Bocal ou la boîte comme paradisgme programmatique : application à la simulation de systèmes productifs

Ayman Hamadeh

THESE

PRESENTEE PAR

HAMADEH Ayman
(ingénieur ENSIMAG)

POUR OBTENIR LE TITRE DE DOCTEUR
DE L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1
(ARRETES MINISTERIELS DU 5 JUILLET 1984 ET
DU 30 MARS 1992)

(SPECIALITE : MATHEMATIQUES APPLIQUEES)

==========

BOCAL OU LA BOÎTE COMME PARADIGME PROGRAMMATIQUE : APPLICATION A LA SIMULATION DE SYSTEMES PRODUCTIFS

==========

DATE DE SOUTENANCE : 27 OCTOBRE 1993

COMPOSITION DU JURY

PRESIDENT :	LADET Pierre
RAPPORTEURS :	PRUNET François
	QUILLIOT Alain
EXAMINATEURS :	BORRIONE Dominique
	UHRY Jean-pierre
INVITE :	LACÔTE Christophe

THESE PREPAREE AU SEIN DU LABORATOIRE : ARTEMIS

A mes parents

Remerciements

Le travail présenté dans cette thèse s'inscrit dans la suite des travaux menés au laboratoire ARTEMIS dans le domaine de la gestion de production. Cette thèse a été préparée en collaboration avec la société ALMA.

Je tiens à remercier :

Monsieur Pierre LADET, Professeur à l'INPG, pour avoir bien voulu accepter de présider le jury de cette thèse.

Monsieur Alain QUILLIOT, Professeur à l'université de Clermont-Ferrand, pour l'intérêt et l'enthousiasme manifestés et pour avoir accepté d'être rapporteur.

Monsieur François PRUNET, Professeur à l'université de Montpellier, pour avoir eu gentillesse de s'intéresser à notre travail et pour avoir accepté d'en être rapporteur.

Madame Dominique BORRIONE, Professeur à l'université Joseph Fourrier, pour l'attention portée à l'examen de cette thèse.

Mes vifs remerciements sont adressés au directeur de la thèse Jean-Pierre UHRY, Directeur de la société ALMA, pour ses idées originales et ses conseils constructifs tout au long de cette recherche. Les longues séances de travail avec Jean-Pierre ont été cardinales pour le développement et la résolution des points difficiles de ce travail.

J'adresse mes remerciements à Christophe Lacôte pour son vif soutien à ce travail que ce soit à l'intérieur de la société ALMA ou au laboratoire ARTEMIS. Merci aussi pour les cours de philosophie et d'histoire.

Je remercie tout le personnel de la société ALMA pour l'accueil chaleureux pendant les années de la thèse. Je remercie, tout particulièrement, Roger pour les fautes d'orthographe qu'il n'a pas cessé de corriger et Alain (Titi) qui m'a souvent aidé à l'impression du manuscrit.

Je remercie tous les amis qui m'ont aidé à la préparation du pot.

Table des matières

Introduction

La planification de production, sous tous ses aspects, a toujours été un domaine d'application privilégié de la Recherche Opérationnelle. Et donc un des champs de recherche du laboratoire ARTEMIS dès son origine. L'évolution simultanée des outils informatiques, devenus interactifs et graphiques, et des impératifs de production qui privilégient les horizons courts ont donné une importance croissante à la modélisation et à la simulation au détriment de la programmation mathématique plus à l'aise dans le long terme.

D'expérience, nous nous sommes aperçus que les outils actuellement disponibles souffraient tous de lacunes diverses sur lesquelles nous aurons l'occasion de revenir. Ces lacunes proviennent sans doute de ce que les concepts sur lesquels ils sont basés sont insuffisants pour l'objet auquel ils prétendent.

Notre démarche a consisté à partir de l'aval, c'est-à-dire à définir deux ou trois caractéristiques que devrait posséder un bon outil de modélisation et d'en tirer des conclusions de type sémantique. Cela nous a conduit à définir un nouveau langage de modélisation, provisoirement appelé BOCAL et qui constitue l'objet central de ce travail.

Il peut paraître paradoxal que des mathématiciens de l'algorithmique s'égarent dans une activité d'ordinaire réservé aux chercheurs en informatique mais cela résulte simplement de ce que les langages sont au coeur de l'activité de modélisation et somme toute on peut voir aussi les mathématiques comme un langage.

Mais le champ de BOCAL va au delà de la simulation des systèmes productifs. La spécification et la simulation d’automates est également un domaine d'application de BOCAL, de même (mais dans une moindre mesure) le maquettage de produits informatiques interactifs.

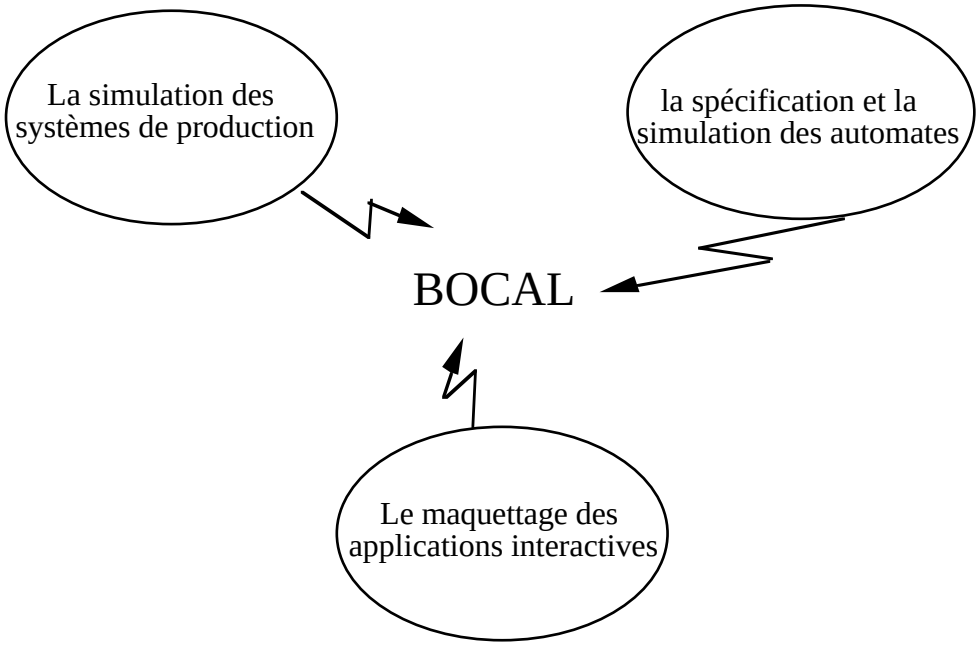


figure Intro.1

La spécification des systèmes aussi complexes que les systèmes de production pose la question sur les méthodes utilisées pour modéliser le système concerné. Cette complexité a entraîné l'apparition croissante des simulateurs visant à rendre la modélisation et la simulation à la portée de "tout le monde". Cependant, en s'approchant de l'utilisateur, ces simulateurs ont perdu leur puissance de spécification, cette tâche reste un domaine privé pour les développeurs et n'est pas encore démocratisée.

Il est commode de classer les utilisateurs de tout produit de simulation en trois catégories : les concepteurs des objets de base (machines, stocks, gammes, transporteurs, etc..), les modélisateurs qui, à partir de ces objets, construisent les systèmes à simuler et finalement les opérateurs qui vont faire fonctionner le système dans divers scénarios (ajustement de paramètres). Bien sûr il peut se faire que dans la réalité, ces utilisateurs ne soient pas complètement différenciés. Ce qui caractérise tous les progiciels de simulation, c'est que ces divers utilisateurs utilisent des langages et des interfaces distinctes.

L'ambition de BOCAL, ce qui en constitue l'innovation majeure, est de permettre à ces trois niveaux de fonctionner dans le même environnement sémantique ou, pour le dire plus simplement en utilisant le même langage de programmation. Les avantages d'une telle approche sont évidents : permettre à tous les utilisateurs de bénéficier de la modularité et de l'extensibilité.

Dans le domaine de la spécification et de la simulation d'automates, BOCAL vise à permettre aux utilisateurs non spécialisés la réalisation d'une maquette par l'assemblage des différents constituants de la maquette et d'apporter eux-mêmes toutes les transformations nécessaires. L'une des difficultés majeures non surmontées aujourd'hui tient à la diversité des intervenants concernés par la phase de spécifications. Citons, sans chercher à être exhaustif : le responsable produit marketing, un représentant de l'utilisateur final ou du client, le chargé du développement, un ergonome...

L'intervention répétée du développeur pour réaliser les transformations de la maquette à la demande des différents intervenants nous conduisent à l'idée d'un "atelier de maquettage" complet permettant au client d'assembler les différents constituants de la maquette et de réaliser lui-même toutes les transformations sans faire recours au développeur. Cet atelier permettra aussi de générer, sous une forme à définir, un document d'aide à la spécification afin de faire profiter les équipes de développement, lors de la rédaction des spécifications détaillées, de tout le travail d'analyse déjà fourni pour l'élaboration de la maquette et qui, actuellement, est effectué une seconde fois lors de cette phase.

Tout d'abord et avant la présentation des chapitres de ce mémoire, nous réclamons l'indulgence du lecteur pour le vocabulaire anglais souvent utilisés dans ce mémoire qui engendre parfois des phrases choquantes pour un puriste. Ce vocabulaire anglais est dû au fait que BOCAL a été développé dans un contexte "industriel", ce qui impose pratiquement l'anglais comme langue de description de la syntaxe.

Le présent mémoire est organisé en six chapitres.

Le premier est consacré à la simulation en productique. Différents progiciels et langages de simulation sont présentés afin de situer notre contribution. Nous en déduisons les caractéristiques principales d'un bon outil de simulation et nous définissons nos objectifs dans ce domaine.

Le deuxième présente les problèmes rencontrés dans le domaine du maquettage d'automates. Nous y décrivons les outils de spécification et les logiciels de maquettage. Nous en déduisons les qualités d'un atelier de maquettage basé sur les concepts de BOCAL.

Le troisième chapitre concerne la présentation de l'approche orientée objet. Nous indiquons ses principales qualités, des langages orientés objets et finalement des environnements de programmation qui servent au maquettage des applications interactives. L'approche orientée objet reste le guide de la conception de BOCAL.

BOCAL, s'il ne répond pas exactement à la définition d'un langage orienté objet, appartient sans aucun doute à la filiation des SIMULA, SMALLTALK et autres C++ et relève de ce vieux rêve de transformer la programmation en assemblage de composants "logiciels" soigneusement testés et éprouvés, la programmation LEGO en quelque sorte. Toutefois, à la différence des langages orientés objets qui ont eu toujours plutôt pour cible le travail du programmeur, sans se soucier outre mesure de l'utilisateur final, BOCAL, lui, a une autre ambition, celle qu'on retrouve au cœur d'un système comme l'HYPERCARD d'Apple, la "démocratisation" de la programmation, ce qui, somme toute, a fait le succès des tableurs.

La première partie du quatrième chapitre introduit aux concepts de BOCAL et tente de montrer comment se sont peu à peu dégagées les spécifications actuelles. Une seconde partie est consacrée à une présentation détaillée des concepts de BOCAL et de sa syntaxe textuelle. Une troisième et dernière partie présente la notion de compatibilité et ses variantes.

Dans le cinquième chapitre, nous construisons une bibliothèque des modèles de base utiles pour toute simulation du système à événements discrets. Et nous commençons la construction d'une bibliothèque dédiée à la modélisation de systèmes de production que nous illustrons par des exemples concrets. La modélisation en BOCAL de ces exemples sert, nous l'espérons, à mettre en évidence les qualités de BOCAL. Des tests de performance sont réalisés pour vérifier l'efficacité du langage.

Enfin nous présentons des extensions possibles du langage qui permettront la poursuite de nos travaux.

Chapitre I

La simulation des systèmes de production

Ce chapitre doit beaucoup aux auteurs du rapport du Pole régionale de productique Rhône-alpes 1988.

Nous présentons dans ce chapitre une brève étude sur le schéma habituel de la modélisation d'un système de production en soulignant l'apport de la simulation aux différentes phases de la mise en place d'un système de production.

Ensuite, la définition des caractéristiques d'un "bon" outil de simulation nous permettra dans la suite de juger et de comparer les produits de simulation présentés. Puis, nous citons les langages et les progiciels de simulation existants dans la littérature de la simulation et sur le marché industriel. Nous présentons plus en détail un logiciel, appelé EXTEND, qui se rapproche par certains points de BOCAL, en soulignant ses qualités et ses lacunes.

Enfin, nous exposons les outils qui se rapprochent de BOCAL par certains points, ainsi que les objectifs visés par BOCAL.

I.1 Introduction

La complexité de mise en place des systèmes de production (machines, chaînes de fabrication, ateliers, etc...) exige le recours à une aide informatisée que ce soit pendant la phase de la conception et de la mise en place d'installations de production ou pendant le fonctionnement du système. Les méthodes analytiques, trop restrictives, peuvent apporter une aide lors d'une évaluation approchée des performances du système envisagé. Par contre, les outils de simulation peuvent apporter aux différents intervenants d'un projet et selon les critères et les besoins de chacun, une connaissance sur le comportement du système étudié. Si la simulation est une aide à la conception, c'est également une aide à l'exploitation.

Un "bon" simulateur devrait prendre en compte naturellement une analyse descendante du système à modéliser et permettre une programmation remontante par composition. Pour faire cela, il doit permettre de qualifier tout élément entrant dans la description du système : pièces avec leurs gammes, machines avec leurs durées opératoires, stocks et leurs règles de gestion, moyens de transport et leurs mécanismes, ressources diverses. Il doit permettre aussi de décrire et de modéliser les aspects enchaînement d'opérations (parallélisme et synchronisation) proches de l'atelier.

Il faut noter que la simulation n'est qu'un des éléments intervenant dans un processus de conception. Elle aidera à valider ou invalider une hypothèse mais n'apportera pas l'hypothèse. Si une simulation conduit à faire évoluer une solution, c'est par l'interprétation des résultats, mais l'interprétation reste un processus non automatisable.

I.2 Modélisation et simulation

La simulation s'impose comme un outil d'aide à la décision sur tous les niveaux de conception d'un système de production [Dol83] [Pro86] [Woi87]. A tous ces niveaux, on retrouve le schéma traditionnel de la simulation, à savoir :

***** réalisation d'un modèle correspondant à un élément précis du système étudié (modèle mathématique, symbolique et graphique, ou pictural),

***** étude de ce modèle (étude des propriétés intrinsèques, étude analytique du comportement dynamique, simulation du modèle).

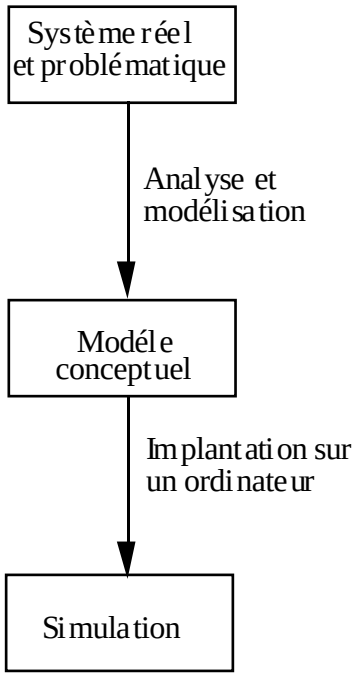


figure I.1 le chemin habituel de la simulation

Dans la constitution des modèles, on distingue traditionnellement trois approches différentes ([Bel83] [Mon85] [Pro84]) :

1) par événements : on étudie le système à travers ses changements d'état,

2) par activités : on étudie les activités en cours à un instant donné,

3) par processus : on identifie le système à un réseau de processus prédéterminés et connus.

Cette distinction semble un peu dépassée par l'apparition des outils associant les différentes approches. De plus, ces différentes approches font intervenir uniquement l'aspect fonctionnel du système, la notion de structures de données leur est totalement inconnue.

La modélisation et la simulation ont en fait trois types d'objectifs (quantitatifs, qualitatifs et stratégiques) associés aux différentes phases de mise en place des moyens de production. [Ala84] [Bel83] [Bel85] [Ben85] [MRT85] [Mon85]

On peut associer des outils aux différents niveaux de conception. L'utilisation des outils différents à chaque niveau de conception pose la question de communication entre les résultats de la simulation des différents niveaux.

I.2.1 Le niveau conception générale

Au niveau conception générale, le but est d'avoir des données quantitatives sur les flux de pièces, la charge des systèmes de transport, les emplacements de stocks à prévoir, le nombre de moyens de production à installer, la réaction du système face à des impondérables, l'évolutivité de ce système, etc...

Les outils utilisables à ce niveau sont :

1) la modélisation mathématique (que ce soit avec la théorie des files d'attente [Par85], les chaînes de MARKOV, etc...) permettant l'analyse stochastique du processus [Dal84]. Un modèle mathématique peut rapidement amener des résultats intéressants en faisant abstraction des règles de décision et en faisant des hypothèses simplificatrices.

2) l'utilisation d'un langage de simulation à événements discrets (Q-GERT, SLAM, Q-NAP, SIMAN, etc...) permettant la simulation du fonctionnement du système [Pri79] [Ped82]. Une simulation permet de valider ou améliorer les choix effectués et un certain nombre de décisions déjà prises.

En général, les modèles mathématiques et les langages de simulation ne sont pas concurrents, mais le plus souvent complémentaires.

Ces langages de simulation permettent la description de la structure du système par des graphes en utilisant des noeuds prédéfinis (noeud file d'attente, noeud de réservation de ressource, etc...) et paramétrables, les arcs entre ces noeuds ont aussi des fonctions déterminées (activités par exemple). Ces langages ne permettent pas la définition d'un nouveau type de noeuds.

Une fois le modèle construit, ces langage permettent de faire circuler dans le réseau des entités typées, et de faire une analyse des résultats et des calculs statistiques. On présentera rapidement, dans le paragraphe I.4.3, les principales caractéristiques de quelques langages de simulation couramment utilisés.

Ces langages ne sont pas accessibles aux utilisateurs de tout niveau. On construit souvent une sur-couche logicielle permettant un accès facile aux utilisateurs non spécialisés dans la programmation.

I.2.2 Le niveau conception détaillée

Au niveau conception détaillée, le but de la simulation est de vérifier le bon fonctionnement des équipements mis en place. La simulation du fonctionnement du système se décompose en deux classes : une simulation d'ordre mécanique, et une simulation d'ordre plutôt logique.

1. Simulation mécanique

La simulation d'ordre mécanique s'intéresse essentiellement à la capacité physique des équipements de travailler dans de bonnes conditions (étude de trajectoires, problèmes d'accessibilité, problèmes d'évacuation, etc...).

Pour faire cela, on utilise donc des logiciels conçus à partir des logiciels de CAO 3D. Ces logiciels permettront de modéliser les équipements en question et de les animer avec possibilité de :

1) déplacement,
2) rotation,
3) zoom,
4) ralentissement, accélération,
etc...

2. Simulation de la logique de pilotage

La simulation de la logique de pilotage s'intéresse aux protocoles de communication entre équipements, aux règles de synchronisation, aux procédures de reprise, etc... Le but est de vérifier la sécurité de leur fonctionnement.

Les outils utilisables à ce niveau sont grafcet [Dal83] et Réseaux de Petri [Hol85] [Vid86] qui permettent une modélisation graphique des protocoles ou des procédures de reprise et d'étudier les propriétés de ces systèmes(absence de "dead-lock", etc...).

I.2.3 Le niveau fonctionnement

A ce niveau, les objectifs de la simulation sont d'ordre stratégiques. Le but est alors d'examiner des orientations possibles :

1) des décisions de prise de commandes ou de marchés,
2) des décisions d'abandon de commandes ou de marchés,
3) des perturbations dans les approvisionnement,
4) des pannes ou immobilisations volontaires des moyens,
5) des modifications de gammes,
etc...

Les logiciels de GPAO comportent de plus en plus souvent des outils de simulation stratégiques. Ces outils sont de plus en plus des outils interactifs, conviviaux et permettent à l'utilisateur de forger ses propres décisions.

I.3 Caractéristiques générales d'un "bon" simulateur

Mais pour juger de l'état actuel des produits de simulation, il convient de développer un peu les quelques principes de base que doit vérifier, à notre avis, un produit de simulation de production.

En précisant les qualités qu'un produit de simulation doit vérifier, on pourra comparer et puis juger les différents progiciels de simulation existant sur le marché, et cela nous permettra aussi de déterminer les objectifs du simulateur BOCAL.

I.3.1 Modularité

On entend par là la capacité du produit à prendre en compte une description modulaire du problème et ceci sous trois aspects :

1) Décomposition

Ceci implique que le produit prenne en compte naturellement la possibilité de décomposer un problème ou un modèle en sous problèmes :

Un atelier se décompose en sous ateliers lesquels se décomposent en postes de travail lesquels ...

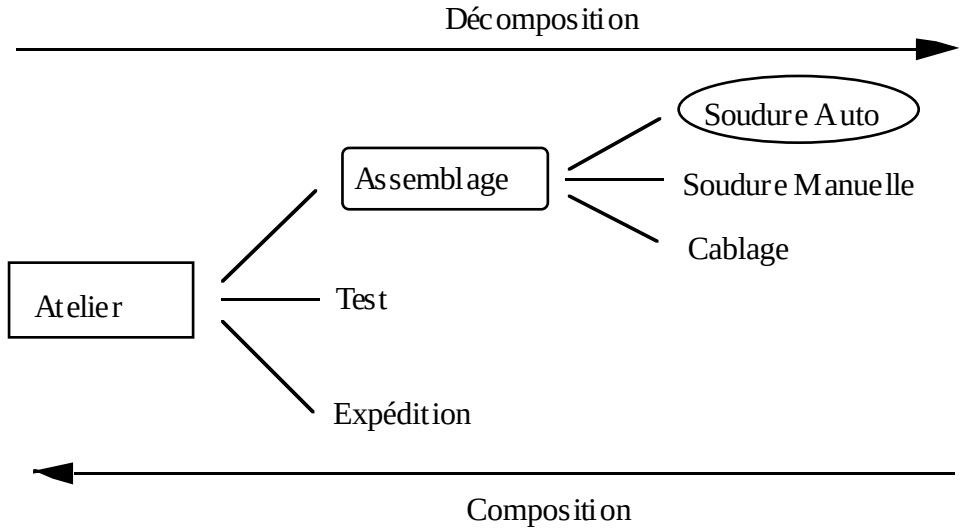


figure I.2
analyse descendante et programmation remontante

2) Composition

Très lié au précédent c'est en quelque sorte l'opération inverse, la capacité du système à regrouper un ensemble de sous modèles en un modèle composé.

On peut penser que si la décomposition est facile, la composition doit l'être également, mais ceci est rarement le cas (tout bricoleur sait d'ailleurs qu'il est plus facile de démonter que de monter). On connaît l'adage "Analyse descendante, programmation ascendante", autrement dit la spécification part du problème général pour le décomposer en problèmes plus simples et la réalisation effective doit suivre le cheminement inverse. On sait aujourd'hui que ce schéma ne colle pas à la réalité de la conception qui, de manière

dialectique, implique un va et vient entre décomposition et composition . Ce point, la capacité à regrouper, est un critère très important de jugement de la praticabilité d'un produit de simulation

3) Réutilisabilité

La réutilisation de portions de programmes est un enjeu majeur de l'informatique qui a du mal à passer du stade artisanal actuel à un style industriel et la vogue des ateliers logiciels en témoigne pour partie.

Mais ce qui est vrai au niveau du programmeur l'est également au niveau de l'utilisateur. Il convient pour des raisons à la fois économiques et de fiabilité que la modélisation d'un sous-système soit réutilisable ailleurs. C'est l'idée de la programmation LEGO, la modélisation s'appuyant sur l'utilisation de "composants" prédéfinis mais aussi enrichissant cette bibliothèque de modèles de base.

I.3.2 Extensibilité

La question qui se pose ici, est l'ouverture du système. Un produit de simulation peut offrir quatre modèles de machines, six modèles de stocks et cinq modèles de transporteurs. Bien sûr ces modèles sont paramétrables. Mais que se passe-t-il si on doit modéliser, par exemple, une machine d'un type non prévu? L'extensibilité mesure la capacité d'ouverture du produit de simulation à des situations nouvelles sans recours au concepteur du programme.

I.3.3 Ergonomie

C'est une caractéristique complexe à définir, au delà des aspects d'interface utilisateur (programmation graphique ou textuelle). Les qualités recherchées ici sont la facilité d'appropriation de l'outil informatique par l'utilisateur à tous les niveaux d'utilisation, y compris conceptuels.

I.3.4 Cohèrence de modélisation

C'est la possibilité que chaque objet dans un modèle ait plusieurs représentations selon la finesse de la modélisation et la granularité du temps. Lors de la phase de conception détaillée, un sous atelier composé de plusieurs postes de travail est représenté par un modèle qui met en évidence le flux de produit entre les postes de travail, ainsi que la communication et la synchronisation entre les différentes machines. Alors, on aimerait voir

le modèle détaillé de l'atelier. Par contre, pendant la phase de conception générale le modélisateur assemble des sous ateliers pour obtenir des ateliers, des ateliers pour obtenir un système de production ... A ce niveau là, on n'a pas besoin de simuler le fonctionnement détaillé de chaque atelier. Les ateliers devraient avoir des fonctionnements approximatifs.

Par souci que la taille du système explose, le logiciel de simulation devrait permettre à un objet d'avoir plusieurs représentations possibles dont chacune sera choisie selon la finesse de la modélisation. Dans ce cas, on peut interchanger cette représentation fine du sous atelier par une autre plus agrégée et on ne verra plus les composantes internes de ce sous atelier. Une pièce ou un chariot entrant dans le sous atelier sort avec des nouvelles valeurs correspondant aux temps d'attente, temps de passage, date de sortie... Ces valeurs dans la représentation agrégée peuvent être calculées approximativement.

Cette caractéristique "consistance de modélisation" nous semble très importante lors de la modélisation des grands systèmes de production.

I.3.5 Autres exigences

La description du flux de produit est une phase très importante dans la spécification d'un système de production. En général, les entités qui circulent dans un système sont des objets passifs pour la plupart des simulateurs et des langages connus. A l'inverse, par exemple, une voiture, qui circule dans un réseau routier, connaît elle-même son chemin ou encore plus la logique de sa circulation. Cela signifie que arrivant à une intersection (équipée d'un feu tricolore ou d'un stop) la voiture (ou plutôt son conducteur) pourra décider quelle direction elle doit emprunter.

Un "bon" simulateur devrait permettre la dualité entre flux de produit et flux de gammes. Chaque pièce peut être équipée de sa gamme de fabrication et elle choisit son chemin dès son entrée dans l'atelier. Toutefois, on peut avoir des ateliers spécialisés où les pièces qui entrent dans ces ateliers sont passives et suivrons un chemin prédéterminé connu par l'atelier. Cette dualité est déjà connue dans les langages de programmation entre données et procédures (ou méthodes).

Un simulateur comportera également, pour une utilisation aisée et un service de qualité, un éditeur graphique autorisant la description et l'animation de synoptiques, une bibliothèque de programmes qui permettra une analyse des résultats de la simulation.

I.4 Les logiciels de simulation

Nous pouvons distinguer deux étapes importantes lors de l'utilisation d'un logiciel de simulation. D'abord une phase de modélisation qui représentera l'architecture de l'atelier ainsi que son fonctionnement, d'autre part une phase d'interprétation et d'analyse des résultats (figure I.1).

On peut classer facilement, selon l'outil de base utilisé, les logiciels de simulation en trois familles : les logiciels dédiés écrits en langages traditionnels (Fortran, Pascal ou C), les logiciels de simulation utilisant un langage spécifique et les logiciels basés sur les réseaux de Petri.

I.4.1 Niveaux d'utilisation

Il est commode de classifier les utilisateurs d'un produit de simulation en trois catégories (figure I.3). Bien sûr il peut se faire que dans la réalité, ces utilisateurs ne soient pas complètement différenciés.

En général, au niveau 1 on utilise un langage général de type C. Au niveau 2 on utilise un langage simplifié (ou un langage de simulation) et au niveau 3, on a droit à un environnement type menu et fenêtre de dialogue.

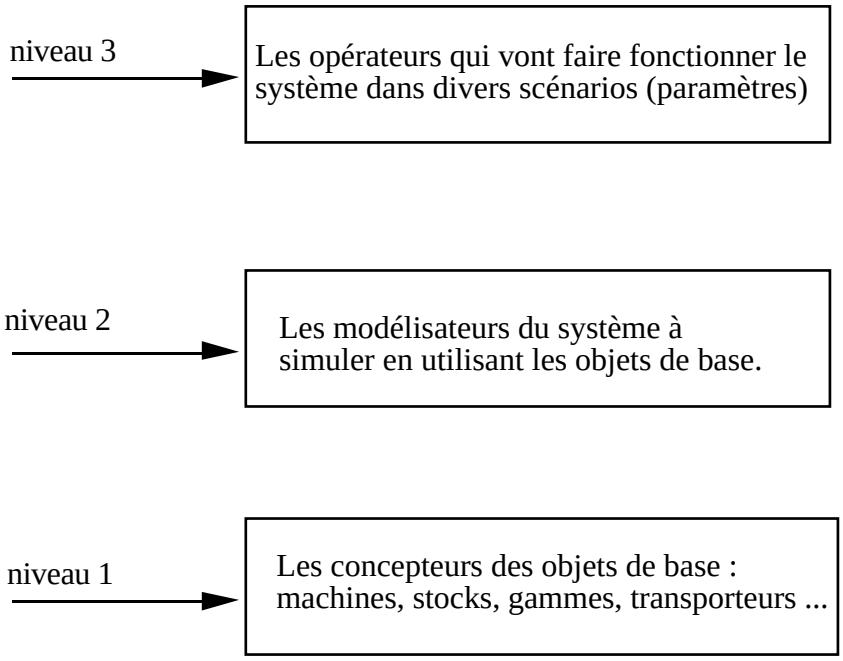


figure I.3

Les différents niveaux d'utilisation

I.4.2 Logiciels dédiés écrits en langages généraux

Face à un type de problèmes particulier, l'utilisateur choisit un logiciel convivial et dédié à ce type de problèmes.

Un logiciel dédié permet à l'utilisateur de résoudre facilement un problème, parce que le programmeur du logiciel a déjà fait une partie de la modélisation, par exemple :

1) il permet la description de la suite des opérations à effectuer par les postes de travail sur les pièces (les gammes).

2) il permet d'associer des priorités aux machines par rapport aux moyens de transport.

...

L'utilisateur n'a qu'à manipuler des objets couramment utilisés dans son métier (machine, stockage, chariot, pièces,...).

Un logiciel dédié est souvent construit directement à partir d'un langage général (Fortran, Pascal ou C). Cependant, il existe des logiciels dédiés basés sur des langages spécifiques, ils présentent les mêmes qualités et les mêmes inconvénients au niveau de l'utilisateur final que ceux écrits en langages généraux.

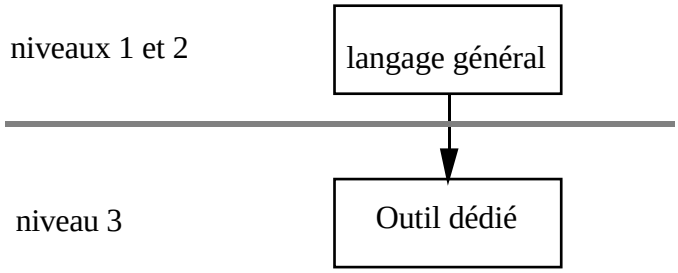


figure I.4 logiciels dédiés écrits en langages généraux

Cette figure montre que les utilisateurs des logiciels dédiés du niveau 1 (les développeurs) et ceux du niveau 2 sont les mêmes.

1) Critiques

L'inconvénient majeur de cette classe de logiciels concerne le nombre de type de problèmes qu'on peut résoudre. Face à un type de problèmes qui sort du cas standard,

l'utilisateur se tourne vers un autre logiciel dédié ou fait recours au programmeur afin de lui apporter les modifications nécessaires.

2) Exemples

On peut citer comme exemple de ce type de programme (sans être exhaustif) SIMFLEX (Simulation orientée ateliers flexibles), C.F.M. (simulation de chaînes flexible de montage) développés à l'INRIA par le groupe Systèmes de Production, OASYS [Bel86] et PARSIFAL (simulation d'ateliers de fabrication par lots) développé au Centre d'Études et de Recherches de Toulouse (CERT/DERA).

I.4.3 Logiciels utilisant un langage spécifique

Ces logiciels tentent de construire un outil convivial et extensible. Un outil est dit convivial s'il n'exige pas un effort particulier d'apprentissage et est dit extensible s'il permet l'ajout de nouvelles fonctions non prévues au départ.

On construit, avec un langage traditionnel, des fonctions pour l'utilisateur (par exemple: saisir une ressource) paramétrables (utiliser la machine M1). On définit un certain nombre de fonctions, plus ou moins abstraites qui constituent avec les règles de synthèse associées un langage spécifique pour la simulation des systèmes. La plupart de tels langages se caractérisent par une symbolique graphique d'une part, une grammaire (comme tout langage de programmation) d'autre part. Le passage du système graphique au langage est souvent automatisé (c'est le cas par exemple pour les langages SLAM et SIMAN).

Un tel langage spécifique constitue un produit très ouvert, pouvant traiter des problèmes aussi différents que l'assemblage ou l'étude des flux d'information dans un réseau, mais pas toujours très convivial car il exige de l'utilisateur un effort (apprendre le langage et l'utiliser en l'adaptant à son problème particulier).

Le constructeur peut décider d'aller plus près de l'utilisateur pour lui offrir un outil convivial. En utilisant un langage spécifique, il développe une couche logicielle dédiée à un certain secteur d'activités. Cette convivialité coûte souvent très cher, l'outil obtenu n'est plus extensible au niveau de l'utilisateur.

Les résultats sont le plus souvent présentés sous formes de tables et de graphiques (histogramme et camembert). Certains, grâce à un générateur d'icônes, offrent des possibilités d'animation (par exemple : SLAM et SIMAN).

La figure I.5 montre le cheminement usuel pour simuler un système en utilisant un langage de simulation :

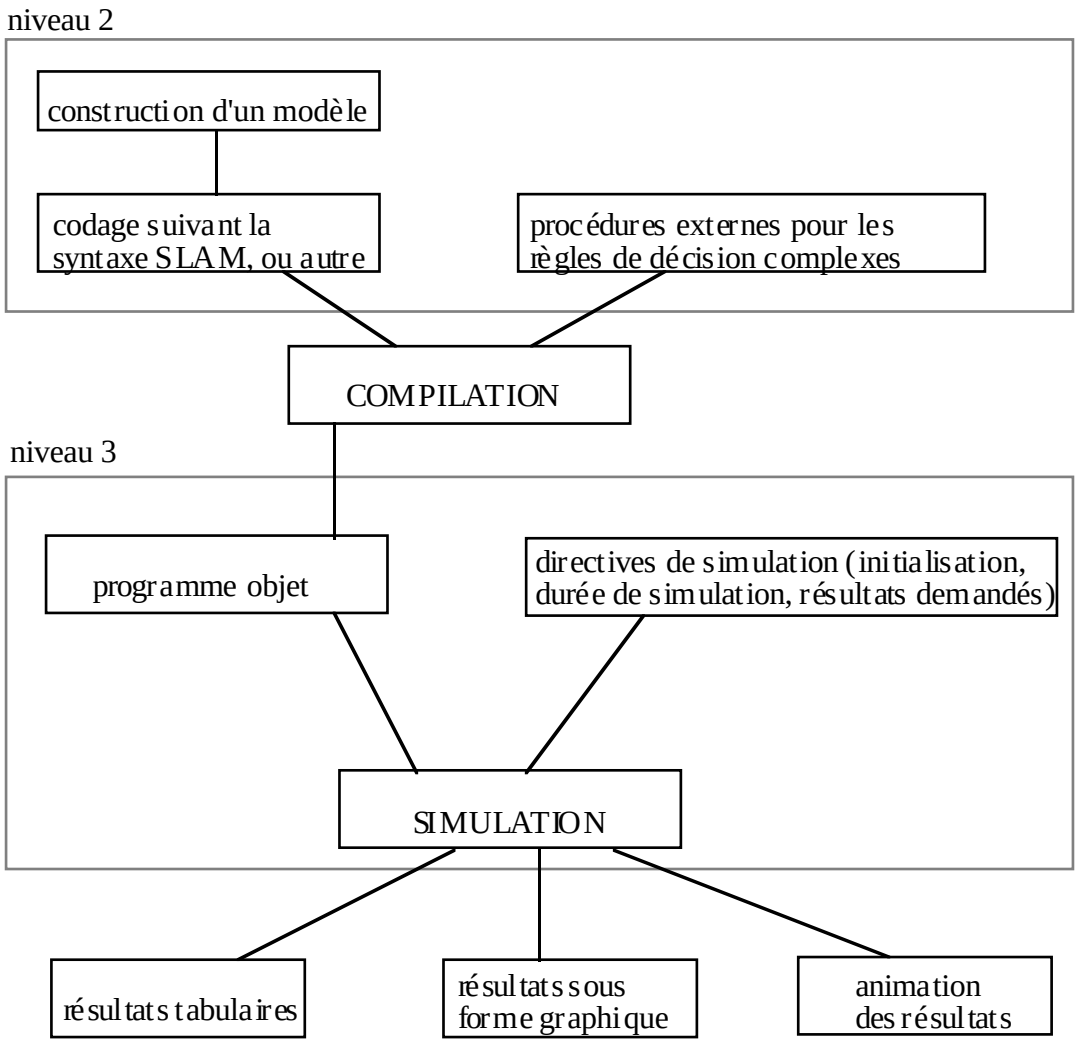


figure I.5
le cheminement usuel de la simulation en utilisant un langage de simulation

La figure I.5 montre aussi les deux derniers niveaux d'utilisation : le niveau 2 qui consiste à l'utilisation du langage SLAM dans la construction du modèle et le niveau 3 qui consiste à ajuster certains paramètres et à lancer la simulation. Pour le même exemple, les utilisateurs du niveau 1 sont les constructeurs qui ont programmé le langage SLAM.

