# Co-développement de spécifications algébriques

Benjamin Blanc et Françoise Tort
LSV, CNRS & ENS de Cachan,
61, avenue du préseident Wilson 94235 CACHAN
{blanc,tort}@lsv.ens-cachan.fr

**Résumé**

Nous proposons une méthode d'écriture des spécifications algébriques. Cette méthode consiste à déterminer une structure intuitive du problème, exprimée graphiquement, puis à utiliser cette structure pour écrire la spécification, de telle sorte que l'on puisse ajouter facilement des informations architecturales. Nous décrivons cette méthode en illustrant son application sur le cahier des charges du contrôle d'accès [10].

## 1 Problématique

Pour appréhender un système informatique complexe, il est souvent utile de connaître son architecture. Celle-ci, en ne prenant en compte que les aspects d'organisation du système, permet d'en avoir une vision globale, de fournir un squelette pour l'intégration des différentes parties, et de pouvoir mieux évaluer des propriétés non fonctionnelles comme la modifiabilité, la robustesse ou la performance.

Il existe plusieurs types de descriptions permettant d'observer l'organisation du système : celui de la structure modulaire, des différents processus impliqués dans l'exécution du système, ou encore celui de la réalité physique de l'implémentation. Lorsque ces différents niveaux de description sont utilisés conjointement, leur cohérence doit être assurée [7].

La prise en compte de la description de l'architecture du système par les méthodes formelles de développement n'est pas un problème trivial. Les idées apportées par les concepteurs de langages de description d'architectures ne semblent pas résoudre ce problème [17, 13, 1]. En effet, leur objectif principal est de fournir une sémantique formelle à la description de l'architecture du système, sans expliquer comment cette description s'insère dans le modèle de développement classique des méthodes formelles, ni fournir de méthodologie permettant de construire cette description.

Le modèle de développement proposé par les méthodes formelles procède en quatre grandes étapes : analyse du problème, écriture d'une spécification, conception, puis implémentation. La phase d'analyse permet de préciser l'énoncé des besoins du cahier des charges. Une fois le problème suffisamment compris, une spécification est écrite dans un langage de spécification basé sur une sémantique formelle, ce qui permet de démontrer que certaines propriétés fondamentales sont respectées. Des méthodes existent pour aider à la construction de la spécification à partir du cahier des charges [18]. Il faut bien noter que cette activité ne se fait pas en une seule passe, mais est un processus *incrémental* et *itératif*. *Incrémental* au sens où la traduction du cahier des charges vers la spécification formelle est effectuée naturellement, par l'ajout pas à pas de chaque nouvelle information. *Itératif* au sens où un même aspect du système sera spécifié en plusieurs fois, avec l'amélioration de la compréhension de cet aspect durant l'analyse des besoins. Certaines méthodes, telles que les spécifications algébriques, permettent que cette spécification fasse abstraction des informations algorithmiques. Un mécanisme de raffinement permet d'ajouter ces détails ultérieurement, lors de la phase de conception (voir [15] pour plus de détails sur ces approches).

Le processus d'écriture de la spécification aboutit à une spécification structurée. Cette structure se compose de morceaux de spécifications nommés, reliés entre eux par des opérateurs communs à la plupart des langages de spécification (pas uniquement algébriques). Cette structuration reflète l'aspect incrémental du processus de traduction, ainsi que des choix de lisibilité, de présentation, et de réutilisation (du texte de la spécification).

La notion d'architecture que l'on peut décrire avec un langage de spécification algébrique est celle d'architecture d'implémentation : une abstraction de la future structure modulaire du système. Cette notion intervient pendant l'étape de conception, une fois le système bien compris, en ajoutant des informations structurelles à la spécification en plus des informations algorithmiques. Ces informations consistent principalement à proposer un découpage de la spécification en unités qui pourront ensuite être raffinées indépendamment les unes des autres, puis implantées et compilées séparément, pour être enfin recomposées dans le système. Techniquement, cela signifie qu'une unité doit contenir toutes les informations concernant un aspect du système (une *feature*), en particulier les opérations doivent être déclarées et définies dans la même unité.

Or la structure de la spécification issue de l'étape d'analyse ne coïncide pas nécessairement avec ce découpage en unités indépendantes. En effet, la structure de la spécification n'étant pas motivée par la distinction des différents aspects du système, ceux-ci sont le plus souvent dispersés parmi les différents morceaux de la spécification. Dès lors, prescrire l'architecture d'implémentation peut aller de la simple transformation syntaxique, regroupant les définitions et les déclarations, à la réécriture totale de la spécification, si il est impossible de constituer des unités indépendantes. Le dernier cas de figure est rencontré, par exemple, si certaines opérations sont définies mutuellement.

Devant ce constat, notre objectif est de proposer une méthode d'écriture de la spécification permettant de faciliter la prescription ultérieure de l'architecture d'implémentation. Nous avons défini une structuration de la spécification tenant compte d'une vision de l'architecture, dès la phase d'analyse, sans toutefois introduire des détails d'implémentation. Nous avons développé une technique, dite de co-développement, autorisant un processus incrémental et itératif d'écriture de spécifications. Cette technique est présentée dans la section 2. Dans la section 3, nous montrerons comment s'applique cette méthode en donnant une spécification abstraite d'un exemple simple : le contrôle d'accès [10].

## 2 Une méthode en trois temps

La méthode que nous présentons dans cette section a été mise en oeuvre pour un langage de spécification algébrique. Nous sommes cependant convaincus qu'elle pourrait être appliquée pour d'autres langages s'intéressant à la vision fonctionnelle d'un système et possédant des opérateurs de structuration similaires. Cette méthode est basée sur une technique d'écriture de spécifications, appelée co-développement, nécessitant d'introduire une structure préliminaire dès la phase d'analyse, donnant une aide méthodologique pour la suite du développement.

