle de l'outil. Il est aussi possible de se connecter à une station distante. Notez qu'il y a moyen de faire fonctionner l'outil en mode non-connecté, auquel cas, on ne recevra pas la définition des variables tournant actuellement sur le simulateur. Une extension future de l'outil devrait prévoir de sauvegarder et de charger les dictionnaires reçus en local, de manière à pouvoir les éditer sans être connecté.

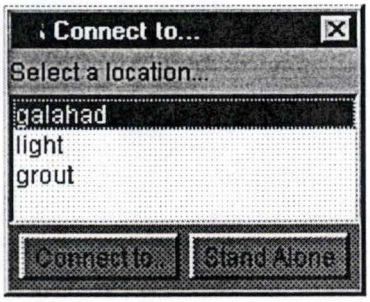


Figure 29. Boîte de dialogue *Connect to...*

La quasi-totalité des fonctionnalités de l'application est reprise sur une barre d'outil sur laquelle sont disposées les icônes représentant ces fonctionnalités. Mais avant toute chose, il faut savoir que l'outil peut fonctionner sur **deux modes distincts**. Le premier mode est le "mode d'édition, le deuxième le mode "animation":

- Le mode Edition: lorsque ce mode est enclenché, une série de fonctionnalité devient disponible. Les boutons ou menus des fonctionnalités indisponibles deviennent grisés (Figure 30). Les fonctionnalités actives pendant le mode Edition seront indiquées par un (E).

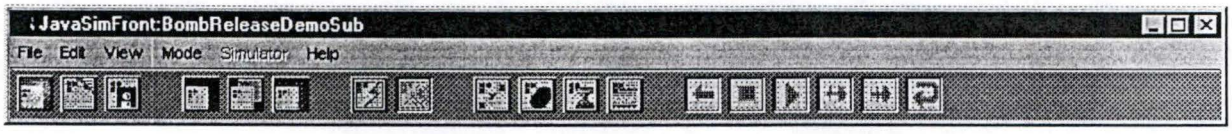


Figure 30. Barre d'outils en mode Edition.

- Le mode Simulation: même remarques que pour le mode Edition. Les fonctionnalités actives pendant le mode Animation seront indiquées par un (A).

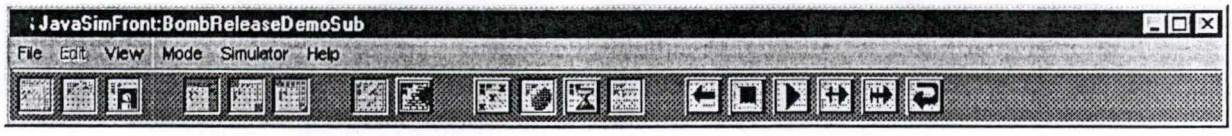


Figure 31. Barre d'outils en mode Animation.

Les fonctionnalités de l'application principale

Nouvelle Fenêtre d'édition : JSF supporte plusieurs fenêtre d'édition à la fois. Cette fonctionnalité permet d'en créer une nouvelle et de l'ouvrir.(E)

Ouvrir une sauvegarde : Cette fonctionnalité permet de charger une animation. Non encore implémenté.(E)

Sauvegarder : Permet de sauver une animation. Non encore implémenté.(E)

Couper, Copier, Coller: les traditionnelles fonctionnalités d'aide à l'édition. Non encore implémenté.(E)

Interface Simulateur : Cette fonctionnalité permet d'afficher ou de rendre invisible la fenêtre de visualisation des variables **SCR**.(A/E).

Interface Beans : Permet d'afficher ou de rendre invisible la fenêtre de sélection des Beans. (E)

Historique : Permet de visualiser ou de cacher l'historique des commandes passées entres le simulateur et l'éditeur (E/A).

Propriétés: Quand un Beans est sélectionné, permet d'afficher la fenêtre d'édition de ses propriétés.(E)

Back, Stop, Play, Step, Micropstep, Replay :Commandes de scénario distant. Permet de définir comment exécuter un scénario qui se trouve sur le simulateur.(A)

Voici donc l'ensemble des fonctionnalités de l'application principale. Chacune des fonctionnalités vues ici possède évidemment son équivalent dans les menus.

3. L'interface du simulateur

Cet objet est sans doute le plus important après celui permettant la manipulation des Beans. Cette interface propose une vision des variables **SCR** tel qu'elles sont exécutées dans le simulateur. Les fonctionnalités que l'on va les retrouver dans le simulateur. Cette fenêtre n'est donc ni plus ni moins qu'une interface alternative distante du simulateur **SCR**(Figure 32). Les variables y sont représentées sous forme d'arbre, le premier niveau faisant la séparation entre variables d'état et variables d'action, le deuxième niveau reprenant les variables elles-mêmes, laissant le troisième niveau aux valeurs des variables du niveau supérieur.

Chaque branche ou nœud de l'arbre peut être "fermée" ou "ouverte", faisant apparaître ou disparaître le sous-arbre correspondant. Les valeurs courantes des variables sont en mode inversé (blanc sur fond noir). Les valeurs initiales sont en bleu. Une valeur bleu inversée est

donc courante et initiale. Les variables entières ou réelles bornées ont leurs bornes marquées par des petites flèches.

Quand l'application est en mode Animation, un double-click sur une valeur permet de soumettre cette valeur au simulateur. Un double click sur "Other value..." permet d'assigner une nouvelle valeur courante aux variables entières ou réelles

Quand l'application est en mode Edition, un drag&drop d'une variable sur un (pseudo-)Bean permet d'associer cette variable à ce Bean.

A tout moment, d'un click-droit, on peut afficher un menu popup proposant quatre fonctionnalités :

- Open entire tree : déploie complètement l'arbre
- Close tree : ferme complètement l'arbre
- Show Current Value Only : seul les valeurs actuelles des variables seront affichées
- Show var info : permet à une fenêtre popup d'afficher les informations additionnelles d'une variable quand le curseur passe sur cette variable

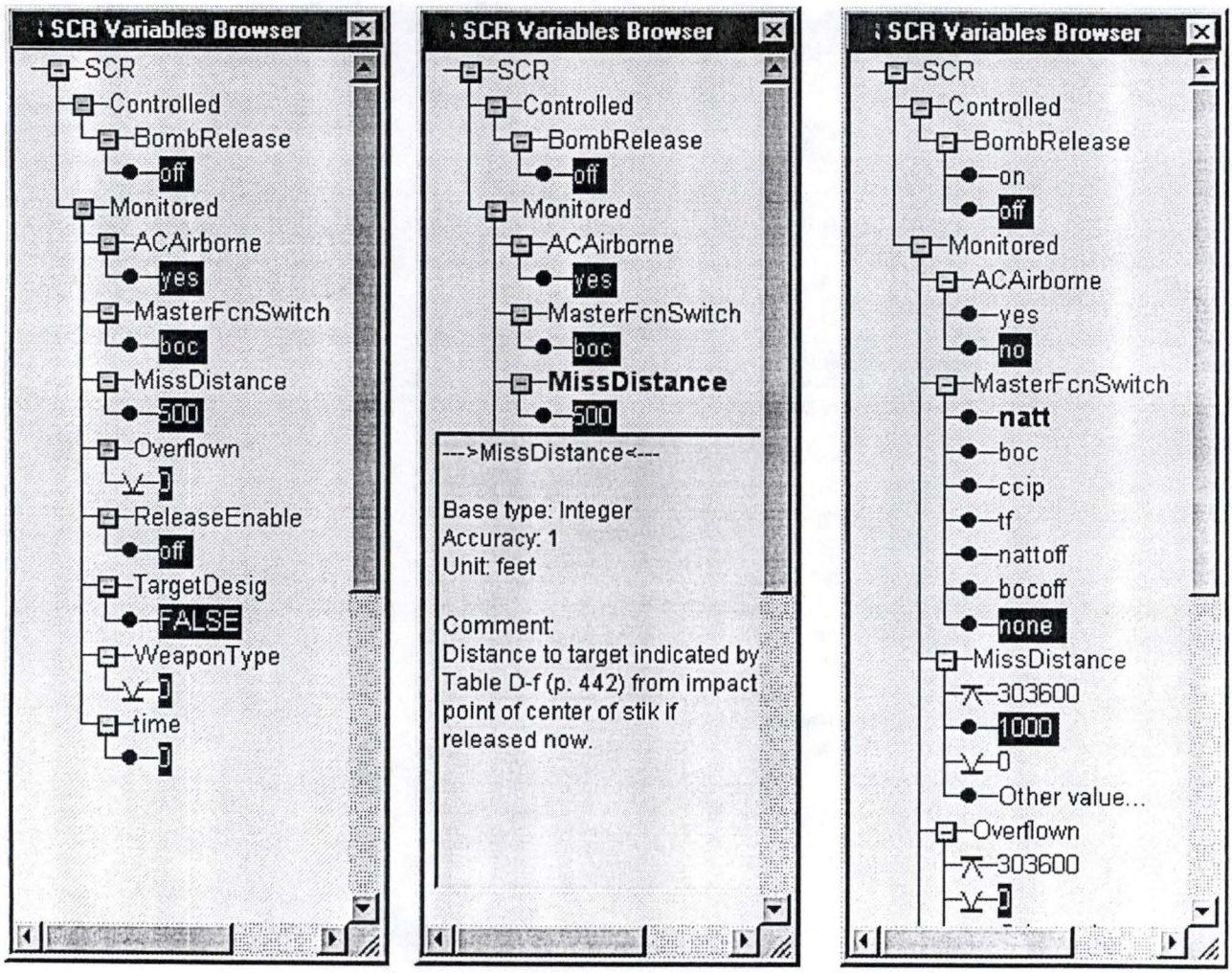


Figure 32. Différents états de la fenêtre de visualisation des variables SCR.

4. Fenêtre d'édition des Beans

Cette fenêtre est le pendant de la fenêtre de visualisation des variables. Cette fenêtre présente simplement les Beans dans une liste, le contenu de la liste pouvant varier selon la catégorie que l'on choisit dans la liste déroulante supérieure ("demo" dans l'exemple). Les catégories sont en fait les sous répertoires de la racine du programme dans lesquelles les Beans ont été mis. Dans le mode édition, on peut, d'un simple Drag&Drop, prendre un Bean et le déposer dans la fenêtre d'édition. Durant le mode animation, aucun Bean n'est évidemment accessible. Chaque Bean est représenté par une icône s'il en est pourvu, et par son nom.