Ceci montre que les produits de simulation basés sur un schéma semblable à celui de la figure I.5 ne répondent pas aux caractéristiques du paragraphe I.3.

Dans la suite, nous allons présenter brièvement cinq langages différents de simulation. Chacun de ces langages possède ses propres qualités et ses propres inconvénients, mais ils présentent tous les mêmes inconvénients en regard de nos caractéristiques.

1) GPSS [Hen88] [Ken87]

GPSS est l'un des premiers langages de simulation discrète entièrement "self contained". En GPSS le modèle est construit en combinant un ensemble de blocs standards (processus pré programmés) en un bloc diagramme (ou réseau) qui définit la structure logique du système. Les objets dynamiques (des transactions) se déplacent séquentiellement de bloc à bloc au fur et à mesure que la simulation se déroule. GPSS était écrit en assembleur IBM et l'horloge ne peut prendre que des valeurs entières.

2) SLAM (Simulation Language for Alternative Modelling) [Pri79]

SLAM est basé sur le FORTRAN. Avec ce langage, la représentation du système se fait sous la forme d'un réseau constitué de symboles (noeuds) interconnectés (branches), à travers lequel circulent des entités. Celles-ci peuvent avoir jusqu'à 100 attributs chacune, permettant de représenter des pièces diversifiées, de dater des temps de passage, de repérer des lots. Les ressources du système sont les éléments nécessaires à la circulation des entités dans le système tels que machines, moyens de manipulation ou de transfert des pièces, personnel de l'atelier...

SLAM fournit donc une structure de travail pour modéliser le flux des entités qui s'engagent dans un processus qui exige un certain nombre de ressources.

Un réseau SLAM est formé de noeuds spécialisés pour modifier les attributs des ressources, des files d'attente, des attentes de ressources et de branches utilisées pour modéliser les activités (par exemple le temps d'usinage) et les cheminements des entités dans le réseau. Les décisions peuvent être modélisées soit par des noeuds spécialisés soit par des conditions ou probabilités associées aux activités. A chaque symbole est associée une primitive permettant la mise en oeuvre de la simulation. Les objets (pièces, chariots) s'écoulent dans ce réseau au cours de l'avance du temps.

SLAM combine la facilité d'utilisation des descriptions par processus et la souplesse de la description par événements.

3) SIMAN (SIMulation and ANalysis program) [Ped82]

SIMAN est un langage très proche de SLAM incluant des possibilités supplémentaires de modélisation des systèmes de transport et de description de stations regroupant plusieurs machines.

4) PAWS (Performance Analyst's Workbench System) [SYS]

PAWS, écrit en FORTRAN lui aussi, est un langage utilisant une description par réseau combinant différents processus pré programmés plutôt adaptés aux systèmes informatiques.

5) QNAP (Queuing Network Analysis Package) [Pot84]

QNAP est un produit destiné à analyser les réseaux de file d'attente. La description du système doit donc se faire sous forme d'un réseau constitué d'un ensemble de processus du type "stations de service" auxquelles sont associées des files d'attente. Des "clients" circulent dans le réseau. Les lois de services dans les stations (fonctionnement particulier du processus) sont écrites par l'utilisateur dans un langage spécifique du type Algol ou Pascal.

QNAP offre ensuite le choix pour l'évaluation des performances entre la simulation et des méthodes numériques exactes ou approchées qui ne peuvent être utilisées que si le modèle remplit certaines conditions.

6) Autres

Il existe d'autres produits de simulation sur le marché français : CADENCE, WITNESS, EXTEND, SIMFACTORY.

Tout les langages et tout les produits de simulation présentés, à l'exception d'Extend, sont tous des produits fermés au niveau de l'utilisateur final, c'est-à-dire ne dépassent pas la paramétrisation d'objets préétablis. Une extension partielle est permise par certains progiciels au travers d'un langage informatique type BASIC, PASCAL ou C.

Par contre l'ergonomie et la connaissance métier incorporées aux produits leaders est importante. Et certains d'entre eux possèdent des possibilités de visualisation très intéressantes.

En résumé, la plupart de ces produits ont une bonne maturité, mais sont bâties sur une architecture logicielle et conceptuelle dépassée.

I.4.4 Logiciels basés sur les Réseaux de Petri

[Bra83] [Jen81] [Mar87]

A la différence des simulateurs présentés ci-dessus, les simulateurs basés sur les Réseaux de Petri (RdP) permettent une analyse du modèle pour vérifier la cohérence par rapport au système physique. Nous présentons ci-après des simulateurs développés autour de modèles RdP :

1) LORIC [Leo85] :est un simulateur à événements discrets écrit en Maclisp.

2) SEDRIC [Val85] :est basé sur les RdP colorés.

3) SICLOP [Ben85].

1) Limites

L'inconvénient majeur des RdP concerne la représentation des données. En effet, un RdP décrit la structure de contrôle mais pas la structure de données. Il définit l'enchaînement des opérations effectuées sur les données mais pas ces opérations elles-mêmes.

Toutes les extensions traditionnelles des réseaux de Petri ne se prêtent pas à la représentation des données, citons quelques unes :

1) RdP synchronisés : ils permettent de tenir compte des événements externes.

2) RdP temporisés : ils permettent de décrire un système dont le fonctionnement dépend du temps. Ils sont utiles pour l'évaluation des performances d'un système.

...

2) Mariage avec les "objets" [Val]

Les réseaux de Petri à Objets [Sib85] constituent un pas très important pour introduire l'aspect données aux RdP. Les trois points suivants présentent une brève définition de ces RdP :

1) les jetons sont des objets appartenant à des classes.

2) les opérations liées aux transitions sont des méthodes concernant les jetons.

3) les filtres du franchissement d'une transition permettent de choisir certains jetons.

Tous les essais de mariage entre les réseaux de Petri et les langages orientés objets n'ont pas eu le succès souhaité. En effet, l'outil résultant du mariage perd certains atouts très

importants des LOO comme l'uniformité de représentation et par conséquence l'héritage (comme en témoigne la méthode HOOD) et la modularité dans certains cas.

3) points faibles

L'utilisation des réseaux de Petri a mis en évidence ses lacunes, à savoir :

1) Les graphes des RdP présentent une mise à plat impraticable pour des systèmes complexes. Les RdP ne connaissent pas l'encapsulation. Certaines extensions permettent de regrouper un ensemble des places en une "macro-place". Ce type de regroupement ne peut être considéré comme une propriété d'encapsulation, en effet il ajoute une nouvelle entité "macro-place" qui n'a pas les mêmes caractéristiques que l'entité place. De même, le regroupement des macros-places ne crée pas une macro-place.

2) Les RdP à objets n'ont pas réussi à surmonter ce handicap parce que, les classes d'objets ne sont pas connues dès le début de la conception. C'est seulement une fois la complexité du système mise en évidence que les classes d'objets sont déduites (pliage du réseau), permettant ainsi, une description plus compacte et plus structurée.

I.4.5 EXTEND : un progiciel de simulation moderne

EXTEND permet de définir des blocs à l'aide d'un langage (pseudo basic), appelé MODL, et de dessiner une icône correspondant au bloc défini. Chaque bloc possède un ensemble d'entrées (bornes blanches) et un ensemble de sorties (bornes noires). La définition du bloc consiste en un programme MODL comportant la mise à jour des sorties en fonction des entrées et des paramètres liés au bloc. Ce calcul doit être fait chaque fois que le temps passe au prochain événement.

Une fois construits tous les blocs nécessaires pour une modélisation, il suffit ensuite de placer ces différents blocs (en n'utilisant que la souris) et lier les entrées des uns aux sorties des autres. Ainsi il permet d'associer à chaque bloc une fenêtre de dialogue permettant la saisie de certains paramètres avant de lancer la simulation.

EXTEND est un produit extensible, il permet d'ajouter facilement un bloc et d'en supprimer un (couper-coller), mais il ne sait ni composer ni décomposer. Les blocs définis en EXTEND sont réutilisables dans d'autres modèles.

EXTEND est très limité en nombre de blocs acceptés dans une modélisation et il ne tourne que sur MacIntosh. (exemple figure I.6).

Il constitue néanmoins ce qui se rapproche le plus selon nous d'une implantation ergonomiquement efficace.

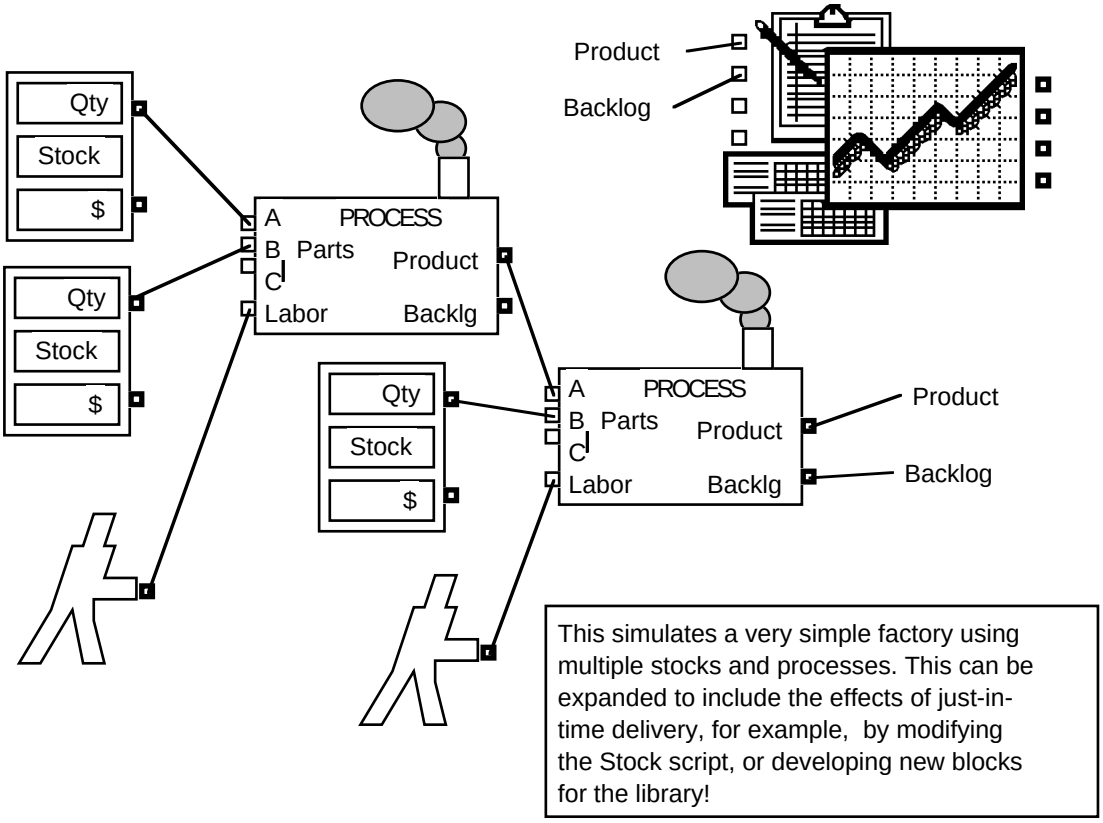


figure I.6 exemple d'un modèle EXTEND

I.5 Les outils généraux

Le sous chapitre précédent a exploré le développement des produits de la simulation. Cette exploration a mis en évidence les lacunes de ces produits vis-à-vis des caractéristiques espérées dans le sous chapitre I.3. On peut dès lors se tourner vers d'autres outils, qui ne sont fonciérement pas destinés à la simulation, mais qui présentent des caractéristiques intéressantes pour cet objet.

On trouve dans cette classe des langages informatiques au premier rang desquels SIMULA, le père des langages dits objets qui, et ce n'est pas un hasard, vient du domaine qui nous intéresse. Le premier descendant de SIMULA fut SMALLTALK, développé chez Xerox pour modéliser l'interaction homme-machine et reprise depuis par le MacIntosh d'Apple, Alan Kay ayant le premier noté les similitudes entre ses deux problématiques [Kay69]. Le reproche qu'on peut faire à ces deux langages, c'est qu'ils se sont intéressés aux

critères évoqués plus haut mais uniquement pour le programmeur (niveau 2). Les concepts de base sont néanmoins essentiels pour comprendre la genèse de BOCAL et s'est pourquoi le chapître III leur est entièrement consacré.

On peut dire qu'ils ne prennent pas en compte le critère d'extensibilité au niveau de l'utilisateur (niveau 3) et que celui-ci ne verra pas de différences entre un simulateur écrit en SIMULA ou en tout autre langage.

A l'opposé, on trouve des produits très intéressants, les "interface builder", qui permettent de maquetter des applications interactives sans référence spéciale à la simulation et dont nous parlerons plus longuement au chapître suivant car on les rencontre plus volontier dans l'activité de maquettage.

En conclusion on peut remarquer que tous les travaux sur ce qu'on peut appeler l'informatique interactive qui ont rejoint les travaux sur la sécurité et la réutilisabilité permettent de se rapprocher de ce que nous avons défini comme spécification. Simplement toutes ces qualités sont dispersés, justement parce qu'il n'existe pas actuellement de base conceptuelle unifiante et que le fossé n'est pas comblé entre la manipulation de "contrôles" sur un écran et la programmation "pure et dure".

I.6 Le simulateur BOCAL

Il est un peu présomptueux de penser que BOCAL va combler cette lacune et définir un mode de représentation des "objets" programmatiques révolutionnaires. Peut-être, et cela ne serait déjà pas si mal, constitue-t-il une avancée dans cette voie.

L'ambition de BOCAL, ce qui en constitue l'innovation majeure et bien évidemment l'incertitude la plus grande, est de permettre aux utilisateurs des trois niveaux dont nous avons largement parlé de fonctionner dans le même environnement sémantique ou, pour le dire plus simplement en utilisant le même langage de programmation.

Les avantages d'une telle approche sont évidents : permettre à tous les utilisateurs de bénéficier de la modularité et de l'extensibilité.

Concrètement, cela permet par exemple au modélisateur de créer un nouvel objet simplement en regroupant un sous-système. Il peut ainsi définir un poste de travail par un certain nombre de machines, de stocks de transport reliés entre eux et réutiliser immédiatement ailleurs dans le système ce poste, devenu à son tour une machine complexe. Il peut remplacer un modèle de sous atelier par un modèle de sous-traitance, ou affiner suivant les besoins la représentation d'un sous-ensemble sans toucher au modèle global.

De la même manière, l'utilisateur du modèle pourra ajouter à sa convenance tout objet d'observation comme l'évolution d'un stock dans le temps, le temps d'occupation d'une machine. Il pourra aussi simplement faire tourner tout ou partie du système, le connecter à des capteurs d'information extérieurs, etc..

Qui dit unité sémantique ne dit pas forcément unité syntaxique. BOCAL sera nécessairement doté pour les utilisateurs de niveau 2 et 3 d'un environnement graphique permettant une programmation visuelle. Mais il n'est pas sûr que les utilisateurs de niveau 1 habitués aux langages classiques ne trouvent pas plus efficace une syntaxe textuelle. Cela n'empêchera pas qu'ils travaillent à partir des mêmes concepts et que les outils de validation et compilation soient identiques.

I.7 Conclusion

Les logiciels de simulation sont souvent partagés entre deux caractéristiques importantes : les logiciels extensibles (ouverts) mais pas assez conviviaux et les logiciels conviviaux mais fermés pour les utilisateurs. Ce qui caractérise tous les progiciels, c'est que les divers utilisateurs de tous les niveaux utilisent des langages et des interfaces distinctes et ils sont tous fermés au niveau de l'utilisateur final (non spécialiste de la programmation).

Cependant la plupart de ces produits ont atteint une bonne maturité, mais sont bâtis sur une architecture logicielle et conceptuelle dépassée.

Afin d'atteindre les objectifs de modularité et d'extensibilité, BOCAL doit s'appuier sur un modèle sémantique unique. Autrement dit à tous les niveaux d'utilisation du logiciel on utilise le même langage de programmation. BOCAL doit permettre de décrire et de modéliser les aspects enchaînement d'opérations (parallélisme et synchronisation), proches de l'atelier, de la machine, et de nature séquentielle. Il procède d'une triple filiation :

- Les outils et les langages de spécifications pour la description des process.
- L'approche orientée objet pour la modularité, l'extensibilité et la réutilisabilité.
- Les générateurs d'interface pour la programmation visuelle

Chapitre II

Spécification et maquettage d'automates et d'applications interactives

BOCAL a en fait une double origine. Les premières spécifications sont issues de l'expérience de la société ALMA, qui a piloté cette thèse, dans le domaine de la planification de production. Mais, devant la complexité des problèmes de pilotage d'atelier, nous avons dans un second temps réorienté notre recherche sur la simulation d'automates "grand public" grâce à une collaboration d'ALMA avec la société DIADEME, spécialiste du domaine. Cette "simplification" technique nous a permis de déboucher sur la première syntaxe de BOCAL, qui nous a permis par la suite de revenir à nos objectifs initiaux.

Dans ce chapitre, on présente les problèmes souvent rencontrés lors de la spécification et du maquettage d'un nouveau produit. On y étudie également les outils de spécification et les logiciels qui offrent une aide à la spécification et à la réalisation des maquettes d'automates grand public.

La réalisation d'une maquette d'un téléphone portatif nous permettra d'étudier et de comparer les différents outils couramment utilisés dans ce but. Cet exemple aidera aussi à définir les caractéristiques d'un tel atelier de maquettage.

BOCAL vise à fournir un "atelier de maquettage". Cet atelier permettra aux différents intervenants à la spécification d'un nouveau produit la réalisation d'une maquette par l'assemblage des différents constituants de la maquette et d'apporter eux-mêmes toutes les transformations nécessaires.

II.1 Problématique

Le passage d'une idée de produit issue d'un service marketing à la spécification technique de ce produit constitue indubitablement une étape cruciale lors d'un nouveau développement : automate bancaire, radio téléphone, pompe à essence... En effet, comme l'exprime le schéma suivant, les outils mis en oeuvre demandent souvent un travail redondant, notamment lorsqu'il s'agit de concilier les aspects ergonomiques concrétisés par une maquette, et techniques décrits par une spécification plus formelle.

L'une des difficultés majeures non surmontées aujourd'hui tient à la diversité des intervenants concernés par la phase de spécifications. Citons, sans chercher à être exhaustif : le responsable produit marketing, un représentant de l'utilisateur final ou du client, le chargé du développement, un ergonome...

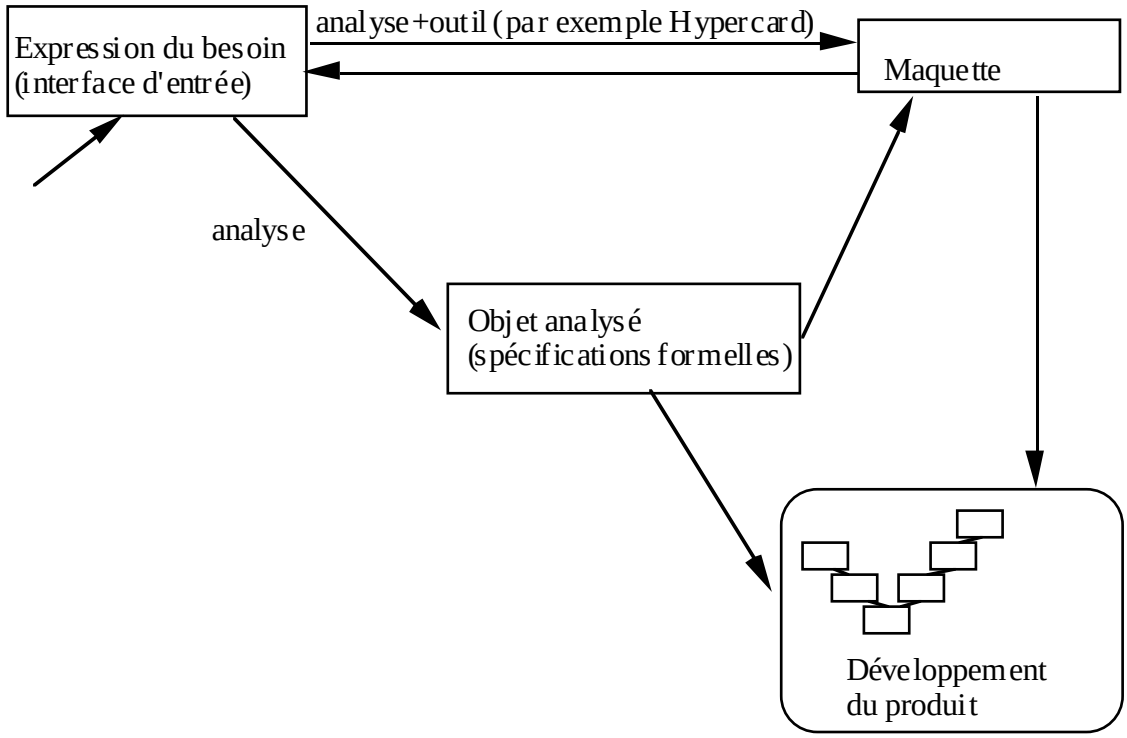


figure II.1
les différentes étapes dans le développement d'un produit

Pratiquement, les personnes chargées d'analyser le projet ne bénéficient pas de l'analyse déjà réalisée pour l'élaboration de la maquette. Au mieux, elles la renouvellent, au pire, les deux analyses sont incohérentes.

Voici un exemple des spécifications d'un téléphone portatif réalisées par des différents intervenants :

Le responsable marketing écrit : "*Le téléphone doit pouvoir entrer en communication en frappant huit chiffres au clavier puis envoi, ou bien en frappant "MEM", une lettre (entre A et J) puis envoi. D'autre part, on doit pouvoir stocker des numéros dans les mémoires*".

L'ergonome ajoute, par souci de cohérence : "*Le stockage des numéros en mémoire s'effectuera en frappant la touche "MEM", puis une lettre, puis huit chiffres et enfin "VAL" pour valider*".

Plus tard le client demande "la possibilité de revenir en arrière sur le dernier chiffre frappé afin de pouvoir corriger une erreur".

Les interventions répétées du "maquetteur" pour réaliser les transformations de la maquette à la demande du client nous conduisent à l'idée d'un "atelier de maquettage" complet permettant au client d'assembler les différents constituants de la maquette et de réaliser lui-même toutes les transformations.

II.2 Les outils de maquettage d'automates

II.2.1 Les applications de prototypage

Cette famille rassemble les outils de prototypage d'applications informatiques. Comme leur nom l'indique, ces outils sont plus particulièrement dédiés au développement de logiciels (boîtes de dialogues, menus déroulants....).

On trouve dans cette famille les "interface builder" comme HyperCard sur Macintosh, VisualBasic sur PC ou Interface Builder sur Next. L'Interface Builder de Next, basé sur un langage orienté objet (Objective-C), offre une palette des briques élémentaires (boutons, menus déroulant, écrans d'affichage, etc...) et permet facilement l'extensibilité de ces objets, mais cette facilité reste à la portée des programmeurs.

1) Exemple du téléphone

La réalisation d'une maquette de téléphone par l'Interface Builder sur Next nous a permis de découvrir les avantages et les inconvénients de cette interface de prototypage. En premier temps, l'Interface Builder a permis rapidement de dessiner l'architecture physique du téléphone. En deuxième temps, l'Interface Builder a permis de donner vie aux composantes physiques de la maquette en les équipant d'une logique de fonctionnement grâce aux classes et aux méthodes à écrire en Objective-C (bien sûr il faut connaître le langage Objective-C).

La figure suivante montre la maquette des composantes du téléphone, cette maquette ne montre pas les communications entre l'écran et les claviers, elle cache la logique du fonctionnement de l'appareil.

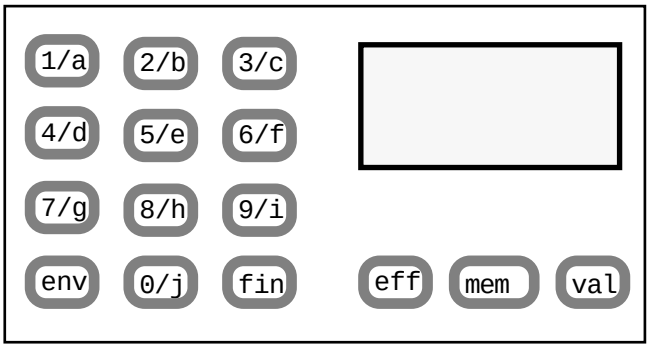


figure II.2
maquette du téléphone réalisée sur Next

2) Qualités

Ces outils sont souvent utilisés pour leurs qualités suivantes :

1) couplage des aspects graphiques et programmatiques,
2) facilité d'interfaçage avec les matériels périphériques (micro, haut-parleur, écran tactile, reconnaissance vocale...),
3) extensibilité du langage par des routines externes en C (ou autres).

3) Points faibles

L'universalité grandissante de ce type d'outils leur fait perdre peu à peu leur simplicité d'utilisation, et par leur manque de contraintes, ne permettent pas de construire à coup sûr une maquette facilement modifiable et réutilisable.

De plus, les "briques" de base de ces outils ne correspondent pas aux besoins spécifiques des automates de dialogues, et la réutilisation de morceaux déjà développés est rendue très difficile par le type de structures mis en place. Enfin, ces structures internes ne permettent pas à une personne qui n'aurait pas participé au développement de la maquette de modifier celle-ci aisément.

En résumé, voici les difficultés souvent rencontrées lors de l'utilisation de ces outils :

1) difficulté de représenter des sous-états,

2) impossibilité de créer des objets (notamment par regroupement) autres que les types pré définis, donc pas de réutilisabilité des composants,

3) différence de nature des interventions à la portée de l'utilisateur (le fabriquant d'appareils télécommunication a accès à quelques paramètres et aux libellés des messages textuels), du maquetteur (programmation en HyperCard/HyperTalk ou en VisualBasic) et du programmeur (Apple ou MicroSoft ont écrit ces langages en C ou en Pascal),

4) lourdeur dans la modification du design des objets (touches, forme générale du terminal).

II.2.2 Outils de description fonctionnelle d'automates

Ce sont des outils qui présentent une aide importante pour définir les fonctionnalités de l'automatisme et n'aident pas particulièrement à la conception de l'architecture physique du produit. En général, ces outils fournissent des diagrammes descriptifs qui sont traduits en programmes (en utilisant un langage général ou autre).

Voici une liste des outils (loin d'être exhaustive) couramment utilisés dans la phase fonctionnelle. Il faut noter que ces outils ont été testés sur le même exemple (la spécification d'un téléphone portatif).

1) Diagrammes états-transitions

Les diagrammes états-transitions essaient de représenter l'ensemble des états possibles d'un système, pour chaque état l'ensemble des transitions qui donnent lieu à une action et l'état dans lequel on arrive. Ce formalisme, si on veut l'appliquer avec succès, nécessite donc que le couple (état courant, transition) soit suffisant pour définir le nouvel état, et par conséquent que ce nouvel état ne soit en aucun cas défini par le résultat de l'action qui a suivi la transition.

Il paraît évident que la restriction à un nouvel état défini seulement par l'état courant et la transition qui s'est produite conduit à une mise à plat inacceptable, au moins visuellement de tous les états possibles. Ainsi, deux états très voisins resteront deux états, et il est très difficile de "parallèliser" les états.

Cependant cette approche est intéressante car c'est la seule qui permette une représentation non équivoque, complète et unique du fonctionnement réel de la machine.

On peut donc penser que des "structures topologiques" de familles d'états apparaissent dans cette représentation et que ces structures permettent d'aboutir à la meilleure représentation possible des états de la machine. Certaines de ces caractéristiques sont étudiées ci-dessous.

On peut essayer de compacter plusieurs états en un seul, par exemple les états :

- possédant les mêmes entrées
- possédant les mêmes sorties
- et qui sont connexes.

En dehors des points cités ici, il paraît évident que cette approche n'est pas exploitable, du moins pour sa représentation. On peut toutefois penser qu'elle permette de définir une structure interne de définition de spécifications car elle est cohérente et non ambiguë.

2) Méthode S.A.R.T [Hat91] [Hop79]

SART est une méthode de spécification des systèmes temps réel. Elle permet de construire deux modèles d'une machine :

- le modèle des besoins
- le modèle d'architecture.

Le modèle des besoins consiste en des "diagrammes de flots de données" (DFD), des "spécifications de traitement", des "diagrammes de flots de contrôle" (DFC), des "spécifications de contrôle", des "spécifications de contraintes de temps" et un "dictionnaire des besoins".

A chaque diagramme de flots de données qui montre les processus (traitements) du système et les flots d'information circulant entre ces processus, est associé un diagramme de flots de contrôle possédant les mêmes processus mais montrant les signaux de contrôle circulant entre ces processus, et une spécification de contrôle utilisant les signaux de contrôle pour fabriquer les "contrôles de processus" qui activent ou désactivent les traitements. C'est dans ces spécifications de contrôle que l'on retrouve les diagrammes états-transitions et les automates à états finis.

On construit ces diagrammes par raffinement successifs. A chaque couche, on trouve un DFD, un DFC et une spécification de contrôle. Chaque processus utilisé dans les DFD et DFC fait l'objet d'une sous couche jusqu'à ce qu'un processus s'exprime de façon triviale en pseudo code dans une spécification de traitement.

Le modèle d'architecture a pour but d'attribuer les différents éléments du modèle des besoins (processus, spécifications de contrôles, données, contrôles) à des unités matérielles et logicielles qui composeront le système. Il intègre les éventuelles contraintes technologiques. Il est constitué de diagrammes de flots d'architecture, de spécifications de modules d'architecture, de diagrammes d'interconnexion d'architecture et d'un dictionnaire d'architecture.

SART est une méthode qui sépare effectivement les notions de données et de contrôles. En revanche, en ce qui concerne les contrôles, elle privilégie l'aspect processus et les différents états de la machine (qui peuvent entre autres être caractérisés par l'ensemble des processus actifs et les variables de contrôle) n'apparaissent pas clairement. Ils sont définis dans les spécifications de contrôle, sans lien apparent avec les diagrammes de flots.

De plus, la séparation des données et des contrôles n'est qu'apparente puisque des spécifications de traitements peuvent générer des contrôles. Ceci permet effectivement la communication entre les deux couches contrôles et données en permettant des "contrôles par les données", mais nuit fortement à la compréhension de l'ensemble.

En revanche, le formalisme utilisé met clairement en évidence les données et les contrôles qui doivent être mémorisés contrairement à ceux qui sont générés et consommés de suite.

SART possède pourtant une caractéristique intéressante : la séparation entre "données" et "contrôles" est une loi de séparation : une information de nature continue, ou de nature discrète pouvant prendre un grand nombre de valeurs est toujours une donnée. Par contre une information de nature discrète et ne pouvant prendre qu'un nombre restreint de valeurs est souvent un contrôle.

En conclusion, l'approche mise en oeuvre dans SART possède les inconvénients majeurs suivants :

1) elle est hiérarchique et globalement descendante, elle demande donc d'avoir dès le début une "bonne" vision globale de la machine que l'on veut spécifier et que l'on va raffiner peu à peu et ne fournit pas la possibilité de remodeler facilement les couches supérieures en fonction de l'avancement du raffinement.

2) la notion fondamentale d'état de la machine n'apparaît que de façon très dispersée dans toutes les spécifications de contrôle, et de plus, les liens entre les différents états et les processus activés ne sont pas mis en valeur.

3) la vision dynamique de la machine en fonctionnement n'apparaît pas de façon visuelle.

3) Grafcet [All89]

Le grafcet est destiné à représenter des automatismes logiques, c'est-à-dire des systèmes dans lesquels les informations ont un caractère "tout ou rien". C'est un outil de spécification qui décrit uniquement la fonction à réaliser, i.e l'automate au sens mathématique, indépendamment de toute technologie, de toute réalisation.

C'est un graphe biparti qui comporte deux types de noeuds, les étapes et les transitions (un grafcet contient au moins une étape et une transition). Des arcs orientés relient soit une étape à une transition, soit une transition à une étape. On associe à chaque étape des actions (à niveau ou impulsionnelle) et à chaque transition une réceptivité (le produit d'un événement et d'une condition). Chaque étape peut avoir deux états actif ou inactif. Les entrées de l'automatisme sont associées aux transitions, tandis que ses sorties sont associées aux étapes.

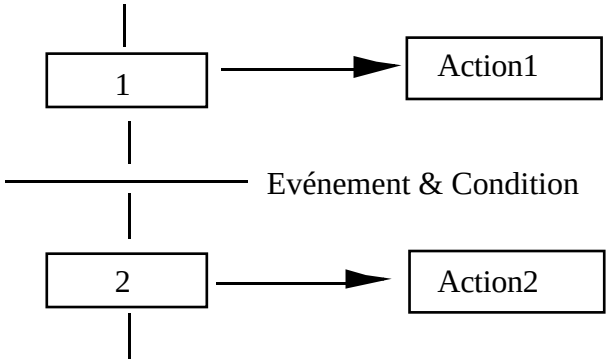


figure II.3
étapes et transitions

L'ensemble des étapes actives à un instant donné définit l'état du système. L'évolution de l'état se fait par franchissement de transitions. Une transition est franchissable si et seulement si :

1) Toutes les étapes qui précèdent la transition sont actives (on dit que la transition est validée).

2) La réceptivité de la transition est vraie.

Le franchissement d'une transition consiste à désactiver toutes les étapes en amont de la transition et à activer les étapes en aval.

Caractéristiques et remarques

Quand on décrit des systèmes complexes, la taille des graphes peut s'accroître de façon telle qu'ils deviennent difficiles à élaborer, puis à comprendre, à corriger et à mettre à jour. Par exemple : La prise en compte des sécurités, notamment, est une raison importante de l'accroissement de la complexité.

Les principales qualités du grafcet :

1) Le grafcet peut décrire n'importe quel comportement entrées/sorties, si ces entrées et sorties sont des variables discrètes indépendamment de toute réalisation.

2) Description facile du parallélisme : le grafcet accepte de franchir plusieurs transitions franchissables simultanément d'où plusieurs étapes actives simultanément.

3) Le grafcet permet les rendez-vous entre processus parallèles.

Le grafcet a introduit la notion de macro-étape et la notion de macro-action dans le but de diminuer la taille grandissant du grafcet et d'avoir une meilleure lisibilité. Le conditionnement d'une transition par l'état interne d'une étape aide le spécificateur à alléger la description des systèmes complexes.

Ces notions utilisés pour diminuer la taille de la description augmentent l'illisibilité du grafcet en cachant un certain nombre des liens et n'augmentent pas la puissance de spécification.

4) Réseaux de Petri [All89]

Un réseau de Petri est un graphe biparti entre deux ensembles de noeuds : places et transitions. Chaque Place contient un nombre entier de marques ou jetons. Le marquage du réseau, M, est défini par le vecteur de ces marques.

Les réseaux de Petri colorés donnent des couleurs aux jetons et associent à chaque transition un ensemble des couleurs, chaque couleur indique une possibilité distincte de franchissement. Les arcs relient une place à une transition ou une transition à une place. Le poids d'un arc est une fonction pré ou post qui établit une correspondance entre chaque couleur de la transition et les couleurs de la place (amont ou aval suivant le sens de l'arc).

Caractéristiques et remarques

Le réseau de Petri est un outil très intéressant pour la représentation des systèmes à événements discrets. Les transitions correspondent aux événements, les places aux activités et aux états d'attente et les suites de places et de transitions à des structures de processus séquentiels instanciées chaque fois qu'elles sont traversées par un jeton.

Une des caractéristiques importantes des réseaux de Petri est de pouvoir présenter graphiquement certaines relations : parallélisme, synchronisation (rendez-vous, sémaphore), partage de ressource, mémorisation, ...

