ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

THÈSE

présentée
pour obtenir

LE GRADE DE DOCTEUR EN SCIENCES
DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN
Spécialité INFORMATIQUE

PAR

Benjamin Blanc

Prise en compte de principes architecturaux lors de la formalisation des besoins

Soutenue le 16 Décembre 2002 devant le jury composé de :

Mme. Jeanine Souquières	Présidente
M. Didier Bert	Rapporteur
Mme. Christine Choppy	Rapporteur
Mme. Françoise Tort	Examinateur
M. Michel Bidoit	Directeur de thèse

Laboratoire **S**pécification et **V**érification – CNRS UMR 8643 ;
ENS de Cachan ; 61, Avenue du Président Wilson ; 94235 Cachan Cedex

Résumé

L'écriture de la spécification algébrique d'un système informatique complexe est une tâche difficile. En observant le processus de développement de cette spécification, nous identifions des caractéristiques importantes de ce processus. Tout d'abord, un aspect incrémental lorsque l'écriture de la spécification procède étapes par étapes pour ne manipuler qu'une partie des informations à fois. Ensuite, un aspect itératif lorsque plusieurs spécifications sont écrites pour un même trait du système, ajoutant progressivement des détails d'implémentation. Enfin, un aspect modulaire, permettant ainsi de partager le développement entre plusieurs équipes indépendantes. Dans le langage CASL, les aspects incrémental et modulaire sont pris en compte par des constructions spécifiques : opérateurs de structuration ou opérateurs architecturaux.

Lorsque le cahier des charges devient complexe, l'aspect itératif devient non négligeable. Cet aspect n'est pas relié à une construction spécifique dans CASL. Nous montrons alors que l'introduction des opérateurs architecturaux pose de nombreux problèmes allant d'une réorganisation générale de la spécification, ce qui nuit à sa revue par des non-spécialistes lors de sa validation, à une réécriture complète.

L'objectif de cette thèse est de proposer une manière de gérer l'aspect itératif sans perdre la souplesse incrémentale. Pour cela nous proposons de définir un guide structurel au début du processus d'écriture ainsi qu'une technique d'écriture suivant ce guide. Une extension du langage de spécification CASL est définie afin de prendre en compte l'aspect itératif et de simplifier cette technique. La formalisation de cette extension donne lieu à une étude approfondie de la sémantique des opérateurs de structuration dans CASL. Cette approche est illustrée dans le cas particulier de la spécification des systèmes réactifs et validée sur une étude de cas : le «Contrôle d'Accès» [Led96].

Mots-clés : spécifications algébriques, CASL, processus de développement, modularité, systèmes réactifs.

Table des matières

1. Introduction

La complexité des tâches confiées aux parties logicielles des appareils qui nous entourent augmente continuellement. Certains logiciels sont utilisés dans la conduite de trains automatiques, la gestion de centrales nucléaires, ou encore le contrôle de l'émission de radiations lors d'une thérapie cancéreuse. Dans la vie de tous les jours, nous avons tous été confrontés à des problèmes imputables à ces parties logicielles. Concernant ces utilisations, les pannes sont sources de colère ou de frustration. Mais lorsqu'il s'agit de pannes dans des systèmes critiques, cela peut provoquer de véritables catastrophes humaines ou matérielles. Certaines de ces pannes informatiques célèbres sont recensées dans [Fle02]. On notera par exemple celle imputée à la partie logicielle du Therac-25, un appareil intervenant dans les thérapies cancéreuses, causant la mort de plusieurs personnes entre 1985 et 1987 à la suite de surdoses de radiations. Il devient donc crucial de faire reposer la confiance que l'on met dans ces logiciels critiques sur des bases solides.

Une des raisons de la présence d'erreurs dans les logiciels réside dans le fait que les personnes qui rédigent les cahiers des charges sont très rarement celles qui programment le logiciel. Or le cahier des charges est très souvent écrit dans un langage naturel comme le français ou l'anglais, ce qui est une source d'ambiguïtés, d'imprécisions, voire d'omissions ou de contradictions, et ce d'autant plus que la complexité des tâches à décrire est grande. Des normes ont été définies afin de standardiser la rédaction du cahier des charges et d'assurer sa qualité. Néanmoins, l'adéquation d'un document à une norme augmente la confiance, mais ne lui confère pas le statut d'une preuve irréfutable, qui est la situation idéale.

Pour cela, il est nécessaire de disposer d'un cadre mathématique, permettant de décrire de manière rigoureuse le programme à construire, et de prouver ensuite la validité d'une propriété que doit respecter le programme. La preuve peut être effectuée soit automatiquement (dans le cas du model-checking), soit guidée par un assistant de preuves (qui devient alors lui-même un logiciel critique !). Les techniques utilisées pour décrire le programme sont spécifiques à chacun de ces cas. L'utilisation de techniques formelles pour le développement et la conception de programmes critiques permet de détecter des erreurs potentielles tôt dans le processus de développement, diminuant ainsi le coût de la modification subséquente du programme.

Nous allons décrire dans ce chapitre trois parties constituant le décor général de cette thèse. Tout d'abord, nous introduisons un processus de développement de logiciels critiques intégrant une technique formelle (section 1). Nous nous intéresserons en particulier au cas de techniques impliquant des preuves non automatiques, et tout particulièrement aux techniques de spécifications algébriques. Lorsque la complexité du cahier des charges dépasse un certain seuil, il est intéressant de décrire le plus formellement possible le contexte dans lequel ce logiciel critique s'insère. Ce contexte définit l'architecture du système global. Nous présentons quelques considérations sur ce sujet dans la section 2. Cette nécessité d'une prise en compte globale se reporte également au niveau de la description formelle. Une extension du processus introduit dans la section 1 est proposée pour intégrer ces considérations architecturales au niveau de la description formelle. La recherche sur les techniques formelles produit principalement des outils à disposition des programmeurs. Peu d'attention est donnée en revanche à la manière dont ces outils peuvent être utilisés. Nous décrirons dans la section 3 quelques propositions de guides méthodologiques facilitant la construction d'un modèle mathématique du programme. Ces trois sections constituent le décor général de cette thèse, nous permettant d'introduire enfin la problématique à un niveau très général, ainsi que les contributions que nous y avons apporté (section 4).

1.1. Processus de développement à l'aide de spécifications formelles

Le principal objectif de la recherche sur les spécifications formelles est de proposer une base formelle pour le développement de programmes corrects vis-à-vis de leur cahier des charges. Étant donné une description informelle d'un programme et sa réalisation, la confiance que l'on peut avoir dans la correction du programme est augmentée si ce programme passe avec succès un certain nombre de tests, ou encore vérifie formellement certaines propriétés. Dans ce dernier cas, on doit disposer d'une description formelle du programme. Supposons que le langage de programmation dans lequel est écrit ce programme possède une sémantique formelle (c'est le cas du langage CAML [CM98] par exemple). Cette description formelle est donc le programme lui-même. Mais dans la plupart des cas (notamment des cas réels), la vérification des propriétés est trop complexe.

Une idée importante est de construire alors une description plus abstraite du programme sur laquelle il sera possible de vérifier des propriétés. Cette représentation est abstraite car elle ne prend pas en compte ce que l'on considère être des détails d'implémentation pour cette propriété. Un autre aspect de cette représentation abstraite est qu'en faisant moins appel à des notions particulières à un langage de programmation, elle se rapproche de la description informelle du programme. On peut donc envisager le problème autrement : étant donnée une description informelle, une représentation abstraite est construite sur laquelle il sera possible de vérifier des propriétés de correction, et de laquelle un programme est dérivé en y ajoutant des détails d'implémentation. La dérivation doit conserver la correction établie sur la représentation abstraite. Pour simplifier ces preuves de conservation, la dérivation se fait par petites touches, progressivement.

Dans le cas d'un système réaliste, le programme, et donc sa représentation abstraite, sont structurés de telle sorte que l'on puisse construire de grandes représentations à partir de plus petites. Cette structure décrit aussi des aspects intangibles de la connaissance du développeur, comme le degré de relations qu'il considère entre deux parties de la représentation. Cette structure pourra être utilisée lors de la dérivation, en identifiant des éléments cruciaux du futur programme, ou par un système de vérification structuré ou encore en permettant une parallélisation de la dérivation.

Nous allons détailler maintenant, d'un point de vue plus technique le processus de développement à l'aide de spécifications formelles que nous venons de présenter. Nous avons mis l'accent sur trois caractéristiques de ce processus : la correction, l'abstraction, et la modularité. Ce processus peut être décomposé en deux étapes principales : la construction d'une spécification formelle du programme à partir du cahier de charges (ce que nous avons appelé la représentation abstraite), puis la dérivation d'un programme par raffinements successifs. Nous présenterons ensuite une famille de langages de spécification utilisant ce processus de développement.

1.1.1. Description

Du cahier des charges à la spécification formelle

L'extraction d'une spécification formelle des besoins à partir d'un cahier des charges est un processus complexe et nécessairement informel. En particulier, cette étape nécessite les activités suivantes :

1. Une analyse approfondie du cahier des charges doit être effectuée. De façon évidente, cela est nécessaire afin d'obtenir une compréhension suffisante du problème devant être spécifié, et cette activité préliminaire n'est en général pas considérée comme faisant partie intégrante du processus de développement. Néanmoins, le genre d'analyse que l'on doit mettre en place avant d'écrire une spécification formelle est très utile pour détecter les points obscurs du cahier des charges qui ne seraient autrement détectés que plus tard dans le développement, ce qui causerait une hausse importante du coût du développement. Une technique d'analyse n'est pas formelle dans le sens d'une utilisation de concepts formels, mais l'est dans la mesure où une procédure systématique peut être mise en place, permettant d'éliciter les besoins qui seront traduits dans la spécification formelle.
2. Une fois qu'une compréhension suffisante du problème à spécifier est obtenue, le cahier des charges peut être traduit vers une spécification formelle. Cette étape nécessite de la part du spécifieur qu'il fournisse des interprétations pour les parties manquantes ou ambigües du cahier des charges. De plus, cette formalisation permet également de détecter des inconsistances dans le cahier des charges. Un problème important ici est de tracer les interprétations faites durant le processus de formalisation, afin de pouvoir, plus tard, prendre en compte des modifications et des changements dans le cahier des charges.
3. Un point essentiel ensuite est de vérifier l'adéquation entre la spécification obtenue et le cahier des charges : cette activité est appelée la validation de la spécification

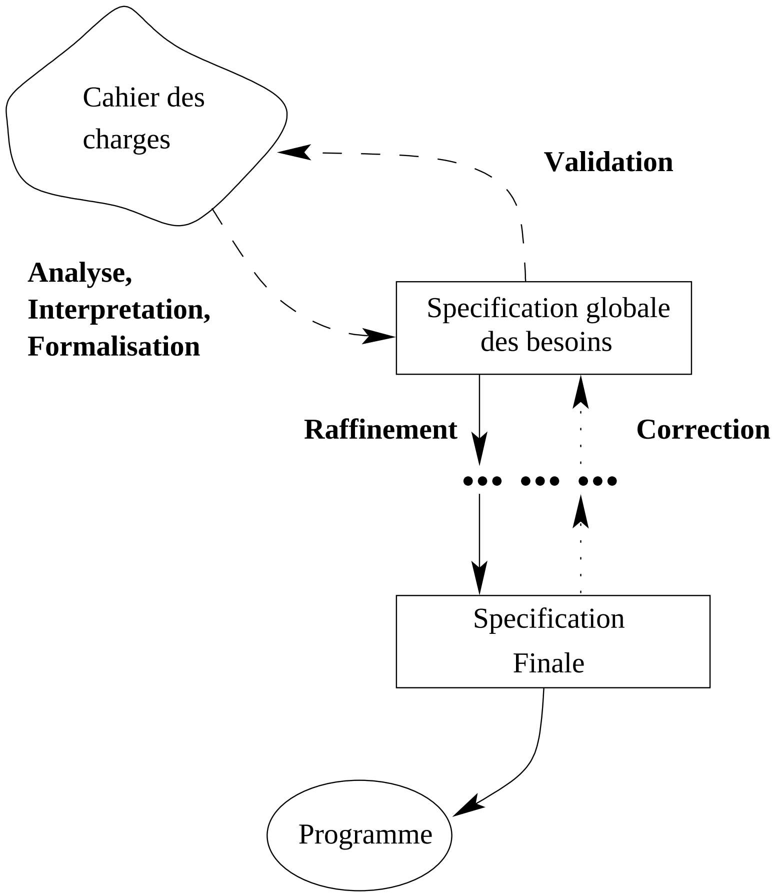

Fig. 1.1.: Processus de développement à l'aide de spécifications formelles

formelle. Cette adéquation ne peut pas être prouvée, puisqu'il n'y a aucun moyen formel d'établir la correspondance entre les interprétations et les formalisations d'une part et le cahier des charges d'autre part. Néanmoins, des expériences peuvent être effectuées, augmentant la confiance que l'on peut avoir dans la spécification formelle. Une expérience peut consister soit en la vérification qu'une propriété donnée est une conséquence de la spécification (validation interne), soit en vérifiant qu'un besoin du cahier des charges, qui n'a pas été pris en compte dans le processus de formalisation, est une conséquence de la spécification (validation externe). Dans ce dernier cas, il est nécessaire de reprendre l'analyse du cahier des charges afin de trouver une propriété de ce type.

La représentation de cette étape sur le schéma de la figure 1.1 est fortement simplifiée. En effet, cette représentation n'exprime pas le caractère incrémental et itératif de la traduction du cahier des charges vers la spécification formelle.

Incrémental Le cahier des charges d'un système réel est souvent volumineux et fort complexe, de telle sorte qu'il est impossible de l'appréhender en une seule fois. Ainsi, le spécifieur ne considère qu'une partie du cahier des charges à la fois et procède par étapes successives. Durant chaque étape, le spécifieur ne s'occupe que d'une partie et la retranscrit en une spécification formelle. La spécification globale est alors obtenue incrémentalement, en rassemblant les différentes parties.

Itératif C'est une conséquence de la nature incrémentale du processus de développement. Puisque le spécifieur ne considère qu'une seule partie du cahier des charges à la fois, il est tout à fait probable que toutes les informations nécessaires pour spécifier un trait important du système ne seront pas disponibles en une seule fois. Au contraire, chaque partie du cahier des charges comprend des informations pour plusieurs traits du système à la fois, qui seront complétées et raffinées ultérieurement.

Cette progression incrémentale facilitera ensuite la traçabilité des interprétations lors de la validation de la spécification. L'aspect itératif peut sous-entendre également le fait de modifier l'interprétation d'une information du cahier de charges, et donc de revenir sur une partie de la spécification déjà écrite pour la corriger.

Il est important de noter que la spécification formelle obtenue à la fin de cette première étape est une spécification dans laquelle est établi, aussi clairement que possible, ce qui doit être fait pour implémenter le programme, mais pas (ou le moins possible) comment le faire. En théorie, une spécification formelle des besoins ne s'occupe que des «quoi» du système, pas des «comment». En pratique, cette distinction n'est pas stricte : certaines spécifications induisent une implémentation évidente, et une implémentation constitue une spécification, même si de nombreux détails inutiles sont alors présents. L'ajout de détails d'implémentation fait partie de la seconde étape.

De la spécification formelle au programme

Une fois la spécification formelle des besoins obtenue et validée, celle-ci est raffinée afin d'obtenir une spécification plus concrète, plus proche d'une implémentation. Le raffinement d'une spécification ne se fait en général pas d'un bloc, mais procède par touches successives, petit à petit. Un exemple classique de raffinement est la transformation d'un type de données abstrait, comme les ensembles ou les collections, en un autre type, plus proche de ceux utilisés dans des langages de programmation, comme les listes ou les tableaux. Ou encore le choix d'un algorithme pour une opération qui n'était jusque là qu'esquissée. On peut imaginer le cas d'une fonction de tri d'une liste d'éléments dont on n'aurait spécifié, pendant la première étape, que le fait qu'elle rende en résultat une liste triée. Un raffinement de cette opération indiquerait alors quel algorithme utiliser pour implémenter cette opération : tri rapide, tri à bulles, tri par tas, etc. Des informations importantes sur le programme sont donc ajoutées pendant cette activité de raffinement.

Chaque pas de raffinement doit conserver la correction de la spécification vis à vis du cahier des charges que l'on a montrée pendant l'activité de validation. Puisque dans ce cas, la correction concerne deux spécifications formelles, la première plus abstraite que la seconde, la conservation de la correction peut, elle, être prouvée. Des obligations de preuves sont générées pour chaque raffinement, en fonction de la notion de raffinement choisie. En effet, il existe plusieurs notions de raffinement selon le type de langage de spécification utilisé, le domaine d'application, ou la relation que l'on veut assurer entre une spécification et ses raffinements.

La dernière activité du processus de développement consiste à dériver un programme de la spécification finale. Dans certaines approches, cette spécification finale est elle-même un programme (par exemple dans PROSPECTRA [HKB93] ou B [Abr96a]).

1.1.2. Une famille de langages

Deux grandes familles de langages de spécification formelle sont conçues autour de ce processus de développement. La première permet de décrire un unique modèle du programme en utilisant des notions mathématiques prédéfinies, comme les ensembles ou les fonctions injectives, etc. Cette famille comprend des langages comme VDM [Jon90], Z [Spi92], ou B [Abr96a]. Dans la seconde, un programme est modélisé par une algèbre décrite syntaxiquement par les spécifications. Plusieurs langages ont été développés sous ce concept : Clear [BG77], ACT ONE [EM85], Larch [GH93], EML [KST97], PLUSS [Bid89], OBJ3 [GWM+92], et récemment, CASL[ABK+02]. Nous nous plaçons dans cette famille de langages pour toute la suite de cette thèse. La présentation de cette famille de langages se fait classiquement selon certains axes : sa syntaxe, les propriétés que l'on peut écrire, sa sémantique, une relation de satisfaction reliant la sémantique aux propriétés, et sa notion de raffinement.

Une des caractéristiques essentielles du processus décrit plus haut est l'abstraction. Dans la famille des langages de spécification algébriques, la notion d'abstraction d'une spécification est caractérisée par l'indépendance vis à vis de la représentation des données. Un type de données est simplement vu comme décrivant un ensemble de valeurs, sans indications sur la manière dont sont représentées informatiquement ces valeurs. En effet la représentation des données n'est pas connue, et n'a pas besoin d'être connue, tôt dans le processus de développement, puisque l'introduction des détails d'implémentation compliquera les preuves de correction. L'implémentation d'un type de données est repoussée le plus tard possible. L'indépendance de la représentation a été popularisée par le paradigme des *types abstraits algébriques* [GTWW78].

Puisque les représentations ne sont pas connues, la manipulation des valeurs des types de données se fait par des opérations, au sens mathématique du terme, sur ces données. Dans la spécification formelle des besoins, les calculs effectués par ces opérations ne sont pas importants, seules les entrées-sorties de ces opérations sont indiquées. On cherchera simplement à définir le type des arguments ainsi que celui du résultat de cette opération, puis à caractériser le résultat sans indiquer comment celui-ci est obtenu. Par exemple, les valeurs du type des listes peuvent être manipulées à partir d'une constante (la liste vide) et d'une opération d'ajout d'un élément dans une liste. Nous avons évoqué ci-dessus le cas d'une opération de tri, prenant en argument une liste d'éléments, et possiblement une relation d'ordre, rendant une liste. Une caractérisation de cette opération est d'indiquer que cette liste contient exactement les mêmes éléments mais triés selon la relation d'ordre fournie en argument.

La notion mathématique correspondant à un ensemble de valeurs manipulé par des opérations est celle d'*algèbre*. Pour pouvoir modéliser des programmes manipulant plusieurs sortes de données, il est nécessaire d'utiliser des algèbres multi-sortées, ou hétérogènes, qui contiennent plusieurs ensembles de valeurs et des fonctions entre ces ensembles.

Dans cette approche, la spécification d'un cahier des charges consiste donc à décrire quelles algèbres sont des réalisations correctes de ce cahier des charges. Une spécification contient des symboles de sortes permettant de décrire des ensembles de valeurs et des symboles d'opération sur ces ensembles. Ces symboles définissent la syntaxe d'une algèbre, autrement appelée la *signature* de l'algèbre que l'on note en général par Σ. Pour préciser cette description, le comportement des opérations est décrit à l'aide de propriétés.

Il existe plusieurs manières d'exprimer ces propriétés. La plus simple est de ne considérer que des équations entre des termes construits sur la signature. Un terme est soit une variable, soit une constante, soit l'application d'un symbole de fonction à d'autres termes. Cette restriction sur la forme de propriétés est très contraignant. En effet, dans certains cas, il n'est pas possible de décrire abstraitement un ensemble de valeurs uniquement à l'aide d'équations, seule une implémentation particulière peut être décrite [CMR99]. C'est pourquoi les différents langages de spécification ont très vite adopté les équations conditionnelles, puis les formules de la logique du premier ordre pour décrire les propriétés des opérations. Une spécification est donc la donnée d'une signature et d'un ensemble de formules du premier ordre, noté Φ. Certains langages, comme CASL, ont une notion plus riche d'algèbre. Les signatures et les propriétés pour décrire ces algèbres sont donc également plus riches.

La sémantique d'une spécification est donnée par des modèles sur la signature Σ de la spécification dont les opérations vérifient les propriétés de Φ. Dans certains langages, la sémantique ne contient qu'un seul modèle (c'est le cas notamment dans Clear, ACT ONE, OBJ3). Cette approche ne semble pas convaincante au niveau de la spécification des besoins, puisqu'à ce stade du processus de développement, plusieurs programmes dérivés de cette spécification peuvent être envisagés, selon les choix effectués pour les structures de données et les algorithmes. Dans d'autres langages, par exemple Larch, EML, PLUSS, CASL, l'ensemble des réalisations possibles sont prises en compte. La sémantique d'une spécification est alors une classe de modèles.

La définition formelle de la sémantique est donnée par deux fonctions sur le texte de la spécification. La première construit une signature à partir du texte de spécification, et vérifie la bonne formation statique de la spécification. La seconde évalue l'ensemble des modèles sur la signature obtenue dont les opérations vérifient les propriétés de la spécification. Les opérations d'une algèbre $\mathcal{A}$ vérifient les propriétés Φ d'une spécification, noté $\mathcal{A} \models \Phi$, si pour toute propriété $\varphi \in \Phi$ et pour toute valuation des variables de φ vers des valeurs de $\mathcal{A}$, l'interprétation de ces propriétés dans $\mathcal{A}$ est vraie dans $\mathcal{A}$.

Bien que les spécifications soient plus abstraites que des programmes, dans la plupart des cas elles sont suffisamment complexes pour nécessiter l'introduction d'une structuration, permettant de maîtriser la taille et la complexité lors de l'écriture d'une spécification. Comme pour les langages de programmation, la structuration est basée sur les principes de la modularité. Des opérateurs de structuration permettent de construire des spécifications à partir d'autres spécifications. Ces opérateurs sont apparus dès le premier langage de spécification algébriques, Clear [BG77].

D'un point de vue méthodologique, la structuration d'une spécification améliore la lisibilité et facilite la réutilisation de textes de spécification. Certains opérateurs de structuration sont présents dans la plupart des langages. On retrouve l'*extension* d'une spécification par une autre spécification, le *renommage* des symboles d'une spécification, la *réduction* à une sous-signature pour masquer des opérations auxiliaires, et la *paramétrisation*. Une spécification peut être nommée pour être réutilisée par une référence à ce nom.

Pour décrire la sémantique de ces opérateurs, c'est-à-dire comment est construite la signature et la classe de modèles d'une spécification à laquelle un opérateur a été appliqué, le paradigme des catégories a été extensivement utilisé. En effet, la plupart des opérateurs de structuration se traduisent par l'application d'un morphisme sur les signatures et par l'application d'un foncteur si l'on donne une structure catégorique à la classe de modèles. Il faut bien noter que pour la plupart des langages, ces opérateurs n'induisent pas une structuration des modèles. Ainsi une spécification plate dénote le même genre d'objets qu'une spécification structurée.

Le dernier point important à formaliser est la notion de raffinement utilisée pour la deuxième étape du processus de développement. Il existe plusieurs manières de procéder selon la relation que l'on veut obtenir entre la spécification abstraite et la spécification raffinée (voir [EKMP82, ONS96] pour des revues complètes). On peut vouloir décrire la même spécification, mais dans un langage plus concret, plus proche d'un langage de programmation que des formules du premier ordre. On peut vouloir que la seconde spécification se comporte comme la première, étant donné une notion d'observation sur le comportement des spécifications. Cette notion est la plus ancienne [Hoa72] et a donné lieu à de nombreuses propositions [GHM78, EKMP82, BBC86]. On peut vouloir enfin restreindre la classe des réalisations possibles d'une spécification, en y ajoutant des détails d'implémentation. Cette notion est la plus générale et correspond à celle que nous avons sous-entendue lors de la description du processus.

Elle a été introduite dans le langage ASL [SW83]. Partant d'une spécification des besoins possédant plusieurs modèles, chaque pas de raffinement restreint la classe des modèles jusqu'à n'obtenir qu'un seul modèle, correspondant au modèle du programme. Formellement, il suffit d'imposer que les réductions des modèles de la spécification raffinée à la signature de la spécification abstraite soient des modèles de cette spécification de départ. Une spécification SP_1 de signature Σ_1 est raffinée par une spécification SP_2, noté $SP_1 \rightsquigarrow SP_2$, si $[\![SP_2]\!]|_{\Sigma_1} \subseteq [\![SP_1]\!]$. Dans la plupart des langages de spécification algébriques, le raffinement ne fait l'objet d'aucune construction syntaxique. Il reste uniquement basé sur la sémantique du langage.

Un problème important dans la vérification des pas de raffinement est que, lors du processus de développement, la spécification évolue selon deux axes. En restant dans un même niveau d'abstraction, la spécification évolue *horizontalement* afin d'ajouter de nouveaux symboles de sorte ou d'opération. Elle évolue ensuite *verticalement*, se rapprochant d'un programme. Ces deux évolutions doivent être compatibles, c'est à dire la composition des spécifications raffinées doit être un raffinement de la composition des spécifications de départ. Par conséquent, la définition de 'comment structurer une spécification' et de 'comment raffiner une spécification' sont souvent interdépendantes.

1.2. Développement par composants

Dans le cas de systèmes complexes, le processus de développement que nous venons de détailler doit être complété par une description et une analyse de la structure globale du système, c'est-à-dire son architecture.

1.2.1. Architecture logicielle

L'étude et l'analyse de l'architecture d'un système est nécessaire lorsque la taille des systèmes à développer devient trop importante pour pouvoir être appréhendée d'un seul tenant. Afin de différencier le terme de conception de celui d'architecture, nous utiliserons le premier lorsqu'il s'agit d'un découpage modulaire du système dans lequel les modules ne peuvent pas être exécutés seuls, tandis que les sous-systèmes d'une architecture sont des programmes indépendants que l'on assemble. La différence est donc principalement liée à une question d'échelle. Le lecteur pourra se reporter à [GP95, PW92, GS94, SG96, BKC97, SNH95] pour une introduction plus complète au domaine de l'architecture logicielle. Nous résumons ci-dessous quelques uns des arguments proposés dans ces articles pour justifier l'explicitation et la documentation de l'architecture :

1. **Communication entre participants** : le développement d'un programme complexe fait intervenir plus d'une seule personne, et il est tout à fait probable qu'il fasse intervenir en fait plusieurs équipes de développement. Il est alors important de

découper le développement en parties indépendantes afin que les différentes équipes puissent évoluer en parallèle, chacune développant une partie du système. L'architecture est une présentation générale du système qui peut être utilisée lors de discussions entre les participants au développement d'un programme. Elle constitue un guide tout au long du développement en indiquant les interfaces des sous-systèmes et les protocoles de communication entre ces sous-systèmes ;

2. **Aspects non-fonctionnels** : les choix architecturaux ont une profonde influence sur des besoins non-fonctionnels, comme la performance, la robustesse, la maintenabilité. Reprenons les exemples donnés dans [Som01, Chapitre 10] :
   - *Performance.* Si la performance est un besoin critique, alors l'architecture doit être construite en regroupant les opérations les plus coûteuses en temps et en ressources dans un ensemble aussi restreint que possible de sous-systèmes. La taille des sous-systèmes sera plutôt grande pour minimiser les communications dans le système.
   - *Sécurité.* Si la sécurité est un besoin critique, alors une architecture en couches pourra être utilisée, en protégeant dans les couches les plus profondes les applications les plus sensibles. La validation du système portera surtout sur ces couches.
   - *Sûreté.* Si la sûreté de fonctionnement est un besoin critique, alors les opérations mettant en jeu cette sûreté seront regroupées dans un même sous-système . Cela réduit le coût de la validation reliée à la sûreté et permet de construire une protection adaptée à ce sous-système.
   - *Disponibilité.* Si la disponibilité est un besoin critique, alors l'architecture doit inclure des redondances dans les sous-systèmes afin de pouvoir mettre à jour ou réparer une partie du système sans l'arrêter.
   - *Maintenabilité.* Si la maintenabilité est un besoin critique, alors l'architecture doit plutôt utiliser de petits sous-systèmes, bien délimités, pour lesquels il sera possible de trouver une solution de remplacement rapidement. Les producteurs de données seront en général séparés des consommateurs de données.

3. **Réutilisation** : l'architecture décrit l'organisation du système et la façon dont les composants interagissent entre eux. Une architecture peut être réutilisée pour d'autres systèmes ayant les mêmes besoins que celui pour lequel cette architecture a été développée. Au niveau des composants, la description du contexte d'utilisation de ce composant au sein du système procure des informations importantes pour la réutilisation de ce composant dans d'autres contextes.

Le choix d'une architecture est inévitable lors du développement de grands systèmes. Puisque l'architecture procure une description de haut niveau du système, celle-ci peut être déterminée très tôt dans le processus de développement. Une première spécification, appelée spécification système, est développée dans un premier temps en identifiant les sous-systèmes et leurs relations. Le processus de création d'une architecture fait intervenir plusieurs activités :

1. Identification de sous-systèmes. Des modèles spécifiques d'organisation en sous-systèmes peuvent être utilisés ici : le modèle d'entrepôt (l'échange de données entre les sous-systèmes se fait par l'intermédiaire d'une base de données partagée, ce qui réduit les communications, découple la production de l'utilisation des données, centralise la sécurité et la disponibilité, et facilite les évolutions), le modèle client-serveur (modèle distribué entre des serveurs produisant des services, comme une imprimante ou un périphérique, et des clients communiquant à travers un réseau par des messages), le modèle en couches (dans lequel chaque couche définit une machine abstraite dont le langage est utilisé par les couches supérieures, voir par exemple le protocole de communication TCP/IP). L'architecture suit alors un *style* particulier, comme décrit dans [GS94, SC97, AAG93].
2. Modélisation du contrôle. Les données ou les messages échangés entre les sous-systèmes doivent faire l'objet d'un contrôle afin de vérifier que les services sont bien fournis au bon endroit et au bon moment. Deux approches peuvent être distinguées : contrôle centralisé (un sous-système a la responsabilité des démarrages et des arrêts des autres sous-systèmes), ou contrôle basé sur des événements (chaque sous-système peut répondre à un certain nombre de messages et en générer d'autres).
3. Découpage modulaire. Chaque sous-système est décomposé en composants (ou modules), suivant les facilités de structuration du langage cible choisi (objets ou modules, principalement).

Le résultat de ce processus est en général multiple, selon plusieurs *vues* de l'architecture. Cette approche multivues de l'architecture est préconisée dans le développement d'applications industrielles [Kru95, HSN95, GHJV95]. On s'intéressera notamment à la vue conceptuelle, dans laquelle on utilisera des éléments du domaine d'application et de l'environnement à partir de leurs relations logiques, ou à la vue d'exécution, représentant l'image de la structure lors d'une exécution dans un scénario particulier, ou encore à la structure du code sur les machines physiques. La description de ces vues peut se faire dans un langage graphique, en identifiant simplement les éléments caractéristiques de chaque vue et leurs relations, en donnant une sémantique précise à ces objets, mais pas formelle. Le langage UML [FS97] est de plus en plus utilisé dans cet objectif, voir par exemple [HSN95]. L'inconvénient de ces descriptions graphiques est qu'elles ne permettent pas de vérifier formellement certaines propriétés globales, comme l'absence de blocage, ou l'ordonnancement. Un certain nombre de langages formels ont donc été développés dans l'objectif de décrire une ou plusieurs vues de l'architecture : Wright [AG97], COMMUNITY [FL97], CHAM [IW95], RAPIDE [LKA+95], SADL [HDRS99]. Ces langages sont pour la plupart basés sur un même modèle mathématique, celui des systèmes de transitions [Bla99].

1.2.2. Processus de développement arborescent

Les langages de description de l'architecture n'utilisent pas les mêmes objets mathématiques que les langages de spécification algébriques. De plus, leurs objectifs sont différents. Les langages de description d'architectures servent à formaliser ces descriptions afin de pouvoir y appliquer des algorithmes de vérification de propriétés globales au niveau de la spécification système. Les langages de spécification algébriques servent à décrire abstraitement le cahier des charges, puis de dériver un programme par des raffinements successifs. Ces deux sortes de langages peuvent être combinés dans un seul processus. Dans un premier temps, la spécification système est développée et décrite par un langage de description d'architectures. Cette description est utilisée, avec le cahier des charges initial, pour développer ensuite chaque sous-système à l'aide des techniques algébriques (ou d'autres).

Nous pouvons cependant proposer une extension du processus décrit dans la section 1.1 afin de prendre en compte des aspects structurels. En effet, comme nous l'avons mentionné, un sous-système peut lui-même être décomposé en modules. Les mêmes motivations que pour l'explicitation de l'architecture, en particulier ce qui concerne la répartition des tâches entre développeurs, peuvent être reprises ici, mais à une moindre échelle, pour introduire des outils de description de structure modulaire dans les langages de spécification algébriques. Les spécifications modulaires de ACT TWO [EM90], reprises dans [Ore99], permettent d'exprimer des modules de spécification à l'intérieur d'un langage de spécification. Un module est décrit par la spécification de ses paramètres, des éléments exportés, de ceux importés, ainsi que du corps de ce module. Le langage EML [KST97] est centré autour du développement de programmes modulaires en Standard ML [MTH90] : ses opérateurs de structuration sont des abstractions des constructions modulaires de ce langage de programmation et décrivent donc des modules de spécification. Cette idée est prolongée dans Casl, où des opérateurs de structuration particuliers ont été définis afin d'exprimer la structure modulaire du programme (nous y reviendrons dans le chapitre suivant).

À quel moment du processus de développement introduire ces descriptions modulaires ? La spécification formelle des besoins est en général trop abstraite pour pouvoir déterminer la structure du programme à ce niveau. Ceci est corroboré par des expériences de développement de systèmes d'information [FJ90], dans lesquels il a été remarqué que la structure de la spécification des besoins est souvent différente de la structure du programme. C'est donc au cours de la seconde étape, lors des pas de raffinement que ces informations modulaires sont ajoutées. On obtient ainsi un développement arborescent : la spécification des besoins est raffinée jusqu'à un point que l'on considère suffisamment avancé pour prescrire la structure modulaire du futur programme, indiquant un branchement dans le développement puisque la spécification de chaque module pourra ensuite être développée indépendamment (voir figure 1.2.2. Nous rappelons ici le problème soulevé à la fin de la section précédente concernant la compatibilité du développement horizontal et vertical au cours du processus de développement. Cela implique de fortes contraintes sur la sémantique des outils de description modulaires, que nous détaillerons dans le chapitre suivant.

1.3. Approches méthodologiques

Le développement d'un grand système est une tâche difficile. Particulièrement en ce qui concerne l'écriture d'une spécification formelle des besoins. D'abord en ce qui concerne l'écriture d'une spécification dans le cas général : lorsqu'il se trouve face à un cahier des

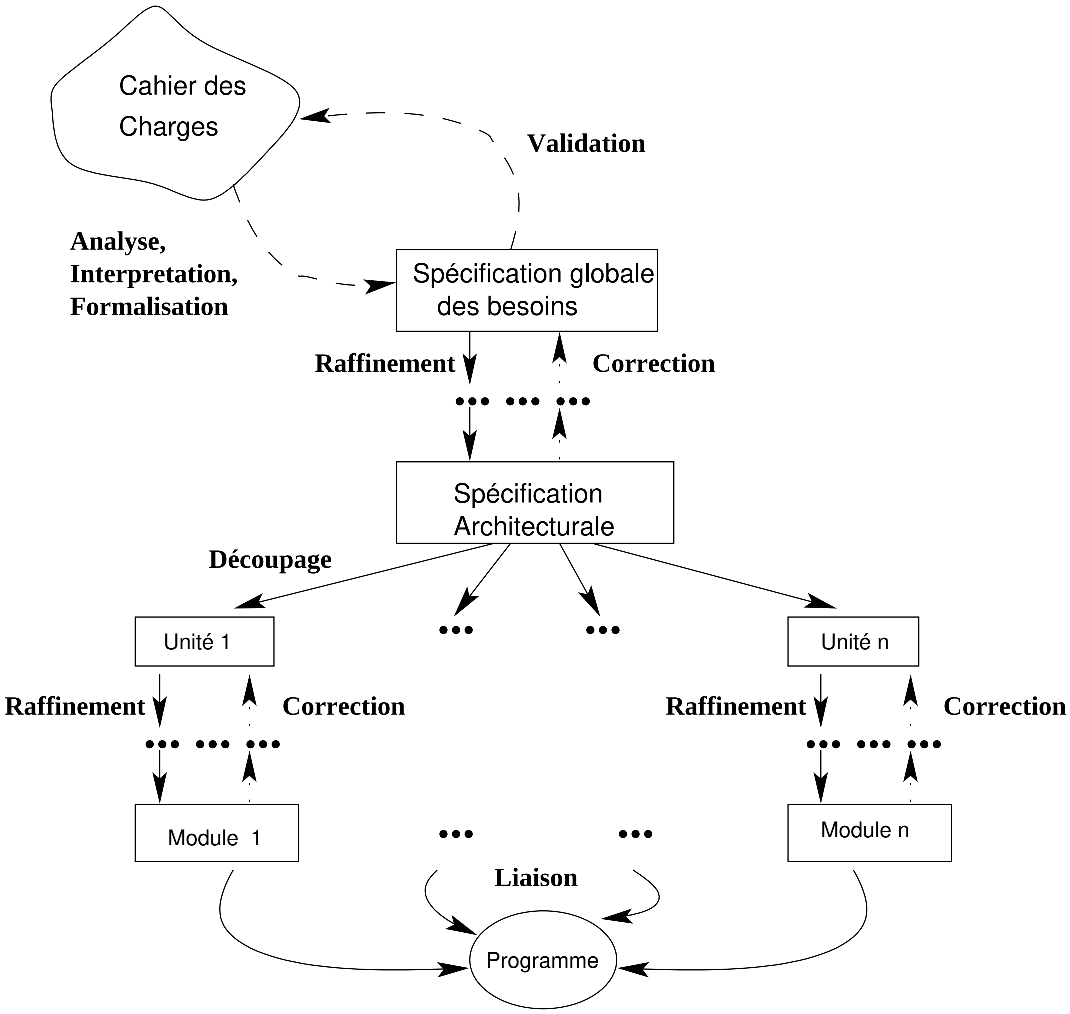

FIG. 1.2.: Processus de développement arborescent

charges, le spécifieur doit faire des choix de modélisation, en particulier de sortes et d'opérations pertinentes, qui influenceront tout le processus de développement. Ensuite en ce qui concerne les aspects formels : plusieurs formalisations d'une même opération sont possibles, qui seront plus ou moins adaptées lors des preuves effectuées pour la validation ou un pas de raffinement. La cohérence d'une spécification, c'est-à-dire l'existence d'au moins un modèle, n'est pas aisément démontrable pour des spécifications complexes. Enfin, les contraintes imposées pour les spécifications modulaires sont souvent difficiles à combiner. Toutes ces difficultés concourent à une faible dissémination des techniques formelles dans un cadre industriel. Un moyen d'atténuer ces difficultés est de proposer des méthodologies autour de ces techniques formelles.

1.3.1. Rôle de la méthodologie

Les cas d'applications de techniques formelles réussis, au niveau industriel, sont pour la plupart célèbres. Ils concernent plusieurs domaines : les transports [BBFM99], les logiciels de contrôle dans le milieu médical [Jac95b], ou interne à une grande firme [HK91], ou encore la vérification de matériel électronique [MS95]. Néanmoins ces succès ne peuvent dissimuler le fait que peu de systèmes industriels sont développés aujourd'hui à l'aide de techniques formelles. Plusieurs raisons sont avancées dans [Som01, Chapitre 9] pour expliquer ce phénomène :

- Réussites sans techniques formelles. L'utilisation d'autres techniques de génie logiciel, telles les méthodes structurées (SSADM [AG90]), la gestion de configuration, ou l'encapsulation de l'information, pendant la conception, ainsi que l'amélioration du processus de développement ont conduits à de nombreuses améliorations dans la qualité des logiciels.
- Autres contraintes. La correction du système vis à vis de son cahier des charges n'est pas la seule contrainte lors du développement d'un système. D'autres contraintes, d'ordre économique, comme le temps de livraison sont souvent primordiales lorsque le domaine dans lequel se situe le système est en évolution rapide. Les techniques formelles demandent en général un temps de développement plus long et donc peut aboutir à la livraison d'un système obsolète.
- Portée limitée. La portée des techniques formelles, telles que nous les avons présentées, est limitée à l'expression des besoins fonctionnels du système. Pourtant, dans certains cas, ce sont des propriétés non fonctionnelles qui sont primordiales (taille du code, performance, etc.). La prise en compte de contraintes méthodologiques comme la maintenabilité, ou l'ergonomie sont encore plus éloignées des techniques formelles.
- Passage à l'échelle. Le passage à l'échelle de techniques formelles est souvent difficile. Pour pouvoir passer à l'échelle, une technique doit être suffisament bien être décrite pour pouvoir être comprise par un grand nombre de personnes, et posséder un outillage bien conçu. Ces méthodes ne peuvent donc être appliquées qu'à un sous ensemble bien délimité du système.

Cette situation pourrait être améliorée si les techniques formelles de spécification étaient intégrées dans une véritable méthode. Pour cela, trois points doivent être précisés.

1. D'abord localiser les besoins d'utilisation d'une technique formelle dans le processus de développement. Tel que nous l'avons décrit dans la première partie, la technique formelle est utilisée depuis l'analyse du cahier de charges jusqu'à l'écriture du programme. D'un point de vue strictement pragmatique, la phase d'analyse est en général très intéressante, à tel point que les bénéfices acquis après cette première activité sont suffisants pour justifier l'utilisation d'outils formels, même si par manque de temps ou d'argent un développement formel complet du système n'est pas entrepris. Ou bien la phase d'écriture de la spécification est aussi effectuée, fournissant une description non ambigüe du cahier des charges et déjà orientée vers l'écriture d'un programme, mais le programme est simplement inspiré de cette spécification, et pas formellement dérivé. Des outils d'assistance au développement peuvent être utilisés tout au long de processus (approches transformationnelles).

2. Ensuite, l'introduction de techniques formelles dans un processus de développement doit tenir compte de l'existant. La modification complète du processus de développement suivi dans une entreprise peut avoir des conséquences sur la productivité. Ainsi, il sera plus efficace de transformer le processus existant afin d'y incorporer des éléments de description formels. De plus, le contexte social dans lequel est utilisé telle ou telle technique a une importance non négligeable sur son efficacité [GL95]. Cette remarque, provenant de J.Goguen, l'un des membres fondateurs de l'application de méthodes algébriques au génie logiciel, peut être prise au sérieux.

3. Enfin, il est important de se placer du point de vue de l'utilisateur pour bien comprendre ses besoins. Du point de vue du chercheur, le principal intérêt est d'utiliser un concept formel pour résoudre un problème donné, tandis que du point de vue de l'utilisateur, une procédure indiquant à quelle étape du développement et comment utiliser un concept formel est nettement plus intéressante. En effet, quel est le besoin réel du spécifieur lorsqu'il se trouve confronté à un cahier des charges ? La description précise des outils est nécessaire, mais pas suffisante. Ce qu'il y manque alors c'est une aide effective, décrivant non pas ce à quoi on doit arriver mais comment on y arrive. On retrouve ces préoccupations par exemple dans [AR00, SH99, Flo95].

   Prenons l'exemple d'un marteau. On peut décrire le manche, ses couleurs, puis la masse à une de ses extrémités, puis la fonctionnalité de cette masse : enfoncer des objets pointus dans des volumes. Le principe physique sous-jacent est celui du moment cinétique : la force transmise à l'objet pointu est proportionnelle au poids de la masse et à la longueur du manche. Pour compléter cette description du point de vue de l'utilisateur, il faudrait ajouter la manière de placer l'objet sur la surface, l'angle de frappe du marteau avec lui, puis la manière de prendre en main le marteau, quand il est bon de donner un grand coup sec ou de petits coups précis, etc. Cette description est à priori infinie, mais apporte dans la plupart des cas une information importante quant à l'utilisation d'un marteau. Dans le cas d'un langage de spécification algébrique, la description du point de vue de l'utilisateur est très compliquée mais chaque tentative apporte une aide précieuse.

1.3.2. Quelques propositions

Quelques propositions ont été faites pour décrire l'utilisation des techniques formelles du point de vue de l'utilisateur. Certaines fournissent une méthode d'écriture pour une technique donnée, d'autres une aide pour éliciter des besoins dans un cahier des charges puis en dériver une spécification, d'autre encore des outils donnant une sémantique centrée sur l'utilisateur aux étapes de raffinement.

Méthode d'écriture

Dans [Win95], J. Wing propose son retour d'expérience pour l'écriture de spécifications formelles. Deux techniques sont concernées : les langages algébriques comme Larch ou CASL et les langages construisant un modèle du système comme Z, B ou VDM. Elle propose un découpage de l'activité d'écriture d'une spécification en étapes comme suit :

1. Définir les entités observables du système, décrites par des sortes.
2. Déterminer, si besoin est, un état initial et un état final.
3. Déterminer un ensemble d'opérations sur ces entités.
4. Caractériser l'effet observable de ces opérations sur les entités.

Pour chaque étape, elle propose un ensemble de questions que le spécifieur peut se poser afin de déterminer des propriétés intéressantes des entités ou des opérations. Par exemple, pour les entités, on pourra se demander si les éléments sont ordonnés, ou si la duplication d'éléments est possible, ou encore si il y a des contraintes de taille à respecter. Pour chaque opération, si c'est une fonction ou une relation quelconque, dans le cas d'une fonction si elle est totale ou partielle, surjective, injective, bijective, si il y a une propriété d'idempotence, de reflexivité, de symétrie, de transitivité, de commutativité, d'associativité, de distributivité. Pour caractériser l'effet observable, on essaiera de définir les pré-conditions à l'application de l'opération, puis de caractériser le cas normal par une post-condition, et enfin de traiter les différents cas d'erreurs.

Cette approche propose un guide au spécifieur pour écrire une spécification, mais ne s'intéresse pas au point de départ, c'est à dire comment extraire du cahier des charges les données pertinentes. C'est l'objectif de la méthode décrite dans la section suivante.

Méthode d'élicitation

Dans [SH99], M. Heisel et J. Souquières proposent une méthode d'élicitation des besoins. L'objectif de cette méthode est de compléter et désambiguïser les besoins exprimés par le client, puis de les formaliser en une spécification des besoins. Ce travail est une continuation du travail de M.Jackson dans [JZ95]. Le principal point de différence se situe lors du passage à la spécification : une spécification doit proposer un modèle du système, et pas seulement en donner une description abstraite à partir de ses propriétés escomptées. Cette méthode est orientée vers l'écriture d'une spécification des traces d'exécution du système.