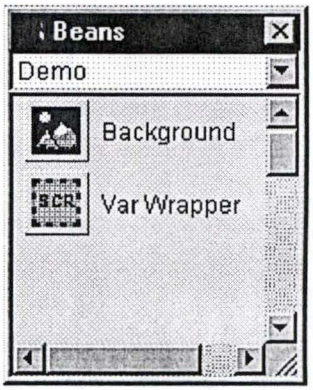


Figure 33. Fenêtre de sélection des Beans.

5. Historique

Cette fenêtre montre simplement les messages tels qu'ils sont échangés entre l'outil et le simulateur. C'est essentiellement un outil de correction. On peut effacer la page en pressant le bouton "Clear Area".

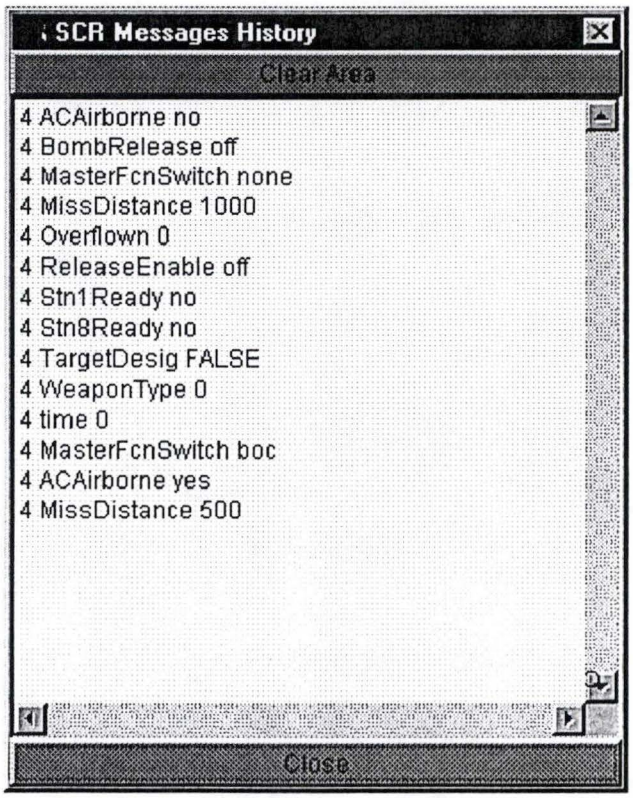


Figure 34. Historique des messages.

6. La fenêtre d'édition

Cette fenêtre est le cœur de la manipulation des Beans. C'est ici qu'ils sont instanciés, manipulés en terme de position et de taille, liés avec des variables. La fenêtre est évidemment éditable quand l'outil est en mode édition. Quand le mode animation est enclenché, tous les objets à l'intérieur de la fenêtre sont figés et non-éditables, et les fenêtres affichants les propriétés disparaissent. A la base, cette fenêtre est blanche. L'exemple ci-dessous montre une fenêtre d'édition avec un fond (l'A-7) et quelques (pseudo-)Beans instanciés (en blanc):

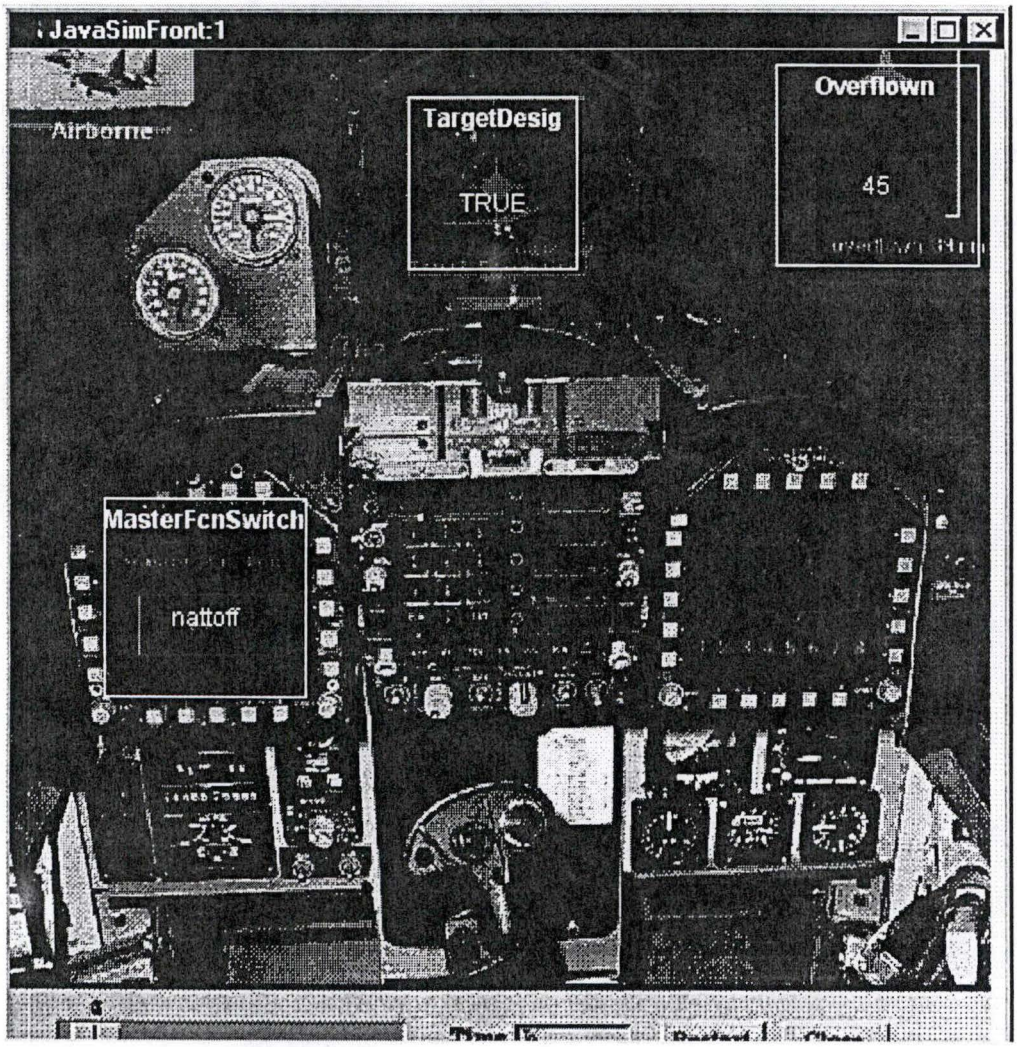


Figure 35. La fenêtre d'édition de l'outil.

Conclusions

L'interface fit plutôt bonne impression aux gens du NRL, ce qui me laisse croire que l'orientation est assez correcte. Par contre, il m'aurait vraiment été agréable de pouvoir fournir à l'équipe du Docteur Heitmeyer un outil vraiment basé sur les Beans. La version précédente de cet outil pouvait instancier des Beans au sein de la fenêtre d'édition et afficher ses propriétés, mais leur manipulation se révélait caduque. J'ose croire qu'avec un peu de temps on pourrait faire de cet outil quelque chose d'utile.

En ce qui concerne les perspectives de ce genre d'approche pour l'animation, elles sont relativement intéressantes. Du point de vue commercial, c'est un pas considérable vers la reconnaissance de ce genre d'outils comme outils d'analyse à part entière. Récemment, [HEITMEYER98] se plaignait encore de l'accueil parfois froid réservé à ce genre d'approche. L'aspect distribué est en ce sens primordiale: le client pouvant tester à distance l'analyse de son système, l'analyste par cela fidélisant le client à son approche.

Enfin, on pourrait envisager aussi d'utiliser les avantages de la programmation distribuée. Prochainement, une version Java de **SCR*** devrait être mise en place. Cette version pourrait très facilement exploiter les facilités de communication offertes par le RMI. Et pourquoi pas inclure directement l'animateur au sein du simulateur.

Durant les quelques pages qui précèdent, nous avons abordé plusieurs aspects de l'ingénierie des besoins. Nous nous sommes baladé du général au particulier, de la définition du projet **SCR** en 1978 à l'utilisation des classes d'introspection sur des Beans, des grands principes d'écriture de cahiers de charge à la création d'agents sous *Albert II*. Incohérent me direz-vous ? Peut-être, peut-être pas. Quoiqu'il en soit, ce que l'on peut retirer de global de ces quelques lignes, c'est que le domaine d'application évolue, l'ingénierie des besoins trouve ses voix. L'approche **SCR,** bien qu'opérationnelle et stricte, rêve d'expressivité et de rencontrer d'autres clients que les ingénieurs auxquels se destinent ses tables. Pour cela, elle s'invente une interface d'animation, histoire de voir bouger le monde. L'approche *Albert II*, tout en expressivité et temporalité trouve sa voix. Bien qu'encore jeune face à sa consœur américaine, elle rêve, elle aussi, de s'animer un peu. Sœurs ennemies ? Que nenni ! Elles peuvent coopérer et toutes deux devenirs les reines du grand bal de la rédaction des cahiers de charge. C'est ce que nous avons pu constater.

Références bibliographiques

[CLEMENTS85] Clement P., *Software Cost Reduction through Disciplined Design*, Computer Science Systems Branch, Naval Research Laboratory, Washington D.C., 1985.

[CLEMENTS87] Clements P., Faulk S., *The NRL Software Cost Reduction (**SCR**) Requirements Specification Methodology*, Computer Science and Systems Branch, Naval Research Laboratory, Washington, D.C., April 1987

[DU_BOIS97] Du Bois P., *The Albert II Reference Manual : Language Constructs and Infromal Semantics (V2.0)*. CEDETI, Facultés Universitaires Notre-Dame de la Paix, Namur, July 1997.

[DUBOIS97] Dubois E., Du Bois P., Zeippen J-M, *On the Use of a Formal Requirements Engineering Language : The Generalized Railroad Crossing Problem*. Proceedings of the Third International Symposium on Requirements Engineering (RE'97), January 1997, Annapolis, MD.

[DUBOIS98a] Dubois E., Cours *Méthodologie de Développement Logiciel, Matière Approfondie*, Facultés Universitaires Notre-Dame de la Paix, Namur - 1998.

[DUBOIS98b] Dubois E., Yu E., Petit M., *From Early to Late Formal Requirements : a Process-Control Case Study*. CREWS Report Series 98-15, In Proceedings of IWSSD98, April 1998 Isobe, Japan.