Ce qui caractérise également les réseaux de Petri vis-à-vis d'autres outils de spécification, c'est que leur comportement dynamique peut être décrit par un système d'équations linéaires en nombres entiers déduit de leur matrice d'incidence. En conséquence, un certain nombre de propriétés caractéristiques (blocages, conflits,...) peuvent être calculées par des algorithmes de programmation linéaire. C'est ce que l'on appelle l'analyse des réseaux de Petri.

Le paragraphe I.4.4 montre l'importance des Réseaux de Petri dans le domaine de production et cite des différents simulateurs basés sur cet outil. Un simulateur RdP peut être utilisé soit pour parfaire une validation commencée par l'analyse formelle des propriétés, soit pour passer directement des spécifications d'une conception à un codage.

Les réseaux de Petri présentent les mêmes inconvénients que le grafcet. Mais beaucoup d'efforts ont été fait pour combler ces inconvénients. Toutefois, la croissance de la taille des RdP les rend illisibles et très difficiles à corriger.

II.3 BOCAL dans le domaine du maquettage

Nous avons noté les similitudes des problèmes rencontrés dans le domaine de la simulation des systèmes de production et dans le domaine de maquettage d'automates, à savoir :

1) la complexité croissante avec la taille du système.

2) les logiciels utilisés sont des systèmes fermés au niveau d'un utilisateur non spécialisé.

3) la notion d'encapsulation est totalement inconnue. En général, ces logiciels permettent une analyse descendante, mais pas de programmation remontante par composition.

II.3.1 Spécifications

BOCAL, tel qu'il est défini dans les chapîtres suivants, permet une analyse descendante de la maquette jusqu'à l'arrivée à des composantes déjà définies pour la réalisation d'autres maquettes et permet aussi la composition de ces composantes pour la construction de la maquette. Cette construction dynamique de la maquette ne présuppose pas une vue d'ensemble du produit. Cette vue d'ensemble sera construite au fur et à mesure de l'avancement. Cette propriété a été un point de repère constant dans l'élaboration de BOCAL.

La programmation BOCAL permettra la description fonctionnelle et structurelle des composantes constituant la maquette.

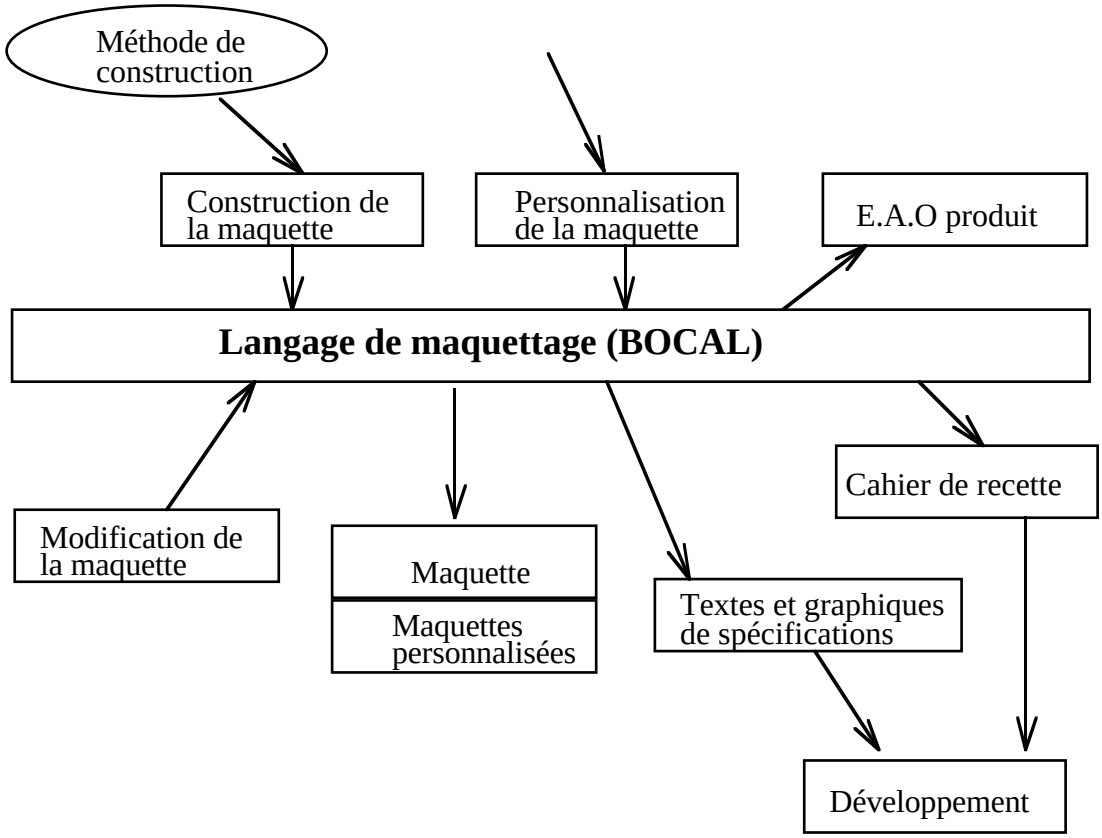


figure II.4
le cycle de vie d'un produit

La figure II.4 considère que la phase de maquettage comme une phase à part entière du développement d'un produit. Elle l'intégre à ce titre dans le cycle de vie et définit les liens concrets entre cette phase et les autres.

BOCAL permettra de développer une "bibliothèque de maquettage" contenant les briques élémentaires dans toute construction d'une maquette, à savoir : des composantes d'interaction (boutons, menus déroulants, champs de saisie, etc...), des composantes sensorielles (sons, textes, inscriptions diverses), des composantes permettant la production des documents d'aide à la spécification et du cahier de charge, etc... L'interface graphique de BOCAL permettra d'associer des représentations parlantes pour les gens du métier à ces composantes de base.

II.3.2 Bénéfices

L'objectif de notre outil est de mieux intégrer la phase de maquettage au développement d'un nouveau produit, et d'en tirer tous les bénéfices potentiels.

Ces bénéfices relèvent de différents ordres :

1) des atouts marketing :

- aide à la formulation des spécifications fonctionnelles du produit,
- aide à la création de son interface homme-machine,

2) des atouts techniques :

- aide à la rédaction détaillée des spécifications,
- aide à la robustesse et à la complétude des spécifications,
- aide à la recette du produit final,

3) des atouts commerciaux :

- aide à l'avant-vente par une personnalisation aisée de la maquette produite,
- aide à la formation au nouveau produit par l'utilisation de la maquette dans un environnement d'E.A.O.

II.4 Conclusion

La définition du langage (BOCAL) s'appuie sur les notions de synchronisation et de parallélisme(Grafcet et Réseaux de Petri) et sur les notions de modularité, extensibilité et réutilisabilité qu'on trouve plus particulièrement dans les langages orientés objets.

Un nouvel outil de spécification devrait, pour apporter un réel mieux aux utilisateurs, posséder les qualités d'extensibilité et encapsulation. C'est le but que nous avons pousuivi dans l'élaboration de BOCAL. Nous verrons que BOCAL ne sépare pas la description physique de la description de la logique du fonctionnement. Par rapport aux outils de spécifications traditionnels comme Grafcet et RdPC qui ne s'occupent que de la logique de fonctionnement en décrivant l'enchaînement des opérations et la synchronisation entre elles, cet aspect devrait lui attirer l'intérêt des utilisateurs. Toutefois, la traduction du grafcet ou du RdPC vers BOCAL reste possible.

Pour que BOCAL devienne un véritable outil de maquettage un certain nombre de travaux complémentaires et spécifiques sur les aspects interactifs et les composants sensoriels seront nécessaires. Cela dépasse largement le cadre de cette thèse qui avait d'abord pour but de définir un ensemble sémantique cohérent et de l'illustrer sur quelques cas simples. Il s'agit donc plutôt de voies à explorer, pour lesquelles nous sommes convaincus que les principes fondamentaux de BOCAL constituent une bonne base.

Chapitre III
Langages orientés objets

Ce chapitre est consacré à la représentation des langages orientés objets. L'approche orientée objet a fait son apparition pour répondre aux besoins des développeurs face à la complexité croissante des problèmes abordés. Cette approche vise à ce que les programmes soient faciles à tester, à améliorer, à réutiliser et à maintenir.

On présente brièvement les principaux mécanismes d'abstraction utilisés par cette approche : classes, héritage,... Grâce à ces principes l'approche orientée objet offre des qualités intéressantes de programmation.

Les langages orientés objets présentent une aide précieuse pour la modélisation de l'interaction homme-machine qui par essence manipule des objets plus que des fonctions. Cependant, ces langages à objets restent destinés aux professionnels de la programmation.

BOCAL tente d'adopter ces mécanismes d'abstraction afin de conserver de bonnes qualités de programmation. Mais BOCAL allège tout de même ces mécanismes, pour garder une syntaxe facile à comprendre et proche de l'utilisateur. BOCAL est prévu pour fonctionner dans un environnement de programmation dynamique où il n'y a plus de différence entre la phase programmation et la phase exécution, un peu comme dans un tableur actuel.

III.1 Introduction au concept d'objet

[Mey90] [Nie88] [Nyg86] [Str67] [Mas90]

La taille des programmes, la réécriture du code et les problèmes de maintenance ont mis en évidence depuis longtemps la nécessité de promouvoir de nouveaux concepts : l'abstraction de données, la modularité et la réutilisabilité des logiciels. Beaucoup de langages ont été développés dans cette direction et les langages "objets" constituent sans doute la réponse la plus appropriée à ces problèmes.

L'approche orientée objet favorise une modélisation de l'univers par abstraction de données. Elle est basée sur des mécanismes d'abstraction à savoir : encapsulation, protection des données, polymorphisme, généricité et héritage.

Contrairement à la programmation structurelle, la programmation objet propose d'organiser l'univers d'une application en termes d'objets plutôt qu'en termes de procédures. Toutefois, elle n'exclut pas pour autant toute composante algorithmique : les méthodes se définissent de la même façon que les procédures.

Les principes de cette approche offrent toutes les qualités pour mettre en oeuvre des méthodes de programmation rigoureuses, tout en préservant la souplesse et la convivialité du cadre de programmation :

III.1.1 Abstraction de données et encapsulation [Sny86]

Le concept d'objet repose sur le principe d'encapsulation : l'objet regroupe des données et les procédures qui les manipulent. Les communications entre objets se font à travers une interface : envoi de messages. Les objets protègent leurs données et cachent l'organisation interne.

Un objet apparaît donc comme une boîte noire. Un objet "client" n'a pas besoin de connaître l'implantation et la représentation interne d'un autre objet pour lui adresser un message.

L'interface d'un objet est parfois considérée comme un contrat de type client-fournisseur [Mey88], les clients de l'objet lui adressent des requêtes pour certains services qu'il garantit savoir fournir.

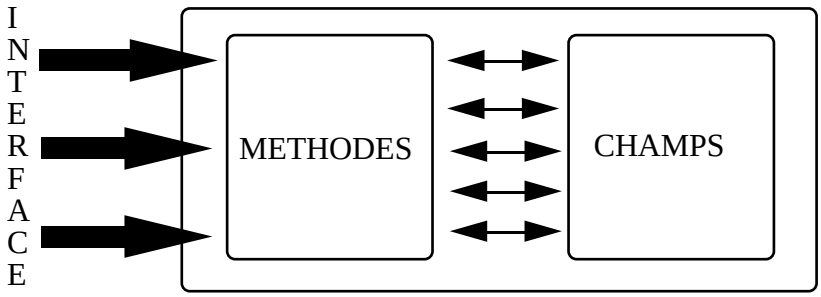


figure III.1 le concept d'objet

III.1.2 L'héritage [Lie86] [Sny86]

La classe doit être considérée comme un réservoir de connaissances à partir duquel il est possible de définir d'autres classes plus spécifiques, complétant les connaissances de leur classe supérieure. Les connaissances les plus générales sont ainsi mises en commun dans des classes qui sont ensuite spécialisées par définitions de sous-classes successives, contenant des connaissances de plus en plus spécifiques. Il s'agit d'un problème de partage efficace de connaissances.

La classe **OBJECT** est la racine de tout graphe orienté d'héritage et définit le comportement commun de tous les objets (instanciation, initialisation, duplication, destruction, ...). Ainsi, les programmeurs pourront étendre ou modifier ces fonctionnalités dans les classes descendantes.

L'héritage permet de factoriser des connaissances pour créer de nouveaux objets, par spécialisation ou extension des objets préexistants. Ce mécanisme repose sur deux composantes :

1) une composante statique, c'est la relation qui lie un objet à celui ou ceux dont il hérite. Elle est représentée par le graphe d'héritage et possède une sémantique.

2) une composante dynamique (liaison dynamique) : c'est le processus qui réalise l'héritage, en donnant aux objets l'accès aux informations dont ils héritent [Duc89]. Plus précisément, la liaison dynamique cherche dynamiquement (au moment de l'envoi de message) la méthode correspondant à un message reçu dans la classe de l'objet concerné et dans ses superclasses.

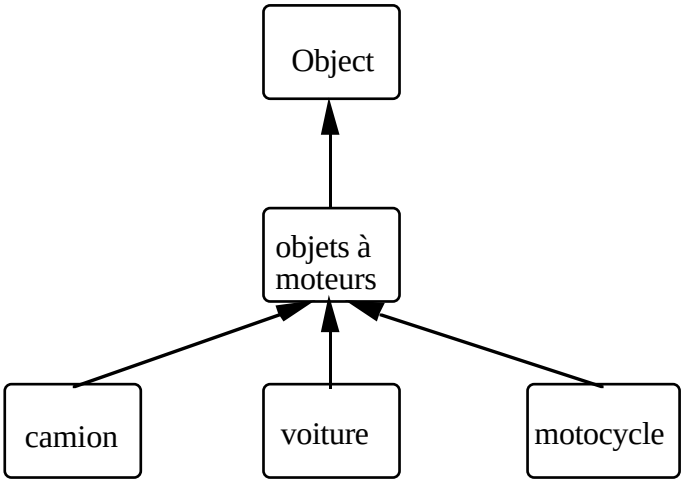


figure III.2
Factorisation des connaissances dans une hiérarchie d'héritage simple

L'héritage simple exige que chaque classe ne possède qu'une superclasse directe et la relation d'héritage est représentée par une arborescence. Par contre, l'héritage multiple permet à une classe d'avoir plusieurs superclasses directes. L'héritage multiple implique que l'ensemble des classes n'est plus structuré en arborescence mais forme un graphe orienté sans circuit. Une classe hérite de l'union des variables et des méthodes de ses superclasses. Ainsi, la classe "voiture" hérite de la classe "objets de luxe" les informations relatives à son prix, mais hérite aussi de la classe "objets à moteurs" les informations concernant son moteur. L'avantage de l'héritage multiple est d'accroître encore la modularité des programmes, et donc d'en faciliter la mise au point et la maintenance. En contrepartie, la définition des classes exige beaucoup d'effort pour qu'elles soient réutilisables.

Un bon partage de connaissances en une hiérarchie des classes (héritage simple ou héritage multiple) suppose que le programmeur ait une parfaite maîtrise du monde à modéliser ce qui n' est généralement pas le cas. En plus, les classes intermédiaires (Object, objets à moteurs, moyens de transport et objets de luxe), appelées "classes abstraites", ne donnent pas naissance à des objets réels même si leur existence sert à factoriser les données et les comportements de la classe voiture en plusieurs classes et à préciser les spécificités des voitures dans leur classe (voir figure III.3).

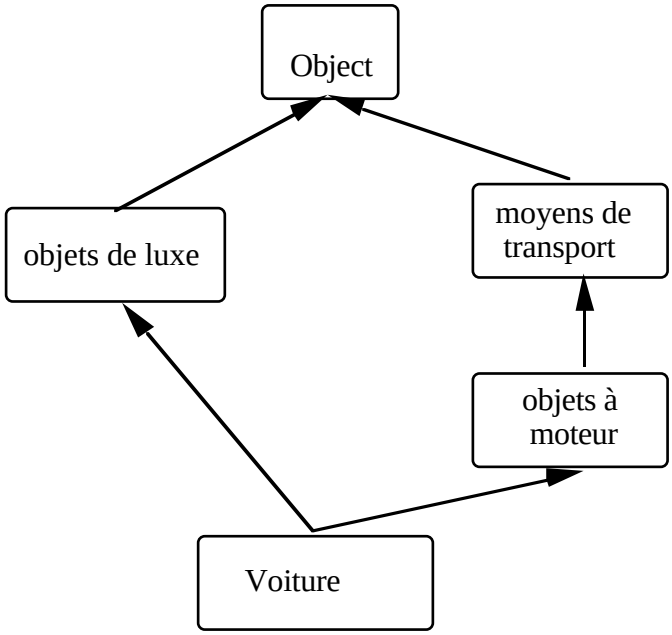


figure III.3
Héritage multiple et problème de conflits

L'héritage multiple pose un problème de conflits. Ainsi une classe, héritant de plusieurs superclasses, ne saura pas de quelle superclasse il héritera une méthode commune à deux ou plus de ses superclasses. La relation d'héritage multiple est une relation d'ordre partiel. Le conflit se pose entre deux classes non liées par cette relation qui ont un descendant commun. Certains conflits ne sont pas du ressort du mécanisme d'héritage parce qu'ils résultent de fautes de conception. Ce sont typiquement des problèmes de collision de noms : les propriétés en conflits portent bien le même nom, mais leur sémantique est différente [Mas90].

La plupart des langages orientés objets, qui adoptent l'héritage multiple, laissent aux programmeurs le soin de résoudre les fautes de conception et permettent l'extension de la relation d'héritage en relation d'ordre total pour résoudre les conflits des propriétés.

III.1.3 L'envoi de message et la liaison dynamique

Conceptuellement, le contrôle dans les langages à objets est assuré par des communications entre les objets, et non plus par des appels de procédures. Pratiquement, un envoi de message est en fait un appel de procédure "distante". La différence essentielle entre un envoi de message et un appel de procédure est que la méthode à appliquer est déterminée à l'exécution.

Ainsi, chaque objet interprète les transmissions qu'il reçoit dans le contexte de ses champs et de ses méthodes. Un même identificateur peut donc désigner des actions sémantiques différentes suivant le type du receveur. Si la méthode "dessiner" est définie pour la classe des objets géométriques alors un cercle interprétera ce message différemment d'un rectangle.

III.1.4 Modularité

C'est le principe de la réutilisabilité [Mey87], qui favorise le prototypage et la programmation incrémentale, surtout avec un environnement de programmation perfectionné, comme celui de SmallTalk-80.

Les langages à objets offrent une modularité par différenciation ou spécialisation (grâce au concept d'héritage). Par exemple, la définition de la classe des chats décrit les attributs et les comportements qui caractérisent les chats par rapport à la classe des animaux.

Les langages à objets sont un progrès dans ce sens : ils permettent de constituer des bibliothèques de classes facilement réutilisables dans différents contextes.

III.1.5 Homogénité

L'approche orientée objet présente un modèle rigoureux uniforme : toutes les entités manipulées sont des objets et la seule structure de contrôle est l'envoi de message [Hew77]. L'interaction avec le système en devient particulièrement simple. Ce principe favorise la convivialité du système et l'évolutivité des logiciels.

III.1.6 Parallélisme

De nombreux efforts sont faits pour créer des langages de programmation parallèle à partir des langages classiques tels que Pascal [Han79]. L'approche orientée objet permet, d'une manière générale, une programmation parallèle explicite car des processus fonctionnant concurremment peuvent être facilement modélisés par des objets communiquant par envoi de messages [Agh86][Nie87], comme en témoigne le premier langage à objets, Simula, qui a été conçu pour gérer et synchroniser des coroutines.

III.4 Présentation des certains langages à objets

III.4.1 Simula

Simula [Dah65] a introduit les principaux concepts de la programmation objet. Simula, comme son nom l'indique, intègre des primitives spécialisées pour la simulation, permettant de gérer la synchronisation entre plusieurs coroutines. En particulier, un objet est considéré comme un programme actif autonome, pouvant communiquer et se synchroniser avec d'autres objets. Il existe aussi une bibliothèque de classes prédéfinies, la plupart de ces classes sont dédiées aux tâches système et aux tâches de simulation.

Il nous semble étonnant que les langages à objets qui ont succédé Simula ne se soient pas intéressés pour autant à la simulation.

III.4.2 Eiffel

Eiffel est un langage à part entière et n'est pas construit à partir d'un autre langage comme c'est le cas pour C++ et Objective-C. Mais pour des raisons d'efficacité et de portabilité, le compilateur Eiffel produit du code C qui est ensuite traité par un compilateur C.

Eiffel [Mey90] est un langage fortement influencé par Simula. Il reflète l'intérêt porté par son concepteur, Bertrand Meyer, aux critères de qualité que le génie logiciel attache aux langages, à savoir l'efficacité, la fiabilité, la réutilisabilité et la portabilité. Il doit être considéré avant tout comme un environnement pour professionnels du génie logiciel, favorisant une spécification rigoureuse du problème et la réutilisation maximale de l'existant.

Eiffel implémente deux mécanismes d'abstraction : l'abstraction verticale correspondant à la notion d'héritage multiple et l'abstraction horizontale correspondant à la notion de généricité. Eiffel permet de définir des classes (qu'on appelle des classes génériques) paramétrisées par d'autres classes. Par exemple la classe LINKED_LIST[T] est paramétrisée par la classe T, cette classe regroupe la classe des listes d'entiers, la classe des listes de réels, ...

En plus de ces caractéristiques classiques, Eiffel propose deux compléments de programmation : les assertions et les exceptions.

Une assertion est une propriété formelle d'une classe qui peut être de trois natures :

1) précondition qui doit être vérifiée avant chaque traitement d'une méthode donnée.

2) postcondition qui doit être vérifiée après chaque traitement d'une méthode donnée.

3) invariant de classe, condition qui doit être remplie à la création de chaque nouvel objet de cette classe et vérifiée à l'exécution en sortie de toutes les méthodes de la classe.

Une exception est une entité déclenchée lorsqu'une erreur se produit à l'exécution. Par exemple lorsqu'une assertion n'est plus vérifiée. Les deux instructions d'exception suivantes sont possibles : "Rescue" et "Retry". "Rescue" introduit la séquence de traitement à réaliser lorsque l'exception a été déclenchée. "Retry" relance l'exécution de la méthode qui a déclenché l'exception.

Cette vérification sémantique avant et après l'exécution d'une méthode nous semble être une propriété très intéressante. Dans le domaine de la simulation, ces conditions peuvent jouer un rôle assez important pour réaliser des contrôles et des synchronisations entre activités. Malheureusement, Eiffel est trop rigoureux pour être destiné aux utilisateurs de tout niveau.

III.4.3 Smalltalk [Gol83]

Le langage SmallTalk est le père spirituel des langages à objets qui implémente toutes leurs caractéristiques : encapsulation, polymorphisme, héritage simple, liaison dynamique.

SmallTalk se distingue de la plupart des autres langages à objets par le fait qu'il propose un modèle uniforme de programmation. Toutes les entités du système sont des objets, que ce soient les nombres, les chaînes de caractères, des entités plus complexes comme les fenêtres, les contrôleurs et les processus, ou encore les nouveaux objets définis pour une application particulière. Toute opération, qu'il s'agisse d'imprimer un fichier, de tester l'égalité de deux nombres ou d'évaluer des expressions arithmétiques, est réalisée grâce à une structure de contrôle unique, l'envoi de message.

III.4.4 Objective-C

Brad J. Cox a conçu Objective-C, dont l'intention première était d'ajouter au langage C les principales caractéristiques de SmallTalk-80 [Cox83]. Comme tout langage construit à partir d'un langage hôte, Objective-C n'est pas complètement unifié : les objets Objective-C, de type id, présentent les entités classiques de C, qui gardent leur comportement habituel. De même, les types classiques de C ne sont pas intégrés à la hiérarchie des classes

Objective-C. Par exemple, une variable de type entier n'est pas considérée comme une instance de la classe Integer.

Il est ainsi possible de réconcilier efficacité et puissance d'abstraction, en tirant profit à la fois de la couche objet et des constructions du langage C. Le programmeur a toujours le choix de représenter une entité par une structure C ou par une instance d'une classe.

Objective-C a servi à définir une bibliothèque de classes permettant la création d'interfaces dans un environnement multifenêtres. La conception de cette bibliothèque est fondé sur l'idée des logiciels "**circuit intégré**". Cette idée, inventée par Brad J. Cox [Cox86], se base sur la conception des modules de logiciel aussi aisément réutilisables que les circuits électroniques intégrés, en les munissant d'interfaces soigneusement spécifiées.

Cependant, cette idée des logiciels "circuit intégré" subit le même sort que les logiciels de simulation. En s'approchant de l'utilisateur, ces logiciels ont perdu certaines qualités : extensibilité, encapsulation.

En effet, l'assemblage d'un circuit intégré se fait toujours à plat, à partir des circuits préexistants. La construction d'un circuit par composition des circuits prédéfinis ne le rend pas utilisable pour la construction de nouveaux circuits. Donc, on perd le principe d'encapsulation.

De plus, l'extensibilité de ces composantes électroniques passe toujours par des méthodes réservées aux programmeurs.

III.4.5 C++ [Str86]

C++ est considéré comme une extension du langage C pour implanter les classes de Simula. La classe C++ est une généralisation de la structure C. C++ est devenu un langage populaire, particulièrement dans le monde UNIX grâce à sa compatibilité avec le langage C. Il est par exemple très facile d'installer C++ sur un site UNIX, et de l'utiliser avec les outils de l'environnement de programmation C, comme le débogueur symbolique dbx. Il est aussi facile d'écrire des bibliothèques de classes pour des tâches précises, telles que la simulation. Ces classes ont beaucoup d'intérêt, mais sont loin d'être destinées aux utilisateurs.

III.5 Les interfaces interactives orientées objets

L'approche orientée objet permet la modélisation des composantes nécessaires pour chaque application interactive comme : les fenêtres, icônes, textes, etc..., en objets et

classes. Le comportement dynamique de ces composantes est assuré par un système d'échange de messages suivant un protocole de communication afin d'activer des méthodes, par exemple ouvrir ou fermer une fenêtre. Les interfaces utilisateurs orientées objets offrent des bibliothèques de composantes afin de permettre aux programmeurs un prototypage rapide de leurs applications interactives.

L'avantage majeur de l'approche orientée objet dans le développement interfaces utilisateurs réside dans la clarté de la décomposition structurelle et fonctionnelle ce qui rend les interfaces faciles à développer, à maintenir et à réutiliser. La relation d'héritage permet de définir de nouvelles composantes comme extension ou spécialisation d'autres composantes existantes. Toutefois, ces avantages restent à la portée des programmeurs.

III.5.1 Les applications interactives en Smalltalk

L'environnement de programmation SmallTalk intègre dans son système : l'ensemble des outils, éditeur graphique, éditeur de textes, débogueur, browsers et gestionnaire de fenêtres. Les objets composant cet environnement sont créés à partir de classes spécifiquement prévues à cette fin et sont bien entendu activés par envoi de message. Ceci permet à l'utilisateur la création d'un environnement de travail personnalisé et le développement des applications interactives, en réutilisant et en enrichissant l'environnement prédéfini.

Le schéma de développement d'applications interactives proposé par SmallTalk est appelé le système Modèle-Vue-Contrôleur (MVC) [Gol84][Kra88]. Il repose sur trois composantes :

1) le modèle, qui représente les données manipulées dans l'application,

2) la vue des données, qui est présentée dans un ensemble de fenêtres sur le terminal et qui constitue l'interface visuelle entre l'utilisateur et les données,

3) le contrôle des données, qui constitue l'interface permettant à l'utilisateur d'agir sur les données, à l'aide du clavier et de la souris.

Ces trois composantes forment un triplet "MVC". Le contrôleur "C" traite les commandes utilisateur et communique avec le modèle "M" et la vue "V". Les communications entre ces trois composantes sont assurées par envoi de messages.

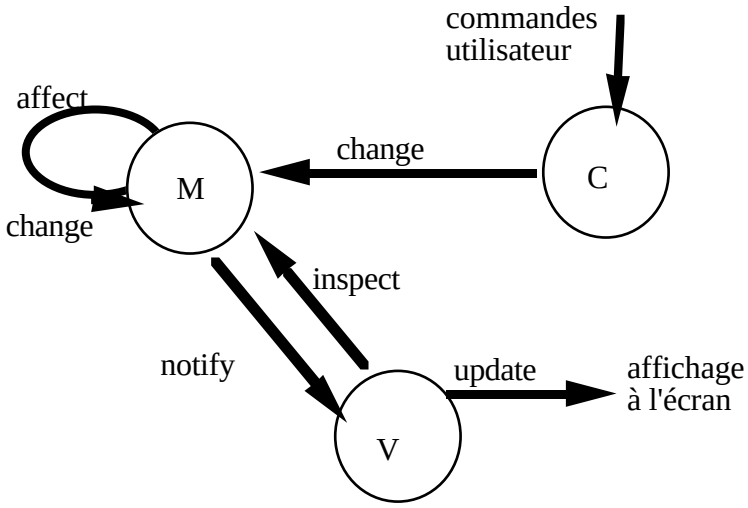


figure III.4
Le protocole VMC de SmallTalk-80

En effet, SmallTalk tient à jour un dictionnaire d'objets (c'est une variable globale) qui exprime les dépendances entre objets. La classe OBJECT définit toutes les méthodes qui manipulent ce dictionnaire. Cette relation de dépendance permet à un objet de confier la responsabilité de certains messages aux objets dont il dépend. Ces liens de dépendance sont réalisés dynamiquement par le programmeur selon ses besoins. Par exemple lorsque l'opérateur appuie sur un bouton d'une application SmallTalk, le bouton confie à un autre objet la responsabilité de répondre en activant une de ces méthodes.

Le système SmallTalk propose donc toute une gamme prédéfinie de classes permettant de créer des fenêtres et des contrôleurs adaptés à différentes manières de présenter des données.

III.5.2 InterViews sous le système X11 Window

InterViews est une bibliothèque des classes C++ permettant la construction des interfaces utilisateurs sous le système X11 Window. InterViews s'appuie sur quatre composantes pour construire des interfaces utilisateurs :

1) Sujet : une collection d'objets représentant les données manipulées dans l'application.

2) Interaction : une collection d'objets interactifs réalisant l'interface visuelle entre l'utilisateur et les données. Cette composante possède deux sous composantes "Sensor" et "Painter" pour assurer le comportement interactif.

3) "Sensor" : une collection d'objets détectant les réactions de l'utilisateur (clavier, souris).

4) "Painter" : une collection d'objets permettant l'affichage des sorties.

En fait, cette logique de composition est similaire à celle du SmallTalk : l'application est séparée de la présentation visuelle, une composante de contrôle assure la cohèrence.

III.5.3 Remarques

Ces interfaces interactives ne sont pas à la portée des utilisateurs non spécialistes de la programmation :

1) Le contrôle des objets d'interaction (fenêtres, boutons, menus déroulants, textes,....) se font par des méthodes définies dans des classes. La définition d'une méthode reste une tâche réservée aux programmeurs.

2) L'utilisateur final n'a pas l'accès aux avantages de l'approche orientée objet : les objets manipulés graphiquement par l'utilisateur ne sont pas autonomes et leur comportement est totalement centralisé dans un dictionnaire global de dépendances.

Les "Interfaces Builder" tentent d'aller plus près de l'utilisateur en lui permettant de réaliser la maquette de son application sans taper, éventuellement, une ligne de code (la programmation visuelle). Grâce à une palette d'objets interactifs, l'utilisateur choisit ses objets, puis dessine des liens entre ces objets pour déléguer la responsabilité de répondre aux événements externes à des classes préexistantes en activant certaines méthodes. Malheureusement, l'extension de ces classes n'est pas à sa portée.

III.6 Conclusion

L'approche orientée objet présente un grand intérêt pour le développement des logiciels grâce aux qualités de programmation que cette approche offre aux développeurs. Les programmes sont devenus faciles à tester, à améliorer, à réutiliser et à maintenir.

Les langages orientés objets restent destinés aux spécialistes de la programmation pour les raisons suivantes :

1) La notion d'abstraction et la structuration des classes en hiérarchie d'héritage supposent que le développeur ait une vue globale et fine de l'application. Dans ce sens, l'approche orientée objet n'aide pas le développeur à analyser son problème.

2) L'extensibilité par sous-classes n'est pas accessible aux utilisateurs de tout niveau.

Pour ces diverses raisons, nous essayerons de concevoir BOCAL pour qu'il puisse offrir aux utilisateurs de tout niveau un langage visuel extensible et modulaire.

Chapitre IV
BOCAL : Langage et concepts

On connaît la réplique de l'architecte de la Révolution Française Louis-Etienne Boullée à qui l'on demandait pourquoi un projet qui paraissait si simple lui avait demandé tant de temps : " C'est justement parce qu'il est simple".

Cette anecdote pourrait s'appliquer aux concepts de base de BOCAL, dont la simplicité apparente masque sans doute le travail de réflexion qui fût nécessaire à leur élaboration.

Il était dès lors tentant pour nous de rendre visible ce travail, en retraçant l'historique des divers avatars du langage et de montrer comment se sont peu à peu dégagées les spécifications actuelles. Mais l'histoire des errements ne conduit sans doute pas à l'exposé le plus pédagogique. Alors, avant de présenter en détail au sous-chapître IV.2 ces concepts, nous nous contenterons de les introduire "intuitivement" à partir des objectifs de cette recherche.

IV.1 Introduction aux concepts de BOCAL

IV.1.1 Le point de départ

Le projet BOCAL n'est pas tombé du ciel. Il s'inscrit dans la suite des travaux menés au laboratoire ARTEMIS depuis plus de dix ans sur la planification de production. En particulier, les travaux de M. Gourgeaud, dont la thèse porte sur l'application de l'approche "objet" à la GPAO. Il n'est pas nécessaire ici de retracer les avantages de cette approche, rengaine banale de l'informatique d'aujourd'hui. Disons simplement que ces langages orientés "objet" (LOO) sont nés de la simulation à événements discrets des systèmes physiques (SIMULA) et qu'ils y trouvent un domaine d'application évident. Alors quelles insuffisances présentent ces langages pour nous avoir lancé dans la conception de BOCAL?

Premier principe : Que l'héritage profite à tous

La première insuffisance des LOO tient à leur cible. Ils sont conçus exclusivement pour le programmeur informatique et non pas pour le modélisateur ou l'utilisateur final. Ce qui est intéressant, avec les "objets", c'est par exemple de pouvoir définir un nouveau type de machine, de convoyeur ou de méthode de conduite à partir d'une famille existante, ce

qu'on appelle l'héritage. Or, à moins de modifier le logiciel, ceci est impossible au modélisateur. Dès lors que lui importe que ce logiciel soit écrit en C ou C++, Fortran ou Smalltalk. L'ambition première de BOCAL est que cette propriété fondamentale d'héritage soit partagée par tous.

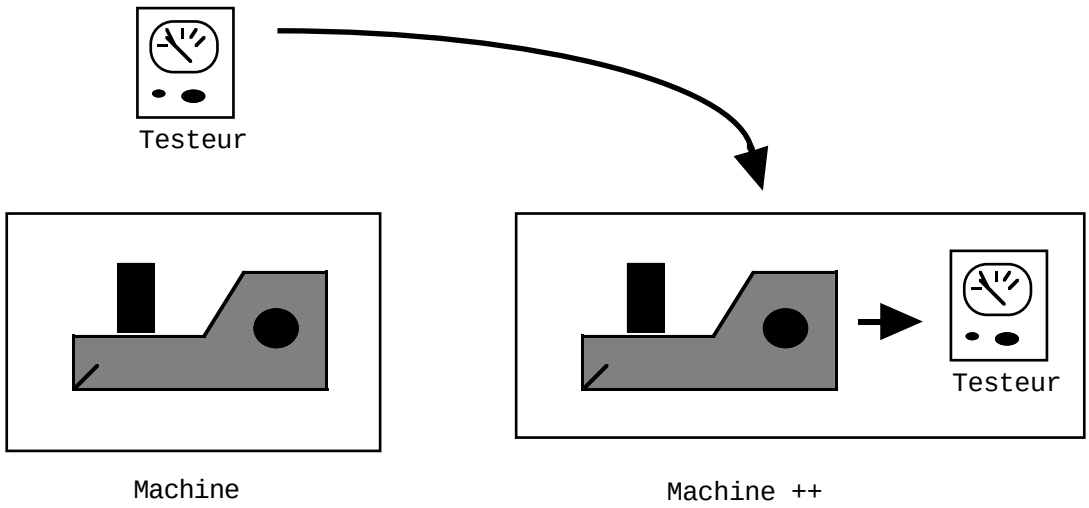


Figure IV.1 L'héritage

Qu'on nous entende bien : nous ne prétendons pas rendre simple ce qui est compliqué, et il est clair que la modélisation de certains processus complexes requiert des qualités "programmatiques" qui ne sont pas données à tous. Mais, si nous arrivons à rendre simplement possible (et facile) ce qui est simple, nous aurions déjà accompli un grand pas par rapport à l'existant, qui est soit fermé et aisé (par exemple on peut choisir entre un certain nombre de type de convoyeurs prédéfinis et paramétrables), soit ouvert et complexe (les langages de programmation).