Leur méthode d'élicitation des besoins est résumée comme suit :

1. Introduire le vocabulaire du domaine ;
2. Identifier les opérations pertinentes avec leurs paramètres ;
3. Identifier tous les événements avec leurs paramètres ;
4. Classer les événements ;
5. Énoncer les faits, suppositions, et besoins ;
6. Formaliser faits, suppositions et contraintes ;

Chaque notion du vocabulaire du domaine doit être en relation avec une opération ou un événement. Il ne doit pas y avoir d'événements commandés par le système et non partagés par l'environnement (pas d'événements internes, on ne s'occupe que de l'interface). Toutes les notions utilisées par les faits, suppositions et contraintes doivent être mises en relation avec le domaine ou les événements des étapes 1 à 3. L'ensemble des contraintes doit être consistant. Pour chaque prédicat introduit lors de la formalisation, les événements qui le rendent vrai et ceux qui le rendent faux doivent être partagés par le système.

Le passage à la spécification consiste alors à formaliser les besoins, puis à les comparer aux faits, suppositions et contraintes, comme suit :

1. Définir une première approximation de l'état du système ;
2. Incrémenter la spécification en incorporant chacun des besoins :
   - Lister les événements présents dans la contrainte ;
   - Pour chaque événement, définir une opération ou mettre à jour une opération existante ;

L'état du système correspond aux entités de la méthode précédente. Les contraintes exprimant les faits ne doivent pas être violées. Tous les événements introduits dans la phase d'élicitation doivent être pris en compte. Les post-conditions doivent être complètement définies. Chaque prédicat doit être défini en termes de l'état du système.

Cette méthode a été appliquée sur plusieurs études de cas, comme le «Contrôle d'accès» [Led96], que nous traiterons également au chapitre 5. Cette méthode est issue d'un travail antérieur des auteurs sur les méthodes transformationnelles, que nous présentons ci-dessous.

Méthodes transformationnelles

Le principe des approches transformationnelles est de considérer que chaque pas de développement consiste en l'application d'un opérateur de transformation. À chaque pas, le spécifieur dispose de plusieurs opérateurs, dont la correction est prouvée une fois pour toutes, structurant ainsi la progression. Des exemples de telles méthodes sont PROSPECTRA [HKB93] ou CIP [BEH⁺87]. Dans ces méthodes, le processus de développement peut donc être lui-même formalisé. Cette formalisation a plusieurs avantages. D'abord cela permet de construire des raffinements corrects par construction. En effet, la correction d'une règle est prouvée une fois pour toutes, et seule la pré-condition doit être vérifiée au moment de l'application. L'alternative classique consiste à écrire une spécification dont on pense qu'elle est un raffinement, puis d'en vérifier la correction. Ensuite, cela permet de tracer avec précision les étapes suivies par le spécifieur au cours du développement, et donc de pouvoir localiser les étapes à modifier le cas échéant, voire même d'automatiser certaines parties de ce processus. Dans certaines méthodes, même les tentatives infructueuses sont tracées, par exemple PROPLANE [SD95]. Dans cette méthode, tout le spectre d'un développement itératif peut donc être présenté.

Certains opérateurs correspondent à des opérateurs de structuration classiques dans les langages de spécification algébriques. D'autres correspondent à des activités plus proches du spécifieur : ajouter un axiome, enlever un axiome, écrire une fonction sous forme récursive primitive, etc. Des outils ont été définis pour formaliser ces notations et permettre un suivi formel du développement. Dans [PW95], une représentation diagrammatique de la structure de la spécification est définie. Un noeud de ce graphe est une unité, qui peut être soit une spécification, soit un module d'un langage de programmation, soit des propriétés qui devront être prouvées. Un arc entre deux noeuds est soit une relation structurelle, soit une relation de propriété entre un noeud de programme et son noeud de propriété correspondant, soit une relation de développement consistant en un raffinement. Chaque opérateur de transformation modifie cette représentation. Cette représentation a été utilisée dans [Hut00] pour définir une méthodologie de développement autour de Casl.

1.4. Une première formulation de la problématique

Dans le cas de cahiers des charges complexes, le processus de développement d'un système passe par les étapes suivantes :

1. Détermination d'une spécification système, dans laquelle les composants principaux du système sont identifiés ainsi que leurs modes de communication.
2. Certains de ces composants sont jugés suffisamment critiques pour être développés à l'aide de techniques formelles, comme par exemple un langage de spécification algébrique.
3. Pour chacun de ces composants critiques, une spécification formelle des besoins est construite en traduisant les informations pertinentes du cahier des charges permettant de cerner le comportement de ces composants.
4. Cette spécification est validée, soit par des propriétés de cohérence interne, soit en déterminant des propriétés qui doivent découler de la formalisation à partir d'informations du cahier des charges non encore prises en compte.
5. La spécification des besoins peut être utilisée pour dériver un programme par des étapes de raffinements successifs. Au cours de ces étapes, un découpage modulaire du composant peut être proposé afin de continuer le raffinement de chacun des modules indépendamment des autres.

6. Une fois chaque module implémenté, le programme associé au composant est reconstruit en associant ces implémentations en fonction du découpage modulaire.

Nous nous concentrons sur les étapes d'écriture de la spécification formelle des besoins d'un composant critique, ainsi que des étapes de raffinement jusqu'à la proposition d'un découpage modulaire. Une des caractéristiques importantes de cette partie du processus est son aspect incrémental et itératif, comme nous l'avons mentionné dans la section 1.1.1. En effet, dès que le cahier des charges est un peu complexe, l'écriture d'une spécification formelle des besoins ne peut pas être effectuée d'un seul tenant, mais est nécessairement incrémentale. Ceci entraîne alors un aspect itératif, c'est-à-dire le fait de ne pas décrire une fonctionnalité ou un type de données dans tous ses détails dès le départ, mais de le compléter voire le modifier au cours du processus. Cet aspect itératif recouvre plusieurs choses : cela peut être un raffinement comme nous l'avons décrit dans la section 1.1.2, soit l'ajout de nouvelles informations concernant un aspect déjà formalisé (au même niveau d'abstraction), soit encore une modification informelle (comme la modification du profil d'une opération).

Au-delà de leur intérêt pour guider le processus de développement par des considérations sur la vision du spécifieur, les approches méthodologiques décrites dans la section précédente facilitent également la phase de validation. En effet, des intervenants extérieurs peuvent intervenir en tant que spécialistes pour déterminer des propriétés importantes du cahier des charges qui n'ont pas encore été prises en compte. Ces intervenants doivent avoir une vision claire de ce qui a été spécifié, et donc pouvoir relire la spécification. La donnée de la spécification finale n'est en général pas suffisante, à cause des concepts techniques. La donnée de tout le processus, avec ses évolutions et une trace des choix effectués donne beaucoup plus d'informations à ces spécialistes. En particulier, les approches transformationnelles permettent de suivre l'évolution de la spécification, et parfois même les corrections intermédiaires. Le suivi du raisonnement du spécifieur est donc une donnée cruciale.

Les langages de spécification possèdent des opérateurs de structuration construisant des spécifications à partir d'autres spécifications. Ces opérateurs permettent de procéder de manière incrémentale à l'écriture de la spécification des besoins, ce qui facilite également le suivi du raisonnement du spécifieur. En revanche, l'aspect itératif de l'écriture ne nous semble en général pas suffisamment bien décrit dans les langages de spécification. Certains langages, comme B [Abr96a], ont une construction pour exprimer une forme de raffinement. Dans le langage PLUSS [Bid89], une construction permet de prendre en compte une partie de l'ajout de nouvelle informations concernant un aspect déjà formalisé. Nous reviendrons plus loin sur ce langage. Néanmoins, pour être vraiment utile, une construction exprimant l'itérativité doit être compatible avec l'aspect incrémental : à chaque instant, le spécifieur doit pouvoir effectuer soit un incrément soit une itération, de manière bien distincte. Or ceci n'est pas complètement le cas dans les langages existants. De plus, les contraintes imposées à la prescription d'une structure modulaire de la spécification autorisant un développement arborescent nécessitent dans la plupart des cas une réorganisation de la spécification lorsque cet aspect itératif n'est pas clairement exprimé. Cette réorganisation nuit à la revue de la spécification par des non-spécialistes car elle fait partie des opérations non mentionnées du processus. Plus pragmatiquement, cette réorganisation implique un travail supplémentaire, allant dans le pire des cas à une réécriture complète de la spécification qui pourrait, nous le pensons, être évité.

1.4.1. Contributions

Nous proposons dans cette thèse une approche méthodologique dans le cadre du langage de spécification algébrique Casl [ABK+02]. Cette approche doit permettre de suivre un processus à la fois incrémental/itératif et arborescent. Plusieurs contributions de cette thèse vont dans ce sens :

- Dès la phase d'analyse du cahier des charges, une structure préliminaire peut être déterminée. Cette structure provient de l'expérience du spécifieur ou de structures prédéfinies pour un domaine d'application particulier. Cette structure préliminaire ne contraint pas la future structure modulaire, mais permet de guider le spécifieur au cours du processus afin qu'il soit possible d'en déterminer une. Nous parlons alors d'un processus de développement structuré.
- Une technique d'écriture distinguant les aspects incrémental et itératif du processus de développement est proposée. Cette technique indique simplement une manière de structurer une spécification.
- Cette technique peut être améliorée par une extension du langage Casl par une construction particulière l'aspect itératif. Cette construction permet de suivre l'évolution de la spécification d'un trait du programme, tout en restant compatible avec un développement incrémental.
- Un théorème vérifiant la correction de la définition formelle de cette construction est ensuite prouvé. La preuve de ce théorème nécessite un long détour par la présentation de la sémantique formelle du langage Casl.
- Cette construction est utilisée dans le cadre des systèmes réactifs. Après avoir exploré deux manières de spécifier un système réactif dans Casl et mis en évidence leurs défauts par rapport aux aspects incrémental/itératif et modulaire, nous proposons une troisième manière basée sur la nouvelle construction du langage. Cette approche est validée sur une étude de cas, le «Contrôle d'accès», [Led96].

1.4.2. Plan de la thèse

Au chapitre 2, nous détaillons la problématique dans le cas du développement de spécifications avec Casl. Nous présenterons quelques traits de ce langage, en particulier la formalisation du découpage modulaire par des *spécifications architecturales*. Nous reviendrons alors sur le processus de développement arborescent. Les contraintes méthodologiques propres à la transition entre une spécification et une spécification modulaire seront précisément décrites. Lors d'une étude de cas portant sur la spécification d'un analyseur lexical, nous mettrons en évidence les interactions entre les aspects incrémental/itératif et modulaire du processus de développement. Pour gérer ces interactions, nous proposons de modifier la manière dont la spécification est structurée.

Au chapitre 3, nous décrivons le processus de développement structuré. Dès la phase d'analyse du cahier des charges, une structure préliminaire du programme est déterminée. Cette structure représente une structuration logique du cahier des charges, mais ne correspond pas nécessairement à une structuration modulaire. Une technique d'écriture de la spécification est proposée pour tenir compte de cette structure préliminaire. Cette technique consiste principalement à regrouper les itérations sur la spécification d'un type d'intérêt dans un ensemble distingué de la spécification. Cette technique peut être améliorée par une construction supplémentaire de CASL. Après avoir énoncé les besoins que doivent combler cette construction ainsi que les différentes manières de la concevoir, nous en proposons une définition formelle. Pour cela, une première approche de la sémantique formelle de CASL est donnée.

Au chapitre 4, nous améliorons cette définition formelle en proposant une optimisation d'une des parties de cette définition. La correction de cette optimisation est prouvée par un théorème. La preuve de ce théorème fait appel aux notions complexes de la sémantique des spécifications structurées de CASL. Celles ci seront présentées auparavant en motivant les différentes notions introduites par des situations concrètes.

Au chapitre 5, nous appliquons le processus de développement structuré et la nouvelle construction du langage au domaine des systèmes réactifs. A priori, les langages de spécification algébriques ne sont pas bien adaptés à la spécification de ces systèmes. Néanmoins, si les propriétés auxquelles on s'intéresse ne peuvent être vérifiées par des méthodes automatiques (comme le Model-Checking par exemple, voir [BBF+01]), l'utilisation d'un tel langage devient pertinente. De nombreux auteurs ont cherché comment présenter la spécification de systèmes réactifs à l'aide de spécifications algébriques. Des schémas de spécification peuvent être définis pour synthétiser ces approches. Aucun de ces schémas n'est convaincant pour traiter à la fois l'aspect incrémental/itératif et modulaire du processus. Nous proposons un schéma synthétique utilisant notre construction. Ce schéma est instancié sur l'étude de cas du Contrôle d'accès.

2. Mise en oeuvre des spécifications architecturales de CASL

Nous avons présenté au chapitre 1 des considérations générales sur le processus de développement à l'aide de spécifications formelles. Deux caractéristiques de ce processus nous semblent inévitables lorsque le cahier des charges du programme à réaliser est suffisamment complexe. En premier lieu, l'écriture de la spécification suit en général une approche incrémentale, la traduction du cahier des charges vers une spécification formelle des besoins ne pouvant se faire d'un seul tenant. Des opérateurs de structuration de la spécification permettent de traduire cette approche incrémentale. Ces opérateurs permettent de construire une spécification à partir d'autres spécifications. Cet aspect incrémental entraîne alors un aspect itératif : la spécification d'un trait du programme ne se fait pas en une seule fois, mais implique plusieurs passes pendant lesquelles les diverses informations relatives à ce trait sont successivement formalisées. Enfin, lorsque la spécification est très complexe, une structure modulaire peut être déterminée afin de permettre le développement indépendant des différentes parties de la spécification.

La problématique que nous soulevons dans ce chapitre concerne les interactions entre ces caractéristiques du processus de développement. Pour cela, nous nous plaçons dans le cadre particulier du langage de spécification algébrique Casl [ABK+02]. Ce langage possède de nombreux opérateurs de structuration traduisant l'approche incrémentale du développement. Des opérateurs particuliers, appelés *opérateurs architecturaux*, permettent d'exprimer la structure modulaire du futur programme, dans le cadre d'un développement arborescent. Mais aucune construction de ce langage ne permet d'expliciter l'aspect itératif du processus. L'absence d'une construction indiquant une évolution de la spécification d'un trait s'avère particulièrement problématique lors de la transition entre une spécification usuelle et une spécification architecturale. Or si l'on ne tient pas compte de cet aspect itératif, les contraintes imposées à cette transition impliquent une réorganisation des éléments de spécification, ce qui fait perdre de nombreuses informations d'ordre méthodologique. La principale de ces informations méthodologiques dont nous discuterons est le suivi du développement, c'est-à-dire la possibilité de suivre le raisonnement du spécifieur dans la structure de la spécification.

Des aspects informels du langage Casl sont présentés dans la section 2.1. Les constructions particulières de ce langage permettant d'exprimer des spécifications modulaires sont exposées un peu plus en détail dans la section 2.2. La correction de ces constructions implique de nombreuses contraintes lors de l'étape de transition que nous détaillons dans la section 2.3. Afin de mettre en évidence la problématique, nous suivrons pas à pas le développement d'un analyseur lexical qui a fait l'objet de nombreuses spécifications, par exemple dans [LEW96, Het94], car il possède le niveau de complexité minimal pour lequel les caractéristiques incrémentales et itératives commencent à prendre tout leur sens. Cet exemple de développement sur une étude de cas simplifiée nous permettra d'illustrer les problèmes apparaissant pendant la phase de découpage. Après avoir discuté des alternatives possibles afin de contourner les problèmes apparus lors de cette étude de cas, nous justifierons une modification du processus afin de prendre en compte les aspects modulaires du système plus en amont, ainsi qu'une extension du langage pour distinguer dans la structure de la spécification ce qui a trait à une itération.

2.1. CASL

Le principal attrait du langage Casl réside dans sa combinaison de concepts et de constructions, plutôt que dans leur nouveauté. La plupart des traits de ce langage se retrouvent d'une manière ou d'une autre dans un ou plusieurs langages existants. Casl n'est pas un «langage noyau» possédant un minimum de constructions à partir desquelles d'autres peuvent être définies comme du sucre syntaxique, mais plutôt un noyau de langages à partir duquel des sur ou sous-langages peuvent être définis par extension ou restriction. Cette présentation assure une uniformité syntaxique se traduisant par une grande cohérence dans sa sémantique.

2.1.1. Domaine d'application

Casl est un langage généraliste qui peut être utilisé a priori dans n'importe quel domaine d'application. La souplesse d'écriture de spécifications structurées et de leurs propriétés grâce à une logique riche et puissante (logique du premier ordre avec partialité, contraintes de génération, et sous-sortage) laisse la possibilité à l'utilisateur d'adapter ce langage au domaine qui l'intéresse. Néanmoins, comme tous les langages de spécification algébrique, Casl est particulièrement adapté à la description des propriétés fonctionnelles des logiciels. Certaines propriétés non-fonctionnelles, comme la performance, ou des propriétés temporelles, ne sont pas directement exprimables. Casl étant conçu comme le noyau d'une famille de langages, des extensions peuvent être proposées afin d'étendre les possibilités du langage. Par exemple, Casl-LTL [RAC99] est une extension dédiée à la spécification des propriétés temporelles des systèmes réactifs.

2.1.2. Niveau d'abstraction

Casl est un langage permettant de décrire des spécifications, pas des programmes. Dans ce sens, Casl n'est pas un langage couvrant tout le spectre du développement. L'utilisation d'une sémantique par classes de modèles (*loose* en anglais) permet cependant de décrire la plupart des étapes de développement à partir de la spécification des besoins jusqu'à la spécification du programme. La possibilité de restreindre la classe des modèles par des contraintes de génération (clause **generated**) ou même de restreindre cette classe au modèle initial lorsqu'il existe (clause **free**) et la possibilité de prescrire la structure modulaire du futur programme par une spécification architecturale [BST02] affinent la description de cette spécification du programme. En revanche, aucune aide n'est fournie en CASL pour traduire un programme en une spécification ou une spécification en un programme. En particulier, le raffiput problematique.texnement, bien que précisément défini au niveau de la sémantique du langage comme une inclusion de classes de modèles, ne peut être que faiblement décrit par un morphisme entre des spécifications (construction **view**). Aucune indication sur d'éventuelles obligations de preuves ne sont indiquées, contrairement à d'autres langages.

2.1.3. Style de programmation

L'implémentation d'une spécification algébrique se fait naturellement vers un langage de programmation fonctionnel. Une spécification dont les propriétés sont exprimées seulement à partir d'équations conditionnelles formant un système de réécriture confluent est exécutable. Certains moteurs de réécriture peuvent donc être utilisés afin d'animer une spécification correspondant à un prototype du futur programme. Dans CASL, la richesse de description des propriétés des spécifications permet de décrire des spécifications non exécutables. Deux restrictions ont été définies afin d'aboutir à des spécifications exécutables dans [Mos99]. Une extension a été définie pour permettre la spécification de certains traits caractéristiques du langage de programmation Haskell à CASL[SM02].

2.1.4. Outillage

De nombreux outils ont été construits pour vérifier que certaines propriétés sont des conséquences d'une spécification. Ces outils utilisent des techniques de réécriture de termes et/ou des systèmes de déduction. Le format d'entrée de ces outils est en général un langage de spécification fixé. Aucun outil de vérification spécifique n'a été développé pour CASL. Néanmoins, puisque CASL reprend des concepts issus de divers langages de spécification algébriques, une traduction de CASL vers un langage utilisé comme format d'entrée d'un outil est possible [Mos03]. L'architecture d'une traduction est présentée dans [MKKB98]. Cette traduction est générique pour pouvoir adapter différents outils : HOL/Isabelle [Pau94], ELAN [BKK+98], ou les Development Graphs [Hut00]. Le premier niveau de traduction, l'analyse syntaxique et statique, a donné lieu à un outil nommé CATS[1].

[1] Voir sur : `http ://www.informatik.uni-bremen.de/cofi/CASL/CATS/home.html`

2.1.5. Processus de développement

L'intégration de Casl dans une méthode transformationnelle décrivant le processus suivi lors du développement de logiciels complexes est un sujet de recherche en cours [AHMS99, Hut00, MAH01, AM02, Mos02]. Cette méthode utilise les Development Graphs, un outil graphique de visualisation de l'avancement du développement permettant d'illustrer des relations entre parties de la spécification. Ces relations sont plus complexes que les opérateurs de structuration habituels. En particulier, la modification d'une spécification est possible, ce qui n'eput problematique.texst pas exprimable par un opérateur usuel. Ces relations expriment également une forme d'itération, mais de manière non compatible avec l'aspect incrémental. Cette méthode propose donc un outil externe au langage pour décrire un processus de développement incrémental et itératif.

2.2. Spécifications architecturales

Cette section présente un aspect original du langage Casl, ses *spécifications architecturales*. L'essentiel du contenu de ce chapitre est issu de [BST02, CoF01, SMT$^+$01].

2.2.1. Motivations

Le modèle arborescent évoqué dans la section 1.2.2 introduit la notion de découpage modulaire dans le processus de développement. Nous utiliserons parfois le terme modulaire, parfois le terme architectural, pour décrire ce découpage. Pratiquement, cela induit un découpage de la spécification globale en morceaux qui seront raffinés indépendamment les uns des autres. Le découpage modulaire correspond à un moment du développement où la structure du composant étudié est fixée. Par conséquent, cette structure doit apparaître au niveau des modèles de la spécification. Comme ce n'est pas le cas des opérateurs classiques (nous détaillons ce point ci-dessous), de nouveaux opérateurs sont définis dans cet objectif.

Le processus de développement arborescent peut être représenté schématiquement comme suit :

$$SP_1 \rightsquigarrow SP_2 \rightsquigarrow \ldots \rightsquigarrow SP_p \rightsquigarrow \left\{ \begin{array}{l} SP_{p,1} \rightsquigarrow SP_{p,1,fin} \\ \vdots \\ SP_{p,n} \rightsquigarrow \left\{ \begin{array}{l} SP_{p,n,1} \rightsquigarrow SP_{p,n,1,1} \rightsquigarrow SP_{p,n,1,fin} \\ \ldots \\ SP_{p,n,m} \rightsquigarrow SP_{p,n,m,fin} \end{array} \right. \end{array} \right.$$

La spécification globale est développée par des raffinements successifs (de SP_1 à SP_p), à la fois horizontaux et verticaux. Chaque SP_i est une spécification structurée. Puis, en considérant que SP_p contient suffisamment de détails pour pouvoir identifier une structure modulaire, cette spécification est découpée en sous-parties qui seront développées indépendamment les unes des autres (de $SP_p, 1$ à SP_p, n). Le processus est ensuite renouvelé sur chaque sous-spécification.

Une question cruciale se pose pour la conservation de la correction de la composition des spécifications finales vis-à-vis du cahier des charges. La correction signifie ici que les modèles de la composition des spécifications finales sont des modèles de la spécification de départ. Si la réalisation par un programme P_i de la partie de la spécification SP_i est remplacée par une autre réalisation P_i' de cette même spécification, alors pour obtenir le nouveau programme global, il doit suffire de recomposer les différentes réalisations, sans rien en modifier. La modification d'une réalisation ne doit pas influer sur les autres réalisations.

Pour cela, nous devons nous demander comment exprimer un découpage modulaire. Soit $LINK$ une construction utilisée au niveau d'un langage de programmation. Cela peut être une simple opération sur le texte du programme, comme en Pascal, une opération d'édition de liens au niveau du code compilé, en C, ou bien une véritable opération sur des modules de code, comme en ML ou en Ada. Nous ferons l'hypothèse dans la suite que le langage de programmation cible possède des constructions modulaires puissantes et bien définies (ce qui est le cas de ML ou Ada). Dans ce cas, les P_i sont des modules de programmes, tels que définis dans ces langages (*structures* en ML, *packages* en Ada) et $LINK$ est une expression de modules, ou un module générique pour lequel les paramètres formels peuvent être remplacés par des programmes P_i. Notons $[\![X]\!]$ l'ensemble des modèles de X, qui peut être une spécification ou un programme. La correction de la recomposition signifie que pour tout P_i réalisant la partie SP_i de la spécification globale, c'est à dire $[\![P_i]\!] \in [\![SP_i]\!]$, alors $[\![LINK(P_1 \ldots P_n)]\!] \in [\![SP]\!]$, où SP est la spécification globale.

On pourrait espérer que le reflet de $LINK$ au niveau du langage de spécification soit un des opérateurs de structuration OP que nous avons présentés au chapitre précédent (l'extension ou le renommage par exemple). Le branchement pourrait alors se voir comme un raffinement ordinaire $SP \rightsquigarrow OP(SP_1 \ldots SP_n)$. La répartition des tâches consisterait alors à développer indépendamment chacune des SP_i, selon la règle suivante de décomposition horizontale :

$$\frac{SP_1 \rightsquigarrow SP_1' \qquad \ldots \qquad SP_n \rightsquigarrow SP_n'}{OP(SP_1 \ldots SP_n) \rightsquigarrow OP(SP_1' \ldots SP_n')}$$

Cette règle correspond exactement à la propriété de monotonicité de l'opérateur OP vis à vis de l'inclusion de classes. La plupart des opérateurs de structuration de CASL vérifient cette propriété. Les quelques exceptions, comme imposer une contrainte d'initialité, peuvent être considérées comme un ajout de contraintes à la spécification plutôt qu'un véritable opérateur de structuration[2].

Néanmoins, cette propriété de monotonicité n'est pas suffisante pour formaliser la correction des interactions entre décomposition horizontale et verticale. En effet, pour certains opérateurs OP, il est possible de donner un modèle pour chaque SP_i, mais qu'il n'existe

[2]Le lecteur intéressé peut se référer au théorème 25, section 4.1 de [CoF01] pour une équivalence entre ces deux visions dans le cas de l'opérateur **free**.

aucun modèle recomposant ces sous-modèles à l'aide de l'opérateur OP. Ce cas ne viole pas la propriété de monotonicité, car la classe de modèles vide est incluse dans n'importe quelle autre classe, mais n'est pas satisfaisant dans la mesure où il n'apporte pas de solution constructive au découpage modulaire.

Exemple 2.1

1. Considérons les spécifications suivantes :
   spec D =
       sort s
       ops $a : s$
           $f : s \to s$
   spec E = D **then** $\forall x : s \bullet f(x) \neq a$

   Prendre cette structuration de la spécification comme un découpage modulaire signifie que pour toute réalisation de D, on doit pouvoir construire une réalisation de E. Or il existe des réalisations de D qui donnent une interprétation de f et de a qui ne sont pas compatibles avec l'axiome de E.

2. Prenons maintenant les spécifications :
   spec A =
       sort s
       ops $a, b : s$
   spec B =
       sort s
       ops $b, c : s$
   spec C = A **and** B

   Alors vouloir implémenter C en combinant les réalisations de A et B ne peut réussir que par chance. En effet, fournir une implémentation pour A signifie entre autre interpréter la constante b, de même pour toute implémentation de B. Si dans le langage utilisé, une politique de coïncidence est suivie (le même symbole dénote le même objet, dans quelque partie de la spécification que ce soit)[3], alors cette coïncidence des deux interprétations ne peut pas être assurée pour *toute* réalisation de A et de B.

□

La vraie propriété dont on aurait besoin n'est donc pas la monotonicité de OP, mais plutôt le fait que OP conserve la cohérence des spécifications. Cela signifie exactement que pour toute réalisation P_i de SP_i, il doit être possible de construire une réalisation $LINK(P_1 \ldots P_n)$ de $OP(SP_1 \ldots SP_n)$. Cette propriété est souvent appelée *constructivité* de OP [ST88]. Or tous les opérateurs de structuration usuels ne possèdent pas cette propriété (l'union de spécifications par exemple), ce qui signifie que raffiner SP en $OP(SP_1 \ldots SP_n)$ n'assure pas de pouvoir construire une réalisation de SP à partir des réalisations des SP_i.

Cette motivation justifie l'introduction dans CASL d'une deuxième sorte d'opérateurs de structuration dédiés à la prescription de la structure modulaire du programme et per-

[3] Ce qui est le cas du langage CASL.

mettant une recomposition constructive du programme étant donné des réalisations de chaque sous-partie de la spécification. Ce sont les *spécifications architecturales*.

2.2.2. Syntaxe

La syntaxe d'une spécification architecturale en CASL consiste en une collection d'unités (U_i), chacune munie d'une spécification (SP_i), et une expression d'unité ($LINK$) indiquant comment composer les unités afin d'obtenir un programme réalisant l'ensemble de la spécification. Le terme «unité» correspond à une entité de programmation, comme un «module» ou un «package». Une unité U_i est une réalisation de la spécification SP_i. Une manière de voir équivalente est de dire que SP_i est le type de l'unité U_i.

Considérons le point de branchement du développement :

$$SP \rightsquigarrow \begin{cases} SP_1 \\ \vdots \\ SP_2 \end{cases}$$

Il correspond à la spécification architecturale :
arch spec
units
    $U_1 : SP_1$
    ...
    $U_n : SP_n$
result $LINK(U_1, \ldots, U_n)$

La manière la plus générale de voir la procédure de liaison $LINK$ est de la considérer comme une fonction prenant en argument les réalisations de chaque spécification et rendant un programme réalisant SP. De manière équivalente, cette procédure de liaison peut se voir comme un opérateur d'unités (à ne pas confondre avec les opérateurs de spécification), et donc être intégré dans la partie de déclaration des unités.

Par exemple, on peut vouloir découper le développement du système en deux parties, la seconde utilisant la première. Soit SP_1 la spécification de la première partie et SP_2 celle de la seconde. On obtient donc deux unités, U_1 et U_2, chacune associée à une partie de la spécification. Réaliser ce découpage signifie que pour toute réalisation U_1 de SP_1, on doit pouvoir construire une réalisation U_2 de SP_2 utilisant cette réalisation particulière de SP_1. Une manière équivalente de voir est de fournir une réalisation générique F de SP_2, prenant la réalisation particulière de SP_1 en argument. On obtient U_2 simplement en instanciant F par U_1 :

arch spec
units
    $U_1 : SP_1$
    $F : SP_1 \rightarrow SP_2$
result $F[U_1]$

Une spécification architecturale est donc en fait une collection d'unités génériques, celles sans argument correspondent à des unités autonomes et celles avec un argument utilisent des unités autonomes pour construire de nouvelles unités autonomes, avec un terme indiquant comment appliquer les unités génériques aux unités autonomes. Les unités génériques correspondent à une forme de modules génériques dans certains langages de programmation (foncteurs en ML, generic packages en Ada). La spécification d'une unité générique indique les propriétés supposées des arguments et des propriétés garanties par le résultat, chacune par une spécification CASL. Ces spécifications sont l'interface pour le développement de ces unités. Les seules informations utilisables pour développer le corps d'une unité générique sont celles de la spécification de l'argument.

Autres opérateurs

Pour des raisons de lisibilité ainsi que de maniabilité pour l'utilisateur, d'autres opérateurs doivent être construits. Ces opérateurs correspondent pour la plupart à une version constructive des opérateurs de structuration classiques et se déduisent aisément des unités génériques. Le principe de ces constructions est le suivant. Si OP est un opérateur de structuration prenant deux spécifications SP_1 et SP_2 en argument, alors la version constructive de cet opérateur est donnée par une unité générique ayant ces deux spécifications en paramètre et produisant une spécification correspondant à l'application de l'opérateur à ces spécifications : $SP_1 \times SP_2 \rightarrow OP(SP_1, SP_2)$. Le cas le plus courant est celui d'une extension, correspondant à la version constructive de l'opérateur **then**. Cet opérateur se note **given** dans une spécification architecturale.

2.2.3. Éléments de sémantique

Une spécification architecturale dénote une *signature architecturale* $A\Sigma$ et une classe de *modèles architecturaux* $A\mathcal{M}$. Ces objets sont différents d'une signature et d'une classe de modèles usuels, car la structure des spécifications apparaît dans chacun d'entre eux. En effet, puisqu'une spécification architecturale prescrit la structure modulaire du futur système, cette structure doit être explicite dans la sémantique. Nous ne détaillerons pas dans cette section ces signatures et modèles architecturaux. Le lecteur intéressé pourra se reporter à [CoF01, SMT$^+$01]. En revanche, afin de bien comprendre pourquoi les spécifications architecturales permettent de résoudre les problèmes de découpage modulaire posés dans l'exemple 2.1, nous détaillons la sémantique des unités génériques.

Persistance

Deux problèmes distincts sont posés dans l'exemple 2.1. Le premier concerne la relation entre les arguments et le corps d'une unité générique. Pour simplifier la présentation, nous ne considérons que le cas d'une unité générique à un seul argument. Reprenons les deux premières spécifications de l'exemple 2.1 :

spec D =

sort s
ops $a : s$
$f : s \to s$
spec E = D **then** $\forall x : s \bullet f(x) \neq a$

L'opérateur d'extension de CASL n'est pas constructif. En effet, les modèles de D qui interprètent f d'une manière incompatible avec l'axiome de E n'ont pas de prolongement vers un modèle de E, bien que la spécification E soit une spécification structurée bien formée en CASL. La sémantique des unités génériques doit donc assurer qu'une spécification architecturale basée sur ce découpage est mal formée :

arch spec
units
U1 : *D*
F : *D* → *E*
result *F*[*U1*]

Lorsqu'on propose un tel découpage dans le processus de développement, cela signifie qu'une équipe va développer une réalisation de D et une autre équipe une réalisation de E, et ce quelle que soit la réalisation de D. Il s'agit donc pour la seconde équipe de construire un modèle générique et persistant de E au dessus de D. La persistance correspond à la notion d'extension conservatrice de [EM85] : une extension est conservatrice si elle ne rajoute pas de nouvelles valeurs dans les anciennes sortes, ni de nouvelles égalités. Dans une sémantique par classes de modèles, cette notion se traduit par le fait que la réduction des modèles de E à la signature de D est égale exactement aux modèles de D. Autrement dit, il s'agit de construire une fonction transformant tout modèle de D en un modèle de E, tout en préservant le modèle de D.

Plus formellement, étant donné $SP \to SP'$ la spécification d'une unité générique, Σ la signature de SP, et Σ' celle de SP' telle que $\Sigma \subseteq \Sigma'$. Alors la classe des modèles génériques et persistants de SP' au-dessus de SP, notée $\mathbf{Mod}[\Sigma \to \Sigma']$, est l'ensemble des fonctions partielles de $\mathbf{Mod}[\Sigma]$ vers $\mathbf{Mod}[\Sigma']$ conservant leur argument :

$$\mathbf{Mod}[\Sigma \to \Sigma'] = \{F \in \mathbf{Mod}[\Sigma] \to ? \ \mathbf{Mod}[\Sigma'] \mid \forall A \in Dom(F) \bullet F(A)|_{\Sigma} = A\},$$

On peut alors définir la sémantique des unités génériques à un argument comme suit :

$$[\![SP \to SP']\!] = \{F \in \mathbf{Mod}[\Sigma \to \Sigma'] \mid \forall A \in [\![SP]\!] \bullet F(A) \text{ est défini et } F(A) \in [\![SP']\!]\}$$

Cette sémantique permet bien de rendre compte du problème de la relation entre un argument et le corps d'une unité générique intervenant au moment du découpage modulaire. Dans le cas de l'exemple ci-dessus $[\![D \to E]\!]$ est vide car les modèles génériques et persistants de E au-dessus de D ne peuvent pas être définis sur tout $[\![D]\!]$. La spécification architecturale correspondante est donc mal formée.

Compatibilité

Lorsqu'on généralise l'approche ci-dessus aux unités génériques ayant plusieurs arguments, la bonne formation d'une spécification architecturale signifie aussi que les réalisations des différents arguments partagent leur interprétation des symboles communs. Ceci correspond au second problème de l'exemple 2.1, et permet de donner la sémantique des autres opérateurs de structuration architecturaux :

spec A =
  sort s
  ops $a, b : s$
spec B =
  sort s
  ops $b, c : s$

Comme nous l'avions remarqué, on ne peut pas construire pour toute réalisation de A et de B une réalisation de leur union, puisque les interprétations des symboles communs ne sont pas forcément identiques. Si on essayait cependant de traduire cette structure de la spécification en une structure du programme, on obtiendrait la spécification architecturale suivante :

arch spec
units
  $U_1 : A$
  $U_2 : B$
  $F : A \times B \to A$ **and** B
result $F[U_1, U_2]$

Lorsqu'on propose un tel découpage dans le processus de développement, cela signifie qu'une équipe va développer une réalisation de A, une autre équipe une réalisation de B, indépendamment de la première équipe, et une dernière la réalisation d'un modèle générique et persistant de l'union A **and** B au dessus de A et de B. Cette dernière équipe doit donc construire une fonction transformant tout modèle de A $\times$ B en un modèle de A **and** B. Un modèle de A $\times$ B peut se voir comme un couple formé d'un modèle de A et d'un modèle de B. Pour pouvoir être recomposés en un modèle de A **and** B, les modèles de ce couple doivent, entre autres, fournir la même interprétation pour les symboles communs de A et de B. De tels couples de modèles sont dit *compatibles*.

Plus formellement, étant donné $SP_1 \times \ldots \times SP_n \to SP$ la spécification d'une unité générique de signature $\Sigma_1 \times \ldots \times \Sigma_n \to \Sigma$, telle que pour tout i, $\Sigma_i \subseteq \Sigma$. Alors l'ensemble des couples compatibles de modèles des arguments est défini comme l'ensemble des modèles sur l'union des signatures. En effet à un modèle $A \in \mathbf{Mod}[\Sigma_1 \cup \ldots \cup \Sigma_n]$ correspond le tuple $\langle A|_{\Sigma_1} \ldots A|_{\Sigma_n} \rangle$. De même pour tout tuple compatible $\langle A_1 \ldots A_n \rangle$, il existe un unique modèle $A \in \mathbf{Mod}[\Sigma_1 \cup \ldots \cup \Sigma_n]$ tel que $A|_{\Sigma_1} = A_1 \ldots A|_{\Sigma_2} = A_2$. A s'appelle l'amalgame de $\langle A_1 \ldots A_n \rangle$.

En résumé, l'ensemble des modèles compatibles construits sur une union de n signatures est donné par un n-uplet de modèles réduits de l'amalgame de ces signatures sur chaque signature composante :

$$\mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n] = \{\langle A|_{\Sigma_1} \ldots A|_{\Sigma_n}\rangle \mid A \in \mathbf{Mod}[\Sigma_1 \cup \ldots \cup \Sigma_n]\}$$

La classe de modèles d'une signature générique à plusieurs arguments est l'ensemble des foncteurs partiels conservant ses arguments :

$$\begin{aligned}&\mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n \to \Sigma] = \\ &\quad \{F : \mathbf{Mod}[\Sigma_1 \times \ldots \times \Sigma_n] \to ?\ \mathbf{Mod}[\Sigma] \mid \forall \langle A_1 \ldots A_n\rangle \in Dom(F) \\ &\qquad F(A_1, \ldots, A_n)|_{\Sigma_i} = A_i,\ \text{pour tout}\ \ i = 1..n\}\end{aligned}$$

Notons $[\![SP_1 \times \ldots \times SP_n]\!]$ l'ensemble des n-uplets de modèles compatibles, alors la classe de modèles dénotée par une spécification générique est donnée par :

$$\begin{aligned}&[\![SP_1 \times \ldots \times SP_n \to SP]\!] = \\ &\quad \{F \in\ \mathbf{Mod}[SP_1 \times \ldots \times SP_n \to SP] \mid \forall \langle A_1 \ldots A_n\rangle \in [\![SP_1 \times \ldots \times SP_n]\!], \\ &\qquad F(A_1, \ldots, A_n)\ \text{est défini et}\ F(A_1, \ldots, A_n) \in [\![SP]\!]\}\end{aligned}$$

Comme dans le cas de la persistance, cette formalisation permet de rendre compte du problème de la relation entre les arguments d'une unité générique, puisque dans le cas de l'exemple ci-dessus $[\![A \times B \to A\ \mathbf{and}\ B]\!]$ est vide car les modèles génériques et persistants de A **and** B ne peuvent pas être définis pour tout couple de modèles de A et de B.

On pourrait néanmoins proposer un découpage bien formé de la spécification permettant effectivement de développer une réalisation de A et une réalisation de B indépendamment l'une de l'autre :

spec $SP_0 =$ **sort** s **ops** $b : s$
arch spec
units
  $U_0 : SP_0$
  $U_A : SP_A$ **given** U_0
  $U_B : SP_B$ **given** U_0
  $F : A \times B \to A$ **and** B
result $F[U_1, U_2]$

En effet, il est alors possible de vérifier que les symboles communs de A et de B ont la même interprétation, puisqu'on peut les tracer vers une unique unité autonome, U_0.

2.3. Activité de découpage

L'activité de découpage architectural est orthogonale à celle de raffinement. Le raffinement algorithmique construit une implémentation du système (ou au moins une spécification concrète) à partir d'une spécification abstraite. Il s'agit de donner une interprétation précise de certaines sortes et certaines opérations qui étaient restées abstraites au moment de la phase de formalisation du cahier des charges. Le découpage architectural correspond à une phase de répartition du développement du système en différentes parties.

Ce choix de procéder à une répartition des tâches peut intervenir a priori à n'importe quel moment de la phase de conception, voire ne jamais être effectué. De plus, le raffinement est concerné par les éléments de base des spécifications (les sortes, prédicats et opérations), tandis que le découpage architectural s'occupe de la structure du futur programme à partir de la structure de la spécification. Cette activité de découpage architectural prend en entrée une spécification structurée du système, par les opérateurs de structuration disponibles dans CASL, et doit produire une spécification architecturale bien formée raffinant cette spécification structurée.

Il y a plusieurs sortes de contraintes portant sur les spécifications pour cette activité. Tout d'abord la spécification architecturale obtenue doit être bien formée selon les critères sémantiques décrits dans la section précédente. D'autre part, le choix d'un découpage architectural a des conséquences sur des propriétés non-fonctionnelles du système. Des contraintes d'ordre méthodologique doivent alors être également prises en compte. Nous détaillons ces différentes contraintes ci-dessous.

2.3.1. Contraintes de bonne formation

Ces contraintes correspondent aux règles sémantiques d'une spécification architecturale. Informellement, la spécification d'une unité doit vérifier deux contraintes :

- Elle doit être complète pour permettre à l'unité d'être développée indépendamment. Cela signifie par exemple qu'une opération ne peut pas être déclarée dans la spécification d'une unité et définie dans celle d'une autre unité, ou qu'un constructeur d'une sorte ne peut pas être déclaré dans une spécification associée à une unité distincte de celle dans laquelle la sorte a été déclarée. Bien sur, cela n'interdit pas qu'une unité fasse appel à des opérations fournies par d'autres unités ;
- Lorsque deux spécifications d'unités partagent une même information, il doit être possible de tracer cette information vers la spécification d'une unique unité. En effet, dans le cas contraire cette information pourrait être implémentée de façon incompatible par ces deux spécifications.

La première contrainte reflète la sémantique des unités génériques. Une unité générique $F : SP_1 \rightarrow SP_2$ est bien formée si la spécification SP_2 est une extension conservatrice de SP_1. Il existe de nombreux cas dans lesquels cette propriété n'est pas vérifiée. La notion de complétude donne une intuition méthodologique des cas dans lesquels cette propriété peut être vérifiée. Néanmoins, la complétude n'équivaut pas formellement à cette propriété : ce n'est pas parce qu'une opération est déclarée et définie dans une même spécification que toute extension de cette spécification est conservatrice.

2.3.2. Contraintes méthodologiques

Comme nous l'avons montré dans le chapitre d'introduction, le choix d'une architecture a une grande importance en ce qui concerne des propriétés non-fonctionnelles du système, comme la performance, l'extensibilité, ou la tolérance aux fautes. Dans le cadre des spécifications architecturales de Casl les aspects strictement fonctionnels du système sont privilégiés, de par le formalisme des spécifications algébriques. Cependant, le choix d'une spécification architecturale déterminant aussi la structure du futur programme, certaines propositions transposables dans notre contexte peuvent être prises en compte lors de la phase de raffinement architectural.

Dans [PCW85], un style de découpage modulaire est proposé, dans lequel le principe d'encapsulation des données est prépondérant. Le critère suivi revient à n'utiliser une structure de données (une sorte) que dans une seule unité à la fois. Des fonctions d'accès à ces données sont définies dans cette unité. Si une opération d'une autre unité nécessite l'utilisation de ces données, elle ne peut le faire qu'au travers des fonctions d'accès. C'est le critère que l'on retrouve à la base du design objet.

Un autre style de découpage modulaire prend une direction différente en proposant de regrouper toutes les structures de données dans une seule unité, permettant de définir une unité correspondant à un état global du système à laquelle peuvent accéder toutes les autres unités. Ce style est appelé «entrepôt de données» dans [SG96, Som01]. Ce style permet de gérer un grand nombre de données, qui n'ont pas besoin de transiter d'une partie du système à une autre, accroît l'indépendance des sous-systèmes entre eux vis-à-vis de l'utilisation des données, centralise la gestion des erreurs. Il est bien adapté pour la spécification de systèmes réactifs.

2.3.3. Autres contraintes

Chaque unité est associée à une spécification, correspondant à l'interface à laquelle ont accès les autres unités. Cette spécification est issue de la spécification globale du système. Comme nous l'avons précisé ci-dessus, cette spécification globale est structurée, c'est-à-dire qu'elle est composée de spécifications nommées reliées entre elles par des opérateurs de structuration. Cette structure ne correspond pas en général à la structure du programme (voir [FJ90] pour des exemples), justifiant ainsi l'introduction des spécifications architecturales. En particulier, cela implique que certaines spécifications associées aux unités ne sont pas directement des spécifications nommées issues de la spécification structurée. En général, il est nécessaire de réorganiser ces spécifications afin de vérifier les critères précédents (voir l'exemple de la section 4.1 de [BST02]). Néanmoins, une contrainte supplémentaire que nous voulons prendre en compte dans l'opération de découpage est de minimiser cette étape de réorganisation afin de ne pas perdre les informations permettant de suivre le raisonnement du spécifieur.

Dans certains cas, le choix d'une structure modulaire bien adaptée, c'est-à-dire satisfaisant des besoins non fonctionnels du système, implique de modifier des choix de modélisation effectués lors des phases d'analyse et de conception. Ces choix minimisent a priori les conséquences implicites sur le design du système, par une volonté d'utiliser des objets les plus abstraits possible, mais n'en sont jamais totalement dépourvus. Nous essayons également de restreindre ces modifications lors de la phase de découpage.

2.4. Mise en oeuvre : Le Lexeur

Après avoir présenté les spécifications architecturales de CASL avec les contraintes imposées sur l'activité de découpage conséquentes, nous illustrons sur une étude de cas cette activité de découpage. Nous mettons l'accent sur le processus de développement en essayant de mettre en évidence l'aspect incrémental et itératif. Nous montrons par cette étude de cas que les contraintes imposées ne sont pas compatibles avec un développement incrémental si aucune construction du langage ne désigne une itération.

Considérons le problème classique suivant. On se propose de spécifier un analyseur lexical. Étant donné un langage défini par un ensemble de lexèmes (représentant les mots-clés du langage) et une chaîne de caractères, l'analyse lexicale consiste à découper la chaîne en sous-chaînes correspondant à des lexèmes. Un lexème est représenté par une expression régulière, dont nous donnerons la définition précise ultérieurement. Dans la conception d'un compilateur, cette analyse est préliminaire à l'analyse syntaxique s'occupant, elle, de la structure de ces sous-chaînes de caractères.

Exemple 2.2
Soit a, b, c des caractères, et $\{ac, a^*, cb, ba\}$ l'ensemble des lexèmes du langage. L'opérateur $\_^*$ signifie zéro ou plusieurs occurrences du symbole auquel il est appliqué. L'analyse lexicale de la chaîne *acbaaac* fournit les sous-chaînes ac, ba, a, ac. □

On remarque très rapidement que l'analyse lexicale peut fournir, dans le cas général, plusieurs résultats ou même aucun, s'il y a par exemple une erreur de typographie dans la chaîne.