[FAULK92] Faulk S., Brackett J., Ward P., and Kirby J., *The CoRE Method for Real-Time Requirements*, IEEE Software, Vol.9, N°.5, September 1992.

[FAULK95] Faulk S., *Software Requirements : A Tutorial*, Center for High Assurance Computer Systems, Information Technology Division, NRL/MR/5546-95-7775, Naval Research Laboratory, November 1995.

[GAO92] U.S. General Accounting Office, *Mission Critical Systems : Defense Attempting to Address Major Software Challenges*, GAO/IMTEC-93-13, December 1992.

[HEITMEYER93] Heitmeyer C. and Labaw B., *Consistency Checks for **SCR**-Style Requirements Specifications*. Technical Report 9586, NRL, Washington DC, December 1993.

[HEITMEYER95a] Heitmeyer C., Labaw B., and Kiskis D., *Consistency Checking of*

***SCR**-Style Requirements Specifications*, in Proceedings, IEEE International Symposium on Requirements Engineering, March 1995.

[HEITMEYER95b] Heitmeyer C., Jeffords R. and Labaw B., *Tools for Analyzing **SCR**-Style Requirements Specifications : A Formal Foundation*, NRL Technical Report NRL-7499, U.S. Naval Research Laboratory, Washington, DC, 1995.

[HEITMEYER95c] Heitmeyer C., Bull A., Gasarch C., Labaw B., ***SCR*** : A Toolset for Specifiying and Analyzing Requirements*, Proceedings of the 10th IEEE Conference on Computer Assurance (COMPASS), Gaithersburg, MD, June 1995.

[HEITMEYER96a] Heitmeyer C., *Requirements Specifications for Hybrid Systems*, in Proceedings of Hybrid Systems Workshop III, Lecture Notes in Computer Science, Naval Research Laboratory, Washington, DC, 1996.

[HEITMEYER96b] Heitmeyer C., Jeffords R., Labaw B., *Automated Consistency Checking of Requirements Specifications*. ACM Transaction on Software Engeneering and Methodology, 5(3):231-261, April-June 1996.

[HEITMEYER96c] Heitmeyer C., Bharadwaj R., *Applying the **SCR** Requirement Specification Method to Practical Systems : A Case Study*, presented at The 21st Software Engineering Workshop, NASA Goddard Space Flight Center, Greenbelt MD, USA, December 1996.

[HEITMEYER97a] Heitmeyer C., Bharadwaj R., *Verifing **SCR** Requirements Specifications Using State Exploration*, In Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software, January 1997.

[HEITMEYER97b] Heitmeyer C., Kirby J., Labaw B., *Tools for Formal Specification, Verification, and Validation of Requirements*, to appear in COMPASS'97, Naval Research Laboratory, Washington DC, 1997.

[HEITMEYER98] Heitmeyer C., Kirby J., Labaw B., *Applying Formal Methods to Practical Systems : An Experience Report*, submittal to 2nd Workshop on Formal Method in Software Practice, Naval Research Laboratory, Washington DC, 1998.

[HEYMANS97a] Heymans P., Dubois E., *Some Thoughts about the Animation of Formal Specifications Written in the Albert II Language*. Computer Science Department, University of Namur, Belgium, January 1997.

[HEYMANS97b] Heymans P., *The Albert II Specification Animator*. Computer Science Department, University of Namur, Belgium, August 1997.

[IM96] Informatique magazine n°23 "*Intranet: pourquoi il va s'imposer*", décembre 1996.

[KIRBY97] Kirby J., **SCR*** *Toolset : The User Guide*, NRL Internal Notes, January 1997.

[MAX97] Vanderburg G., *Maximum Java 1.1*, ed. SamsNet, 1997

[NASA95] National Aeronautics and Space Administration (NASA), Office of Safety and Mission Assurance, *Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I : Planning and Technology Insertion.* NASA-GB-002-95 Release 1.0, July 95.

[NEUMANN96] Peter G. Neumann, *Using Formal Methods to Reduce Risks*, in Communications of the ACM, Vol.39, N° 7, July 1996.

[NUTSHELL97] Flanagan D., *Java in a Nutshell : a desktop quick reference, second edition.* O'REILLY, 1997.

[PARNAS78] Parnas D., Heninger K., Kallander J. and Shore J., *Software Requirements for the A-7E Aircraft.* Technical Report NRL-3876, Naval Research Laboratory, Washington, DC, 1978.

[PARNAS91] Parnas D., and Madey J., *Functional Documentation for Computer Systems Engineering (Version 2)*, CRL Report N°. 237, McMaster University, Hamilton, Ontario, Canada, Septembre 1991.

[POHL96] Pohl K., *Process Centered Requirements Engineering.* John Wiley, 1996.

[SECMM95] Carnegie Mellon University, SEI, *A Systems Engineering Capability Maturity Model, Version 1.1*, CMU/SEI-95-MM-003, November 1995.

[SUN97] Graham Hamilton, *Sun Microsystems JavaBeans™*, Versions 1.01, Sun Microsystems, July 24, 1997.

[VANHELSUWE97] Vanhelsuwé L., *Mastering JavaBeans*, Sybex, 1997.

[ZOWGHI95] Zowghi D., *Requirement Engeniering : Scope*, http://www-comp.mpce.mq.edu.au/~didar/seweb/scope.html .

Annexes

Annexe 1
Le Problème du Monte-Charge

Introduction

Afin d'illustrer le contenu théorique de l'exposé, s'est posé le problème du choix d'un système à spécifier. Il fallait que le problème ne soit pas trop grand, mais qu'il soit cependant représentatif de manière à couvrir un ensemble correct des difficultés que l'on peut rencontrer lorsque l'on spécifie un système. Afin de ne pas refaire ce qui avait déjà été fait ailleurs, il fallait aussi être original. Pour cela, a été retenu un système bien connu des gens de l'Institut d'Informatique : le monte-charge, qui dessert une partie des étages du bâtiment. Or, ce système a un comportement qui lui est propre. Le problème présenté ici est évidemment une version simplifiée du véritable monte-charge de l'Institut, certains aspects comme la présence d'une alarme ou encore la gestion de la surcharge ayant été mis de côté afin de ne pas alourdir l'illustration; et d'autres aspects ont été modifiés afin de donner au système un comportement qui va nécessiter un certain effort de spécification. Pour ces mêmes raisons, le texte est volontairement obscur sur certains points (présence de capteurs et d'effecteurs, éléments à représenter absolument,...). Cela s'explique par le désir de pouvoir analyser les apports des deux langages sur ces points précis. Voici donc l'exemple retenu.

Exposé du problème

Le problème du monte-charge peut s'exprimer comme suit :

L'objectif est de réaliser un monte-charge (MC) desservant les étages -1 à 3 de l'Institut d'Informatique. Les utilisateurs peuvent appeler le monte-charge à chaque étage. Ils peuvent aussi choisir une destination au MC à partir de l'intérieur du MC. Quand le MC arrive à un étage où il doit s'arrêter, ses portes s'ouvrent pour une période de 5 secondes. Après quoi, celles-ci se referment. Si lors de la fermeture des portes, quelque chose bloque celles-ci, elles s'ouvrent à nouveau.

Le MC ne se déplace que lorsque les portes sont fermées. Lorsque plusieurs destinations sont choisies à la suite, le MC ira vers l'étage le plus proche dans le sens correspondant au sens du service précédent (Par exemple, si le MC est à l'étage 1, et qu'on lui demande d'aller en -1, 2 et 3, et que le MC vient du deuxième étage, il ira en -1.). Sinon, il ira à l'étage le plus proche. La règle est la même pour les appels. Si le MC passe devant un étage où un utilisateur a pressé le bouton d'appel, le MC ignore l'appel s'il monte, et s'arrête à cet étage s'il descend. Enfin, après chaque arrêt, tous les utilisateurs non encore servis doivent refaire leur sélection (appels et destinations).♦

Annexe 2

Notation SCR : Description et syntaxe[6]

Description des dictionnaires

Dictionnaire des variables

Champs	Description	Obligatoire	Exemples
Nom	Nom unique pour les variables ou les expressions issues de l'alphabet [A-Za-z0-9_]. Le nom ne peut commencer avec un chiffre et est *case sensitive*.	oui	mACAirbone tDesireSpeed mEtageAppel
Type	Types de base acceptés : Integer, Boolean, Real. Types définis dans le dictionnaire des types acceptés.	oui	Boolean yRPM
Valeur initiale	Valeur initiale de la variable	oui	FALSE 0
Précision	N/A ou pour les réels et les entiers, une valeur légale de la variable.	oui	N/A 10
Commentaires	-	non	mACAirbone = TRUE ssi l'avion est posé.

Dictionnaire des types

Champs	Description	Obligatoire	Exemples
Nom	Nom unique pour le type issu de l'alphabet [A-Za-z0-9_]. Le nom ne peut commencer avec un chiffre et est *case sensitive*.	oui	ySpeed yValve
Type de base	Types de base acceptés: Integer, Boolean, Real, Enumerated Types définis dans le dictionnaire des types acceptés.	oui	Real Enumerated
Unités	Unités dans lesquelles les constantes ou les variables peuvent être exprimées. Entrez N/A dans les cas où ça ne s'applique pas.	oui	miles/hour N/A
Valeurs légales	Description des valeurs acceptées par une variable de ce type. Pour les types énumérés, une liste de valeurs symboliques séparées par une virgule. Pour les valeurs numériques, les bornes supérieures et inférieures de l'ensemble des valeurs acceptées entre crochets et séparés par une virgule.	oui	[0.0, 100.0] open, closed
Commentaires	-	non	

Dictionnaire des constantes

Champs	Description	Obligatoire	Exemples
Nom	Nom unique pour les variables ou les expressions issues de l'alphabet [A-Za-z0-9_]. Le nom ne peut commencer avec un chiffre et est *case sensitive*.	oui	kTolerance
Type	Types de base acceptés: Integer, Boolean, Real. Types définis dans le dictionnaire des types acceptés.	oui	yTemperature
Valeur	Valeur constante légale de la variable	oui	10
Commentaires	-	non	