D'où le premier principe fondamental de BOCAL : offrir un modèle sémantique commun, gage de l'extensibilité du langage, à tous les utilisateurs, du programmeur du noyau à l'utilisateur final. C'est évidemment un pari un peu fou, puisque ceci n'est réalisé par aucun système aujourd'hui mais à quoi servirait la recherche si elle n'explorait des pistes à priori impossibles.

Une conséquence ergonomique de ce premier principe est que BOCAL doit se prêter "naturellement" à une programmation visuelle dans les systèmes de fenêtrage actuels, car ce type de programmation représente la forme syntaxique la plus adaptée (car la moins abstraite) pour l'utilisateur final.

Second principe : L'encapsulation

La réalité (ou tout du moins sa représentation) est hiérarchisée. Une usine est composée d'ateliers. Chaque atelier comprend des opérateurs et des machines. Et ainsi de

suite. Remarquons que l'activité de modélisation consiste souvent à construire cette hiérarchie qui n'est pas toujours aussi simple que ci-dessus. Mais il n'existe pas de sens privilégié. Parfois la conception est descendante. Parfois on procède par regroupement. Les langages orientés "objet" peuvent plus ou moins représenter cette hiérarchie par la notion de classe. Mais la distinction qu'ils établissent tous entre méthodes et objets rend la définition d'une classe par regroupement compliquée. A l'inverse, durant toute la phase de conception de BOCAL, nous fûmes guidés par une image très simple : un utilisateur devant son écran sur lequel étaient représentés des "objets". Par une simple opération de souris, il traçait un rectangle autour de certains objets. Aussitôt ce rectangle se contractait en une nouvelle figure. Un nouveau type d'objet venait d'être créé et pouvait être déplacé et dupliqué.

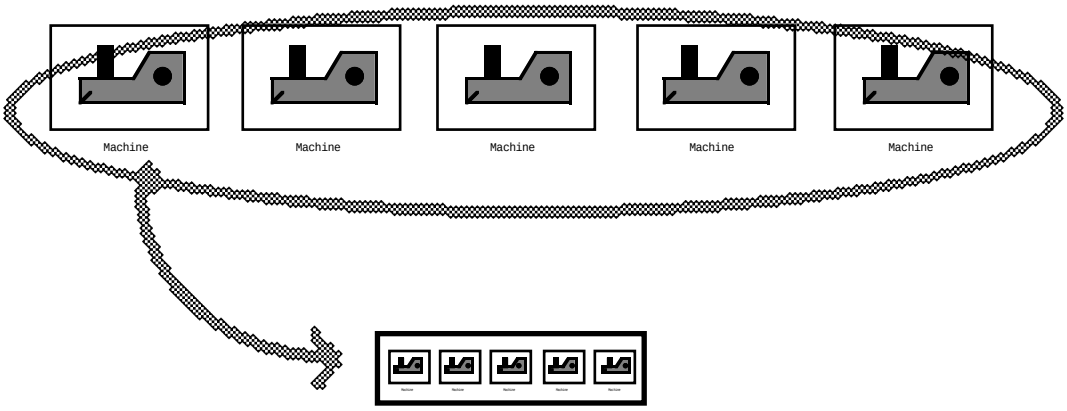


figure IV.2 L'encapsulation

Cette image reste le fil rouge de la spécification de BOCAL. Elle rappelle le bureau du Macintosh, avec ses dossiers. Sauf que ceux-ci ne sont pas des objets programmatiques alors que BOCAL va leur donner vie, les activer, les faire bouger.

IV.1.2 Les boîtes

Le principe d'encapsulation implique pratiquement d'avoir une entité unique en BOCAL, condition d'un regroupement simple. BOCAL appelle **boîte** cette entité unique. Chaque boîte appartient à une famille nommée **modèle**. Une **boîte** est un assemblage d'autres boîtes (qu'on appelle ses **filles**), lesquelles...et ainsi de suite jusqu'aux boîtes dites terminales.

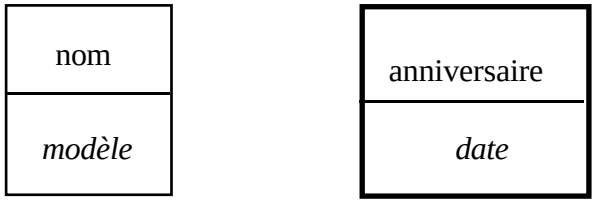


figure IV.3
chaque boîte appartient à un modèle

Si la notion de modèle rappelle celle de classe dans les LOO, celle de boîte recouvre à la fois la notion d'objet et de méthode. On voit par là que BOCAL récuse l'antagonisme objets/fonctions.

IV.1.3 Les alias

Comment faire communiquer une boîte avec l'extérieur. La réponse classique : chaque boîte reçoit et envoie des messages. Ces messages contiennent des ordres et les informations nécessaires à leur exécution. L'implantation d'une telle approche par une programmation visuelle est très difficile. Dans BOCAL cette communication avec l'extérieur est réalisée par deux mécanismes très simples.

Le premier consiste à définir une équivalence entre certaines filles d'une boîte et des filles de sa mère. Ce mécanisme est très similaire au passage d'arguments dans un langage procédural classique.

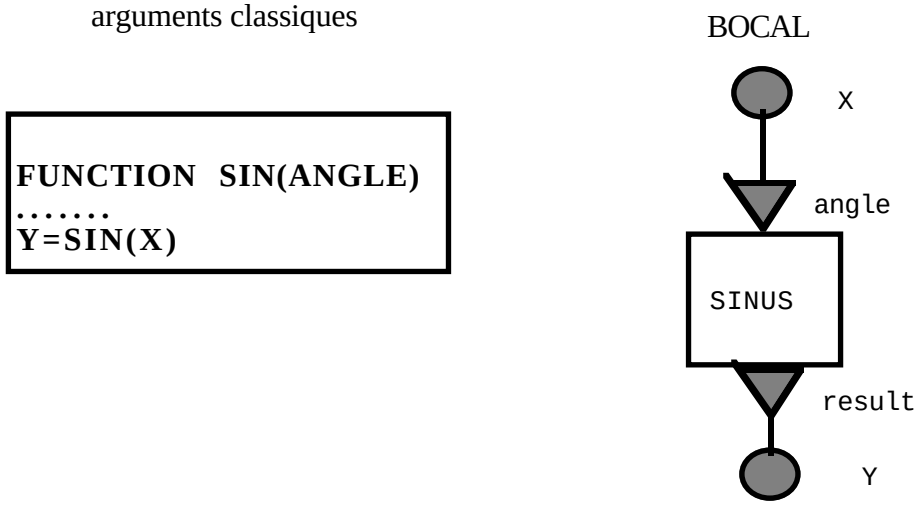


figure IV.4
l'alias en BOCAL

Il existe cependant une différence entre ces deux mécanismes. En BOCAL le lien (**alias**) entre le nom interne et le nom externe d'un argument est explicite. Dans un langage classique, cette correspondance est implicite. A l'appel de la procédure on lui envoie une liste d'arguments qui est identifiée avec la liste des arguments définie dans la déclaration de la procédure.

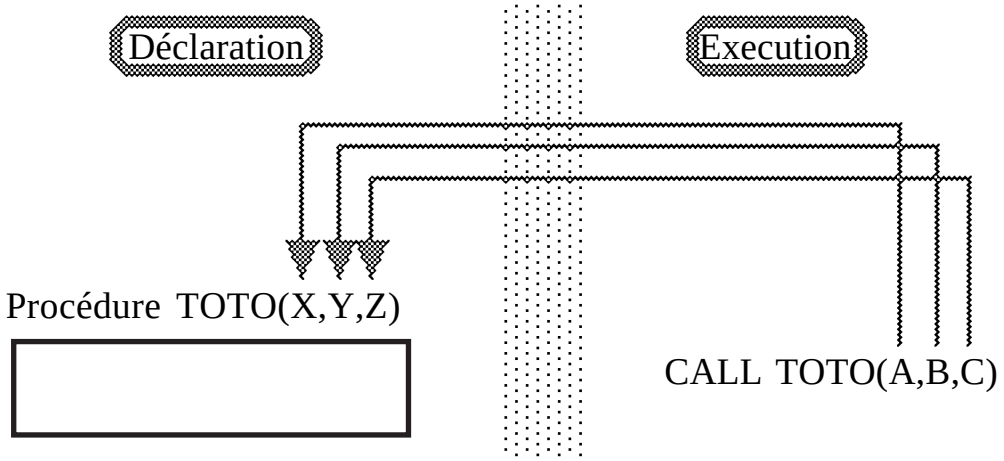


figure IV.5
la correspondance entre arguments de l'appelant et ceux de l'appelé

Le choix d'un lien explicite s'est imposé pour deux raisons. La première est d'ordre ergonomique. Les boîtes de BOCAL seront dessinées en 2D ou 3D. La correspondance implicite suppose un ordre sur les arguments. Or ces espaces ne sont pas ordonnés. Il faudrait donc dessiner les arguments dans un sous-espace de dimension 1 (par exemple un bornier sur le côté gauche de la boîte pour les argument d'entrée et un à droite pour les arguments de sortie).

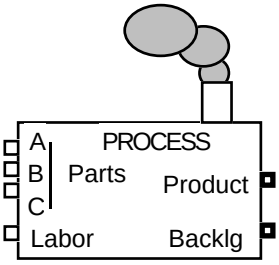


figure IV.6
connexions d'entrées et de sorties

Ceci complique le dessin des connexions. Mais la seconde raison est plus importante. L'usage d'un modèle peut-être différent selon son environnement. Une voiture doit

"exporter" ses pneus devant un gonfleur et son réservoir devant une pompe à essence. La méthode classique par identification implicite conduirait à exporter tous les arguments possibles et à ne connecter que ceux qui sont utilisés dans l'environnement particulier. On imagine sans peine ce qu'une telle méthode impliquerait pour des objets complexes. La notion d'alias explicite permet d'avoir une vision ensembliste et non ordonnée : dans un contexte particulier les boîtes d'interface d'un modèle sont un sous-ensemble de ses boîtes filles.

IV.1.4 Précondition et postcondition

Jusqu'ici notre description des boîtes en font des objets inanimés. On va maintenant les animer en les dotant d'une propriété particulière : l'activation. Une boîte, qui peut-être vue comme un process, sera donc active ou inactive. Il faut dès lors doter les modèles de règles d'activation et d'inactivation. En BOCAL la règle d'activation est la précondition. La règle de désactivation est la postcondition.

Intuitivement on imagine que pour qu'une boîte s'active, il est nécessaire qu'un certain nombre de ressources soient disponibles. BOCAL rejoint complètement l'intuition avec une petite subtilité sur le sens de disponible. Une ressource peut-être disponible si elle est présente (le carburant pour activer le moteur). Mais dans certain cas, c'est l'inverse une ressource est disponible si elle est absente (c'est le cas des emplacements où un process dépose les objets produits). Enfin pour être disponible une ressource ne doit pas être bloquée par une autre boîte qui est en train de l'utiliser.

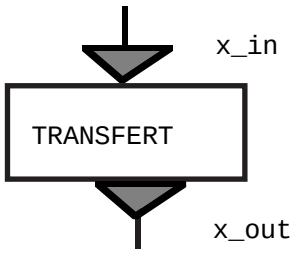


figure IV.7
la précondition de la boîte TRANSFERT

En général une précondition est simplement la liste de ces ressources (avec pour chacune un flag pour dire si on exige sa présence ou son absence). Idem pour la postcondition. Et les deux règles d'animation de BOCAL :

une boîte est activable si

1) sa mère est active

2) sa précondition est vérifiée

une boîte est désactivable si

1) toutes ses filles sont inactives

2) sa postcondition est vérifiée

IV.1.5 Les événements

Bien sûr tout ce qui vient d'être dit sera précisé au paragraphe suivant, mais l'essence de BOCAL se trouve décrit là. Juste un dernier point. On peut s'étonner de ne pas voir apparaître ici la notion d'événement qui est au coeur de tous les langages de simulation et d'interaction. C'est que les événements sont implicites en BOCAL. Les seuls événements qui se produisent en BOCAL sont les modifications de disponibilité de ressources. Dès qu'un tel événement se produit il est automatiquement signalé par le superviseur BOCAL à toutes les boîtes intéressées. Par exemple supposons que plusieurs process attendent une imprimante occupée par un process P. Dès que P "libère" l'imprimante, le superviseur BOCAL parcours l'arborescence des process pour leur signaler la disponibilité de la ressource (imprimante) libèrée. L'un d'eux la bloquera et transmettra cette information à ses filles et ainsi de suite jusqu'à atteindre une boîte d'impression qui pourra enfin s'activer.

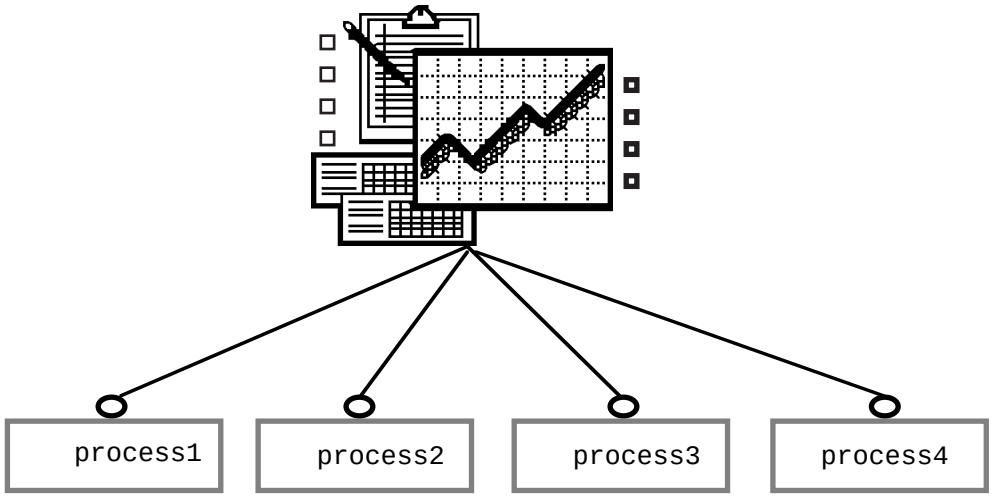


figure IV.8
exemple d'une ressource partagée par plusieurs boîtes

IV.2 Syntaxe et concepts

IV.2.1 Modèles et boîtes

BOCAL se présente comme une collection de **modèles**. Un modèle contient des représentants d'autres modèles. On les appelle des **boîtes**.

On aurait pu appeler **objets** les boîtes si cela ne renvoyait pas à une dualité objet-fonction qui n'apparaît pas dans BOCAL. Il est plus approprié de penser une boîte comme un système ou un process.

Ainsi, une boîte regroupe d'autres boîtes sans distinction entre données et méthodes ou procédures. Une boîte peut donc présenter une fonction, des données ou le regroupement des données et des fonctions.

Chaque boîte porte un identificateur. (Par comparaison avec des langages classiques, le modèle est le type abstrait de la variable et l'identificateur, son nom).

Chacune de ces boîtes s'appelle boîte fille de la boîte modèle qu'on appelle la boîte mère. Tout simplement, la relation mère-fille entre boîtes signifie "englobante-englobée". Parmi ces boîtes filles, on distingue celles qui sont locales des autres qu'on appelle boîtes d'interface. Chaque boîte fille contient elle-même (et selon la déclaration de son modèle) des boîtes filles, lesquelles ....

Les boîtes locales ne peuvent en aucun cas être connues à l'extérieur de la boîte mère. Elles ne peuvent ni être exportées ni être importées. Les boîtes locales correspondent dans un sens aux variables locales des langages classiques et les boîtes non-locales ressemblent aux arguments d'une fonction ou aux composantes d'une structure. Toutefois les arguments sont ici potentiels. C'est suivant l'usage qu'il voudra faire d'une boîte que l'utilisateur exportera ou non un argument. Alors que dans les langages classiques les arguments se présentent comme une liste, dans BOCAL ils constituent un sous-ensemble des boîtes non-locales.

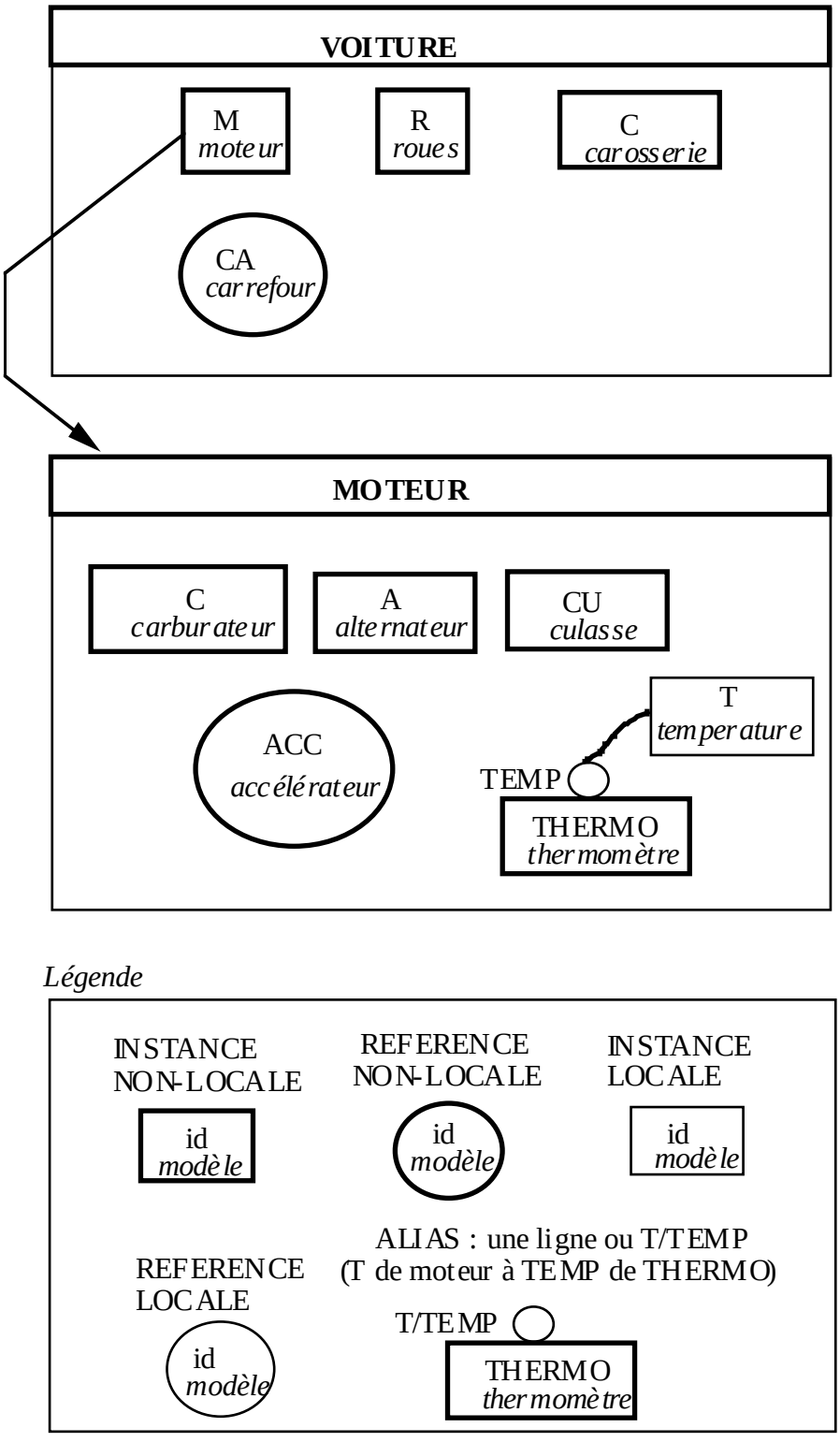


figure IV.9

Exemple d'un modèle voiture

Les boîtes filles se divisent en deux groupes : Les boîtes **instances** (représentées par des carrés) et les boîtes **références** (représentées par des ronds). Les boîtes instances se présentent comme des composantes physiques de la boîte mère (ces sont : les roues et la carrosserie d'une voiture, l'écran et le clavier d'un ordinateur ou encore les machines dans

un atelier). Les boîtes références se présentent comme les arguments d'une fonction. Ces sont les boîtes importées ou exportées vers le monde extérieur (voir figure IV.9).

En résumé, il existe quatre types de boîtes :

Référence locale
Référence non locale
Instance locale
Instance non locale

La distinction entre boîtes instances et boîtes références a des conséquences sur l'opération de transfert. Quand une voiture se déplace, son moteur, sa carrosserie et ses roues se déplacent avec la voiture, par contre l'état du feu, qui est une fille référence dans le modèle voiture, ne se déplace pas avec la voiture. On remarquera que le transfert d'une boîte est une opération récursive sur les boîtes filles instances.

IV.2.2 Les alias

Un modèle déclare un ensemble des boîtes (qu'il contient) et un ensemble d'**alias**. Un alias entre deux boîtes indique que ces deux boîtes distinctes correspondent en fait au même objet. Dans un langage classique, ceci se fait implicitement lors d'un passage d'argument par le rang dans la liste d'arguments (le 3ème argument de l'appelé est identique au 3ème argument de l'appelant). En BOCAL ces liens sont explicites et constituent l'outil unique de communication entre boîtes.

EXEMPLE: Atelier

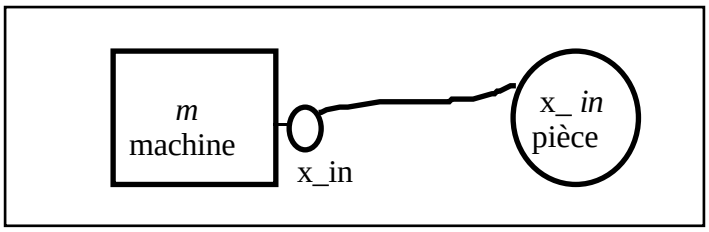


figure IV.10 exemple d'alias

Le modèle atelier (figure IV.10) déclare qu'il possède une machine et reçoit une pièce à l'entrée. Le modèle de la machine exige une pièce qui est en alias avec la boîte nommée "x_in" de l'atelier. Cet alias signifie que si une pièce entre dans l'atelier, elle est dans la machine aussi parce que les deux boîtes ("x_in" de la machine et "x_in" de l'atelier) représentent le même objet. Attention, cet alias ne constitue pas un transfert de l'entrée de l'atelier vers l'entrée de la machine mais une identification physique.

Puisque les jeux d'alias font qu'un objet est représenté par plusieurs boîtes il faut déterminer la boîte qui contient vraiment cet objet. Chacune de ces boîtes se trouve dans une boîte mère différente. On a souligné dans le paragraphe précédent la différence entre les boîtes instances et les boîtes références (une boîte instance est une composante physique dans la boîte mère par contre une boîte référence n'appartient pas réellement à la boîte mère). D'où la règle suivante :

l'ensemble des boîtes reliées par un jeu d'alias ne doit pas contenir plus d'une boîte instance.

Une boîte référence, par définition, n'appartient pas réellement à la boîte mère. Grâce aux alias, chaque boîte référence appartient : soit au monde extérieur de la boîte mère, soit au monde intérieur (c'est à dire elle appartient à une boîte fille de la boîte mère ou à une boîte petite-fille, ou ....). Dans ce dernier cas, la boîte référence s'appelle boîte référence **propriétaire ("owned")**.

L'alias entre deux boîtes induit des alias implicites sur leurs filles instances respectivement. Si on dit que deux représentants du modèle “VOITURE” correspondent au même objet, alors leurs roues seront également identifiées et cela continue récursivement, c'est à dire que les jantes des roues sont également identifiées, etc...

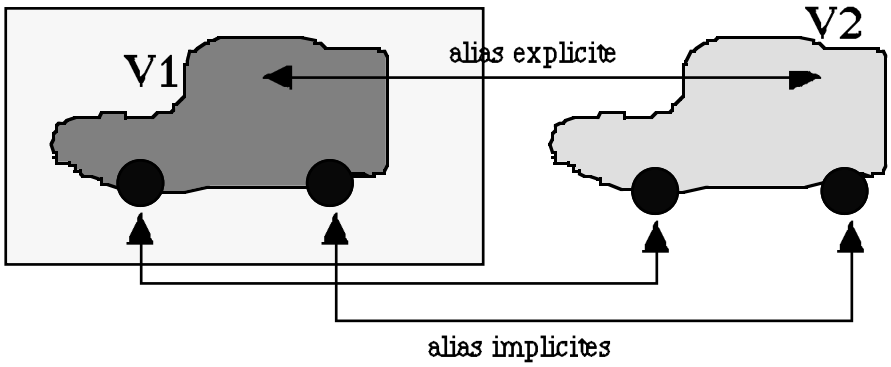


figure IV.11 alias implicites

Par contre si cette voiture admet un feu de circulation en référence, ce feu pourra être un objet différent dans les deux représentations. De la même manière si on détruit ou transporte une boîte, ses instances suivront mais pas ses références.

Pour cette raison on utilise souvent les termes : "on exporte une boîte fille instance" et "on reçoit (ou importe) une fille référence non propriétaire (non owned)".

L’alias pose la question de la compatibilité des modèles. Il ne doit pas être possible d’aliasser “CORNICHON” à “PROGRAMMEUR” mais par contre rien ne s’oppose à l’alias de “CERCLE” à “OBJET GEOMETRIQUE”. **BOCAL** donne une réponse souple à cette question qui sera traitée plus loin.

IV.2.3 Meta modèles

A l’intérieur d’une boîte, il y a des boîtes à l'intérieur desquelles … Il faut bien que cela s’arrête. On appelle modèle terminal un modèle qu’on ne peut plus ouvrir. BOCAL distingue quatre famille de modèles dont trois sont terminales. On remarquera que les modèles présentés sont décrits de deux manières équivalentes, soit par une syntaxe textuelle facile à comprendre, soit par un dessin graphique plus parlant et qui présente une grande aide pour mieux comprendre l'activité du modèle, et même pourra se présenter comme un outil de spécification et de modélisation.

1) Modèles de données élémentaires

Cette famille de modèles réserve une zone de mémoire pour y affecter des données. On retrouve dans cette famille les modèles usuels (entier, réel, caractère) mais l’utilisateur peut définir tout autre modèle.

Ci-dessous les principaux modèles de données élémentaires :

```
BOX Real [8];
ENDBOX

BOX Integer [4];
ENDBOX

BOX Char [1];
ENDBOX

BOX String [32];
ENDBOX

BOX Token [0];
ENDBOX

BOX Generic;
ENDBOX
```

On aura compris qu’entre crochets figure le nombre d’octets de données. Le modèle Token (jeton) est particulièrement utile car c’est une ressource exigée par pratiquement toutes les boîtes pour l’activation (Token $start) et c'est une ressource produite à la fin de l'activation (Token $stop), ce modèle joue le rôle des variables booléennes dans les

langages classiques et servira comme une ressource virtuelle pour synchroniser l'activation et la désactivation des différentes boîtes. Les jetons "$stop" et "$start" sont expliqués plus en détail au paragraphe IV.2.7.

Le modèle **Generic** n'est pas tout à fait un modèle de données. Il est le modèle le plus général et c'est l'équivalent de la classe **OBJECT** dans les langages orientés objets. Autrement dit, tous les modèles sont compatibles avec le modèle Generic. Ces modèles de données sont des boîtes terminales et ne comportent pas de boîtes filles.

2) Modèles prédéfinis

Comme la famille précédente, cette famille est terminale et correspond en gros aux instructions élémentaires.

On retrouve ici les opérations génériques (create, delete, affect, copy). On y trouve également les opérateurs logiques et arithmétiques. Ce sont les instructions codées en “dur” dans le langage.

A titre d’illustration de la syntaxe de BOCAL nous donnons les plus importantes (qui se trouvent dans le fichier kernel.boc) :

```
#include "basic.boc"

/* ce modèle copie y dans x */
BOX copy;
REFERENCES
 Generic x,y;
ENDREF
PRECOND
 y AND ~x;
PREDEF 1
ENDBOX

/* ce modele détruit la boîte x */
BOX delete ;
REFERENCES
 Generic x;
ENDREF
PRECOND
 x;
PREDEF 2
ENDBOX


/* ce modele crée la boîte x */
BOX create ;
REFERENCES
 Token start,stop ;
 Generic x;
ENDREF
PRECOND
 ~x;
PREDEF 4
ENDBOX
```

Pour illustrer la supériorité du visuel, les mêmes dessinés :

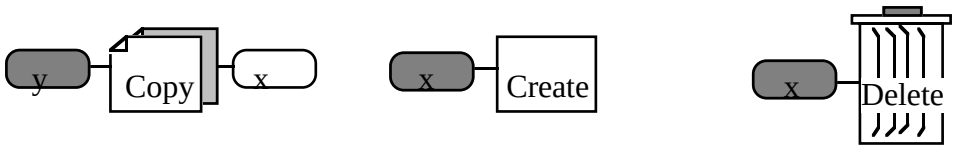


figure IV.12 exemples des modèles prédéfinies

3) Modèles EXEC (Interface avec le langage C)

Bien qu'on puisse en théorie tout programmer en BOCAL, ce langage n'est pas commode pour implanter des algorithmes à flot de données complexes. BOCAL permet à l'utilisateur de définir ces propres modèles terminaux et de les programmer en langage C. Cette famille porte le nom de modèles **EXEC**. Pour interfacer deux langages différents, il faut définir une correspondance entre données et un mécanisme de passage d'arguments. Pour créer un modèle EXEC il faut :

1) Ecrire une routine C
2) Décrire un modèle BOCAL correspondant

Syntaxe d'une boîte d'interface :

```
/* exemple d'une boîte qui réalise la concaténation des chaînes */
BOX concatenate;
REFERENCES
 String x,y,result;
ENDREF
PRECOND
 x and y and ~result;
EXEC
PUSH x,y;         /* liste des arguments */
boc_strcat;       /* nom de la procédure */
PULL result;      /* liste des retours   */
ENDBOX
```

BOCAL offre deux procédures : **bocal_pull** et **bocal_push**, bocal_pull permet de récupérer les arguments empilés sur la pile du système BOCAL et c'est dans l'ordre inverse de leur déclaration dans la section **push**, par contre bocal_push permet d'empiler les résultats dans la pile du système bocal et c'est dans l'ordre inverse de leur déclaration dans la section **pull**. Les instructions bocal_pull devraient être faites au début de la procédure et les instructions bocal_push à la fin de la procédure. Par exemple la procédure boc_strcat devrait avoir l'allure suivante :

```
void boc_strcat()
{
 char *x,*y,*result;
 y= (char *) bocal_pull();
 x= (char *) bocal_pull();
 ...
 bocal_push((void *) result);
}
```

Une telle procédure devrait pouvoir recevoir tout modèle de boîtes, pour cela le système BOCAL devrait convertir les modèles des arguments en des types bien connus par le langage C.

En effet, le compilateur BOCAL produit un fichier ".h" pour chaque fichier ".boc" qui contient un type struct pour chaque modèle qui se trouve dans le fichier ".boc".

Aux modèles élémentaires correspondent les types suivants :

* au modèle Integer correspond le type : (long *).
* au modèle Real correspond le type : (double *).
* au modèle Char correspond le type : (char *).
* au modèle String correspond le type : (char *).

Pour les modèles composés correspond un struct où les champs correspondent aux boîtes filles instances et non locales comme le montre l'exemple suivant qui se trouve dans le fichier "rectangle.boc" :

```
#include "basic.boc";
BOX point;
INSTANCES
      Integer x,y;
ENDINST
ENDBOX

BOX Rectangle;
INSTANCES
      point origine;
      Integer longueur,largeur;
      LOCAL Real a1,a2;
      ...
ENDINST
REFERENCES
      Token dessiner;
      LOCAL Token j;
      ...
ENDREF
ENDBOX
```

Les structures associées à ces modèles sont les suivantes et se trouvent dans le fichier "rectangle.h" :

```
typedef struct st_point {
        long *x;
        long *y;
}st_point,*pt_point;

typedef struct st_rectangle {
        struct *st_point origine;
        long *largeur;
        long *longueur;
}st_rectangle,*pt_rectangle;
```

L'ordre des champs de la structure coïncide avec l'ordre de la déclaration de ces boîtes filles dans la section INSTANCES. Alors supposons qu'on veut écrire un modèle qui calcule la superficie d'un rectangle :

```
#include "rectangle.boc";

BOX superficie;
REFERENCES
      rectangle R;
      Integer result;
ENDREF
PRECOND
 R and ~result;
EXEC
PUSH R;           /* liste des arguments */
boc_superficie;   /* nom de la procédure */
PULL result;      /* liste des retours   */
ENDBOX
```

La procédure boc_superficie a l'allure suivante :

```
#include "rectangle.h"
void boc_superficie()
{
 pt_rectangle R;
 long result;
 R= (pt_rectangle) bocal_pull();
 result= (*(R->longueur)) * (*(R->largeur));
 bocal_push((void *) (&result));
}
```

Toutefois, ce modèle superficie suppose que R est du modèle rectangle et il ne saura pas traiter un autre modèle même s'il est compatible avec le modèle rectangle parce que malheureusement le langage C ne connaît pas la notion d'héritage. Une interface avec C++ pourra-t-elle nous donner une réponse à ce problème?.

4) Modèles complexes

Cette famille est la famille des boîtes “normales” qui s’obtiennent par encapsulation de représentants des modèles de base ou d’autres modèles complexes. A titre d’exemple, construisons le modèle transfert à partir de copy et delete :

```
BOX transfert;
REFERENCES
 Generic x,y;
ENDREF
ALIASES
Copy  :x      == x; /* alias entre (copy:x) la fille "x" de la boîte
                       copy et la fille "x" de la boîte transfert */
Copy  :y      == y;
Delete:x      == y;
Copy  :$start == $active;
Copy  :$stop  == Delete:$start;  /* la fille "$stop" de copy est */
                    /* en alias avec la fille "$start" de delete" */
PRECOND
 y AND ~x;
ENDBOX
```

Ce qui est, encore une fois plus simple en visuel :

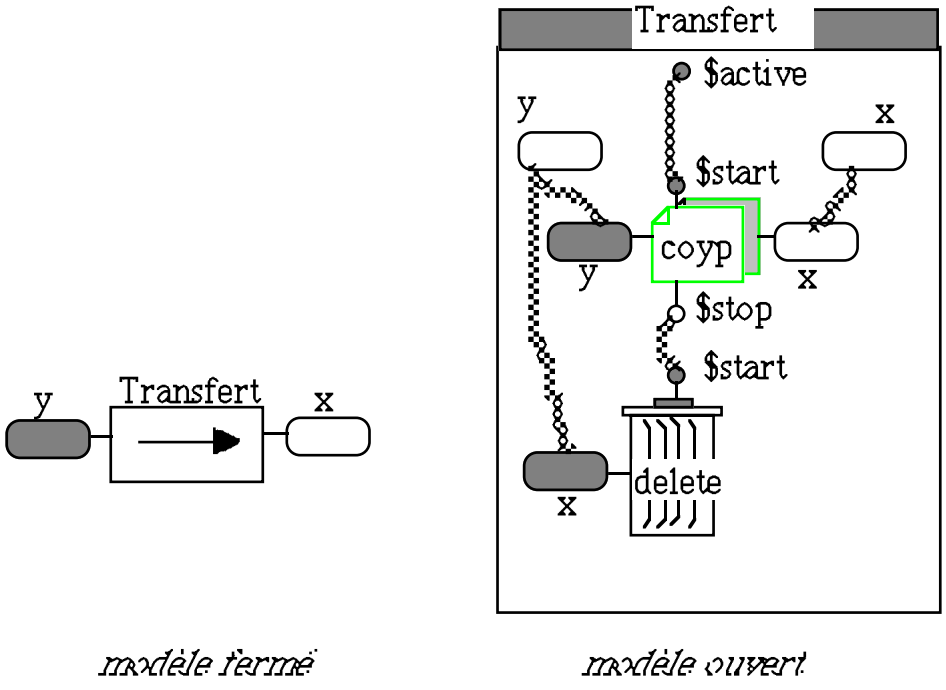


figure IV.13 exemple d'un modèle composé

Les deux boîtes delete et copy sont deux boîtes instances locales et sont "initialisables", c'est-à-dire que la création d'une boîte transfert implique la création de ces deux filles "copy" et "delete". Ces deux boîtes sont locales parce que le modèle Transfert n'est pas conçu pour avoir des données. Si la boîte copy (ou delete) est déclarée comme référence non locale, à ce moment là une boîte transfert attend que son environnement lui fournisse une "fonction" compatible avec cette fille.

Les jetons $start, $stop et $active sont gérés par le système BOCAL pour délimiter le début et la fin de chaque activité. (voir IV.2.7)

Une fois la définition du modèle Transfert finie, le modèle devient fermé et l'utilisation de ce modèle pour la définition d'autres modèles ne dépend que de son interface et on n'a pas à savoir s'il s'agit d'un modèle composé ou d'un modèle prédéfini.