Exemple 2.3
Avec le même langage que dans l'exemple précédent, l'analyse lexicale de la chaîne *acbaaac* fournit également les sous-chaînes a, cb, aa, ac. En revanche, l'analyse de la chaîne *acbc* ne fournit aucun résultat. □

2.4.1. Phase d'analyse

Une première tâche dans la spécification d'un système en CASL est d'identifier des noms de sortes dénotant les types de données du système, ainsi que des opérations et des prédicats sur ces sortes. Les noms choisis doivent dériver du cahier des charges afin d'améliorer la traçabilité. Les sortes identifiées ici sont : les listes (permettant de coder les ensembles), les chaînes de caractères, et les expressions régulières, que nous définissons dans un premier temps.

Listes. La spécification LIST[ELEM] suivante utilise pleinement les facilités de CASL. On déclare en une seule fois la sorte, les constructeurs, et les destructeurs des listes. Ces derniers sont définis comme des opérations partielles (représenté par un point d'interrogation), car leur domaine d'application exclut la liste vide. Les axiomes correspondants sont automatiquement générés dans la sémantique. L'utilisation du **free** permet de restreindre la sémantique au modèle libre. Les opérations usuelles sur les listes : concaténation, longueur, suppression d'un élément, et le prédicat d'appartenance sont également présentés ici. Leur comportement est contraint par des axiomes de la logique du premier ordre universellement quantifiés. Cette spécification est paramétrée par le type des éléments de la liste, et importe la spécification des entiers naturels pour ne pas interagir avec un futur paramètre contenant également la spécification des entiers.[4]

spec NAT =
  generated type $Nat ::= 0 \mid Succ\_\_(Nat)$
spec LIST [**sort** $Elem$] **given** NAT =
  free type $List[Elem] ::= Nil \mid Cons(Hd? : Elem; Tl? : List[Elem])$
  op $Append : List[Elem] \times List[Elem] \to List[Elem]$;
    $Length : List[Elem] \to Nat$;
    $Cancel : Elem \times List[Elem] \to List[Elem]$
  pred $\_\_ \in \_\_ : Elem \times List[Elem]$
  $\forall l, l' : List[Elem]; e, e' : Elem$
  • $Append(Nil, l') = l'$;
  • $Append(Cons(e, l), l') = Cons(e, Append(l, l'))$;
  • $Length(Nil) = 0$;
  • $Length(Cons(e, l)) = Succ(Length(l))$;
  • $\neg\, e \in Nil$;
  • $e \in Cons(e', l) \Leftrightarrow e = e' \vee e \in l$;
  • $\neg\, e \in Cancel(e, l)$;
end

Chaînes de caractères. Les chaînes de caractères sont vues comme des listes de caractères. La spécification des listes est donc instanciée par une spécification contenant la sorte des caractères, laissée abstraite. Certaines opérations sur les listes sont renommées pour être conformes aux notations habituelles des chaînes de caractères. Les listes de chaînes de caractères sont également définies, car c'est le type du résultat de la fonction principale. On obtient les spécifications STRING et LIST_OF_STRING suivantes :

spec CHAR = **sort** $Char$
spec STRING = LIST[CHAR **fit** $Elem \mapsto Char$] **with** $List[Char] \mapsto String, Nil \mapsto Empty,$
  $Cons \mapsto \_\_.\_\_$
spec LIST_OF_STRING = LIST[STRING **fit** $Elem \mapsto String$]

[4]Nous reviendrons sur la différence entre imports et paramètres au chapitre 4.

Expressions Régulières. L'ensemble des expressions régulières est le plus petit ensemble contenant les caractères, les concaténations d'expressions régulières, les sommes d'expressions régulières, et les itérations finies. Le symbole utilisé pour la concaténation surcharge le constructeur des chaînes de caractères. Cette ambiguïté est déchargée à l'analyse statique par le profil de ces symboles. La spécification REGEXPR suivante contient la déclaration du type correspondant, et la spécification LIST_OF_REGEXPR la spécification d'une liste d'expressions régulières représentant les lexèmes du langage :

spec REGEXPR = CHAR **then**
  free type $RegExpr ::= \varepsilon \mid sort\ Char \mid \_\_.\_\_(RegExpr; RegExpr)$
  $\mid \_\_ \mid\mid \_\_(RegExpr; RegExpr) \mid \_\_^*(RegExpr)$
spec LIST_OF_REGEXPR = LIST[REGEXPR **fit** $Elem \mapsto RegExpr$]

Une expression régulière représente un ensemble de chaînes de caractères. Cet ensemble est constitué de toutes les chaînes de caractères filtrant l'expression régulière. Le prédicat de filtrage, noté $Match$, est construit inductivement comme suit :

- ε ne filtre que la chaîne vide ;
- un caractère c ne filtre que lui-même ;
- une expression régulière $r_1.r_2$ filtre une chaîne s si et seulement si il existe deux sous-chaînes s_1 et s_2 de s telles que leur concaténation soit égale à s, r_1 filtre s_1, et r_2 filtre s_2 ;
- une expression régulière $r_1||r_2$ filtre une chaîne s si et seulement si r_1 filtre s ou r_2 filtre s (ou les deux) ;
- une expression régulière r^* filtre une chaîne s si et seulement si soit s est la chaîne vide, soit s peut être découpée en sous-chaînes dont la concaténation soit égale à s et chacune filtrant r.

La concaténation de plus de deux chaînes de caractères n'a pas encore été définie. Cette fonction est utilisée dans le dernier cas du filtrage. Celle-ci est présentée avant le prédicat de filtrage $Match$ dans la spécification MATCH :

spec MATCH = REGEXPR **and** LIST_OF_STRING **then**
  op $Concat : List[String] \to String$
  $\forall c : Char; s, s', s_1, s_2 : String; ls : List[String]$
  $\bullet\ Concat(Nil) = Empty$
  $\bullet\ Concat(Cons(s, ls)) = Append(s, Concat(ls))$
then pred $Match : String \times RegExpr$
  $\forall r, r_1, r_2 : RegExpr; c : Char; s : String;$
  $\bullet\ Match(Empty, \varepsilon);$
  $\bullet\ Match(c.Empty, c);$
  $\bullet\ Match(s, r_1.r_2) \Leftrightarrow$
    $\exists\, s_1, s_2 : String \bullet (s = Append(s_1, s_2) \wedge Match(s_1, r_1) \wedge Match(s_2, r_2));$
  $\bullet\ Match(s, r_1 \mid\mid r_2) \Leftrightarrow$
    $Match(s, r_1) \vee Match(s, r_2);$
  $\bullet\ Match(s, r^*) \Leftrightarrow$
    $(s = Nil$

$\vee \exists ls : List[String] \bullet (s = Concat(ls) \wedge$
$(\forall s' : String \bullet s' \in ls \Rightarrow Match(s', r))))$

end

Fonction principale. Un analyseur lexical peut être spécifié comme une fonction partielle *Lex* prenant deux arguments : une chaîne de caractères et la liste des expressions régulières du langage, et rendant une solution parmi toutes celles possibles. Une solution, lorsqu'elle existe, est une liste de sous-chaînes caractérisée par le fait que la concaténation des chaînes de caractères obtenue est égale à la chaîne de départ, puis que chaque sous-chaîne filtre une des expressions régulières passées en argument. Cette spécification de la fonction *Lex* est abstraite dans deux sens : d'abord car elle est indépendante du langage étudié et de la forme des chaînes de caractères autorisées ; ensuite car elle ne fournit qu'une caractérisation du résultat, mais n'indique pas comment celui-ci est calculé. Cette possibilité d'abstraction est une des forces des approches algébriques.

spec LEXEUR = MATCH **and** LIST_OF_REGEXPR **then**
op $Lex : String \times List[RegExpr] \rightarrow ? \; List[String]$
$\forall s, s' : String; lre : List[RegExpr]$
$\bullet \; Concat(Lex(s, lre)) = s$
$\bullet \; \forall s_1 : String \bullet s_1 \in Lex(s, lre) \Rightarrow$
$\exists re : RegExpr \bullet (re \in lre \wedge Match(s_1, re))$

end

La structure de la spécification est représentée dans la Figure 2.4.1. Les spécifications ont donc été parsées avec succès par l'outil CATS. Cet outil effectue une analyse statique du texte de spécification et construit automatiquement cette représentation de la structure.

2.4.2. Phase de conception

Une fois obtenue une spécification globale du problème à un niveau abstrait, l'objectif est de se rapprocher de l'implémentation. Dans un premier temps, seuls les raffinements algorithmiques nous intéressent. Ils concernent la sorte des caractères et quelques fonctions dont nous n'avons donné qu'une caractérisation de leur résultat dans la première phase.

Nous avons remarqué que l'analyse lexicale peut rendre zéro ou plusieurs résultats dans le cas général. Il y a plusieurs possibilités pour désambiguïser cette analyse : soit en modifiant la liste de lexèmes, soit en contraignant plus l'algorithme lui-même. Étant donné que nous voulons construire un analyseur lexical indépendant de cette liste, nous choisissons de modifier l'algorithme en suivant l'algorithme du plus long préfixe, qui permet de découper de manière unique la chaîne de départ. Si une expression régulière filtre un préfixe de la chaîne de départ, et qu'une autre expression régulière filtre un autre préfixe, on choisit le plus long des deux préfixes. De plus ce préfixe doit être non vide, sinon il devient possible de découper la chaîne en une infinité de sous-chaînes vides.

Cet algorithme utilise une fonction intermédiaire, notée $Scan$, prenant les mêmes arguments que *Lex*, mais rendant comme résultat deux choses : une liste des sous-chaînes de

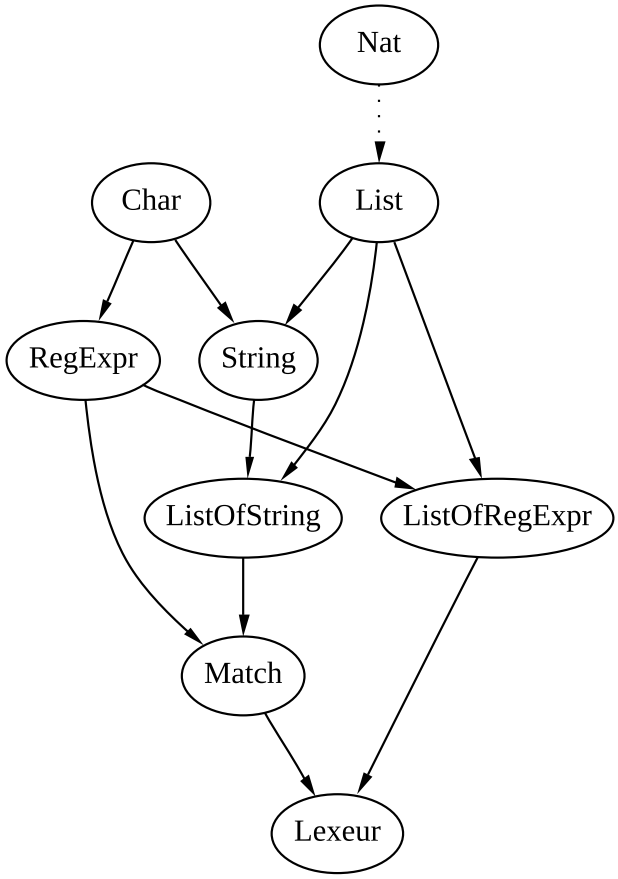

FIG. 2.1.: Analyse - Structure de la spécification

la chaîne de départ et un postfixe de la chaîne de départ n'ayant pas pu être décomposé. Le résultat de la fonction Lex est égal à la première partie dans le cas où le postfixe est vide. Elle n'est pas définie sinon.

Exemple 2.4
En reprenant l'exemple de la première section, $Scan(acbaaac, \{ac, a^*, cb, ba\}) = ((ac, ba, a, ac), Nil)$. Puis, $Scan(acbc, \{ac, a^*, cb, ba\}) = ((ac), bc)$. Dans ce dernier cas, la fonction Lex n'est pas définie. □

Les étapes suivantes du développement sont claires à partir de cette analyse : spécifier le prédicat de plus long préfixe filtrant une chaîne de caractères, puis spécifier la fonction $Scan$ en la reliant à la fonction Lex.

Plus long préfixe. La notion de préfixe et de postfixe n'ont pas été encore spécifiées. La spécification LONGEST_PREFIX_MATCH suivante contient une définition abstraite de ces prédicats, puis celle du prédicat LPM, pour *Longest Prefix Match*, prenant en argument deux chaînes de caractères et une liste d'expressions régulières. Il est vrai lorsque la première chaîne est le plus long préfixe non vide de la seconde filtrant une expression régulière de la liste passée en argument :

spec LONGEST_PREFIX_MATCH= MATCH **and** LIST_OF_REGEXPR **then**
  preds $\mathit{Isprefix} : \mathit{String} \times \mathit{String}$;
    $\mathit{Ispostfix} : \mathit{String} \times \mathit{String}$
  $\forall s, s', s_1, s_2 : \mathit{String}$
  • $\mathit{Isprefix}(s, s') \Leftrightarrow \exists\, s_1 : \mathit{String} \mid s' = \mathit{Append}(s, s_1)$;
  • $\mathit{Ispostfix}(s, s') \Leftrightarrow \exists\, s_2 : \mathit{String} \mid s' = \mathit{Append}(s_2, s)$
then **pred** $\mathit{LPM} : \mathit{String} \times \mathit{String} \times \mathit{List}[\mathit{RegExpr}]$
  $\forall s_1, s_2 : \mathit{String}; r : \mathit{RegExpr}; \mathit{lre} : \mathit{List}[\mathit{RegExpr}]$
  • $\mathit{LPM}(s_1, s_2, \mathit{lre}) \Leftrightarrow (s_1 \neq \mathit{Empty} \wedge \mathit{Isprefix}(s_1, s_2)$
    $\wedge\, \exists\, r : \mathit{RegExpr} \bullet (r \in \mathit{lre} \wedge \mathit{Match}(s_1, r))$
    $\wedge\, \forall\, s_3 : \mathit{String}, r' : \mathit{RegExpr} \bullet (\mathit{Match}(s_3, r') \wedge \mathit{Isprefix}(s_3, s_2) \wedge s_3 \neq s_1)$
      $\Rightarrow (\mathit{Length}(s_3) \leq \mathit{Length}(s_1)))$

Fonction Scan. On peut maintenant caractériser la fonction $Scan$. L'application de cette fonction à une chaîne s et une liste d'expressions régulières lre rend un couple formé d'une liste de chaîne et d'une chaîne $< ls, s' >$, tel que : s' doit être un postfixe de s ; chaque chaîne de la liste ls est une sous-chaîne de s ; la concaténation de cette liste et de s' est égale à la chaîne de départ ; si la liste ls est non vide, alors tout élément de cette liste est le plus long préfixe de la chaîne formée par cette sous-chaîne, le reste de la liste et le second élément du couple. Afin de pouvoir écrire les axiomes correspondant, nous définissons au préalable la fonction de postfixe à partir d'un élément donné sur les chaînes de caractères, ainsi qu'un type de données pour pouvoir rendre un couple en résultat de la fonction $Scan$:

spec SCAN = LIST_OF_STRING **and** LONGEST_PREFIX_MATCH **then**
  free type $\mathit{ScanResult} ::= < \mathit{Fst} : \mathit{List}[\mathit{String}]; \mathit{Snd} : \mathit{String} >$
then **op** $\mathit{Rest} : \mathit{String} \times \mathit{List}[\mathit{String}] \to \mathit{List}[\mathit{String}]$
  $\forall s, s' : \mathit{String}; \mathit{ls} : \mathit{List}[\mathit{String}]$
  • $\mathit{Rest}(s, \mathit{Nil}) = \mathit{Nil}$;
  • $s = s' \Rightarrow \mathit{Rest}(s, \mathit{Cons}(s', \mathit{ls})) = \mathit{ls}$;
  • $s \neq s' \Rightarrow \mathit{Rest}(s, \mathit{Cons}(s', \mathit{ls})) = \mathit{Rest}(s, \mathit{ls})$
then **op** $\mathit{Scan} : \mathit{String} \times \mathit{List}[\mathit{RegExpr}] \to \mathit{ScanResult}$
  $\forall s, s' : \mathit{String}; \mathit{lre} : \mathit{List}[\mathit{RegExpr}]; \mathit{ls} : \mathit{List}[\mathit{String}]$
  • $\mathit{Scan}(s, \mathit{lre}) = < \mathit{ls}, s' > \Rightarrow (\mathit{Ispostfix}(\mathit{Snd}(\mathit{Scan}(s, \mathit{lre})), s)$
    $\wedge\, \mathit{Append}(\mathit{Concat}(\mathit{ls}), s') = s$
    $\wedge\, \forall\, s_1 : \mathit{String} \bullet (s_1 \in \mathit{ls} \Rightarrow$
      $\mathit{LPM}(s_1, \mathit{Append}(\mathit{Append}(s_1, \mathit{Concat}(\mathit{Rest}(s', \mathit{ls}))), s'), \mathit{lre}))$
end
spec LEXEUR_IMPL = LEXEUR **and** SCAN **then**
  $\forall s, s' : \mathit{String}; \mathit{lre} : \mathit{List}[\mathit{RegExpr}]; \mathit{ls} : \mathit{List}[\mathit{String}]$
  • $\mathit{def}\ \mathit{Lex}(s, \mathit{lre}) \Leftrightarrow \mathit{Snd}(\mathit{Scan}(s, \mathit{lre})) = \mathit{Nil}$;

- $Lex(s, lre) = Fst(Scan(s, lre))$;

La Figure 2.4.2 représente la structure de la spécification après le début de la phase de conception. Avant de donner des descriptions constructives des divers prédicats et fonctions dont nous n'avons donné qu'une description abstraite, nous voulons découper cette spécification en parties indépendantes afin de répartir la suite de la conception entre plusieurs personnes. Il s'agit donc de construire une spécification architecturale.

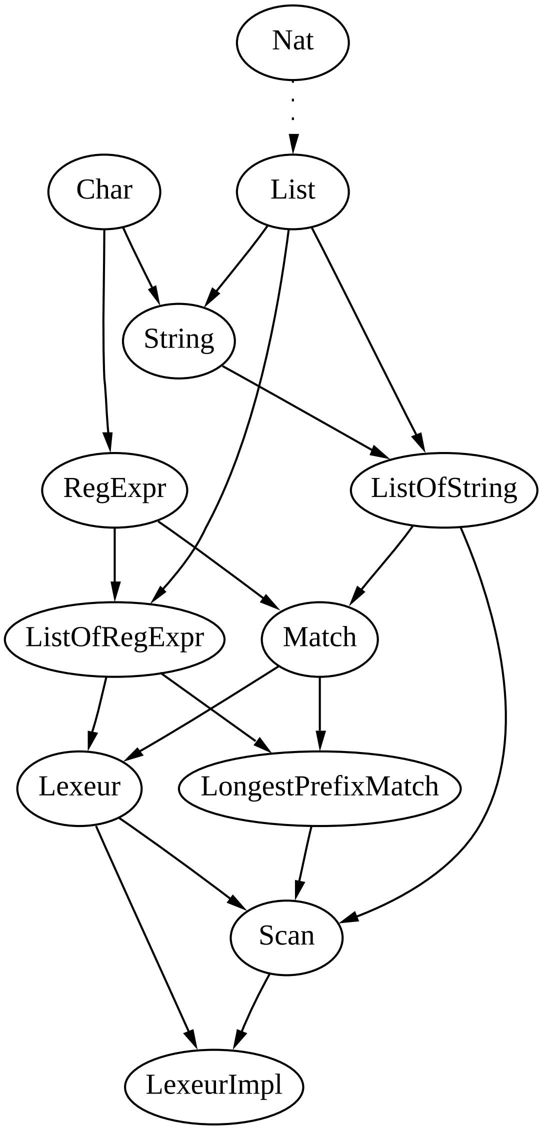

FIG. 2.2.: Conception - Structure de la spécification

2.4.3. Phase de découpage

L'objectif de cette phase est de construire une spécification architecturale bien formée, en prenant en compte les différentes contraintes mentionnées auparavant. Nous proposons une approche méthodique pour construire cette spécification architecturale à partir de la spécification globale. Le principe de cette approche est de partir d'une spécification nommée située à la fin d'un chemin du graphe de dépendance (il peut y en avoir plusieurs), puis en remontant le graphe le spécifieur choisit où couper l'arbre de dépendance pour délimiter la spécification d'une unité, en suivant les contraintes. Puis la procédure reprend sur le sous arbre restant.

Comme première contrainte méthodologique, nous choisissons le style de découpage modulaire consistant à associer la spécification d'un type d'intérêt par unité. La spécification nommée dont nous partons est LEXEUR_IMPL. Cette spécification étend LEXEUR et SCAN. Comme le domaine de définition de la fonction *Lex* déclarée dans LEXEUR est défini dans LEXEUR_IMPL, ces deux éléments de spécification doivent se trouver dans la spécification de la même unité. De même, la fonction *Scan* étant une fonction auxiliaire utilisée pour la définition de *Lex*, nous décidons de regrouper ces trois spécifications dans la même unité. SCAN contient la définition d'une fonction intermédiaire sur les listes de chaînes de caractères (*Rest*). Suivant la contrainte méthodologique, cette opération devrait plutôt faire partie d'une unité sur les listes de chaînes. Nous déplaçons donc la déclaration et la définition de cette opération dans LIST_OF_STRING, pour obtenir les spécifications :

spec LIST_OF_STRING = LIST[STRING **fit** $Elem \mapsto String$] **then**
  op $Rest : String \times List[String] \to List[String]$
  %% Les axiomes de la fonction Rest
end
spec SCAN = LIST_OF_STRING **and** LONGEST_PREFIX_MATCH **then**
  free type $ScanResult ::= < Fst : List[String]; Snd : String >$
then op $Scan : String \times List[RegExpr] \to ScanResult$
  %% Les axiomes de la fonction Scan

On redémarre de LONGEST_PREFIX_MATCH, qui peut être regroupée avec MATCH, mais en déplaçant également les opérations intermédiaires pour définir un préfixe et un postfixe dans la spécification nommée STRING :

spec STRING = LIST[CHAR **fit** $Elem \mapsto Char$] **with** $List[Char] \mapsto String, Nil \mapsto Empty,$
  $Cons \mapsto$ __.__ **then**
  preds $Isprefix : String \times String;$
    $Ispostfix : String \times String$
  %% Les axiomes de Isprefix et Ispostfix
end
spec LONGEST_PREFIX_MATCH= MATCH **and** LIST_OF_REGEXPR **then**
  pred $LPM : String \times String \times List[RegExpr]$
  %% Les axiomes de LPM
end

En ce qui concerne ListOfRegExpr, nous sommes amenés à nous poser la question de savoir si on veut regrouper toutes les opérations sur les listes, et toutes ses instanciations dans une seule unité. Cela implique de modifier encore les spécifications afin de rendre générique les prédicats et opérations sur les chaînes de caractères ou les listes de chaînes de caractères. Ce choix parait néanmoins facile à mettre en oeuvre. On obtient les spécifications :

spec List[**sort** *Elem*] **given** Nat =
%% Toutes les opérations présentes précédemment, puis
then op $Rest : Elem \times List[Elem] \rightarrow List[Elem]$
%% Les axiomes de la fonction Rest, génériques
then preds $Isprefix : List[Elem] \times List[Elem]$;
$Ispostfix : List[Elem] \times List[Elem]$
%% Les axiomes de Isprefix et Ispostfix, génériques
end
spec List_Of_String = List[String **fit** $Elem \mapsto String$]
spec String = List[Char **fit** $Elem \mapsto Char$] **with** $List[Char] \mapsto String, Nil \mapsto Empty,$
$Cons \mapsto$ __.__

On associe ensuite une unité pour les expressions régulières, puisque nous serions amenés à raffiner cette description du langage en une structure plus efficace, en suivant [BDDG93]. La spécification Char, que l'on a laissée la plus abstraite possible, est un paramètre du problème. Il est intéressant de l'associer à une unité indépendante, de telle sorte que toutes les autres unités soient paramétrées par la réalisation particulière de cette spécification.

On obtient la spécification architecturale suivante :

arch spec Lexeur =
units
U_1 : Elem;
U_2 : Char;
F : Elem → List[Elem];
$U_3 = F[U_2]$;
$U_4 = F[U_3]$;
U_5 = RegExpr **given** U_2;
$U_6 = F[U_5]$;
U_7 : Longest_Prefix_Match **given** U_4, U_6;
U_8 : Lexeur_Impl **given** U_7;
result $F[U_1, U_2]$

Cette spécification architecturale propose un découpage de la spécification globale en six modules contenant les informations nécessaires au développement d'un trait important du système et pouvant être raffinés indépendamment les uns des autres. Ce sont les unités $U_1, U_2, F, U_5, U_7, U_8$. Ces unités correspondent aux vraies tâches d'implémentation. D'autres unités n'impliquent aucune tâche, mais correspondent à des spécifications utilisées dans la spécification : U_3 correspond à la spécification String, U_4 à List_Of_String, et U_6 à List_Of_RegExpr. Notons que grâce aux réorganisations effectuées sur la spécification, les fonctions sur les listes introduites au cours du développement ont été rassemblées avec les autres opérations sur les listes, ce qui simplifie les relations entre les unités.

2.5. Discussion

Les modifications nécessaires à la construction d'une spécification architecturale posent un problème méthodologique important. La réorganisation des éléments de spécification dans les spécifications nommées correspondantes, afin de pouvoir désigner ces spécifications nommées comme les interfaces des unités architecturales, fait perdre des informations importantes. La traçabilité des interprétations du cahier des charges lors de l'écriture de la spécification, puis de ses raffinements successifs, est un point crucial du développement. La traçabilité simplifie la lecture de la spécification, en particulier par une personne non spécialiste du langage lors de la validation. D'un point de vue méthodologique, la traçabilité peut être améliorée si une approche incrémentale de l'écriture de la spécification est suivie. Ainsi, le raisonnement du spécifieur peut être suivi en détail, les informations nécessaires n'étant introduites qu'au fur et à mesure. Lors de la phase de découpage, cet effort de traçabilité est diminué par les réorganisations des éléments de spécification, puisqu'alors les spécifications nommées définies pour un pas de spécification ne contiennent plus les informations correspondant à ce pas.

Dans la formalisation de l'analyse lexicale, les opérations et prédicats ne sont introduits que lorsqu'ils sont nécessaires. Certaines opérations ou prédicats sur les listes (les prédicats $IsPostfix$ et $IsPrefix$ par exemple), qui ne sont pas des opérations de base sur les listes et ne sont introduits qu'au moment opportun. Leur introduction correspond à un pas de spécification précis (l'introduction du prédicat $Longest\_Prefix\_Match$). Le déplacement de certains éléments lors de la phase de découpage (la déclaration et la définition de ces prédicats dans la spécification des listes), nécessaire pour satisfaire les différentes contraintes de l'activité de découpage, aboutit à des spécifications qui ne correspondent plus directement à un pas de spécification. Transposé à une grande échelle, ces réorganisations nuisent à la lisibilité de la spécification et à la traçabilité des interprétations. De plus, ces réorganisations impliquent un travail supplémentaire qui aurait pu être pris en compte plus tôt dans le processus.

Une question à soulever est de se demander si l'approche incrémentale est vraiment nécessaire. Nous pensons que cette approche est pleinement justifiée, en particulier dans le cas de développements complexes. Les évolutions les plus récentes dans l'étude des processus de développement vont dans cette direction de l'incrémentalité : développement «en spirale» de [Boe88], ou «incrémental» décrit dans [Som01, Chapitre 3]. Cette approche semble efficace pour permettre la traçabilité des interprétations et simplifier la lisibilité de la spécification. Au contraire, R.Hettler propose dans [Het94] une spécification des besoins du même exemple, mais en ne fournissant que la version finale, réorganisée, de la spécification. Certains choix, en particulier sur l'introduction de telle opération ou de tel prédicat, ne peuvent être justifié qu'à postériori, ce qui nous semble peu convaincant.

Une solution possible à ce problème est de formaliser les modifications apportées pendant la phase de découpage par l'application d'opérateurs de transformation issus des approches transformationnelles évoquées à la section 1.3.2. L'application de tels opérateurs serait mentionnée dans le processus, ce qui permettrait de conserver la traçabilité des interprétations. En particulier, la méthode PROPLANE [SD95] permettrait de tracer ces modifications. En effet, dans cette méthode, les opérateurs permettent de construire un développement ($Case - decomposition$, $Add - invariant$, $Add - a - component$, etc.), ou de modifier un développement ($Backtrack$). Dans notre cas, nous pourrions utiliser l'opérateur de modification, puis de construction de nouvelles spécifications.

Le principal inconvénient, à nos yeux, de cette approche est que certaines opérations sur les spécifications ne peuvent avoir de contrepartie formelle sur la sémantique. Certains opérateurs de construction s'apparentent à des opérateurs de construction de spécifications comme l'extension ou l'union. L'ajout d'un composant, consistant à modifier le profil d'une opération afin d'y ajouter un argument supplémentaire, ne correspond à aucune opération sur la classe de modèles. De même en ce qui concerne les opérations de modification, en particulier celles permettant de faire un retour sur le processus. La sémantique de l'opérateur de restriction sur les spécifications ne correspond pas à cette action. Nous pensons qu'il est préférable de ne modifier les spécifications que par des opérateurs ayant une sémantique précise sur la sémantique des spécifications, et donc de rester à l'intérieur du langage.

Reprenons le problème au niveau de l'activité de découpage. Le problème que l'on a rencontré est de vouloir regrouper les informations constitutives d'un trait du système, que ce soit une structure de données ou une opération centrale, pour suivre les contraintes de bonne formation et méthodologiques de l'activité de découpage. Le choix d'un autre style de découpage modulaire n'aurait pas changé ce besoin de regroupement. Or il est apparu que les informations constitutives d'un trait sont souvent réparties sur plusieurs spécifications nommées. En effet, puisque la tâche d'écriture de la spécification est incrémentale et itérative, la structure de la spécification reflète deux aspects antagonistes :

- D'une part la structure reflète le processus d'écriture, c'est à dire l'ajout de nouvelles informations au fur et à mesure que la compréhension du problème se précise. Cet aspect comprend deux choses : soit il s'agit d'informations concernant un nouveau trait du système, soit il s'agit d'informations venant compléter un trait déjà évoqué. Comme nous l'avions remarqué, une conséquence de l'incrémentalité est l'itérativité de la spécification d'un trait, ce qui signifie qu'un trait est spécifié en plusieurs fois. Par exemple, la fonction *Lexeur* est spécifiée en plusieurs fois, de même pour le trait des listes de chaînes de caractères, où des opérations supplémentaires sont ajoutées en temps voulu ;
- D'autre part, la structure reflète une structuration naturelle du problème, apparaissant avec l'expérience du spécifieur. Celui-ci, s'il a déjà été confronté à un problème similaire, connaît par avance certains traits du problème qu'il faudra traiter et qu'il dégage souvent plus tôt dans le processus que ne l'aurait fait un novice. Cette structure naturelle est souvent la base d'une spécification architecturale possible pour le système.

Nous pensons qu'il est préférable de différencier ces deux aspects de la structure au cours du processus de développement. En particulier, nous pensons que l'objectif dans lequel une spécification est écrite a une influence prépondérante sur sa structure. Une spécification écrite dans l'objectif de fournir une spécification formelle de besoins, mais qui ne sera pas prolongée en une spécification concrète en vue d'une implémentation, ne sera pas structurée de la même manière qu'une spécification conçue dans l'objectif de s'en servir pour tout le processus de développement. Nous voudrions proposer une manière de structurer la spécification différenciant ces deux aspects, prenant en compte à la fois le suivi du processus ainsi que la structure naturelle émergeant au fur et à mesure de l'écriture.

2.6. Proposition

Nous pensons que les problèmes liés à la structuration de la spécification peuvent être pris en charge au moment de l'écriture de la spécification. Lors de l'ajout de nouvelles informations, le spécifieur peut se poser la question de savoir si cet ajout concerne l'évolution d'un trait déjà spécifié (itération), ou bien si cela concerne un nouveau trait (incrément). Pour cela, certains traits caractéristiques du système peuvent être identifiés dès le départ, selon l'expérience du spécifieur, servant de guide à ces questions d'organisation. Cette structure préliminaire synthétise l'expérience du spécifieur en l'équivalent d'un style architectural au niveau de chaque sous-système. La structure préliminaire peut être construite selon un style de découpage modulaire choisi, et selon le domaine d'application du système. L'évolution de la spécification de ces traits se fait en parallèle, le long du processus de développement.

L'évolution de la spécification d'un trait correspond à deux choses. La première, classique, est le passage d'un niveau abstrait de spécification à un niveau plus concret, c'est-à-dire à un raffinement. La seconde est la complétion de la spécification inachevée d'un trait. Il est important, pour le suivi du raisonnement et donc la traçabilité des interprétations, de bien distinguer dans la structure ces différentes notions. Dans CASL, ces points sont souvent mélangés à cause de la triple interprétation possible de l'opérateur d'extension **then**. En effet, il peut être utilisé à la fois pour un raffinement, une complétion, ou une extension horizontale puisque la sémantique de ces trois cas est identique formellement. Mais chaque utilisation revêt une signification bien différente pour l'utilisateur. Cet distinction entre sémantique formelle et «sémantique utilisateur» a été soulevée dans [Bau99]. Nous pensons qu'il manque dans le langage une construction pour exprimer souplement l'évolution d'une spécification inachevée. Nous allons explorer plusieurs possibilités dans le chapitre suivant pour distinguer les différents aspects de la structure du point de vue de l'utilisateur.

Cette différence entre les différents aspects de la structure a déjà été étudiée dans le langage de spécification algébrique PLUSS [Bid89]. Dans ce langage, l'aspect itératif est identifié par l'utilisation de constructions spécifiques selon l'avancement de la spécification d'un «type d'intérêt». Chaque construction est associée à une notation et une sémantique particulière. La sémantique des spécifications passe ainsi de toute la classe des modèles sur la signature validant les axiomes, aux seuls modèles engendrés par des constructeurs. Des opérateurs de structuration spécifiques sont également associés à chaque notation exprimant l'aspect incrémental. Le passage d'une construction à une autre est soumis à la réalisation de certains critères stricts. Lorsque la spécification d'un type d'intérêt est suffisamment avancée, cette spécification est transformée en une spécification modulaire pouvant être développée indépendamment du reste de la spécification globale. Le passage d'une spécification à une spécification modulaire n'est donc pas effectué d'un bloc, comme dans le processus arborescent, mais successivement pour chaque type d'intérêt. Nous pensons que ce traitement de l'aspect itératif ne peut pas être transposé dans CASL à cause de la manière dont sont traitées les spécifications architecturales.

Notre approche comprend donc deux parties. Dans un premier temps, une structure préliminaire est ébauchée afin de fixer la connaissance préalable du spécifieur. Puis, en suivant cette structure préliminaire, la spécification de chacun des composants est développée en parallèle, le long des incréments du processus. L'évolution de la spécification de chaque composant est explicite dans la structure de la spécification, de telle sorte qu'il n'y ait plus de pertes d'informations et de temps au moment du découpage architectural.

3. Développement structuré

Nous avons proposé dans le chapitre 1, dans le cas de systèmes complexes, une extension du processus classique de développement à l'aide de techniques formelles. Dans cette extension, une première spécification du système décrit la structure globale du système (son architecture). Les principaux composants sont identifiés ainsi que les interfaces entre ces composants, en terme de protocoles de communication. Cette description peut être formalisée dans un langage spécifique, puis analysée. Une fois les composants critiques identifiés, chacun peut être développé en suivant le processus formel. Pour chaque composant, à un certain degré d'avancement du développement, un découpage modulaire peut être proposé afin de fixer la structure modulaire de ce composant. La spécification de chaque partie de cette structure modulaire est ensuite développée indépendamment. La structure du système intervient deux fois dans ce processus : une première fois au niveau de la spécification système, et une seconde fois lors du découpage modulaire des composants. Dans le premier cas il s'agit d'une structure conceptuelle (pour un système donné), dans le second d'une structure concrète (pour une implémentation donnée).

Nous avons soulevé dans le chapitre 2 une contradiction dans le déroulement du processus d'écriture de la spécification des besoins, mise en évidence lors de la phase de découpage modulaire. D'un côté, si le spécifieur suit une approche incrémentale, et donc itérative, pour l'écriture de la spécification formelle des besoins, alors la structure de cette spécification reflète deux aspects : d'une part le processus d'écriture lui-même, d'autre part une structure naturelle du problème. Mais d'un autre côté, si le processus de développement suit la description précédente, les contraintes imposées sur la structure modulaire nécessitent dans la plupart des cas une réorganisation des informations à l'intérieur de la spécification lors de la phase de découpage modulaire, ce qui fait perdre les avantages de l'approche incrémentale.

Nous proposons dans ce chapitre de prendre en compte la structure modulaire du composant critique dès la phase d'analyse du cahier des charges. Pour cela, une structure préliminaire de la spécification peut être déterminée avant le début de l'écriture (section 3.1.1). Cette structure préliminaire guide l'organisation des éléments de spécification dans la structure, mais ne contraint pas la structure modulaire de la spécification. Ce point est essentiel, puisque dans de nombreux cas la structure de la spécification est distincte de la structure d'une implémentation. Nous détaillons dans la section 3.1.2 une technique de structuration de la spécification suivant cette structure préliminaire. Le principal objectif de cette technique est de différencier les aspects incrémental et itératif du processus de développement. Après avoir appliqué cette technique à l'exemple de l'analyse lexicale, nous discuterons des avantages d'accentuer encore cette distinction en définissant une construction du langage CASL spécifiquement dédiée à l'aspect itératif (section 3.1.3). Le reste de ce chapitre est consacré à la définition formelle de cette construction dans CASL. Une première présentation de la sémantique formelle de CASL est donnée dans la section 3.2. Différentes définitions de la construction que nous proposons sont envisageables : pour illustrer les avantages et inconvénients de chacune, nous procéderons par analogie sur un langage simplifié dans la section 3.3.1. La définition choisie est alors présentée en 3.3.2.

3.1. Processus de développement structuré

3.1.1. Pour une structure préliminaire

Nous proposons donc dans une première étape d'insérer pendant la phase d'analyse du cahier des charges une étude de la structure du problème pour y déceler des traits caractéristiques. Il s'agit d'identifier des fonctionnalités ou des structures de données nécessaires à la spécification d'un composant avant de les formaliser, et possiblement de les organiser en une structure par des liens de définition/utilisation. Cette structure préliminaire permettra de guider l'écriture de la spécification en développant ensuite la spécification de ces traits en parallèle. La structure à laquelle on s'intéresse à ce stade du développement est une structure logique ou conceptuelle, et qui correspond à la structure du problème, telle qu'exprimée dans le cahier des charges. Il est important de noter que cette structure préliminaire reflète la structure du problème, mais pas forcément celle d'une implémentation de ce problème. Au niveau des composants, on s'intéresse aux interfaces et aux protocoles. Au niveau modulaire, aux structures de données et aux algorithmes.

Les traits identifiés sont des traits caractéristiques d'un domaine d'application donné et dépendent aussi de l'expérience du spécifieur. Une fois qu'un domaine d'application est identifié pour un sous-système donné, un modèle conceptuel générique, relié à ce domaine d'application, est choisi. Un tel modèle est générique car il est construit pour synthétiser l'expérience dans le développement de systèmes dans ce domaine d'application.

Cette approche est celle que M. Jackson décrit dans [Jac95a]. Il propose une approche basée sur des *Problem Frames* pour généraliser une classe de problèmes, permettant de trier le problème traité dans une certaine catégorie. L'idée est de fournir une assistance au développement, en particulier lors de la phase d'analyse, et de choisir une méthode pour résoudre le problème (avec bien sûr la certitude qu'il n'existe aucune méthode générale). Un *Problem Frame* consiste en un ensemble de traits caractéristiques du problème avec une description du domaine, des besoins, et du design du système. La sémantique d'un trait est donnée par une description informelle de ce trait. Un lien entre deux traits désigne une relation d'utilisation entre ces traits : si T_1 est relié à T_2, alors T_2 utilise des informations contenues dans T_1. La sémantique d'une structure préliminaire est volontairement peu précise pour ne pas contraindre la phase d'écriture. En complément de cette structure préliminaire, le spécifieur a la possibilité de choisir à priori un style de découpage modulaire : regrouper les déclarations de sortes dans un seul module ou un module par sorte, par exemple. Les *Design Patterns* de [GHJV95] correspondent à la même idée de fournir une structure préliminaire, guidant le développement, mais dans le cadre de langages de spécification utilisant le paradigme objet.

Comment utiliser cette description au cours du développement ? Dans [CR99], C. Choppy et G. Reggio proposent une méthodologie d'application des *Problem Frames* avec le langage CASL. Leur objectif est de formaliser les *Problem Frames* sous la forme d'une structure de la spécification, complétée par des contraintes de correction entre les différentes spécifications nommées. Dans les exemples qu'elles traitent, le problème est entièrement inclus dans un *Problem Frames* : aucun nouveau module n'apparaît au cours du développement, aucune modification de la structure n'est envisagée. Leur méthode ne prend pas en compte l'aspect incrémental du développement qui est essentiel pour nous. Nous proposons plutôt de considérer la structure préliminaire comme une indication de structure de la spécification, pouvant évoluer au cours du développement vers une structure modulaire. À chaque incrément, le spécifieur se pose alors la question de savoir si les nouvelles informations qu'il ajoute correspondent à la spécification de tel ou tel trait, comme nous le verrons plus loin.

Application à l'exemple de l'analyse lexicale

Le *Problem Frame* correspondant au domaine d'application de l'analyse lexicale est celui de «traduction». Ce *Problem Frame* s'utilise lorsque le problème prend un flot de données en entrée, traite ce flot de données, puis rend un autre flot de données pouvant être utilisé par un autre composant. La description de ce cadre est très générale, et pourrait s'appliquer à de nombreux problèmes. Mais certains aspects importants de ces problèmes ne seraient alors peut être pas pris en compte. En revanche, en ce qui concerne des problèmes liés à la compilation, ce cadre capture bien l'intuition.

Ce cadre est constitué d'un composant contenant les types de données des entrées (Inputs), un autre ceux de sorties (Outputs), un contenant la relation entre entrées et sorties, c'est à dire la sémantique du langage, (I/O Relationships) et un dernier composant contenant la description de la traduction (Machine). Le style de découpage naturel associé à ce cadre est de développer une unité par sorte. Dans notre cas, Inputs contient la description des chaînes de caractères et des listes d'expressions régulières ; Outputs est une liste de chaînes de caractères ; I/O Relationships contient le prédicat de filtrage ; et Machine est la fonction principale, *Lexeur*. Dans [CR99], les auteurs ajoutent des conditions de correction sur cette structure préliminaire. Par exemple, Machine doit être une extension conservatrice de Inputs et Outputs. Cette condition correspondrait à une contrainte de correction lors de la phase de découpage modulaire si cette structure était conservée jusque là.

On obtient la structure préliminaire suivante :

spec INPUTS =
%% Les chaînes de caractères et listes d'expressions régulières

spec OUTPUTS =
%% Les listes de chaînes de caractères

spec I/O_RELATIONSHIPS = INPUTS **and** OUTPUTS **THEN**
%% Le prédicat de filtrage

spec MACHINE = I/O_RELATIONSHIPS **then**
%% La fonction principale

Des traits caractéristiques du problème ont donc été identifiés dans cette structure préliminaire. En comparant avec notre traitement de l'exemple dans la section 2.4, tous les traits essentiels ont bien été identifiés. Cette structure donne un guide pour le développement, en identifiant des tâches de spécification précises. Il reste à examiner comment cette structure préliminaire va évoluer pendant l'écriture de la spécification, en particulier comment combiner l'approche incrémentale avec la donnée d'une structure préliminaire.

3.1.2. Technique de structuration

Il est entendu maintenant que la spécification doit refléter d'une certaine manière le processus de développement et ne pas fournir seulement la version finale, réorganisée. D'un autre côté, la technique de structuration que nous avons suivie lors du traitement de l'exemple dans la section 2.4 n'est pas convaincante puisque les informations constitutives d'un trait sont dispersées dans plusieurs spécifications nommées, ce qui implique un travail de réorganisation lors de l'étape de découpage modulaire. La technique que nous proposons peut être décrite comme suit. Une spécification nommée est définie pour chaque incrément de spécification. Cette spécification est nommée en référence au trait principal correspondant à cet incrément. Elle contient tous les éléments définis à l'occasion de cet incrément, en particulier des éléments constitutifs d'un trait distinct du trait principal. Cette technique peut être améliorée afin de prendre en compte à la fois la structure préliminaire et l'approche incrémentale.

Pour chaque incrément, il suffit de répartir les éléments de spécification résultant de la formalisation de cet incrément entre différentes spécifications nommées. Nous pouvons distinguer trois cas :

1. l'incrément correspond à un seul trait ;
2. l'incrément correspond à plusieurs traits ;
3. l'incrément correspond à un nouveau trait non encore identifié

Dans le premier cas, il suffit d'étendre la spécification nommée contenant la spécification du trait en question. Il est intéressant d'expliciter cette relation d'évolution de la spécification d'un trait en donnant à l'extension le même nom avec un numéro de version supérieur. Par exemple, si l'incrément actuel concerne un trait formalisé en partie dans la spécification nommée EXEMPLE, alors l'extension sera nommée EXEMPLE_2. Dans le deuxième cas, les extensions nécessaires sont définies, en suivant la même convention de nommage. Dans le troisième cas, une nouvelle spécification nommée est définie.

En suivant cette technique de structuration de la spécification, il est facile d'identifier les spécifications contenant toutes les informations constitutives d'un trait pour pouvoir les associer à une unité architecturale. La spécification complète d'un trait est celle portant le numéro de version le plus élevé. Au cours de la phase d'écriture, la structure préliminaire aura évoluée, passant de la structure du problème à une structure possible d'une solution. La structure de la spécification à la fin peut donc être utilisée pour prescrire la structure modulaire du futur programme. D'autres choix restent bien entendu possibles : plusieurs traits peuvent être associés à une seule unité, un même trait peut être associé à plusieurs unités. Dans ce dernier cas, une réorganisation à l'intérieur d'une suite de spécifications nommées sera sans doute nécessaire.

Le principe de cette technique de structuration est de prendre en compte au moment de l'écriture toutes les réorganisations que l'on pourrait être amené à faire lors de la phase de découpage. Cette question de structuration est orthogonale à toutes les questions de modélisation et d'axiomatisation, elle n'interfère pas avec elles, et n'apporte aucune difficulté supplémentaire.

Application à l'exemple de l'analyse lexicale

Avec un langage de spécification algébrique, la phase d'analyse commence souvent par une série de déclaration de types de données de base. Dans [CR99], les auteurs placent également la déclaration des types de base au départ. Dans notre cas, il s'agit du type des listes, des chaînes de caractères et des expressions régulières. Ces spécifications pourraient être vues comme la spécification d'un trait supplémentaire de la structure préliminaire, mais sont laissées indépendantes pour l'instant. Nous ne notons que la séquence de déclaration de spécifications nommées, leur contenu étant identique à celui de la section 2.4 :

spec $\textsc{List}[\textbf{sort } \mathit{Elem}]$ **given** $\textsc{Nat} = \ldots$
spec $\textsc{Char} = \ldots$
spec $\textsc{String} = \textsc{List}[\textsc{Char}\ \textbf{fit}\ \mathit{Elem} \mapsto \mathit{Char}]$ **with** $\mathit{List}[\mathit{Char}] \mapsto \mathit{String}, \mathit{Nil} \mapsto \mathit{Empty}, \mathit{Cons} \mapsto \_\_.\_\_$
spec $\textsc{RegExpr} = \ldots$

La spécification des entrées est une extension de la spécification des chaînes de caractères et des listes d'expressions régulières :

spec $\textsc{Inputs} = \textsc{String}$ **and** $\textsc{List}[\textsc{RegExpr}\ \textbf{fit}\ \mathit{Elem} \mapsto \mathit{RegExpr}]$

La spécification des sorties, c'est-à-dire des listes de chaînes de caractères, est donnée par :

spec $\textsc{Outputs} = \textsc{List}[\textsc{String}\ \textbf{fit}\ \mathit{Elem} \mapsto \mathit{String}]$

Les relations entre entrées et sorties sont exprimées par le prédicat de filtrage entre une chaîne de caractères et une liste d'expressions régulières :

spec $\textsc{I/O\_Relationships} = \textsc{Inputs}$ **and** $\textsc{Outputs}$ **then**
%% Le prédicat de filtrage *Match*
spec $\textsc{Machine} = \textsc{I/O\_Relationships}$ **then**
%% La fonction principale *Lexeur*

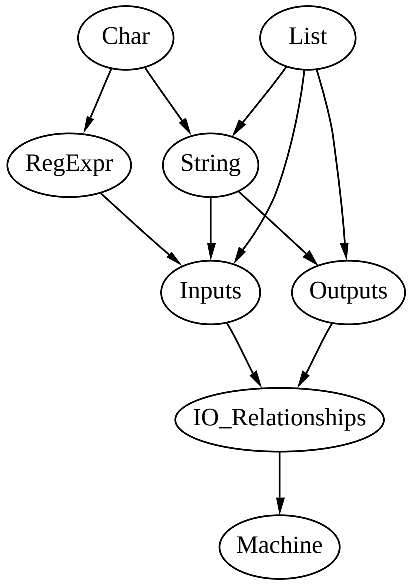

FIG. 3.1.: Analyse - Structure de la spécification

La structure de la spécification à la fin de la phase d'analyse est illustrée dans la figure 3.1.