Dictionnaire des modes

Champs	Description	Obligatoire	Exemple
Nom	Nom unique pour les modes issus de l'alphabet [A-Za-z0-9_]. Le nom ne peut commencer avec un chiffre et est *case sensitive*.	oui	smFurnaceOperation smWeapon
Modes	Liste des modes séparés par des virgules. Chaque nom de mode doit être unique et issu de l'alphabet [A-Za-z0-9_]. Le nom ne peut commencer avec un chiffre et est *case sensitive*.	oui	sStandby,SIgnition,sShutDown,srunning,sNattack,sCCIP,sNone
Mode initial	Etat initial de la classe des modes	oui	sRunning
Commentaires	-	non	

[6] Les définitions suivantes sont issues de [KIRBY97]

Dictionnaire des assertions

Champs	Description	Obligatoire	Exemples
Nom	Nom unique pour les assertions issues de l'alphabet [A-Za-z0-9_]. Le nom ne peut commencer avec un chiffre et est *case sensitive*.	oui	aReleaseOK
Expression	Les expressions sont généralement des conditions booléennes. Elles décrivent des invariants qui doivent être respectés pour toute exécution des spécifications. Un variable avec un prime (e.g., *mReleaseEnable'*) correspond à la valeur de la variable dans son nouvel état. Si au moins une variable a un prime, et que le prime est absent sur les autres variables, elles correspondent aux valeurs de ces variables à l'état précédent. Si aucune variable n'a de prime, elles réfèrent toutes à leurs valeurs dans leur nouvel état.	oui	cBombRelease = on => mReleaseEnable = pressed cBombRelease' = on => mReleaseEnable = pressed
Commentaires	-	non	

Description des tables fonctionnelles

Table des événements

Champs	Description	Editable	Exemples
Nom	Le nom de la variable dont l'événement spécifie la valeur.	oui	cBombRelease tReadyStn
Type de table	Le type de table	non	Event
Classe	La classe auquel la variable spécifiée appartient	non	Controlled Term Unknown
Classe des modes	Le nom de la classe	oui	smWeapon
Modes	Une liste des modes de la classe séparés par des virgules	oui	sNattack sCCIP,BOC,None
Evénements	Une expression définissant l'événement pour lequel les valeurs de la variable devraient changer.	oui	@T(mReleaeEnable) WHEN NOT tShrike @F(mACAirborne)
Valeur	Une expression définissant la valeur que la variable devrait recevoir	oui	TRUE mTime-tSolutionTime

Table des conditions

Champs	Description	Editable	Exemples
Nom	Le nom de la variable dont l'événement spécifie la valeur.	oui	cBombRelease tReadyStn
Type de table	Le type de table	non	Condition
Classe	La classe auquel la variable spécifiée appartient	non	Controlled Term Unknown
Classe des modes	Le nom de la classe	oui	smWeapon
Modes	Une liste des modes de la classe séparés par des virgules	oui	sNattack sCCIP,BOC,None
Conditions	Une expression conditionnelle.	oui	mReleaeEnable AND NOT tShrike mMissDistance = 10
Valeur	Une expression définissant la valeur que la variable devrait recevoir	oui	TRUE mTime-tSolutionTime

Table des transitions

Champs	Description	Editable	Exemples
Nom	Le nom de la classe des modes	oui	smMonteCharge
Type de table	Le type de table	non	Mode Transition
Classe	La classe auquel la variable spécifiée appartient	non	Mode Class
Mode Source	Une liste des modes de la classe séparés par des virgules	oui	sMCArrêté
Evénements	Un événement qui définit quand une transition d'un mode vers un autre devrait survenir.	oui	@T(mReleaeEnable) WHEN NOT tShrike @F(mACAirborne)
Mode Destination	Un des modes de la classe	oui	sMCMontant

Description de la syntaxe des expressions

expression ::=	full_ifandonlyif | o_envent | ε
full_ifandonlyif ::=	full_implication ifandonlyif
ifandonlyif ::=	IFF full_implication ifandonlyif | ε
full_implication ::=	o_cond implication
implication ::=	IMPLIES o_cond implicaiotn | ε
o_event ::=	a-event | o_event OR a_event
a_event ::=	simple_event | a_event AND simple_event
simple_event ::=	event | cond_event | not_event | LPAREN o_event close_expression | ALWAYS | NEVER
o_cond ::=	a_cond | o_cond OR a_cond
a_cond ::=	simple_cond | a_cond AND simple_cond
simple_cond ::=	not_cond | relat_exp | inmode_exp | arb_exp
not_cond ::=	NOT literal | NOT variable | NOT inmode_exp | NOT LPAREN full_ifandonlyif close_expression | NOT LPAREN not_cond close_expression
not_event ::=	NOT event | NOT cond_event | NOT LPAREN not_event close_expression | NOT LPAREN event close_expression | NOT LPAREN cond_event close_expression
relat_exp ::=	arb_exp RELOP arb_exp
arb_exp ::=	arith_exp | u_arith_exp | literal | LPAREN full_ifandonlyif close_expression
arith_exp ::=	term | arith_exp MINUS term | arith_exp PLUS term
u_arith_exp ::=	u_term | u_arith_exp MINUS term | u_arith_exp PLUS term
term ::=	num_literal | variable | term MULT num_literal | term DIV num_literal | term MULT variable | term DIV variable
u_term ::=	unary_term | u_term MULT num_literal | u_term DIV num_literal | u_term MULT variable | u_term DIV variable
unary_term ::=	MINUS num_literal | MINUS variable | PLUS num_literal | PLUS variable
cond_event ::=	event WHEN ful_ifandonlyif
event ::=	AT_TRUE LPAREN full_ifandonlyif vlose_expression | AT_FALSE LPAREN full_ifandonlyif close_expression
inmode_exp ::=	INMODE

literal ::=	STRING | CHARACTER | BOOL_TRUE | BOOL_FALSE
num_literal ::=	INTEGER | FLOAT
variable ::=	NAME | NAME PRIME | LPAREN arith_exp close_expression | LPAREN u_arith_exp close_expression
close_expression ::=	RPAREN | RPAREN PRIME
RPAREN ::=	")"
PRIME ::=	"'"
LPAREN ::=	"("
INTEGER ::=	[0-9]+
FLOAT ::=	[0-9]* "." [0-9]+
STRING ::=	\"[A-Za-z0-9!@#$%^&*()_+|~'\\=-[\]{};':,./<>?]*\"
CHARACTER ::=	[A-Za-z0-9!@#$%^&*()_+|~'\\=-[\]{};':,./<>?]
BOOL_TRUE ::=	[Tt][Rr][Uu][Ee]
BOOL_FALSE ::=	[Ff][Aa][Ll][Ss][Ee]
INMODE ::=	[Ii][Nn][Mm][Oo][Dd][Ee]
AT_TRUE ::=	"@T"
AT_FALSE ::=	"@F"
WHEN ::=	[Ww][Hh][Ee][Nn]
MINUS ::=	"-"
PLUS ::=	"+"
MULT ::=	"*"
DIV ::=	"/"
RELOP ::=	"=" | "!=" | "<" | "<=" | ">" | ">="
NOT ::=	[Nn][Oo][Tt]
AND ::=	[Aa][Nn][Dd]
OR ::=	[Oo][Rr]
ALWAYS ::=	[Aa][Ll][Ww][Aa][Yy][Ss]
NEVER ::=	[Nn][Ee][Vv][Ee][Rr]
IMPLIES ::=	"=>"
IFF ::=	"<=>"

Annexe 3
Spécifications SCR du Monte-Charge

Dictionnaires:

Dictionnaire des variables

Nom	Type	Valeur initiale	Précision	Commentaires
mEtageCourant	yEtage	0	N/A	Etage actuel du MC.
mEtageAppel	yEtage	0	N/A	Etage où un bouton d'appel a été pressé.
mEtageDest	yEtage	0	N/A	Etage dont le bouton de sélection a été pressé.
mPorte	yEtatPorte	Fermée	N/A	Etat de la porte devant laquelle se trouve le MC.
cArrêtMC	Switch	Off	N/A	Le moteur arrête le MC.
cMontéeMC	Switch	Off	N/A	Le moteur monte le MC.
cDescenteMC	Switch	Off	N/A	Le moteur descend le MC.
cOuverturePorte	Switch	Off	N/A	Le moteur ferme la porte.
cFermeturePorte	Switch	Off	N/A	Le moteur ouvre la porte.
tOuvertureAutorisée	Boolean	True	N/A	Le moteur de la porte est autorisé à ouvrir la porte quand cette variable est True
tFermetureAutorisée	Boolean	False	N/A	Le moteur de la porte est autorisé à fermer la porte quand cette variable est True
tDestination	yEtage	0	N/A	Prochain arrêt du MC.
tDirection	yDir	Stop	N/A	Direction prise par le MC.
tDerDir	yDerDir	Haut	N/A	Dernière direction prise par le MC
tDirLocked	Boolean	False	N/A	Retient si un étage destination se trouvant dans le sens précédent au MC a été proposé.
tPlusProche	Boolean	False	N/A	L'appel ou la destination soumise n'est pas plus proche que la destination du MC
tArriveEtage	Boolean	True	N/A	Indique si le MC est exactement devant une porte.

Table 14. Dictionnaire des variables du MC.

Dictionnaire des types

Nom	Type de base	Unités	Valeurs légales	Commentaires
yEtage	Integer	N/A	[-1,3]	Représentation des étages du bâtiment.
yEtatPorte	Enumerated	N/A	Ouvert,Fermé,Bloquée	La porte peut être signalée Ouverte, Fermée ou Bloquée.
yDir	Enumerated	N/A	Haut,Stop,Bas	Directions possibles du MC.
yDerDir	Enumarated	N/A	Haut,Bas	Dernirères directions possibles pour le MC

Table 15. Dictionnaire des types du MC.

Dictionnaire des constantes

Nom	Type	Valeur	Commentaires
kAttentePorte	Integer	5000	Le temps d'ouverture de la porte du MC est fixé à 5 secondes (5000 ms).

Table 16. Dictionnaire des constantes du MC.

Dictionnaire des modes

Nom	Modes	Mode initiale	Commentaires
mcMonteCharge	MCMontant,MCDescendant,MCArrêté.	MCArrêté	Le MC peut soit être en train de monter, de descendre ou être à l'arrêt.
mcPorteMC	Fermée,Ouverture,Ouverte,Fermeture	Fermée	La porte du MC peut soit être fermée, s'ouvrir, être ouverte ou se fermer.

Table 17. Dictionnaire des modes du MC.