IV.2.4 Boîtes activables et boîtes passives

Il faut bien que certaines boîtes puissent s'exécuter pour pouvoir parler d'un programme BOCAL. Le système BOCAL distingue bien les boîtes passives des boîtes activables et les modèles passifs des modèles activables. Les modèles de données élémentaires sont des modèles passifs, par contre les modèles prédéfinis et les modèles EXEC sont activables.

Les modèles de données sont passifs. Ils ne sont pas activables. Cette propriété est composable. Un modèle complexe (voir plus loin) qui ne comporte que des boîtes de données est lui-même une boîte de données et, par là, est passif. Les modèles complexes de données recouvrent la notion de record du PASCAL ou de struct du C.

Les modèles complexes sont activables ou passifs selon l'état de leurs composantes. Un modèle est activable si et seulement s'il contient au moins une boîte activable. Une boîte est activable si elle appartient à un modèle activable et si elle exporte au moins une boîte fille référence (elle est prête à communiquer).

Un modèle est passif s'il n'est pas activable et une boîte est passive si elle n'est pas activable.

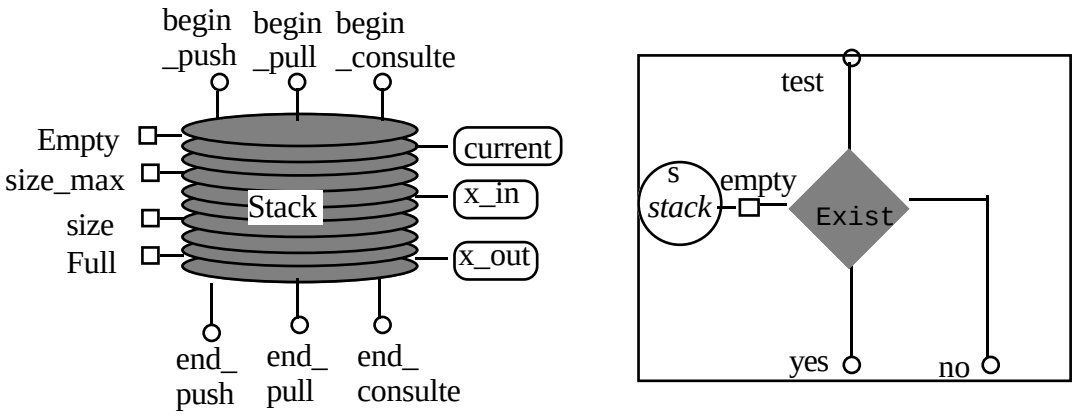


figure IV.14 exemple d'une boîte passive dont le modèle est activable

Toutefois une boîte dont le modèle est activable peut être passive dans un contexte particulier si elle ne fait exporter que ses filles instances. Par exemple le modèle **stack** (pile) est un modèle activable qui sait empiler et dépiler des éléments de modèle Generic. Alors que (s) est une boîte passive dans le modèle de la figure IV.14 qui consulte l'état du jeton "**empty"** de la boîte s.

IV.2.5 Activation et désactivation

Seules les boîtes activables peuvent s'activer si un certain nombre des ressources sont disponibles pour lancer l'activité de la boîte. Chaque boîte activable est considérée comme un process et ses boîtes filles activables comme des process enfants. Chaque process peut être dans l'un des trois états : actif, inactif, une boîte active pouvant être vraiment active ou calme(ou endormie).

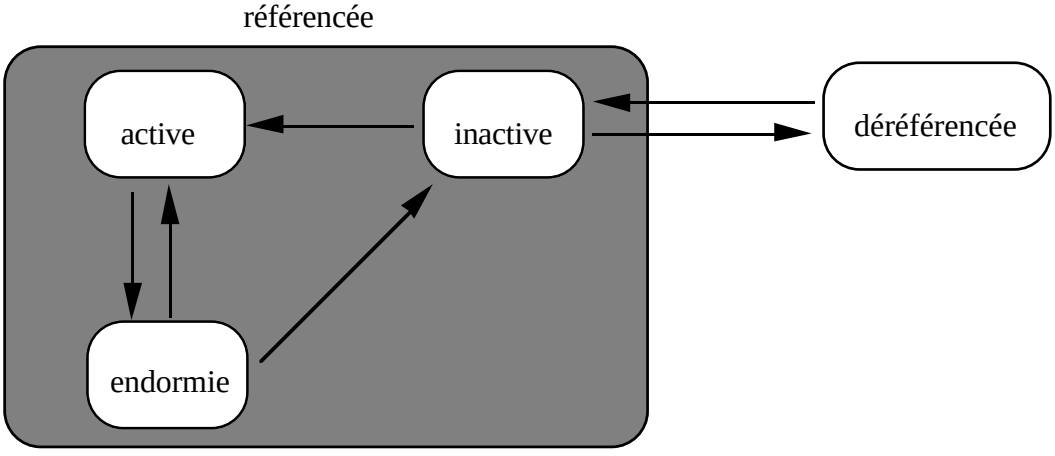


figure IV.15
les différents états d'une boîte activable(process)

La description des boîtes filles d'un modèle ne leur donne pas pour autant une existence physique. Il faut un acte de transfert ou de création pour leur donner cette existence, faute de quoi leur emplacement reste vide. On dit (improprement) qu'une boîte est référencée si elle existe, et déréferencée dans le cas contraire. Certaines boîtes filles instances ont un indicateur montrant qu'elles seront automatiquement référencées à la création de leur mère, et ce récursivement. On les appelle des boîtes initialisables.

Les boîtes non référencées ne peuvent pas être actives parce qu'elles n'existent pas. Une fois une boîte activable créée, elle peut rentrer dans ce cycle "activation-désactivation-endormie" (figure IV.15).

Une boîte activable passe de l'état inactif à l'état actif si :
1) Sa boîte mère est active
2) Un certain nombre de ressources sont disponibles : la précondition est vérifiée.

On appelle **précondition** la condition que doivent satisfaire les ressources d'une boîte (c'est-à-dire ses boîtes filles non locales) pour permettre son activation. Dans la version actuelle de BOCAL, cette condition est une simple expression booléenne, par exemple :

```
(b1 OR b2) AND (b3 OR ~b4);
```

Une précondition teste le référencement ou le non référencement d'un ensemble de boîtes. Ce dernier cas s'écrit en faisant précéder le nom de la boîte du signe ~.

On peut s'étonner que l'on doive tester la non-existence d'une boîte dans une précondition, mais cela correspond à une situation très fréquente. Dès qu'une boîte crée ou transporte un objet, elle doit vérifier avant de commencer que la destination est inoccupée, c'est à dire non référencée.

Une précondition teste aussi la disponibilité des ressources, parce qu'il ne suffit pas de tester qu'une ressource existe ou pas. Faut-il également s'assurer qu'un autre process ne l'utilise pas. Quand une boîte devient active, elle bloque toutes les ressources[1] qui figurent dans sa précondition. Pour être plus précis, ces ressources deviennent bloquées pour le monde extérieur à la boîte mais restent disponibles à l'intérieur pour les boîtes filles activables.

---

[1]De fait pour certaines ressources qu'on ne fait que consulter, BOCAL permet d'éviter le blocage mais c'est une pratique qui conduit facilement à des erreurs.

Toutefois une précondition peut tester uniquement la disponibilité sans tenir compte de son existence. Ce cas s'écrit en faisant précéder le nom de la boîte du signe ?.

Une précondition revient donc en général à écrire que les ressources qu'on consomme sont présentes et celles qu'on produit sont absentes et que personne n'y touche entre temps (la disponibilité).

On dit qu'une boîte active devient **endormie** ou **calme** si toutes ses boîtes filles sont calmes ou inactives et aucun événement ne se produit à l'intérieur de la boîte. Autrement dit une boîte calme ne change pas d'état si on la considère comme process isolé.

Une boîte active devient inactive si :
- elle est calme,
- toutes ses boîtes filles sont inactives ou déréférencées,
- sa postcondition est vérifiée.

La postcondition est une expression booléenne qui permet de tester la présence de certains ressources et l'absence d'autres pour que la boîte puisse devenir inactive. A la différence des préconditions, une postcondition ne teste pas la disponibilité des ressources. C'est au contraire, quand la postcondition est vérifiée, que la boîte devient inactive et libère les ressources bloquées par sa précondition.

Toutefois, il existe des boîtes activables sans précondition ou (et) sans postcondition. Une boîte activable sans précondition a toujours la valeur "vrai" pour l'évaluation de sa précondition, de même pour les boîtes sans postcondition.

Ces notions sont intuitives, si on se représente par exemple une calculette. L'état actif ou inactif s'obtient par l'interrupteur on/off. La calculette est endormie quand elle attend une entrée clavier. Cela veut dira qu'il ne se passera plus rien dans la boîte sans événement extérieur.

La propriété d'être active est locale. Le même objet peut-être actif dans une boîte et inactif dans une autre. Ainsi une imprimante partagée par plusieurs applications sera active dans la boîte en train d'imprimer et inactive dans les autres. Comme une boîte peut exécuter plusieurs tâches en parallèle, il est fort possible qu'une même boîte soit active dans plusieurs boîtes mères.

IV.2.6 Blocage d'une ressource

BOCAL est un langage parallèle. Ceci pose la question du partage des ressources. Quand une boîte teste une précondition, elle vérifie que toutes les boîtes intervenant dans la condition sont libres ou disponibles. Si la précondition est vérifiée, les boîtes sont instantanément bloquées. Quand la boîte s'éteint, les filles bloquées redeviennent libres.

On peut remarquer qu'on bloque aussi bien des ressources présentes que des ressources absentes. Ainsi, si on suppose à l'activation qu'une destination est disponible, il ne faut pas que durant le process, l'extérieur s'avise de l'occuper.

Il faut bien comprendre que quand une boîte bloque des ressources, elle les bloque pour l'extérieur, mais qu'elles restent disponibles pour l'intérieur. Le blocage est donc, comme l'activation, une marque locale, même si la disponibilité est globale. De même quand on débloque une ressource, elle redevient libre mais pas pour tout le monde, elle reste bloquée pour la boîte mère de la boîte qui vient de libérer cette ressource.

Le blocage des ressources pose le problème de conflits de partage de ressources comme le montre la figure IV.16. Un véhicule arrivant à une intersection des routes, une de ces deux rues acceptera d'accueillir ce véhicule. Puisque la précondition de chaque rue est que x_in soit présent, les deux boîtes sont candidates à s'activer. BOCAL, dans l'état actuel, laisse aux "programmeurs" le soin de donner une priorité à une de ces rues par rapport à l'autre selon un critère. Ceci se fait en exigeant la présence d'autres ressources (virtuelles ou réelles) dans la précondition de la rue défavorisée, faute de quoi BOCAL résout le conflit en activant une de ces deux boîtes aléatoirement.

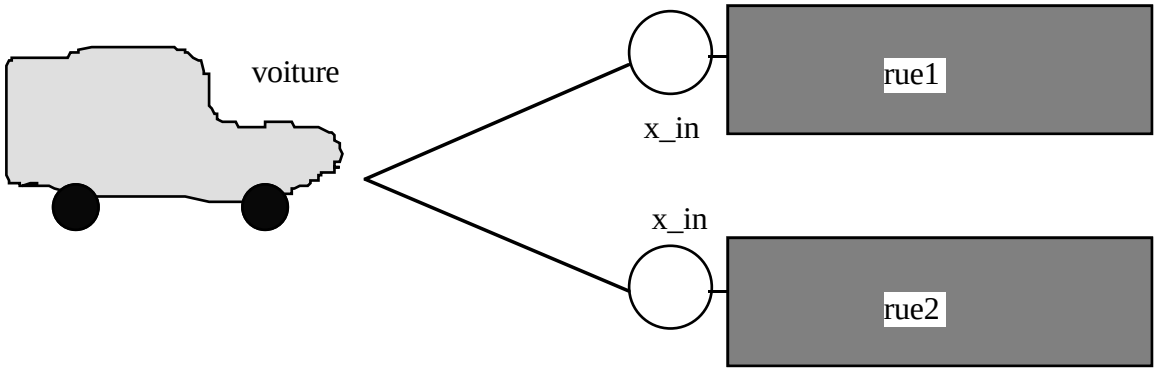


figure IV.16 exemple d'un conflit

Toutefois une boîte peut devenir active sans bloquer de ressources : il suffit que son modèle n'ait pas déclaré de précondition.

IV.2.7 Jetons systèmes

L'activation d'une boîte est conditionnée d'une manière générale par la présence d'un certain nombre de ressources et par l'absence d'autres. En général, une boîte active consomme les ressources exigées présentes dans la précondition et produit les ressources exigées absentes dans la précondition.

Les jetons sont des boîtes appartenant au modèle **token** et servent à contrôler et à synchroniser l'activation des boîtes. Les jetons présentent un intérêt dans la phase de modélisation et rappellent les jetons des **réseaux de Petri**.

Prenons par exemple la boîte copy :

copy sans jetons de contrôle

copy avec jetons de contrôle

figure IV.17 les deux boîtes de copy

Ces deux modèles exigent que la ressource y soit présente et x soit absente. A la fin de l'activation, la ressource y est copiée dans x sans aucun changement de la ressource y. Une fois y présent, y est copié dans x chaque fois que x est consommé. Cet automaticité peut-être gênante mais pour avoir plus de contrôle sur l'activation de la boîte Copy1, il nous faut définir un autre modèle Copy2 qui exige la présence d'une ressource fictive start et l'absence de stop. A la fin de la copie le jeton start est consommé et le jeton stop est produit. Ces deux boîtes start et stop appartiennent au modèle **Token** qui est un modèle élémentaire et qui n'a pas de données. Un jeton (Token) n'a donc pas de valeur : il est simplement présent ou absent.

Pour éviter d'avoir à décliner tous les modèles en "automatique" ou "contrôlée", BOCAL permet d'ajouter ces jetons de contrôle dynamiquement selon le besoin de l'utilisateur. Ainsi le modèle Copy n'a pas besoin de déclarer les jetons start et stop : l'utilisateur peut les ajouter localement sous les noms **$start** et **$stop**(respectivement) pour contrôler l'activation de la boîte Copy et plus généralement n'importe quelle boîte. Les

noms de ces jetons commenecent par le caractère '$' pour les distinguer des boîtes nommées par l'utilisateur.

Toutefois, le "programmeur" a toujours le droit d'imposer des jetons de contrôle dans la déclaration des modèles. Le système BOCAL offre d'autres jetons et d'autres pour augmenter le contrôle et diminuer le nombre de boîtes de test. Toutes ces boîtes ont des noms significatifs commençant par la lettre $.

Voici les jetons de contrôle reconnus par la version actuelle de BOCAL :

- **$quiet**.
- **$active**.
- **$start**.
- **$stop**.
- **$absent**.
- **$present**.
- **$kill**.

Les deux contrôles "$quiet" et "$active" concernent les modèles et les autres contrôles concernent les boîtes. Le monde extérieur à une boîte n'a pas un accès aux filles $quiet et $active de celle-ci qui ces sont des propriétés internes, à l'inverse c'est le monde extérieur qui lui définit ses filles $start, $stop, $kill, $absent et $present.

1) Le jeton "$quiet"

Chaque modèle peut utiliser, sans déclaration, une boîte fille nommée "$quiet" dans sa section d'alias. Quand une boîte active s'endort, si la postcondition est vérifiée elle devient inactive ; Sinon elle produit le jeton nommé "$quiet" si son modèle utilise une boîte fille "$quiet". Ce jeton permet de relancer la vie dans la boîte endormie.

2) Le jeton "$active"

Chaque modèle peut utiliser, sans déclaration, une boîte fille nommée "$active" dans sa section d'alias. Quand une boîte devient active, ce jeton $active est immédiatement produit.

3) Les jetons "$start", "$stop", "$present" et "$absent"

Ces jetons de contrôle jouent sur la précondition de la boîte et constituent une précondition supplémentaire. Chacune de ces boîtes est une boîte multiple (voir IV.2.8), c'est-à-dire elle peut avoir plusieurs filles "$start", plusieurs filles "$stop", plusieurs filles "$present" et plusieurs filles "absent".

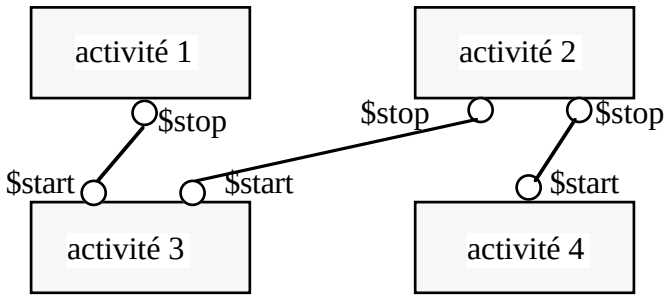


figure IV.18 exemple de synchronisation entre boîtes

Cette figure montre que l'activité 3 ne peut pas s'activer avant la fin de l'activité 1 et la fin de l'activité 2. Par contre l'activité 4 peut s'activer tout de suite après la fin de l'activité 2.

La précondition supplémentaire est :

```
$start and ~$stop and $present and ~$absent
```

Cette précondition supplémentaire vérifie que :

1) chaque fille "$start" soit présente et disponible
2) chaque fille "$present" soit présente et disponible
3) chaque fille "$stop" soit absente et disponible
4) chaque fille "$absent" soit absente et disponible.

Si cette précondition supplémentaire est vide alors elle est toujours vraie.

L'activation d'une boîte bloque les ressources exigées dans la précondition, y compris les filles "$start" et les filles "$stop". Une fois la boîte devenue active les boîtes filles "$start" sont consommées, et quand la boîte devient inactive les boîtes filles "$stop" sont produites.

Les boîtes filles "$present" et les boîtes filles "$absent" ne sont pas bloquées et elles ne sont ni consommées ni produites (contrairement aux boîtes filles "$start" et "$stop"). Simplement, la présence des "$present" et l'absence des "$absent" servent à contrôler l'activation de la boîte.

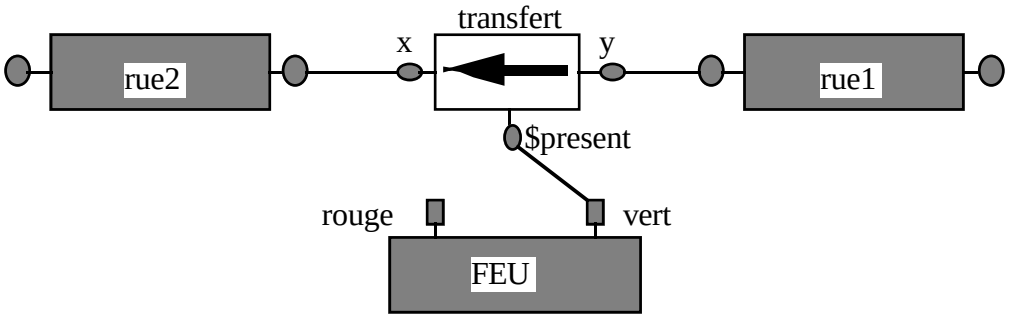


figure IV.19
exemple d'un feu contrôlant deux rues

Cet exemple montre l'utilisation de la boîte "$present" pour contrôler la boîte de transfert ; Le passage des véhicules de la première rue à la deuxième rue est donc conditionné par la présence du jeton vert.

4) Le jeton "$kill"

Ce jeton de contrôle joue sur la postcondition de la boîte et sert à désactiver la boîte quand la boîte devient endormie même si sa postcondition initiale n'est pas vérifiée. Parce que la postcondition devient :

```
postcondition_initiale or $kill
```

Si une boîte passe de l'état actif à l'état endormi, alors elle et toutes ses boîtes filles sont désactivées si elle possède une fille "$kill" présente ; Sinon elle génère éventuellement un jeton "$quiet" si le modèle de la boîte l'exige.

IV.2.8 Multiplicité

Lors de la déclaration d'un modèle, certaines filles références non locales possèdent le flag "multiple" et sont appelées filles multiples.

Au départ, cette notion fût introduite pour définir les boîtes **conjoncteur** et **disjoncteur**. Le conjoncteur permet de réaliser le rendez-vous entre plusieurs jetons et le disjoncteur permet de produire plusieurs jetons simultanément.

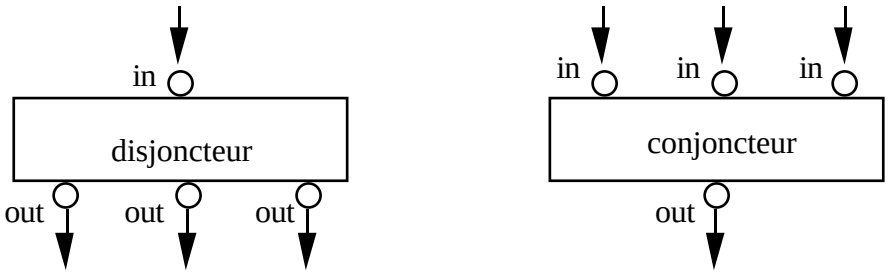


figure IV.20 la fille out (disjoncteur) et la fille in (conjoncteur) sont des boîtes multiples

A priori le nombre des boîtes filles "out" dans le disjoncteur n'est pas connu lors de la déclaration du modèle disjoncteur. C'est pour cette raison que fût introduite la multiplicité des filles. Ainsi le disjoncteur déclare une seule fille nommée "out" qui est multiple et l'utilisation du disjoncteur dans la définition d'autres modèles permet de déterminer le nombre des filles "out" propre à un contexte particulier.

```
BOX conjoncteur;
REFERENCES
      Token *in;   /* fille multiple */
      Token out;
ENDREF
PREDEF 60 /* numéro d'identification */
PRECOND
 in and ~out;
ENDBOX

BOX disjoncteur;
REFERENCES
      Token in;
      Token *out;  /* fille multiple */
ENDREF
PREDEF 61 /* numéro d'identification */
PRECOND
 in and ~out;
ENDBOX
```

La précondition du conjoncteur attend que les boîtes "in" soient présentes et que la fille "out" soit absente pour consommer toutes les "in" et produire le jeton "out".

A ce titre les jetons systèmes : $start, $stop, $absent et $present sont des boîtes multiples, autrement dit on peut utiliser plusieurs $start pour activer la boîte et produire plusieurs jetons $stop à la fin de l'activité.

Cette propriété de multiplicité peut être transmise par alias à la boîte englobante :

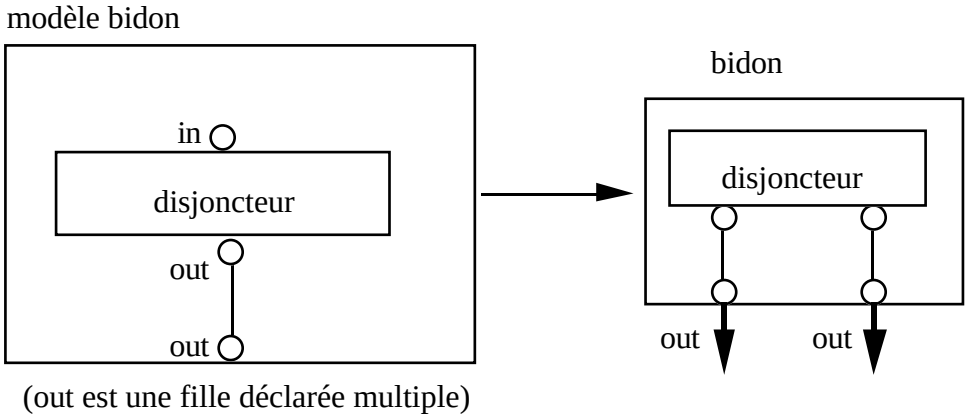


figure IV.21 une fille multiple est en alias
avec une autre boîte multiple

IV.2.9 Allocation dynamiques de ressources

On utilise souvent le terme ressource pour désigner un argument d'une activité ou plus précisément une fille référence non locale. En général, un modèle qui exige une ressource R, déclare une fille référence non locale nommée R du modèle désiré et exige dans sa précondition que R soit présente pour démarrer sa vie active. Alors cette exigence est adressée à la boîte englobante qui à son tour soit peut la satisfaire, soit la transmettre au niveau supérieur et ainsi de suite.

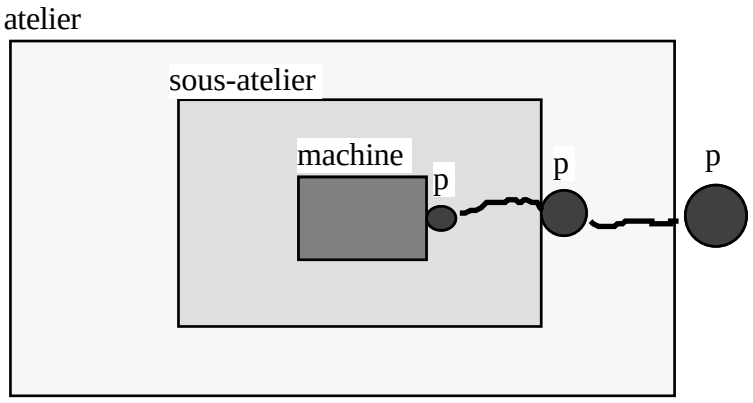


figure IV.22 le passage de la demande d'une ressource p

La figure IV.22 montre que si le modèle de la machine demande une pièce p, alors le modèle du sous-atelier est obligé de faire une demande de pièce p et puis le modèle de l'atelier demande à son tour une pièce p.

Ce mécanisme statique d'allocation exige évidemment qu'une mère connaisse parfaitement tous les besoins de ses filles. Ceci peut-être impraticable en particulier pour les boîtes qui ont besoin des ressources systèmes, en général partagées par plusieurs boîtes. Prenons, pour illustrer ce propos, l'exemple des modèles élémentaires d'entrée/sortie : le modèle write et le modèle read :

```
BOX read;
REFERENCES
      Generic x;
      Stream stdiout?; /* ressource à chercher dynamiquement */
ENDREF
PREDEF 62              /* numéro d'identification */
PRECOND
 ~x and stdiout;
ENDBOX

BOX write;
REFERENCES
      Generic x;
      Stream stdiout?; /* ressource à chercher dynamiquement */
ENDREF
PREDEF 63              /* numéro d'identification */
PRECOND
 x and stdiout;
ENDBOX
```

Chaque boîte d'entrée/sortie a besoin de réserver la console pendant qu'elle est active parce que dans un système parallèle il est très important de partager cette ressource commune sans provoquer des conflits imprévisibles. La pratique montre qu'il n'est pas commode de passer la demande de cette ressource dans tout modèle qui englobe une boîte entrée/sortie. Le remède consiste à déclarer que cette ressource doit être cherchée dynamiquement. Le système BOCAL cherche la boîte INSTANCE créée et disponible dont le modèle est Stream (c'est le modèle de la boîte stdiout) ou un autre modèle compatible avec le modèle Stream et qui soit "la plus proche" du demandeur. La recherche peut échouer si une telle ressource n'existe pas dans le monde BOCAL ou si la ressource existe mais est occupée.

Si la recherche aboutit à plusieurs ressources alors celle qui est "la plus proche" sera allouée pour la boîte qui a déclenché la recherche. Pour bien expliquer la procédure de recherche dynamique, il faut qu'on définisse une notion de distance d'une ressource par rapport à la boîte qui a déclenché la recherche.

Dans le système BOCAL, toutes les boîtes ont un ancêtre commun qui est la boîte racine (ou le monde) ; Cette boîte est créée chaque fois l'exécuteur BOCAL lancé. Chaque boîte dans le monde BOCAL a la boîte racine comme boîte mère, boîte grand-mère, ou boîte arrière grand-mère, etc...

On définit l'ancêtre de deux boîtes comme la première grand-mère commune à ces deux boîtes. La figure suivante montre un exemple de l'ancêtre commun de deux boîtes :

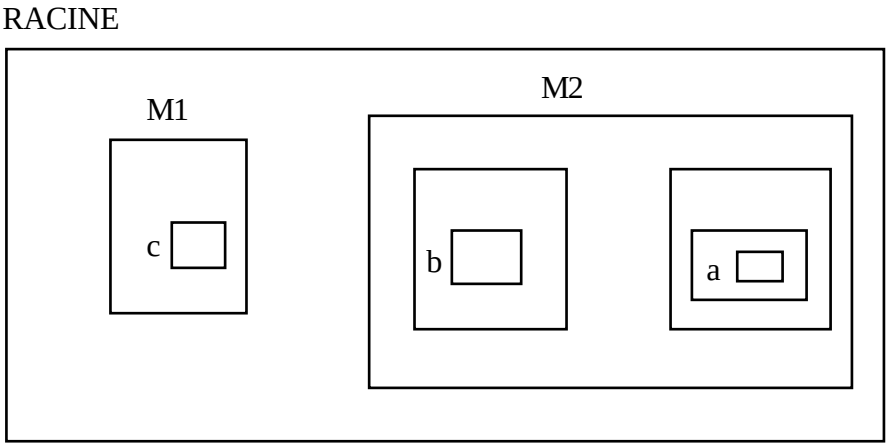


figure IV.23 l'ancêtre commun de a et b est la boîte M2
l'ancêtre commun de a et c est la boîte RACINE

Soient a et b deux boîtes quelconques et M leur boîte ancêtre commune, on définit la distance de b par rapport à a par le nombre des boîtes mères qui séparent M de la boîte mère de a.

Dans l'exemple IV.23 :
1) la distance de b par rapport à a = 2
2) la distance de a par rapport à b = 1
3) la distance de c par rapport à a = la distance de c par rapport à b = 1
4) la distance de a par rapport à c = 3
5) la distance de b par rapport à c = 2.

Alors le terme de "ressource la plus proche" signifie : la ressource qui a la plus petite distance par rapport à la ressource cherchée (stdiout pour la boîte read).

La recherche de la ressource commence dans la boîte qui déclenché la demande (distance 0), puis dans sa boîte mère (distance 1), puis dans la boîte grand-mère (distance 2), etc...jusqu'à la boîte racine. Si la recherche dans la boîte racine est négative alors le monde BOCAL ne possède pas la ressource cherchée.

La recherche est actuellement faite par modèle. L'atelier peut dire : "trouvez moi un Ajusteur" mais pas " trouvez moi Dupont". Mais rien ne s'oppose à implanter les deux modes de recherche dans une extension ultérieure.

IV.2.10 Syntaxe textuelle

BOCAL a été conçu pour fonctionner dans un environnement graphique interactif. Toutefois, BOCAL possède une syntaxe textuelle. BOCAL est organisé en fichiers de modèles.

```
/* Inclusion de modèles */
#include "basic.boc";
#include "math.boc";
#include "mecanique.boc";

BOX voiture;                          /* nom de modèle */

INSTANCES                          /* Section instances */
moteur m;
roue  avg,avd,arg,ard;
carosserie ca;
thermometre temp;
LOCAL integer i [5];       /* i locale initialisée à 5 */
ENDINST                            /* Fin des instances */

REFERENCES                        /* Section références */
Token *control;                      /* fille multiple */
LOCAL temperature t;
clock ck?;      /* ressource à chercher dynamiquement */
ENDREF
ALIASES
thermo temp t;   /* Alias de la fille temp de thermo à la fille t
du modèle. On peut aussi créer un alias de petite-fille à petite-
fille
…
ENDALIASES
PRECONDITION
~control AND ck;
POSTCONDITION
control;
ENDBOX

BOX xxxxx;
…
ENDBOX

BOX xxxxx;
…
ENDBOX

BOX xxxxx;
…
ENDBOX
```

IV.3 La notion de compatibilité en BOCAL

Nous commençons par la définition de quelques termes souvent utilisés au cours de ce paragraphe. On rappelle que la relation d'alias divise le monde des boîtes en classes d'équivalence. On appelle **objet** tout ensemble des boîtes liées par un jeu d'alias (i.e une classe d'équivalence est appelée **objet**), on dira que les boîtes d'une classe d'équivalence sont des **représentations** de l'**objet**.

IV.3.1 Introduction à la notion de compatibilité

Tout un chacun connaît intuitivement la notion de compatibilité. Dans l'échange une roue crevée par la roue de secours qui n'a pas forcément les mêmes caractéristiques mais qui n' empêchera pas la voiture de réaliser sa fonction de rouler ou dans le remplacement d'un composant électronique d'un circuit intégré par un autre composant qui respecte les branchements avec les composants voisins. On connaît aussi la notion de compatibilité pour des machines qui savent traiter toute une gamme d'objets, par exemple on peut mettre dans une machine à laver des chemises ou des draps, par contre on ne pourra pas laver de la vaisselle avec cette machine.

La notion de compatibilité trouve ses racines dans la notion d'héritage de l'approche orientée objet où chaque classe hérite ses attributs et ses comportements des classes supérieures. On a vu qu'un modèle est défini par un ensemble des boîtes locales et un ensemble des boîtes d'interface (instances et références). Seules les boîtes d'interface interviennent dans les règles de compatibilité parce que la question de compatibilité se pose uniquement pour les opérations d'alias et pour les opérations de transfert et d'affectations. Toutes opérations qui ne concernent que les boîtes d'interface.

Nos recherches pour spécifier ces règles ont abouti à deux solutions possibles qui ont chacune leurs avantages et leurs inconvénients, la compatibilité par modèle et la compatibilité par boîte.

IV.3.2 La compatibilité des modèles

La compatibilité entre modèles est définie par une relation d'ordre partiel qui va être précisée ci-dessous. Deux modèles seront dit compatibles si ils sont comparables pour cette relation, le modèle supérieur étant le plus spécialisé ("persan" est supérieur à "chat"). Ces définitions auront des conséquences sur les règles d'alias et sur les opérations de transfert.

1) Relation d'ordre entre modèles

Cette relation de compatibilité ne fait intervenir que les boîtes d'interface des modèles. On dira que le modèle M1 est **supérieur ou égal** au modèle M2 (ou M2 est **inférieur ou égal** au modèle M1) si et seulement si :

1) Pour chaque boîte instance non locale b2 de M2, il existe une boîte b1 instance non locale dans le modèle M1 et qui vérifie les deux conditions suivantes :

***** b1 a le même nom que b2

***** le modèle de b1 est supérieur ou égal au modèle de b2.

2) Pour chaque boîte référence et non locale b2 (les boîtes qui n'appartiennent pas au modèle) de M2, il existe une boîte b1 référence non locale dans le modèle M1 et qui vérifie les deux conditions suivantes :

***** b1 a le même nom que b2

***** le modèle de b1 est inférieur ou égal au modèle de b2.

On remarque que les boîtes filles locales et les boîtes filles références propriétaires ("owned") n'interviennent pas dans la compatibilité des modèles. Les jeux d'alias dans les deux modèles M1 et M2 n'ont aucun rôle sur la compatibilité, de même que la précondition ou la postcondition. On rappelle qu'une boîte fille référence est "owned" si elle appartient indirectement au modèle, si elle appartient directement ou indirectement à une fille du modèle.

On dit que le modèle M1 est **strictement supérieur** au modèle M2 (ou M2 est **strictement inférieur** au modèle M1) si et seulement si M1 est supérieur ou égal à M2 et si une des conditions suivantes est vérifiée :

1) Il existe une boîte b1 instance non locale dans M1 et il n'y a aucune boîte instance non locale dans M2 et qui a le même nom que b1.

2) Il existe une boîte b1 référence non locale et non propriétaire (non "owned") dans M1 et il n'y a aucune boîte référence non locale et non "owned" dans M2 et qui a le même nom que b1.

3) Il existe une boîte b2 instance non locale dans M2 telle que le modèle de la boîte b1, qui est instance non locale et qui a le même nom que b2, est strictement supérieur au modèle de b2.

4) Il existe une boîte b2 référence non propriétaire (non owned) et non locale dans M2 telle que le modèle de la boîte b1, qui est référence non owned et non locale et qui a le même nom que b2, est strictement inférieur au modèle de b2.

2) Compatibilité et alias

On a vu que l'alias entre boîtes pose la question de compatibilité entre ces boîtes parce que ces boîtes deviennent représentantes du même objet. Par exemple si A est du modèle ANIMAL, B du modèle LAPIN, C du modèle RHINOCEROS (et si le programmeur a pris soin d'établir soigneusement les noms des filles) on peut établir un alias entre A et B ou entre A et C mais pas les deux simultanément. En effet le premier alias donne à l'objet commun le type LAPIN qui dès lors ne peut plus être confondu avec un RHINOCEROS.

Cette notion de compatibilité détermine le modèle de chaque objet dans le monde BOCAL. On définit le modèle d'un objet comme le modèle maximum selon la relation d'ordre qui lie ces boîtes, ce modèle maximum devant être le modèle d'une représentation de cet objet.

Dans l'exemple suivant (figure IV.24), qui est un exemple fictif, on voit quatre boîtes des modèles différents liées par des alias. Le modèle maximum de cet objet est le modèle de la voiture.