La méthode comporte trois étapes :

1. Nous déterminons, à l'aide de l'expérience du domaine et du cahier des charges, une structure préliminaire exprimée graphiquement sous forme d'un schéma de composants.
2. Nous écrivons la spécification de chaque composant en parallèle (on «co-développe» ces composants), en apportant des modifications à cette structure préliminaire, jusqu'à obtenir une spé-

cification complète du système et un schéma de composants final.

3. Nous prescrivons ensuite l'architecture d'implémentation, en utilisant la spécification et ce schéma de composants final.

Nous allons détailler ces étapes, et expliquer en particulier à quoi correspond cette structure préliminaire. Nous proposerons un guide méthodologique pour la construire. Nous regarderons ensuite comment le processus d'écriture de la spécification peut s'appuyer sur cette structure et comment celui-ci peut faire évoluer cette structure. Nous utiliserons le langage CASL pour écrire les spécifications (voir Annexe A). Nous utiliserons enfin une description de l'architecture basée sur les *spécifications architecturales* de CASL [3].

## 2.1 Première étape

Le type de description de la structure auquel on s'intéresse dans cette première étape est proche de celui de structure logique ou conceptuelle. Ce que l'on exprime ressemble à la structure du problème lui-même, tel qu'il se présente dans le cahier des charges. La représentation de cette structure est un schéma de composants (un schéma «box and line»), dans lequel un composant représente un aspect du système, c'est à dire une opération ou un type de données essentiels, et un lien entre composants est une relation de définition ou d'utilisation entre deux aspects : un composant $C_2$ est relié à un composant $C_1$ si $C_2$ utilise des informations contenues dans $C_2$.

Il est important de noter que cette structure préliminaire ne représente pas la structure modulaire du programme, qui n'interviendra qu'au moment du choix de l'architecture d'implémentation, ni un découpage en processus. Il s'agit juste de dégager des éléments caractéristiques du système le plus tôt possible.

Nous proposons maintenant un guide permettant de construire ce schéma de composants :

a) On essaye d'abord d'identifier le(s) domaine(s) de l'application. Cela se fait, ainsi que pour choisir un «problem frame» [8], en détectant la présence de certains termes dans le cahier des charges et en déterminant le rôle du logiciel.

b) On choisit un modèle conceptuel générique pour chaque domaine précédemment identifié. On appelle modèle conceptuel un modèle dans lequel sont décrits les principaux événements qui ont lieu dans le système, avec une organisation de ces événements selon leur rapport au système (classification de [9]). Ces modèles peuvent se trouver dans la littérature du domaine. Leur généricité provient du fait qu'ils ont été conçus pour synthétiser l'expérience du domaine. Ils sont à rapprocher des styles architecturaux de [16] ou des «design patterns» de [6].

c) On instancie ces modèles dans le contexte étudié, en identifiant les événements de chaque catégorie avec des informations du cahier des charges.

d) À partir de ces modèles, on construit un découpage du problème en composants. Pour cela, on part de trois classes d'événements qui sont soit observables soit contrôlés par le système informatique, pour identifier des aspects du problème. Dans certains cas, de par le domaine et le modèle conceptuel choisi, la détermination de certains composants supplémentaires peut être systématique.

On obtient à la fin de cette étape un schéma de composants du problème donnant les principaux aspects du système ainsi que leurs relations possibles. Les relations non explicitées à ce niveau d'abstraction seront précisées pendant la deuxième étape.

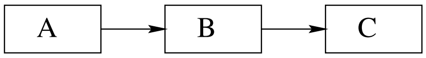
## 2.2 Deuxième étape

L'objectif de cette étape est d'écrire la spécification du système en faisant attention à ce que toutes les informations concernant un même aspect du système soient regroupées afin de simplifier l'écriture d'une spécification architecturale, troisième étape de notre méthode. Nous voulons utiliser le schéma de composants issu de la première étape comme une ébauche de structure de la spécification. La spécification de chaque aspect sera regroupée dans un morceau de spécification portant le nom de cet aspect. Un lien entre deux aspects sera traduit par une extension entre les spécifications correspondantes, vérifiant que les modèles du second aspect ne modifient pas ceux du premier (extension *conservative* cf. [12]).

Puisque la compréhension du problème s'améliorera durant cette étape, des modifications seront certainement apportées à la structure de la spécification. Ces modifications peuvent être l'ajout d'un nouveau composant, le découpage en sous-composants, l'ajout/retrait d'un lien. Le schéma de composants suivra alors dynamiquement ces modifications. En revanche, il n'est pas nécessaire de reporter des modifications mineures, comme l'introduction de la spécification des listes ou celle des ensembles.

Le schéma de composants sert donc de guide pour trier les informations provenant du cahier des charges au cours du processus d'écriture de la spécification. Nous distinguons, pour le moment, trois types d'informations :
- celles qui incrémentent la spécification d'un composant ;
- celles qui incrémentent la spécification de plusieurs composants ;
- celles relatives à un nouveau composant.

Nous illustrons sur un exemple trivial les modifications des spécifications et de la structure pour chacun des trois cas. Étant donnés trois composants $A$, $B$, et $C$ reliés dans un schéma de composants comme dans la figure ci-dessous, dans laquelle les boites sont les composants et les flèches des liens d'utilisation (lire $C$ utilise $B$ et $B$ utilise $A$) :




Nous essayons de trier chaque nouvel élément d'information dans la spécification d'un des composants selon les trois cas. Ces spécifications sont nommées $SP_A, SP_B$ et $SP_C$.