Table des transitions :

mcMonteCharge

Mode Source	Evénement	Mode Destination
MCArrêté	@T(tDestination>mEtageCourant) WHEN mPorte=Fermée	MCMontant
MCArrêté	@T(tDestination<mEtageCourant) WHEN mPorte=Fermée	MCDescendant
MCMontant, MCDescendant	@T(tDestination=mEtageCourant) ∧ tArriveEtage	MCArrêté

Ligne 1: Si le MC est arrêté et que l'étage de destination attribué au MC est haut dessus de l'étage actuel du MC et que la porte est fermée, alors le MC monte.
Ligne 2: Si le MC est arrêté et que l'étage de destination attribué au MC est en dessous de l'étage actuel du MC et que la porte est fermée, alors le MC descend.
Ligne 3: Si le MC monte ou descend et que l'étage de destination attribué au MC est le même que l'étage actuel du MC, alors le MC s'arrête.

mcPorteMonteCharge

Mode Source	Evénement	Mode Destination
Fermée	@T(tOuvertureAutorisée)	Ouverture
Ouverture	@T(mPorte=Ouvert)	Ouverte
Ouverte	@T(tFermetureAutorisée)	Fermeture
Fermeture	@T(mPorte=Bloquée) WHEN tOuvertureAutorisée	Ouverture
Fermeture	@T(mPorte=Fermé)	Fermée

Ligne 1: Si la porte est fermée et que son ouverture est autorisée, alors la porte s'ouvre.
Ligne 2: Si la porte est en train de s'ouvrir, et que les battants de la porte sont complètement ouverts, alors la porte est ouverte.
Ligne 3: Si la porte est ouverte et que sa fermeture est autorisée, alors la porte se ferme.
Ligne 4: Si la porte est en train de se fermer et que les batants se bloquent alors que l'ouverture de la porte est autorisée, la porte s'ouvre.
Ligne 5: Si la porte se ferme et que le battants se touchent, alors la porte est fermée.

Table des événements :

tDirection

Mode	Evénements		
MCArrêté	@T(mPorte=Fermé) WHEN (tDestination>mEtageCourant)	CHANGED(tDestination) WHEN NOT (mPorte=Fermé)	@T(mPorte=Fermé) WHEN (tDestination< mEtageCourant)
MCMontant	@T(mEtageCourant < tDestination)	@T(tDestination=mEtageCourant) WHEN tArriveEtage	NEVER
MCDescendant	NEVER	@T(tDestination=mEtageCourant) WHEN tArriveEtage	@T(mEtageCourant > tDestination)
tDirection	Haut	Stop	Bas

Case 1, 1: Si le MC est à l'arrêt et que la porte est fermée quand une destination assignée au MC est située plus haut que l'étage actuel du MC, alors la direction prise par le MC est le haut.
Case 1, 2: Si le MC est à l'arrêt et que la destination assignée au MC a changé alors que la porte n'est pas fermée, alors le MC ne bouge pas.
Case 1, 3: Si le MC est à l'arrêt et que la porte est fermée quand une destination assignée au MC est située plus bas que l'étage actuel du MC, alors la direction prise par le MC est le bas.
Case 2, 1: Si le MC monte et que le MC est situé plus bas que la destination du MC, alors la direction est le haut.
Case 2, 2: Si le MC monte et que le MC arrive à la destination assignée, alors le MC se stoppe.
Case 2, 3: Aucun événement ne peut donner la direction du bas au MC lorsque celui-ci monte.
Case 3, 1: Aucun événement ne peut donner la direction du haut au MC lorsque celui-ci descend.
Case 3, 2: Si le MC descend et que le MC arrive à la destination assignée, alors le MC se stoppe.
Case 3, 3: Si le MC descend et que le MC est situé plus bas que la destination du MC, alors la direction est le bas.

tDestination

Mode	Evénements	
MCArrêté	CHANGED(mEtageDest) WHEN ((tDirLocked AND tPlusProche) OR (NOT(tDirLocked) AND tPlusProche))	CHANGED(mEtageAppel) WHEN ((tDirLocked AND tPlusProche) OR (NOT(tDirLocked) AND tPlusProche))
MCMontant	@T(mEtageDest<tDestination) WHEN (tDerDir = Haut)	NEVER
MCDescendant	@T(mEtageDest>tDestination) WHEN (tDerDir = Bas)	@T(mEtageAppel<tDestination) WHEN (tDerDir = Bas)
tDestination	mEtageDest	mEtageAppel

Case 1,1: Si le MC est à l'arrêt et que l'étage de destination proposé a changé alors qu'une assignation précédente allant dans la précédente direction du MC et que cette proposition est plus proche que la destination assignée actuelle, ou alors qu'il n'y a pas encore eut de nouvelle destination allant de le sens précédent au MC et que celle-ci est plus proche que l'étage assigné précédent, le destination devient l'étage destination demandé.
Case 1,2: Si le MC est à l'arrêt et qu'un appel du MC survient alors qu'une assignation précédente allant dans la précédente direction du MC et que cette proposition est plus proche que la destination assignée actuelle, ou alors qu'il n'y a pas encore eut de nouvelle destination allant de le sens précédent au MC et que celle-ci est plus proche que l'étage assigné précédent, le destination devient l'étage destination demandé.
Case 2,1: Si le MC monte et que l'étage destination proposé est plus bas que la destination actuelle alors que la dernière direction prise par le MC est le haut, la nouvelle destination est l'étage proposé.
Case 2,2: Quand le MC monte, aucun étage où un appel a été fait ne peut être assigné comme nouvelle destination du MC.
Case 3,1: Si le MC descend et que l'étage destination proposé est plus haut que la destination actuelle alors que la dernière direction prise par le MC est le bas, la nouvelle destination est l'étage proposé.
Case 3,2: Si le MC descend et qu'un appel au MC est plus bas que la destination actuelle alors que la dernière direction prise par le MC est le bas, la nouvelle destination est l'étage d'appel.

tDirLocked

Mode	Evénements	
MCArrêté	@T(mEtageDest>mEtageCourant OR mEtageAppel>mEtageCourant) WHEN TDerDir=Haut OR @T(mEtageDest<mEtageCourant OR mEtageAppel<mEtageCourant) WHEN tDerDir=Bas	@T(Inmode)
MCMontant, MCDescendant	@T(Inmode)	NEVER
tDirLocked	True	False

Case 1,1: Si le MC est arrêté, toute demande de service (appel ou destination) allant dans le sens de la dernière direction prise par le MC, rend tDirLocked vraie.
Case 1,2: Dès que le MC s'arrête, tDirLocked est faux.
Case 2,1: Dès que le MC monte ou descend, tDirLocked est vraie.
Case 2,2: Durant la montée ou la descente du MC, aucun événement ne peut changer la valeur de tDirLocked en faux.

tPlusProche

Mode	Evénements	
MCArrêté, MCMontant, MCDescendant	@T(mEtageDest>=mEtageCourant OR mEtageAppel>=mEtageCourant) WHEN (mEtageDest<=tDestination OR mEtageAppel<=tDestination) OR @T(mEtageDest<=mEtageCourant OR mEtageAppel<=mEtageCourant) WHEN (mEtageDest>=tDestination OR mEtageAppel>=tDestination)	@T(mEtageDest>mEtageCourant OR mEtageAppel>mEtageCourant) WHEN (mEtageDest>tDestination OR mEtageAppel>tDestination) OR @T(mEtageDest<mEtageCourant OR mEtageAppel<mEtageCourant) WHEN (mEtageDest<tDestination OR mEtageAppel<tDestination)
tPlusProche	True	False

Case 1,1: Quelque soit l'état du MC, toute demande de service (appel ou destination) étant plus proche que la destination assignée au MC, rend tPlusProche vraie.
Case 1,2: Dans les autres cas, tPlusProche est faux.

tDerDir

Mode	Evénements	
MCArrêté	NEVER	NEVER
MCMontant	NEVER	@T(Inmode)
MCDescendant	@T(Inmode)	NEVER
tDerDir	Bas	Haut

Global: la dernère direction prise par le MC est la direction Bas lorsque le MC commence à descendre, ou Haut lorsque le MC commence à monter.

tArriveEtage

Mode	Evénements	
MCArrêté	@T(Inmode)	@F(Inmode)
MCMontant, MCDescendant	CHANGED (mEtageCourant)	@T(Inmode)
tArriveEtage	True	False

Global: Est vrai dès que le MC est en face d'une porte. Faux dans les autres cas.

tOuvertureAutorisée

Mode	Evénements	
Fermée	@T(mEtageCourant=mEtageAppel) WHEN (tDirection=Stop) OR @T(tDestination=mEtageCourant) WHEN tArriveEtage	@T(Inmode)
Ouverture	@T(Inmode)	@T(mPorte=Ouvert)
Ouverte	NEVER	@T(Inmode)
Fermeture	@T(mPorte=Bloqué)	@T(Inmode)
tOuvertureAutorisée	true	false

Ligne 1: Dès que la porte est fermée, son ouverture n'est plus autorisée, mais si la porte devinet fermée alors que le MC est arrêté ou qu'un appel correspondant à l'étage actuel du MC survient (quelqu'un presse le bouton d'appel alors que le MC est en attente portes fermées à ce même étage), alors l'ouverutre de laporte est autorisée.
Ligne 2: Dès que la porte commence à s'ouvrir son ouverture est autorisée, et quand la porte est complètement ouverte, l'autorisation disparaît.
Ligne 3: L'ouverture n'est jamais autorisée quand la porte est ouverte.
Ligne 4: L'ouverture de la porte n'est jamais autorisée quand le porte se ferme, sauf lorsque la porte se bloque.

tFermetureAutorisée

Mode	Evénements	
Fermée,Ouverture	NEVER	@T(Inmode)
Ouverte	@T(Duration(mPorte=Ouvert)=kAttentePorte	@T(Duration(mPorte=Ouvert) <kAttentePorte
Fermeture	@T(Inmode)	@T(mPorte=Bloqué) OR @T(mPorte=Fermée)
tFermetureAutorisée	true	false

Ligne 1: Lors de l'ouverture de la porte ou lorsqu'elle est fermée, la fermeture n'est jamais autorisée.
Ligne 2: lorsque la porte est ouverte, sa fermeture n'est autorisée qu'après un temps défini kAttentePorte.
Ligne 3: Durant la fermeture de la porte, sa fermeture est autorisée sauf lorsqu'elle est bloquée ou lorsqu'elle est complètement fermée.