Au début de la phase de conception, le prédicat de filtrage est raffiné en un prédicat de plus long préfixe. La spécification de ce prédicat utilise des prédicats sur les chaînes de caractères qui sont introduits auparavant comme une extension de la spécification des entrées. Nous sommes ici dans le deuxième cas où un incrément concerne deux traits. Notons que l'extension de I/O_RELATIONSHIPS doit aussi étendre la nouvelle spécification des chaînes de caractères afin de pouvoir utiliser les nouveaux prédicats.

spec INPUTS_2 = INPUTS **then**
%% Les prédicats *Isprefix* et *Ispostfix*
spec I/O_RELATIONSHIPS_2 = I/O_RELATIONSHIPS **and** INPUTS_2 **then**
%% Le prédicat *Longest_Prefix_Match*

L'incrément suivant concerne la fonction intermédiaire *Scan*. La définition de cette opération nécessite l'introduction d'une sorte *ScanResult* et d'une opération sur les listes de chaînes de caractères. Comme nous suivons un style de découpage où nous associons une sorte par composant, la déclaration de cette sorte se fait dans la même spécification nommée que pour la définition de l'opération *Scan*, c'est-à-dire une extension de MACHINE. L'opération *Rest*, utilisée dans l'axiomatisation de la fonction *Scan*, est définie auparavant dans une extension des sorties :

spec OUTPUTS_2 = OUTPUTS **then**
%% L'opération *Rest*

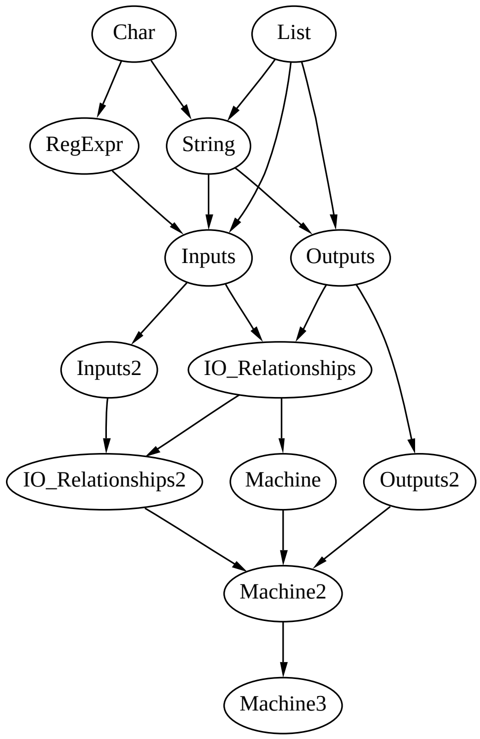

FIG. 3.2.: Conception - Structure de la spécification

spec MACHINE_2 = MACHINE **and** I/O_RELATIONSHIPS_2 **and** OUTPUTS_2 **then**
%% La sorte *ScanResult*, puis l'opération *Scan*

Enfin le domaine de définition de l'opération principale est défini ainsi que la relation entre cette opération et $Scan$:

spec MACHINE_3 = MACHINE_2 **then**
%% Caractérisation du domaine de *Lexeur* et calcul avec *Scan*

La structure de la spécification est alors présentée dans la figure 3.1.2.

Lors de la phase de découpage architectural, les spécifications nommées MACHINE_X sont associées à une seule unité, de même pour INPUTS_X, OUTPUTS_X et MATCH_X. Une unité est associée à toutes les structures de données de base. Pour chaque trait, seule la version la plus récente, contenant toutes les informations, est retenue. On obtient la spécification architecturale :

arch spec
units
U_1 : BASIC;
U_2 : INPUTS_2 **given** U_1;
U_3 : OUTPUTS_2 **given** U_1;

U_4 : I/O_RELATIONSHIPS_2 **given** U_2, U_3;
U_5 : MACHINE_3 **given** U_4
result U_5

Nous avons énoncé dans la section 2.3 les différentes contraintes de la phase de découpage architectural : d'une part la déclaration et la définition d'une opération doivent être localisées dans une même unité, les constructeurs des sortes également, d'autre part le partage d'informations doit être explicite entre unités, enfin les réorganisations au sein de la structure de la spécification doivent être minimales. Les deux premières contraintes permettent de limiter les cas de malformation de la spécification architecturale, tandis que la dernière a une signification plus méthodologique, permettant à un relecteur de suivre le processus de développement.

Toutes ces contraintes sont vérifiées par la spécification architecturale ci-dessus. L'identification des traits principaux du problème dès la phase d'analyse, puis la prise en compte de ces traits lors de l'écriture par une discipline de nommage des spécifications permet de minimiser les réorganisations qui auraient été nécessaires pour satisfaire les deux premières contraintes. Ainsi, la définition d'une opération déclarée lors d'un incrément précédent est regroupée dans la même suite de spécification par la technique de structuration. De même pour les constructeurs de sortes. Toutes les informations constitutives d'un trait sont associées à la même unité. En ce qui concerne le partage, un nouveau composant peut être défini en cas de besoin, comme nous l'avons fait pour les types de base. La spécification la plus évoluée d'un trait est directement identifiable dans la structure grâce aux numéros de version.

3.1.3. Prise en compte du processus au niveau du langage

Le principal point que nous voulions traiter dans notre méthode était de permettre au spécifieur de suivre une approche incrémentale lors de l'écriture de la spécification et d'identifier les spécifications associées à des unités architecturales lors de la phase de découpage. La proposition d'une structure préliminaire au début de la phase d'écriture, puis d'une technique de structuration (et de dénomination des spécifications) ont permis de différencier dans la structure ce qui correspond à l'évolution d'une spécification inachevée de ce qui correspond à une nouvelle spécification. Cette différenciation exprime la vision du spécifieur sur les éléments de spécification.

Néanmoins, il nous semble que la structure de la spécification pourrait être encore simplifiée en ne laissant apparaître qu'un seul nom de spécification pour chaque trait au lieu de toutes les versions successives. Puisque la formalisation d'un incrément peut être séparée entre plusieurs spécifications nommées, certains ne contiennent que peu d'information. Ces «petites» spécifications compliquent la structure de la spécification. De plus, donner le même statut aux différentes versions d'une spécification qu'à la spécification de départ entraîne des redondances dans la structure. Par exemple, lorsque nous ajoutons les prédicats $Isprefix$ et $Ispostfix$ à la spécification des entrées, puis que nous ajoutons la définition du prédicat $Longest\_Prefix\_Match$ dans I/O_RELATIONSHIPS_2, nous devons explicitement étendre INPUTS_2, bien que MATCH étendait déjà INPUTS.

Ces redondances nous interrogent sur le statut des itérations dans CASL. Dans la technique d'écriture, une itération est identifiée par le nom de spécification utilisé. Lorsque la spécification d'un trait évolue, les nouveaux éléments sont localisés dans une spécification portant le nom du trait, indexé par le numéro de version. Cette solution se rapproche des constructions du langage PLUSS [Bid89] pour identifier une itération. Dans ce langage, une itération est identifiée par le mot clé du début de sa déclaration : **sketch**, **draft**, ou **spec**, selon le degré d'avancement de la spécification de ce trait (un trait n'est associé alors qu'aux sortes). La codification de l'utilisation d'un mot clé est très stricte. Essentiellement, ces critères concernent la connaissance des constructeurs de la sorte concernée par le trait. Ces critères permettent de donner une sémantique différente selon le degré d'avancement jusqu'à obtenir une spécification suffisamment complète pour être développée indépendamment des autres. Les niveaux de spécification sont donc mélangés dans ce langage entre une spécification normale et une spécification architecturale. Cette confusion a pour conséquence d'obliger le spécifieur a ne développer la spécification que d'un trait à la fois. Or, comme nous l'avons vu dans le traitement de l'exemple de l'analyse lexicale, les différents traits sont développés en parallèle, causant par là même les redondances dans la structure.

Pour éliminer ces redondances, nous voudrions ne garder dans la structure que les relations entre les traits et pas les relations intermédiaires entre les différentes versions. Nous avons donc besoin de deux choses : disposer d'une construction pour exprimer une itération, et que les nouvelles versions d'un trait soient automatiquement prises en compte par les autres spécifications qui utilisent la spécification de ce trait. La solution la plus simple serait de pouvoir utiliser le même nom de spécification pour décrire plusieurs versions de la spécification du même trait. Pour identifier ces différentes versions, nous proposons de définir une construction du langage CASL permettant de redéfinir une spécification nommée.

La redéfinition d'un nom de spécification correspond à l'idée de ne vouloir garder qu'un seul nom pour exprimer les différentes versions. Ensuite, une propagation de cette redéfinition dans la sémantique des autres spécifications est nécessaire afin de ne garder que les relations entre traits. La redéfinition d'une spécification permet d'identifier dans le processus de développement l'évolution itérative d'une spécification inachevée, tout en gardant une structure de la spécification claire dans l'objectif d'un découpage architectural. Cette construction est souvent utilisée pour implémenter l'extensibilité ou la modifiabilité, par exemple dans un langage de programmation [Dam98, Mor97]. Elle correspond à une notion très faible de liaison dynamique.

Application à l'exemple de l'analyse lexicale

Nous ne reprenons pas tout le traitement de l'exemple, mais nous mettons en valeur les apports d'un tel mécanisme de redéfinition. Dans le traitement de l'exemple, la définition des prédicats $Isprefix$ et $Ispostfix$ peut se placer dans une redéfinition de la spécification des entrées. Ceci se noterait alors :

spec INPUTS $\stackrel{\text{re}}{=}$ INPUTS **then** ...

Ainsi, la spécification des entrées est redéfinie comme une extension de la spécification de la version précédente par les nouvelles définitions. Si toutes les spécifications ont été mises à jour, alors la définition du prédicat $Longest\_Prefix\_Match$ dans I/O_RELATIONSHIPS ne nécessite pas d'introduire une redondance dans la structure. Il suffit d'étendre l'ancienne spécification par les nouvelles informations, comme suit :
spec I/O_RELATIONSHIPS $\stackrel{re}{=}$ I/O_RELATIONSHIPS **then** ...

En effet, l'ancienne version de I/O_RELATIONSHIPS contiendrait alors la définition des prédicats $Isprefix$ et $Ispostfix$ de par la mise à jour de la sémantique. Cette construction permet donc de simplifier la structure de la spécification en traitant les redondances par la mise à jour de la sémantique des spécifications qui utilisaient la spécification redéfinie.

Nous allons définir dans la suite de ce chapitre la construction de redéfinition d'une spécification nommée. Pour cela, il faut préciser les conditions de déclaration d'une spécification nommée dans une spécification CASL, puis étudier le mécanisme de référence à un nom de spécification. Nous introduisons dans la section suivante les notions de la sémantique de CASL formalisant ces opérations.

3.2. Sémantique formelle de CASL

La sémantique du langage CASL est donnée en trois parties dans [CoF01]. La première partie décrit la sémantique des spécifications de base, c'est à dire des déclarations de sortes, des symboles de fonctions ou prédicats, ainsi que des axiomes contraignant l'interprétation de ces symboles. La deuxième partie présente la sémantique des spécifications structurées, indépendamment de la première partie. La troisième partie enfin décrit la sémantique formelle des spécifications architecturales, que nous avons exposé par ailleurs au chapitre 2.

L'objectif de cette section est de fournir une introduction à la sémantique des spécifications structurées. Ce chapitre précise certains points difficiles d'accès de [CoF01]. Ces rappels nous permettront de définir dans la section suivante la sémantique d'une redéfinition.

3.2.1. Généralités

Dans CASL, une spécification bien formée dénote une signature et une classe de modèles. Tous les modèles validant les axiomes qui contraignent les symboles déclarés dans la signature, suivant ainsi une approche non restrictive (*loose* en anglais). Cette signature et cette classe de modèles sont définies inductivement sur la syntaxe abstraite d'une spécification. La sémantique est présentée sous formes de règles définissant une relation entre un contexte, une spécification et un objet sémantique, qui sont de deux sortes :

– les règles statiques vérifient la bonne formation des phrases de la syntaxe abstraite et construisent un objet syntaxique en résultat. Par exemple, pour une spécification de base, la sémantique statique produit un enrichissement du contexte contenant les sortes, les symboles de fonction et de prédicat appartenant à la spécification. Les règles statiques sont de la forme $contexte \vdash phrase \rhd resultat$.

– les règles de modèles sont appliquées uniquement aux phrases bien formées selon la sémantique statique et produisent une classe de modèles au dessus de la signature générée par la sémantique statique. Les prémisses des règles statiques sont donc systématiquement incorporés dans les règles de modèles. Ces règles sont de la forme $contexte \vdash phrase \Rightarrow resultat$.

On peut distinguer la sémantique des déclarations de spécification de celle des expressions de spécification quant à la forme du contexte utilisé. Pour une déclaration de spécification nommée de type **spec** $SN = SP$, le contexte est constitué de la liste des sémantiques des spécifications nommées précédentes (auxquelles SP peut faire référence), et le résultat est une extension de cette liste par la sémantique de SN. On appelle *environnement global* cette liste, et on la note Γ. Pour une expression de spécification SP, on peut encore distinguer deux sous-cas : l'expression de spécification est une spécification structurée, dans ce cas le contexte est composé de la liste précédente et de la partie de l'expression déjà évaluée, le résultat est une extension de cette partie ; ou bien l'expression de spécification est une spécification de base, et dans ce cas le contexte n'est constitué que de la partie déjà évaluée. Cette partie de l'expression de spécification déjà évaluée est appelée *environnement local*. La sémantique ainsi définie est compositionnelle : la sémantique d'une spécification est entièrement déterminée par la sémantique de ses parties.

Remarques

1. Notez bien que le résultat de la sémantique statique dans le cas d'une expression de spécification n'est pas une signature, mais une extension de signature. En effet, imaginez le texte d'une spécification. L'application des règles à cette spécification peut se voir comme une ligne descendant le long du texte. À un moment donné, tout ce qui est au-dessus de cette ligne est déjà évalué, et ce qui est en-dessous reste à évaluer. La partie au-dessus dénote une signature correspondant au contexte dans lequel la partie en-dessous est évaluée. Cette partie en-dessous ne dénote donc pas une signature complète, mais une extension de l'environnement local. La signature globale de la spécification est l'union de la signature du contexte et de l'extension de signature.
2. Une classe de modèles est toujours définie sur une signature, non pas une extension de signature. Dans la métaphore ci-dessus, seule la partie au-dessus dénote une classe de modèles.
3. La visibilité de la déclaration de symboles est linéaire, cela signifie qu'on ne peut pas utiliser un symbole qui sera déclaré ultérieurement, hormis le cas de définitions mutuellement récursives. Le contexte grossit donc en suivant le déroulement de l'évaluation de la spécification.

Notations

Afin de bien différencier la syntaxe abstraite d'une spécification d'avec la spécification elle-même, nous noterons la syntaxe abstraite en `polices typographiques`. Par exemple, la déclaration d'une opération que l'on notait **op** $f : s \to s$, se notera dans la syntaxe abstraite `op f :s -> s`.

3.2.2. Syntaxe abstraite

La syntaxe abstraite est complètement donnée dans l'annexe A. Structurer une spécification améliore sa lisibilité, mais surtout permet d'en réutiliser certaines parties. Une partie de spécification peut donc être nommée de telle sorte que sa réutilisation consiste à faire une référence à ce nom. La spécification globale se présente alors comme une séquence de déclarations de spécifications nommées, définissant ainsi une *bibliothèque* de spécifications. Les règles de la sémantique des spécifications structurées ont donc comme point de départ une bibliothèque, suivant la syntaxe abstraite :

```
LIBRARY ::= SPEC-DEFN*
```

Dans CASL, toute spécification est considérée comme générique. Une spécification non générique est considérée comme un cas particulier de spécification générique n'ayant aucun paramètre. Une déclaration de spécification nommée suit donc la syntaxe abstraite :

```
SPEC-DEFN ::=  spec-defn SPEC-NAME GENERICITY SPEC
GENERICITY ::=  genericity PARAMS IMPORTS
PARAMS ::=  params SPEC*
IMPORTS ::=  imports SPEC*
```

Il est nécessaire de différencier imports et paramètres dans la partie générique. Les paramètres indiquent les parties de la spécification qui sont supposées varier d'une référence à l'autre. Les imports, au contraire sont fixes, mais identifient les parties de la spécification qui sont susceptibles de contenir des symboles communs avec un éventuel argument. Si ces parties communes ne sont pas identifiées, il se peut que l'instanciation ne puisse pas être construite lorsqu'un argument partage plus de symboles avec le corps que ceux contenus dans le paramètre. Cette situation provient de la sémantique de l'instanciation, que nous verrons plus loin en détail.

Exemple 3.1
Si la spécification générique des listes est structurée comme suit :

spec LIST[**sort** *Elem*] = NAT **then**
  generated type $List[Elem] ::= Nil \mid cons(Elem; List[Elem]) \ldots$
  ops $\ldots;$
    $Length : List[Elem] \to Nat$
  $\forall l : List[Elem] \bullet Length(Nil) = 0;$
  $\ldots$

Alors l'instanciation LIST[NAT] est mal formée, puisque l'argument NAT partage la constante 0 avec le corps de LIST[**sort** *Elem*], qui n'est pas contenue dans le paramètre **sort** *Elem*.

Il faudrait plutôt écrire :

spec LIST[**sort** *Elem*] **given** NAT = ...

□

La spécification SPEC du corps d'une spécification générique, ou des imports ou paramètres, est une spécification structurée, construite à partir de la syntaxe abstraite :

```
SPEC ::= BASIC-SPEC | translation SPEC RENAMING | reduction SPEC REVEAL
       | restriction SPEC HIDE | union SPEC+ | extension SPEC+
       | free-extension SPEC | spec-inst SPEC FIT-ARG*
```

BASIC-SPEC est une spécification de base. Des symboles d'une spécification peuvent être renommés, certains peuvent être révélés (et possiblement renommés), ou encore masqués. Des spécifications peuvent être réunies, étendues, possiblement par une extension libre. Une spécification peut être référencée par son nom, ce qui correspond à une instanciation. Nous ne rentrons pas dans les détails de la syntaxe abstraite des renommages ou des restrictions.

3.2.3. Bibliothèques

Une spécification nommée est implicitement close, c'est à dire qu'elle doit être évaluée dans un environnement local vide. En revanche, le contexte doit contenir la sémantique des spécifications nommées déjà évaluées, afin de pouvoir gérer les références. Le contexte d'évaluation d'une spécification nommée est constitué d'un environnement global. L'environnement global stocke les associations entre les noms de spécifications et leur signature et leur classe de modèles. La sémantique d'une bibliothèque est définie par l'environnement global contenant toutes les associations.

Plus précisément, dans CASL, la sémantique d'une bibliothèque est donnée par deux environnements compatibles : un environnement statique Γ_s et un environnement de modèles Γ_m. Comme toutes les spécifications sont supposées génériques, la sémantique statique associée à un nom de spécification n'est pas seulement une signature, mais une signature générique consistant en l'union de la signature des spécifications importées Σ_I, une extension Δ_i de cette signature pour chacun des paramètres, et une extension Δ_B de l'union : $\Sigma_I \cup (\Delta_1 \cup \ldots \cup \Delta_n)$.[1] L'environnement statique est donc une liste d'associations entre des noms de spécification et leur signature générique. De même, la sémantique de modèles d'une spécification générique consiste en une classe de modèles pour les imports, d'une liste de classe de modèles pour les paramètres, et d'une classe de modèles pour le corps. L'environnement de modèles est une liste d'associations entre des noms de spécification et leur classe de modèles.[2]

Enfin, la compatibilité entre Γ_s et Γ_m signifie que :

– la longueur de ces environnements est identique, notée l ;

[1] L'union de signatures, dépassant notre propos ici, sera définie ultérieurement.

[2] Dans [CoF01], les environnements contiennent de plus des association pour les *views* et les spécifications architecturales, que nous ne prenons pas en compte ici.

- pour tout $i \in [1..l]$, $\Gamma_s[i]$ et $\Gamma_m[i]$ définissent toutes les deux une association pour le même nom de spécification ;
- pour chaque association $\texttt{SN} \rightarrow (\Sigma_I, (\Delta_1 \ldots \Delta_n), \Delta_B)$ et $\texttt{SN} \rightarrow (\mathcal{M}_I, (\mathcal{M}_1 \ldots \mathcal{M}_{n'}), \mathcal{M}_B)$, alors :
  - n = n' ;
  - $\mathcal{M}_I$ est une classe de Σ_I-modèles ;
  - pour tout i, chaque $\mathcal{M}_i$ est une classe de $\Sigma_I \cup \Delta_i$- modèles qui étendent un modèle de $\mathcal{M}_I$;
  - $\mathcal{M}_B$ est une classe de $\Sigma_I \cup (\Delta_1 \cup \ldots \cup \Delta_n) \cup \Delta_B$-modèles, chacun étendant un modèle de $\mathcal{M}_i$ pour tout i.

La sémantique d'une bibliothèque est donnée par les règles :

$$\frac{\begin{array}{c}\emptyset \vdash \texttt{SPEC-DEFN}_1 \rhd \Gamma_{s,1} \\ \Gamma_{s,1} \vdash \texttt{SPEC-DEFN}_2 \rhd \Gamma_{s,2} \\ \ldots \\ \Gamma_{s,n-1} \vdash \texttt{SPEC-DEFN}_n \rhd \Gamma_s\end{array}}{\vdash \texttt{SPEC-DEFN}_1 \ldots \texttt{SPEC-DEFN}_n \rhd \Gamma_s} \quad (1)$$

$$\frac{\begin{array}{c}\emptyset \vdash \texttt{SPEC-DEFN}_1 \Rightarrow \Gamma_{m,1} \\ \Gamma_{s,1}, \Gamma_{m,1} \vdash \texttt{SPEC-DEFN}_2 \Rightarrow \Gamma_{m,2} \\ \ldots \\ \Gamma_{s,n-1}, \Gamma_{m,n-1} \vdash \texttt{SPEC-DEFN}_n \Rightarrow \Gamma_m\end{array}}{\vdash \texttt{SPEC-DEFN}_1 \ldots \texttt{SPEC-DEFN}_n \Rightarrow \Gamma_m} \quad (2)$$

Une bibliothèque est évaluée dans un contexte vide, puisque c'est l'objet principal à partir duquel la sémantique est construite. En revanche, le contexte dans lequel les déclarations de spécification sont évaluées est constitué de l'environnement global statique pour les règles statiques, et des environnements statiques et de modèles pour les règles de modèles. On a besoin de l'environnement statique pour les règles de modèles car celles-ci ne sont définies que pour des spécifications bien formées selon les règles statiques. Les environnements croissent donc au fur et à mesure de l'évaluation de la bibliothèque, qui vérifie donc une propriété de visibilité linéaire.

3.2.4. Déclaration

Pour toute déclaration de spécification nommée rencontrée lors de l'évaluation d'une bibliothèque, une association est ajoutée pour le nom défini dans chacun des deux environnements, si ce nom n'était pas déjà utilisé. Les règles pour la catégorie syntaxique `SPEC-DEFN` sont :

$$\frac{\begin{array}{c}\texttt{SPEC-NAME} \notin Dom(\Gamma_s) \\ \Gamma_s \vdash \texttt{GENERICITY} \rhd (\Sigma_I, (\Delta_1 \ldots \Delta_n)) \\ \Sigma_I \cup (\Delta_1 \cup \ldots \cup \Delta_n), \Gamma_s \vdash \texttt{SPEC} \rhd \Delta_B \\ \Gamma'_s = \Gamma_s :: \{\texttt{SPEC-NAME} \rightarrow (\Sigma_I, (\Delta_1 \ldots \Delta_n), \Delta_B)\}\end{array}}{\Gamma_s \vdash \texttt{spec-defn SPEC-NAME GENERICITY SPEC} \rhd \Gamma'_s} \quad (3)$$

$$\frac{\begin{array}{c}\Gamma_s \vdash \texttt{GENERICITY} \rhd (\Sigma_I, (\Delta_1 \ldots \Delta_n)) \\ \Gamma_s, \Gamma_m \vdash \texttt{GENERICITY} \Rightarrow (\mathcal{M}_I, (\mathcal{M}_1 \ldots \mathcal{M}_n)) \\ \mathcal{M}_P = \{M \in \mathbf{Mod}(\Sigma_I \cup (\Delta_1 \cup \ldots \cup \Delta_n)) \mid M|_{\Sigma_I} \in \mathcal{M}_I, M|_{\Sigma_I \cup \Delta_i} \in \mathcal{M}_i, 1 \leq i \leq n\} \\ \Sigma_I \cup (\Delta_1 \cup \ldots \cup \Delta_n), \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}_B \\ \Gamma'_m = \Gamma_m :: \{\texttt{SPEC-NAME} \rightarrow (\mathcal{M}_I, (\mathcal{M}_1 \ldots \mathcal{M}_n), \mathcal{M}_B)\}\end{array}}{\Gamma_s, \Gamma_m \vdash \texttt{spec-defn SPEC-NAME GENERICITY SPEC} \Rightarrow \Gamma'_m} \quad (4)$$

Une déclaration de spécification nommée est évaluée dans un contexte contenant l'environnement global, et un environnement local vide, qui est omis. Les paramètres et imports (qui constituent la catégorie syntaxique `GENERICITY`) sont également évalués dans ce contexte, puisque la spécification des imports est une spécification close, et que les paramètres définissent des extensions de cette signature (qui sera donc ajoutée dans l'environnement local lors de l'évaluation des paramètres). En revanche, le corps de la spécification est évalué dans un contexte contenant deux sortes d'informations : les informations provenant de l'environnement global puisqu'une référence à un nom de spécification peut être fait dans le corps de la spécification, et les informations provenant de l'évaluation des paramètres et imports puisque la signature du corps est définie comme une extension de la signature résultant de l'union de la signature des imports et des paramètres.

3.2.5. Référence

Comme toute spécification est supposée générique, la référence à un nom de spécification dans une spécification structurée correspond à une instanciation dans le cas général. Les règles correspondantes seront données ultérieurement. Nous ne nous occupons pour l'instant que du cas particulier des spécifications génériques à aucun paramètre, ni import. La référence à un nom de spécification est correcte si ce nom est présent dans l'environnement global et construit une extension de l'environnement local par la signature associée à cette référence pour la règle statique, et une classe de modèles au dessus de cette extension pour la règle de modèles. La notation $\iota_{\Sigma \subseteq \Sigma'}$ signifie que l'on prend l'unique inclusion de signature de Σ dans Σ'.

$$\frac{(\texttt{SPEC-NAME} \mapsto (\emptyset, (), \Sigma_B)) \in \Gamma_s}{\Sigma, \Gamma_s \vdash \texttt{spec-inst SPEC-NAME} \rhd \iota_{\Sigma \subseteq (\Sigma \cup \Sigma_B)}} \quad (5)$$

$$\frac{\begin{array}{c}(\texttt{SPEC-NAME} \mapsto (\emptyset, (), \Sigma_B)) \in \Gamma_s \\ (\texttt{SPEC-NAME} \mapsto (\mathcal{M}_\perp, (), \mathcal{M}_B)) \in \Gamma_m \\ \mathcal{M}' = \{M \in \mathbf{Mod}[\Sigma \cup \Sigma_B] \mid M|_\Sigma \in \mathcal{M},\ M|_{\Sigma_B} \in \mathcal{M}_B\}\end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{spec-inst}\ \texttt{SPEC-NAME} \Rightarrow \mathcal{M}'} \quad (6)$$

3.3. Redéfinition de spécifications nommées

Afin de motiver les choix d'implémentation de la redéfinition de spécifications nommées dans Casl, nous illustrons sur l'exemple d'un mini(voir micro)-langage de programmation les différentes possibilités. Ce mini-langage peut se voir comme une abstraction de Casl dans laquelle nous n'aurions conservé que le principe de gestion des noms de spécification. Cette abstraction permet de dégager des principes généraux que nous pourrons transférer ensuite dans la sémantique de Casl.

3.3.1. Une analogie

Considérons le mini-langage suivant : un programme p est constitué d'une suite d'affectations entre un nom de variable pris dans un ensemble Var et une expression e pouvant être une référence à un nom de variable, un entier naturel, ou une addition de deux sous-expressions. La bonne formation statique d'un programme est évaluée à partir de la grammaire :

```
P ::= AFF*
AFF ::=  VAR:=EXPR
VAR ::= x | y | ...
EXPR ::= VAR | NAT | e1 + e2
NAT ::= 0 | 1 | ...
```

La sémantique d'un programme est déterminée par la valeur des variables affectées dans ce programme une fois tout le texte du programme évalué (sémantique naturelle). Pour construire la sémantique d'un programme, nous utiliserons une notion d'environnement dans lequel sont stockées des informations sur les variables.

Nous allons présenter plusieurs versions de la sémantique, partant du langage le plus rudimentaire, passant par un langage avec redéfinitions, et aboutissant enfin à un langage avec redéfinitions et propagation. Cette étude sur ce langage sera transposée ensuite dans le cas de la sémantique de Casl, en remplaçant la notion de programme par celle de bibliothèque, une affectation par une déclaration de spécification nommée, une variable par un nom de spécification, une expression par une expression de spécification, les entiers par les signatures et les classes de modèles sur ces signatures.

Affectations simples

Considérons le cas le plus restrictif : la suite des affectations d'un programme ne doit pas concerner plus d'une fois le même nom de variable. Chaque variable n'est donc affectée qu'une seule fois au cours d'un programme.

La sémantique d'un tel langage utilise une notion d'environnement dans lequel les noms de variables sont associées à des valeurs entières. Un programme est évalué dans un environnement vide. Chaque affectation du programme est évaluée dans un environnement contenant les associations pour les variables déjà évaluées. On teste si la variable concernée par l'affectation courante n'est pas déjà présente dans l'environnement (auquel cas le programme est mal formé), puis on calcule la valeur de l'expression dans l'environnement courant en remplaçant les références à des noms de variables par leur valeur dans l'environnement courant. On impose également une visibilité linéaire : une référence à un nom de variable non présente dans l'environnement courant rend le programme mal formé. L'environnement courant est enfin augmenté par une nouvelle association entre ce nom et cette valeur. La sémantique du programme correspond à l'environnement final.

Redéfinitions

Une première évolution consiste à autoriser les affectations multiples d'une variable au cours d'un programme, sans propagation. Dans cette première extension de la sémantique du langage, le programme suivant est bien formé :

```
x:=0 ; y:=x+1 ; x:=1
```

Pour implémenter cette extension, il suffit de modifier la sémantique des affectations. Chaque affectation est toujours évaluée dans un environnement contenant les associations des variables déjà évaluées. L'expression est calculée de la même manière que précédemment. La seule différence consiste à supprimer le test de présence dans l'environnement de la variable concernée, puis à définir une fonction de substitution d'une association dans un environnement. Une fois l'expression évaluée, l'association entre le nom de variable et cette expression est substituée dans l'environnement courant. Cette fonction de substitution est définie comme suit : elle parcourt l'environnement, si elle ne rencontre pas d'association pour le nom de variable alors elle augmente l'environnement par cette nouvelle association, sinon elle remplace l'ancienne valeur par la nouvelle.

Notons que dans cette sémantique des affectations de la forme `x :=x+1` sont correctes si l'environnement dans lequel est évaluée cette affectation contient déjà une association pour `x`. Des programmes présentant une certaine forme de circularité sont corrects, comme par exemple :

```
x:=0 ; y:=x+1 ; x:=y+1
```

Dans ce programme, `y` dépend de `x` qui dépend de `y` dans sa seconde affectation. Néanmoins, la présence d'une association entre `x` et une valeur permet de déplier cette circularité apparente. La sémantique de ce programme est $\{x \rightarrow 2; y \rightarrow 1\}$.

Redéfinitions et propagation

En l'absence de propagation, la référence à un nom de variable se fait dans l'environnement de définition, pas dans l'environnement d'appel. Par exemple, si on rajoute au programme précédent une affectation `z :=y+x`, la valeur de `y` est celle calculée lorsque `x` valait 0, pas 2. Il existe cependant des situations dans lesquelles on veut que la valeur de `y` soit affectée par les modifications de la valeur de `x`, grâce à un mécanisme de propagation.

Deux implémentations de ce mécanisme peuvent être proposées :

1. La première consiste à effectuer une transformation syntaxique sur le programme afin d'éliminer les redéfinitions. Cette transformation se fait au moment de l'analyse syntaxique du programme, c'est-à-dire lors de la construction de l'arbre de syntaxe abstraite du programme. Les affectations sont triées selon les variables concernées par ces affectations : toutes celles affectant une même variable sont rassemblées avant de passer à celles concernant une autre variable. L'ordre d'apparition de ces blocs correspond à l'ordre des premières affectations de chaque variable. Pour chaque bloc d'affectations concernant une même variable, des variables fraîches (c'est à dire qui n'ont pas été encore utilisées dans le programme) sont utilisées à la place de la variable redéfinie afin de distinguer les différentes occurrences de cette variable. Les références à cette variable dans les expressions sont remplacées par la dernière variable fraîche introduite au moment où cette expression est rencontrée.

   Par exemple, le programme suivant :

   ```
   x:=0 ; y:=x+1 ; x:=1
   ```

   est transformé en :

   ```
   x:=0 ; x1:= 1; y:=x1+1
   ```

   Ce dernier programme est un programme dans lequel ne se trouvent que des affectations simples que l'on peut traiter comme nous l'avons fait plus haut.

   Cette implémentation permet de différencier deux formes de circularité dans les références. Une référence peut être directe ou indirecte : directe lorsque la variable redéfinie apparaît dans l'expression, indirecte lorsque c'est une tierce variable, faisant référence à la variable redéfinie, qui apparaît dans l'expression. Le programme :

   ```
   x:=0 ; y:=x+1 ; x:=y+1
   ```

   comporte une circularité indirecte dans la redéfinition de `x`. La transformation syntaxique de ce programme donne :

   ```
   x:=0 ; x1:= y+1; y:=x1+1 ;
   ```

   qui est incorrecte selon la sémantique des affectations simples.

   En revanche, le programme :

   ```
   x:=0 ; y:=x+1 ; x:=x+1
   ```

   est lui transformé en :

```
x:=0 ; x1:=x+1; y:=x1+1 ;
```

qui est correct.

Cette distinction entre ces deux types de circularité dans les références correspond à notre intuition sur le rôle joué par les variables dans un programme. Les variables doivent pouvoir être classées selon un ordre partiel afin de pouvoir les évaluer séquentiellement, ce qui est contredit par les références indirectes. Une référence directe correspond également à une utilisation naturelle de la redéfinition, par exemple lorsqu'on incrémente la valeur d'une variable.

Néanmoins, ce traitement syntaxique de la propagation n'est pas convaincant. L'utilisation des variables fraîches impose de revenir à l'analyse syntaxique dès que le programme est étendu par une nouvelle affectation. En effet, la nouvelle variable ne doit pas correspondre à une variable fraîche utilisée précédemment. Si c'est le cas, celle ci ne doit pas être confondue avec une redéfinition de cette variable fraîche. Il faudrait pouvoir distinguer les variables originales des variables fraîches dans le programme modifié. Or toutes les variables ont le même statut dans la sémantique des associations simples. De plus, lors de l'ajout d'une redéfinition dans le programme, seules les associations des variables faisant référence à la variable redéfinie doivent être mises à jour. Un traitement sémantique de la propagation serait plus approprié pour prendre en compte ce point.

2. La seconde implémentation se situe donc plus au niveau sémantique. Plutôt que de transformer le texte du programme pour supprimer les redéfinitions, une construction syntaxique particulière est définie afin de gérer les redéfinitions. L'utilisateur signale donc lorsqu'il définit ou redéfinit une variable. La sémantique des affectations est celle utilisée pour les affectations simples, tandis que celle des redéfinitions est associée à un traitement sémantique particulier : le nom de la variable utilisée doit déjà figurer dans l'environnement ; une fois l'expression évaluée, la nouvelle valeur est substituée à l'ancienne puis propagée dans l'environnement pour toutes les associations comportant une référence à la variable redéfinie. L'identification de ces associations se fait sur l'arbre de syntaxe abstraite du programme. Les références peuvent être directes ou indirectes : la liste de toutes les variables mises à jour doit être disponible lors de la propagation dans l'environnement.

   Dans cette implémentation, tous les programmes sont corrects. Le programme suivant, dans lequel on note `:='` une redéfinition :

   ```
   x:=0 ; y:=x+1 ; x:=' x+y
   ```

   peut être évaluée en l'environnement $\Gamma = \{x \rightarrow 1, y \rightarrow 2\}$. Mais cette circularité dans les références ne devrait pas être autorisée, puisque cela entraîne une confusion sur le rôle joué par chaque variable. Une condition supplémentaire peut alors être ajoutée à la règle sémantique des redéfinitions, interdisant les références indirectes à la variable en train d'être redéfinie. Les références directes doivent, elles, être autorisées, car elles correspondent à un cas naturel d'utilisation du mécanisme de redéfinition.

Renversement de l'analogie

Cette analogie nous a permis d'étudier le mécanisme de gestion des variables dans la sémantique de CASL sans s'occuper de détails techniques ayant trait aux signatures et classes de modèles. Pour renverser l'analogie, il suffit de remplacer la notion de programme par celle de bibliothèque, affectation par déclaration de spécification nommée, variable par nom de spécification, expression par expression de spécification, l'addition par un opérateur de structuration, les entiers naturels par les symboles utilisés pour écrire une spécification ou les signatures. Dans CASL, la sémantique ne permet que des affectations simples puisque, selon la règle (3), un nom de spécification d'une déclaration ne peut pas être déjà utilisé.

Pour implémenter la redéfinition de spécifications nommées dans CASL selon la deuxième implémentation, nous devons définir une nouvelle catégorie syntaxique permettant de distinguer dans une bibliothèque une déclaration simple d'une redéfinition, puis définir les deux règles sémantiques correspondantes, en particulier exprimer la condition de bonne formation concernant la circularité, ainsi que les deux fonctions de substitution et de propagation.

3.3.2. Extension de CASL

Le mécanisme de redéfinition dans CASL permet de compléter la spécification d'un trait du système au cours de la phase d'écriture. En suivant l'approche incrémentale, toutes les informations nécessaires à la spécification d'un trait ne sont souvent pas disponibles en une fois, celle-ci est donc construite petit à petit. Lors d'un incrément, certaines informations peuvent concerner un trait dont la spécification est contenu dans une spécification nommée A. Celle-ci peut être redéfinie afin d'incorporer les nouvelles informations. La spécification d'un trait, identifié lors de la phase d'analyse ou au cours du processus, est alors regroupée dans la même spécification nommée. Lors de la phase de découpage modulaire, il sera simple d'associer une spécification à une unité pouvant être développée indépendamment.

Pour distinguer clairement une déclaration simple d'une redéfinition, la syntaxe abstraite du langage est modifiée comme suit :

```
LIBRARY ::= SPEC-DEFN* | SPEC-REDEFN*
SPEC-REDEFN ::= spec-redefn SPEC-NAME GENERICITY SPEC
```

Dans la syntaxe concrète (dans le texte), une redéfinition est représentée par :
spec $\mathrm{A} \overset{\mathrm{re}}{=} \mathrm{SP}$
et `spec A =re= SP` pour la syntaxe concrète de l'outil de vérification syntaxique.

Comme pour une déclaration (règles (3) et (4)), la sémantique d'une redéfinition est donnée par une règle statique et une règle de modèles indiquant comment les environnements Γ_s et Γ_m sont modifiés lorsqu'une redéfinition est rencontrée dans l'évaluation d'une bibliothèque. Ces règles ne doivent pas briser la propriété de visibilité linéaire des environnements et empêcher les circularités indirectes dans les références. Pour cela, le contexte dans lequel est évaluée une redéfinition doit être précisément défini. Étant donné une redéfinition `spec A =re= SP`, trois choses doivent être précisées : vérifier que A reste bien formée après sa redéfinition, indiquer comment sa sémantique est mise à jour, et enfin propager la nouvelle valeur de la sémantique dans les deux environnements. Comme les actions à effectuer sont identiques pour les deux environnements, nous noterons dans la suite Γ l'un d'entre eux.

Définitions préliminaires

Pour exprimer la condition de bonne formation d'une redéfinition, nous aurons besoin des définitions suivantes :

Définition 3.2
Étant donné un nom de spécification `SN` *et son expression de spécification associée,* `SPEC`*, on note* Ref(`SPEC`) *l'ensemble des références présentes dans* `SPEC`*.*
$Ref^+_{\mathcal{L}}$(`SPEC`) *dénote la clôture transitive de* Ref(`SPEC`) *dans une bibliothèque de syntaxe abstraite* $\mathcal{L}$*.*

Puisque les environnements sont structurés comme des listes, il est possible de déterminer la partie située avant une association et celle après :

Définition 3.3
Étant donné un nom de spécification `SN`*,* Γ *un environnement global,* $\Gamma(until$ `SN`$)$ *dénote le préfixe de* Γ *jusqu'à l'association pour* `SN` *incluse.*
De même, $\Gamma(from$ `SN`$)$ *dénote le postfixe de* Γ *à partir de l'association pour* `SN` *exclue.*

Bonne formation

Lorsqu'une spécification nommée est définie avec la construction usuelle de déclaration (règle (3)), la bonne formation de cette spécification signifie que le nom de la spécification n'est pas déjà utilisé, puis qu'il est possible de déterminer sa signature d'après les règles statiques. En particulier, la règle (5) vérifie que les références apparaissant dans l'expression de spécification appartiennent bien à l'environnement, ce qui interdit la présence de toute forme de circularité dans les références.

Dans le cas d'une redéfinition `spec A =re= SP`, la condition de bonne formation doit autoriser les circularités directes dans la structure de l'expression de spécification. En effet, le mécanisme de redéfinition permet de compléter la spécification d'un trait. Cet ajout consiste la plupart du temps à étendre la spécification originale par de nouvelles informations, l'expression de spécification `SP` est souvent proche de `A then ...`. En général, `SP` contient donc une référence directe à `A`, soit : `A` $\in Ref$(`SP`). En revanche, la condition de bonne formation doit exclure les circularités indirectes. En effet, d'un point de vue méthodologique, une spécification nommée contient les informations relatives à un trait du système. La présence de circularités indirectes rendrait ce principe confus, puisqu'alors toutes les spécifications nommées contenues dans la boucle de références d'une circularité indirecte seraient concernées par le même trait. Une référence indirecte à `A` signifie que `A` $\in Ref^+_{\mathcal{L}}$(`SP`). Si `A` comporte des paramètres ou des imports, que nous rassemblons en une spécification `GEN`, toute forme de circularité doit être interdite dans leur structure. Pour éviter les références indirectes, une première solution est de poser la condition suivante à la bonne formation d'une redéfinition :

$$Ref^+_{\mathcal{L}}(\mathtt{SP}) \subseteq \Gamma(until\ \mathtt{A}) \wedge \mathtt{A} \notin Ref(\mathtt{GEN})$$

Cette condition exprime le fait que les références de SP et de GEN sont celles qui étaient disponibles au moment de la définition originale de A, ce qui préserve des références indirectes à A. En effet, A appartient à $\Gamma(until\ \mathtt{A})$, mais les noms de spécifications présents dans $\Gamma(until\ \mathtt{A})\backslash\mathtt{A}$ ne peuvent pas faire référence à A grâce à la propriété de visibilité linéaire des bibliothèques. Pour intégrer cette condition aux règles sémantiques, il suffit de choisir d'évaluer la signature de SP et de GEN dans l'environnement global $\Gamma(until\ \mathtt{A})$.

Exemple 3.4
Étant donnée la bibliothèque $\mathcal{L}$ suivante, dans laquelle les SP_i sont des expressions de spécifications quelconques qui ne contiennent aucune référence :
spec A = SP_A
spec B = A **then** SP_B
spec C = B **then** SP_C
spec D = SP_D
Alors pour la redéfinition A $\overset{\text{re}}{=}$ A **then** Δ_A avec Δ_A une spécification de base, A reste bien formée, tandis que ce ne serait pas le cas avec A $\overset{\text{re}}{=}$ A **then** C pour des problèmes de circularité indirecte (puisque A $\in Ref^+_{\mathcal{L}}(\mathrm{C})$), ni pour A $\overset{\text{re}}{=}$ A **then** D puisque dans ce cas la redéfinition contient une référence à un nom de spécification postérieur à A dans l'ordre des déclarations ($Ref^+_{\mathcal{L}}(\mathrm{A}\ \mathbf{then}\ \mathrm{D}) \cap \Gamma(from\ \mathrm{A}) = \{D\}$).

□

Néanmoins, cette condition est trop forte. Comme le montre le dernier cas de l'exemple ci-dessus, une étape de redéfinition dans le processus de spécification peut nécessiter la déclaration d'une spécification nommée intermédiaire (rôle de D dans notre exemple) qui sera ensuite incorporée à la redéfinition. Ce cas ne satisfait pas la condition puisqu'alors la référence à ce nom de spécification intermédiaire dans SP rend l'ensemble $Ref^+_{\mathcal{L}}(\mathtt{SP}) \cap \Gamma(from\ \mathtt{A})$ non vide. Une seconde formulation consiste à tester directement si les références de SP ne contiennent A que de manière directe, soit :

$$\mathtt{A} \notin Ref^+_{\mathcal{L}}(\mathtt{SP})\backslash Ref(\mathtt{SP}) \wedge \mathtt{A} \notin Ref(\mathtt{GEN})$$

Mise à jour

Une fois évaluée, la nouvelle sémantique de A est substituée à l'ancienne dans les deux environnements Γ_s et Γ_m. Comme il n'existe qu'une seule association par nom de spécification dans les environnements, il suffit de modifier la dernière association des environnements $\Gamma_s(until\ \mathtt{A})$ et $\Gamma_m(until\ \mathtt{A})$. Dans le premier, on la remplace par une association entre A et la nouvelle signature ; dans le second, par une association entre A et la nouvelle classe de modèles. Cette opération est définie par les opérations :

$$Subst\_\_in\_\_ : Association \times Environment \rightarrow Environment$$

Propagation

Une fois que la nouvelle sémantique de `A` est évaluée, il reste à la propager dans l'environnement. Une opération de propagation est définie sur les environnements, prenant en argument une liste de spécifications nommées qui ont déjà été modifiées, la bibliothèque (c'est-à-dire la liste des arbres de syntaxe abstraite correspondant aux déclarations de spécification), et l'environnement devant être modifié (soit de signatures, soit de modèles selon les cas) :

$$\begin{aligned}&Propage\_\_accordingto\_\_in\_\_ :\\ &\qquad List[SpecName] \times Library \times Environment \rightarrow Environment\end{aligned}$$

Cette fonction détermine quelles associations doivent être modifiées dans l'environnement passé en paramètre, à partir des références dans la syntaxe abstraite des déclarations de spécification contenues dans la bibliothèque. Cette fonction effectue une récurrence sur la longueur de la bibliothèque : pour chaque déclaration de spécification, on teste si une des références de cette déclaration est contenue dans la liste contenant tous les noms de spécification déjà modifiés (ce qui permet de gérer les références indirectes à `A`) : si oui, alors la signature et la classe de modèles de cette déclaration sont réévaluées puis mises à jour, et enfin ce nom est ajouté à la liste ; sinon, on passe à la déclaration suivante.