Supposons qu'une première information concerne un seul composant, disons $A$. Cela peut être la définition d'une opération déclarée dans $SP_A$, ou bien la déclaration d'une sorte ou d'une opération concernant le même aspect du système (selon le choix du spécifieur). Ce premier cas met en valeur la nature itérative du processus d'écriture. Comme nous voulons conserver le fait que la spécification d'un aspect soit regroupée sous le même nom de spécification, nous redéfinissons la spécification $SP_A$ en y incorporant ce qui était déjà présent auparavant. Cela s'écrit :

$$SP_A \stackrel{co}{=} SP_A \ \mathbf{then} \ \Delta_A$$

La nouvelle spécification conserve le même nom ($SP_A$). Le $\Delta_A$ correspond à l'information ajoutée. La relation $\stackrel{co}{=}$ permet de différencier la définition d'un nouveau nom de spécification, notée $=$, d'un co-développement.

Au niveau sémantique, la relation $\stackrel{co}{=}$ se traduit par l'utilisation d'une règle différente de celle utilisée pour la définition usuelle. D'autre part, il faut tenir compte de ce que toutes les spécifications du système faisant intervenir $SP_A$ sont modifiées par cet enrichissement. Nous pensons qu'un mécanisme de liaison dynamique pourrait gérer efficacement ces mises à jour. La notion de liaison dynamique pour un langage de programmation signifie que la valeur associée à un nom de variable, stockée dans un environnement, sera celle rencontrée en dernier dans le texte du programme. Par exemple, étant donné le programme suivant :

$$x := 0\ ;\ y := x + 1\ ;\ x := 1$$

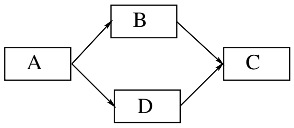
Un appel à la valeur de $y$ donnera 2 dans un langage avec liaison dynamique, 1 sinon. L'analogie consiste alors à considérer les noms de spécification comme des variables, et leur sémantique comme la valeur associée à ces variables.

Le deuxième cas apparaît lorsqu'une information concernant le composant $C$, par exemple, nécessite un complément concernant un autre aspect, disons $A$. Le point important est d'être sûr que l'enrichissement de $SP_C$ prend en compte celui de $SP_A$. Comme précédemment, cela est géré par la liaison dynamique des noms de spécifications dans l'environnement global. En effet, la spécification associée à un nom de spécification sera toujours celle rencontrée en dernier dans le texte. Nous obtenons :

$$\begin{array}{lcl} SP_A & \stackrel{co}{=} & SP_A \ \mathbf{then}\ \Delta_A 2 \\ SP_C & \stackrel{co}{=} & SP_C \ \mathbf{then}\ \Delta_C \end{array}$$

On est alors sûr que $SP_B$, et donc $SP_C$, utilise bien la dernière version de $SP_A$, puisque celle-ci aura été mise à jour après la première ligne.

Le dernier cas, lorsqu'un nouvel aspect du système est identifié, implique une modification du schéma en y ajoutant un nouveau composant, ce qui permet de suivre graphiquement l'évolution de la spécification. Si $SP_C$ nécessite une nouvelle opération qui ne concerne pas son composant, ni $A$, ou $B$, on peut créer un composant $D$. Le composant $C$ l'utilisant (dans cet exemple), $D$ est donc relié à $C$. Supposons aussi que $D$ utilise des informations de $A$ mais pas de $B$, on obtient le schéma :




Ceci est une modification naturelle, puisque le processus de spécification est incrémental, on découvrira des aspects au fur et à mesure de notre compréhension du problème. La structure de la spécification est modifiée comme suit :

$$\begin{array}{lcl} SP_D & = & SP_A \ \mathbf{then}\ \Delta_D \\ SP_C & \stackrel{co}{=} & SP_C \ \mathbf{and}\ SP_D \ \mathbf{then}\ \Delta_C 3 \end{array}$$

Une fois la spécification des besoins écrite, il est nécessaire d'y apporter des raffinements afin de se rapprocher d'un spécification de l'implémentation. Une étape importante dans ces raffinements sera la prescription de l'architecture d'implémentation du système, qui représente la future structure modulaire du programme.

## 2.3 Troisième étape

La troisième phase de notre méthode consiste à écrire une spécification architecturale du système. Cette spécification architecturale identifie les unités qui pourront être ensuite développées indépendamment. Nous renvoyons le lecteur à l'annexe A pour une brève présentation des spécifications architecturales de CASL.

La deuxième étape a produit une spécification ainsi qu'un schéma de composants. Ce schéma de composants a une nature différente de celui de la première étape : il reproduit, par construction, la partie conservative de la structure de la spécification. Pour déterminer les morceaux de spécifications correspondant aux unités, il suffit de découper la spécification en suivant ce schéma. Selon la granularité désirée, on pourra associer chaque spécification d'un composant à une unité ou bien en regrouper plusieurs.

Par exemple, en continuant l'exemple précédent, décidons d'associer une unité à chacun des composants. On obtient la spécification architecturale suivante :

**arch spec** =
**units**
$U_A$ : $SP_A$
$U_B$ : $SP_B$ **given** $U_A$
$U_D$ : $SP_D$ **given** $U_A$
$U_C$ : $SP_C$ **given** $U_B$ **and** $U_D$
**result**
**end**

La notation $U_A : SP_A$ signifie que $U_A$ est une algèbre satisfaisant la spécification $SP_A$. $U_B$ est une algèbre qui, étant donné l'algèbre $U_A$, satisfait la spécification $SP_B$. La sémantique des spécifications architecturales assure que, quelle que soit l'algèbre $U_A$, $U_B$ existe. Cela permet donc de raffiner la spécification $SP_A$ indépendamment de la tâche de fournir une implémentation pour $SP_B$.

# 3 Application : le contrôle d'accès

Nous illustrons dans cette section notre démarche pour développer l'étude de cas du contrôle d'accès proposée dans [10].