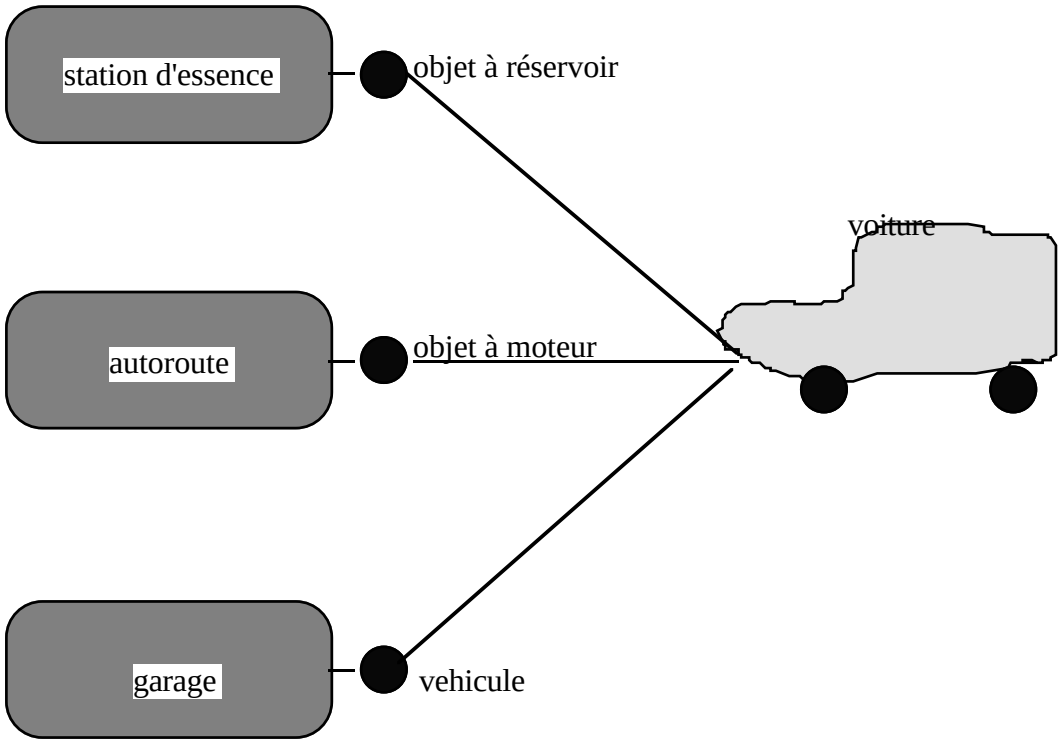


figure IV.24 exemple de compatibilité par modèles

Il faut noter que si le modèle de station d'essence déclare que son entrée est de modèle voiture (à la place de modèle "objets à réservoir") alors la station d'essence ne pourra pas recevoir des camions ou des motocycles.

Si on admet que le modèle du garage et le modèle de la station d'essence sont ceux de la figure IV.24 alors la définition de la compatibilité implique que le modèle de la station d'essence est supérieur à celui du modèle du garage parce que le modèle du véhicule est supérieur au modèle "objets à réservoir".

Voici d'autres conséquences de la relation de compatibilité :

1) L'opération de création (ou référencement) fera développer le modèle maximum de l'objet dans toutes ses représentations.

2) L'opération de transfert impose que le modèle de l'objet transféré soit compatible supérieur ou égal au modèle de la destination. On voit que le transfert peut donner à l'objet destinataire un modèle strictement supérieur à son modèle, donc qu'il peut changer de modèle dynamiquement(par exemple en transférant un chat dans un animal).

3) Modèles génériques et abstraits

Cette notion de compatibilité entre modèles nous impose souvent la déclaration des modèles abstraits qui n'ont aucune existence physique (le modèle ANIMAL ou encore le modèle GENERIC qui est utilisé dans les opérations de création, de destruction et de transfert). Ceci implique que le modélisateur ait une vue globale du problème à modéliser.

A la différence avec l'approche orientée objet, l'ordre de compatibilité entre modèles n'est pas déclaré statiquement. BOCAL établit l'ordre de compatibilité entre modèles dynamiquement. La déclaration d'un nouveau modèle ne l'introduit pas dans le graphe de compatibilité. Une fois une boîte de ce modèle utilisée dans un alias ou dans une opération de transfert, le modèle est inséré dans le graphe de compatibilité. En plus, BOCAL permet, à tout moment, de déclarer des nouveaux modèles même dans l'exécution, c'est-à-dire, l'utilisateur peut suspendre sa simulation, il peut changer la composition d'un modèle et peut également déclarer de nouveaux modèles et reprendre l'exécution de la simulation.

4) Extensions et limites

La règle d'alias imposée par la compatibilité conduit à changer la nature des modèles déclarés, par exemple la figure IV.25 montre que le modèle "âne" peut avoir la peau tachetée à la place de sa peau habituelle et des griffes dans ses pattes grâce aux alias et à la règle de modèle maximum.

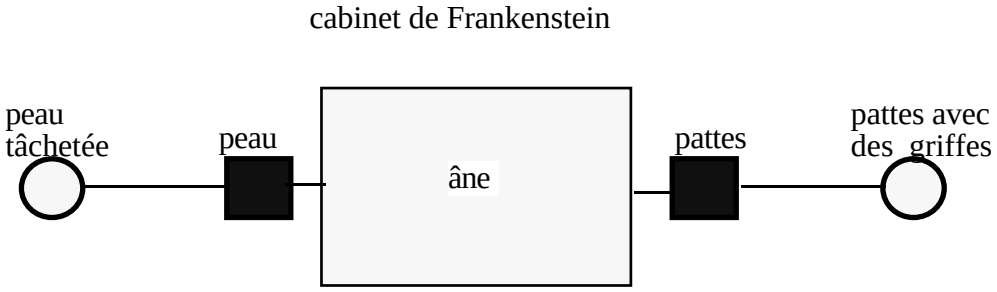


figure IV.25 exemple d'un modèle générique

Dans cet exemple le modèle âne déclare une fille instance non locale de modèle peau et une autre fille instance non locale pour les pattes. Pour transformer le modèle âne en un animal inconnu il suffit d' aliasser la boîte peau à une boîte de modèle peau tachetée et d'aliasser la boîte pattes à une boîte de modèle "pattes avec des griffes".

On voit bien que la règle de compatibilité maximum a des conséquences "dangereuses", le programmeur peut changer la nature d'un modèle par un simple alias.

L'opération de transfert peut elle-aussi changer la nature d'un objet, mais uniquement à l'exécution.

Afin d'interdire de telles situations on peut exiger du modélisateur que la boîte instance ait le modèle maximum de l'objet. Cette exigence est possible, parce que chaque classe d'équivalence (objet) possède au plus une seule boîte instance.

Il faut interdire aussi le transfert d'un objet dont le modèle est strictement supérieur au modèle de l'objet destinataire. En fait, ces interdictions rendent la programmation BOCAL trop contraignante et on se demande s'il faut vraiment imposer de telles règles.

La version actuelle de BOCAL adopte intégralement la compatibilité des modèles et la règle du modèle maximum. Elle rejette ces dernières interdictions.

IV.3.3 La compatibilité par prototype

On a vu que la définition des modèles abstraits est un handicap pour modéliser des applications complexes.

Cette abstraction suppose une connaissance globale et fine des objets modélisés ce qui n'est toujours pas le cas, surtout si le langage est destiné aux utilisateurs de tout niveau. Par

exemple supposons que la voiture soit définie par une carosserie, un moteur, un réservoir et quatre roues. La station d'essence, qui reçoit tout objet à réservoir, peut recevoir alors une voiture, mais ceci oblige le modélisateur à définir le modèle "objets à réservoir", qui est une généralisation du modèle voiture, pour permettre à la station d'essence de recevoir voitures, camions, motocycles, etc... Ensuite, le modélisateur serait obligé d'introduire le modèle "objets à roues", le modèle "objets à moteur", etc...

Ce passage du concret à l'abstrait présente certes des avantages : structuration et sécurisation, minimisation du code, réutilisabilité mais est en fait très lourd pour un système comme BOCAL ouvert à toutes les catégories d'utilisateur. Pourquoi exiger la définition du modèle "objets à réservoir" et ne pas permettre au modèle de la station d'essence de décrire les objets recevables par un prototype. On pourrait par exemple définir la station d'essence avec un argument de modèle voiture. La station utilise alors la voiture comme un prototype pour les objets qu'elle sait traiter et n'utilise de la voiture que son réservoir.

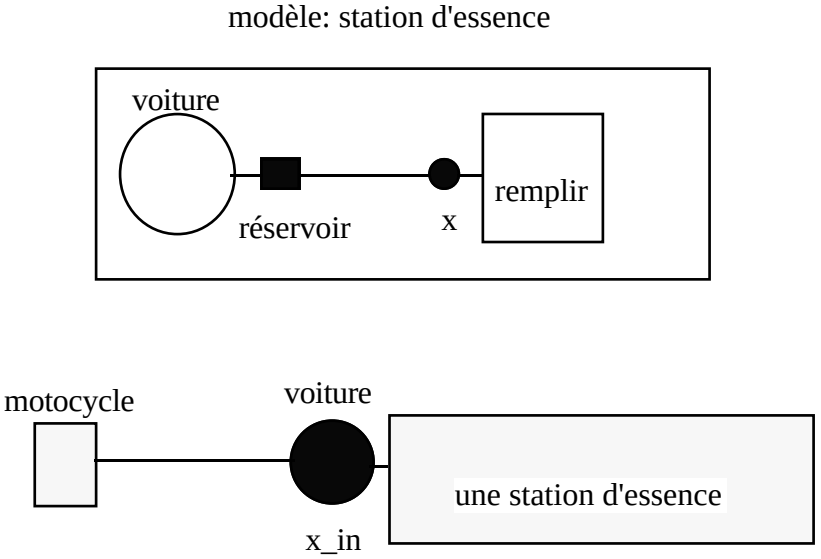


figure IV.26 utiliser la voiture comme prototype pour un objet à réservoir

L'alias liant le motocycle à l'entrée de la station d'essence pose la question de compatibilité entre ces deux boîtes et plus entre les modèles de ces deux boîtes. Cet alias exige alors que la boîte motocycle doit avoir au moins une boîte fille nommée réservoir. Toutefois le modèle voiture n'est pas compatible avec le modèle motocycle.

1) Règles d'alias

Les alias entre boîtes ne posent plus la question de compatibilité parce que le système BOCAL ne peut pas connaître statiquement le modèle de l'objet reçu par ces boîtes. Alors la compatibilité est détectée dynamiquement selon l'objet créé. Ceci permet à l'utilisateur d'aliasser un chat et un chien, c'est l'usage dynamique qui validera ou pas cet alias.

2) Opérations de transfert

Pour valider une opération de transfert, il faut définir la compatibilité de l'objet transféré avec l'ensemble des boîtes de l'objet destinataire. On a deux possibilités :

1) soit déterminer le modèle de chaque objet statiquement (paragraphe précédent IV.3.2) lors des déclarations d'alias. Pour une opération de transfert, il faut tester la compatibilité entre le modèle de l'objet apporté et le modèle de l'objet destinataire.

2) soit laisser ce modèle indéterminé et tester la compatibilité à chaque opération de transfert et à chaque opération de référencement. Ce test se fait de la manière suivante :

Soit A un objet créé et M son modèle et B l'objet destinataire. Le transfert de A vers B est possible si la consition suivante est vérifiée :

L'interface du modèle de 'objet A inclut l'interface de chaque représentation de l'objet B.

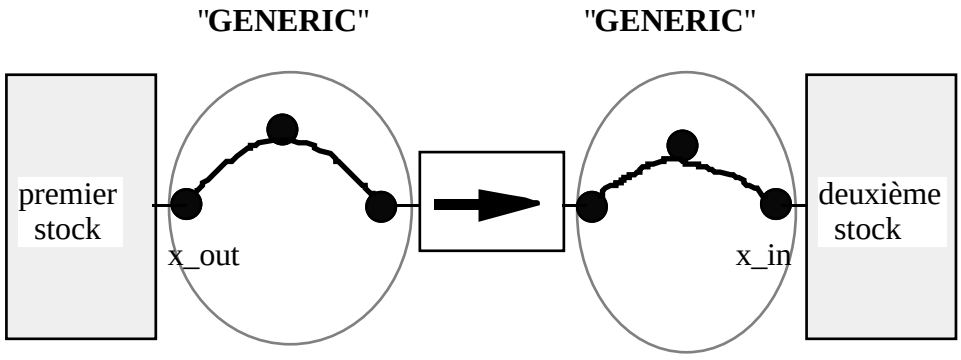


figure IV.27 transférer les objets d'un stock à un autre

La figure IV.27 montre que l'objet à la sortie du premier stock a un modèle "GENERIC" afin d'accepter tous les objets se trouvant dans ce stock et qui peuvent être de nature incompatibles, par exemple : un chien, une chaise, une voiture,... L'objet à l'entrée du

deuxième stock a un modèle "GENERIC" pour accepter la nature de tous les objets transférés par la boîte de transfert.

Cet exemple montre bien qu'on ne peut pas déterminer statiquement le modèle d'un objet. Toutefois on peut parler d'un modèle minimum de l'objet défini par l'union des interfaces de ses représentations. BOCAL vérifie dans ce cas que toute instanciation est supérieure à ce minimum.

Ce modèle minimum peut ne pas avoir été explicitement déclaré. Ainsi dans l'exemple suivant (IV.28) le modèle minimum est défini par deux filles instances nommées moteur et carrosserie et par une fille référence nommée feu. On n'a pas à déclarer ce modèle abstrait parce que cette classe exige de tout objet qu'il respecte cet interface avant d'être transféré.

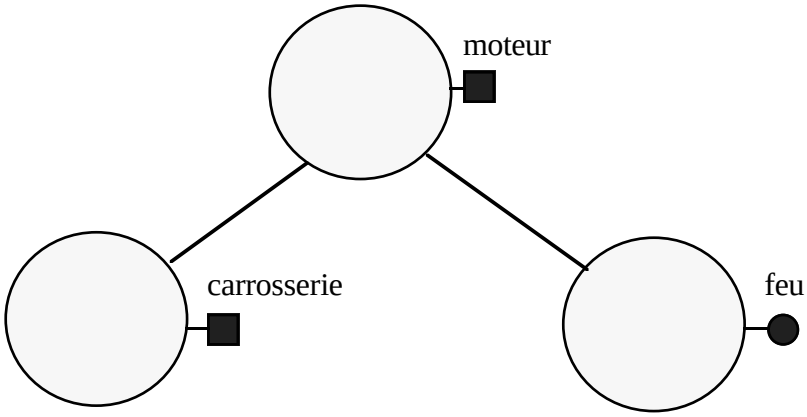


figure IV.28 le modèle minimum de cet objet contient un moteur et une carrosserie et peut recevoir un feu

Il faut noter que l'inconvénient de cette règle de modèle minimum est qu'elle supprime toutes les vérifications statiques de compatibilité.

On remarquera enfin que le modèle GENERIC (qui ne déclare aucune interface) n'a plus d'utilité sémantique : tout modèle peut le remplacer. Il reste néanmoins important pour la lisibilité.

IV.4 Conclusion

Les spécifications de BOCAL ont abouti à un compilateur écrit en langage C et un "linkeur-exécuteur". Toutefois, ces spécifications restent un sujet de recherche et nous pensons qu'on peut encore développer et affiner ses concepts. Mais le plus important reste de démontrer que BOCAL est autre chose qu'un langage informatique de plus, même s'il est novateur, et qu'il peut apporter une aide effective à la modélisation et à la simulation de systèmes complexes. En un mot que BOCAL est aussi un outil pratique.

Chapitre V

Modélisation et simulation en BOCAL

Dans ce chapitre, on définit la bibliothèque des modèles de base qui compléteront les modèles des fonctions prédéfinies. Ces modèles de base serviront dans la construction du simulateur BOCAL. Ensuite, on définit la bibliothèque des modèles de base nécessaires à la représentation des systèmes de production. Enfin, le simulateur et ces modèles seront utilisés dans la modélisation et la simulation d'exemples réels de lignes de production : un atelier de fabrication d'ascenseurs et une chaîne d'expédition de meubles.

La construction d'une bibliothèque adaptée à la simulation passe donc par la création des boîtes, les plus générales possible, effectuant les opérations de base. Le traitement des exemples réels nous permettra de compléter et d'enrichir la bibliothèque de simulation. Ces exemples nous permettront de mettre en évidence les qualités principales de BOCAL : modularité, généricité, extensibilité et réutilisabilité.

V.1 Les modèles de base

Le noyau de BOCAL regroupe les modèles des données élémentaires et les modèles des fonctions prédéfinies. Cette dernière famille propose les opérations génériques (création, destruction, affectation, lecture, écriture ...), les opérations logiques et arithmétiques, les opérations de comparaison, ainsi que les opérations d'empilement et de dépilement.

A partir de ces boîtes de base, on va commencer à créer nos propres boîtes BOCAL dite complexes qui s'obtiennent par encapsulation de représentants de modèles de base ou d'autres modèles complexes.

V.1.1 Les modèles de stockage

On définit trois types de modèles de stockage différents appelés : "**FIFO**", "**LIFO**" et "**DICTIONARY**". Chacun de ces modèles peut réaliser trois tâches différentes : empiler un

élément, dépiler un élément et consulter un élément. Les deux premières opérations tiennent à jour un ensemble des variables d'état permettant au monde externe de connaître l'état de la boîte de stockage.

On remarque que x_in et x_out sont du modèle "**Generic**" ce qui permet le stockage d'éléments de natures différentes sans aucune contrainte de compatibilité. L'opération de consultation met à jour la boîte "**current**" sans dépiler l'élément du stock, cette opération est réalisée grâce à un alias dynamique entre la boîte "current" et l'élément courant du stock. Au départ la boîte "current" n'a pas d'existence, alors la première opération de consultation permet à la boîte "current" de pointer sur le premier élément, puis chaque opération de consultation fait avancer l'alias à l'élément suivant. Arrivant au dernier élément du stock, la boîte "current" perd de nouveau son existence (en attendant une nouvelle opération du consultation)

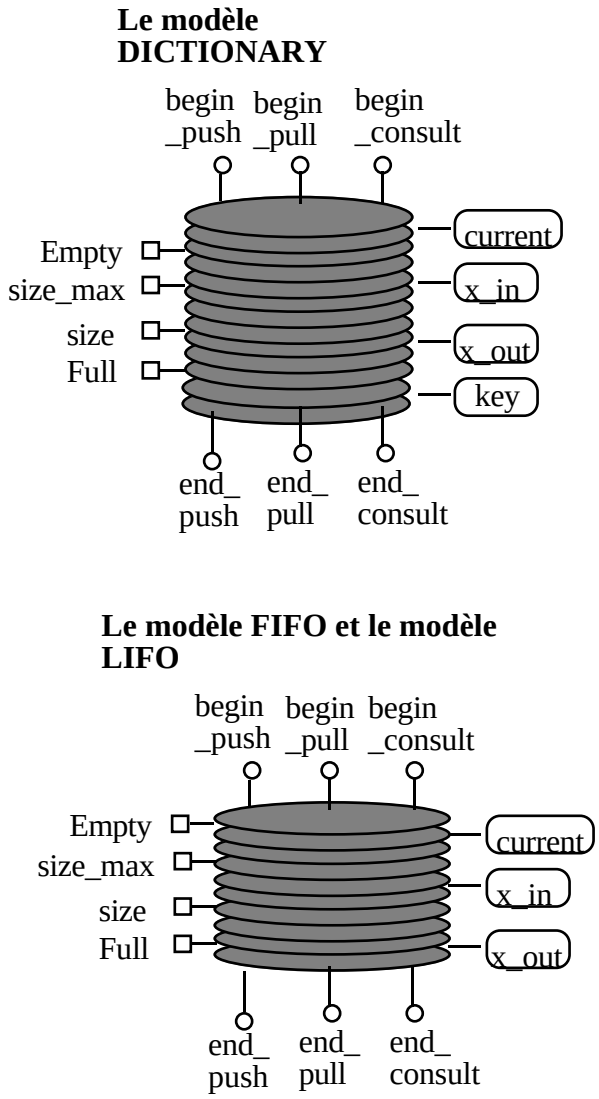


figure V.1 Les trois modèles de stockage

Dans le cas du modèle "**DICTIONARY**", les éléments sont stockés sous forme de couple où chaque couple est composé d'un objet et d'une clé ("**value**" et "**key**"). Ces couples sont ordonnés dans l'ordre croissant des valeurs des clés. L'opération d'empilement "push" consiste à créer un nouveau couple contenant l'objet "x_in" et la clé "key" et à l'insérer dans la liste des couples existant selon la valeur de "key". L'opération "pull" enlève le couple dont la clé est égale à "key" si l'objet "key" existe, le premier couple est enlevé dans le cas contraire. Dans les deux cas, l'objet du couple enlevé est transféré vers x_out. Les deux opérations "push" et "pull" mettent à jour les boîtes : "empty", "full" et "size".

L'opération de consultation met à jour la boîte "current" qui est en alias avec le couple courant de la liste. Chaque opération de consultation fait passer cet alias dynamique au couple suivant.

Ces trois modèles de stockage exigent un jeton spécifique à chaque opération sans lequel l'opération ne peut pas s'exécuter. On aurait pu offrir des modèles de stockage sans les jetons (begin_push, begin_pull, end_push et end_pull). Un tel modèle empilerait dès qu'il y a un élément en x_in et dépilerait un élément dès que x_out est vide mais son fonctionnement serait difficile à contrôler et il serait délicat de résoudre les conflits entre ces différentes tâches. Cependant, la bibliothèque des modèles de base offre actuellement un seul modèle identique au modèle de stockage FIFO, mais sans les jetons de contrôle. Ce modèle est appelé AUTO_FIFO.

V.1.2 Les modèles du gestion du temps

Pour simuler les activités réelles des systèmes de production, il faut gérer le temps. Suivant les logiciels de simulation, la gestion du temps est soit à pas constant selon une unité de temps choisie par l'utilisateur soit événementielle (i.e. l'incrémentation du temps se fait d'une date d'événement à l'autre). La première solution présente plusieurs inconvénients. Elle peut notamment s'avérer coûteuse lorsque le pas de simulation choisi est trop petit, ou introduire des erreurs (événements non répertoriés) s'il est trop grand.

En BOCAL la gestion du temps est événementielle. Toutefois, elle n'est pas inscrite "en dur" dans le langage (elle se fait grâce à des boîtes BOCAL). On pourrait donc envisager d'autres approches.

La gestion du temps consiste à avoir une pile d'événements et deux modèles de boîtes, le premier modèle permet d'empiler un événement dans la pile d'événements et le second modèle permet de dépiler le premier événement (dans l'ordre chronologique) et de faire

avancer le temps à la date de l'événement dépilé. La boîte "**clock**" contient la pile d'événements "stack" qui est du modèle **dictionnaire** et la valeur courante du temps "TIME". En plus, elle permet de stocker un événement qui est un couple dont la clé est une "date" et l'objet associé est l'adresse de la boîte "dest" (par exemple : wait ou waito) qui a empilé l'événement. Les événements sont stockés dans l'ordre chronologique de leur arrivée.

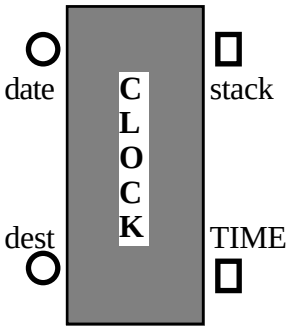


figure V.2 La représentation fermée du modèle clock

La boîte "**clockSystem**" est chargée de "faire avancer le temps" c'est-à-dire que chaque fois qu'on l'active cette boîte, elle dépile un événement de la pile, fait avancer le temps à la date de cet événement et crée un jeton pour la destination "dest" (ou les destinations) qui a inscrit cet événement. Voici la représentation ouverte du modèle clockSystem :

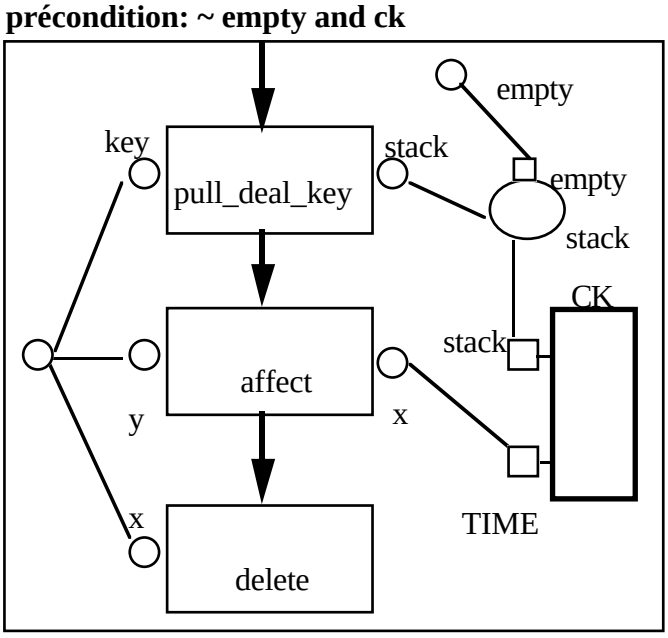


figure V.3
le modèle clockSystem

Pour plus de lisibilité, on utilise des flèches (en gras) pour présenter le flux du jeton d'activation. La première flèche présente l'alias entre la boîte "$active" et la boîte "$start" de la boîte "pull_deal_key", par contre la deuxième flèche, ainsi que la troisième, relient une boîte fille "$stop" à une boîte fille "$start". La boîte "deal_pull_key" dépile le premier événement de la pile "stack" et affecte la date de l'événement dépilé, qui est la plus petite, à la boîte "key".

Toutefois, pour conserver au langage un caractère simple, ces deux boîtes ne sont pas utilisées directement. Le seul moyen d'agir sur le temps est donc d'utiliser l'une des boîtes **WAIT** ou **WAITO** qui fait attendre la boîte pendant un temps **t** (pour wait) ou jusqu'à une date **date** (pour waito). La figure suivante montre la représentation ouverte de ces deux modèles :

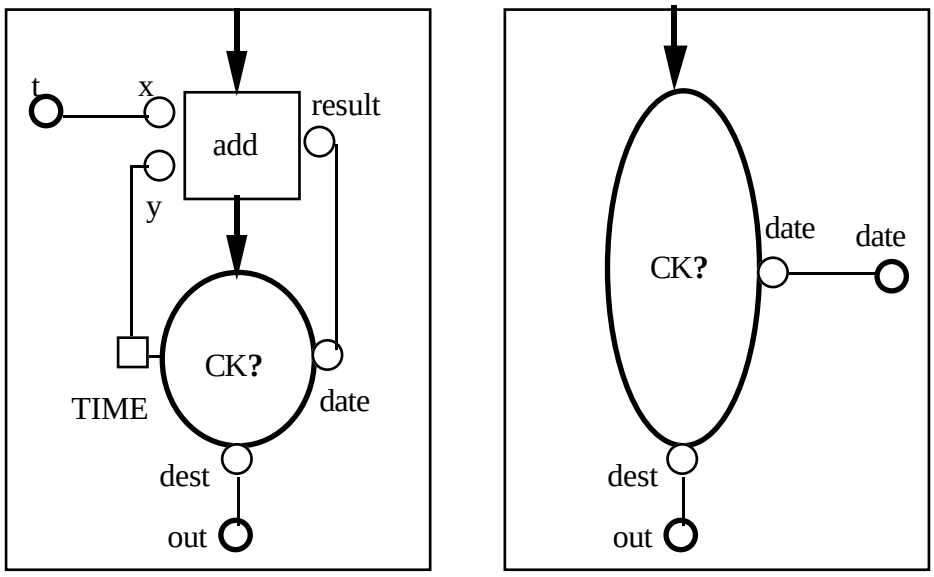


figure V.4 le modèle wait (t) et le modèle waito (date)

Ainsi, si on veut modéliser une boîte de transfert qui nécessite un temps donné, on écrira (ou plutôt on dessinera) :

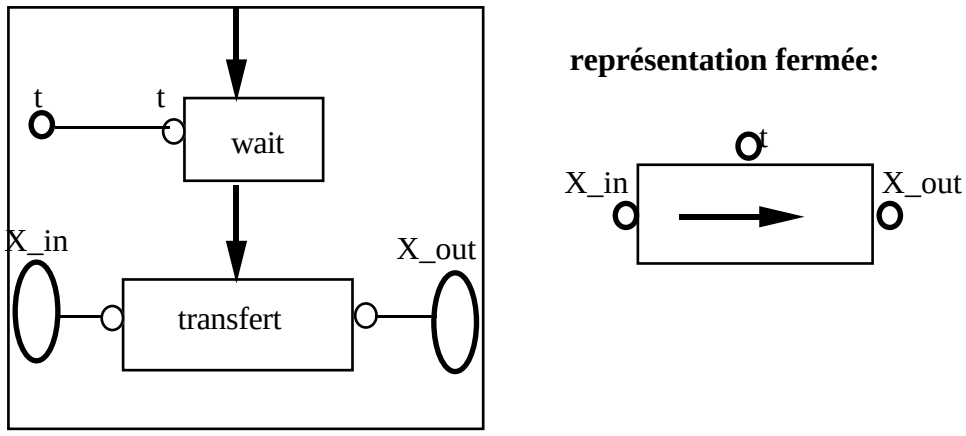


figure V.5 le modèle du transfet temporisé (wait_transfert)

La figure V.5 montre que la boîte "**transfert_wait**" ne cherche pas à aliasser la "clock" de la boîte wait parce que la boîte wait effectue une recherche dynamique de la boîte "clock" la plus proche, en effet, le modèle wait déclare la boîte clock(ck?).

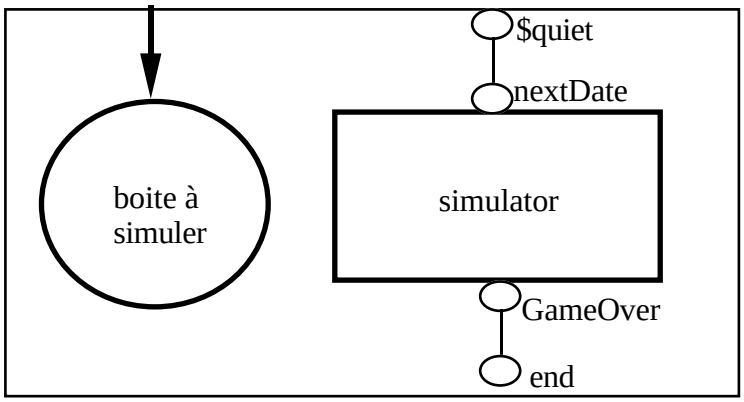
L'activation de la boîte "**transfert_wait**", si on suppose qu'elle intervient au temps t0, provoque celle de la boîte wait. Celle-ci ne s'éteindra qu'après un temps t, c'est-à-dire au temps t0+t. Donc, le jeton **$stop** de la boîte wait ne sera créé qu'à cette date, et puisqu'il est nécessaire à l'activation du transfert, celui-ci sera effectivement réalisé au temps t0+t.

L'exemple précédent montre qu'on peut concevoir les boîtes qui exigent du temps pour remplir leurs tâches en utilisant les deux boîtes "wait" et "waito", et indépendamment de la boîte "clockSystem".

V.1.3 Le simulateur BOCAL

Le simulateur BOCAL doit assurer l'avancement du temps et la gestion de la pile d'événements créés par le modèle simulé. Il doit également permettre au modèle simulé de demander à l'utilisateur d'entrer le temps total de la simulation et de récupérer un jeton à la fin de la simulation.

Pour cela, on a défini trois modèles de boîtes : **simulator**, **init_simulator** et **run_simulator**. Le modèle run_simulator contient deux boîtes : le modèle qu'on veut simuler et une boîte "simulator" qui répondra aux événements provoqués par le modèle à simuler. La figure suivante :



postcondition: end

figure V.6 le modèle run_simulator

La simulation du fonctionnement d'une boîte consiste à activer la boîte run_simulator qui active à son tour la boîte à simuler, chaque fois la boîte "run_simulator" devient endormie, elle génère le jeton $quiet pour que la boîte "simulator" fasse avancer le temps "**TIME**". Si la pile d'événements est vide la boîte "simulator" génère un jeton "**GameOver**" qui mettra fin à l'activation de la boîte "run_simulator".

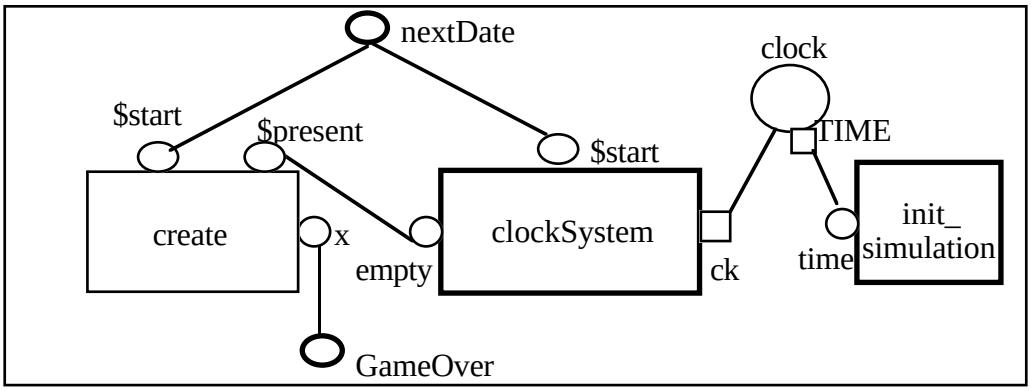


figure V.7 le modèle simulator

Le modèle "simulator" contient une boîte "clockSystem" bien instanciée permettant ainsi à toutes les boîtes qui cherchent une horloge (clock) pendant la simulation de prendre celle de la boîte clockSystem, tout particulièrement les boîtes "wait" et les boîtes "waito". L'activation de la boîte "simulator" implique l'activation de la boîte "clockSystem" si la pile d'événements n'est pas vide et l'activation de la boîte "create" dans le cas contraire. La boîte "init_simulator" permet la lecture de "time_simulation" qui est la date de fin de simulation, elle compare cette valeur et la valeur de TIME chaque fois que la clockSystem avance le temps, elle crée un jeton TimeOut une fois que TIME a dépassé la valeur de time_simulation.

On remarque que le simulateur BOCAL est entièrement programmé en BOCAL (en utilisant très peu des boîtes élémentaires) ce qui permettra aux développeurs de le modifier facilement en échangeant certains modèles par d'autres compatibles et plus performants.

V.1.4 Les générateurs d'objets

Dans toute application, on a besoin de modéliser l'arrivée des objets dans une file d'attente (des voitures qui arrivent devant un feu rouge, des clients qui arrivent devant un guichet, etc...). En général, l'arrivée des clients dans une file d'attente est supposée aléatoire selon une distribution poissonienne et par conséquent le temps qui sépare l'arrivée des deux objets suit une loi exponentielle.

En BOCAL, le modèle "**rand_generator**" crée un objet après un temps calculé par une boîte élémentaire représentant la fonction aléatoire qui simule l'arrivée des objets. Toutefois, la fonction aléatoire n'est pas locale pour pouvoir l'échanger par une autre fonction statiquement lors de la définition d'un modèle englobant ou même dynamiquement pendant la simulation. Ainsi, ce générateur peut générer tout type d'objet et on n'a pas à définir un générateur des véhicules et un autre pour les animaux par exemple.

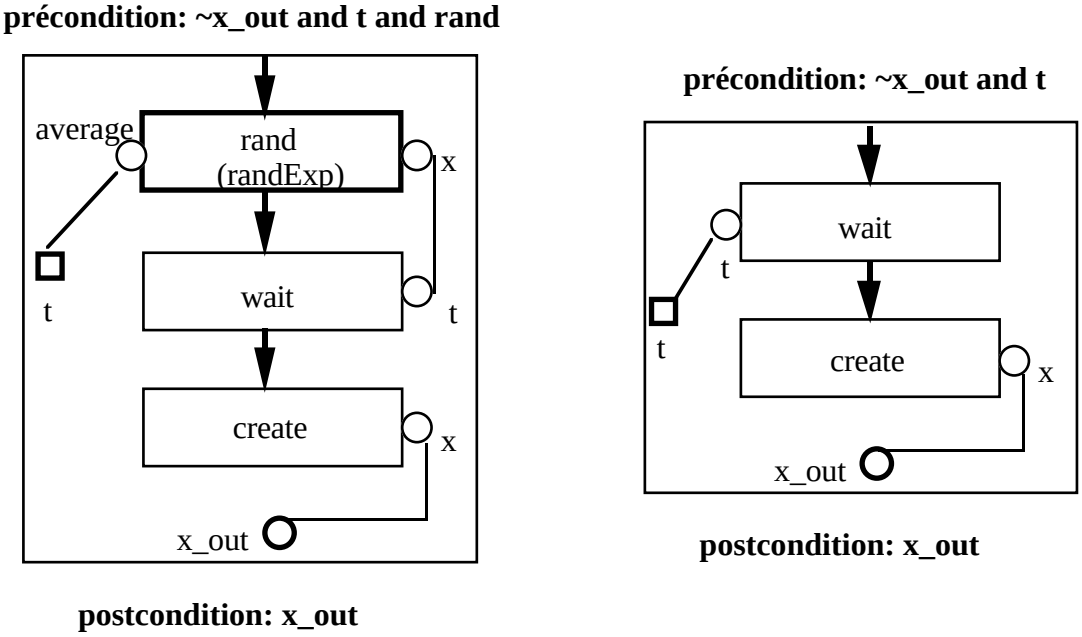


figure V.8 le modèle rand_generator et le modèle cst_generator

Le modèle "**cst_generator**" crée un objet après un temps constant t. Mais ceci n'empêche la valeur de t de changer pendant la simulation.