Tables des conditions :

cArrêtMC

Mode	Conditions	
MCArrêté	True	False
MCMontant, MCDescendant	tDirection = Stop	NOT (tDirection = Stop)
cArrêtMC	On	Off

Colonne 1: Si le MC est arrêté ou que le MC monte ou descend et que la direction assignée au MC est le stop, la commande cArrêtMC est toujours On.
Colonne 2: Si la direction assignée au MC n'est pas le stop quand le MC se déplace, cArrêtMC est toujours Off.

cMontéeMC

Mode	Conditions	
MCArrêté, MCMontant	tDirection = Haut	NOT (tDirection = Haut)
MCDescendant	False	True
cMontéeMC	On	Off

Colonne 1: Si le MC est arrêté ou monte est que la direction assignée est le haut, la commande de montée passe à On.
Colonne 2: Si le MC monte ou est à l'arrêt et que le direction assignée n'est pas le haut, ou que le MC descend, la commande de montée est à Off.

cDescenteMC

Mode	Conditions	
MCArrêté, MCDescendant	tDirection = Bas	NOT (tDirection =Bas)
MCMontant,	False	True
cDescenteMC	On	Off

Colonne 1: Si le MC est arrêté ou descend est que la direction assignée est le bas, la commande de descente passe à On.
Colonne 2: Si le MC descend ou est à l'arrêt et que le direction assignée n'est pas le bas, ou que le MC descend, la commande de montée est à Off.

cOuverturePorte

Mode	Conditions	
MCArrêté	tOuvertureAutorisée	NOT tOuvertureAutorisée
MCMontant, MCDescendant	False	True
cOuverturePorte	On	Off

Colonne 1: Si le MC est arrêté et que l'ouverture de la porte est autorisée, la commande d'ouverture de porte passe à On.
Colonne 2: Si le MC est arrêté et que l'ouverture de la porte n'est pas autorisée, ou que le MC se déplace, la commande d'ouverture de la porte est à Off.

cFermeturePorte

Mode	Conditions	
MCArrêté	tFermetureAutorisée	NOT tFermetureAutorisée
MCMontant, MCDescendant	False	True
cFermeturePorte	On	Off

Colonne 1: Si le MC est arrêté et que la fermeture de la porte est autorisée, la commande de fermeture de porte passe à On.
Colonne 2: Si le MC est arrêté et que la fermeture de la porte n'est pas autorisée, ou que le MC se déplace, la commande de fermeture de la porte est à Off.

Annexe 4
Spécifications Albert II du Monte-Charge

1. Data types et Operations

Data Types and Operations

BASIC TYPES

CONSTRUCTED TYPES

| Ce type décrit les différents états que peut prendre la porte du monte-charge. Ainsi une porte peut être ouverte, fermée, bloquée, en train de s'ouvrir ou en train de se fermer. Ces états sont exclusifs.

TEtatPorte = ENUM[*Ouvert, se_Ferme, Fermee, s_Ouvre, Bloquee*]

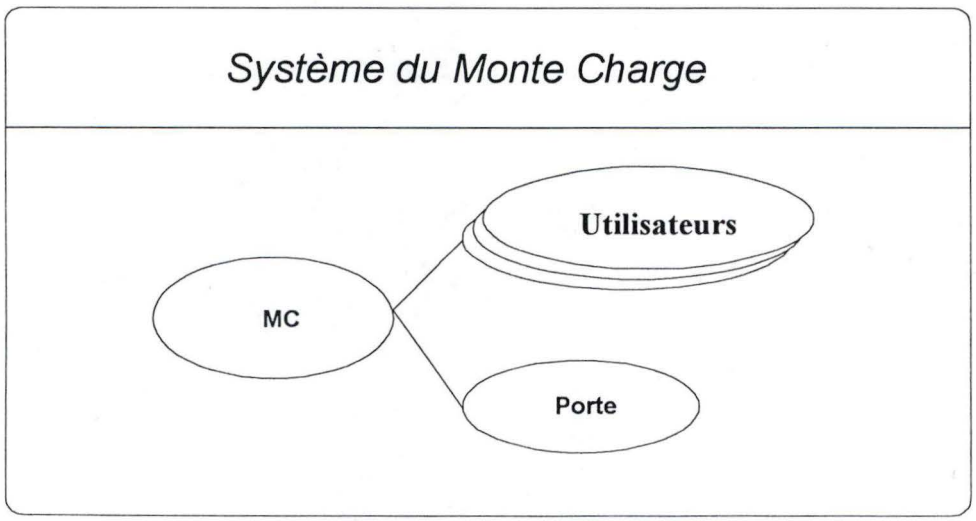
| Ce type définit les différents états dans lequel le monte-charge est. Ainsi, le monte-charge peut monter, descendre ou être à l'arrêt. Ces états sont exclusifs.

TEtatMC = ENUM[*Montant, Descendant, Arrete*]

| Ce type définit les différents états des dernières directions prises par le MC.

TDerdir = ENUM[*Haut,Bas*]

2. Structure de la société Système du Monte Charge



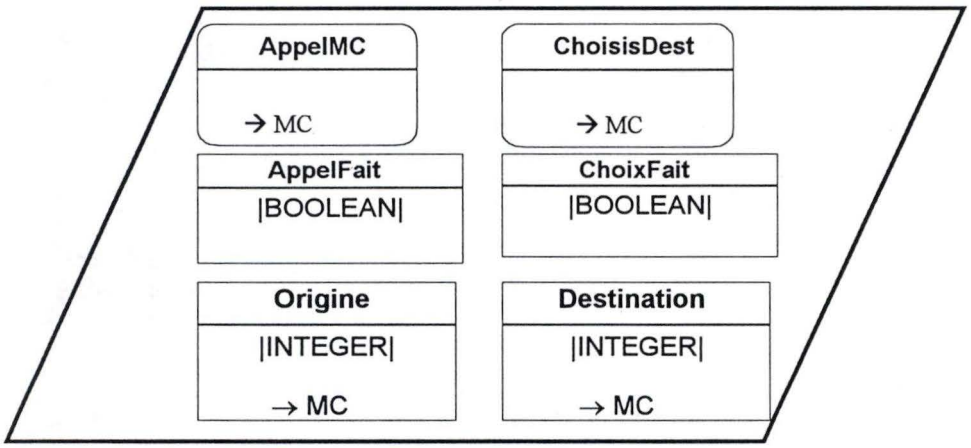
Société : **Système du Monte Charge**

| Système du Monte Charge est composé d'un monte-charge (MC), d'une porte, et d'utilisateurs désirant utiliser les services du MC.

(*MC*)
(*Porte*)
(*Utilisateurs*)

3. Les agents de la société système du Monte Charge

3.1. L'agent Utilisateurs



Agent : **Utilisateurs**

DECLARATION

STATE COMPONENTS

```
| Représente l'étage de destination de l'utilisateur.
```
Destination instance-of *Integer*
```
| Représente l'étage d'où l'utilisateur veut prendre le MC.
```
Origine instance-of *Integer*
```
| Est vrai lorsque l'utilisateur à appeler le MC.
```
AppelFait instance-of *Boolean*
```
| Est vrai lorsque l'utilisateur à fait son choix de destination.
```
ChoixFait instance-of *Boolean*

BASIC CONSTRAINTS

INITIAL VALUE

AppelFait = FALSE
ChoixFait = FALSE

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

| La destination doit être comprise entre le niveau -1 et le 3.
[] $-1 \leq Destination \leq 3$
| L'origine doit être comprise entre le niveau -1 et le 3.
[] $-1 \leq Origine \leq 3$

OPERATIONAL CONSTRAINTS

EFFECTS OF ACTIONS

| Est vrai lorsque, l'appel est considéré comme réalisé, puis il passe à faux.

AppelMC : *AppelFait* := TRUE
[] (1.1)
AppelFait := FALSE

| Est vrai lorsque l'appel est considéré comme réalisé, puis il passe à faux.

ChoisiDest : *ChoixFait* := TRUE
[] (1.2)
ChoixFait := FALSE

COOPERATION CONSTRAINTS

ACTION INFORMATION

| Lorsque l'utilisateur fait un appel, le MC connaît la volonté d'un utilisateur d'aller à un étage donné.

*XX(AppelMC.MC /*AppelFait*)* (1.3)

| Lorsque l'utilisateur choisit une destination, le MC connaît la volonté d'un utilisateur d'aller à un étage donné.

XX(ChoisiDest.MC / ChoixFait*)* (1.4)

STATE INFORMATION

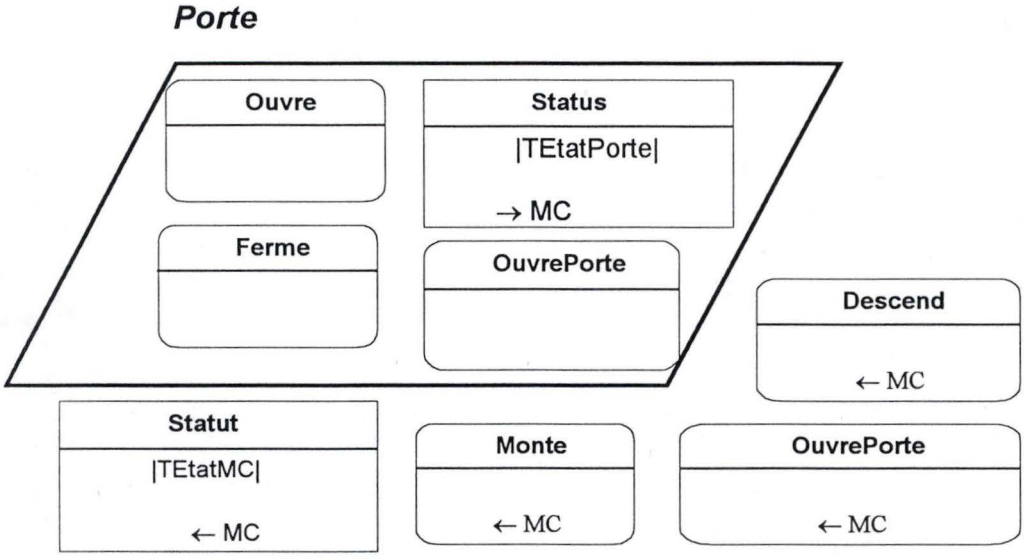
| Lorsque la destination est choisie, le MC connaît l'étage où l'utilisateur veut aller

XX(Destination.MC / ChoixFait*)* (1.5)

| Lorsqu'un appel est fait, le MC connaît l'étage où un utilisateur à fait un appel.