## 3.1 Première étape

Afin de déterminer un schéma de composants pour le système de contrôle d'accès, nous suivons le guide décrit dans la section 2.1 :

a) *Identification des domaines d'application du système.* On reprend pour cela la description faite dans [19] : « L'objectif est de gérer l'accès de personnes à un ensemble de bâtiments et de mémoriser ces accès en vue de répondre à des requêtes sur la présence de personnes dans les bâtiments».

   Le système doit *gérer l'accès* de personnes et peut ouvrir les portes à distance ([10], section 3.1. §3) : il s'agit donc d'un système de contrôle-commande. On doit pouvoir également effectuer des *requêtes* sur la présence de personnes : c'est donc aussi un système d'information dynamique, comportant une base de données régulièrement enrichie.

b) *Choix des modèles génériques.* Un modèle générique pour un système de contrôle-commande est celui de [11], qui décrit la boucle de contrôle (voir figure 1).

   La hiérarchie d'événements proposée dans [9] se retrouve comme suit : les événements contrôlés par l'environnement et observables par le système sont les *variables contrôlées* du modèle, les événements contrôlés par le système et observables par l'environnement sont les *variables manipulées* du modèle. Les événements contrôlés par l'environnement et non partagés par le système (comme l'entrée de personnes dans un bâtiment), ne sont pas utiles pour l'écriture de

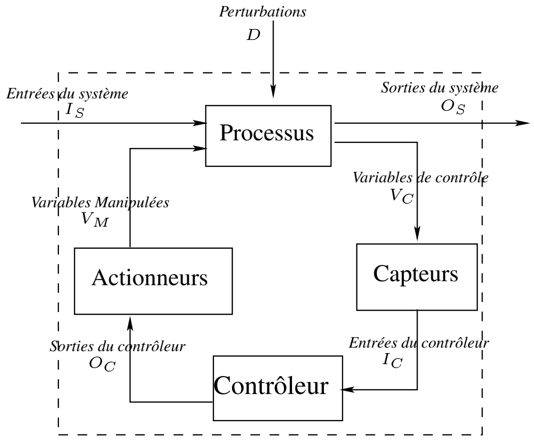



FIG. 1 – Modèle de contrôle-commande

la spécification du programme. Quant aux événements contrôlés par le système et non partagés par l'environnement, ce sont les actions internes du programme qui n'apparaissent pas ici.

Pour les systèmes d'informations dynamiques, nous pouvons proposer le modèle générique de la figure 2.

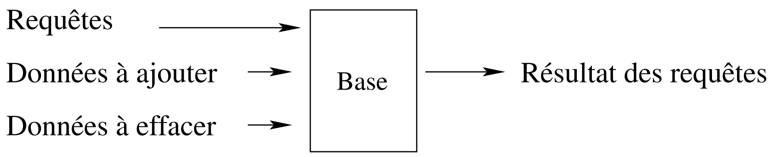



FIG. 2 – Modèle d'information dynamique

c) *Instantiation des modèles génériques.* Cela se fait en identifiant la terminologie introduite dans ces modèles. Certains termes de ces modèles ne trouvent pas de contrepartie dans le problème étudié, ce que nous signalons par : *inconnues*.

D'abord pour celui de contrôle-commande :
- Entrées du système : *inconnues* ;
- Sorties du système : *inconnues* ;
- Processus : l'ensemble des bâtiments ;
- Capteurs : lecteurs de cartes, état des portes, capteur d'incendie ;
- Contrôleur : le programme de contrôle-commande que l'on veut écrire ;
- Actionneurs : serrures des portes ;
- Variables de contrôle : insertion de carte, ouverture ou fermeture de porte ;
- Variables manipulées : état des serrures ;
- Entrées du contrôleur : `lecture(id_lecteur,id_carte)`, ... (cf. §4.2.1 de [10]) ;
- Sorties du contrôleur : `bloquer(id_porte)`, ... (cf. §4.2.2 de [10]) ;
- Perturbations : *inconnues* ;

Puis pour celui de système d'information dynamique :
- Ajout de données : lorsqu'une personne entre ou sort dans un bâtiment ;
- Retrait de donnés : *pas d'effacement* ;

– Requêtes : est-ce qu'une personne est présente dans un bâtiment ? Liste des personnes présentes dans un bâtiment ;
– Résultat des requêtes : une liste ou un nom de personnes ;

d) *Découpage en composants.* On se place dorénavant du point de vue du programme. Les événements pris en compte ne sont plus que ceux que le programme peut observer et ceux sur lequel il a une action.

Les premiers composants que l'on peut distinguer sont ceux provenant du modèle de contrôle-commande : le premier est le protocole d'ouverture qui constitue le coeur du contrôleur (composant $Envoi$) ; le second est une description de l'image de l'environnement dans lequel le contrôleur est plongé (sous forme de types de données) ; un autre décrit la communication de ce contrôleur avec l'extérieur en terme d'entrées/sorties. Nous ne nous occupons pas de l'architecture physique du système de communication sous forme de bus. Le composant $Entres/Sorties$ est relié au composant $Environnement$ puisque les messages ont comme paramètres des types de l'environnement. $Envoi$ est relié à $Entres/Sorties$ puisque l'objectif de ce composant est de déterminer les messages de sorties à partir de ceux d'entrée.

Le second modèle utilisé nous donne un seul composant : une description abstraite de la base de données (le journal dans [10]) sous la forme d'un type, ainsi que les requêtes que l'on voudra effectuer comme des opérations sur ce type. Le lien entre ce composant et les précédents n'est pas clair à cette étape, on le laisse donc non-spécifié.