Les préconditions indiquées dans la figure V.8 montrent que chaque générateur s'active une fois que l'objet précédemment créé est consommé ou transféré ailleurs. Ceci impose que l'objet créé soit transféré immédiatement, sinon la boîte "**x_out**" risque d'être bloquée pendant un certain temps qui perturbera les résultats de la simulation si on espère avoir des objets arrivant selon une certaine distribution aléatoire.

Afin de remédier à ce problème, on propose de placer une file AUTO_FIFO juste avant la sortie du générateur comme le montre la figure V.9 (le modèle stk_generator). La boîte FIFO est une boîte compatible avec la boîte AUTO_FIFO qui empile un élément dès que x_in est présent et dépile un élément dès que x_out est vide (sans les jetons de contrôle begin_push, end_push, begin_pull et end_pull). Le modèle "**stk_generator**" possède deux jetons : "**begin**" pour mettre le générateur en marche et "**end**" pour éteindre le générateur.

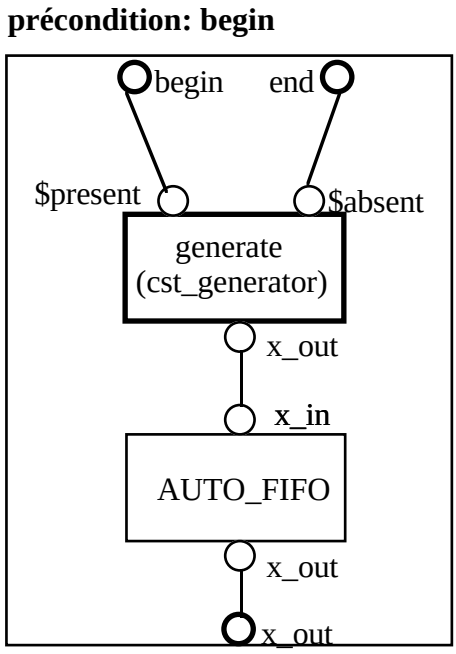


figure V.9 Le modèle stk_generator

V.1.5 Les accumulateurs

La figure V.5 montre le modèle "**transfert_wait**" qui reçoit un objet générique en entrée **x_in** et le transfère vers la sortie **x_out** pendant un temps **t**. Mais durant ce temps, la boîte "transfert_wait" bloque l'entrée x_in et elle ne peut pas recevoir d'autres objets avant que le transfert ne soit fait. D'où le modèle "**accumulator**", ce modèle reçoit un objet en entrée qui sortira après un temps t qui peut être spécifique à chaque objet entrant. L'objet

entrant sera immédiatement empilé dans un stock et ne bloque pas l'entrée pendant son transfert. Les objets sortant entrent dans une boîte "**AUTO_FIFO**" pour que les objets ne restent pas dans l'accumulateur plus de temps que prévu.

Ce modèle peut modéliser un tapis roulant ou un tronçon de route où les objets (véhicules) entrent avec des vitesses différentes ce qui implique qu'un objet pourra dépasser d'autres objets pendant son passage dans l'accumulateur.

V.1.6 Exemple : Un carrefour routier à stop

Cet exemple consiste à simuler le passage des véhicules dans une rue à sens unique et prioritaire par rapport à une deuxième rue (voir figure V.10)

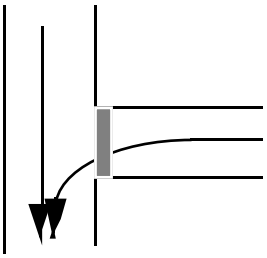


figure V.10 Carrefour à stop

La modélisation en BOCAL de cet exemple donne un modèle traduisant la description physique et fonctionnelle de l'exemple.

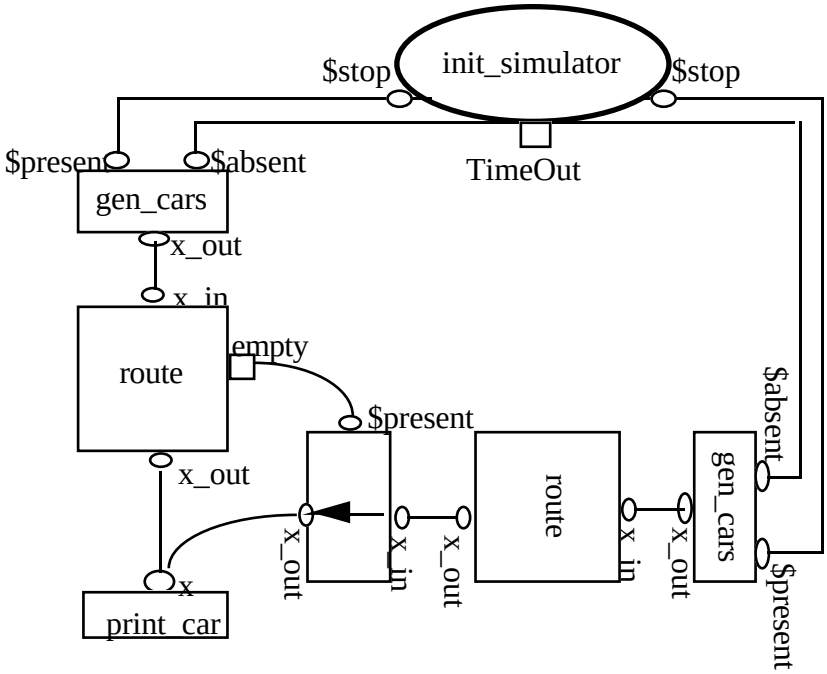


figure V.11 Le modèle carrefour à stop

La boîte "init_simulator" demande la date de fin de la simulation. Ensuite, les deux générateurs des voitures "gen_cars" sont mis en marche. Le modèle gen_cars est un "**rand_generator**" suivi des opérations qui spécifient le véhicule produit (numéro de série, couleur, une marque, etc...). Les deux générateurs sont contrôlés par le jeton "**TimeOut**" qui fait arrêter les deux générateurs de véhicules.

Le modèle "route" est un accumulateur où le temps de la traversée est généré aléatoirement par une boîte "random" ce qui permettra éventuellement à un véhicule de dépasser un autre à l'intérieur de l'accumulateur. Le modèle "printcar" constitue la poubelle de la simulation où les véhicules disparaissent après leur parcours et permet d'imprimer des informations concernant le véhicule arrivé.

V.2 La modélisation des systèmes de production

L'analyse de deux exemples réels de lignes de production (un atelier de fabrication d'ascenseurs et une chaîne d'expédition de meubles) nous a permis de dégager les boîtes de base de la simulation des systèmes de production. La modélisation de ces exemples illustrent aussi la démarche du modélisateur : décomposition du problème en sous-problèmes puis recomposition des sous-problèmes (analyse descendante suivie d'une programmation ascendante).

Cette première analyse fait apparaître la nécessité de créer :

1) les boîtes circulant dans l'atelier (les pièces).

2) les boîtes simulant les machines d'usinage.

3) les boîtes effectuant le transport des pièces, ainsi que le choix de la destination.

4) les boîtes simulant les pannes des machines, ainsi que les réparations.

5) les boîtes absorbant les éléments en fin de simulation et réalisant des statistiques et analyse des données.

V.2.1 Les pièces

Une pièce est un ensemble d'attributs qui déterminent ses caractéristiques. Par exemple la boîte qui doit représenter les profilés métalliques circulant dans la ligne de l'atelier de fabrication d'ascenseurs contient les instances qui permettent de la décrire et de la différencier.

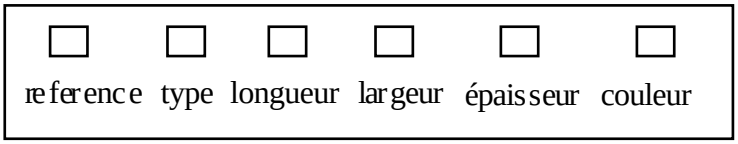


figure V.12 le modèle profilé

La solution généralement utilisée en production pour repérer les pièces à usiner consiste à leur attribuer un numéro (une référence). Il semble donc logique de procéder de la même manière lors de la simulation. La boîte contiendra donc une instance (de type entier) qui indiquera son numéro. En outre, elle possédera des instances décrivant son aspect (sa couleur, ses dimensions ...) et son type.

Avant de fixer les caractéristiques des boîtes représentant les pièces, il faut s'intéresser aux résultats que l'on désire obtenir en fin de simulation. Pour conserver une trace de leur passage, on a décidé de munir ces éléments, d'une boîte de type "**FIFO**", appelée **historic**, dans laquelle seront empilées des "étiquettes" correspondant aux différentes machines qu'elles traverseront. L'analyse de la simulation se fera alors en dépilant cette historique quand l'élément arrivera à la fin de la chaîne. Par dualité, on installe des boîtes historiques à l'intérieur des machines.

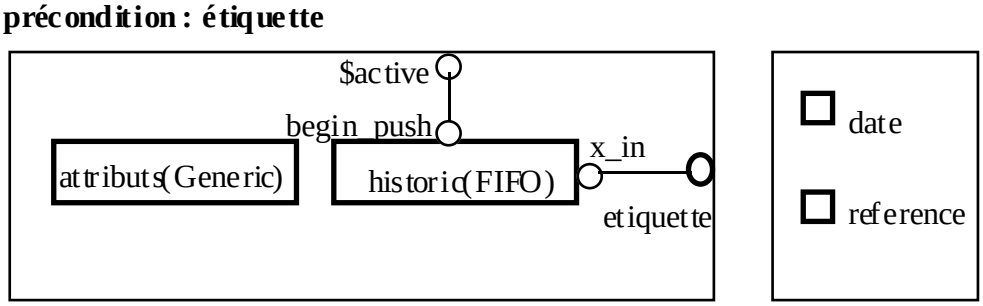


figure V.13
le modèle d'une pièce et le modèle d'une étiquette

Les pièces circulent entre les machines en tant qu'objets passifs, mais chaque pièce est capable de s'activer si elle reçoit une étiquette. Une étiquette contient la date de l'entrée de la pièce dans la machine, ainsi que la référence de la machine.

V.2.2 Les machines

Dans les exemples traités, les machines modélisées ne créent pas de nouveaux types d'éléments, elles se contentent d'en modifier un ou plusieurs attributs.

Le modèle "**usinage**" bloque la pièce pendant le temps d'usinage "**t**" et peut modifier certaines caractéristiques de la pièce, il met à jour le temps d'occupation de la machine. Il contient un générateur aléatoire qui simule la tombée en panne de la machine, la machine reste bloquée pendant la présence du jeton "**failure**". La présence de ce jeton est signalé immédiatement aux équipes de maintenance voir le paragraphe suivant.

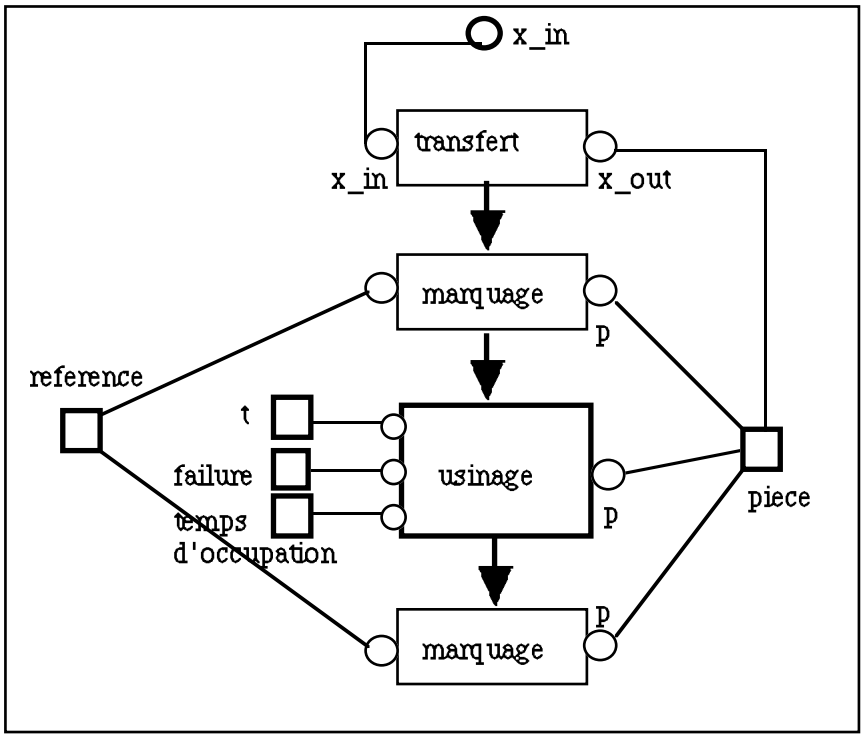


figure V.14
le modèle machine

Chaque machine possède une boîte fille instance nommée "**référence**" qui sert à l'identifier. Cette référence serait inscrite sur les deux étiquettes générées par les boîtes de marquage qui entourent la boîte d'usinage. La boîte marquage crée et initialise l'étiquette et l'empile dans l'historique de la pièce.

BOCAL possède deux modèles de machine différents : le premier modèle présente une machine avec un bouton "power on" et un autre "power off", cette machine s'active sur le bouton "power on" et peut recevoir des pièces sans arrêt. Par contre le deuxième modèle présente une machine qui s'active dès qu'elle reçoit une pièce.

V.2.3 La simulation des pannes

Le but est de fabriquer des boîtes BOCAL, pouvant être mises dans les boîtes simulant des machines, pour générer des pannes. Une panne est une boîte contenant des instances décrivant la panne (type et gravité). Cette panne est créée par une boite "**simulate_failure**". Une fois la panne "créée", on bloque le fonctionnement de la boîte machine et on attend qu'une boîte réparation vienne détruire la panne.

1) le générateur de pannes (simulate_failure)

Cette boite crée une panne en utilisant les boîtes "rand_generator" et "generate_failure". La boîte "rand_generator" génère une panne après un temps t de son activation, cette boîte peut être remplacée par d'autres générateurs. La boîte "generate_failure" donne à la panne générée un type donné et une gravité aléatoire. La gravité suit, par exemple, une loi exponentielle de moyenne "average". La boîte "generate_failure" peut être également remplacée par d'autres boîtes compatibles.

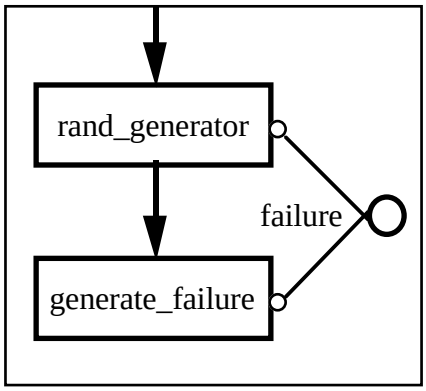


figure V.15 le modèle simulate_failure

Par exemple, on peut remplacer la boîte rand_generator par une boîte qui produit une panne selon un taux de panne et sans attente, ce modèle est intéressant pour une machine qui s'active chaque fois qu'elle reçoit une pièce. A chaque activation de la machine, elle peut tomber en panne avec un certain pourcentage.

2) la boîte (repair_failure)

Cette boîte représente une équipe de maintenance et peut recevoir plusieurs machines en panne. Cette boîte reçoit en entrée plusieurs machines en panne (failures est une fille multiple), elle choisit une machine en panne disponible (non bloquée par une autre équipe de réparation). La boîte "repair_failure" doit bloquer la panne (la machine sélectionnée),

afin qu'une autre équipe de réparation ne travaille pas aussi sur cette panne et ne pas consommer la panne avant que le temps de réparation ne se soit écoulé. De plus, le temps de réparation est proportionnel à la gravité de la panne multipliée par l'efficacité de l'équipe de réparation.

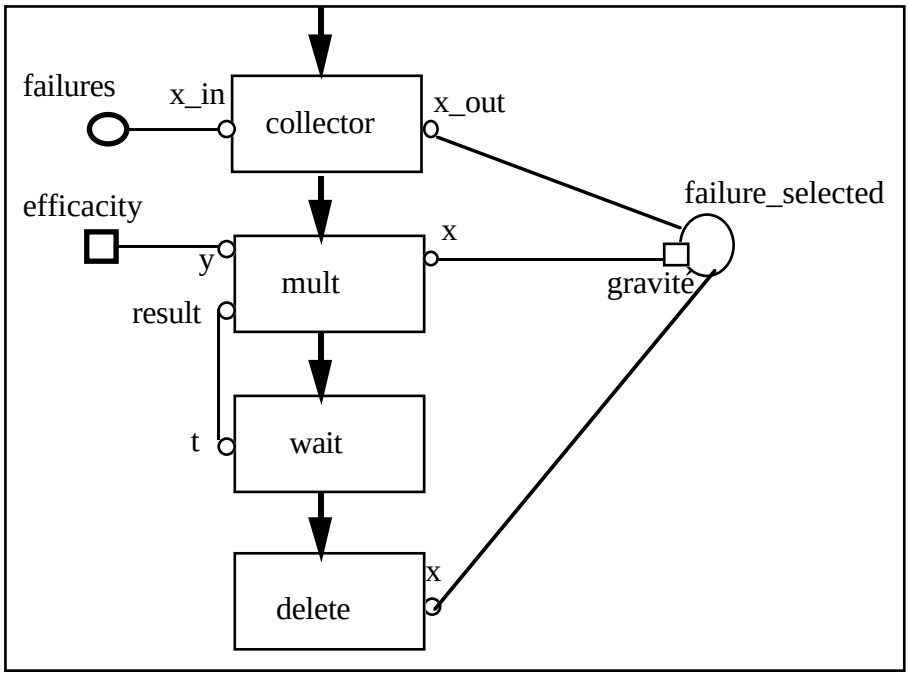


figure V.16 le modèle repair_failure

3) Exemple

Le modèle d'un atelier composé de 3 machines (en série par exemple) et 2 équipes de réparation est le suivant :

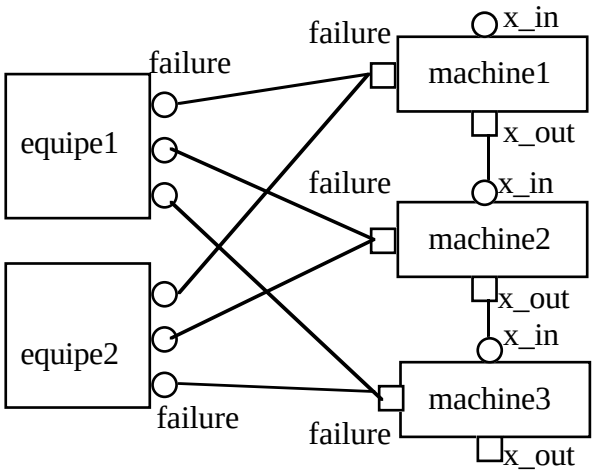


figure V.17 exemple

V.2.4 Le traitement en fin de simulation

Ce sont les boîtes qui se trouvent en fin de chaîne. Elles doivent absorber les éléments qu'elles reçoivent en entrée (pour ne pas bloquer ceux qui suivent) et fournir le compte-rendu associé. En somme, elles dépilent et éditent l'historique puis détruisent l'élément.

Dans un version ultérieure de BOCAL, on aura des boîtes qui font des statistiques et qui visualisent les résultats sous forme de courbes, camemberts ou autres. En attendant les exemples qui suivent utiliseront des boîtes style "**print_car**" utilisées dans l'exemple V.11.

V.3 Modélisation des exemples réels

V.3.1 La chaîne de fabrication d'ascenseurs

La ligne de fabrication à modéliser produit des sous-ensembles à partir de profilés métalliques. Trois opérations se succèdent :

1) La découpe des profilés sur deux scies pour la mise à longueur.

2) Le poinçonnage de trous dans les profilés à l'aide de quatre machines, selon le type du profilé (type A, machine 1; type B, machine 2 ...).

3) Le regroupement des pièces d'un même sous-ensemble.

Deux modélisations seront effectuées : l'une en opérant un tri des profilés pour les envoyer sur l'une des quatre machines montées en parallèle, l'autre en faisant passer chaque profilé sur les quatre machines montées en série.

1) montage parallèle

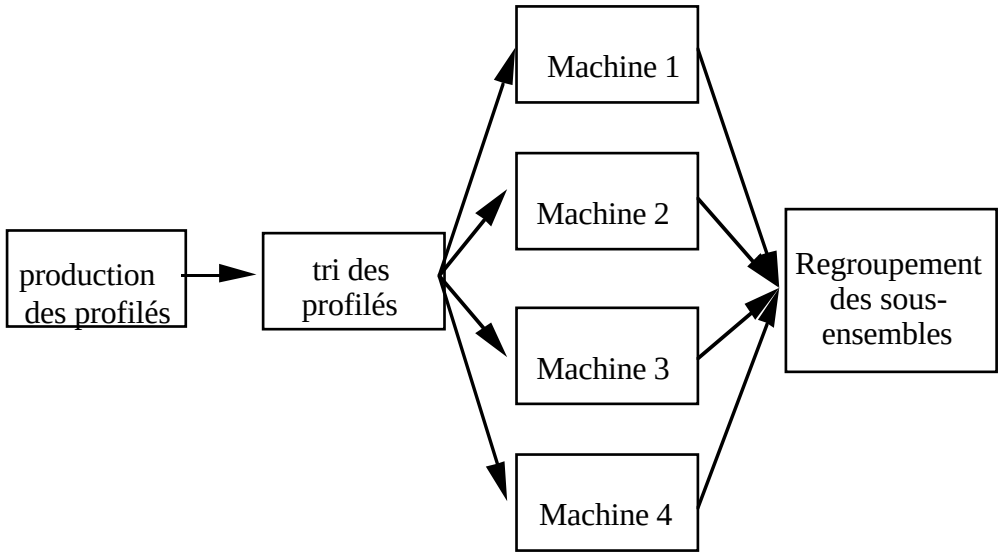


figure V.18 montage parallèle

La figure V.18 montre la description de la ligne de fabrication en montage parallèle. La modélisation de ce montage (figure v.19) utilise une boîte génératrice qui produit des profilés métalliques de types différents (1, 2, 3 ou 4), le type de chaque profilé est produit aléatoirement par une boîte (**randU_int**).

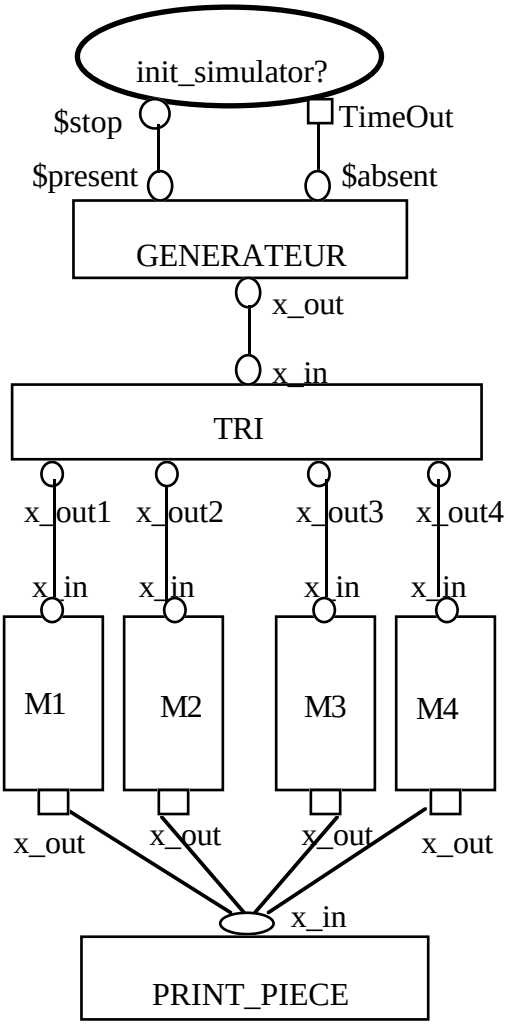


figure V.19 modélisation de l'atelier.montage parallèle

La boîte **tri** transfère l'élément en entrée vers l'une des quatre sorties selon le type (1,2,3 ou 4) du profilé. Elle se compose tout simplement de quatre boîtes de transfert conditionnelles (**transfert_cond**) qui effectuent des transferts si le type de l'élément en entrée est identique à leur boîte fille "ref" (on utilise la boîte de test de l'égalité **eq**).

L'expérience montre que ce type de boîte est souvent utilisé mais très difficile à rendre réutilisable parce que le nombre de sorties n'est toujours pas le même et que le test porte sur des filles différentes. BOCAL, dans sa version actuelle, ne permet pas d'ajouter une précondition supplémentaire dynamiquement (sauf la précondition constituée par les jetons $start, $stop, $present et $absent).

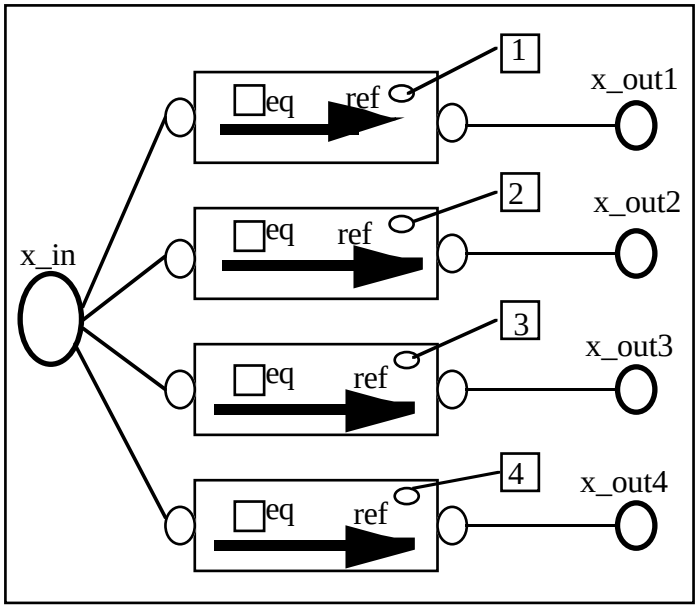


figure V.20 la boite tri

2) montage série

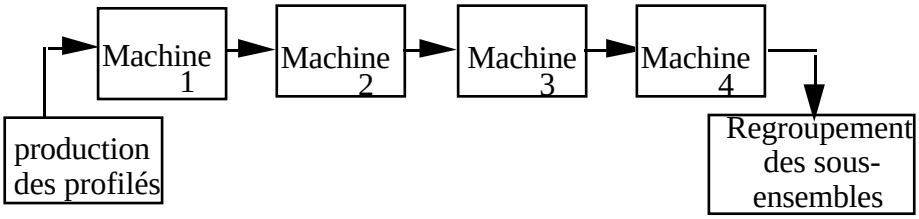


figure V.21 montage série

La modélisation de ce montage utilise les mêmes boîtes que la modélisation précédente mais organisées différemment comme le montre la figure V.22. Le profilé arrive devant une machine alors

- soit il est usiné par la machine si elle sait traiter ce type de profilés (avec un temps d'usinage et un temps de passage vers la machine suivante)
- soit il est transféré vers la machine suivante (avec un temps de passage).

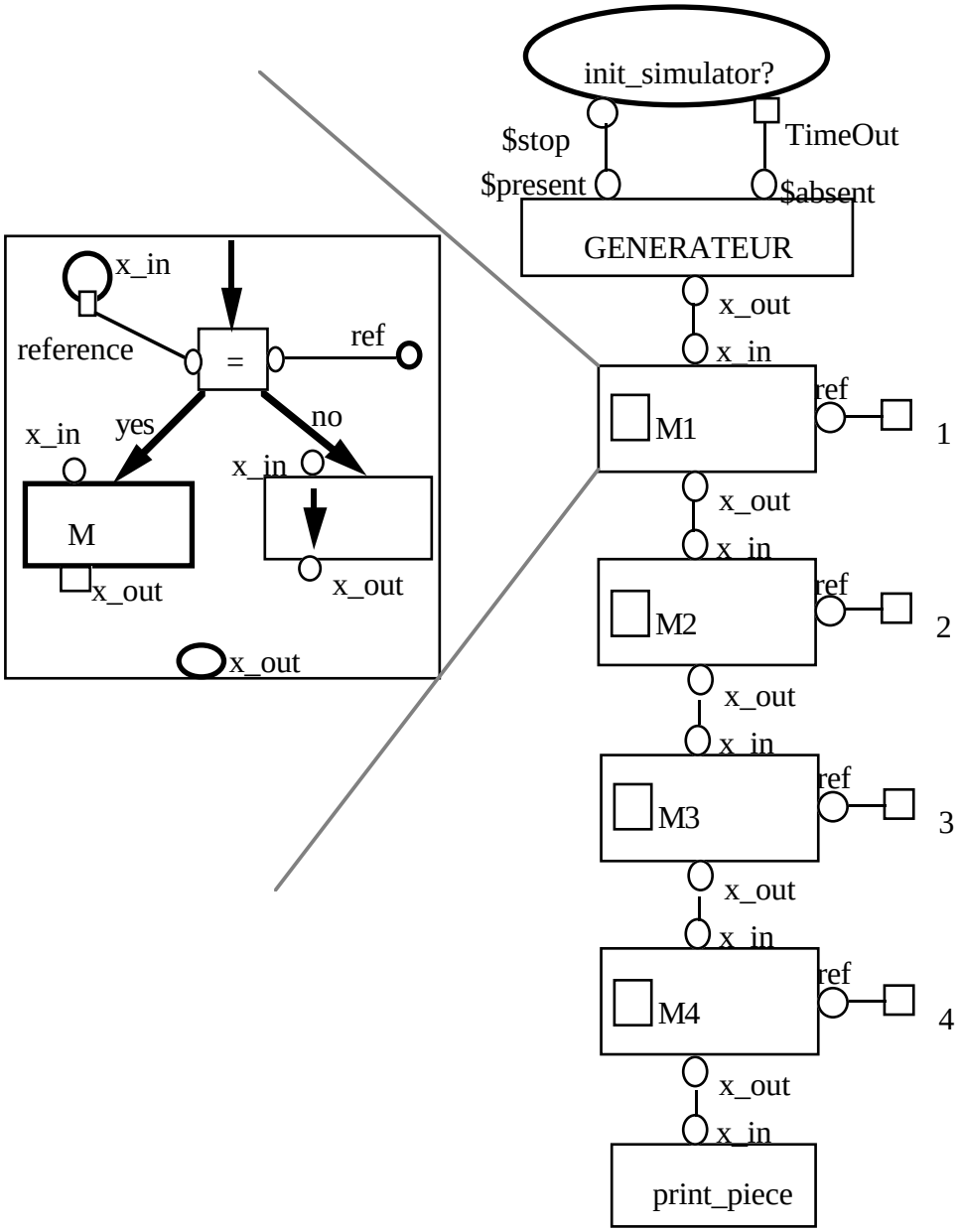


figure V.22 modélisation de l'atelier montage série

Cette modélisation utilise un modèle composé à la place de la machine. Ce modèle est composé d'une machine, d'une boîte de transfert temporisé et d'une boîte de test qui sert à aiguiller la pièce soit vers la machine soit vers la boîte de transfert.

V.3.2 Expédition des meubles

1) Présentation du problème

Il s'agit de modéliser une chaîne d'expédition qui prend en charge les meubles à la sortie de l'usine et les amène jusqu'aux camions. Pour l’expédition, les meubles sont

organisés par tournées. Une tournée étant l'ensemble des livraisons que doit faire un camion. De plus, les tournées sont composées de telle sorte qu'à chaque livraison on n'ait qu'une seule tournée à décharger. Les meubles arrivent finis de sept quais différents (H1, H2, A1, D1, D2, B1, B2). Ensuite, ils empruntent tous la même bande transporteuse qui les amène à un quai de présentation. Une équipe les transporte alors vers une zone de stockage correspondant à leur tournée (une des zones numérotées de 1 à 21). Une fois que la tournée est complète, elle avance automatiquement vers les zones Ai. Ensuite, un chariot prend les tournées prêtes et les dépose sur un quai de chargement où une des cinq équipes va charger la tournée dans le camion.

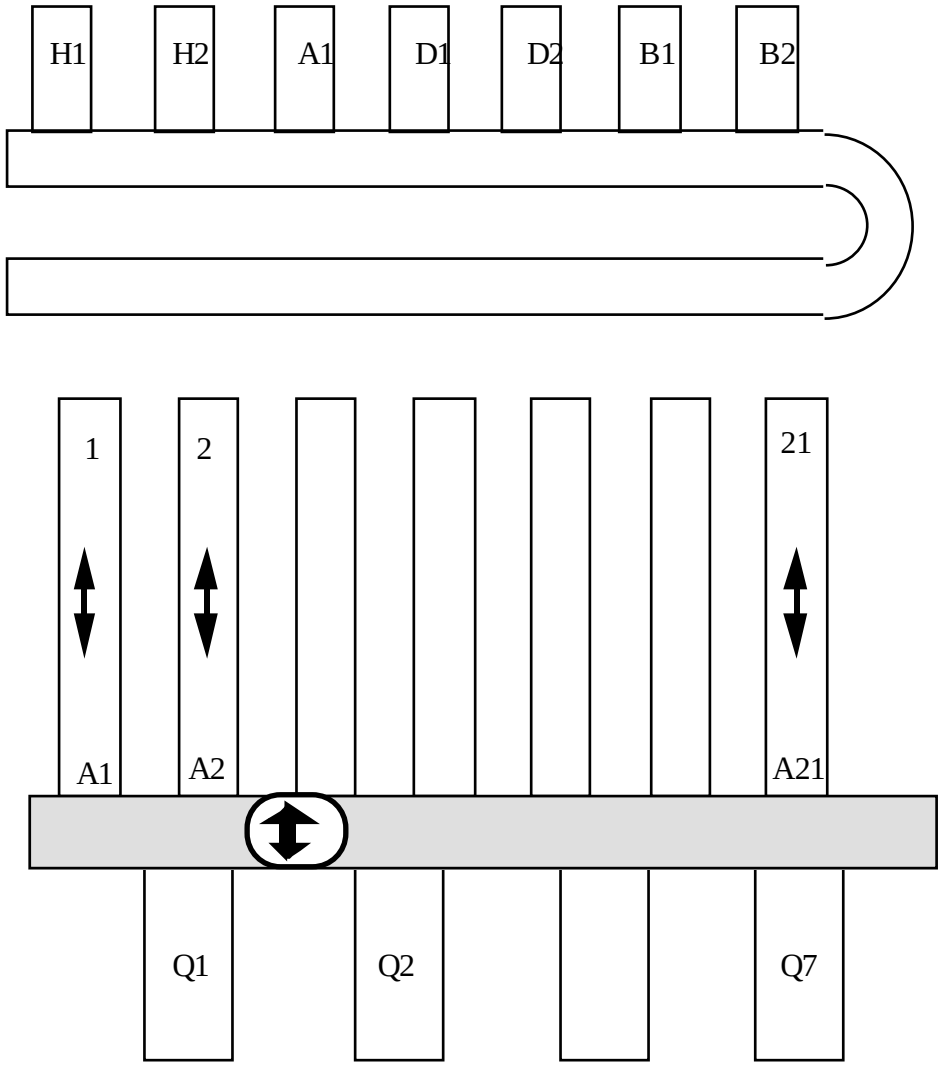


figure V.23 l'expédition des meubles

2) Modélisation en BOCAL

Pour rendre le programme BOCAL plus facile à appréhender, on a divisé la chaîne d'expédition en trois parties. La première partie comprend la représentation des quais d'arrivée des meubles (H1, ...), la bande transporteuse et l'équipe s'occupant de trier les

meubles en fonction de leur numéro de tournée. La deuxième partie regroupe les zones de stockages. La troisième partie simule le chariot et le chargement des tournées dans les camions.