Un pseudo-code de ces opérations est donné plus loin.

Règles pour la sémantique

Les règles sémantiques pour la construction de redéfinition peuvent maintenant être définies. Dans le cas de la règle statique, l'évaluation de la signature pour une redéfinition d'une spécification nommée `SN` n'est définie que s'il existe déjà une association pour le même nom dans l'environnement statique. La condition de bonne formation d'une redéfinition, excluant les circularités indirectes, est prise en compte explicitement. Si $\Sigma_{\mathtt{SN}}$ dénote la signature générique de `SN` avant sa redéfinition (c'est à dire $(\Sigma_I, (\Delta_1 \cup \ldots \cup \Delta_n), \Delta_B)$), on note $\Sigma'_{\mathtt{SN}}$ sa signature après sa redéfinition.

La propagation de la nouvelle signature associée à `SN` est ensuite effectuée dans l'environnement $\Gamma_s(from\ \mathtt{SN})$. L'environnement final, Γ'_s, est donc la concaténation de $Subst(\mathtt{SN} \rightarrow \Sigma'_{\mathtt{SN}})\ in\ \Gamma_s(until\ \mathtt{SN})$ et de $Propage\{\mathtt{SN}\}\ accordingto\ \mathcal{L}\ in\ \Gamma_s(from\ \mathtt{SN})$. Le paramètre $\mathcal{L}$ est implicite dans cette règle, puis qu'il n'apparaît pas directement dans le résultat mais est l'objet sur lequel les règles agissent. On obtient donc la règle :

$$\frac{\begin{array}{c}\texttt{SPEC-NAME} \in Dom(\Gamma_s)\\ \texttt{SPEC-NAME} \notin Ref^+_{\mathcal{L}}(\texttt{SPEC})\backslash Ref(\texttt{SPEC})\\ \texttt{SPEC-NAME} \notin Ref(\texttt{GENERICITY})\\ \Gamma_s \vdash \texttt{GENERICITY} \rhd (\Sigma_I, (\Delta_1 \ldots \Delta_n))\\ \Sigma_I \cup (\Delta_1 \cup \ldots \cup \Delta_n), \Gamma_s \vdash \texttt{SPEC} \rhd \Delta_B\\ \Gamma'_s = Subst(\texttt{SPEC-NAME} \to (\Sigma_I, (\Delta_1 \ldots \Delta_n), \Delta_B))\ in\ \Gamma_s(until\ \texttt{SPEC-NAME})\\ :: Propage\{\texttt{SPEC-NAME}\}\ accordingto\ \mathcal{L}\ in\ \Gamma_s(from\ \texttt{SPEC-NAME})\end{array}}{\Gamma_s \vdash \texttt{spec-redefn SPEC-NAME GENERICITY SPEC} \rhd \Gamma'_s} \quad (7)$$

Pour la règle de modèles, une association pour le nom de spécification doit être présent. Si $\mathcal{M}_{\texttt{SN}}$ est la classe de modèles associée à SN avant sa redéfinition, alors $\mathcal{M}'_{\texttt{SN}}$ est la classe de tous les modèles au dessus de $\Sigma'_{\texttt{SN}}$ validant les contraintes imposées dans SPEC et dont les réduits sur $\Sigma_{\texttt{SN}}$ sont dans $\mathcal{M}_{\texttt{SN}}$. Puisque la classe de modèles n'est évaluée que pour des spécifications bien formées selon les règles statiques, les prémisses de la règle précédente sont ajoutés. On obtient la règle :

$$\frac{\begin{array}{c}\texttt{SPEC-NAME} \in Dom(\Gamma_s)\\ \texttt{SPEC-NAME} \notin Ref^+_{\mathcal{L}}(\texttt{SPEC})\backslash Ref(\texttt{SPEC})\\ \texttt{SPEC-NAME} \notin Ref(\texttt{GENERICITY})\\ \Gamma_s \vdash \texttt{GENERICITY} \rhd (\Sigma_I, (\Delta_1 \ldots \Delta_n))\\ \Sigma_I \cup (\Delta_1 \cup \ldots \cup \Delta_n), \Gamma_s \vdash \texttt{SPEC} \rhd \Delta_B\\ \Gamma_m \vdash \texttt{GENERICITY} \Rightarrow (\mathcal{M}_I, (\mathcal{M}_1 \ldots \mathcal{M}_n))\\ \mathcal{M}_P = \{M \in \mathbf{Mod}(\Sigma_I \cup (\Delta_1 \cup \ldots \cup \Delta_n)) \mid M|_{\Sigma_I} \in \mathcal{M}_I, M|_{\Sigma_I \cup \Delta_i} \in \mathcal{M}_i, 1 \leq i \leq n\}\\ \Gamma_s, \Gamma_m, \mathcal{M}_P, \Sigma_I \cup (\Delta_1 \cup \ldots \cup \Delta_n) \vdash \texttt{SPEC} \Rightarrow \mathcal{M}'_{\texttt{SN}}\\ \Gamma'_m = Subst(\texttt{SPEC-NAME} \to \mathcal{M}'_{\texttt{SN}})\ in\ \Gamma_m(until\ \texttt{SPEC-NAME}) ::\\ Propage\{\texttt{SPEC-NAME}\}\ accordingto\ \mathcal{L}\ in\ \Gamma_m(from\ \texttt{SPEC-NAME})\end{array}}{\Gamma_s, \Gamma_m \vdash \texttt{spec-redefn SPEC-NAME GENERICITY SPEC} \Rightarrow \Gamma'_m} \quad (8)$$

Correction

Les bibliothèques de spécification ont une structure très simple. La seule propriété que l'on peut vérifier sur une bibliothèque est le fait que le graphe de dépendance entre les noms de spécification est acyclique, mais pas forcément connexe. Cette propriété est évidemment vérifiée dans Casl grâce à la propriété de visibilité linéaire. Cette propriété doit toujours être vérifiée dans l'extension que nous venons de proposer.

Le graphe de dépendance d'une bibliothèque est un graphe dont les sommets sont les noms de spécification, qui sont reliés par une arête orientée lorsque le second sommet fait référence au premier dans l'expression de spécification qui lui est associé dans la bibliothèque. Si la bibliothèque est bien formée, c'est-à-dire dont on a pu construire l'environnement statique, alors la propriété suivante est vérifiée :

Proposition 3.5
Si $\mathcal{L}$ est une bibliothèque bien formée, alors son graphe de dépendance est acyclique.

Preuve: Cette propriété se prouve par récurrence sur la longueur de la bibliothèque. Deux cas se présentent :

1. Soit le prochain élément est une déclaration de spécification `spec-defn SN PAR IMP SP`. Dans ce cas, par la première prémisse de la règle (3) un nouveau sommet est ajouté, et autant d'arêtes que de noms de spécification dans $Ref(\texttt{SP}) \cup Ref(\texttt{PAR}) \cup Ref(\texttt{IMP})$. Ces arêtes ont toutes pour but le noeud `SN`, puisque par la prémisse de la règle (5), les références doivent être présentes dans l'environnement avant la déclaration de `SN`. Ces arêtes ne peuvent pas créer de cycle car il faudrait pour cela que `SN` soit l'origine d'au moins une arête pour cela.
2. Le prochain élément est une redéfinition de spécification `spec-redefn SN PAR IMP SP`. Dans ce cas, aucun sommet n'est ajouté puisque la première prémisse de la règle (8) assure que `SN` était déjà utilisé. De nouvelles arêtes sont ajoutées comme précédemment. Ces arêtes ne peuvent créer un cycle, puisqu'il faudrait alors qu'il existe un nom de spécification `X` dans $Ref(\texttt{SP}) \cup Ref(\texttt{PAR}) \cup Ref(\texttt{IMP})$ tel que $\texttt{SN} \in Ref^+_{\mathcal{L}}(\texttt{X})$. Or la condition de bonne formation : $\texttt{SN} \notin Ref^+_{\mathcal{L}}(\texttt{SP}) \backslash Ref(\texttt{SP}) \wedge \texttt{SN} \notin Ref(\texttt{PAR}) \wedge \texttt{SN} \notin Ref(\texttt{IMP})$, empêche explicitement cette situation. Le graphe reste également acyclique dans ce cas.

□

Pseudo-code

Le pseudo-code ML suivant permet de se faire une idée plus précise du fonctionnement de ces opérations. Ces fonctions traitent du cas des signatures. `Eval_sig_lib` est la fonction d'analyse statique globale d'une bibliothèque. Les deux cas d'une déclaration et d'une redéfinition sont traités. Le premier cas reprend la règle (3), le second la règle (8) avec la condition de non-circularité indirecte. `Eval_sig` est la fonction d'évaluation de la signature d'une expression de spécification. Elle prend en argument un environnement de signatures, une expression de spécification et la signature de l'environnement local. Cette fonction code les règles de calcul des signatures à partir de la syntaxe abstraite. Ces règles seront données en détail dans le chapitre suivant pour les spécifications structurées. `Expr` est une fonction prenant en argument un nom de spécification et une bibliothèque de syntaxe abstraite. Elle rend la syntaxe abstraite de l'expression de spécification associée au nom de spécification passé en argument dans la bibliothèque passée en argument. Comme trois expressions peuvent être associées à un même nom (les paramètres, les imports, et le corps), trois sous-fonctions sont définies pour chaque cas.

`L` est la syntaxe abstraite d'une bibliothèque, `G` un environnement de signatures, `S1,S3` des signatures, `S2` une liste de signatures, `l` une liste de noms de spécifications.

```
Eval_sig_lib  L =
```

```
let G = EmptySet in                     /* Initialisation */
  for i=0 to (Length L) do
  /* Cas d'une déclaration    */
    if (L[i] = spec-defn SN PAR IMP SP) &  not(SN in G) then
        begin
        S1 = Eval_sig G IMP ();
        S2 = Eval_sig G PAR S1;
        S3 = Eval_sig G SP (Union S2);
        G:= Cons(G, (SN->(S1,S2,S3)));
        end;
  /* Cas d'une redefinition  */
    if (L[i] = spec-redefn SN PAR IMP SP)
              &  (SN in G) & not(SN in (Ref* L SP))
              &  not(SN in Ref(PAR)) & not(SN in Ref(IMP)) then
        begin
        S1 = Eval_sig G IMP ();
        S2 = Eval_sig G PAR S1;
        S3 = Eval_sig G SP (Union S2);
        G:= Cons(Subst (SN->(S1,S2,S3)) in (G until SN),
                   Propage [SN] accordingto L in (G from SN));
        end;


Subst (SN->(S1,S2,S3)) in G =
   if G = Nil then Nil
     else /* G est de la forme Cons(SN'->(S1',S2',S3'), G') */
         if SN = SN' then  Cons(SN->(S1,S2,S3),G')
         else   Cons(SN'->(S1',S2',S3'), Subst (SN->(S1,S2,S3)) in G'



Propage l accordingto L in G =
    bool b = 0;
    if G = Nil then Nil
     else /* G est de la forme Cons(SN->(S1,S2,S3), G') */
       if  (Ref(Expr_imports SN L) in l ) then
            {S1+ =  Eval_sig (Expr_imports SN L) (); b = 1;}
       if  (Ref(Expr_params SN L) in l ) then
            {S2+ =  Eval_sig (Expr_imports SN L) S1+; b = 1;}
       if  (Ref(Expr_corps SN L) in l ) then
            {S3+ =  Eval_sig (Expr_imports SN L) (Union S2+); b = 1;}
       if b = 1 then
          Cons(SN->(S1+,S2+,S3+),
```

```
                Propage Cons(l,SN) accordingto L in G'));
       else  Cons(SN->(S1,S2,S3),Propage l accordingto L in G')
```

L'opération de propagation, telle que nous l'avons définie, est très inefficace. En effet, pour chaque expression de spécification devant être mise à jour après une redéfinition, toute la signature est de nouveau évaluée. Or dans le cas le plus courant, c'est à dire d'une redéfinition par extension, seule l'extension de signature provenant de la redéfinition pourrait être évaluée. Nous proposons d'étudier une amélioration de cette opération dans le chapitre suivant.

4. Optimisation de la propagation dans les environnements statiques

La méthode décrite dans le chapitre précédent pour la propagation d'une redéfinition dans les environnements Γ_s et Γ_m ne peut pas être implémentée directement. L'algorithme que nous avons proposé est très rudimentaire : il identifie les spécifications nommées devant être mises à jour en testant si ces spécifications font référence à un nom de spécification dont la sémantique a déjà été mise à jour, puis réévalue complètement leur sémantique en utilisant la fonction `Eval_sig` codant les règles de la sémantique statique de CASL.

Nous étudions dans ce chapitre le cas particulier des redéfinitions par extension de spécifications non génériques. Ce cas particulier de redéfinition correspond à une utilisation naturelle de la redéfinition. En effet, dans la pratique, une redéfinition consiste la plupart du temps à ajouter une information dans une spécification nommée inachevée, c'est-à-dire étendre une spécification nommée par de nouvelles informations. Dans ce cas particulier, il est alors possible d'optimiser le calcul de la mise à jour de la sémantique statique. Nous ne nous occupons dans ce chapitre que d'une optimisation de la sémantique statique. Les règles de la sémantique par classe de modèles ne peuvent pas être codées en une fonction `Eval_mod` évaluant la classe des modèles associée à une expression de spécification. A fortiori, aucune optimisation de cette éventuelle fonction ne peut être proposée.

L'optimisation du calcul de la mise à jour de la sémantique statique consiste à remplacer dans l'algorithme de la fonction de propagation l'appel à la fonction `Eval_sig` par des formules prédéfinies. Plutôt que de réévaluer toute la sémantique, ces formules codent la modification à apporter à l'ancienne sémantique en fonction de la sémantique des nouvelles informations. Nous verrons que dans le cas des redéfinitions par extension, la nouvelle sémantique correspond simplement à l'ancienne sémantique étendue par celle de l'extension, moyennant quelques adaptations. La partie correspondant à l'ancienne sémantique n'a pas besoin d'être réévaluée. Les règles statiques, codées par la fonction `Eval_sig`, servent à deux choses : évaluer la signature et vérifier les conditions de bonne formation syntaxique. L'application de formules à la place de ces règles doit donc être combinée avec une preuve de la bonne formation de la spécification résultante.

Considérons une bibliothèque de spécification $\mathcal{L}$, consistant en une suite d'arbres de syntaxe abstraite de spécifications nommées, munie de sa sémantique statique sous la forme d'un environnement Γ_s. Soit SN une spécification nommée bien formée déclarée dans $\mathcal{L}$, et SM une autre spécification nommée utilisant SN par un opérateur de structuration **op**

comme suit :
spec SN = *SPN*
spec *SM* = **op** (SN, SPM)

Supposons ensuite que SN soit redéfinie plus loin dans le texte de la bibliothèque par une extension de la forme :
spec SN $\stackrel{\text{re}}{=}$ SN **then** *SPO*
où SPO représente une spécification bien formée quelconque. L'intuition derrière notre optimisation est la suivante : sachant que SM avant la redéfinition et SPO sont bien formées, les conditions de bonne formation de SM après la propagation de la redéfinition de SN peuvent se déduire, dans la plupart des cas, de la bonne formation de ces deux spécifications, et n'ont donc pas besoin d'être revérifiées. Le cas dans lequel cette bonne formation ne peut être conservée automatiquement est lorsque **op** est l'opération d'instanciation. Que ce soit dans le cas où SN est utilisé comme paramètre effectif de l'instanciation ou bien comme paramètre formel de la spécification que l'on instancie ou de SM, la bonne formation ne peut être conservée. Dans tous les autres cas, la bonne formation des spécifications est donc conservée.

Dans les cas où la bonne formation est conservée, la mise à jour de la sémantique de SM correspond en partie à son ancienne sémantique dans Γ_s et en partie à l'évaluation de l'extension dénotée par SPO sous **op**, c'est-à-dire $op(SPO)$, où *op* est l'opération sur la signature correspondant à **op**. Si on note Δ_O l'extension de signature dénotée par SPO, et les nouvelles évaluations après la redéfinition par un $\_^+$, la nouvelle sémantique statique de SM, Σ_M^+, est donnée par une formule utilisant Σ_M, Δ_O, et *op*.

Ce chapitre est organisé comme suit. Nous devons dans un premier temps énoncer les règles de bonne formation des spécifications structurées dans Casl. La sémantique précise des opérateurs de structuration est relativement complexe dans Casl. Contrairement à d'autres langages de spécification algébriques, pour lesquels la sémantique des opérateurs peut être donnée en quelques lignes, Casl est un langage basé sur une grammaire précise, et possède des facilités syntaxiques afin d'alléger l'écriture des spécifications. De nombreuses règles intermédiaires sont nécessaires pour faire le lien entre les symboles de la syntaxe abstraite d'une spécification et les objets de la sémantique. Ce point sera discuté dans la section 4.1. Nous complétons ensuite la grammaire de la syntaxe abstraite pour les spécifications structurées dans la section 4.2.1. Nous donnerons, à partir de la section 4.2.2, la sémantique précise de chaque opérateur. La sémantique des spécifications de base, c'est à dire les spécifications ne contenant que des déclarations de signature et d'axiomes, est considérée ici en tant que cas particulier de spécification structurée. Pour chaque opérateur nous mettrons en évidence les conditions de bonne formation correspondante. Nous ne nous intéressons qu'à un sous-ensemble des opérateurs les plus représentatifs de Casl. Le cas de la paramétrisation est étudié séparément à la fin de ce chapitre. L'extension de notre travail aux opérateurs auxiliaires (comme le fait de pouvoir définir une spécification localement, ou de pouvoir évaluer une spécification dans un environnement vide) ne comporte aucune difficulté, puisque ceux ci sont définis à partir des opérateurs que nous aurons étudiés. Nous montrerons ensuite que les conditions de bonne formation sont préservées lors d'une redéfinition par extension, puis nous donnerons dans la section 4.3 des formules reliant les anciennes valeur de la sémantique de SM à la nouvelle. Le cas où SN est un paramètre formel ou effectif de SM doit être traité à part, puisque dans ce cas la correction de la spécification SM ne peut pas être assurée.

4.1. Introduction à la sémantique des opérateurs

Le langage Casl est conçu comme une synthèse des idées provenant des études sur divers langages de spécification algébriques (comme EML, ASL, Spectrum, etc.). De manière classique, la sémantique d'un opérateur de structuration de spécifications correspond à une opération sur la signature et la classe de modèles associées à cette spécification. Ces opérations utilisent intensivement la notion de morphisme de signature et de sa contrepartie sur la classe de modèles, c'est à dire le foncteur de réduction. Le centre d'intérêt des études sur ces langages était plutôt de déterminer quels objets prendre dans la sémantique et comment les structurer. Casl hérite de tous ces travaux, et par conséquent les concepteurs peuvent porter leur attention sur d'autres points.

Dans les langages utilisés pour décrire des concepts sémantiques abstraits, la notion de signature et de morphisme de signature sont des notions de première classe. Cela implique que dans le calcul de la sémantique des opérateurs, seuls ces objets sont manipulés. Dans Casl, au contraire, la sémantique est conçue à partir d'une syntaxe abstraite, déclarant des identificateurs munis de leur qualification. Les objets de première classe sont ces déclarations de symboles auxquelles il faut associer *in fine* une signature ou un morphisme. Le principal attrait de prendre ces déclarations comme objets de première classe est d'alléger l'écriture des spécifications. Lors d'un renommage, par exemple, il est possible de ne mentionner que les symboles que l'on veut modifier :

spec Exemple1 =
  sort s, s'
  ops $f : s \to s'; g : s \times s \to s'$
with $f \mapsto h$

Un morphisme est alors reconstruit à partir de cette description, puis une signature résultant de l'application de ce morphisme à la signature de départ. Mais de nombreuses facilités syntaxiques compliquent cette reconstruction, que ce soit pour un morphisme comme dans l'exemple ci-dessus, ou pour une signature dans le cas d'une restriction. Une première facilité est d'omettre le profil d'un symbole d'opération ou de prédicat lors de la déclaration d'un renommage ou d'une restriction. La plupart du temps, le profil se déduit aisément du contexte dans lequel ce symbole apparaît. Dans l'exemple précédent, le profil du symbole f est omis lors de son renommage. La signature résultante de cette spécification contient un symbole h dont le profil est issu de celui de f, $h : s \to s'$. Ce profil doit être ajusté si le renommage concerne également une sorte. Par exemple, si la clause $s \mapsto t$ est ajoutée à la déclaration de renommage, le profil du symbole h est alors $h : t \mapsto s'$.

Une autre facilité syntaxique de Casl est la possibilité de surcharger un symbole. Un même symbole peut donc être utilisé avec des profils distincts, comme dans la spécification :

spec EXEMPLE2 =
  sort s, s'
  ops $f : s \to s'; f : s \times s \to s'$
  pred $f : s \times s'$
with $f \mapsto h$

Dans ce cas, dans une opération comme le renommage, si les symboles apparaissant dans la déclaration d'une liste ou d'une application de symboles ne sont pas qualifiés, tous les symboles surchargés sont affectés. La signature de la spécification EXEMPLE2 comporte donc trois symboles portant l'identificateur h : $h : s \to s'$, $h : s \times s \to s'$, $h : s \times s'$. Cette souplesse d'écriture des déclarations des applications ou des listes de symboles peut apporter parfois une ambiguïté non souhaitée. Pour lever cette ambiguïté, deux solutions sont possibles. La première est d'insérer dans la déclaration de la liste ou de l'application de symboles des genres permettant de différencier les symboles. Si dans l'exemple, seules les opérations doivent être renommées, on aurait :

spec EXEMPLE3 = ...
with op $f \mapsto h$

La seconde possibilité est de qualifier explicitement les symboles. Si dans l'exemple, seule la première opération doit être renommée, on aurait :

spec EXEMPLE3 = ...
with $f : s \to s' \mapsto h$

Les deux solutions pouvant bien entendu se combiner.

Un dernier point rend la reconstruction des morphismes et des signatures difficile. Même lorsqu'une unique application (ou liste) de symboles est déterminée à partir d'une déclaration, il se peut que plusieurs morphismes (ou signatures) correspondent. Dans le cas d'un renommage, seule la signature de départ est donnée. La signature d'arrivée est seulement déduite de l'application d'un morphisme à cette signature de départ. Il existe une infinité de signatures d'arrivée. Il faut donc restreindre de manière unique cette signature pour déterminer une unique reconstruction de morphisme à partir d'une déclaration d'application de symboles. De plus, la gestion des relations de sous-sortage dans une reconstruction est problématique. Un sous-sortage entre deux sortes correspond à une application des valeurs de la première vers celles de la seconde.[1] Cette relation de sous-sortage permet par exemple de spécifier une opération sur chacune des sous-parties de son domaine de manière progressive. Pour reconstruire de manière unique une signature ou un morphisme, il est nécessaire en plus de contraindre la relation de sous-sortage dans la signature d'arrivée.

Le lecteur peut se rendre compte, à l'évocation de ces difficultés, de la complexité de la reconstruction des signatures et des morphismes de signature à partir des déclarations de listes ou d'applications de symboles. Cette approche est nécessaire lorsqu'on se situe dans le cadre d'un langage possédant une syntaxe abstraite bien définie dont on veut définir une sémantique formelle. Nous présentons les notions de base de la sémantique avant de définir la grammaire de la syntaxe abstraite.

[1] En général, il s'agit d'une inclusion, mais la notion de plongement permet de décrire des situations plus complexes.

4.1.1. Notions de base

Symboles. Afin de pouvoir être traités par les règles de la sémantique, les symboles sont considérés également comme des objets sémantiques. Ce sont les objets sémantiques de première classe dans la sémantique de CASL. Un symbole $SY \in Sym$ est soit un identificateur de sorte s, soit un identificateur de fonction totale qualifié f^{t}_{ws} où $ws \in Sort^* \times Sort$, soit un identificateur de fonction partielle qualifié f^{p}_{ws}, soit un identificateur de prédicat qualifié p_w. Le genre des symboles est déterminé par les mots clés dans la déclaration de la signature.

La notion de symbole brut sera utile dans la suite. Un symbole brut $RSY \in RawSym$ est soit un symbole qualifié, soit un symbole qualifié muni d'un genre parmi $(implicit, sort, op, pred)$.

Signatures. Une signature de CASL est un ensemble de symboles structuré défini par un quintuplet : $\Sigma = (S, TF, PF, P, \leq)$. S est un ensemble de symboles de sortes, TF et PF sont des ensembles disjoints de symboles d'opérations totales ou partielles munis de leur profil ws, P est un ensemble de symboles de prédicats munis de leur profil w, et $\leq$ est un préordre (une relation réflexive et transitive) sur l'ensemble S des sortes. Un même identificateur peut être surchargé avec plus d'un profil. Dans le cas où ces profils utilisent des sortes du préordre, une relation de surcharge est définie entre des symboles d'opérations ou des symboles de prédicats permettant d'utiliser le profil adapté au contexte.

Étant donnée une signature $\Sigma = (S, TF, PF, P, \leq)$, l'ensemble de symboles sous-jacent qu'elle détermine, noté $|\Sigma|$, est donné par :

$$\begin{aligned} |\Sigma| = S &\cup \{f^{\mathsf{t}}_{ws} \mid f \in TF_{ws}\} \\ &\cup \{f^{\mathsf{p}}_{ws} \mid f \in PF_{ws}\} \\ &\cup \{p_w \mid p \in P_w\} \end{aligned}$$

L'union de deux signatures est définie par l'union composante par composante des symboles, et par la clôture transitive de l'union des préordres.

Morphismes de signature. Un morphisme de signature $\sigma : (S, TF, PF, P, \leq) \rightarrow (S', TF', PF', P', \leq')$ est une application de symboles structurée conservant le genre des symboles : un symbole de S est transformé en un symbole de S', un symbole de TF_{ws} en un symbole de $TF'_{\sigma(ws)}$, un symbole de PF_{ws} en un symbole de $TF'_{\sigma(ws)} \cup PF'_{\sigma(ws)}$ (un morphisme ne conserve pas la partialité d'une opération, mais il n'y a pas de distinction entre opération totale ou partielle dans le genre), et un symbole de P_{ws} en un symbole de $P'_{\sigma(ws)}$. Un morphisme conserve de plus le préordre de sous-sortage : si $s \leq s'$ alors $\sigma(s) \leq \sigma(s')$, et les relations de surcharge.

À tout morphisme de signature σ est associé une application de symboles qualifiés sous-jacente $|\sigma|$. Cette application de symboles est construite trivialement en associant les symboles de chaque signature.

Une extension Δ d'une signature Σ est définie comme un morphisme d'inclusion entre Σ et $\Sigma' = \Sigma \cup \Delta$. On notera alors cette extension $\Delta : \Sigma \rightarrow \Sigma'$.

```
SPEC ::= BASIC-SPEC | EXTENSION | UNION | RESTRICTION
       | REDUCTION | RENOMMAGE |  SPEC-INST


EXTENSION ::= extension SPEC+
UNION ::= union SPEC+
RESTRICTION ::= restriction SPEC SYMB-ITEMS+
REDUCTION ::= reduction SPEC SYMB-MAP-ITEMS+
RENOMMAGE ::= renaming SPEC SYMB-MAP-ITEMS+

SPEC-INST ::= spec-inst SPEC-NAME FIT-ARG
FIT-ARG ::= fit-arg SPEC SYMB-MAP-ITEMS*

SYMB-MAP-ITEMS ::= symb-map-items (SYMB-KIND SYMB-MAP)+
SYMB-MAP ::= symb-map SYMB SYMB
SYMB-ITEMS ::= symb-items (SYMB-KIND SYMB)+
SYMB-KIND ::= implicit | sort | op | pred
SYMB ::= ID | QUAL-ID
QUAL-ID ::= qual-id ID TYPE
TYPE ::= OP-TYPE | PRED-TYPE | SORT
OP-TYPE ::=  total-op-type SORTS* SORT | partial-op-type SORTS* SORT
PRED-TYPE ::= pred-type SORTS+
```

FIG. 4.1.: Syntaxe abstraite des spécifications structurées

Modèles. Un modèle $M \in \mathbf{Mod}(\Sigma)$ consiste en un support non vide pour chaque sorte dans S, une fonction totale pour chaque symbole de fonction dans TF, une fonction partielle pour chaque symbole de fonction dans PF, une relation ou un sous-ensemble pour chaque prédicat dans P et un plongement pour chaque paire de sortes reliées par la relation de sous-sortage. Les plongements sont des injections quelconques, pas simplement des inclusions de supports. Ils déterminent également des projections partielles (des sur-sortes vers les sous-sortes) et un prédicat d'appartenance (qui est valide sur les valeurs d'une sur-sorte qui sont images par le plongement d'une valeur d'une sous-sorte).

4.2. Sémantique formelle de CASL : suite

4.2.1. Syntaxe abstraite

Cette partie de la grammaire prolonge celle donnée dans la section 3.2.2. Une spécification `SPEC` est soit une spécification de base, soit une extension d'une ou plusieurs spécifications, soit l'union d'une ou plusieurs spécifications, soit une restriction d'une spécification (on indique alors la partie de la signature que l'on désire ne plus conserver), soit une réduction (on indique quelle partie de la signature on désire conserver), soit un renommage d'une autre spécification, soit enfin une référence à une spécification nommée déjà déclarée. Dans ce dernier cas, puisque toute spécification est générique, une référence consiste à instancier une spécification générique par des arguments possiblement ajustés par un renommage. Dans le cas où cette spécification n'a aucun paramètre, l'instanciation est alors une vraie référence. La syntaxe des spécifications de base n'est pas abordée ici.

Un renommage, une réduction, ou l'ajustement d'un paramètre utilisent des applications de symboles. Une application de symboles est composée d'une liste d'associations de couples $(genre, symbole)$. Un genre détermine la catégorie syntaxique d'un symbole (une sorte, une opération, un prédicat, ou bien un genre indéterminé). Cette liste peut être scindée en différentes parties en y insérant les mots clés **sort**, **ops**, ou **preds**, indiquant de façon explicite le genre des symboles. Dans le cas contraire, tous les symboles ont un genre implicite. Dans CASL, l'accent est mis sur la souplesse d'écriture des spécifications. Ainsi le spécifieur peut ne pas indiquer le profil des symboles qu'il est en train de renommer, dans le cas où ce profil peut être déduit du contexte. Cela est également possible lorsqu'il déclare la liste des symboles qui ne doivent plus apparaître après une restriction. De plus, les symboles de la spécification de départ qui ne sont pas renommés peuvent ne pas apparaître dans l'application de symboles, comme nous l'avons évoqué dans la section précédente.

4.2.2. Spécifications de base

Une spécification de base consiste en une séquence de déclarations de symboles de sortes, d'opérations, de prédicats, et d'un ensemble d'axiomes contraignant les interprétations possibles de ces symboles dans les modèles. Ces déclarations suivent une syntaxe abstraite précise, déterminant l'ensemble des phrases correctes du langage (cf. l'annexe A.1). Une spécification de base fait partie de la catégorie syntaxique `BASIC-SPEC`. Les règles statiques produisent un ensemble de symboles formant une signature et un ensemble d'axiomes, les règles de modèles une classe de modèles au dessus de cette signature satisfaisant les axiomes.

L'objet de cette section n'est pas de détailler cette syntaxe abstraite ni les règles correspondantes, puisque nous voulons nous concentrer sur la sémantique des opérateurs de structuration. La catégorie syntaxique `BASIC-SPEC` est donc considérée comme un symbole terminal de la syntaxe abstraite des spécifications structurées. Par conséquent, les règles pour une spécification de base sont décrites comme des cas particuliers de règles de spécifications structurées pour lesquelles il n'y a pas de prémisses.

Soit Σ la signature de l'environnement local correspondant à la partie de la spécifcation déjà évaluée, et $\mathcal{M}$ une classe de modèles au dessus de Σ. Une spécification de base dénote un ensemble de symboles $SYs \in Sym$ déterminant un enrichissement de signature Δ et un ensemble d'axiomes Ψ, c'est-à-dire $\Sigma \vdash \texttt{BASIC-SPEC} \rhd (\Delta, \Psi)$, ainsi qu'une classe de modèles $\mathcal{M}'$ au-dessus de $\Sigma' = \Sigma \cup \Delta$, qui étendent les modèles dans $\mathcal{M}$, et satisfont les axiomes de Ψ. $\mathcal{M}'$ étend $\mathcal{M}$ si pour tout modèle $M' \in \mathcal{M}'$ il existe un modèle $M \in \mathcal{M}$ tel que la réduction de M' à la signature Σ (notée $M'|_\Sigma$) est égale à M. L'oubli des interprétations des symboles de Δ doit donc donner un modèle de Σ.

$$\frac{\mathcal{M}' = \{M' \in \mathbf{Mod}[\Sigma \cup \Delta] \mid M'|_\Sigma \in \mathcal{M} \text{ et } \forall \psi \in \Psi . M' \models \psi\}}{\Sigma, \mathcal{M} \vdash \texttt{BASIC-SPEC} \Rightarrow \mathcal{M}'} \quad (9)$$

4.2.3. Extension

Une extension se note :

$$SP_1 \textbf{ then } SP_2 \ldots \textbf{then } SP_n$$

Une extension de spécification SP_i d'une spécification SP_{i-1} détermine une extension de signature $\Delta : \Sigma_{i-1} \to \Sigma_i$ et une classe de modèles $\mathcal{M}_i$ au-dessus de Σ_i telle que tout modèle de $\mathcal{M}_i$ se réduise sur un modèle appartenant à la classe des modèles de SP_{i-1}, notée $\mathcal{M}_{i-1}$. L'environnement local croit lors de l'évaluation d'extensions successives.

Il s'agit de la notion la plus générale d'extension. Il est possible d'ajouter des annotations (c'est à dire des mots clés qui n'ont pas de contrepartie dans la sémantique, mais imposent des obligations de preuves), afin de contraindre la forme des extensions aux extensions conservatives. Une extension est conservative si tout modèle de $\mathcal{M}_{i-1}$ est le réduit d'un modèle de $\mathcal{M}_i$, c'est à dire qu'aucun modèle de $\mathcal{M}_{i-1}$ n'est perdu par l'extension.

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SP1} \rhd \Delta_1 \\ \Sigma \cup \Delta_1, \Gamma_s \vdash \texttt{SP2} \rhd \Delta_2 \\ \ldots \\ \Sigma \cup \Delta_1 \ldots \cup \Delta_{n-1} \vdash \texttt{SPn} \rhd \Delta_n\end{array}}{\Sigma, \Gamma_s \vdash \texttt{extension SP1 SP2 ... SPn} \rhd \Delta_1 \cup \ldots \cup \Delta_2 \cup \Delta_n} \quad (10)$$

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SP1} \rhd \Delta_1 \\ \Sigma \cup \Delta_1, \Gamma_s \vdash \texttt{SP2} \rhd \Delta_2 \\ \ldots \\ \Sigma \cup \Delta_1 \ldots \cup \Delta_{n-1} \vdash \texttt{SPn} \rhd \Delta_n \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SP1} \Rightarrow \mathcal{M}_1 \\ \ldots \\ \Sigma, \mathcal{M}_{n-1}, \Gamma_s, \Gamma_m \vdash \texttt{SPn} \Rightarrow \mathcal{M}_n\end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{extension SP1 SP2 ... SPn} \Rightarrow \mathcal{M}_n} \quad (11)$$

4.2.4. Union

L'union de plusieurs spécifications nommées se note :

$$SP_1 \textbf{ and } SP_2 \ldots \textbf{ and } SP_n$$

Lorsque l'environnement local est vide, chaque SP_i doit déterminer une signature complète. La signature de l'union est obtenue par l'union ensembliste (pas l'union disjointe), ainsi chaque occurrence d'un symbole est interprétée de la même manière. En revanche, lorsque l'environnement local n'est pas vide, chaque SP_i doit déterminer une extension de la signature de l'environnement local. L'union des signatures est une union ensembliste structurée, construite composante par composante.

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SP1} \rhd \Delta_1 \\ \dots \\ \Sigma, \Gamma_s \vdash \texttt{SPn} \rhd \Delta_n \\ \Delta = \Delta_1 \cup \Delta_2 \cup \dots \Delta_n\end{array}}{\Sigma, \Gamma_s \vdash \texttt{union SP1 ...SPn} \rhd \Delta} \quad (12)$$

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SP1} \rhd \Delta_1 \\ \dots \\ \Sigma, \Gamma_s \vdash \texttt{SPn} \rhd \Delta_n \\ \Delta = \Delta_1 \cup \Delta_2 \cup \dots \Delta_n \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SP1} \Rightarrow \mathcal{M}_1 \\ \dots \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SPn} \Rightarrow \mathcal{M}_n \\ \mathcal{M}' = \{M \in \mathbf{Mod}(\Sigma \cup \Delta) \mid M|_{\Sigma \cup \Delta_i} \in \mathcal{M}_i, 1 \leq i \leq n\}\end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{union SP1 ...SPn} \Rightarrow \mathcal{M}'} \quad (13)$$

4.2.5. Listes de symboles

Les listes de symboles sont utilisées lors d'une restriction. Une liste de symboles contient les symboles de la spécification courante que l'on veut masquer. La sémantique de cette opération consiste à construire une signature contenant les symboles de la signature courante moins ceux apparaissant dans la liste. Comme nous l'avons mentionné plus haut, il est possible de n'indiquer ni le profil ni le genre des symboles dans une liste de symboles, afin d'alléger l'écriture de la spécification. Or une signature est un ensemble structuré de symboles qualifiés. Il s'agit donc dans une première étape de construire un ensemble de symboles qualifiés à partir de la déclaration d'une liste de symboles. Notons que le terme «symboles» est identique pour dénoter des objets de la syntaxe abstraite ou des objets participant à la sémantique, donc des objets au sens mathématique du terme.

Une liste de symboles avec un genre implicite est notée :

$$SY_1, \dots, SY_n$$

où chaque SY_i est un symbole dont le profil peut être omis. Il s'agit alors de simples identificateurs. Dans le cas où SY_i n'est pas qualifié, tous les symboles ayant le même identificateur dans l'environnement local courant sont pris en compte. Cette liste peut être sectionnée en insérant les mots clés **sort**, **op**, **pred**, explicitant ainsi le genre des symboles. La présence de ces mots clés permet par exemple de lever une ambiguïté entre un symbole de constante qualifié et un symbole de prédicat qualifié. Dans le cas où ces mots clés n'apparaissent pas dans la liste, le genre des symboles est noté *implicit*.

Les règles suivantes construisent dans un premier temps, dans un environnement vide, une liste de symboles tel que chaque symbole est soit qualifié, soit muni d'un genre, à partir d'une liste de symboles SYMB-ITEMS+. On appelle symboles bruts ces symboles. Les deux premières règles permettent de séparer des sous-listes à l'intérieur d'une liste afin de tenir compte des mots clés possiblement introduits.

$$\frac{\begin{array}{c}\vdash \texttt{SYMB-ITEMS} \rhd RSYs \\ \vdash \texttt{SYMB-ITEMS+} \rhd RSYs'\end{array}}{\vdash \texttt{SYMB-ITEMS}\,\texttt{SYMB-ITEMS+} \rhd RSYs \cup RSYs'} \quad (14)$$

$$\frac{\vdash \texttt{SYMB-ITEMS} \rhd RSYs}{\vdash \texttt{SYMB-ITEMS}\ \mathrm{qua}\ \texttt{SYMB-ITEMS+} \rhd RSYs} \quad (15)$$

Pour chaque sous-liste de symboles sous la portée d'un unique genre k, un ensemble de symboles est déterminé :

$$\frac{k \vdash \texttt{SYMB} \rhd RSY}{\vdash \texttt{symb-items}\ k\ \texttt{SYMB} \rhd \{RSYs\}} \quad (16)$$

$$\frac{\begin{array}{c}k \vdash \texttt{SYMB} \rhd RSY \\ \vdash \texttt{symb-items}\ k\ \texttt{SYMB} \rhd RSYs\end{array}}{\vdash \texttt{symb-items}\ k\ \texttt{SYMB}\ \texttt{SYMB+} \rhd \{RSY\} \cup RSYs} \quad (17)$$

Si le genre est explicite, alors les règles suivantes sont utilisées. Dans ces règles, les prémisses concernant les identificateurs ne sont pas détaillées, puisque nous n'avons pas donné la syntaxe abstraite d'un identificateur. Les identificateurs sont donc notés comme des symboles terminaux dans ces règles.

$$\frac{s \in \texttt{ID}}{sort \vdash s \rhd s} \quad (18)$$

$$\frac{f \in \texttt{ID}}{fun \vdash f \rhd (fun, f)} \quad (19)$$

$$\frac{f \in \texttt{ID}}{fun \vdash \texttt{qual-id}\ f\ (\texttt{total-op-type}(\texttt{sorts}\ s_1, \ldots, s_n)\ s) \rhd f^{\mathtt{t}}_{s_1,\ldots,s_n\ s}} \quad (20)$$

$$\frac{f \in \texttt{ID}}{fun \vdash \texttt{qual-id}\ f\ (\texttt{partial-op-type}(\texttt{sorts}\ s_1, \ldots, s_n)\ s) \rhd f^{\mathtt{p}}_{s_1,\ldots,s_n\ s}} \quad (21)$$

$$\frac{p \in \texttt{ID}}{pred \vdash p \rhd (pred, p)} \quad (22)$$

$$\frac{p \in \mathtt{ID}}{\mathit{pred} \vdash \mathtt{qual\text{-}id}\ p\ (\mathtt{pred\text{-}type}(\mathtt{sorts}\ s_1, \ldots, s_n)) \rhd p_{s_1,\ldots,s_n}} \ (23)$$

Dans le cas contraire où le genre n'est pas explicité par des mots clés dans la liste de symboles, les règles suivantes sont utilisées :

$$\frac{\mathit{Ident} \in \mathtt{ID}}{\mathit{implicit} \vdash \mathit{Ident} \rhd (\mathit{implicit}, \mathit{Ident})} \ (24)$$

$$\frac{f \in \mathtt{ID}}{\mathit{implicit} \vdash \mathtt{qual\text{-}id}\ f\ (\mathtt{total\text{-}op\text{-}type}(\mathtt{sorts}\ s_1, \ldots, s_n)\ s) \rhd f^{\mathtt{t}}_{s_1,\ldots,s_n\ s}} \ (25)$$

$$\frac{f \in \mathtt{ID}}{\mathit{implicit} \vdash \mathtt{qual\text{-}id}\ f\ (\mathtt{partial\text{-}op\text{-}type}(\mathtt{sorts}\ s_1, \ldots, s_n)\ s) \rhd f^{\mathtt{p}}_{s_1,\ldots,s_n\ s}} \ (26)$$

$$\frac{p \in \mathtt{ID}}{\mathit{implicit} \vdash \mathtt{qual\text{-}id}\ p\ (\mathtt{pred\text{-}type}(\mathtt{sorts}\ s_1, \ldots, s_n)) \rhd p_{s_1,\ldots,s_n}} \ (27)$$

Une fois qu'un ensemble de symboles bruts $RSYs$ contenant soit des symboles qualifiés soit des symboles munis d'un genre a été construit, les profils manquants doivent être reconstitués à partir du contexte dans lequel cette liste a été définie. Soit SYs l'ensemble de symboles de l'environnement local courant. Un symbole SY_i non qualifié peut correspondre à plusieurs symboles qualifiés dans SYs (si ces symboles sont surchargés). Une relation de filtrage permet de déterminer les symboles de l'environnement local correspondant aux symboles non qualifiés de $RSYs$:

Définition 4.1 (Filtrage) *La relation de filtrage $\sqsubseteq$ entre des symboles qualifiés et des symboles nus munis d'un genre est définie par :*

- $s \sqsubseteq (implicit, s)$ *et* $s \sqsubseteq (sort, s)$ *pour toute sorte* s *;*
- $f^{\mathtt{t}}_{ws} \sqsubseteq (implicit, f)$ *et* $f^{\mathtt{t}}_{ws} \sqsubseteq (fun, f)$ *pour toute fonction* f_{ws} *;*
- $f^{\mathtt{p}}_{ws} \sqsubseteq (implicit, f)$ *et* $f^{\mathtt{p}}_{ws} \sqsubseteq (fun, f)$ *pour toute fonction* f_{ws} *;*
- $p_{ws} \sqsubseteq (implicit, p)$ *et* $p_{ws} \sqsubseteq (pred, p)$ *pour tout prédicat* p_{ws}

La règle suivante regroupe les différentes étapes : la première ligne associe à un ensemble de symboles bruts contenant des symboles qualifiés et/ou des symboles non qualifiés munis d'un genre à la déclaration d'une liste de symboles. Puis on complète le profil des symboles non qualifiés à partir des symboles de l'environnement courant en suivant la relation de filtrage. Si le profil d'un symbole non qualifié ne peut pas être retrouvé, cela cause une erreur de bonne formation dans la spécification courante.

$$\frac{\begin{array}{c}\vdash \mathtt{SYMB\text{-}ITEMS+} \rhd RSYs \\ SYs' = \{SY \in SYs \mid \exists RSY, SY \sqsubseteq RSY \in RSYs\}\end{array}}{SYs \vdash \mathtt{SYMB\text{-}ITEMS+} \rhd SYs'} \ (28)$$

4.2.6. Restriction

Une fois que l'on a complété le profil de la liste de symboles d'une restriction SP **hide** SL, il reste à construire la signature contenant les mêmes symboles que celle de SP moins ceux listés par SL. Il existe plusieurs choix possibles pour construire cette signature. Dans CASL, cette signature est la plus grande sous signature pleine de SP ne contenant pas ces symboles. La notion de sous-signature pleine est définie avant de présenter cette construction.

Si la signature ne comporte pas de relation de sous-sortage, toute sous-signature est une signature pleine. Dans le cas contraire le préordre de sous-sortage doit contenir toutes les relations issues du préordre de départ concernant les sortes dans la sous-signature :

Définition 4.2 (Sous-signature pleine)
Une sous-signature Σ' de Σ est une sous-signature pleine si et seulement si $\leq' = \leq \cap (S' \times S')$, où S' est l'ensemble des sortes de Σ'.

L'ensemble de symboles $SYs \subseteq |\Sigma|$ est dit fermé dans Σ si il y a une sous-signature pleine Σ' de Σ ayant SYs comme ensemble de symboles sous-jacent. La signature d'une restriction est définie par :

Définition 4.3 (Restriction)
Soit $SYs \subseteq |\Sigma|$ un ensemble de symboles. Il existe une unique sous-signature pleine Σ' de Σ telle que $|\Sigma'|$ est le plus grand ensemble disjoint de SYs fermé dans Σ. Σ' est appelée la signature cogénérée par SYs et se note $\Sigma|^{SYs}$.

Soit $\Sigma = (S, TF, PF, P, \leq)$, SYs un ensemble de symboles inclus dans $|\Sigma|$, et SYs' l'union de SYs et des symboles de fonction ou de prédicat dont une des sortes de leur profil est dans SYs, alors $\Sigma|^{SYs} = (S', TF', PF', P', \leq \cap (S' \times S'))$ avec :

$$\begin{aligned} S' &= \{s | s^{sym} \in |\Sigma| \backslash SYs'\} \\ TF'(ws) &= \{f | f^{\mathsf{t}}_{ws} \in |\Sigma| \backslash SYs'\} \\ PF'(ws) &= \{f | f^{\mathsf{p}}_{ws} \in |\Sigma| \backslash SYs'\} \\ P'(ws) &= \{p | p_w \in |\Sigma| \backslash SYs'\} \end{aligned}$$

Notons que si une sorte est masquée, alors tous les prédicats et opérations ayant cette sorte dans leur profil sont également masqués.