L'instanciation est terminée. Nous regardons si des termes identifiés dans l'étape $a)$ n'ont pas été couverts par l'instantiation. C'est le cas de la notion de groupe intervient pour déterminer l'accès aux portes ; de plus il devra être possible de modifier la constitution de ces groupes : nous créons un composant qui contiendra ces informations. La figure 3 présente le schéma de composants obtenu à la fin de cette première étape :

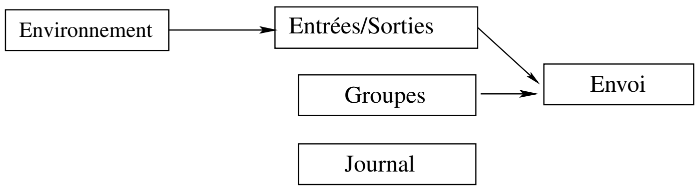



FIG. 3 – Schéma de composants

## 3.2 Deuxième étape

La spécification que nous allons proposer dans cette section ne décrit pas tout le système dans ses détails. Il s'agit d'une spécification abstraite qui pourra être raffinée vers une spécification concrète dans une étape suivante. Nous nous sommes attachés principalement à montrer les mécanismes de la technique de co-développement, en mentionnant à chaque fois les informations du cahier des charges dont nous partons. .

1) *L'environnement est composé de bâtiments ayant des portes et des capteurs d'incendie*. Puisqu'il y a un capteur par porte et qu'une porte ne sert qu'à entrer ou à sortir, il y a bijection entre les lecteurs de cartes et les portes, de même pour les serrures. Nous choisissons la perspective des portes, auxquelles on associe un bâtiment. Les sortes nécessaires pour décrire l'environnement sont déclarés dans la spécification suivante :

**spec** ENVIRONNEMENT=
  **sorts** $Porte, Capteur, Batiment$
  **op** $acces : Porte \to Batiment$

2) *Les entrées/sorties du programme sont les messages décrits dans la section 4.2.* Nous les décrivons sous forme de deux types de données énumérés : un pour les messages que le programme reçoit, un autre pour ceux qu'il envoie, sans prendre en compte les accusés de réception dans un premier temps :

**spec** IO= ENVIRONNEMENT **then**
  **type** $R\_Messages ::= lecture(Porte, Carte) \mid ouverture(Porte) \mid fermeture(Porte) \mid incendie(Capteur);$
  **type** $S\_Messages ::= bloquer(Porte) \mid debloquer(Porte) \mid allumer\_rouge(Porte) \mid allumer\_vert(Porte) \mid eteindre\_rouge(Porte) \mid eteindre\_vert(Porte);$

3) *L'accès est accordé sur la base de l'affiliation d'une personne à un groupe.* On déclare une sorte pour les groupes, une pour les cartes qui sont en bijection avec les personnes. On ajoute de plus une opération pour associer une carte à un groupe et un prédicat pour définir les autorisations d'un groupe pour un bâtiment. La spécification enrichit la spécification de l'environnement de la manière suivante :

**spec** GROUPES= ENVIRONNEMENT **then**
  **sorts** $Groupe, Carte$
  **ops** $appartenir : Carte \to Groupe$
  **pred** $autorise : Groupe \times Batiment$

4) *Une personne appartient à un groupe et un seul.* On rajoute donc un axiome à la spécification précédente, que l'on co-développe :

**spec** GROUPES $\overset{co}{=}$ GROUPES **then**
  **vars** $c : Carte;$
  **axioms** $\exists! g : Groupe \bullet appartenir(c) = g$

5) *Tout accès est enregistré tant en entrée qu'en sortie d'un bâtiment.* Le journal du système, qui constitue la base de données dynamique, est décrite sous la forme d'un type construit par l'ajout de données. On ajoute une donnée dès qu'on pense qu'une personne est entrée ou sortie d'un bâtiment (après l'ouverture d'une porte). L'information est constituée du numéro de la carte insérée dans le lecteur, de la porte franchie et de la date.

   Nous avons ici un exemple d'information nécessitant l'introduction d'un nouveau composant. En effet, on a besoin aussi d'une spécification d'un timer donnant la date des mouvements. Cette spécification fournit une sorte pour la date ainsi qu'une opération permettant de dater un message.

   Les requêtes sont spécifiées sous forme d'opérations sur ce type : une pour déterminer si une personne est présente au moment où la requête est lancée (partielle car on ne sait pas exprimer l'absence de bâtiments), une autre pour donner la liste des présents dans un bâtiment à une certaine date. Nous introduisons alors une spécification classique des listes, en tant que spécification auxiliaire. La spécification du journal enrichit donc celle des groupes, du timer et des listes de cartes :

**spec** TIMER = IO **then**

**sort** $Date$

**op** $dater : R\_Message \to Date;$

**spec** JOURNAL= TIMER **and** LIST[GROUPES **fit** $Elem \mapsto Carte$] **then**

**generated type** $Journal ::= empty$

$| \ enregistrer(\_\_, \_\_, \_\_, \_\_)(Carte; Batiment; Date; Journal)$

**ops** $etre\_dans : Carte \times Journal \to? Batiment;$

$liste\_presents : Batiment \times Date \times Journal \to List[Carte];$

6) Nous traduisons maintenant le protocole d'accès. Nous reprenons pour cela la description détaillée faite dans [2], section 5.1. Le coeur de la spécification est la fonction $envoi$ qui calcule les messages à envoyer, sous forme d'ensembles. Les messages à envoyer sont déclenchés de deux façons : soit après réception d'un message, soit lorsqu'une section critique se referme après que le délai accordé soit expiré. Une section critique est ouverte si une une porte est débloquée (section critique d'ouverture) ou vient d'être ouverte (section critique fermeture).

Nous introduisons deux nouveaux composants : le premier pour l'état du système, le second pour la boucle principale qui gère les tics d'horloge. L'état du système est spécifié de manière abstraite comme une sorte avec des observateurs correspondant aux informations dont la fonction d'envoi a besoin. Le co-développement est ici crucial, puisqu'il permet l'ajout incrémental de ces observateurs au fur et à mesure des besoins. Ces observateurs ne sont pas connus pour l'instant :

**spec** ETAT = JOURNAL **then**

**sort** $Etat$

**spec** ENVOI = ETAT **then**

**ops** $envoi : R\_Message \times Etat \to Set[S\_Messages]$

**spec** SYSTEME = ENVOI **then**

**op** $Loop : Etat \times r\_Message \to Etat$

Le calcul des messages à envoyer dépend de préconditions qui vont déboucher sur plusieurs étapes de co-développement.