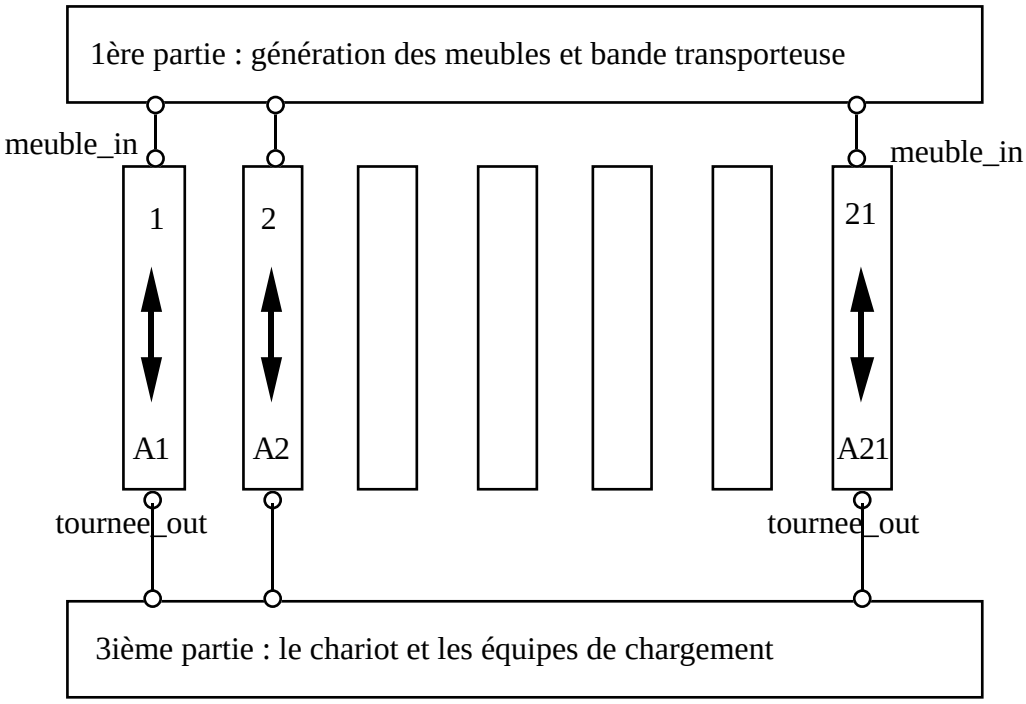


figure V.24 la première décomposition de l'atelier

3) La première partie

Les quais H1, H2, ... sont modélisés par des "génératrices" de meubles qui alimentent le reste de la chaîne. Un meuble est une boîte contenant certaines caractéristiques pour l'identifier, ainsi qu'un numéro de tournée fourni par le quai générateur. Le tapis roulant est représenté par un accumulateur. Les meubles sortant des quais sont portés par le tapis pendant un temps qui dépend du quai de sortie (le meuble sortant de H1 met plus de temps que celui sortant de H2).

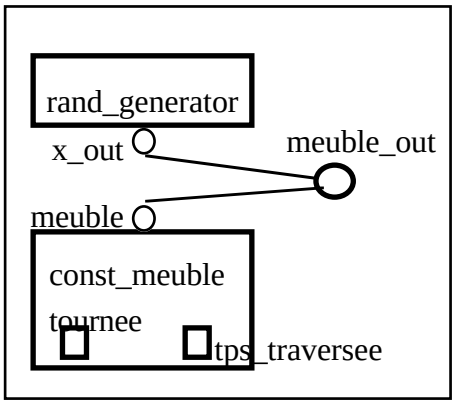


figure V.25 le modèle d'un quai

La boîte "const_meuble" caractérise le meuble généré et lui donne un numéro de tournée (entre 1 et 21) et un temps pour la traversée de la bande transporteuse (ce temps de traversée sert à modéliser la position du quai par rapport à la bande transporteuse). Les boîtes "rand_generator" et "const_meuble" sont exportables afin de permettre la modélisation des quais dans autres scénarios (par exemple des fonctions aléatoires différentes).

La première équipe reçoit les meubles qui arrivent au bout de la bande transporteuse. Cette équipe envoie donc le meuble qu'elle reçoit sur la zone de stockage correspondant à sa tournée. Cette équipe est modélisée par une boîte tri (voir figure V.20) avec 21 sorties possibles selon la tournée du meuble reçu en entrée.

Les tournées sont des files FIFO et se trouvent dans les zones de stockages.

4) La deuxième partie

Le modèle de cette chaîne d'expédition possède 21 zones de stockages correspondant aux 21 tournées. Le meuble reçu par la boîte "tri" est envoyé vers l'une de ces zones de stockage.

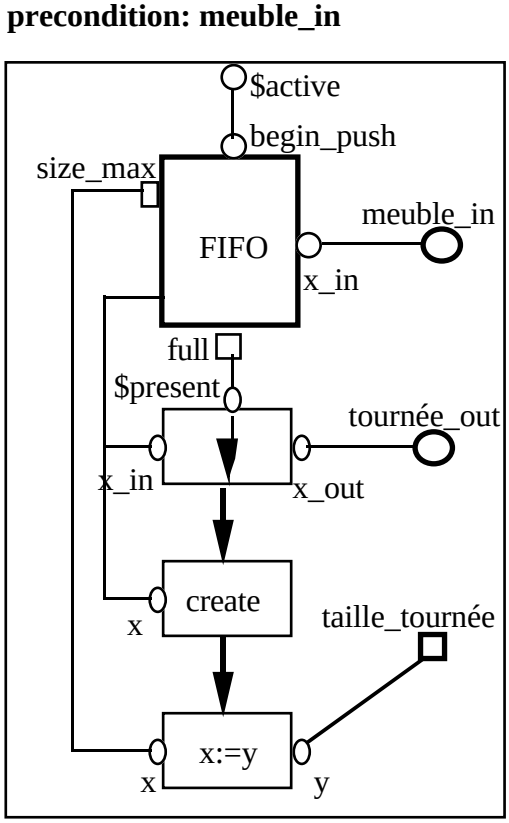


figure V.26 le modèle d'une zone de stockage

5) La troisième partie

Le chariot est une boîte un peu plus complexe. Il comprend un collecteur qui prend une des entrées disponibles (en l'occurrence une tournée complète). Ensuite, il envoie la tournée sur un switch qui aiguillonne la tournée vers un quai de chargement libre. Une boîte **wait** liée à ces boîtes permet de bloquer la tournée en cours de déplacement pendant le temps nécessaire.

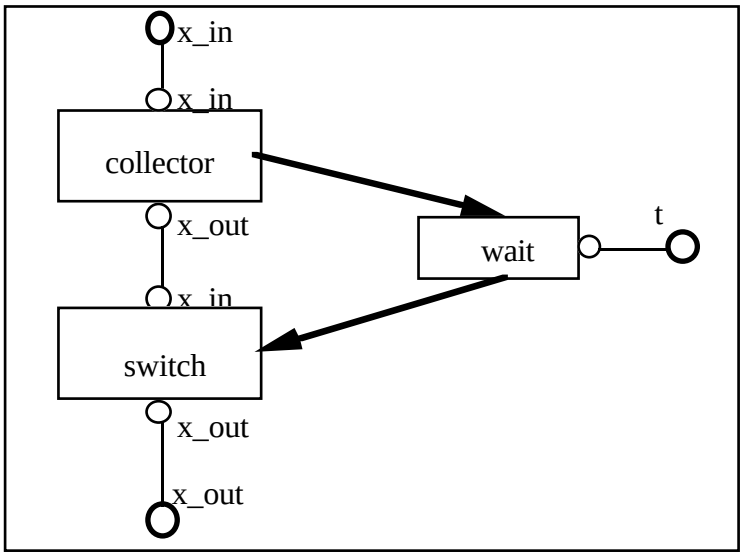


figure V.27 modélisation du chariot en BOCAL

Dans cet exemple, on se contente de modéliser une équipe de chargement comme un chariot. Elle cherche un quai plein, prend la tournée et la dépose sur un camion.

D'une manière générale, on modélise une équipe par : un calendrier de travail, nombre de personnes, la tâche à remplir (dans notre cas c'est le chargement) et bien d'autres caractéristiques...

6) Le modèle détaillé de la chaîne d'expédition

La figure suivante montre le modèle de la chaîne d'expédition. Dans cette représentation, on peut facilement modifier le nombre d'équipes travaillant à chaque niveau de la chaîne ou augmenter le nombre de quai de chargement, etc... On s'aperçoit rapidement que ce modèle reflète directement la description physique et fonctionnelle de la chaîne d'expédition. Ce modèle envoie les tournées en fin de simulation à la poubelle.

Ce modèle ne s'occupe pas des résultats de la simulation. Toutefois, on peut insérer, n'importe où dans le modèle, des boîtes d'affichage et de calculs statistiques.

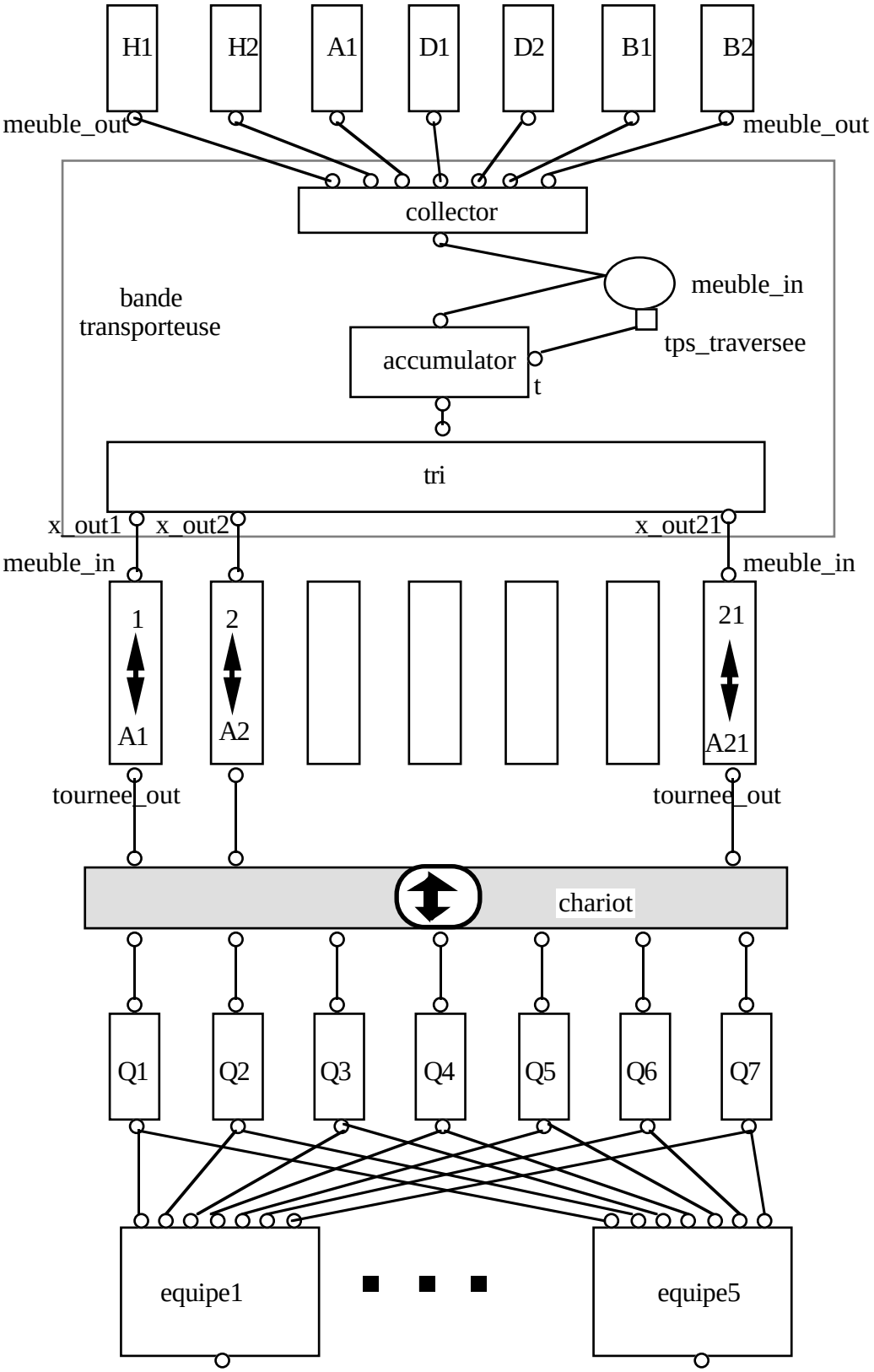


figure V.28

le modèle de la chaîne d'expédition des meubles

V.5 Performances

Afin de tester l'efficacité de BOCAL, nous avons procédé à quelques tests. Nous nous sommes à la fois intéressés à la place mémoire nécessaire au programme, et au temps d'exécution (temps CPU). Les exemples suivants sont réalisés sur une station HP 9000 série 715.

Pour cela, nous avons construit deux modèles de test basés sur les modèles "gen_cars" et "route" voir V.1.6. Un troisième test est effectué concernant le modèle de la chaîne d'expédition de meubles (figure V.28).

V.5.1 Premier test

Le premier modèle de test est constitué des routes branchées en série sans aucune signalisation routière.

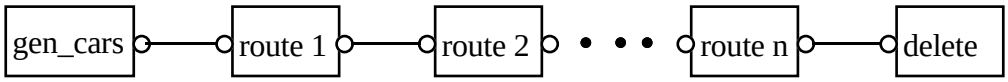


figure V.29 n routes en série

La première série de mesures concerne la simulation de ce modèle pendant un temps suffisant pour que "gen_cars" puisse lancer deux véhicules sur ces routes.

série

n	temps (s)	mem (Ko)
1	0,3	177
5	0,5	226
10	0,8	308
15	1,1	380
20	1,5	460
25	1,8	534
30	2,2	614

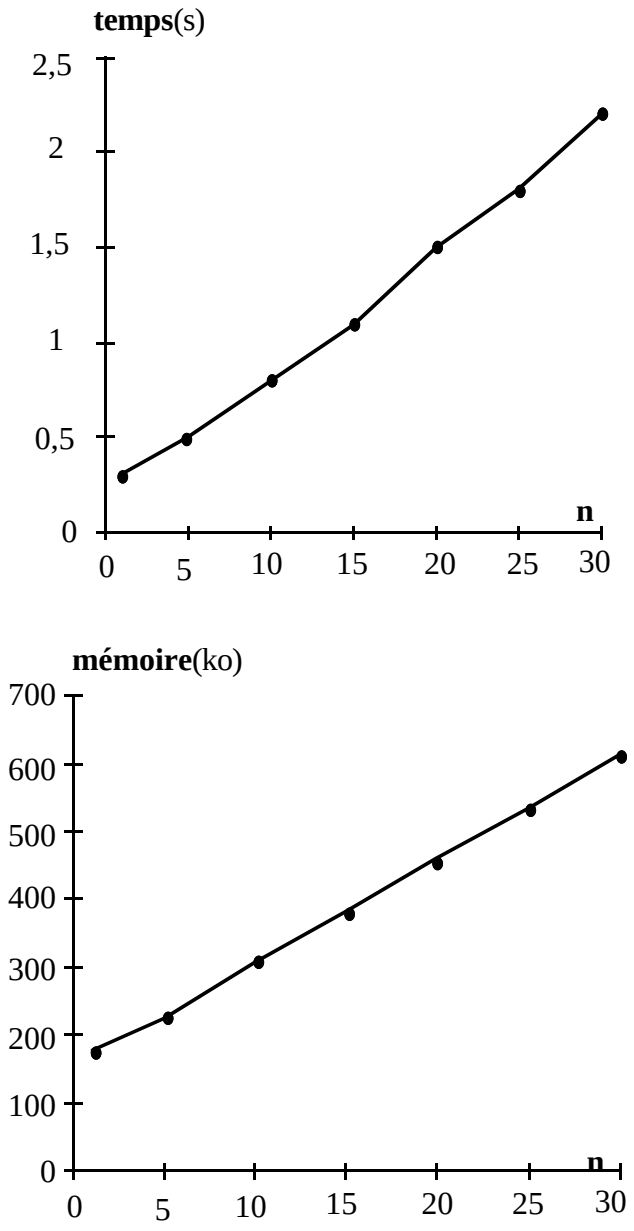


figure V.30 mesures concernant n routes en série

Les deux courbes obtenues sont pratiquement linéaires. Ces résultats nous montrent que le système BOCAL n'explose pas exponentiellement avec la taille du problème modélisé que ce soit en temps d'exécution ou en taille mémoire, ce à quoi on pouvait s'attendre mais qu'il est néanmoins rassurant de vérifier.

V.5.2 Deuxième test

Le but est de tester l'influence des niveaux d'imbrication. Le deuxième modèle de test est constitué des 10 routes branchées en série sans aucune signalisation routière. Chaque route est encapsulée par n boîtes qui ne devraient pas changer le fonctionnement du modèle.

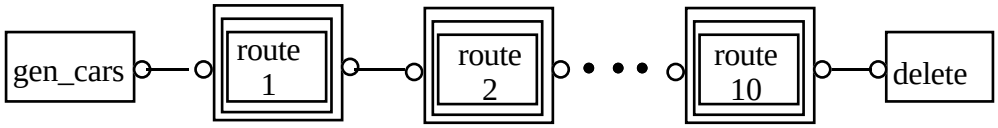


figure V.31
10 routes en série (dont chacune est en profondeur n)

Voici les résultats obtenus de la simulation du modèle (figure V.31) pour plusieurs valeurs de n différentes :

profondeur

n	temps (s)	mem (Ko)
1	0,9	319
5	0,9	343
10	1	367
20	1,2	425
30	1,5	484
40	1,7	542
50	2	600
60	2,2	658
70	2,5	715
80	2,8	776
90	3	832
100	3,3	892

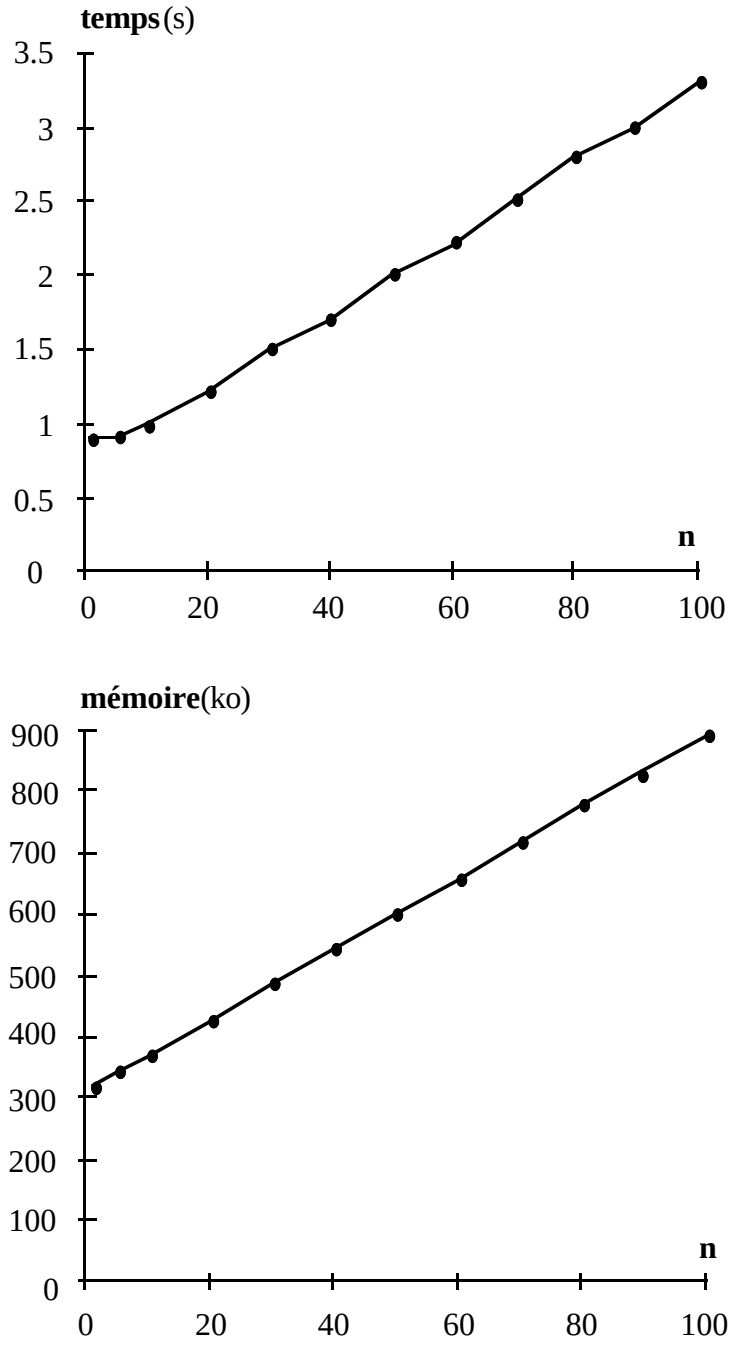


figure V.32
mesures concernant 10 routes à profondeur n

Ces deux courbes nous rassurent aussi. Parce que l'encapsulation est une caractéristique importante du langage et constitue l'atout de BOCAL.

V.5.3 Troisième test

La simulation d'un modèle réel comme la chaîne d'expédition de meubles permet aux lecteurs d'avoir une idée sur le temps d'exécution et la taille mémoire nécessaire pour son exécution. Chaque ligne du tableau ci-dessous montre les résultats de la simulation pour un nombre différent de meubles produits. Une tournée est composée de quatre meubles.

nombre de meubles	nombre de tournées	temps (s)	mem (Ko)
21	1	3	520
37	2	4,5	577
46	4	5,5	588
64	6	7,3	620
110	18	13,2	698
150	25	18,3	746

V.6 Conclusion

En résumé, l'étude de ces exemples nous conduit à classer les objets à modéliser en différents types :

1) ceux qui génèrent les éléments de départ (les générateurs).

2) ceux qui transportent les éléments (chariots, transporteurs, accumulateurs...).

3) ceux qui stockent les éléments dans des files d'attentes.

4) ceux qui trient les éléments selon un numéro ou une composante d'une manière générale.

5) ceux qui usinent les éléments en modifiant leurs caractéristiques.

6) ceux qui récupèrent les éléments en fin de simulation et traitent les informations les concernant.

Ces exemples tentent de montrer la facilité de modéliser un problème réel, de modifier le modèle et finalement de réutiliser ce modèle dans la définition d'autres modèles.

La bibliothèque construite dans ce chapitre est loin d'être exhaustive. On a montré seulement les modèles utiles dans la modélisation des exemples. L'enrichissement de cette bibliothèque est une condition nécessaire à une utilisation étendue de BOCAL dans ce domaine d'application.

Extensions et perspectives

Les spécifications du langage BOCAL se sont concrétisées en une première version d'un compilateur et d'un exécuteur. Ce "démonstrateur" nous a permis de valider et d'améliorer un certain nombre d'hypothèses et de démarrer la construction d'une bibliothèque de modèles de base pour la simulation de systèmes de production .

Mais BOCAL ne prendra tout son sens qu'une fois développé son environnement de programmation visuel, pour lequel il a été dès le départ conçu. La société ALMA poursuit actuellement la mise au point de cet environnement qui seul permettra de vérifier le principe de départ : offrir à tous les utilisateurs un modèle sémantique unique. Mais il demeure bien des domaines de recherche autour de BOCAL dont nous mentionnons quelques uns ci dessous :

***** Le polymorphisme : C'est une des caractéristiques importantes de l'approche orientée objet. Malheureusement BOCAL n'a pas encore bien spécifié ce concept. L'approche orientée objet accepte qu'une classe définisse plusieurs méthodes du même nom mais avec des arguments différents en nombre et en type. L'approche orientée objet définit le polymorphisme comme le mécanisme qui choisit la bonne méthode selon les arguments utilisés. Il faut voir que cette question du polymorphisme se pose différemment en BOCAL pour la simple raison que BOCAL a unifié la notion d'attributs et la notion de méthodes qui sont au coeur de l'approche orientée objet.

En BOCAL, il s'agit d'accepter plusieurs modèles du même nom. A l'activation d'une boîte, le système BOCAL devrait chercher à développer le modèle qui correspond "au mieux" aux modèles de ses boîtes filles exportables.

***** La compatibilité par prototype : la version actuelle implémente la notion de compatibilité entre modèles. Ce serait intéressent d'implémenter et de tester la compatibilité par prototype, puis de comparer les deux approches.

Vers les applications

Mais le plus important des travaux futurs autour de BOCAL concerne la facilité d'usage et l'ergonomie du langage. Dans cette thèse nous avons cherché à rester très "puristes" par rapport aux concepts de départ (ce purisme se retrouve souvent au début des

langages (LISP, PROLOG, SMALLTALK, etc..) avant que les nécéssités "pratiques" ne conduisent à un certain abâtardissement). Le meilleur exemple en est C++, qui a sans doute permis à l'approche objet de décoller vraiment, mais qui est loin de l'esthétique de SMALLTALK. Parmi ces améliorations indispensables on peut citer :

***** La protection de données : Chaque boîte préciserait la nature de ses boîtes exportables : les boîtes exportées en lecture et les boîtes exportées en écriture. La version actuelle permet aux boîtes voisines toutes les modifications possibles des boîtes filles exportables sans aucune limite.

***** Les boîtes "basic" : toute notre démonstration sur l'intérêt de notre travail reposait sur l'idée d'une sémantique unique pour tous les utilisateurs. En fait nous savons bien qu'il n'existe pas d'esperanto informatique : chaque langage a son domaine de modélisation de prédilection. Pour BOCAL c'est très clairement la représentation de processus physico-logiques. La meilleure manière de programmer en BOCAL c'est souvent de se dire : comment aurait-on réaliser "physiquement" le système à représenter. Mais BOCAL n'aime guère la mathématique numérique où l'algorithmique complexe, car il faut alors réaliser de véritables pseudo-processeurs dédiés.

On peut actuellement pour les calculs complexes utiliser les modèles, appelés C_exec (voir paragraphe IV.2.3), mais l'utilisation de ces modèles oblige à recompiler le système BOCAL, ce qui n'est pas bien pratique. On pense à développer un langage "pseudo basic" pour définir les instructions élémentaires qui utilisent les expressions arithmétiques.

La définition d'un tel langage définirait un nouveau type de modèles élémentaires : les modèles élémentaires basic. Par exemple si on veut définir le modèle qui réalise l'addition des nombres complexes on pourrait avoir une syntaxe de ce type :

```
#include "basic.boc";

BOX complex;
INSTANCES
      Real x,y;
ENDINST
ENDBOX

BOX add_complex;
REFERENCES
      complex z1,z2,result;
ENDREF
Precond
 z1 and z2 and ?result;
BASIC
 result.x= z1.x + z2.x;
 result.y= z1.y + z2.y;
ENDBASIC
ENDBOX
```

Les modèles "basic" devraient permettre l'accès aux "champs" (les boîtes filles instances non locales).

***** Interface graphique : la syntaxe textuelle n'étant pas le moyen le plus pratique de décrire les objets de la simulation de production. Pourtant, les carences de celle-ci ont permis de mettre en évidence, justement, la nécessité de disposer d'un moyen de visualisation, autant pour la conception des boîtes BOCAL (qui s'est d'ailleurs systématiquement faite à partir de schémas visuels comme en témoigne le nombre élevé de graphiques de ce rapport) que pour l'observation des phénomènes étudiés (le "suivi" des objets de la simulation). Ce dernier point étant capital car il est bien plus pratique de constater visuellement un blocage dans une chaîne de production que "d'éplucher" des pages de données, résultats d'une simulation. Pour BOCAL, cette visualisation aurait aussi l'avantage de rendre le debogage beaucoup plus facile.

***** Entrées / sorties d'une boîte : Le langage BOCAL ne distingue pas les arguments qui constituent les entrées de ceux qui constituent les sorties car cette distinction n'a pas de nécéssité "théorique". Cette indifférence rend la modélisation difficle à lire et contradictoire avec les diagrammes habituels du flux de produit. Il faudra aménager l'interface graphique pour qu'elle permette aux boîtes d'indiquer leurs entrées et leurs sorties.

Extensions applicatives

***** Compléter et expérimenter la bibliothèque de gestion de production.

***** Définir les objets de base pour le maquettage d'automates et construire un atelier de maquettage basé sur le noyau BOCAL.

***** Evaluer BOCAL comme base d'un langage "auteur" pour le multimédia et la didactique.

En conclusion, on peut dire que, si l'intérêt académique de BOCAL est, à notre avis, incontestable, ce qui sans doute suffisant pour ce travail, il lui reste à gagner des batailles commerciales pour prouver qu'il peut-être, au delà de ses vertus théoriques, un outil pratique.

Bibliographie

[Agh86] G. Agha.
Actors: A model of concurrent computation in distributed systems.
MIT Press, Cambridge, Massachussets, 1986.

[Ala84] P. Alanche, B. Ancelin.
Outils de simulation généraux : évaluation des simulateurs à événements discrets. Le nouvel automatisme. Mars 1984.

[All89] H. Alla, R. David.
Du grafcet aux réseaux de Petri.
Hermès, Paris, 1989.

[Bel83] G. Bel.
Méthodes et langages de simulation pour la production automatisée: principes, choix, utilisation.
Congrés Automatique AFCET, Besançon. 1983.

[Bel85] G. Bel, D. Dubois.
Modélisation et simulation de systèmes automatisés de production.
APII volume 19 n° 1, 1985.

[Bel86] G. Bel, M. Blanchard.
OASYS.
Rapport final convention ADI n° 84/714, 1986.

[Ben85] K. Benzakour.
SICLOP : simulateur de commande logique et de procédés.
Thèse 3° cycle, LAAS, 1985.

[Bra83] G. W. Brams.
Réseaux de Petri : théorie et pratique", Eyrolles, 1983.

[Cox83] B.J. Cox.
The object oriented pre-compiler, Programming Smalltalk-80 methods in C language.
ACM SIGPLAN Notices, 18(1):15-22, 1983.

[Cox86] B.J. Cox.
Object oriented programming.
Addison Wesley, Reading, Massachusetts, 1986.

[Dah65] O.J. Dahl and K. Nygaard.
Simula, a language for programming and language description of discrete event systems. Introduction and user's manuel. Norwegian computing center, Oslo, Norway, 1965.

[Dal83] Y. Dallery, H. Deneux R. David.
Recherche d'une même base de description en vue de la simulation et de la commande d'un atelier flexible: utilisation du grafcet.
Actes du congrés Automatique AFCET, Besançon. 1983.

[Dal84] Y. Dallery.
Une méthode analytique pour l'évaluation des performances d'un atelier flexible.
Thèse docteur ingénieur, INPG. 1984.

[Dol83] F. Dolle, M. Moalla, P. Rodriguez.
Concevoir des systèmes automatisés de production.
Le Nouvel utomatisme, mars 1983.

[Duc89] R. Ducournau et M. Habib.
La multiplicité de l'héritage dans les langages à objets.
Techniques et Sciences Informatiques, 8(1):41-62, 1989.

[Gol83] A. Goldberg.
Smalltalk-80, the language and its implementation.
Addison Wesley, Reading, Massachusetts, 1983.

[Gol84] A. Goldberg.
Smalltalk-80, the interactive programming environment.
Addison Wesley, Reading, Massachusetts, 1984.

[Han79] P. Brinch Hansen.
A keynote addess on concurrent programming.
Computer, 12(5):50-56, 1979.

[Hat91] D.J. Hatley, I.A. Pirbhai.
Stratégie de spécification des systèmes temps réel (SA-RT).
MASSON. 1991.

[Hen88] Henriksen, J.O. et al.
GPSS/H User's Manuel, 3rd ed.
Wolverine Software Corporation, annanadale, Virginia. 1988.

[Hew77] C.E Hewitt.
Viewing control structure as patterns of passing messages.
Artificial intelligence, 8:323-364 1977.

[Hol85] D. Hollinger.
Utilisation pratique des Réseaux de Petri dans la conception des systèmes de production.
TSI 85/06. 1985.

[Hop79] Hopcroft, John, et Jeffrey Ullman.
Introduction to automata theory, langages and computation.
Reading, Mass. : Addison-wesley, 1979.

[Jen81] K. Jensen.
Coloured Petri Nets and the Invariant Method.
Theoritical Computer Science, 14. North Holland Publ. Co., pp. 317-336.

[Kay69] A. Kay.
The reactive engine (ph. d. thesis). 1969.

[Ken87] J.C. Kenvin, J.T. Sommerfeld.
Discrete event simulation of large scale poliomyelitis vaccine production.
Process Biochemistry 22, N° 3, 74-77.

[Kra88] G.E. Krasner and S.A. Pope.
A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80.
Journal of object oriented programming, 1(3):26-49, 1988.

[Leo85] V. I. Leopoulos.
LORIC : un simulateur RdP écrit en Maclisp.
Rapport de recherche n° 371, INRIA. 1985.

[Lie86] H. Lieberman.
Delegation and inheritance: two mechanisms for sharing knowledge in object oriented systems. Actes des 3èmes JLOO, Bigre+Globule 48, pages 79-89, Paris, 1986.

[Mar87] F. Martin.
Méthodologie de modélisation et simulation de systèmes complexes décrits par Réseaux de Petri Colorés.
Thèse de Doctorat de L'INPG. 1987.

[Mas90] G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre.
Les langages à objets.
InterEditions, (deuxième tirage), 1990.

[Mey87] B. Meyer.
Reusability: The case for object oriented design.
IEEE Software,4(3):50-64, 1987.

[Mey88] B. Meyer.
Programming as contracting.
Interactive software engineering, Inc., 1988.

[Mey90] B. Meyer.
Conception et programmation par objets, pour du logiciel de qualité.
InterEditions, Paris, 1990.

[Mon85] D. Monteuil.
Simulation et aide à la gestion d'ateliers SAGA.
Thèse de l'INSA de Toulouse. 1985.

[MRT85] MRT: Ministère de la Recherche et de la Technologie.
Programme productique. Automatisation intégrée de production.
Rapport septembre. 1985.

[Nie87] O.M. Nierstrasz.
What is the "object" in object oriented programming?
In D. Tsichritzis, editor, Objects and Things, pages 1-13, Centre universitaire d'informatique, Université de Genève, 1987.

[Nie88] O.M. Nierstrasz.
A survey of object oriented concepts.
In D. Tsichritzis, editor, Active object environment, pages 1-17, Centre universitaire d'informatique, Université de Genève, 1988.

[Nyg86] K. Nygaard.
Basic concepts in object oriented programming.
ACM SIGPLAN Noices, 21(10):128-132, 1986.

[Par85] M. Parent, F. Prunet.
"La modélisation, outil de conception en productique", Méthodologie générale d'automatisation pour les industries manufacturières, Convention Automatique Productique, Paris, décembre 1985.

[Ped82] C. D. Pedgen.
Introduction to SIMAN.
Syst. Model Corporation, state college, Penn. 1982.

[Pot84] D. Potier.
QNAP : new users'introduction to QNAP2.
Rapport technique n°40, INRIA. 1984.

[Pri79] A. A. Pritsker, C. Pedgen.
Introduction to simulation and SLAM.
John WILEY Publishers. 1979.

[Pro84] Productique : où sont passées les méthodes?
Le nouvel automatisme. Mai 1984.

[Pro86] J.M. Proth, H. Quentin de Grommard.
Systèmes flexibles de production.
Masson, Paris. 1986.

[Sib85] C. Sibertin-Blanc.
High level Petri nets with data structures, 6 th European Workshop on Applications and Theory of Petri nets, Helsinki, Finland, june 1985.

[Sny86] A. Snyder.
Encapsulation and inheritance in object oriented programming languages. In proceedings of the 1st OOPSLA, pages 38-45, Portland, Oregon, 1986.

[Str67] C. Strachey.
Fundamental concepts in programming languages.
In lecture notes for international summer school in computer programming, Copenhagen, Denmark, 1967.

[Str86] B. Stroustrup.
An overview of C++.
ACM SIGPLAN Notices, 21(10):7-18, 1986.

[SYS] SYSECA.
PAWS : performances analyst's workbench system.
Groupe Syseca.

[Val] R. Valette, M. Paludetto, B. Porcher Labreuille, P. Farail.
Approche orientée objet HOOD et réseaux de Petri pour la conception de logiciel temps réel.

[Val85] R. Valette, V. Thomas, S. Bachmann.
SEDRIC : un simulateur à événements discrets basé sur les Réseaux de Petri.
APII vol 15 n° 5. 1985.

[Vid86] G. Vidal Naquet.
Les Réseaux de Petri et leurs applications.
ACET interfaces n° 43. Mai 1986.

[Woi87] P. Woiret.
Modélisation et simulation pour l'aide à la conception de systèmes de convoyage.
Thèse INSA Lyon I. 1987.

RESUME

Le sujet présenté dans cette thèse consiste à concevoir un langage de programmation, appelé BOCAL, spécialement destiné à la modélisation et à la simulation de systèmes de production. La spécification et la simulation d'automates est également un domaine d'application de BOCAL. Les logiciels sont souvent partagés entre deux caractéristiques importantes : les logiciels extensibles (ouverts) mais pas assez conviviaux et les logiciels conviviaux mais fermés pour les utilisateurs. L'objectif de ce langage est de permettre à tous les niveaux d'utilisation (conception des objets de base, construction du modèle et réglage des paramètres) de fonctionner dans le même environnement sémantique afin de permettre à tous les utilisateurs de bénéficier de la modularité et de l'extensibilité.

MOTS CLES

Simulation, Modélisation, Automates, Orienté-objet, systèmes de production.