*XX(Origine.MC /*AppelFait*)* (1.6)

3.2. L'agent Porte



Agent : **Porte**

DECLARATION

STATE COMPONENTS

| Cet état définit les différents états que peut prendre la porte du MC.

Statut instance-of *TEtatPorte*

BASIC CONSTRAINTS

INITIAL VALUE

Statut = Fermee

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

| Pendant que le MC monte, la porte est fermée.

[MC.monte] Statut = Fermee *(2.1)*

| Pendant que le MC descend, la porte est fermée.

[MC.descend] Statut = Fermee *(2.2)*

| Quand la porte est ouverte, elle l'est pendant 5 secondes, après elle se ferme.

[]Statut = Ouverte $\Rightarrow$ *(*Futr$_{5''}$ *Statut = se_Ferme)* *(2.3)*

ACTION COMPOSITION

| Quand le MC ouvre la porte, l'action ouvre débute.

OuvrePorte $\leftrightarrow$ *Ouvre* $|\Rightarrow$ *MC.OuvrePorte* *(2.4)*

OPERATIONAL CONSTRAINTS

PRECONDITIONS

```
On ne peut ouvrir les portes que quand le MC est arrêté et que, ou la
porte est fermée, ou qu'elle est bloquée.
```

Ouvre : MC.Statut := Arrete ∧ (Statut = Ferme ∨ Statut = Bloquee) *(2.5)*

```
On ne peut fermer les portes que quand elles sont ouvertes depuis 5
secondes.
```

Ferme : $\text{Lasted}_{5''}$ *(Statut = Ouvert)* *(2.6)*

EFFECTS OF ACTIONS

Ouvre : *Statut := s_Ouvre*
[] *(2.7)*
Statut := Ouvert

Ferme : *Statut := se_Ferme*
[] *(2.8)*
Statut := Fermee

TRIGGERINGS

```
L'action Ouvre est enclenchée si le MC est arrêté et que la porte est
fermée ou bloquée.
```

MC.Statut := Arrete ∧ (Statut = Ferme ∨ Statut = Bloquee)/TRUE ~> *Ouvre* *(2.9)*

```
L'action Ferme est enclenchée si la porte est ouverte depuis au moins 5
secondes.
```

$\text{Lasted}_{5''}$ *(Statut = Ouvert)*/TRUE ~> *Ferme* *(2.10)*

COOPERATION CONSTRAINTS

ACTION PERCEPTION

$\mathcal{XX}$*(MC.Monte /* TRUE*)* *(2.11)*
$\mathcal{XX}$*(MC.Descend /* TRUE*)* *(2.12)*
$\mathcal{XX}$*(MC.OuvrePorte / Statut = Fermee)* *(2.13)*

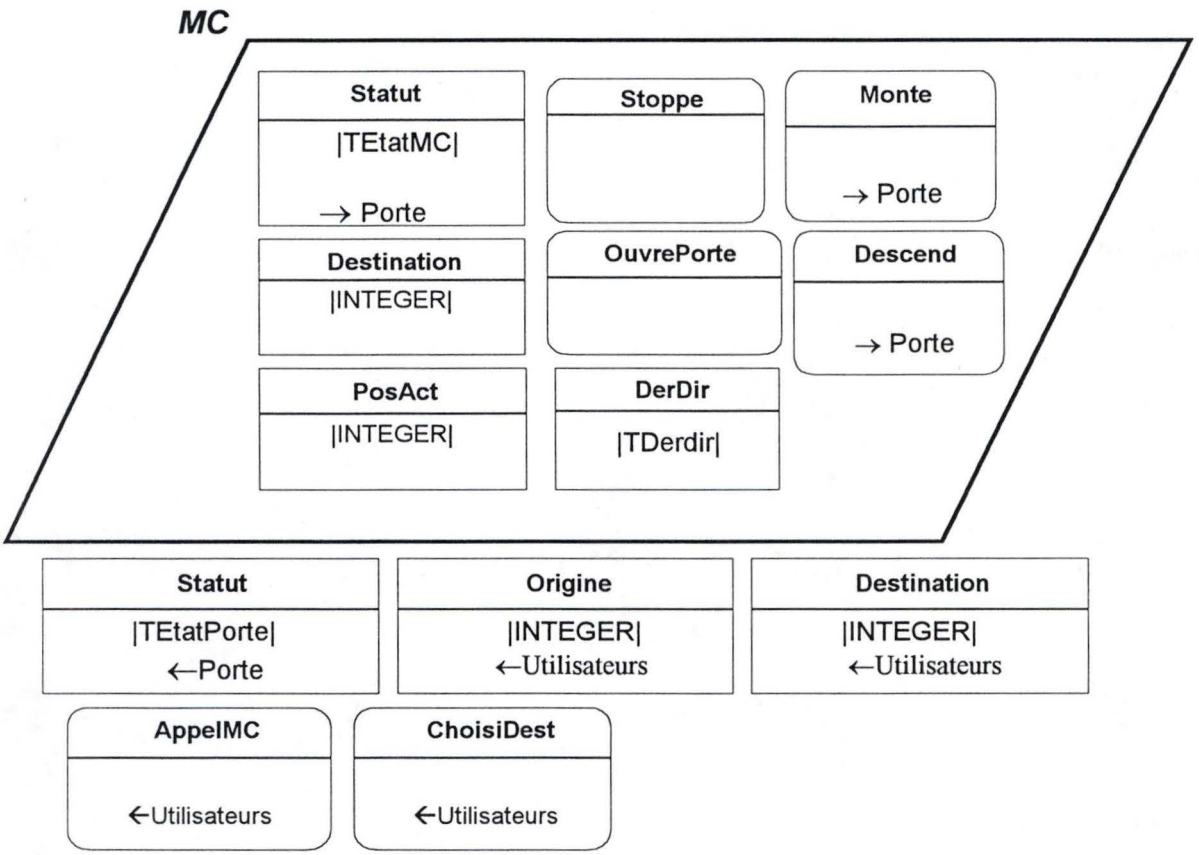
STATE PERCEPTION

$\mathcal{XX}$*(MC.Statut/* TRUE*)* *(2.14)*

STATE INFORMATION

$\mathcal{XX}$ (*Statut.MC /* TRUE) *(2.15)*

3.3. L'agent MC



Agent : MC

DECLARATION

STATE COMPONENTS

| Cet état décrit les différents le statut du MC.
Statut instance-of *TEtatMC*
| Cet état décrit l'étage actuel du MC.
PosAct instance-of *Integer**
| Cet état décrit la prochaine destination du MC.
Destination instance-of *Integer*
| Cet état décrit la dernière direction prise par le MC.
DerDir instance-of *Tderdir*

BASIC CONSTRAINTS

INITIAL VALUE

Statut = Arrete
PosAct = 0

Destination = 0
DerDir = Haut

DECLARATIVE CONSTRAINTS

STATE BEHAVIOUR

La Position Actuelle du MC est entre l'étage -1 et 3 et la valeur de l'étage change de +- 1 en alternance avec la valeur undef (modélise la position intermédiaire du MC).

[] $-1 \leq PosAct \leq 3 \wedge PosAct = p \Rightarrow$ SomF$(PosAct = p' \wedge (p' =$ UNDEF $\Rightarrow$ SomF $(PosAct = p'' \wedge p'' = p+1 \vee p'' = p-1)))$

La Destination du MC est entre l'étage -1 et 3

[] $-1 \leq Destination \leq 3$

A un moment donné, le MC doit arriver à destination

[] $(PosAct \neq Destination) \Rightarrow$ SomF $(PostAct = Destination)$ (3.1)

Quand le MC monte, la porte doit être fermée, le statut du MC est montant, la position actuelle du MC est au plus en dessous de sa destination et il doit arriver à destination, la dernière direction prise par le MC est le haut et si la destination du MC était différente de l'étage d'appel d'un utilisateur, la destination doit encore être différente (exprime le fait que quand le MC monte, aucun appel n'interrompt la montée du MC).

[Monte] (3.2)

$Porte.Statut = Fermee \wedge Statut = Descendant \wedge PosAct \leq Destination \wedge PosAct < Destination \Rightarrow$ SomF $PostAct = Destination \wedge DerDir = Haut \wedge Destination \neq Utilisateurs.Origine$ Since $Destination \neq Utilisateurs.Origine$

Quand le MC descend, la porte doit être fermée, le Statut du MC est descendant, la position actuelle du MC est au plus au dessus de sa destination et il doit arriver à destination, la dernière direction prise par le MC est le bas.

[Descend] (3.3)

$Porte.Statut = Fermee \wedge Statut = Montant \wedge PosAct \geq Destination \wedge PosAct > Destination \Rightarrow$ SomF $PostAct = Destination \wedge DerDir = Bas$

Quand le MC est stoppé, son Statut est arrêté et, ou bien aucune nouvelle destination n'a été attribuée au MC, ou bien une nouvelle destination est attribuée au MC mais les portes ne sont pas fermées.

[Stoppe] (3.4)

$Statut = Arrete \wedge (PosAct = Destination \vee (PosAct \neq Destination \wedge Porte.Statut \neq Fermee))$

Quand le MC est ouvre la porte, son Statut est arrêté et aucune nouvelle destination n'a été attribuée au MC et les portes ne sont pas ouvertes.

[OuvrePorte] (3.5)

$Statut = Arrete \wedge PosAct = Destination \wedge Porte.Statut \neq Ouverte$

La future destination du MC est la plus proche **destination** attribuée se trouvant dans le sens contraire de sens du service précédent, jusqu'a qu'il y ait des attributions se trouvant dans le sens du service précédent. Auquel cas, on prend la plus proche.

[] (3.6)

$Destination = Utilisateurs.Destination$ Since $((Destination = d' \wedge (PosAct \leq Utilisateurs.Destination \leq d' \wedge DerDir \neq Haut) \vee (d' \geq Utilisateurs.Destination \geq PosAct \wedge DerDir \neq Bas))$ Until!

$(Destination = d'' \wedge (PosAct \leq Utilisateurs.Destination \leq d'' \wedge DerDir = Haut) \vee (d'' \geq Utilisateurs.Destination \geq PosAct \wedge DerDir = Bas)))$

La future destination du MC est le plus proche **appel** attribué se trouvant dans le sens contraire du sens du service précédent, jusqu'a qu'il y ait

| des attributions se trouvant dans le sens du service précédent. Auquel cas, on prend la plus proche.