7) *La validité d'une carte peut être testée. La liste des cartes perdues ou volées sont enregistrées.* On rajoute donc un prédicat testant la validité et un autre pour les cartes hors service :

**spec** GROUPES $\stackrel{co}{=}$ GROUPES **then**

**pred** $valide : Carte;$

$hors\_service : Carte$

8) *On ne peut sortir d'un bâtiment que si on y est préalablement entré.* Il faut donc différencier les portes entrantes et sortantes, ce que l'on fait à l'aide de prédicats s'excluant mutuellement :

**spec** ENVIRONNEMENT $\stackrel{co}{=}$ ENVIRONNEMENT **then**

**sorts** $is\_in, is\_out : Porte \times Batiment$

**vars** $p : Porte; b : Batiment$

**axioms** $is\_in(p, b) \Leftrightarrow \neg is\_out(p, b)$

9) Nous pouvons alors définir les préconditions statiques d'entrée et de sortie :

**spec** ENVOI$\overset{co}{=}$ ENVOI **then**

**pred** $autorisation\_in(c : Carte; p : Porte; j : Journal) \Leftrightarrow valide(c) \wedge is\_in(p, acces(p)) \wedge autorise(appartenir(c), acces(p)) \wedge \neg \exists b \bullet etre\_dans(c, j) = b;$

**pred** $autorisation\_out(c : Carte; p : Porte; j : Journal) \Leftrightarrow is\_out(p, acces(p)) \wedge etre\_dans(c, j) = acces(p);$

10) Pour calculer les ensembles de messages à envoyer, nous allons avoir besoin du journal courant et des sections critiques ouvertes. Nous rajoutons ces deux observateurs sur l'état du système, notés $Obs\_$ X. Des fonctions serviront à mettre à jour ces observateurs dans la boucle principale, notées $New_$X. Nous utilisons une vision très abstraite du temps entre deux dates. L'opération $\_ - unit$ soustrait une unité de temps à un délai. Nous donnons les premiers axiomes pour le calcul des messages[1] :

**spec** ENVIRONNEMENT$\overset{co}{=}$ ENVIRONNEMENT **then**

**free type** $Status ::= Open \mid Closed$

**spec** IO$\overset{co}{=}$ IO **and** SET[IO **fit** $Elem \mapsto S\_Messages$] **then**

**ops** $\_\_ + \_\_ : S\_Messages \times S\_Messages \to Set[S\_Messages];$

**spec** TIMER$\overset{co}{=}$ TIMER **then**

**sort** $Temps$

**op** $delai\_ouverture, delai\_fermeture, 0 : Temps;$
$\_ - unit : Temps \to Temps$

**pred** $\_\_ < \_\_ : Temps \times Temps$

**spec** ETAT$\overset{co}{=}$ ETAT **and** LIST[ENVIRONNEMENT **and** TIMER **fit** $Elem \mapsto Porte \times Nat \times Status$] **then**

**op** $Obs\_journal : Etat \to Journal;$
$Obs\_sec\_cri : Etat \to List[Porte \times Temps \times Status]$

**spec** ENVOI $\overset{co}{=}$ ENVOI **then**

**op** $New\_sec\_cri : Etat \times R\_Message \to List[Porte \times Nat \times Status];$
$New\_journal : Etat \times R\_Message \to Journal$

**vars** $m : R\_Message; p : Porte; c : Carte; cap : Capteur; e : Etat; s : Status$

[1]Par manque de cas nous ne produisons pas toute la spécification. Celle-ci est disponible en s'adréssant aux auteurs

**axioms**

%% Message recu: lecture, porte entree, autorise

$m = lecture(p, c) \wedge autorisation\_in(c, p, Obs\_journal(e))$

$\Rightarrow envoi(m, e) = allumer\_vert(p) + debloquer(p) \wedge$

$New\_sec\_cri(m, e) = cons((p, delai\_ouverture, Open), Obs\_sec\_cri(e));$

...

%% Delai expire

$\exists p \bullet member((p, 0, s), Obs\_sec\_cri(e))$

$\Rightarrow \forall m \bullet envoi(m, e) = bloquer(p) + allumer\_rouge(p)$

11) On peut alors spécifier la boucle principale. Nous commençons d'abord par raffiner la spécification des états en un tuple de variables, puis nous exprimons le passage du temps en enlevant une unité à tous les délais des sections critiques :

**spec** ETAT$\stackrel{co}{=}$ ETAT **then**

**free type** $Etat ::=< \_\_, \_\_ >$
$(jou : Journal; sec\_cri : List[Porte \times Temps \times Status])$

**vars** $e : Etat$

**axiom** $Obs\_journal(e) = jou(e);$
$Obs\_sec\_cri(e) = sec\_cri(e)$

**spec** SYSTEME$\stackrel{co}{=}$ = SYSTEME **then**

**op** $tick : List[Porte \times Temps \times Status] \rightarrow List[Porte \times Temps \times Status]$

**vars** $e : Etat; m : R\_Message; l : List[Porte \times Temps \times Status]; p : Porte; t : Temps; s : Status$

**axioms** $tick(empty) = empty;$
$tick(cons((p, t, s), l)) = cons((p, t - unit, s), tick(l));$
$Loop(e, m) =< New\_journal(m, e); tick(New\_sec\_cri(m, e)) >$

Ce processus de spécification a fait évoluer le schéma de composants initial, en particulier en explicitant le lien entre les journal et le reste du système et en rajoutant un composant pour les états et un pour la boucle principale.