Soit Σ la signature de l'environnement courant, $\Delta : \Sigma \rightarrow \Sigma'$ l'extension de signature dénotée par SP, et SYs l'ensemble de symboles construit à partir de SL dans l'environnement constitué des symboles de Σ'. Cet ensemble doit être disjoint des symboles de Σ, car une restriction ne doit pas affecter de symboles autres que ceux de SP. La signature d'une restriction est la signature cogénérée par SYs dans Σ'.

$$\frac{\begin{array}{c}|\Sigma'| \vdash \texttt{SYMB-ITEMS+} \rhd SYs \\ SYs \cap |\Sigma| = \emptyset \\ \Sigma_1 = \Sigma'|^{SYs}\end{array}}{\Delta : \Sigma \to \Sigma' \vdash \texttt{SYMB-ITEMS+} \rhd \iota_{\Sigma_1 \subseteq \Sigma}} \quad (29)$$

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SPEC} \rhd \Delta : \Sigma \to \Sigma' \\ \Delta : \Sigma \to \Sigma' \vdash \texttt{SYMB-ITEMS+} \rhd \iota_{\Sigma_1 \subseteq \Sigma}\end{array}}{\Sigma, \Gamma_s \vdash \texttt{restriction SPEC SYMB-ITEMS+} \rhd \Sigma_1} \quad (30)$$

La classe des modèles d'une restriction est donnée par l'ensemble des modèles au dessus de Σ_1 qui sont des oublis d'un modèle au dessus de Σ' :

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SPEC} \rhd \Delta : \Sigma \to \Sigma' \\ \Delta : \Sigma \to \Sigma' \vdash \texttt{SYMB-ITEMS+} \rhd \iota_{\Sigma_1 \subseteq \Sigma} \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}' \\ \mathcal{M}_1 = \{M \in \mathbf{Mod}(\Sigma_1) \mid \exists M' \in \mathcal{M}', M'|_{\Sigma_1} = M\}\end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{restriction SPEC SYMB-ITEMS+} \Rightarrow \mathcal{M}_1} \quad (31)$$

4.2.7. Applications de symboles

Les applications de symboles sont utilisées pour l'opération de renommage, de réduction, ou d'ajustement de paramètres lors d'une instanciation. La déclaration d'une application de symboles contient les symboles de la spécification courante que l'on veut renommer, et qui correspondent à ceux que l'on veut conserver dans le cas d'une réduction. Comme pour les listes de symboles, le spécifieur peut omettre le profil de certains symboles dans la déclaration d'une application de symboles. La première étape consiste donc à restaurer ces informations manquantes.

Une application de symboles se note :

$$SY_1 \mapsto SY'_1, \ldots, SY_n \mapsto SY'_n$$

Les symboles du domaine d'une application de symboles doivent être distincts. Les symboles surchargés peuvent être désambiguisés par leur qualification explicite, dans le cas contraire tous les symboles basés sur le même identificateur du contexte sont listés et transformés uniformément en SY'_i. Cette liste peut être scindée en différentes parties en y insérant les mots clés **sort**, **ops**, ou **preds**, indiquant de façon explicite le genre des symboles. Le genre des symboles doit être identique pour chaque couple (SY_i, SY'_i). Il est déterminé de la même manière que pour les listes de symboles. La relation de filtrage est également utilisée pour restaurer les profils manquants.

Les règles suivantes construisent un ensemble de couples de symboles bruts à partir d'une liste d'application de symboles `SYMB-MAP-ITEMS+`. Comme pour les listes de symboles, les deux premières règles permettent de découper la liste d'application en sous-listes pour tenir compte de l'éventuelle présence de genres explicites :

$$\frac{\begin{array}{c}\vdash \texttt{SYMB-MAP-ITEMS} \rhd r \\ \vdash \texttt{SYMB-MAP-ITEMS+} \rhd r'\end{array}}{\vdash \texttt{SYMB-MAP-ITEMS}\ \texttt{SYMB-MAP-ITEMS+} \rhd r \cup r'} \quad (32)$$

$$\frac{\vdash \texttt{SYMB-MAP-ITEMS} \rhd r}{\vdash \texttt{SYMB-MAP-ITEMS}\ \mathrm{qua}\ \texttt{SYMB-MAP-ITEMS+} \rhd r} \quad (33)$$

Pour chaque sous-liste sous la portée d'un genre donné, une liste de couples est construite :

$$\frac{\begin{array}{c}k \vdash \texttt{SYMB-MAP} \rhd r \\ \vdash \texttt{symb-map-items}\ k\ \texttt{SYMB-MAP} \rhd r'\end{array}}{\vdash \texttt{symb-map-items}\ k\ \texttt{SYMB-MAP}\ \texttt{SYMB-MAP+} \rhd r \cup r'} \quad (34)$$

$$\frac{k \vdash \texttt{SYMB-MAP} \rhd r}{\vdash \texttt{symb-map-items}\ k\ \texttt{SYMB-MAP} \rhd r} \quad (35)$$

Puis chaque symbole du couple provenant de l'application de symboles est analysé à partir des règles de la section 4.2.5 :

$$\frac{\begin{array}{c}k \vdash \texttt{SYMB} \rhd RSY \\ k \vdash \texttt{SYMB'} \rhd RSY'\end{array}}{k \vdash \texttt{symb-map}\ \texttt{SYMB}\ \texttt{SYMB'} \rhd \{(RSY, RSY')\}} \quad (36)$$

Comme les applications de symboles servent dans deux opérations, la déclaration d'une application de symboles détermine d'une part une transformation r définie par un ensemble de couples de symboles bruts construit à partir des règles précédentes, et d'autre part un ensemble SYs' de symboles qualifiés de l'environnement local courant, obtenus à partir des symboles du domaine r et du contexte. Les symboles du domaine de r doivent être des symboles de l'environnement local courant. L'ensemble SYs' servira à construire la signature réduite à partir de cet ensemble, tandis que la transformation permettra de construire le morphisme de signature correspondant. On obtient la règle suivante :

$$\frac{\begin{array}{c}\vdash \texttt{SYMB-MAP-ITEMS+} \rhd r \\ \text{Pour tout } RSY \in Dom(r)\text{, il existe } SY \in SYs \sqsubseteq RSY \\ SYs' = \{SY \in SYs \mid SY \sqsubseteq RSY \in Dom(r)\}\end{array}}{SYs \vdash \texttt{SYMB-MAP-ITEMS+} \rhd (SYs', r)} \quad (37)$$

4.2.8. Renommage

Une fois que le profil des symboles d'une application de symboles d'un renommage SP **with** SM a été restauré, il reste à construire le morphisme de signature permettant de transformer la signature de SP vers une nouvelle signature. Il existe plusieurs manières de construire ce morphisme. Dans CASL, ce morphisme est le plus petit morphisme de signature contenant l'application de symboles, c'est à dire le morphisme étant égal à l'identité sur le plus grand domaine de la signature de départ, préservant la relation de sous-sortage. En effet, dans CASL le spécifieur peut omettre lors d'un renommage les symboles qui sont transformés en eux-mêmes, afin de simplifier l'écriture. Il faut donc compléter le domaine de SM de manière unique. Ce morphisme est appelé morphisme *final*. Le terme *final* se réfère à une notion catégorique et correspond à une forme de minimalité de la signature d'arrivée.

La notion de morphisme final est proche de la notion de surjectivité. En la présence de relations de sous-sortage, il faut y ajouter une condition de minimalité du préordre de la signature résultante :

Définition 4.4 (Morphisme final)
Soit deux signatures $\Sigma = (S, TF, PF, P, \leq)$ *et* $\Sigma' = (S', TF', PF', P', \leq')$*. Le morphisme de signature* $\sigma : \Sigma \to \Sigma'$ *est final si et seulement si* $|\sigma|$ *est surjectif et si* $\leq'$ *est le plus petit préordre vérifiant :* $\sigma(s_1) \leq' \sigma(s_2)$ *si* $s_1 \leq s_2$.

Le morphisme final induit par une application de symboles vérifie deux propriétés : il doit correspondre à l'application de symboles sur leur domaine commun et valoir l'identité en dehors. La première propriété peut se formaliser comme suit :

Définition 4.5 (Filtrage de morphisme)
Soit $\sigma : \Sigma \to \Sigma'$ *un morphisme de signature et* $r \subseteq Sym \times Sym$ *une application de symboles. On dit que* σ filtre r*, noté* $\sigma \sqsubseteq r$*, si :*
- *Pour tout* $(RSY, RSY') \in r$*, il existe un* $SY \in |\Sigma|$ *tel que* $SY \sqsubseteq RSY$.
- *Pour tout* $(RSY, RSY') \in r$ *et* $SY \in |\Sigma|$, $SY \sqsubseteq RSY$ *implique* $|\sigma|(SY) \sqsubseteq RSY'$.

Nous dirons de plus que σ vaut l'identité en dehors de r si pour tout $SY \in |\Sigma|$ ne filtrant aucun RSY du domaine de r, alors $|\sigma|(SY) = SY_\sigma$, c'est-à-dire SY dont le profil est ajusté. Le morphisme final induit par une application de symboles est défini par :

Définition 4.6 (Morphisme induit)
Soit Σ *une signature et* $r \subseteq Sym \times Sym$ *une application de symboles. Un morphisme de signature final* $\sigma : \Sigma \to \Sigma'$ *filtrant* r *et valant l'identité en dehors est appelé le* morphisme induit par r sur Σ, *si* σ *est l'unique morphisme vérifiant ces propriétés. S'il existe, on le note* $r|^{\Sigma}$.

Soit Δ l'enrichissement de signature dénoté par SP par rapport à l'environnement local Σ, (SYs, r) l'ensemble de symboles et l'application déterminés par SM induisant un unique morphisme de signature final σ sur Σ' et ne renommant pas de symboles de Σ, on a les règles suivantes :

$$\frac{\begin{array}{c}|\Sigma'| \vdash \texttt{SYMB-MAP-ITEMS+} \rhd (SYs, r) \\ \sigma : \Sigma' \to \Sigma'' = r|^{\Sigma'} \text{ est défini}\end{array}}{\Sigma' \vdash \texttt{SYMB-MAP-ITEMS+} \rhd \sigma : \Sigma' \to \Sigma''} \quad (38)$$

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SPEC} \rhd \Delta : \Sigma \to \Sigma' \\ \Sigma' \vdash \texttt{SYMB-MAP-ITEMS+} \rhd \sigma : \Sigma' \to \Sigma'' \\ |\sigma| \text{ vaut l'identité sur } |\Sigma|\end{array}}{\Sigma, \Gamma_s \vdash \texttt{translation SPEC SYMB-MAP-ITEMS+} \rhd \Sigma''} \quad (39)$$

Soit $\mathcal{M}'$ la classe de modèles au dessus de Σ', alors la classe de modèles dénotée par SP **with** SM est l'ensemble des modèles sur Σ'' qui sont image d'un modèle de $\mathcal{M}'$ par σ :

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SPEC} \rhd \Delta : \Sigma \to \Sigma' \\ \Sigma' \vdash \texttt{SYMB-MAP-ITEMS+} \rhd \sigma : \Sigma' \to \Sigma'' \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}' \\ \mathcal{M}'' = \{M \in \mathbf{Mod}(\Sigma'') \mid M|_\sigma \in \mathcal{M}'\}\end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{translation SPEC SYMB-MAP-ITEMS+} \Rightarrow \mathcal{M}''} \quad (40)$$

4.2.9. Réduction

Pour une réduction SP **reveal** SM, les symboles de SP qui ne font pas partie du domaine de SM ne doivent pas faire partie de la signature finale. Il convient dans un premier temps de déterminer une signature à partir des symboles du domaine de SM, puis de renommer ces symboles. Cette signature est la plus petite sous-signature pleine basée sur cet ensemble de symboles.

Définition 4.7 (Réduction)
Soit $SYs \subseteq |\Sigma|$ un ensemble de symboles. Il existe une unique sous-signature pleine Σ' de Σ tel que $|\Sigma'|$ est le plus petit ensemble contenant SYs et fermé dans Σ. Σ' est appelée la signature générée par SYs et se note $\Sigma|_{SYs}$.

Soit $\Sigma = (S, TF, PF, P, \leq)$, SYs un ensemble de symboles, et SYs' l'union de SYs et des sortes apparaissant dans le profil d'une opération ou d'un symbole de prédicat qui n'étaient pas déjà dans SYs, alors $\Sigma|_{SYs} = (S', TF', PF', P', \leq \cap (S' \times S'))$ avec :

$$\begin{array}{c} S' = \{s | s^{sym} \in SYs'\} \\ TF'(ws) = \{f | f^{\mathtt{t}}_{ws} \in SYs'\} \\ PF'(ws) = \{f | f^{\mathtt{p}}_{ws} \in SYs'\} \\ P'(ws) = \{p | p_w \in SYs'\} \end{array}$$

Notons que si une opération est révélée, alors toutes les sortes apparaissant dans son profil sont également révélées.

Soit Δ l'extension de signature de l'environnement local courant Σ dénotée par SP, (SYs, r) l'application de symboles r et l'ensemble de symboles issus du filtrage de la signature de SP avec le domaine de définition de r déterminés par SM. r induit un unique morphisme de signature final σ sur la signature générée par $SYs \cup |\Sigma|$ (des symboles de l'environnement local peuvent être révélés) dans $\Sigma' = \Sigma \cup \Delta$. On a :

$$\frac{\begin{array}{c}|\Sigma'| \vdash \texttt{SYMB-MAP-ITEMS+} \rhd (SYs, r) \\ SYs \cap |\Sigma| = \emptyset \\ \Sigma_2 = \Sigma'|_{SYs \cup |\Sigma|} \\ \sigma : \Sigma_2 \to \Sigma'' = r|^{\Sigma_2} \text{ est défini}\end{array}}{\Delta : \Sigma \to \Sigma' \vdash \texttt{SYMB-MAP-ITEMS+} \rhd (\iota_{\Sigma_2 \subseteq \Sigma}, \sigma : \Sigma_2 \to \Sigma'')} \quad (41)$$

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SPEC} \rhd \Delta : \Sigma \to \Sigma' \\ \Delta : \Sigma \to \Sigma' \vdash \texttt{SYMB-MAP-ITEMS+} \rhd (\iota_{\Sigma_2 \subseteq \Sigma}, \sigma : \Sigma_2 \to \Sigma'')\end{array}}{\Sigma, \Gamma_s \vdash \texttt{reduction SPEC SYMB-MAP-ITEMS+} \rhd \Sigma_2} \quad (42)$$

Soit $\mathcal{M}'$ la classe de modèles au dessus de Σ'. La classe de modèles dénotée par SP **reveal** SM est l'ensemble des modèles au dessus de Σ_2 dont la translation par σ sont des modèles sur Σ' réduits à Σ_1.

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \texttt{SPEC} \rhd \Delta : \Sigma \to \Sigma' \\ \Delta : \Sigma \to \Sigma' \vdash \texttt{SYMB-MAP-ITEMS+} \rhd (\iota_{\Sigma_2 \subseteq \Sigma}, \sigma : \Sigma_2 \to \Sigma'') \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}' \\ \mathcal{M}_2 = \{M \in \mathbf{Mod}(\Sigma_2) \mid \exists M' \in \mathcal{M}', M'|_{\Sigma_1} = M|_{\Sigma}\}\end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{reduction SPEC SYMB-MAP-ITEMS+} \Rightarrow \mathcal{M}_2} \quad (43)$$

4.2.10. Référence

Comme toute spécification nommée est considérée comme étant une spécification générique, la référence à un nom de spécification correspond à une instanciation de cette spécification générique par des arguments. Nous avons donné dans la section 3.2.5 le cas de spécifications ne comportant aucun paramètre formel. Cela nous a permis, au cours de l'analogie de pouvoir illustrer le mécanisme de référence de CASL. Nous traitons maintenant le cas général.

Dans CASL, une instanciation d'une spécification générique $SN[SP_1] \dots [SP_n] = SP$ se note $SN[FA_1] \dots [FA_n]$, où chaque FA_i est la spécification d'un argument effectif possiblement ajusté par une application de symboles, SP'_i **fit** SM_i. Il existe plusieurs manières de concevoir l'instanciation comme une opération sur les signatures et les classes de modèles (voir [SST92] pour un survey). Dans CASL, cette opération est vue comme un produit fibré (*pushout* en anglais) entre les paramètres formels SP_i, le corps de la spécification générique SP, et les arguments effectifs SP'_i. Le produit fibré est une construction catégorique permettant de construire une spécification à partir de deux autres partageant une partie de leur signature possiblement renommée. Le paramètre formel correspond à la partie commune entre l'argument effectif et le corps, le renommage à l'ajustement de l'argument à la syntaxe du paramètre. La généralisation à plusieurs paramètres suit le même principe en remplaçant le corps par le résultat de l'instanciation des premiers paramètres.

Cependant, cette construction n'est définie qu'à isomorphisme près, ce qui fait perdre le contrôle sur les noms de symboles. En effet, si la signature du corps de la spécification générique et celle de l'argument partagent des symboles qui n'étaient pas dans le paramètre formel, alors ces symboles doivent être différenciés dans la signature résultante. Ceci est donc en contradiction avec le principe «même nom, même signification» utilisé dans CASL. Le choix qui est donc fait dans CASL est de construire l'instanciation comme une union entre la signature de l'argument et celle du corps de la spécification ajustée, puis de vérifier que cette union est bien un pushout. Comme pour la construction de morphisme ou de sous-signature, il existe plusieurs manières de construire cette union. Encore une fois, le choix le plus simple consiste à prendre la plus petite signature contenant les deux signatures. On appelle union *finale* cette union. Nous définissons cette notion avant d'introduire les règles pour le cas général de l'instanciation.

Étant donné deux signatures $\Sigma = (S, TF, PF, P, \leq)$ et $\Sigma' = (S', TF', PF', P', \leq')$, alors la signature $\Sigma'' = \Sigma \cup \Sigma'$ est définie par l'union composante par composante. La notion d'union finale est équivalente à une propriété de clôture sur les relations de surcharge des opérations et prédicats. Pour une signature $\Sigma = (S, TF, PF, P, \leq)$, deux symboles d'opération f_{w_1,s_1} et f_{w_2,s_2} sont dans la relation de surcharge (noté $f_{w_1,s_1} \sim_F f_{w_2,s_2}$) si et seulement si il existe un $w \in S^*$ et un $s \in S$ tels que $w \leq w_1, w_2$ et $s_1, s_2 \leq s$. De même pour $\sim_P$, la relation de surcharge des prédicats. Soit $\equiv_F$ et $\equiv_P$ les clôtures transitives de ces relations, on a :

Définition 4.8 (Union finale)
Une union $\Sigma_1 \cup \Sigma_2$ est finale si et seulement si la relation $\equiv_F^{\Sigma_1 \cup \Sigma_2}$ est la clôture transitive de l'union des relations $\equiv_F^{\Sigma_1}$ et $\equiv_F^{\Sigma_2}$ (de même pour $\equiv_P$).

Considérons pour l'instant le cas d'une spécification générique à un seul paramètre. Notons Σ_P la signature du paramètre formel, Δ l'extension de signature du corps d'une spécification générique et Σ_A la signature d'un argument. La première étape dans la construction de la signature de l'instanciation est de déterminer un morphisme de signature entre Σ_P et Σ_A à partir de l'application de symboles r issue de l'ajustement SM. Contrairement au cas d'un renommage, la signature du codomaine de ce morphisme est donnée. Il s'agit donc de construire un morphisme filtrant r de manière maximal sur son domaine, comme suit :

Définition 4.9 (Morphisme induit de Σ vers Σ')
Soit Σ et Σ' deux signatures et $r \subseteq Sym \times Sym$ une application de symboles. Alors r détermine un ensemble de morphismes σ tels qu'il existe un ensemble de symboles SYs avec :
- *σ filtre r et vaut l'identité sur SYs ;*
- *SYs est maximal selon la propriété que pour tout morphisme de signature θ, θ filtre r et vaut l'identité sur SYs.*

Si cet ensemble est réduit à un singleton, on appelle morphisme induit par r de Σ vers Σ' *ce singleton, noté* $r|_{\Sigma'}^{\Sigma}$.

Un argument d'une spécification générique est correct si le morphisme induit correspondant existe. On note σ ce morphisme. Le paramètre formel contient les symboles pouvant être partagés entre le corps de la spécification générique et un argument. Comme l'utilisation des symboles suit le principe «même nom, même signification», les symboles du paramètre doivent également être renommés au niveau du corps. Ce renommage est une adaptation de σ à la signature du corps. Il est construit en retrouvant une application de symboles à partir de σ grâce à une fonction *graph* définie par : pour tout f, $graph(f) = \{(x, f(x) \mid x \in Dom(f)\}$. Puis un morphisme induit est déterminé à partir de cette application de symboles sur Δ, que l'on note $\sigma(\Delta)$. Notons $\Sigma_A(\Delta)$ la signature du codomaine de $\sigma(\Delta)$. La signature résultant de l'instanciation est ensuite l'union finale entre Σ_A et $\Sigma_A(\Delta)$.

La propriété suivante explicite la relation entre cette union finale et un pushout :

Proposition 4.10

Si l'extension de $\sigma : \Sigma \to \Sigma_A$ *le long de* $\Delta : \Sigma \to \Sigma'$ *existe, si* $|\Sigma_A| \cap |\Sigma_A(\Delta)| \subseteq |\sigma|(|\Sigma|)$, *et si* $ker(|r|^{\Sigma'}|) \subseteq ker(|\sigma|)$ *alors le diagramme suivant est un diagramme de pushout.*

$$\begin{tikzcd}
\Sigma \arrow[r, "\Delta"] \arrow[d, "\sigma"'] & \Sigma' \arrow[d, dashed, "\sigma(\Delta)"] \\
\Sigma_A \arrow[r, dashed] & \Sigma_A \cup \Sigma_A(\Delta)
\end{tikzcd}$$

La définition précédente sous-entend que la spécification générique ne possède qu'un seul paramètre. Dans le cas d'une spécification générique ayant plus d'un paramètre, on se ramène au premier cas en construisant l'union des signatures des arguments et l'union σ des morphismes d'ajustement σ_i en suivant la définition :

Définition 4.11 (Compatibilité de morphismes)

Deux morphismes de signature $\sigma_1 : \Sigma_1 \to \Sigma_1'$ *et* $\sigma_2 : \Sigma_2 \to \Sigma_2'$ *sont compatibles si les applications de symboles sous-jacentes* $|\sigma_1|$ *et* $|\sigma_2|$ *coincident sur l'intersection* $|\Sigma_1| \cap |\Sigma_2|$.

Définition 4.12 (Union de morphismes)

Deux morphismes de signature $\sigma_1 : \Sigma_1 \to \Sigma_1'$ *et* $\sigma_2 : \Sigma_2 \to \Sigma_2'$ *définissent un morphisme de signature* $\sigma_1 \cup \sigma_2 : \Sigma_1 \cup \Sigma_2 \to \Sigma_1' \cup \Sigma_2'$ *si l'union* $\Sigma_1 \cup \Sigma_2$ *est finale et si* σ_1 *et* σ_2 *sont compatibles.*

Soit Σ la signature de l'environnement local, Σ_I la signature de l'union des imports, Δ_P l'extension de signature de $\Sigma \cup \Sigma_I$ dénotée par l'union des paramètres, et σ le morphisme d'ajustement de $\Sigma_I \cup \Delta_P$ vers $(\Sigma \cup \Sigma_I) \cup \Delta_A$, qui doit être égal à l'identité sur Σ_I. Δ_A est l'extension de la signature des imports dénotée par l'argument.

Soit une classe de modèles $\mathcal{M}$ au-dessus de Σ, une classe $\mathcal{M}_I$ au-dessus de Σ_I, $\mathcal{M}_P$ une classe au-dessus de $\Sigma_I \cup \Delta_P$ dont chaque modèle étend un modèle de $\mathcal{M}_I$, et $\mathcal{M}_A$ une classe de modèles au-dessus de $(\Sigma \cup \Sigma_I) \cup \Delta_A$ dont les modèles étendent un modèle de $\mathcal{M}$ et un modèle de $\mathcal{M}_I$. De plus, chaque modèle de $\mathcal{M}_A$ doit être le σ-réduit d'un modèle de $\mathcal{M}_P$.

$(\Sigma_I \cup \Delta_P, \mathcal{M}_P)$ est la sémantique d'un paramètre formel, et $((\Sigma \cup \Sigma_I) \cup \Delta_A, \mathcal{M}_A)$ celle de l'argument correspondant, avec σ pour morphisme d'ajustement reconstruit d'après la définition 4.9 . Les règles suivantes formalisent cette construction :

$$\frac{\begin{array}{c}\Sigma \cup \Sigma_I, \Gamma_s \vdash \texttt{SPEC} \rhd \Delta_A \\ |\Sigma_I \cup \Delta_P| \vdash \texttt{SYMB-MAP-ITEMS+} \rhd h \\ \sigma = h|^{(\Sigma\cup\Sigma_I)\cup\Delta_A}_{\Sigma_I\cup\Delta_P}\end{array}}{\Sigma, \Sigma_I, \Delta_P, \Gamma_s \vdash \texttt{fit-arg SPEC SYMB-MAP-ITEMS+} \rhd (\sigma, \Delta_A)} \quad (44)$$

$$\frac{\begin{array}{c}\Sigma \cup \Sigma_I, \Gamma_s \vdash \texttt{SPEC} \rhd \Delta_A \\ |\Sigma_I \cup \Delta_P| \vdash \texttt{SYMB-MAP-ITEMS+} \rhd h \\ \sigma = h|^{(\Sigma\cup\Sigma_I)\cup\Delta_A}_{\Sigma_I\cup\Delta_P} \\ \mathcal{M}' = \{M \in \mathbf{Mod}(\Sigma \cup \Sigma_I) \mid M|_\Sigma \in \mathcal{M}, M|_{\Sigma_I} \in \mathcal{M}_I\} \\ \Sigma \cup \Sigma_I, \mathcal{M}', \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}_A \\ \mathcal{M}_A|^\sigma \subseteq \mathcal{M}_P\end{array}}{\Sigma, \Sigma_I, \Delta_P, \mathcal{M}, \mathcal{M}_I, \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \texttt{fit-arg SPEC SYMB-MAP-ITEMS+} \Rightarrow \mathcal{M}_A} \quad (45)$$

Pour une instanciation, on vérifie que le nom de la spécification apparaît bien dans l'environnement, puis on utilise les règles de construction de la sémantique des arguments. La signature de l'instanciation est définie si et seulement si l'union finale de la signature de l'argument avec le corps ajusté est un pushout.

$$\frac{\begin{array}{c}\exists i \mid \Gamma_s[i] = \texttt{SPEC-NAME} \to (\Sigma_I, (\Delta_1 \ldots \Delta_n), \Delta_B) \\ \Sigma, \Sigma_I, \Delta_1, \Gamma_s \vdash \texttt{FIT-ARG} \rhd (\sigma_1, \Delta_1^A) \\ \ldots \\ \Sigma, \Sigma_I, \Delta_n, \Gamma_s \vdash \texttt{FIT-ARGn} \rhd (\sigma_n, \Delta_n^A) \\ (\Sigma_A \cup \Sigma_A(\Delta_B), \sigma_f(\Delta_B)) \text{ est défini}\end{array}}{\Sigma, \Gamma_s \vdash \texttt{spec-inst SPEC-NAME FIT-ARG1...FIT-ARGn} \rhd \iota_{\Sigma\subseteq(\Sigma_A\cup\Sigma_A(\Delta_B))}} \quad (46)$$

$$\frac{\begin{array}{c}\exists i \mid \Gamma_s[i] = \texttt{SPEC-NAME} \to (\Sigma_I, (\Delta_1 \ldots \Delta_n), \Delta_B) \\ \Sigma, \Sigma_I, \Delta_1, \Gamma_s \vdash \texttt{FIT-ARG1} \rhd (\sigma_1, \Delta_1^A) \\ \ldots \\ \Sigma, \Sigma_I, \Delta_n, \Gamma_s \vdash \texttt{FIT-ARGn} \rhd (\sigma_n, \Delta_n^A) \\ \Gamma_m[i] = \texttt{SPEC-NAME} \to (\mathcal{M}_I, (\mathcal{M}_1 \ldots \mathcal{M}_n), \mathcal{M}_B) \\ \Sigma, \Sigma_I, \Delta_1, \mathcal{M}, \mathcal{M}_I, \mathcal{M}_1, \Gamma_s, \Gamma_m \vdash \texttt{FIT-ARG1} \Rightarrow \mathcal{M}_1^A \\ \ldots \\ \Sigma, \Sigma_I, \Delta_n, \mathcal{M}, \mathcal{M}_I, \mathcal{M}_1, \Gamma_s, \Gamma_m \vdash \texttt{FIT-ARGn} \Rightarrow \mathcal{M}_n^A \\ \mathcal{M}' = \{M \in \mathbf{Mod}(\Sigma_A \cup \Sigma_A(\Delta_B)) \mid M|_{\Sigma_i^A} \in \mathcal{M}_i^A 1 \ \leq i \leq n,\ M|_{\sigma_f} \in \mathcal{M}_B\}\end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{spec-inst SPEC-NAME FIT-ARG1...FIT-ARGn} \Rightarrow \mathcal{M}'} \quad (47)$$

4.3. Redéfinition par extension

Cet exposé des règles de la sémantique des opérateurs de structuration de CASL nous a éloigné de l'objectif initial de ce chapitre. Rappelons la situation dans laquelle nous nous plaçons. Nous considérons une bibliothèque de spécification $\mathcal{L}$, et deux spécifications nommées SN et SM bien formées déclarées dans $\mathcal{L}$, telles que SM utilise SN par un opérateur de structuration **op**. Nous supposons ensuite que SN est redéfinie par une extension avec une spécification bien formée SPO.

L'intuition derrière notre optimisation est la suivante : sachant que SM avant la propagation et SPO sont bien formées, les conditions de bonne formation de SM après la propagation peuvent se déduire, dans la plupart des cas, de la bonne formation de ces deux spécifications. La bonne formation des spécifications est donc conservée au cours de la propagation et n'a donc pas besoin d'être vérifiée par les règles statiques, sauf dans le cas où **op** est une instanciation, ou lorsque SN est utilisé comme paramètre formel par SM. De plus, lorsqu'on réévalue la sémantique statique de SM, une partie correspond à l'ancienne sémantique de SM dans Γ_s et l'autre partie à l'évaluation de l'extension dénotée par SPO sous **op**, c'est à dire *op*(SPO), avec *op* l'opération correspondante sur la signature. Si on note Δ_O l'extension de signature dénotée par SPO, et les nouvelles évaluations après la redéfinition par un $\_^+$,la nouvelle sémantique de SM, Σ_M^+, est donnée par une formule utilisant Σ_M, Δ_O, et *op*.

Dans la suite de cette section, nous allons définir formellement cette optimisation, en proposant une règle sémantique particulière qui sera traduite dans le pseudo-code. Cette règle traite du cas général de la propagation d'une redéfinition d'une spécification SN par une extension SPO. La sémantique statique de toutes les spécifications nommées SM faisant référence à SN, et qui ont déjà été évaluées, doit donc être modifiée. Nous montrerons ensuite que cette règle est correcte : la sémantique évaluée par cette règle est identique à celle qui aurait été évaluée sans cette optimisation. Les cas particuliers où la bonne formation ne peut être conservée sera traité en détail à la fin de cette section.

4.3.1. Règle sémantique

Nous formalisons ce cas particulier par une règle sémantique appropriée pour la sémantique statique.

Bonne formation. Comme pour une redéfinition quelconque, cette règle doit conserver la propriété de visibilité linéaire de l'environnement. Dans le cas général, l'expression de spécification utilisée dans une redéfinition est quelconque, et la spécification redéfinie peut être paramétrée. La condition de bonne formation porte alors sur le corps ainsi que sur ces paramètres. Dans le cas présent, le corps de SN a une forme particulière et ne comporte pas de paramètres.

La condition de bonne formation doit empêcher les références indirectes afin de ne pas créer de cycles dans le graphe de dépendances de la bibliothèque. Les références directes sont en revanche autorisées. Puisque l'on ne traite maintenant que des expressions de spécifications de la forme SN **then** SPO pour le corps de la redéfinition de SN, nous considérons que la seule référence directe à SN est celle apparaissant explicitement dans cette expression. Par conséquent, ils suffit d'interdire la présence de références à SN dans SPO pour assurer la bonne formation :

$$SN \notin Ref^{+}_{\mathcal{L}}(SPO)$$

Propagation. Une fois l'extension de signature Δ_O dénotée par SPO évaluée, la signature de SN est mise à jour dans l'environnement courant par la fonction de substitution $Subst$ que nous avons définie dans le chapitre précédent. Cette mise à jour est ensuite propagée dans cet environnement pour toutes les spécifications SM faisant référence à SN. L'optimisation consiste à réutiliser la signature Σ_M de SM issue de l'environnement et d'y adjoindre une extension correspondant à Δ_O que l'on a ajustée au préalable selon les opérateurs de structuration utilisés dans chaque SM. Nous ne nous occupons pour le moment que des cas où la bonne formation peut être conservée. La référence à SN ne doit donc pas se faire dans les parties génériques des spécifications que l'on veut mettre à jour, ni dans le paramètre effectif d'une instanciation utilisée dans le corps.

La fonction de propagation optimisée prend en argument une liste de noms de spécification dont la sémantique a déjà été mise à jour, la bibliothèque de syntaxe abstraite, l'extension de signature Δ_O, l'environnement statique courant, et rend un environnement dans lequel les signatures des spécifications SM faisant référence à SN ont été mises à jour :

$$\begin{array}{l} Propage\_\_opt\_\_accordingto\_\_with\_\_in\_\_ : \\ \qquad List[SpecName] \times Library \times Signature \times Environment \rightarrow Environement \end{array}$$

La propagation consiste à parcourir l'environnement et à tester si l'expression de spécification associée au corps de l'association fait référence à une spécification dans la liste passée en argument, puis à calculer la nouvelle signature de l'expression de spécification du corps.

Cette signature est donnée par l'union de l'ancienne signature conservée dans l'environnement avec une extension calculée selon les opérateurs de structuration utilisés dans cette expression de spécification et l'extension de signature Δ_O. Nous définissons une fonction *Eval_Delta* évaluant cette extension. Cette fonction prend en argument l'expression de spécification du corps, l'extension de signature Δ_O, la liste des noms de spécification dont la sémantique a été mise à jour, et la bibliothèque de syntaxe abstraite, et rend l'extension à ajouter à la signature de départ pour obtenir la signature mise à jour. :

$$Eval\_Delta : SpecExpr \times Signature \times List[SpecName] \times Library \rightarrow Signature$$

Cette fonction effectue une récurrence structurelle sur l'expression de spécification passée en argument et détermine l'extension de spécification à ajouter pour obtenir la mise à jour de la signature de l'expression. Si l'expression est une spécification de base, l'extension est vide. S'il s'agit d'une extension de sous-expressions, l'extension à ajouter est formée de l'union des extensions déterminés par chaque sous-expression. De même dans le cas d'une union. Dans le cas de la restriction d'une sous-expression, l'extension est donnée par l'extension à ajouter pour cette sous-expression ne contenant pas les symboles spécifiés. Dans le cas du renommage d'une sous-expression, l'extension à ajouter est l'image de celle déterminée par la sous-expression dont on aura renommé les symboles comme indiqué. Le cas de la réduction d'une sous-expression est donné par l'extension à ajouter pour cette sous-expression contenant seulement les symboles spécifiés, qui peuvent être possiblement renommés. Dans le cas d'une instanciation non générique (référence), soit il s'agit de la spécification redéfinie au départ et dans ce cas l'extension à ajouter est celle passée en argument, soit il s'agit d'une référence à un nom quelconque et dans ce cas l'extension à ajouter est celle calculée pour cette spécification nommée. Nous ne traitons pas le cas d'une instanciation générique.

Nous donnons plus loin le pseudo-code de cette fonction.

Règle. La règle suivante formalise le cas d'une redéfinition par extension. Cette règle est écrite pour les catégories syntaxiques de la grammaire formelle de CASL : elle concerne un nom de spécification quelconque SN de la catégorie `SPEC-NAME` et une expression quelconque SPO de la catégorie `SPEC` dans une bibliothèque quelconque $\mathcal{L}$. Comme pour une redéfinition quelconque, la spécification SN que l'on redéfinit doit déjà figurer dans l'environnement. Soit Σ la signature du corps de cette spécification avant la redéfinition. Pour que la redéfinition soit correcte, il faut interdire les circularités indirectes (voir la règle (8) du chapitre 3). Contrairement au cas général, comme nous connaissons la forme de la redéfinition, il suffit de faire porter cette condition sur l'extension SPO : cette expression de spécification ne doit pas faire référence au nom de la spécification redéfinie. L'extension de signature Δ_O dénotée par SPO est ensuite calculée dans l'environnement local contenant la signature Σ. D'après la sémantique de l'extension, la nouvelle signature du corps, Σ^+ est l'union de l'ancienne signature et de cette extension : $\Sigma^+ = \Sigma \cup \Delta_O$. La sémantique de la spécification nommée est mise à jour puis propagée optimalement dans l'environnement situé après l'association correspondante.

$$\frac{\begin{array}{c}\texttt{SPEC-NAME} \to (\emptyset, (), \Sigma) \in \Gamma_s \\ \texttt{SPEC-NAME} \notin Ref^+_{\mathcal{L}}(\texttt{SPEC}) \\ \Sigma, \Gamma_s \vdash \texttt{SPEC} \rhd \Delta_O : \Sigma \to \Sigma^+ \\ \Gamma'_s = Subst(\texttt{SPEC-NAME} \to (\emptyset, (), \Sigma^+))\ in\ \Gamma_s(until\ \texttt{SPEC-NAME}) :: \\ Propage\_opt\ \{\texttt{SPEC-NAME}\}\ accordingto\ \mathcal{L}\ with\ \Delta_O\ in\ \Gamma_s(from\ \texttt{SPEC-NAME})\end{array}}{\Gamma_s \vdash \texttt{spec-redefn SPEC-NAME (extension (spec-inst SPEC-NAME) SPEC)} \rhd \Gamma'_s} \quad (48)$$

Pseudo-code. Nous pouvons alors étendre le pseudo-code donné dans le chapitre précédent. Le cas d'une redéfinition par extension est inséré avant celui d'une redéfinition quelconque lors de l'évaluation d'une bibliothèque afin de pouvoir y appliquer l'optimisation le cas échéant. La propagation fait appel à la fonction `Eval_Delta` lorsque l'expression associée au corps d'une association fait référence à un nom situé dans la liste passée en argument. Si cette expression comporte une instanciation, nous utilisons la fonction d'évaluation classique `Eval_sig`. La détection d'un opérateur d'instanciation dans la structure de cette expression se fait par le biais d'une exception que l'on lève dans la fonction `Eval_Delta`. `Symb` prend en argument la déclaration d'une liste de symboles ou d'une application de symboles et retourne la liste ou l'application correspondante. `sigma`, `cogen`, `gen` sont les opérations sur les signatures correspondants aux opérateurs de renommage, restriction et réduction. Dans ce pseudo-code, nous simplifions les appels récursifs à `Eval_Delta` en ne mentionnant pas les arguments qui restent constants d'un appel à l'autre.

```
Eval_sig_lib_opt  L =
    let G = Nil in
     for i=0 to (Length L) do
        /* Declaration */
       if (L[i] = spec-defn SN PAR IMP SP) ...
        /* Redefinition par extension */
       if (L[i] = spec-redefn SN  (extension SN SPO)) &
                   (SN in G) & not(SN in (Ref+ L SPO)) &  then
            S = Assoc(SN,G);
            Delta_O = (Eval_sig G SPO S) \ S;
            G = Cons(Subst (SN->(0,(),S U Delta_O)) in (G until SN),
                     Propage_opt [SN] accordingto L with Delta_O
                       in (G from SN));
        /* Autres cas de redefinition*/
       if (L[i] = spec-redefn SN PAR IMP SP) ...


Propage_opt l accordingto L with D in G =
```

```
  if G = Nil then Nil
  else /* G est de la forme  Cons(SN->(S1,S2,S3), G') */
    if   (Ref(Expr_corps SN L) in l ) &
      not(Ref(Expr_imports SN L) in l) &
      not(Ref(Expr_params SN L) in l) &
    then
       try
         S3+ =  S3 U (Eval_Delta (Expr_corps SN L) D l  L)
        with Inst -> S3+ = Eval_sig G (Expr_corps SN L) (Union S2);
        Cons(SN->(S1,S2,S3+),
             Propage_opt Cons(l,SN) accordingto L with D in G'));
    else  Cons(SN->(S1,S2,S3),
               Propage_opt l accordingto L with D in G')


Eval_delta SP D l L=
    match SP with
     | basic-spec SP'-> ()
     | extension SP1...SPn -> (Eval_delta SP1) ... U (Eval_delta SPn)
     | union SP1...SPn -> (Eval_delta SP1) ... U (Eval_delta SPn)
     | restriction SP' SL -> cogen(Eval_delta(SP'),Symb(SL))
     | renaming SP' SM -> sigma(Eval_delta(SP'))
     | reduction SP' SM -> sigma(gen(Eval_delta(SP'),Symb(SM)))
     | spec-inst X -> if (X = Head(l)) then D
                     else Eval_delta (Expr X L)
     | spec-inst Y FA1...FAn -> raise Inst;
```

Correction de l'optimisation

Pour montrer que cette optimisation est correcte, nous devons montrer que la sémantique calculée par `Eval_sig_lib_opt` est identique à celle que l'on aurait calculée par `Eval_sig_lib`. Ceci est l'objet des sections suivantes.

Soit $\mathcal{L}$ une bibliothèque de syntaxe abstraite. Notons $[\![\mathcal{L}]\!]_s$ sa sémantique statique évaluée par la fonction `Eval_sig_lib`, et $[\![\mathcal{L}]\!]_s^{opt}$ sa sémantique évaluée par la fonction `Eval_sig_lib_opt`. Nous voulons donc démontrer le théorème suivant :

Théorème 4.13 *Pour toute bibliothèque de syntaxe abstraite bien formée $\mathcal{L}$, $[\![\mathcal{L}]\!]_s = [\![\mathcal{L}]\!]_s^{opt}$*

La preuve de ce théorème s'effectue par récurrence sur la longueur de la bibliothèque $\mathcal{L}$. Le seul cas différenciant les deux fonctions d'évaluation de la sémantique est celui d'une redéfinition d'une spécification non générique SN par extension SPO. Nous devons montrer plusieurs choses : d'abord qu'une redéfinition par extension ne crée pas de cycle dans le graphe de dépendance associé à cette bibliothèque, puis que les fonctions de propagation sont équivalentes. Le premier point se déduit directement de la condition de bonne formation d'une redéfinition par extension. Pour le second, nous devons montrer que les environnements calculés par les fonctions de propagation et de propagation optimisée sont équivalents. Cela signifie que si le premier était bien formé, alors le second l'est aussi, et que la signature des spécifications mises à jour se calcule bien comme l'union de l'ancienne signature avec une extension issue de celle dénotée par SPO. Nous démontrons deux lemmes pour chacune de ces propriétés.

4.3.2. Conservation de la bonne formation au cours de la propagation

L'objet de cette section est de démontrer le lemme suivant :

Lemme 4.14
Étant donné une bibliothèque $\mathcal{L}$ bien formée et son environnement statique Γ_s, la propagation dans Γ_s de la redéfinition d'une spécification nommée SN par extension SPO bien formée conserve la bonne formation des spécifications déjà évaluées, lorsque celles-ci ne contiennent pas d'instanciation générique.

La démonstration de ce lemme se fait par induction sur la longueur de l'environnement Γ_s, puis, pour chaque association, par une induction structurelle sur la dernière expression de spécification associée dans $\mathcal{L}$. Cette démonstration utilise plusieurs fois la propriété suivante, que nous démontrons dans un premier temps :

Proposition 4.15
Pour toute expression de spécification structurée SP bien formée évaluée dans un environnement global Γ_s et un environnement local Σ, SP reste bien formée après la propagation de la redéfinition d'une spécification SN par une extension SPO.

Preuve: Nous effectuons cette preuve par récurrence sur la structure de SP :

- SP est une spécification de base, elle ne contient donc aucune référence à SN. Sa bonne formation est conservée ;
- SP est une extension de la forme SP_1 **then**... **then** SP_n. D'après la règle (10), la bonne formation d'une extension est donnée par la bonne formation de ses composantes. Celles-ci restent bien formées d'après l'hypothèse d'induction, donc SP reste bien formée.
- SP est une union de la forme SP_1 **and** ... **and** SP_n. D'après la règle (12), la bonne formation d'une union est donnée par la bonne formation de ses composantes. Celles-ci restent bien formées d'après l'hypothèse d'induction, donc SP reste bien formée.
- SP est une restriction de la forme SPR **hide** SL, où SL est la déclaration d'une liste de symboles telles que nous les avons définies dans la section 4.2.5. D'après la règle (29), cette expression est bien formée si les symboles dénotés par SL dans l'ensemble

des symboles de la signature mise à jour de SPR n'intersectent pas de symboles dans l'environnement local $|\Sigma|$.
Par hypothèse d'induction, l'expression de signature SPR reste bien formée. Notons Σ_R^+ sa signature mise à jour et SYs^+ le nouvel ensemble de symboles déterminé par SL dans l'environnement $|\Sigma_R^+|$. D'après la règle (28), SL détermine le même ensemble de symboles bruts $RSYs$, puisque cet ensemble est calculé dans un environnement vide. En revanche, lors de la restauration du profil des symboles non qualifiés, de nouveaux symboles peuvent apparaître du fait de la modification de l'environnement. Si des symboles de Σ_R^+ provenant de la propagation de la redéfinition de SN sont basés sur un identificateur non qualifié présent dans $RSYs$, alors ces symboles seront contenus dans l'ensemble de symboles qualifié calculé par la règle (28). En effet, si un symbole contenu dans la liste SL n'est pas qualifié, alors tous les symboles basés sur le même identificateur sont listés. Notons S_Δ cet ensemble. On a donc $SYs^+ = SYs \cup S_\Delta$.
Pour que SP reste bien formée après la propagation, il suffit donc de montrer que $SYs^+ \cap |\Sigma| = \emptyset$. Par hypothèse, SP était bien formée avant la redéfinition de SN, donc $SYs \cap |\Sigma| = \emptyset$. La propriété suivante montre la deuxième partie de la condition :

Proposition 4.16 *Si $SYs \cap |\Sigma| = \emptyset$, alors $S_\Delta \cap |\Sigma| = \emptyset$.*

Preuve:
Par l'absurde : supposons qu'il existe un symbole qualifié f_{ws} commun à S_Δ et à $|\Sigma|$. $f_{ws} \in S_\Delta$ implique que l'identificateur f est présent dans la liste SL comme un symbole non qualifié. Et comme SP était bien formée, il existe forcément un symbole de $|\Sigma_R|$ ayant f comme identificateur contenu dans SYs. De plus, comme $\Sigma \subseteq \Sigma_R$, $f_{ws} \in \Sigma$ signifie $f_{ws} \in \Sigma_R$ et donc que f_{ws} était aussi listé dans SYs. Donc $f_{ws} \in SYs \cap \Sigma$. Or par hypothèse $SYs \cap |\Sigma| = \emptyset$, il y a donc une contradiction. □
La bonne formation de SP est donc conservée.