[] (3.7)

Destination = Utilisateurs.Origine Since *((Destination = d' ∧ (PosAct ≤ Utilisateurs.Origine ≤ d' ∧ DerDir ≠ Haut) ∨ (d' ≥ Utilisateurs.Origine ≥ PosAct ∧ DerDir ≠ Bas))* Until!
(Destination = d'' ∧ (PosAct ≤ Utilisateurs.Origine ≤ d'' ∧ DerDir = Haut) ∨ (d'' ≥ Utilisateurs.Origine ≥ PosAct ∧ DerDir = Bas)))

OPERATIONAL CONSTRAINTS

PRECONDITIONS

| On ne peut monter que quand la porte est fermée, que quand la future destination est plus haute que la position actuelle du MC et que le MC est arrêté.

Monte : Porte.Statut = Fermee ∧ Destination > PosAct ∧ Statut = Arrete (3.8)

| On ne peut descendre que quand la porte est fermée, que quand la future destination est plus basse que la position actuelle du MC et que le MC est arrêté.

Descend : Porte.Statut = Fermee ∧ Destination < PosAct ∧ Statut = Arrete (3.9)

| On ne peut stopper le MC que quand la porte est fermée, que quand la position actuelle du MC est la future destination est plus haut et que le MC n'est pas arrêté.

Stoppe : Porte.Statut = Fermee ∧ Destination = PosAct ∧ Statut ≠ Arrete (3.10)

| On ne peut ouvrire la porte que quand celle-ci est fermée, que quand le MC est a destination et que le MC est arrêté.

OuvrePorte : Porte.Statut = Fermee ∧ Destination = PosAct ∧ Statut = Arrete (3.11)

EFFECTS OF ACTIONS

Monte : *DerDir := Haut*
Statut := Montant
[] (3.12)

Stoppe : *Statut := Arrete*
[] (3.13)

Descend : *DerDir := Bas*
Statut := Descendant
[] (3.14)

Utilisateurs.ChoisiDest: *Destination := Utilisateur.Destination*
[] (3.15)

Utilisateurs.AppelMC: *Destination := Utilisateur.Origine*
[] (3.16)

TRIGGERINGS

| L'action Monte est enclenchée si la porte est fermée et que la destination du MC se trouve au dessus de la position actuelle alors que le MC est arrêté.

*Porte.Statut = Fermee ∧ Destination > PosAct ∧ Statut = Arrete/*TRUE*~>Monte* (3.19)

| L'action Descend est enclenchée si la porte est fermée et que la destination du MC se trouve en dessous de la position actuelle alors que le MC est arrêté.

*Porte.Statut = Fermee ∧ Destination < PosAct ∧ Statut = Arrete/*TRUE*~>Descend* (3.20)

| L'action Stoppe est enclenchée si la porte est fermée et que le MC est à destination et qu'il n'est pas arrêté.

Porte.Statut = Fermee ∧ Destination = PosAct ∧ Statut ≠ Arrete/TRUE~>*Stoppe* *(3.21)*

L'action OuvrePorte est enclenchée si la porte est fermée et que le MC est à destination alors que le MC est arrêté.

Porte.Statut = Fermee ∧ Destination = PosAct ∧ Statut = Arrete/TRUE~>*OuvrePorte* *(3.22)*

COOPERATION CONSTRAINTS

ACTION PERCEPTION

$\mathcal{XR}$*(Utilisateurs.AppelMC /* TRUE*)* *(3.23)*
$\mathcal{XR}$*(Utilisateurs.ChoisiDest /* TRUE*)* *(3.24)*

ACTION INFORMATION

$\mathcal{XR}$*(Monte.Porte /* TRUE*)* *(3.25)*
$\mathcal{XR}$*(Descend.Porte /* TRUE*)* *(3.26)*
$\mathcal{XR}$*(OuvrePorte.Porte / Statut = Arrete)* *(3.27)*

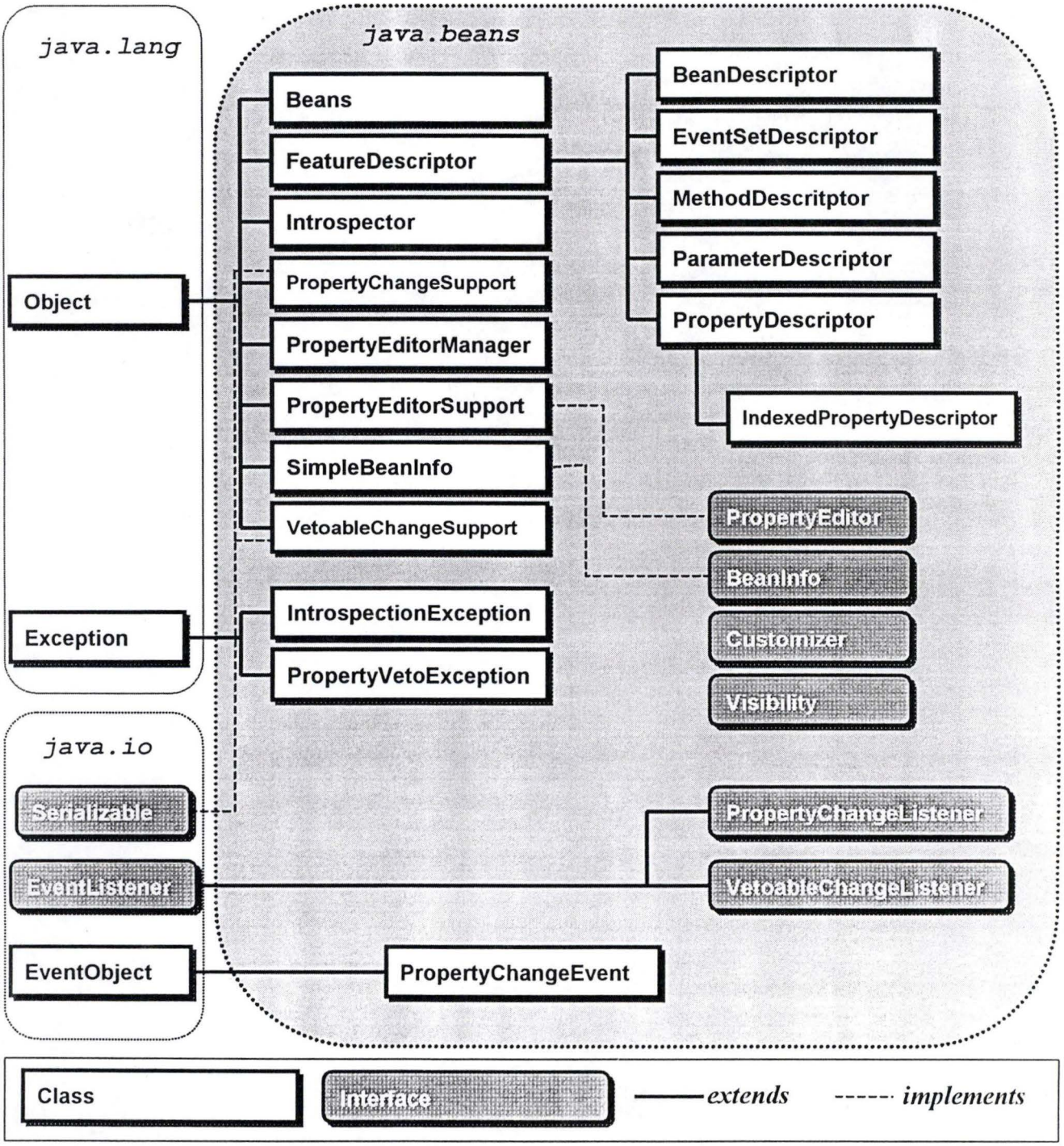
STATE PERCEPTION

$\mathcal{XR}$*(Porte.Statut /* TRUE*)* *(3.28)*
$\mathcal{XR}$*(Utilisateurs.Destination /* TRUE*)* *(3.29)*
$\mathcal{XR}$*(Utilisateurs.Origine /* TRUE*)* *(3.30)*

STATE INFORMATION

$\mathcal{R}$*(Statut.Porte / TRUE)* *(3.31)*

Annexe 5
Le package `java.beans`



Erreur! Signet non défini.*java.beans.BeanInfo*[7]

```
public abstract interface BeanInfo {
// Constants
public static final int ICON_COLOR_16x16;
public static final int ICON_COLOR_32x32;
public static final int ICON_MONO_16x16;
public static final int ICON_MONO_32x32;
// Public Instance Methods
public abstract BeanInfo [ ] getAdditionalBeanInfo();
public abstract BeanDescriptor getBeanDescriptor();
public abstract int getDefaultEventIndex();
public abstract int getDefaultPropertyIndex();
public abstract EventSetDescriptor [ ] getEventSetDescriptors();
public abstract Image getIcon(int iconKind);
public abstract MethodDescriptor [ ] getMethodDescriptors();
public abstract PropertyDescriptor [ ] getPropertyDescriptors();
}
```

Erreur! Signet non défini.*java.beans.Introspector*

```
public class Introspector extends Object{
// Class Methods
public static String decapitalize(String Name);
public static BeanInfo getBeanInfo(Class beanClass) throws IntrospectionException;
public static BeanInfo getBeanInfo(Class beanClass, Class stopClass) throws IntrospectionException;
public static String[ ] getBeanInfoSearchPath();
public static void setBeanInfoSearchPath(String[ ] path);
}
```

java.beans.PropertyDescriptor

```
public class PropertyDescriptor extends FeatureDescriptor{
// Public Constructors
public PropertyDescriptor(String propertyName, Class beanClass) throws IntrospectionException;
public PropertyDescriptor(String propertyName, Class beanClass, String getterName,
                          String setterName) throws IntrospectionException;
public PropertyDescriptor(String propertyName, Method getter, Method setter) throws
                          IntrospectionException;
// Public Instance Methods
public Class getPropertyEditorClass( );
public Class getPropertyType( );
public Method getReadMethod( );
public Method getWriteMethod( );
public boolean isBound( );
public boolean isConstrained( );
public void setBound(boolean bound);
public void setConstrained(boolean constrained);
public void setPropertyEditorClass(Class propertyEditorClass);
}
```

...

[7] Issu de [NUTSHELL97].