Le schéma final est le suivant :

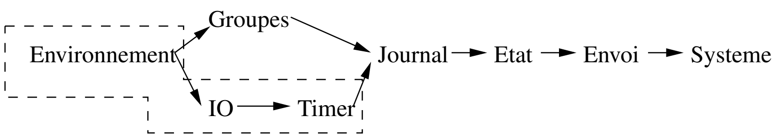



FIG. 4 – Schéma de composants final

Le cahier des charges ne donne pas le nombre de bâtiments, ni de portes ou de capteurs d'incendie. Néanmoins, nous pouvons imaginer que ces informations seront rajoutées ultérieurement. Les sortes correspondantes seront raffinées en des types énumérés. Nous pourrons effectuer ces raffinements au niveau de la spécification architecturale suivante.

### 3.3 Troisième étape

À partir du schéma de composants final et des spécifications associées, nous pouvons construire une spécification architecturale. Pour cela, on peut regrouper les morceaux de spécification concernant les types de données (ENVIRONNEMENT, I/O, TIMER) dans une même unité et associer ensuite une unité à chaque composant. On obtient, après une simplification des liens, la spécification architecturale :

**arch spec** =
**units**
$U_A$ : TYPES
$U_B$ : GROUPES **given** $U_A$
$U_C$ : JOURNAL **given** $U_B$
$U_D$ : ETAT **given** $U_C$
$U_E$ : ENVOI **given** $U_D$
$U_F$ : SYSTEME **given** $U_E$
**result** $U_F$
**end**

Nous pouvons comparer ce résultat avec la spécification de [2]. En effet, le langage de spécification utilisé par Didier Bert (LPG) permet de découper la spécification globale en fichiers indépendants qui sont ensuite assemblés. Ces fichiers peuvent être assimilés aux unités dans notre approche. On peut déduire ces fichiers de notre spécification architecturale, ce qui montre que notre travail peut être vu comme un préliminaire à celui décrit dans [2].

## 4 Travaux analogues

L'esprit de la méthode d'élicitation des besoins proposée dans [18] est proche de nos objectifs. En effet, la volonté de détailler la construction d'une spécification d'un point de vue méthodologique nous motive également. La méthode décrite par ces auteurs permet d'expliciter leur expertise dans ce domaine. Le résultat principal de cette méthode est une réécriture formelle du cahier des charges permettant de détecter des ambiguïtés, des redondances, des incohérences ou des manques. Le passage de ce cahier des charges transformé vers une spécification formelle, n'est pour le moment pas complètement détaillé. Notre méthode se rapprochant de cette deuxième partie de la méthode exposée dans [18], elle pourrait lui être complémentaire.

Néanmoins, la vision fonctionnelle que nous adoptons, dûe au fait que nous utilisons un langage de spécification algébrique, ne correspond pas à la vision événementielle du langage Z utilisée dans [18]. Notre point de vue consiste à analyser le problème en vue de son implémentation, et donc d'un découpage éventuel en unités architecturales. Cette contrainte de découpage est caractéristique de notre méthode et ne fait pas partie des préoccupations de [18].

L'idée d'introduire la connaissance du domaine dans l'écriture d'une spécification formelle est similaire à celle présente dans [14]. Les auteurs préconisent l'utilisation d'une méthode semi-formelle au préalable de l'écriture de la spécification. La spécification obtenue par leur méthode est structurée, mais pas selon le critère de «découpabilité» que nous recherchons. On retrouve le problème, décrit dans notre introduction, de la difficulté de transformer une telle spécification en une spécification architecturale. De plus, nous ne trouvons pas satisfaisant que des concepts de structuration proposés dans une première étape (l'héritage en particulier dans UML) ne soient plus disponibles dans le langage d'écriture de la spécification (méthode B). La traduction de UML vers B doit prendre en compte également la partie structurelle.

On retrouve l'idée de la décomposition du problème ainsi que sa structuration dans les «problem frames» de [8], qui permettent d'identifier des classes de problèmes. Il manque cependant dans cette approche un guide méthodologique portant sur le choix d'un de ces «problem frames» par rapport au problème étudié. Le lien entre ce découpage et la structure de la spécification est envisagé dans [4] en utilisant le langage CASL, ce qui se rapproche de nos deux premières étapes, mais pas dans l'objectif de pouvoir découper la spécification in fine.

## 5 Conclusion

Nous avons proposé une approche méthodologique permettant d'aborder le problème de la prescription de l'architecture d'implémentation. Notre méthode se base sur une étude préliminaire du problème, dans le style de [18], déterminant un découpage du problème dès la phase d'analyse sous forme d'un schéma de composants. Le critère de découpage utilisé à ce niveau est suffisamment souple pour ne pas décider de l'architecture d'implémentation trop tôt dans le développement. Il se rapproche d'un découpage conceptuel du problème, tel qu'exposé dans le cahier des charges. Nous avons proposé un guide avec des étapes et nous avons eu recours à une notation graphique pour construire ce schéma.

Le co-développement de spécifications est une technique originale guidant l'écriture des spécifications. Cette technique permet de rendre compte du processus incrémental et itératif du développement d'une spécification. Il permet également de prescrire plus simplement une architecture d'implémentation pour le système.

Les perspectives de ce travail sont diverses. D'abord nous aimerions préciser l'utilisation des résultats de l'analyse effectuée par la méthode exposée dans [18]. Cette méthode nous identifie des termes, des concepts et des événements que nous réutilisons dans notre première étape. Ensuite, nous cherchons à savoir s'il est intéressant d'utiliser un langage semi-formel pour exprimer le schéma de composants. Cela pourrait donner des éléments de description plus fins pour nos composants. Enfin, le co-développement implique l'utilisation du mécanisme de liaison dynamique, comme nous l'avons expliqué dans la section 2.2. Nous souhaitons proposer une extension de CASL dans cette direction.