- SP est un renommage de la forme SPR **with** SM, où SM est la déclaration d'une application de symboles (voir section 4.2.7). D'après la règle (38), cette déclaration est bien formée si le morphisme induit par SM sur la signature de SPR, est bien défini, ce qui signifie qu'il est l'unique morphisme final prolongeant l'application de symboles déterminée par SM (voir la définition 4.6). Ensuite, la règle (39) nous donne une autre condition de bonne formation : que ce morphisme ne renomme pas de symboles de l'environnement local, c'est à dire de Σ. Cette seconde condition est proche de celle étudiée dans le cas de la restriction.
Par hypothèse d'induction, SPR est une expression de spécification bien formée. Notons (SYs^+, r^+) l'ensemble de symboles et l'application de symboles déterminée par SM dans l'environnement $|\Sigma_R^+|$. D'après la règle (37), l'application de symboles n'est pas modifiée puisqu'elle est évaluée dans un environnement vide. En revanche, comme pour le cas précédent, SYs^+ est composé de l'ancien ensemble SYs et de l'ensemble des symboles de Σ_R^+ basés sur un identificateur présent dans le domaine de r en tant que symbole non qualifié. Comme précédemment, $SYs^+ = SYs \cup S_\Delta$.
Pour que SP reste bien formée, il faut donc dans un premier temps montrer qu'il

existe un unique morphisme final prolongeant r sur la signature Σ_R^+. Soit $\sigma^+ : \Sigma_R^+ \to \Sigma_P^+$ le morphisme surjectif, prolongeant $\sigma = r|^{\Sigma_R}$, défini comme suit :

$$\sigma^{+S}(s) = \begin{cases} \sigma^S(s) & si\ s \in SYs^+ \\ s & sinon \end{cases} \qquad \sigma^{+TF}_{ws}(f) = \begin{cases} \sigma^{TF}_{ws}(f)^{\mathsf{t}}_{\sigma^S(ws)} & si\ f^{\mathsf{t}}_{ws} \in SYs^+ \\ f^{\mathsf{t}}_{\sigma^S(ws)} & sinon \end{cases}$$

$$\sigma^{+PF}_{ws}(f) = \begin{cases} \sigma^{PF}_{ws}(f)^{\mathsf{p}}_{\sigma^S(ws)} & si\ f^{\mathsf{p}}_{ws} \in SYs^+ \\ f^{\mathsf{p}}_{\sigma^S(ws)} & sinon \end{cases} \qquad \sigma^{+P}_{w}(p) = \begin{cases} \sigma^{P}_{w}(p)_{\sigma^S(w)} & si\ p_w \in SYs^+ \\ p_{\sigma^S(w)} & sinon \end{cases}$$

On associe le préordre de sous-sortage suivant à la signature source de ce morphisme : $\leq_P^+ = \sigma^+(\leq_R^+)$. La propriété suivante assure que σ^+ est l'unique morphisme final prolongeant σ vers Σ_P^+ :

Proposition 4.17 $\sigma^+ = r|^{\Sigma_R^+}$

Preuve:

1. σ^+ est un morphisme. Cela se déduit directement de sa définition par le fait que σ est un morphisme.
2. σ^+ est final : $|\sigma^+|$ est surjectif car σ^+ l'est, et $\leq_P^+$ est le plus petit préordre vérifiant $s_1 \leq_R^+ s_2 \Rightarrow \sigma^+(s_1) \leq \sigma^+(s_2)$. En effet soit $\leq^+ \subseteq \leq_P^+$ un autre préordre. Alors $s_1^1 \leq^+ s_2^1$ implique par surjectivité qu'il existe s_1 et s_2 dans Σ_R^+ tels que $s_1^1 = \sigma^+(s_1)$ et $s_2^1 = \sigma^+(s_2)$. $\leq_R^+$ est composé de la fermeture transitive de l'union de $\leq_R$ et du préordre issu de l'extension ajoutée à Σ_R pour sa mise à jour, noté $\leq_\Delta$. Si on a $s_1 \leq_R s_2$, après application de σ^+, on a $s_1^1 \leq_B^+ s_2^1$. De même si $s_1 \leq_\Delta s_2$ ou si on se trouve dans le cas de la transitivité. On applique ensuite la propriété 4.4.
3. σ^+ filtre r : pour tout $(RSY_1, RSY_2) \in r$, il existe $SY \in SYs \subseteq SYs^+$ qui filtre RSY_1, car par hypothèse σ filtre r ; pour tout $(RSY_1, RSY_2) \in r$ et $SY \in |\Sigma_R^+|$ filtrant RSY_1 (donc $SY \in SYs^+$), on a $|\sigma^+|(SY) = |\sigma|(SY)$, par définition, qui filtre RSY_2 par hypothèse ; σ^+ est l'identité en dehors de r (c'est à dire que σ^+ ne modifie pas la partie nue d'un symbole ne se trouvant pas dans SYs^+) par définition de σ^+.
4. σ^+ est l'unique morphisme surjectif vérifiant ces propriétés. Soit σ_2^+ un autre morphisme, on a :
   - si $SY \in SYs^+$, soit RSY le symbole qui filtre SY et RSY' tel que $(RSY, RSY') \in r$. Alors $|\sigma_2^+|$ filtre RSY' implique que $|\sigma_2^+|(SY) = |\sigma^+|(SY)$, puis par construction de $|\_|$, $\sigma_2^+(SY) = \sigma^+(SY)$.
   - si $SY \notin SYs'$, alors $|\sigma_2^+|(SY) = SY$, soit $\sigma_2^+(SY) = \sigma^+(SY)$.

$\square$

La seconde condition pour que SP reste bien formée est que $|\sigma^+|$ soit égal à l'identité sur $|\Sigma|$. Or par hypothèse, $|\sigma|$ vaut l'identité sur $|\Sigma|$. Il suffit ensuite de montrer que les nouveaux symboles du domaine de σ^+ n'intersectent pas de symboles de l'environnement local, ce qui est assuré par la propriété 4.16.

- SP est une réduction de la forme SPR **reveal** SM, où SM est la déclaration d'une application de symboles. La bonne formation de SP combine les conditions d'une restriction et d'un renommage (voir règle (41)) : d'une part les symboles qualifiés de Σ_R^+ filtrant ceux du domaine de l'application de symboles, notés SYs^+ doivent être disjoints de ceux de l'environnement local ; d'autre part, le morphisme induit par SM sur la plus petite signature contenant les symboles de SYs^+ est bien défini (il est l'unique morphisme final prolongeant l'application de symboles déterminée par SM sur Σ_R^+).
  Par hypothèse d'induction, SPR est bien formée. Comme dans le cas d'un renommage, l'ensemble de symboles et l'application de symboles dénotés par SM dans $|\Sigma_R^+|$ sont donnés par $(SYs \cup S_\Delta, r)$. La première condition à vérifier pour que SP reste bien formée est que les symboles dénotés par SM ne soient pas dans l'environnement local : $(SYs \cup S_\Delta) \cap |\Sigma| = \emptyset$. Ceci est assuré par la propriété 4.16.
  La seconde condition de bonne formation est qu'il n'existe qu'un unique morphisme final prolongeant r sur la plus petite sous-signature de Σ_R^+ contenant les symboles du domaine de r, notée Σ_2^+. Soit $\sigma^+ : \Sigma_2^+ \to \Sigma_P^+$ le morphisme surjectif, prolongeant $\sigma = r|^{\Sigma_2}$, défini comme pour le renommage. Alors par la propriété 4.17 ce morphisme convient. SP reste donc bien formée dans ce cas également.
- SP est une référence simple de la forme SNX. Une référence simple à un nom de spécification SNX est bien formée si et seulement si il existe une association dans l'environnement global Γ_s pour ce nom. Par hypothèse, SP était bien formée avant la redéfinition de SN, donc SNX est bien référencé dans Γ_s. SP reste bien formée.

□

Preuve du Lemme

Nous pouvons effectuer la preuve du lemme 4.14 à partir de la propriété précédente.
Preuve: La preuve du lemme se fait par une induction sur la longueur de l'environnement Γ_s puis, pour chaque association dans Γ_s, par une induction structurelle sur l'expression de spécification la plus récente associée dans $\mathcal{L}$.

Soit SM le nom d'une spécification dans Γ_s. Comme ce nom de spécification peut apparaître plusieurs fois dans la bibliothèque à cause de redéfinitions éventuelles, son expression de spécification associée est la dernière rencontrée avant la redéfinition de SN. Cette expression de spécification peut être de deux formes :

– Soit il s'agit d'une déclaration de spécification de la forme **spec** $SM[PM]$ **given** $IM = SPM$. Les conditions de bonne formation de cette expression sont données par la règle (3). Il faut que le nom SM n'apparaisse pas dans l'environnement global, tel qu'il était au moment de cette déclaration, c'est-à-dire $\Gamma_s(\mathit{until}\ SM)$. La redéfinition de SN ne modifie pas cette condition. Ensuite, la bonne formation dépend de la bonne formation de la partie générique et du corps. Comme les imports et les paramètres ne contiennent pas de référence à SN, par hypothèse, leur bonne formation est conservée. La dernière condition de bonne formation de SM est celle concernant la bonne

formation de l'expression de spécification SPM dans un environnement local non modifié. Cette condition est démontrée par la propriété précédente.
- Soit il s'agit d'une redéfinition de la forme **spec** $SM[PM]$ **given** $IM \overset{\text{re}}{=} SPM$. Les conditions de bonne formation sont symétriques du cas précédent. La présence d'une association pour SM dans $\Gamma_s(\mathit{until}\ SM)$ n'est pas modifiée par la redéfinition de SN. La bonne formation de l'expression de spécification SPM est prouvée par la propriété précédente.

□

4.3.3. Formules de propagation

Le lemme suivant montre la correction de la fonction `Eval_Delta` : la mise à jour de la signature d'une expression de spécification SP est bien donnée par l'union de la signature avant la propagation avec une extension issue de l'extension dénotée par SPO.

Lemme 4.18
Soit $\mathcal{L}$ une bibliothèque, et Γ_s l'environnement contenant la sémantique des spécifications déjà évaluées. La mise à jour de l'extension de signature Δ dénotée par une expression de spécification SP après la propagation de la redéfinition d'une spécification nommée SN par une extension SPO dans un environnement local Σ est donnée par :

$$\Delta^+ = \Delta \cup Eval\_Delta(SP, \Delta_O)$$

où Δ_O est l'extension de signature dénotée par SPO.

Propriétés préliminaires

Pour démontrer ce lemme, nous aurons besoin des propriétés suivantes des opérations sur les signatures :

Proposition 4.19 *Soit Σ_1, Σ_2, et Σ_3 trois signatures, SYs un ensemble de symboles inclus dans $\Sigma_1 \cup \Sigma_2$, et σ un morphisme de signature sur $\Sigma_1 \cup \Sigma_2$. Les relations suivantes sont vraies :*

1. $\Sigma_1 \cup \Sigma_2 = \Sigma_2 \cup \Sigma_1$
2. $(\Sigma_1 \cup \Sigma_2) \cup \Sigma_3 = \Sigma_1 \cup (\Sigma_2 \cup \Sigma_3)$
3. $(\Sigma_1 \cup \Sigma_2)|^{SYs} = \Sigma_1|^{SYs} \cup \Sigma_2|^{SYs}$
4. $(\Sigma_1 \cup \Sigma_2)|_{SYs} = \Sigma_1|_{SYs} \cup \Sigma_2|_{SYs}$
5. $\sigma(\Sigma_1 \cup \Sigma_2) = \sigma(\Sigma_1) \cup \sigma(\Sigma_2)$

Preuve:

1. La première égalité est une conséquence triviale de la construction de l'union de deux signatures. Celle-ci est construite composante par composante, puis par une union ensembliste sur chacune de ces composantes qui possède la propriété de commutativité. Notons que l'union des préordres est donnée par la clôture transitive de ces préordres : si $\leq_1$ est le préordre de Σ_1 et $\leq_2$ celui de Σ_2, alors le préordre de $\Sigma_1 \cup \Sigma_2$ est $(\leq_1 \cup \leq_2)^*$.
2. De même pour la propriété d'associativité.
3. Soit SYs un ensemble de symboles dans $|\Sigma_1 \cup \Sigma_2|$. Notons SYs' l'union de SYs et de l'ensemble des symboles de fonctions ou de prédicats de $|\Sigma_1 \cup \Sigma_2|$ dont l'une des sortes du profil est dans SYs. De même pour SYs'_1 ceux de $|\Sigma_1|$ et SYs'_2 ceux de $|\Sigma_2|$. On a $SYs' = SYs'_1 \cup SYs'_2$, car tout symbole de SYs est soit dans $|\Sigma_1|$ soit dans $\Sigma_2|$, et que les ensembles SYs', SYs'_1 et SYs'_2 sont fermés. Donc $(|\Sigma_1 \cup \Sigma_2| \setminus SYs' = (|\Sigma_1| \setminus SYs'_1) \cup (|\Sigma_2| \setminus SYs'_2)$. D'après la définition 4.3, cela nous donne la propriété escomptée.
4. Un raisonnement similaire démontre cette propriété.
5. La dernière propriété est également une conséquence directe de la construction d'une union de signature. L'union de morphisme du second membre définit bien un morphisme selon la définition 4.12 car il s'agit de l'union du même morphisme sur les deux sous-parties de son domaine.

□

Preuve du Lemme 4.18

Preuve: Comme pour le lemme précédent, la preuve se fait par induction sur la structure de l'expression de spécification SP.

- SP est une spécification de base, qui n'est donc pas concernée par la redéfinition de SN. Sa signature est identique, donc $Eval\_Delta(SP, \Delta_O) = \emptyset$.
- SP est une extension de la forme SP_1 **then** ... **then** SP_n. Par hypothèse de récurrence, chaque SP_i dénote une extension de signature de la forme $\Delta_i \cup Eval\_Delta(SP_i, \Delta_O)$ dans un environnement local croissant. En utilisant la commutativité de l'opération d'union de signatures, l'extension de signature dénotée par SP devient :

$$\begin{aligned}
\Delta^+ &= (\Delta_1 \cup Eval_Delta(SP_1, \Delta_O)) \cup \ldots \cup (\Delta_n \cup Eval_Delta(SP_n, \Delta_O)) \\
&= (\Delta_1 \cup \Delta_2 \ldots \cup \Delta_n) \cup (Eval_Delta(SP_1, \Delta_O) \ldots \cup Eval_Delta(SP_n, \Delta_O)) \\
&= \Delta \cup (Eval_Delta(SP_1, \Delta_O) \ldots \cup Eval_Delta(SP_n, \Delta_O))
\end{aligned}$$

- SP est une union de la forme SP_1 **and** ... **and** SP_n. Par hypothèse de récurrence, chaque SP_i dénote une extension de signature de la forme $\Delta_i \cup Eval\_Delta(SP_i, \Delta_O)$ dans un environnement local fixe. La nouvelle signature de SP est identique au cas précédent.

- SP est une restriction de la forme SPR **hide** SL. Par hypothèse de récurrence, SPR dénote une signature Σ_R^+ de la forme $\Sigma_R \cup Eval\_Delta(SPR, \Delta_O)$. D'après la définition 4.3, la signature de SP mise à jour est la plus grande sous-signature pleine de Σ_R^+ ne contenant pas de symboles dans l'ensemble déterminé par SL dans l'environnement $|\Sigma_R^+|$. Dans la preuve du lemme 4.14, cet ensemble a été noté $SYs^+ = SYs \cup S_\Delta$. En utilisant la propriété de distributivité de l'opération de cogénération sur l'union, on a donc :

$$\begin{array}{rcl} \Delta^+ & = & (\Sigma_R \cup Eval\_Delta(SPR, \Delta_O))|^{SYs^+} \\ & = & \Sigma_R|^{SYs^+} \cup Eval\_Delta(SPR, \Delta_O)|^{SYs^+} \end{array}$$

  Par définition, les nouveaux symboles de SY^+ qui n'étaient pas dans SYs sont des symboles de $Eval\_Delta(SPR, \Delta_O)$. Donc :

$$\begin{array}{rcl} \Delta^+ & = & \Sigma_R|^{SYs} \cup Eval\_Delta(SPR, \Delta_O)|^{SYs^+} \\ & = & \Delta \cup Eval\_Delta(SPR, \Delta_O)|^{SYs^+} \end{array}$$

- SP est un renommage de la forme SPR **with** SM. Par hypothèse de récurrence, SPR dénote une signature Σ_R^+ de la forme $\Sigma_R \cup Eval\_Delta(SPR, \Delta_O)$. Soit σ^+ le nouveau morphisme induit par SM sur Σ_R^+ tel que nous l'avons défini dans la preuve du théorème précédent. La signature de SP après la propagation est ensuite calculée comme suit :

$$\begin{array}{rcl} \Delta^+ & = & \sigma^+(\Sigma_R \cup Eval\_Delta(SPR, \Delta_O)) \\ & = & \sigma^+(\Sigma_R) \cup \sigma^+(Eval\_Delta(SPR, \Delta_O)) \\ & = & \sigma(\Sigma_R) \cup \sigma^+(Eval\_Delta(SPR, \Delta_O)) \\ & = & \Delta \cup \sigma^+(Eval\_Delta(SPR, \Delta_O)) \end{array}$$

  On utilise la propriété de distributivité de l'application d'un morphisme sur une opération d'union, puis que σ^+ ne diffère pas de σ sur les symboles de Σ_R.
- SP est une réduction de la forme SPR **reveal** SM. Par hypothèse de récurrence, SPR dénote une signature Σ_R^+ de la forme $\Sigma_R \cup Eval\_Delta(SPR, \Delta_O)$. D'après la définition 4.7, la signature de SP est la plus petite sous-signature pleine de Σ_R^+ contenant les symboles du domaine du morphisme induit par SM dans l'environnement local Σ étendu par celle de Σ_R^+. Ce morphisme est construit comme précédemment, et est appliqué sur cette sous-signature. En utilisant le propriétés de distributivité de l'opération de génération sur l'union, on obtient :

$$\begin{array}{rcl} \Delta^+ & = & \sigma^+(\Sigma_R^+|_{SYs^+ \cup |\Sigma|}) \\ & = & \sigma^+((\Sigma_R \cup Eval\_Delta(SPR, \Delta_O))|_{SYs^+ \cup |\Sigma|}) \\ & = & \sigma^+(\Sigma_R|_{SYs^+ \cup |\Sigma|} \cup Eval\_Delta(SPR, \Delta_O)|_{SYs^+ \cup |\Sigma|}) \end{array}$$

Comme dans le cas de la restriction, les nouveaux symboles induits par SM sont des symboles de $Eval\_Delta(SPR, \Delta_O)$, et σ^+ ne diffère pas de σ sur Σ_R, donc :

$$\begin{aligned}\Delta^+ &= \sigma^+(\Sigma_R|_{SYs\cup|\Sigma|} \cup Eval\_Delta(SPR, \Delta_O)|_{SYs^+\cup|\Sigma|}) \\ &= \sigma(\Sigma_R|_{SYs\cup|\Sigma|}) \cup \sigma^+(Eval\_Delta(SPR, \Delta_O)|_{SYs^+\cup|\Sigma|}) \\ &= \Delta \cup \sigma^+(Eval\_Delta(SPR, \Delta_O)|_{SYs^+\cup|\Sigma|})\end{aligned}$$

- SP est une référence simple de la forme SNX. Si $SNX = SN$, alors $Eval\_Delta(SPR, \Delta_O) = \Delta_O$. Sinon, par hypothèse de récurrence, la signature du corps de SNX est $\Sigma_X^+ = \Sigma_X \cup Eval\_Delta(SP_X, \Delta_O)$, qui est égale à la nouvelle signature de SP d'après la règle (5).

□

4.3.4. Cas d'une instanciation

Nous traitons maintenant le cas d'une instanciation. Dans ce cas, le premier lemme n'est pas vérifié. Nous allons montrer en détail les causes de cette limitation.

Nous distinguons deux situations à l'intérieur de ce cas. La première situation concerne le cas où une spécification nommée SN est utilisée en tant que paramètre formel d'une spécification SM et instanciée par une autre spécification SR pour obtenir une dernière spécification SO. La redéfinition de SN rend en général cette dernière spécification mal formée. La seconde situation concerne le cas où une spécification nommée SN est utilisée en tant que paramètre effectif pour instancier une spécification SM dans le corps d'une spécification SR. Cette dernière spécification devient également mal formée après une redéfinition de SN.

Nous exposons en détail ces deux cas ci-dessous.

Paramètre formel

Considérons la bibliothèque de spécification suivante :

spec SN = **sort** s **op** $f : s \to s$
spec SM [SN] = **op** $g : s \times s \to s$
spec SR = **sort** s_1 **op** $h : s_1 \to s_1$
spec SO = SM [SR **fit** $s \mapsto s_1, f \mapsto h$] **then pred** $p : s_1$

L'environnement Γ_s correspondant à l'évaluation de cette bibliothèque est :

$$\begin{aligned}\Gamma_s = \{&SN \to (\emptyset, (), (s, f^t_{ss})), \\ &SM \to (\emptyset, (s, f^t_{ss}), (g^t_{sss})), \\ &SR \to (\emptyset, (), (s_1, h^t_{s_1s_1})), \\ &SO \to (\emptyset, (), (s_1, h^t_{s_1s_1}, g^t_{s_1s_1s_1}, p_{s_1}))\}\end{aligned}$$

Supposons que la déclaration suivante de cette bibliothèque est la redéfinition suivante :

spec SN $\overset{\text{re}}{=}$ SN **then op** $k : s\times \to s$

L'application de la règle de redéfinition par extension (48) à cette dernière déclaration modifie la signature associée à SN dans Γ_s, en y ajoutant un symbole de fonction k^t_{sss}. Pour pouvoir continuer à appliquer cette règle, il faudrait s'assurer que toutes les spécifications faisant référence à SN restent bien formées pendant la propagation. Il n'y a aucun problème pour SM et SR. En revanche , SO n'est plus bien formée. En effet, un des symboles du paramètre formel SN n'est plus pris en compte par le paramètre effectif SR. Lors de la reconstruction d'un morphisme induit par l'application de symboles d'ajustement dans la règle (44), aucun morphisme ne peut être construit puisque le nouveau symbole de SN introduit lors de sa redéfinition n'a pas de contrepartie dans le paramètre effectif SR.

Nous ne pouvons donc pas maintenir la propriété de bonne formation d'une spécification instanciée dans le cas général.

Paramètre effectif

Considérons maintenant la situation suivante :

spec SM [**sorts** s, s'] = **op** $g : s \times s \to s'$
spec SN =
  sorts s_1, s_2;
    $s_1 \leq s_2$
  ops $f : s_1 \to s_1$;
    $f : s_2 \to s_2$
spec SR = SM [SN **fit** $s \mapsto s_1, s' \mapsto s_2$]

L'environnement Γ_s correspondant à l'évaluation de cette bibliothèque est :

$$\begin{array}{rcl}\Gamma_s & = & \{SM \to (\emptyset, (s, s'), (g^t_{sss'})),\\ & & SN \to (\emptyset, (), (s_1, s_2, f^t_{s_1 s_1}, f^t_{s_2 s_2}, s_1 \leq s_2)),\\ & & SR \to (\emptyset, (), (s_1, s_2, f^t_{s_1 s_1}, f^t_{s_2 s_2}, g^t_{s_1 s_1 s_2}, s_1 \leq s_2))\}\end{array}$$

Puis supposons que la déclaration suivante dans cette bibliothèque soit la redéfinition suivante :

spec SN $\overset{\text{re}}{=}$ SN **then op** $g : s_1 \times s_2 \to s_2$

Si nous appliquons la règle de redéfinition par extension (48), alors la signature de SN est mise à jour en y ajoutant un nouveau symbole de fonction $g^t_{s_1 s_2 s_2}$. La propagation de cette nouvelle signature dans la sémantique de SR rend celle-ci mal formée. Un unique morphisme induit par la déclaration de l'application d'ajustement peut être déterminé à partir de celui déterminée au moment de l'évaluation de SR. L'application de symboles calculée par la règle (37) n'est pas modifiée par la redéfinition. Le nouveau morphisme est simplement égal à l'ancien auquel on étend le codomaine à la nouvelle signature de SN. ce morphisme est défini de manière unique.

Il reste ensuite à montrer que l'union de la signature de SN avec celle du corps de SR ajustée est finale. Or la redéfinition de SN a introduit un symbole qui surcharge un symbole du corps de SR et qui n'était pas mentionné dans le paramètre formel. D'après la propriété 4.8, la finalité d'une union revient à regarder si les relations de surcharge ($\sim_F$ et $\sim_P$) induites par le préordre de sous-sortage de la signature de l'union sont égales aux clôtures transitives respectives de l'union des relations de chaque composante. Or il se peut que le fragment de signature apporté par la redéfinition ajoute un élément dans la relation de surcharge du corps de la spécification, brisant ainsi cette propriété. Dans la situation que nous avons prise en exemple, la relation de surcharge de SN avant sa redéfinition est :

$$\sim_F^{SN}= \{(f_{s_1 s_1}, f_{s_2 s_2})\}$$

qui est identique à celle de SR.

Après la redéfinition, la nouvelle relation de surcharge de SR est :

$$\sim_F^{R}= \{(f_{s_1 s_1}, f_{s_2 s_2}), (g_{s_1 s_1 s_2}, g_{s_1 s_2, s_2})\}$$

qui n'est plus égale à la clôture transitive de l'union des relations de surcharge de chacune des composantes. L'union finale ne pouvant être construite, l'instanciation est mal formée.

Dans les deux cas d'un paramètre formel ou d'un paramètre effectif, la bonne formation des spécification génériques ne peut être conservée de manière formelle pendant la propagation d'une redéfinition.

4.3.5. Conclusion

Nous avons proposé dans ce chapitre une optimisation de la propagation pour la sémantique statique dans le cas d'une redéfinition par extension d'une spécification non générique. Cette optimisation a été prouvée correcte. Pour cela, nous avons utilisé deux lemmes. Le premier vérifie que la bonne formation des spécifications déjà évaluées est conservée au cours de la propagation. Le second montre que la signature de chacune de ces spécifications après la propagation est l'union de la signature précédente avec l'extension de signature provenant de la redéfinition, à qui on aura préalablement appliqué des opérations correspondant à la structure de la spécification.

La preuve de ces lemmes nécessite l'introduction de nombreuses notations et concepts techniques. Ces concepts permettent de décrire précisément la sémantique des opérateurs de structuration de Casl. La complexité de ces concepts est la conséquence de la souplesse d'écriture des spécifications. La reconstruction des objets sémantiques, signatures et morphismes, à partir des déclarations est compliquée par la possibilité d'omettre le profil des symboles manipulés. La reconstruction doit donc se faire par étapes successives.

La principale restriction du lemme de conservation de la bonne formation concerne l'occurrence d'instanciations dans la bibliothèque mise à jour. Comme on pouvait s'y attendre, lorsqu'un paramètre formel est étendu, l'argument doit en général être également étendu afin de proposer une instanciation des nouveaux symboles. En revanche, dans le cas où c'est l'argument effectif qui est étendu, nous aurions pu penser que les conditions de bonne formation auraient pu être préservées. Ceci n'est pas le cas à cause de la construction spécifique de l'instanciation dans Casl comme une union finale.

Néanmoins, notre optimisation simplifie une éventuelle implémentation de la redéfinition dans un analyseur syntaxique de spécifications Casl. Pour mettre à jour la sémantique d'une bibliothèque après une redéfinition par extension, il suffit d'évaluer la signature de l'extension, puis d'y appliquer les différentes opérations nécessaires, en réutilisant au maximum les informations disponibles.

5. Spécification de systèmes réactifs

Après avoir motivé (chapitre 2), puis défini (chapitre 3 et 4) la redéfinition de spécifications nommées, nous allons illustrer l'utilisation de cette construction dans le domaine particulier des systèmes réactifs. Un système réactif est un système plongé dans un environnement auquel il réagit en permanence. Un tel système peut prendre des décisions, en proposant des actions correctives sur l'environnement, à l'aide d'actuateurs. Ces actions correctives sont élaborées à partir d'informations provenant de capteurs, et permettent de maintenir des invariants tout au long du fonctionnement du système. Le système peut réagir à deux sortes de stimuli : à des événements que l'on suppose instantanés (comme la réception d'un message) ou bien au passage du temps. Un système réactif combine une partie matérielle et une partie logicielle, pour laquelle la sûreté de fonctionnement est souvent une propriété cruciale.

Par conséquent, ce domaine particulier fait l'objet de nombreuses études pour les chercheurs en techniques formelles. Deux sortes d'études rentrent dans le cadre de cette thèse. La première concerne la recherche de styles architecturaux pour les systèmes réactifs. La donnée de ces styles architecturaux, ainsi que leur formalisation, permettrait d'exprimer et de vérifier des propriétés caractéristiques une fois pour toutes à un haut niveau d'abstraction. Un *Problem Frame* est dédié à la spécification de ces systèmes, qui nous servira de guide lors de la détermination d'une structure préliminaire à l'écriture de la spécification d'un système réactif. La seconde sorte d'études concerne la modélisation de ces systèmes en utilisant des langages de spécification algébrique. A priori, un langage de spécification algébrique ne paraît pas bien adapté pour cet objectif. Néanmoins, un système réactif comprend au moins trois parties qui sont : l'environnement physique, un programme, et un système de communication entre eux. Si l'on ne considère pas les aspects temps-réel du système, alors un langage de spécification algébrique peut être utilisé pour décrire le programme de contrôle. Cette description du programme par une spécification algébrique pourra être complétée par d'autres modèles pour vérifier ces aspects temps-réel.[1]

Deux techniques d'écriture pour spécifier un programme de contrôle se retrouvent dans la littérature :

1. On s'intéresse principalement à l'état global du système. Les interactions avec l'en-

[1] Il existe de nombreuses approches parfaitement adaptées à la spécification de protocoles de communication ou de systèmes temps-réel. Ces approches peuvent également incorporer des éléments algébriques (comme la spécification de types de données abstraits), et sont alors référencées comme des approches algébriques (voir [ABR99] par exemple). Néanmoins, nous nous restreignons ici uniquement aux langages de spécification algébriques comme CASL.

vironnement sont perçues à travers les modifications de cet état global. L'écriture de la spécification consiste alors à construire une description de cet état global par des variables d'état, puis de déterminer les différentes opérations qui modifient la valeur de ces variables d'état en fonction des événements de l'environnement observables par le programme. Cette technique permet de vérifier, par exemple, qu'une certaine variable reste dans un intervalle donné pendant le fonctionnement. Une extension de Casl, Casl-LTL [RAC99], a été proposée pour supporter cette technique.

2. On s'intéresse principalement à l'historique des échanges de messages entre le programme et l'environnement. La valeur de certaines données peut être calculée à partir de cet historique. L'écriture de la spécification consiste à décrire les séquences d'événements avec les réactions appropriés du programme. Cette technique peut être utilisée pour détecter des séquences problématiques dans l'enchainement des événements. Manfred Broy, entre autres, a beaucoup participé au développement de cette technique d'écriture des spécifications, voir [Bro96]. Une étude de cas conséquente, le Steam-Boiler [Abr96b], a été développée par d'autres auteurs en suivant cette technique : voir [BCPR95].

Ces deux techniques ne sont pas exclusives : l'historique peut être vu comme une variable d'état particulière, et la valeur d'un paramètre du système peut être retrouvé à partir de l'historique. Formellement, si l'on considère les modèles mathématiques sous-jacents, ces deux techniques sont expressivement équivalentes. Dans [AR00], les auteurs comparent l'expressivité de ces deux techniques au niveau de la sémantique de schémas de spécification traduisant ces techniques. Selon leurs schémas, ces techniques ne sont pas équivalentes. Nous ne suivrons pas ces mêmes schémas afin de rétablir l'équivalence sous-jacente. Dans ce chapitre, nous ne nous intéresserons pas à des résultats d'expressivité, mais à des critères méthodologiques liés au point de vue d'un utilisateur pour lesquels chacune de ces techniques possède des qualités propres. Les critères qui nous intéressent sont la possibilité de visualiser l'aspect incrémental et itératif de l'écriture dans la spécification, et de pouvoir proposer un découpage architectural correct en minimisant les remaniements de la spécification.

Nous allons montrer dans ce chapitre que ces deux techniques ne possèdent pas un comportement satisfaisant sur ces deux critères simultanément. Après avoir proposé un schéma de spécification pour chacune de ces techniques, nous en montrons les limites sur l'un ou l'autre des critères. Une technique intermédiaire, utilisant la redéfinition de spécifications nommées, combine les avantages de chaque approche sans leurs inconvénients. Nous pouvons ainsi proposer un schéma de spécification souple pour la spécification de systèmes réactifs. Nous illustrerons cette proposition sur une étude de cas : le contrôle d'accès présenté dans [Led96] dont le cahier des charges se trouve dans l'annexe B.

5.1. Spécifier une machine à états

5.1.1. Préliminaires

La spécification d'une machine à états est basée sur une formalisation mathématique de ces machines formulée dans [Mea55], et d'une modélisation proposée dans [vdPHdJ99]. Cette formalisation est constituée de plusieurs éléments. Tout d'abord, d'une description des différents états possibles du système. Un état est caractérisé par un ensemble de valeurs correspondant à des instances des différentes variables d'états. Ces variables dépendent entièrement du problème à traiter. Ce sont des paramètres du système que l'on observe et à partir desquels des inférences sont faites pour prendre des décisions sur le contrôle. Ces variables sont introduites en fonction des actions correctives que le programme peut entreprendre sur le système. Lorsque les valeurs associées à ces variables sont des valeurs réelles, l'ensemble des états possibles est infini.

Les événements auxquels le système réagit sont répertoriés. D'une manière générale, ces événements sont des messages transmis par les capteurs physiques au programme par le biais d'un canal de communication. Nous supposerons ce canal parfait et permettant une transmission instantanée. Les événements produits par le programme (ou les messages) sont différenciés des précédents afin de pouvoir exprimer clairement les réponses du programme en fonction des messages reçus. Tous les événements peuvent être paramétrés par des valeurs, et donc encore une fois être en nombre infini lorsque ces valeurs varient dans un domaine infini.

Les deux fonctions principales du système sont ensuite décrites. La fonction de transition d'un état à un autre est décrite par une fonction prenant en argument l'état courant du système et un événement, et rendant un nouvel état. Cette fonction peut être partielle, car au moment de la phase d'analyse, toutes les transitions ne sont pas forcément explicitées, en particulier en ce qui concerne la gestion des cas d'erreurs. La seconde fonction évalue la réponse du programme à partir de l'état courant et d'un événement. Cette fonction peut aussi être partielle.

Certaines formalisations utilisent des relations à la place des fonctions précédentes. Ceci est justifié lorsque le système à décrire fait intervenir du non-déterminisme. Dans ce cas, plusieurs choix sont alors possibles lors d'une transition ou d'une réponse du programme, ce que la notion de fonction ne peut capturer. Or les systèmes auxquels pourraient être appliqués cette technique sont des systèmes critiques pour lesquels le comportement du programme doit être clairement spécifié. Au moment de la phase d'analyse, où tous les comportements ne sont pas décrits, on préférera s'abstenir plutôt que de surcharger un même comportement pour des stimuli distincts, ce qui introduit du non-déterminisme. Dans [vdPHdJ99], les auteurs font le choix contraire et autorisent la surcharge dans la phase d'analyse, quitte à la dissiper lors de la conception. De même, le choix de déclencher les réponses du système à partir d'un état et d'un événement plutôt que directement à partir des transitions d'états n'a formellement aucune importance, mais en pratique le premier choix est plus naturel.

Un dernier élément est souvent ajouté à cette formalisation. Il s'agit de la caractérisation d'un état initial (ou quelquefois d'un ensemble d'états initiaux).

La définition suivante correspond à cette formalisation mathématique des machines à états :

Définition 5.1 (Machine à états)
Une machine à états est définie par la donnée d'un sextuplet $< Q, I, O, T, R, q_0 >$, où :
- *Q est un ensemble possiblement infini d'états ;*
- *I est un ensemble possiblement infini de messages d'entrée ;*
- *O est un ensemble possiblement infini de messages de sortie ;*
- *$T : Q \times I \rightarrow Q$ est la fonction de transition entre états ;*
- *$R : Q \times I \rightarrow O$ est la fonction de réponse ;*
- *q_0 est un état initial.*

5.1.2. Modélisation

Nous présentons dans cette section un schéma de spécification pour une machine à états. Un schéma de spécification définit un cadre, indiquant les types de données et opérations principaux, à partir duquel une spécification concrète est déduite en instanciant ces éléments en fonction du cas traité. Le schéma de spécification d'une machine à états suit la définition précédente. Il consiste en un type de données pour les états sous la forme d'un enregistrement de variables d'état, c'est à dire un ensemble structuré muni d'opérations pour accéder aux valeurs de chaque champ ; d'un type de données pour les messages d'entrée et d'un autre pour les messages de sortie ; puis d'opérations partielles pour les fonctions de transition et de réponse.

Pour spécifier la fonction de transition, nous introduisons un prédicat représentant les hypothèses du programme sur l'environnement. Ce prédicat teste des préconditions dynamiques avant de permettre une transition. Il prend en argument l'état courant et une entrée. Des exemples de tests classiques consistent à comparer la valeur d'un paramètre effectivement reçue avec une prévision que l'on aura calculée auparavant, ou encore vérifier l'occurrence de certains messages attendus (comme des accusés de réception par exemple). La fonction de transition est alors spécifiée selon la validité de ce prédicat. De plus, cette fonction peut être spécifiée progressivement. Pour chacune des variables d'états introduites dans la déclaration du type des états, une fonction auxiliaire évalue la transition de cette variable. Le profil de ces opérations intermédiaires fait intervenir le domaine de valeurs de chaque variable, que l'on note $V(i)$ pour chaque variable d'état v_i. La fonction de transition globale appelle ces opérations intermédiaires.

La spécification CASL suivante décrit un schéma de spécification pour une machine à états. Une spécification concrète se déduit de ce schéma en instanciant les types de données des états et des entrées/sorties, puis les fonctions T et R.

spec SYSTEME = ... **then** %% Spécifications à importer
  generated type *States* ::=< ... >%% Variables d'état
  generated type *Inputs* ::= ... %% Messages d'entrée

generated type *Outputs* ::= ... %% Messages de sortie
ops q_0 : *States*;
T : *States* $\times$ *Inputs* $\rightarrow?$ *States*;
R : *States* $\times$ *Inputs* $\rightarrow?$ *Outputs*;
$T_1, \ldots, T_n$: *States* $\times$ *Inputs* $\rightarrow?$ $V(i)$
pred *Pre* : *States* $\times$ *Inputs*
$\forall q, q'$: *States*, i : *Inputs*;
%% Définition des préconditions :
• $Pre(q, i) \Leftrightarrow \ldots$
%% Définition des fonctions de transition auxiliaires :
• $Pre(q, i) \Rightarrow T_1(q, i) = \ldots;$
• $\neg Pre(q, i) \Rightarrow T_1(q, i) = \ldots;$
• ...
%% Définition de la fonction de transition :
• $T(q, i) =< T_1(q, i), \ldots, T_n(q, i) >$
%% Définition de la fonction de réponse :
• ...
end

Pour valider une spécification instanciée de ce schéma de spécification, deux sortes de propriétés peuvent être prouvées :

– On peut vérifier que des propriétés conséquences de la spécification sont vraies. Il s'agit d'une validation interne de la spécification. En principe, la cohérence de la spécification (l'existence d'au moins un modèle) devrait être vérifiée, mais cette propriété est indécidable en général.
– Des éléments du cahier des charges n'ayant pas été pris en compte peuvent constituer des propriétés intéressantes à vérifier. Il s'agit alors d'une validation externe. Pour cela, nous devons reprendre le cahier des charges, identifier ces éléments, puis les formaliser et prouver qu'ils sont bien une conséquence logique de la spécification.

Nous avons insisté dans la présentation des systèmes réactifs sur le fait qu'un tel système doit maintenir tout au long de son fonctionnement un certain nombre d'invariants. Ces invariants ne sont pas encore intervenus dans la description du schéma de spécification, il est intéressant de montrer que ces invariants sont bien des conséquences de la spécification. Un prédicat *Inv* : *States* est défini, formalisant les différents invariants. Mais, en général, l'invariant n'est pas vrai pour tous les états décrits par le type de données *States*. Le domaine défini par le type des états est souvent plus vaste que celui du système que l'on décrit. En effet, certaines combinaisons de valeurs des variables d'états ne correspondent peut-être à aucun état réellement réalisable par le système. La conservation de l'invariant doit être vérifiée uniquement sur des états accessibles du système. Ces états peuvent être caractérisés par un prédicat d'accessibilité identifiant tous les états que l'on pourra atteindre à partir de l'état initial et de la fonction de transition. Ce prédicat correspond à la plus petite relation contenant la clôture transitive du prédicat de transition, à partir d'un état initial $q0$ spécifié préalablement. Dans CASL, cette plus petite relation se traduit par une construction libre, comme suit :

$$\begin{array}{ll}\textbf{free} & \{\ \textbf{pred}\ \mathit{Reach} : \mathit{States} \\ & \forall\, q, q', q0 : \mathit{States};\, i : \mathit{Inputs} \\ & \bullet\ \mathit{Reach}(q0) \\ & \bullet\ (\mathit{Reach}(q) \wedge T(q,i) = q') \ \Rightarrow\ \mathit{Reach}(q')\ \}\end{array}$$

Ce prédicat d'accessibilité définit un schéma de récurrence sur les états, ce qui permet de prouver les propriétés sur les états par une induction structurelle. La conservation de l'invariant s'exprime alors par :

$$\forall q : \mathit{States} \bullet \mathit{Reach}(q) \Rightarrow \mathit{Inv}(q)$$

Ce schéma de spécification est semblable à celui proposé dans une autre extension de Casl, Casl-LTL [RAC99]. Dans cette extension, les types des entrées et des sorties ne sont pas distingués. Une construction syntaxique est définie afin de déclarer en une seule fois le type des états, des étiquettes (les entrées et les sorties), et d'un prédicat de transition. Les variables d'états ne sont pas identifiées dans cette construction, car l'accent est mis sur les transitions plus que sur les états. La sémantique d'une telle déclaration étend la sémantique usuelle de Casl par un ensemble de chemins par lequel le système peut passer. Nous détaillerons cette notion de chemin dans la section suivante. Notre approche, pour cette technique, consiste plutôt à détailler la structure des états et à décrire la fonction de transition selon les transformations effectuées sur les variables d'états.

Cette approche est fortement inspirée de celle proposée dans [vdPHdJ99]. Le modèle mathématique de départ est proche du notre. Les points de différence concernent le mode de déclenchement des réponses et la modélisation des transitions et des réponses. Dans leur modélisation, les réponses sont déclenchées par des transitions d'états plutôt que par l'occurrence d'un événement dans un état donné. Ce choix est justifié si plusieurs sortes d'événements sont possibles. Dans ce cas, la sémantique utilisateur d'une réponse n'est pas évidemment reliée à un événement. Dans notre cas, nous ne considérons qu'un seul type d'événement, la réception de messages provenant des capteurs. La liaison entre une réponse et un message est alors claire. De plus, les auteurs choisissent de décrire les transitions et les réponses par des relations plutôt que des fonctions. Nous pensons que ce choix n'est pas judicieux, car, lors de la phase de conception, la distinction entre deux comportements possibles peut s'avérer difficile. Il nous parait préférable de ne pas spécifier le comportement du programme pour un état et une entrée donnés, plutôt que de le spécifier avec un comportement similaire à une autre situation. Certaines preuves de validation pourraient n'être pas vérifiées dans ce cas. Leur choix est justifié par le fait que dans leur cas, l'une de ces propriétés de validation est l'absence de blocage nécessitant pour cela d'avoir un comportement pour chaque cas, quitte à ce que ce comportement soit modifié pendant la phase de conception. Nous nous restreignons ici à des propriétés de sûreté.

5.1.3. Comportement méthodologique

Nous allons évaluer cette modélisation selon les critères méthodologiques qui nous intéressent : visibilité du processus de développement, évolution vers une structure modulaire.

Modularité

Ce schéma de spécification peut se découper naturellement en blocs, chaque bloc pouvant contenir une ou plusieurs spécifications nommées, pouvant être associé à des unités architecturales et être ensuite développées indépendamment. Il s'agit donc d'une structure préliminaire du programme, telle que nous l'avons décrite dans le chapitre 3.

- Un premier bloc de structure concerne l'interface entre le système et l'environnement, c'est à dire les événements auxquels le système réagit (avec possiblement une séparation entre entrées et sorties). Ces événements peuvent manipuler des données, dont il faudra déclarer le type auparavant.
- Un deuxième bloc concerne des aspects internes du système, en particulier la déclaration du type des états reflétant la connaissance du système sur son environnement. La contrainte d'accessibilité est intégrée dans ce bloc, ainsi que la définition d'états particuliers du système comme l'état initial, par exemple.
- Un troisième bloc contient les préconditions dynamiques de la fonction de transition.
- La fonction de réponse peut se trouver dans un bloc distinct, car elle ne fait appel qu'à la structure des états et à des raisonnements par cas.
- La dernière partie contient principalement la fonction de transition avec les différents prédicats ou opérations intermédiaires nécessaires à son expression simplifiée.

Des spécifications intermédiaires peuvent évidemment être ajoutées pour améliorer la lisibilité de la spécification. Ce découpage modulaire est relativement standard. Il se retrouve en particulier dans le découpage proposé dans [vdPHdJ99], mais aussi dans de nombreux articles sur le découpage architectural des systèmes réactifs. Il correspond à un style architectural centré sur les données, particulièrement intéressant lorsque diverses fonctionnalités du système peuvent fonctionner en parallèle en utilisant la même structure des états, voir par exemple dans [SvK97, ST99, BKBdJ98]. Dans notre cas, cette structure n'est utilisée que pour une seule tâche (répondre aux stimuli afin de maintenir un invariant), mais on pourrait imaginer une extension concurrente de ce modèle. La propriété principale de ce style est que chacun de ces blocs principaux peut être ensuite développé indépendamment.

Visibilité du processus

La donnée de cette structure préliminaire montre que le processus de spécification peut se dérouler de manière incrémentale, en s'occupant d'un bloc à la fois. Le cahier des charges peut donc être traduit progressivement. Néanmoins, il parait probable que certains aspects de la spécification ne peuvent pas être traduits en une seule fois, mais demandent plusieurs itérations selon la complexité du cahier des charges. En particulier, la connaissance de toutes les variables d'états nécessaires à la prise de décision au début du processus de développement parait illusoire. En effet, dans la structure préliminaire, le bloc de spécifications contenant la déclaration du type intervient tôt car la structure des états est utilisé pour le calcul des préconditions, et des fonctions de réponse et de transition.

Certaines variables d'états peuvent être déterminées par avance, que ce soit par expérience ou par une simple analyse du cahier des charges. Mais dans des cas complexes, d'autres variables n'apparaitront qu'au fur et à mesure du processus de développement. Cette situation sera bien illustrée par notre traitement de l'étude cas du Contrôle d'Accès, dans la section 5.4. L'apparition de ces variables implique de modifier le type de données des états en y a ajoutant ces nouvelles variables. Cette modification ne peut pas être qu'une extension du type de données précédent : si à un certain moment du développement un état est constitué de deux variables $< v_1 : V(1), v_2 : V(2) >$ et qu'une nouvelle variable apparaît, le type des états peut être modifié en un nouveau type $States_2$ construit par $< e_1 : States, v_3 : V(3) >$. Néanmoins, il sera nécessaire de modifier toutes les occurrences du type $States$ dans le profil des opérations et prédicats par le nouveau type, ce qui ne peut être traduit formellement.

Une autre manière de procéder consiste à développer indépendamment les spécifications associées à une nouvelle variable d'état, puis à effectuer un produit d'automates sur les spécifications finales. Dans le cas précédent, plutôt que de modifier le type des états, un autre type $States_2$ est défini par $< v_3 : V(3) >$ avec une fonction de transition particulière ainsi qu'une fonction de réponse. Une spécification générique codant le produit synchronisé d'automates est instanciée par les deux spécifications obtenues pour reconstruire la spécification globale. Cependant, cette description générique du produit d'automates est donnée pour une valeur particulière du nombre de paramètres à synchroniser (le nombre de fonctions, de prédicats intermédiaires, etc.). Si de nouvelles fonctions apparaissent au cours du processus, cette spécification générique du produit doit être modifiée, perdant encore une fois les versions intermédiaires.