## Références


[1] Mario Bass, Rick Kazman, and Paul Clements. *Software Architecture in Practice*. Addison Wesley, 1997.

[2] Didier Bert. Spécification algébrique et prototypage du Controle d'accès en LPG. In *AFADL*, pages 142–157. LSR-IMAG, 2000.

[3] Michel Bidoit, Don Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In *Proc. 7th Int. Conf. Algebraic Methodology and Software Technology (AMAST'98)*, volume 1548 of *LNCS*, pages 341–357. Springer-Verlag, January 1999.

[4] Christine Choppy and Gianna Reggio. Using CASL to specify the requirements and the design : a problem specific approach. In *WADT'99*, number 1827 in LNCS, 1999.

[5] CoFI Task Group on Language Design. CASL-the CoFI algebraic specification language-summary (version 1.0.1). http ://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/, 2000.

[6] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns - Elements of Reusable Object-Oriented Software Architecture*. Addison-Wesley, 1995.

[7] P. Fradet, D. LeMetayer, and M. Perin. Consistency checking for multiple view software architectures. In *Proceedings of the joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, LNCS, Toulouse, France, September 1999. Springer-Verlag.

[8] Michael Jackson. *Software requirements and specification : A lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.

[9] Michael Jackson and Pamela Zave. Deriving specifications from requirements : an example. In *17th ICSE*, pages 15–24. ACM Press, 1995.

[10] Yves Ledru. Étude de cas : système de contrôle d'accès. http ://www.lsr-imag.fr/Les.Personnes/Yves.Ledru/Controle_acces/.

[11] Nancy Leveson, Mats Heimdahl, and Jon Dammon Reese. Designing specification language for process control systems : lessons learned and steps to the future. In *Proceedings of the Seventh ACM/SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1708 of *LNCS*. Springer-Verlag, 1999.

[12] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of abstract data types*. Wiley-Teubner, 1996.

[13] D. Luckham, J.J Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of software architecture using Rapide. In *IEEE Transactions on Software Engineering*, pages 336–355, April 1995.

[14] Raphael Marcano, Eric Meyer, Nicole Levy, and Jeanine Souquières. Utilisation de patterns dans la construction de spécifications en UML et B. In *AFADL*, pages 66–80. LSR-IMAG, 2000.

[15] Donald Sannella and Andrzej Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9 :229–269, 1997.

[16] M. Shaw and P. Clements. A field guide to boxology : Preliminary classification of architectural styles for software systems. In *COMPSAC97, 1st Int'l Computer Software and Applications*, August 1997.

[17] Mary Shaw and David Garlan. *Software Architecture. Perspectives on an emerging discipline*. Prentice-Hall, 1996.

[18] Jeanine Souquières and Maritta Heisel. A method for requirements elicitation and formal specification. In J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, editors, *Proceedings 18th International Conference on Conceptual Modeling, ER'99*, volume 1728 of *LNCS*, pages 309–324. Springer-Verlag, 1999.

[19] Jeanine Souquières and Maritta Heisel. Une méthode pour l'élicitation des besoins : application au système de contrôle d'accès. In *AFADL*, pages 36–50. LSR-IMAG, 2000.

# A Présentation de CASL

CASL (pour *Common Algebraic Specification Language*) est un langage de spécification qui regroupe et organise les concepts utilisés dans la communauté algébrique [5]. Le paradigme suivi est d'abstraire un programme sous forme de types de données et d'opérations travaillant sur ces types de données. Le point fondamental est l'indépendance de la représentation : la représentation des types de données est inconnue et leur implémentation retardée au maximum.

Une spécification décrit une signature, où sont déclarés les éléments syntaxiques, et exprime le comportement de ces éléments par des axiomes écrits dans une logique du premier ordre. La signature est composée des noms de données (**sorts**) définissant les domaines du discours, d'opérations munies de profils (**ops**) sur ces domaines, de prédicats (**preds**). La caractéristique de CASL par rapport aux autres langages algébriques existants est de proposer la combinaison suivante d'extensions : sous-sortage (permettant de prolonger des domaines ou de définir des ensembles de définition), contraintes de générations (**generated** ou **free**, pour restreindre les domaines aux valeurs atteignables), surcharge (qui permet de n'avoir qu'un seul symbole pour des fonctions définies sur des domaines dans la relation de sous-sortage), opérations partielles (le traitement de la partialité est que toute fonction appliquée à un argument non défini a une valeur indéfinie).

La sémantique d'une telle spécification est constituée de la signature et de la classe des algèbres sur cette signature vérifiant les axiomes. CASL est donc un langage dans lequel on ne construit pas un modèle du programme (comme dans la méthode $B$), mais on exprime les propriétés vérifiées par un tel modèle.

Pour des raisons de lisibilité et de réutilisabilité, une spécification peut être découpé en plusieurs morceaux. Chaque morceau est alors nommé et peut être relié aux autres par les opérateurs suivants : renommage (**with**), dissimulation (**hide** ou **reveal**), union (**and**), extension (**then**), extension libre (**free**), généricité et instanciation. Il est fondamental de noter que cette structuration n'apparaît pas dans la classe des modèles de la sémantique. Elle ne représente donc pas la structure du futur logiciel.

Ce point est si important qu'il nécessite un traitement particulier par le biais des spécifications architecturales de CASL. Une spécification architecturale consiste en une liste de déclarations ou de définitions d'unités architecturales ($U_i$) associées à une spécification ($SP_i$), et une expression ($OP(U_1, ..., U_n)$) décrivant la manière dont ces unités seront combinées. Les opérateurs de structuration ($OP$) utilisés sont différents de ceux ci-dessus. En effet la propriété fondamentale est de permettre le développement indépendant des unités puis d'assurer une recomposition correcte ultérieure (voir [3] pour une description plus précise).