La constitution progressive du type de données des états est fortement itérative dans un système complexe. Les variables d'états sont identifiées au fur et à mesure des besoins et ne peuvent, en général, être toutes déterminées en une seule fois. La mise à jour du type des états ne peut pas se faire de façon à conserver les versions intermédiaires et visualiser l'évolution du processus dans la spécification. Par conséquent, cette technique de spécification n'est pas bien adaptée à une approche incrémentale et itérative du développement.

5.2. Spécifier l'historique d'un système

5.2.1. Préliminaires

Il y a une équivalence bien connue (théorème de Kleene) entre certains systèmes de transitions et des classes de langages définis par des grammaires formelles. Dans le cas des systèmes réactifs on ne peut pas parler de langage reconnu par un système de transition puisqu'il n'y a pas de notion d'état acceptant. On peut cependant étudier l'historique des événements auxquels le système a réagi. Cet historique définit un mot sur l'alphabet des événements et est appelé trace du système.

Étant donné une machine à états telle que décrite précédemment, la définition formelle de son comportement est donnée par l'arbre des transitions partant de l'état initial. Un noeud de cet arbre représente un état du système de transition et les branches de l'arbre sont étiquetées comme les transitions, par des messages d'entrée. On peut choisir de conserver l'historique des messages d'entrée, de sortie, ou les deux à la fois. Dans la suite nous ne regarderons que l'historique des messages d'entrée. Soit I^* l'ensemble des mots finis sur I, l'ensemble des messages d'entrée. Une trace $T = i_1 i_2 \ldots i_n$ à partir d'un état quelconque $q \in Q$ d'un système de transition est un mot sur I^* tel que pour tout $j \in [1..n]$ il existe une transition $T(q_j, i_j) = q'_{j+1}$, avec $q_1 = q$.

5.2.2. Modélisation

La définition précédente des traces d'un système utilise explicitement la relation de transition utilisée pour décrire la machine à états correspondante. Cela signifie qu'une modélisation possible de cette définition prolongerait simplement le schéma de spécification précédent. Nous proposons une modélisation indépendante de la première, centrée sur les interactions entre le programme et l'environnement. L'historique du système est le concept de base à partir duquel est construite toute la spécification. Ce modèle suit une approche centrée sur les processus, plutôt que sur les données dans le premier cas.

La spécification Casl de l'historique d'un système se compose d'un type de données pour les entrées, d'un type pour les sorties, et d'un type pour l'historique du système. L'historique du système est défini récursivement, à partir d'une situation initiale et d'un constructeur modélisant la reconnaissance d'une entrée pour le programme. Lorsque le système réagit également au passage du temps, cette reconnaissance peut être assimilée à la synchronisation du programme avec une horloge externe. Chaque application du constructeur signifie que l'horloge a parcouru une certaine unité de temps. Il est intéressant de noter que cette modélisation ne décrit pas un déroulement particulier, mais tous les déroulements possibles à partir d'une situation initiale donnée. Cette structure permet ensuite de décrire le système de manière observationnelle. Le fonctionnement du programme est modélisé par des opérations ou des prédicats définis inductivement sur l'historique du système. Ces opérations sont appelées observateurs car elles prennent en argument un historique et peut-être d'autres paramètres, et rendent une donnée d'un type quelconque distinct d'un historique. Comme l'évolution d'un historique suit la réception de messages, une opération prenant en argument des messages d'entrée ne peut pas être vue comme un historique. C'est une restriction cohérente sur le profil des observateurs.

Ces observateurs modélisent les paramètres d'intérêt du système et sont en général mutuellement récursifs. Ils jouent un rôle similaire aux variables d'état dans la modélisation par une machine à états, en permettant une analyse dynamique de la succession des événements. Par exemple, un observateur O_1 peut recenser les divers cas d'erreurs dans la succession des événements en fonction de la valeur d'autres observateurs (si O_j vaut tant, alors tel événement ne doit pas se produire, etc.). Il correspond au prédicat de préconditions dynamiques Pre de la première approche. Les autres observateurs O_i sont définis ensuite par induction sur l'historique, puis par cas selon la valeur de O_1 (ou d'une condition plus complexe contenant au moins une clause sur O_1). Comme dans la section précédente, la gestion des cas d'erreurs peut être reportée au moment de la phase de conception. La fonction de réponse conserve la même modélisation que dans le premier schéma, prenant en argument un historique et des messages d'entrée, et évalue des messages de sortie. Cette fonction du programme ne peut être vue comme un observateur.

On obtient le schéma de spécification suivant :

spec SYSTEM = %% Imports **then**
  generated type *Inputs* ::= ... %% Entrées
  generated type *Outputs* ::= ... %% Sorties
  generated type *History* ::= *Init* | __.__(*History*; *Inputs*)
  ops $O_1, \ldots O_n$: *History* → ... %% Les observateurs
    R : *History* × *Inputs* → *Outputs* %% La fonction de réponse
  $\forall h : History; i : Inputs; o : Outputs$
  %% Définition de l'analyse dynamique
  • $O_1(Init) = \ldots$
  • $Cond(O_i(h), i, o) \Rightarrow O_1(h.i) = \ldots$
  %% Définition des autres observateurs
  • $O_i(Init) = \ldots$
  • $O_1(h) = \ldots \Rightarrow O_i(h.i) = \ldots$
  • $O_1(h) = \ldots \Rightarrow O_i(h.i) = \ldots$
  %% Définition de la fonction de réponse
  ...
end

La spécification du système par des observateurs permet d'exprimer un grand nombre de propriétés permettant de valider la spécification obtenue en utilisant le fait que les observateurs sont mutuellements définis. Supposons qu'un observateur O_i est défini à partir d'un observateur O_j, lui même défini à partir de O_k. Une propriété de cohérence peut être exprimée avec une propriété entre O_i et O_k. Il est possible en général d'exprimer par cette méthode que les invariants du système sont conservés.

La spécification de systèmes réactifs avec des spécifications algébriques en suivant le modèle des traces a été particulièrement développée par Manfred Broy, voir par exemple [Bro96]. Dans ce modèle, la sémantique d'un système déterministe est donnée par une fonction d'entrée-sortie $f : M^* \to M^*$ appelée *stream processing function*, où M est l'ensemble indifférencié des messages échangeables entre le programme et les unités physiques. La spécification d'un système est alors donnée par un prédicat sur la fonction de comportement, c'est à dire un observateur. Cette modélisation est donc proche du schéma, mis à part le fait que les messages d'entrée sont explicitement différenciés des messages de sortie dans notre schéma.

Mise en oeuvre

Une étude de cas a été menée dans [BCPR95] en suivant ce schéma de spécification : le Steam-Boiler [Abr96b]. Le système étudié est une chaudière alimentée par quatre pompes déversant de l'eau dans une cuve chauffée afin de produire de la vapeur. Le système physique est commandé par des contrôleurs de pompes permettant de fermer ou ouvrir certaines pompes afin de régler le niveau d'eau dans la cuve, et des capteurs sont disposés afin de construire une image abstraite du système (niveau d'eau, quantité de vapeur, statut des pompes, etc.). Le programme gérant le système reçoit régulièrement des messages provenant des capteurs qu'il doit traiter afin de maintenir le niveau d'eau dans un intervalle précis. Dans le cas d'une détection d'éventuelles pannes, le programme passe par une succession de modes de fonctionnements dégradés avant de réclamer l'arrêt du système.

Nous reproduisons ici les aspects essentiels de la spécification illustrant le schéma de spécification de l'historique d'un système. La spécification nommée Basic contient la déclaration de types de données pour les messages entrant et sortant. Ces messages sont paramétrés par les unités physiques émettrices, qui sont également déclarées comme des types de données préliminaires.

spec Basic =
%% Les types des entrées/sorties

La spécification SBCS contient la définition de l'historique des messages d'entrée par un constructeur représentant une synchronisation avec une horloge externe envoyant un signal toutes les cinq secondes (la durée d'un cycle). Pendant ce cycle, le programme récupère les messages arrivés, effectue des traitements, puis envoie des messages à certaines unités physiques. Un observateur sur le mode courant du système est déclaré en même temps. Selon la validité présumée des unités physiques, le système sera dans un mode plus ou moins dégradé, allant du mode normal à l'arrêt complet. L'action principale du contrôleur, qui est l'envoi de messages aux actuateurs, est également déclarée dans cette spécification.

spec SBCS = Basic **then**
  generated type $History ::= Init \mid \_\_.Tick(\_\_)(History, Inputs)$
  ops $\_\_.Mode : History \to Modes;$
    $MessagesSent : History \times Inputs \to Outputs$

À chaque cycle, le contrôleur envoie plusieurs sortes de messages : le mode courant, les ouvertures/fermetures de pompes nécessaires au maintient de l'invariant, des messages de détection de défaillance aux unités concernés, des messages de remise en route après la réception de messages correspondant à la réparation de ces unités défaillantes. Une analyse dynamique détaillée doit donc être effectuée afin de déduire la validité des unités physiques pour déterminer l'ensemble des messages envoyés à chaque cycle. Dans la spécification suivante, un observateur $FailureStatus$ défini inductivement sur l'historique évalue le statut des différentes unités, qui peut être soit normal, soit en panne non déclarée, soit en panne déclarée :

spec Failures = SBCS **then**
  free type $Status ::= Ok \mid FailureWithAck \mid FailureWithoutAck$
  op $FailureStatus : History \times PhysicalUnit \to Status$

pred $FailureDetected : History \times Inputs \times PhysicalUnit$
%% Axiomatisation de $FailureStatus(h.Tick(msgs), PU)$ en fonction de
%% $FailureStatus(h, PU)$ et $FailureDetected(h, msgs, PU)$

La détection de défaillances (prédicat $FailureDetected$) se base, selon le type d'unité physique traité, sur des comparaisons entre les messages reçus et les messages envoyés au cycle précédent ou même deux cycles avant, ou avec les messages reçus au cycle précédent, ou encore avec des prévisions de valeur estimées à l'aide d'un modèle physique du fonctionnement du système. Dans notre schéma, ces informations se traduiraient par des observateurs sur l'historique. Dans leur présentation, les auteurs préfèrent présenter les axiomatisations pour le cycle $n + i$, où n est le cycle courant et i est le nombre de cycles pendant lesquels des informations doivent être collectées, plutôt que d'introduire explicitement ces observateurs. La spécification suivante montre l'exemple des contrôleurs de pompes, pour lequel $i = 2$:

spec PUMPS_CONTROLLER_FAILURES = FAILURES **then**
%% Axiomatisation de $FailureDetected(h.Tick(msgs).Tick(msgs1), msgs2, PControl(Pn))$
%% en fonction de $MessagesSent(h, msgs)$, $MessagesSent(h.Tick(msgs), msgs1)$,
%% $FailureStatus(h.Tick(msgs), Pump(Pn))$, et
%% $FailureStatus(h.Tick(msgs).Tick(msgs1), Pump(Pn))$

Dans cet exemple, le statut des pompes deux cycles avant le cycle courant est nécessaire, ainsi que les deux derniers ensembles de messages, pour déterminer le statut des contrôleurs de pompes.

Le mode courant du contrôleur est ensuite déterminé à partir de l'analyse dynamique :

spec MODE =
%% Imports : Toutes les spécifications de détection de défaillances
then
%% Axiomatisation de $h.Tick(msgs).Mode$ en fonction de $h.Mode$
%% et de $FailureStatus(h.Tick(msgs), PU)$

L'analyse dynamique et le mode sont utilisés pour déterminer les messages à envoyer :

spec MESSAGESSENT = MODE **then**
%% Axiomatisation de $MessagesSent(s, msgs)$ en fonction de $FailureStatus(s.Tick(msgs), PU)$,
%% de $s.Tick(msgs).Mode$, et du contenu de $msgs$

Cette mise en oeuvre du schéma de spécification illustre bien les circularités dans la définition des observateurs. Par exemple, les messages à envoyer sont déterminés en fonction de l'analyse dynamique, elle-même effectuée en fonction des messages envoyés au cycle précédent. De même, les estimations du niveau d'eau ou de vapeur, qui sont utilisées pour la détection de défaillances des capteurs correspondant, sont calculées en fonction de la validité de ces unités physiques, elle-même évaluée en fonction de la détection de défaillances à partir des estimations du cycle précédent.

5.2.3. Comportement méthodologique

Visibilité du processus

Dans ce schéma de spécification, la complexité du problème se traduira au niveau de la spécification de l'analyse dynamique, et du choix des observateurs. Comme l'ont montré les auteurs de [BCPR95], ce schéma se prête bien à une approche incrémentale et itérative du processus de développement. Le traitement des différents cas de l'analyse dynamique, ou bien l'introduction progressive des opérations et prédicats est particulièrement bien illustrée dans cet exemple. La déclaration et la définition d'une opération ou d'un prédicat peuvent également être localisées dans des spécifications nommées distinctes (par exemple pour $MessagesSent$, et pour beaucoup d'autres que nous n'avons pas mentionné), permettant ainsi de décrire la justification de l'introduction des opérations et prédicats nécessaires avant de procéder progressivement à leur définition. Le lecteur de la spécification peut ainsi suivre l'évolution du processus, d'abord d'un point de vue global, puis en rentrant petit à petit dans les détails.

Les observateurs nécessaires à l'analyse dynamique ou à la prise en compte de nouveaux aspects du système peuvent être introduits tout au long de la phase d'analyse, au fur et à mesure des besoins. Le seul lien entre ces observateurs est la structure très simple de l'historique définie une fois pour toutes au départ du processus de spécification. Dans l'exemple précédent, les auteurs n'ont pas mis l'accent sur l'identification d'observateurs. Comme nous l'avons remarqué, ils préfèrent présenter les axiomatisations de manière à pouvoir se passer de la plupart des observateurs. Nous pensons que ce choix n'est pas judicieux pour la suite du développement. Nous reviendrons sur ce point dans la section suivante.

Modularité

Une conséquence gênante de cette souplesse de spécification est qu'il est difficile de proposer un découpage naturel de ce schéma de spécification. Un bloc de spécification peut contenir la définition des types de données essentiels ainsi que les informations intermédiaires nécessaires à leur spécification. Mais ensuite, à cause de la définition mutuelle des différents observateurs, il sera impossible de proposer un découpage de la spécification en unités indépendantes. Par exemple, si on voulait suivre un style de découpage dans lequel une unité contient les informations pour chaque observateur, la définition mutuelle implique une boucle dans la dépendance des unités ce qui rend la spécification architecturale correspondante mal formée.

Dans le traitement de l'étude de cas du Steam-Boiler, la répartition des différentes fonctionnalités du programme (statut des unités physiques, détection des défaillances, évaluation du mode ou des messages à envoyer) ne peuvent pas être disposées dans des unités distinctes à cause de leur définition mutuellement récursives. Au délà de la séparation entre déclarations et définitions dans le texte de la spécification, c'est le schéma de modélisation en lui-même qui entraîne ces circularités.

Ce schéma de spécification est donc bien adapté à la phase d'analyse, mais pas à la phase de découpage architectural.

5.3. Une approche synthétique

Chacun des schéma de spécification proposés possède ses qualités propres vis à vis des critères méthodologiques qui nous intéressent : pouvoir suivre un processus de développement incrémental et itératif, puis pouvoir proposer une découpage architectural sans perdre les avantages de cette approche incrémentale.

Le schéma de spécification par une machine à états est bien structuré : on peut proposer une structuration préliminaire de la spécification permettant de diriger le développement et aboutissant à une spécification facilement découpable. Mais le processus de développement ne peut pas être bien intégré dans ce schéma : pour fournir une spécification correcte, certaines évolutions doivent écraser les informations précédentes alors qu'on voudrait simplement les ajouter (redéfinition du type des états lors de l'introduction d'une nouvelle variable d'état).

Le schéma de spécification par l'historique du système permet lui d'incrémenter la spécification sans écraser les versions antérieures. Le processus de développement est donc visible dans la spécification. Mais cette souplesse empêche de définir une structure préliminaire bien adaptée à un découpage architectural, en particulier à cause de la définition mutuelle des observateurs.

La redéfinition des noms de spécification et la propagation de leur sémantique permet de définir un troisième schéma de spécification pour les systèmes réactifs regroupant les avantages des deux précédents sans leurs inconvénients, que nous proposons maintenant.

5.3.1. Modélisation

Pour combiner de manière positive les deux schémas de spécification précédents, nous associons un schéma à chaque phase de la spécification. Pendant la phase d'analyse, le système est décrit à partir d'observateurs explicites sur l'historique des messages d'entrée. Ainsi, le développement de la spécification des besoins peut suivre un processus incrémental et itératif. Les paramètres importants du système sont identifiés par ces observateurs et incorporés au fur et à mesure du développement. Pendant la phase de conception, cette structure est traduite en une machine à états permettant de découper la spécification en unités indépendantes.

La traduction d'un schéma à un autre peut se voir comme l'application d'un opérateur transformationnel (voir les méthodes décrites en 1.3.2). Pour définir cette traduction, nous devons expliciter comment construire la structure des états à partir du type des historiques et des observateurs, ainsi que les fonctions de transition et de réponse. Les états du système sont constitués de variables issues des observateurs. Soit $O_i : History \ldots \rightarrow V(i)$ un observateur identifié au cours du processus. Il lui est associée une variable $< \ldots v_i : V(i) \ldots >$ dans la structure des états et une fonction de transition partielle. Tous les observateurs ne sont pas systématiquement associés à des variables d'états. La fonction de transition est spécifiée en utilisant des fonctions de transition partielles définies à partir des axiomes associés aux observateurs. Supposons que l'observateur O_j soit défini par les axiomes :

$$\begin{array}{l} \forall h : History, i : Inputs \\ \bullet\ O_j(Init) = a; \\ \bullet\ Cond_1 \Rightarrow O_j(h.i) = F(O_j(h)); \\ \bullet\ Cond_2 \Rightarrow O_j(h.i) = G(O_j(h)) \end{array}$$

Les conditions $Cond_1$ et $Cond_2$ correspondent à la valeur du prédicat Pre sur les entrées. Alors la fonction de transition partielle T_j correspondante est définie par :

$$\begin{array}{l} \forall q : States; i : Inputs \\ \bullet\ Pre(q, i) \Rightarrow T_j(q, i) = F(q); \\ \bullet\ \neg Pre(q, i) \Rightarrow T_j(q, i) = G(q); \end{array}$$

La fonction de transition globale est ensuite reconstruite comme dans le premier schéma.

La description de ce schéma comporte une description de sa structuration ainsi que de sa progression au cours du processus de développement.

– Comme pour les schémas précédents, le processus démarre avec la spécification des événements du système observables par le programme, c'est à dire de l'interface statique du programme par des messages d'entrée et de sortie.
  spec STATIC_INTERFACE
  sort *Inputs* : . . . ;
  Outputs : . . .
– Une ébauche rapide du programme est ensuite effectuée. Une spécification nommée contient le type des états. À ce niveau d'abstraction, un état est simplement décrit comme un objet d'une sorte *Etats*. La structure des états est laissée entièrement libre. Les variables d'états sont ajoutées au fur et à mesure, dans le même module de spécification grâce au mécanisme de redéfinition, sous forme d'observateurs sur cette sorte. Un observateur est ajouté dès que l'on a besoin de connaître la valeur d'une certaine quantité avant l'arrivée du dernier événement (ou le cycle précédent) pour évaluer la réponse présente du programme. Ces observateurs permettent de déplier les récursions mutuelles entre les observateurs qui posaient problème dans le second schéma.
  spec INFORMATION_MODEL = STATIC_INTERFACE **then**
  sort *Etats*
  ops *Obs_1* : *Etats* → *s*; *Obs_2* : *Etats* → *s′*
– La deuxième passe consiste à donner une définition partielle de la fonction de transition. Pour chaque observateur, une fonction intermédiaire est définie indiquant comment sera modifié l'observateur au tour suivant en fonction de l'état courant et des messages reçus. Chaque fonction pourra être déclarée et définie dans des spécifications nommées distinctes, étendant la spécification nommée précédente. Ces fonctions

de transition partielles pourront être totales ou partielles selon la complexité du problème. Ces spécifications nommées pourront de plus être développés indépendamment les unes de autres.

spec A = INFORMATION_MODEL **then**
pred *New_1* : *Etats* × *Messages* → *s*

spec B = INFORMATION_MODEL **then**
pred *New_2* : *Etats* × *Messages* → *s'*

- L'opération d'envoi des messages aux unités physiques se fera dans une spécification séparée. Cette opération prend en argument l'état courant et le dernier ensemble de messages reçus. Lorsqu'une information concernant une variable est nécessaire pour calculer l'ensemble des messages à envoyer, on fera appel aux observateurs.

  spec ENVOI = INFORMATION_MODEL **then**
  op *Envoi* : *Etats* × *Inputs* → *Outputs*

- Lorsqu'un nouvel observateur est identifié, une redéfinition par extension de la spécification des états est donnée, dans laquelle apparait ce nouvel observateur. Il faudra également définir la fonction de mise à jour correspondante.
- Enfin, pendant la phase de conception, un raffinement de cette structure de données vers un tuple de variables d'état peut être effectué. Ce raffinement prend son sens lorsque toutes les variables d'états sont connues. La fonction de transition est définie à partir de ses définitions partielles. Le prédicat d'accessibilité, nécessaire pour les preuves de validation, est défini ensuite :

  spec ETATS $\stackrel{re}{=}$ INFORMATION_MODEL **then**
  free **type** *Etats* $::=< Obs_1 : s; Obs_2 : s'; \ldots >$

  spec SYSTEM = A **and** B **then**
  op $T : Etats \times Inputs \to Etats$
  $\forall q : Etats; i : Inputs;$
  $\bullet\ T(q, i) =< New\_1(q, i), New\_2(q, i), \ldots >$
  then **free** **{**
  pred *Reach* : *States*
  $\forall\ q, q' : State; m : Messages$
  $\bullet\ Reach(Init)$
  $\bullet\ (Reach(q) \land q \xrightarrow{m} q') \Rightarrow Reach(q')$ **}**

Chacun des modules de spécification pourra ensuite être associé à une unité architecturale lors de la phase de découpage. La contrainte de complétude de la spécification des unités est vérifiée. De plus, on peut facilement prendre en compte le partage (de ETATS par A et B).

La principale différence entre ce schéma de spécification et celui suivi par les auteurs de [BCPR95] est dans l'explicitation systématique des observateurs ainsi que de la fonction de transition partielle associée. Cette explicitation permet de distinguer l'observation d'une valeur sur un état de la modification de cette valeur en fonction de l'évolution du système. Cette distinction n'est pas faite dans leur formalisation, ce qui entraîne les récursions mutuelles. Le mécanisme de redéfinition permet en outre de regrouper tous les observateurs dans la spécification des états pour simplifier la lecture de la spécification.

Par exemple, le statut des unités physiques, qui est un observateur explicite dans leur traitement, aurait été déclaré dans une redéfinition de SBCS, sous le nom $Obs\_FailureStatus$. Une fonction de transition partielle $New\_FailureStatus$ serait définie dans une spécification nommée distincte. Cette opération prend un historique et des messages d'entrée en argument et retourne le statut des différentes unités pour le cycle suivant. L'axiomatisation de cette opération reprend celle donnée pour $FailureStatus$ dans la spécification originale en remplaçant les références récursives par un appel à $Obs\_FailureStatus$. Ainsi les récursions mutuelles sont dépliées une par une. L'axiomatisation de la détection de défaillances pour les contrôleurs de pompes aurait fait intervenir de nouveaux observateurs : les messages envoyés au cycle précédent, ceux envoyés deux cycles auparavant, le statut des unités deux cycles auparavant. Les estimations apparaitraient également comme des observateurs explicites. L'axiomatisation concerne alors les fonctions de transition partielles, qui prend en argument un état et des messages d'entrée, et fait appel à ces observateurs.

Travaux afférents

Passer d'une description observationnelle à une description constructive est une manière classique de raffiner une spécification. On la retrouve par exemple dans la spécification en CASL de la gestion d'un dépôt, par D.Bert et H. Baumeister [BB00]. L'intérêt de notre approche apparaît clairement lorsqu'on regarde en détail cette spécification. Après une phase d'analyse et une phase de conception, les auteurs proposent également un découpage modulaire de leur spécification sous la forme d'une spécification architecturale. Ce découpage intervient naturellement dans le processus de spécification, et il est intéressant de noter que la spécification architecturale proposée est incorrecte, à cause de l'interaction entre le processus incrémental et l'identification de traits caractéristiques. Le problème peut être résumé comme suit.

Pendant la phase d'analyse, les auteurs proposent une description observationnelle d'une sorte *Order*. Deux observateurs sont définis : *Ref* identifiant la référence de l'objet de la commande, et *Qty* sa quantité. Comme le processus de spécification n'en est qu'à ses débuts au moment de cette déclaration, d'autres observateurs peuvent apparaître, motivant ainsi cette description observationnelle.

spec ORDER =
    sorts $Order, Product, Pos$
    ops $Ref : Order \to Product;$
        $Qty : Order \to Pos$

Pendant la phase de conception, s'étant avéré qu'aucune autre information n'est nécessaire dans la constitution d'une commande, cette sorte est raffinée en un couple de valeurs composés d'une référence et d'une quantité.

spec WHS = ORDER **then**

ops $Mk\_Order : Product \times Pos \rightarrow Order$
$\forall a : Product; b : Pos$
• $Ref(Mk\_Order(a, b)) = a;$
• $Qty(Mk\_Order(a, b)) = b$

Une spécification architecturale est présentée à la fin de l'article. Seule la partie suivante nous intéresse pour notre discussion :

arch spec
units
U_1 : ORDER;
F_1 : ORDER $\rightarrow$ WHS
result $F_1[U_1]$

Cette spécification architecturale n'est pas bien formée. En effet, comme nous l'avons expliqué dans le chapitre 2, la sémantique des unités génériques est l'ensemble des foncteurs d'algèbres persistants définis sur l'ensemble des algèbres modèles de la spécification de départ. Or certains modèles de ORDER ne peuvent pas être prolongés vers des modèles de WHS. Soit $\mathcal{A}$ le modèle de ORDER suivant :

$$\mathcal{A}(Product) = \mathbb{N}, \mathcal{A}(Pos) = \mathbb{N}^+, \mathcal{A}(Order) = \{(0,1); (1,1)\}$$
$$\mathcal{A}(Ref)(0,1) = 2; \mathcal{A}(Ref)(1,1) = 0$$
$$\mathcal{A}(Qty)(0,1) = 3; \mathcal{A}(Qty)(1,1) = 4$$

Ce modèle ne peut être étendu de manière persistante en un modèle de WHS. En effet, cette extension consiste à fournir une réalisation de Mk_Order satisfaisant les axiomes dans WHS qui soit compatible avec les réalisations de $Order$, de Ref et de Qty. Or aucun choix compatible satisfaisant les axiomes n'est possible pour Mk_Order. Le problème vient du fait que la réalisation de la sorte $Order$ vient trop tôt dans la spécification architecturale. Les spécifications nommées ORDER et WHS ne devraient être associées qu'à une seule unité, autrement dit WHS devrait redéfinir ORDER.

5.4. Application : le Contrôle d'accès

5.4.1. Présentation

Le cahier des charges du système de contrôle d'accès est donné dans [Led96]. Cette étude de cas a servi de base dans une comparaison de diverses approches formelles pour le développement de systèmes complexes. Nous reproduisons le cahier des charges original dans l'appendice B.

Résumé du cahier des charges

Il s'agit de gérer l'accès de personnes dans un certain nombre de bâtiments. Les accès ne sont pas libres : toute personne ne peut pas pénétrer à l'intérieur de tous les bâtiments. Les accès sont accordés en fonction de l'affiliation des personnes à un groupe et à un seul à la fois. Ces accès sont mémorisés afin de pouvoir lister les personnes présentes dans un bâtiment à un certain moment ou au moment présent. Ces informations peuvent être utilisées en cas d'incident informatique ou d'accident. Un bâtiment dispose d'un ensemble de portes permettant soit l'entrée, soit la sortie (exclusivement). Chaque porte est munie d'un lecteur de carte magnétique, assignée à une unique personne. Selon l'autorisation, un voyant rouge ou un voyant vert s'allume à la lecture de la carte. S'il est vert, la porte est débloquée par le programme, puis rebloquée 30 secondes après. S'il est rouge, la porte reste bloquée, puis le voyant s'éteint 2 secondes plus tard. Lorsque la porte a été débloquée, une ligne est ajoutée dans un journal des accès spécifiant l'heure et la porte en question. Le système peut également détecter des fumées suspectes par des capteurs spécifiques. Dans ce cas, toutes les portes sont débloquées jusqu'à une réinitialisation du système.

Le système est composé d'unités physiques (lecteurs de cartes, réseau, serrures électromagnétiques, capteurs d'incendie), d'utilisateurs (possédant une carte) et d'un programme. La communication entre utilisateurs et unités physiques se fait par l'intermédiaire des cartes, celle entre unités physiques et le programme par l'intermédiaire de messages formatés (voir section B.4.2) et d'accusés de réception de messages. Les messages d'entrée sont distingués des messages de sortie.

Le comportement de ce système est complètement déterministe. Pour chaque message reçu par le contrôleur, celui-ci ne produit qu'une seule réponse.

5.4.2. Structure préliminaire

Dans cette étude de cas, nous allons suivre la méthode de développement structurée décrite au chapitre 3. Selon la taille du système, une première étape consiste à proposer une architecture du système, en spécifiant des composants ainsi que leurs protocoles de communication. Cette activité n'a pas été décrite dans cette thèse car les langages de spécification algébriques, que nous voulons utiliser, ne permettent pas de décrire une telle architecture. Dans cette architecture, les composants critiques sont identifiés et développés en suivant le processus : phase d'analyse, phase de conception y compris un découpage modulaire, puis codage. Au début de la phase d'analyse, une structure préliminaire est choisie. Cette structure représente la structure du problème, identifiant des traits caractéristiques et leurs relations en terme de définition/utilisation. Cette structure sert de guide lors de la phase d'analyse, en proposant des regroupements d'informations au sein de la spécification des besoins, et aussi pour le découpage modulaire. Précisons encore une fois que le découpage modulaire n'est pas identique à cette structure préliminaire, mais qu'il s'en inspire.

Dans le cas présent, le système peut se décomposer en quatre parties : le programme de contrôle, le système de communication entre ce programme et les unités physiques, une base de données compilant les accès aux bâtiments, et les utilisateurs. La spécification du système de communication par un bus (section A.4.2), ainsi que tous les détails entourant les divers choix techniques possibles pour le mettre en oeuvre, ne sont pas pris en compte dans la suite. En effet, un langage de spécification algébrique comme Casl n'est pas adapté à la spécification d'un médium de communication et de ses pertes éventuelles. L'échange de messages entre les unités physiques et le programme est donc supposé instantané et parfait. La base de données peut être décrite très grossièrement par un type de données algébrique. Nous utiliserons cette possibilité afin de décrire les interactions entre le programme et cette base de données. Les utilisateurs ne sont décrits qu'en tant que membres d'un groupe filtrant l'accès aux bâtiments. Leur comportement n'est pas étudié. Enfin, le fonctionnement du programme de contrôle peut lui être précisément décrit dans CASL. C'est sur lui que se portera notre attention dans la suite.

Le *Control Frame* est le cadre d'étude dans lequel nous pouvons situer le programme de contrôle d'accès. Ce cadre est composé :

- d'un contrôleur, qui est notre programme ;
- du domaine de contrôle, ce qui correspond aux événements partagés entre l'environnement et le programme. Les événements de l'environnement peuvent être distingués en quatre catégories : ceux contrôlés par l'environnement et observables par le programme, ceux contrôlés par l'environnement et non observables par le programme, ceux contrôlés par le programme et observables par l'environnement, ceux contrôlés par le programme et non observables par l'environnement. Les événements partagés sont ceux qui sont contrôlés et observables par l'une ou l'autre partie.
- du comportement attendu des utilisateurs, qui est une contrainte sur les événements et les états du domaine que le contrôleur est sensé maintenir. De ce comportement attendu dérivent les invariants du système.

Pour utiliser ce cadre, il s'agit de décrire le domaine de contrôle et d'identifier les événements partagés entre l'environnement et le programme. Le programme contiendra une description de ces événements partagés.

Les événements contrôlés par l'environnement et observables par le système sont les événements liés à l'utilisation d'un capteur et de l'envoi d'un message au programme. Ce sont l'insertion d'une carte, l'ouverture ou la fermeture d'une porte, la détection de fumées. Le principal événement contrôlé par l'environnement et non observable par le système est le passage effectif ou non d'une personne après l'ouverture d'une porte. Les événements contrôlés par le programme et observables par l'environnement sont liés à l'utilisation d'actuateurs. Ce sont le blocage et le déblocage des portes, l'allumage ou l'extinction des voyants. Un événement contrôlé par le programme et non observable par l'environnement est par exemple l'envoi d'une requête à la base de données.

Un premier élément de la structure préliminaire est donc constitué des informations nécessaires pour décrire les unités physiques de l'environnement observables ou contrôlables par le programme, et les groupes avec les bâtiments auxquels ils ont droit d'accéder. Un deuxième élément contient une description des messages d'entrées/sorties permettant la communication entre ces unités physiques et le programme. Ces messages sont paramétrés par les unités physiques desquelles ils émanent. Cet élément utilise donc le précédent.

En suivant le schéma de spécification d'un système réactif, un élément contiendra une description abstraite de l'état du système avec ses différents observateurs. Les valeurs de retour des observateurs sont liées aux unités physiques. La spécification des états étend donc au moins celle des unités physiques. Un autre élément est la spécification de la fonction de transition, représentant l'exécution du programme.

Le coeur du contrôleur est le protocole d'ouverture de la porte, qui correspond au comportement attendu des utilisateurs. Un élément contiendra la spécification de ce protocole, qui utilisera la description de l'état et sera utilisé par la spécification de la fonction de transition.

On obtient la structure préliminaire suivante :

spec DomaineControle =
%% Les capteurs et actuateurs, ainsi que les groupes avec leurs autorisations

spec Inputs/Outputs = DomaineControle **then**
%% Les messages d'entrée/sorties

spec Etats =DomaineControle **then**
%% Les états du système

spec Envoi = Etats **and** Inputs/Outputs **then**
%% Le protocole d'accès

spec System = Envoi **then**
%% La fonction de transition

Le contrôleur réagit à deux sortes de stimuli : d'une part aux messages émis par les unités physiques, d'autre part au passage du temps (pour temporiser le blocage et le déblocage des portes). Ces deux sortes de stimuli sont de nature différente : les premiers sont discrets (ils dépendent soit de la présence d'utilisateurs, soit d'une incendie), tandis que les seconds sont continus. Leur gestion par la fonction de transition ne peut donc pas être identique. Nous choisissons de favoriser les stimuli temporels. Deux choix de modélisation au moins sont alors possibles. Soit le contrôleur est synchronisé à une horloge externe, lui envoyant régulièrement un battement d'horloge. À chaque battement, les messages provenant des unités physiques sont collectés puis traités, et enfin des messages sont envoyés en retour. C'est le choix de modélisation fait dans [BCPR95]. L'autre choix consiste à munir le contrôleur d'une horloge interne, vue comme une variable d'état quelconque. La fonction de transition correspond alors à un battement de cette horloge, pendant lequel les autres variables d'états sont également mises à jour. Nous suivrons cette modélisation dans notre traitement de l'exemple.

5.4.3. Phase d'analyse

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. Lorsqu'un élément de spécification est relié à une information provenant du cahier des charges, celle-ci est mentionnée au début du paragraphe correspondant.

Environnement physique. *L'environnement est composé de bâtiments ayant des portes et des capteurs d'incendie.* Puisqu'il y a un seul lecteur de cartes par porte il y a bijection entre les lecteurs de cartes et les portes. Nous choisissons la perspective des portes. Une opération associe un bâtiment à chaque porte, et une autre un bâtiment à chaque capteur d'incendie. Ces opérations doivent être surjectives : chaque bâtiment est muni d'au moins un accès et d'un capteur d'incendie. Les sortes nécessaires pour décrire l'environnement sont déclarés dans la spécification suivante :

spec DOMAINE_CONTROLE=
    sorts *Porte*, *Capteur*, *Batiment*
    ops $PorteToBat : Porte \rightarrow Batiment;$
        $CapteurToBat : Capteur \rightarrow Batiment;$
    $\bullet\ \forall b : Batiment \bullet \exists\, p : Porte \bullet PorteToBat(p) = b;$
    $\bullet\ \forall b : Batiment \bullet \exists\, c : Capteur \bullet CapteurToBat(c) = b$

Utilisateurs. *L'accès à un bâtiment est accordé sur la base de l'affiliation d'une personne à un groupe.* L'autre partie du domaine de contrôle est constituée des utilisateurs. On déclare une sorte pour les groupes, et une pour les cartes qui sont en bijection avec les personnes, puisqu'une personne ne peut avoir qu'une seule carte en service à la fois et que les personnes non munies de cartes ne sont pas encore recensées. Une opération associe une carte à un groupe et un prédicat définit les autorisations d'un groupe pour un bâtiment. La modélisation de la constitution des groupes par une opération permet de prendre en compte la demande qu'*une personne appartient à un groupe et un seul*. Cette opération est partielle car on suppose que les cartes signalées perdues ne sont associées à aucun groupe. Un prédicat permet d'identifier les cartes perdues des cartes en circulation. Le domaine de définition de l'opération peut donc être spécifié comme l'ensemble des cartes pour lequel le prédicat est vrai. La spécification des groupes redéfinit la spécification de l'environnement de la manière suivante :

spec DOMAINE_CONTROLE $\stackrel{\text{re}}{=}$ DOMAINE_CONTROLE **then**
    sorts *Groupe*, *Carte*
    op $Appartenir : Carte \rightarrow ?\ Groupe$
    preds $Autorise : Groupe \times Batiment;$
        $EnCirculation : Carte$
    $\forall c : Carte \bullet def(Appartenir(c)) \Leftrightarrow EnCirculation(c)$

Entrées/Sorties Les entrées/sorties du programme sont les messages décrits dans la section B.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. Les accusés de réception sont adressés aux portes ou aux capteurs d'incendie indifféremment. Un type désignant l'ensemble des unités physiques est décrit dans un premier temps dans une redéfinition de DOMAINE_CONTROLE. Les messages prenant en paramètre des identificateurs de portes, de carte, ou de capteur d'incendie, la spécification des messages étend la spécification du domaine de controle dont la sémantique a été étendue par la redéfinition.

spec DOMAINE_CONTROLE $\stackrel{\text{re}}{=}$ DOMAINE_CONTROLE **then**
  free type *PhysicalUnit* ::= **sort** *Porte* | **sort** *Capteur*

spec INPUTS/OUTPUTS= DOMAINE_CONTROLE **then**
  free type *R_Messages* ::= *Lecture*(*Porte*, *Carte*) | *Ouverture*(*Porte*)
    | *Fermeture*(*Porte*) | *Incendie*(*Capteur*) |*Okr*(*PhysicalUnit*);
  free type *S_Messages* ::= *Bloquer*(*Porte*) | *Debloquer*(*Porte*) |
    Allumer_rouge(*Porte*) | *Allumer_vert*(*Porte*) |
    Eteindre_rouge(*Porte*) | *Eteindre_vert*(*Porte*) | *Oks*(*PhysicalUnit*);

Journal des accès. *Tout accès est enregistré (personne, bâtiment, heure).* Le journal des accès, qui constitue la base de données dynamique, est décrit sous la forme d'un type abstrait sur lequel on dispose d'une opération d'ajout de données. Une donnée est ajoutée dans le journal après la réception d'un message d'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 à laquelle le message d'ouverture de la porte a été enregistrée.

Nous avons ici un exemple d'information concernant plusieurs traits du système. En effet, pour pouvoir spécifier le journal, nous devons disposer d'une description de la date. Comme nous l'avons indiqué ci-dessus, la date est considérée comme une variable d'état du système, variant avec la fonction de transition. L'état du système est spécifié de manière abstraite comme une sorte munie des observateurs correspondant aux informations dont les diverses fonctions ont besoin. Un premier observateur est défini pour donner la date courante du système. La date est définie elle-même comme une sorte abstraite, sans lui donner un format particulier, dans une redéfinition de la spécification du domaine de controle. Les autres observateurs ne sont pas connus pour l'instant.

La fonction de transition partielle associée à cet observateur consiste simplement à incrémenter le temps de la durée d'un cycle. Cette donnée n'étant pas fournie dans le cahier des charges, nous laissons cette durée abstraite. Une fonction *Tick* qui modifie la date courante est ajoutée à la spécification du domaine de contrôle.

La spécification du type des états étend celle du domaine de contrôle, dont la sémantique a été mise à jour.

spec DOMAINE_CONTROLE $\stackrel{\text{re}}{=}$ DOMAINE_CONTROLE **then**
  sort *Date*
  op *Tick* : *Date* $\rightarrow$ *Date*

spec ETATS = DOMAINE_CONTROLE **then**
  sort *Etats*
  op *Init* : *Etats*;
    Obs_DateCourante : *Etats* $\rightarrow$ *Date*;

spec JOURNAL= DOMAINE_CONTROLE **then**
  type *Journal* ::= *Empty*
    | *Enregistrer*(*Carte*; *Porte*; *Date*; *Journal*)

Protocole d'accès. Nous traduisons maintenant le protocole d'accès, en formant un scénario précis puis en formalisant ce scénario. L'utilisateur introduit sa carte dans le lecteur associé à une porte. Un message de lecture de carte est envoyé au contrôleur. Si cette carte est valide, recensée comme étant en circulation, que la personne appartient à un groupe autorisé à pénétrer dans le bâtiment, et qu'elle ne soit pas déjà considérée comme présente dans le bâtiment si elle veut y entrer ou absente si elle veut en sortir, alors le voyant vert de ce lecteur est allumé et la porte associée est débloquée. Un minuteur se met en marche, laissant trente secondes avant la réception d'un message d'ouverture de la porte débloquée. Si le message est reçu dans cet intervalle de temps, le journal est mis à jour, puis le programme attend un message de fermeture provenant de la porte avant de la rebloquer et d'éteindre le voyant vert. Sinon la porte est rebloquée et le voyant vert est éteint. Si une des conditions de départ n'est pas satisfaite, le voyant rouge est allumé pendant deux secondes, puis éteint. Nous appelons *s*ection critique la situation reliée à l'état d'une porte lors de la mise en route du délai de réception d'un message d'ouverture ou du délai pendant lequel le voyant rouge est allumé.

Le coeur de la spécification est la fonction *Envoi* qui calcule les messages à envoyer, en fonction de l'état du système et des messages reçus. 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 ait expiré. Il y a deux sortes de sections critiques selon le voyant allumé. Si c'est le voyant vert, alors une section critique est ouverte lorsqu'une porte est débloquée et est refermée lorsque le message d'ouverture de la porte a été reçu. Si c'est le voyant rouge, la section critique est ouverte au moment où le voyant est allumé et se referme deux secondes plus tard. Il peut donc y avoir plusieurs messages à envoyer simultanément aux unités physiques associées à une même porte ou bien à plusieurs (débloquer une porte et allumer un voyant vert, ou bien fermer une section critique et répondre à une demande, etc.).

Nous allons reprendre progressivement ce scénario afin de le formaliser. La spécification des ensembles est introduite pour pouvoir donner le profil de l'opération d'envoi. Cette spécification contient une déclaration du type des ensembles d'éléments quelconques ainsi que les opérations et prédicats usuels sur les ensembles. La spécification des messages est ensuite redéfinie pour intégrer cette spécification des ensembles. La spécification du système contient la fonction de transition. Les étiquettes des transitions sont les messages éventuellement reçus dans une boucle.

$$
\begin{array}{l}
\textbf{spec}\ \textsc{Set}[\textbf{sort}\ \mathit{Elem}] = \\
\quad \textbf{generated type}\ \mathit{Set}[\mathit{Elem}] ::= \mathit{EmptySet} \mid \{\_\_\}(\mathit{Elem}) \mid \_\_ + \_\_(\mathit{Elem}; \mathit{Set}[\mathit{Elem}]) \\
\quad \textbf{ops}\quad \_\_ \cup \_\_ : \mathit{Set}[\mathit{Elem}] \times \mathit{Set}[\mathit{Elem}] \to \mathit{Set}[\mathit{Elem}]; \\
\quad \phantom{\textbf{ops}}\quad \_\_ \cap \_\_ : \mathit{Set}[\mathit{Elem}] \times \mathit{Set}[\mathit{Elem}] \to \mathit{Set}[\mathit{Elem}]; \\
\quad \phantom{\textbf{ops}}\quad \_\_ \backslash \_\_ : \mathit{Set}[\mathit{Elem}] \times \mathit{Set}[\mathit{Elem}] \to \mathit{Set}[\mathit{Elem}]; \\
\quad \textbf{preds}\ \_\_ \in \_\_ : \mathit{Elem} \times \mathit{Set}[\mathit{Elem}] \\
\quad \phantom{\textbf{preds}}\ \_\_ \subseteq \_\_ : \mathit{Set}[\mathit{Elem}] \times \mathit{Set}[\mathit{Elem}]
\end{array}
$$

$$
\begin{array}{l}
\textbf{spec}\ \textsc{Inputs/Outputs} \stackrel{\mathrm{re}}{=} \textsc{Set}[\textsc{Inputs/Outputs}\ \textbf{with}\ \mathit{Elem} \mapsto \mathit{R\_Messages}] \\
\quad \textbf{and}\ \ \textsc{Set}[\textsc{Inputs/Outputs}\ \textbf{with}\ \mathit{Elem} \mapsto \mathit{S\_Messages}]
\end{array}
$$

spec ENVOI = ETATS **and** INPUTS/OUTPUTS **then**
ops $Envoi : Set[R\_Messages] \times Etats \rightarrow Set[S\_Messages]$

spec SYSTEME = ETATS **and** INPUTS/OUTPUTS **then**
op $T : Set[R\_Messages] \times Etats \rightarrow Etats$

Préconditions de l'opération d'envoi. Nous formalisons les différents tests effectués dans le protocole avant chaque envoi de messages.

La validité d'une carte peut être testée. Dans le paragraphe B.3.1 est donnée une description précise du code utilisé pour chaque carte. Ce code comprend une partie destinée à empêcher les utilisations frauduleuses ou les erreurs de lectures. Dans un premier temps, nous déclarons simplement un prédicat testant la validité d'une carte, sans spécifier la description du code, en indiquant que les cartes en circulation doivent être valides. Ce prédicat se place avec la spécification des cartes, que l'on redéfinit :

spec DOMAINE_CONTROLE $\stackrel{\text{re}}{=}$ DOMAINE_CONTROLE **then**
pred $Valide : Carte;$
$\forall c : Carte \bullet EnCirculation(c) \Rightarrow Valide(c)$

On ne peut sortir d'un bâtiment que si on y est préalablement entré. Une précision apportée dans les discussions suivant le cahier des charges indique qu'une porte ne peut être utilisée qu'en entrée ou en sortie, mais pas les deux à la fois. Ainsi une porte est soit une porte d'entrée, soit une porte de sortie. Nous ajoutons deux prédicats s'excluant mutuellement pour modéliser ce point, puis des axiomes indiquant qu'un batiment doit posséder au moins une porte d'entrée et une porte de sortie. Ces prédicats et axiomes sont ajoutés dans une redéfinition de la spécification de l'environnement. Si nous avions utilisé une structuration plus classique de la spécification, ces informations auraient dû être réorganisées pendant la phase de découpage architectural pour obtenir une spécification architecturale bien formée. Une requête sur le journal des accès, pour savoir si une personne est à l'intérieur d'un bâtiment donné, est enfin déclarée.

spec DOMAINE_CONTROLE $\stackrel{\text{re}}{=}$ DOMAINE_CONTROLE **then**
preds $Is\_in, Is\_out : Porte$
$\forall p : Porte;$
$\bullet\ Is\_in(p) \Leftrightarrow \neg\, Is\_out(p);$
$\bullet\ \forall b : Batiment \bullet (\exists p1 : Porte \bullet (Is\_in(p1) \wedge PorteToBat(p1) = b))$
$\wedge\ (\exists p2 : Porte \bullet (Is\_out(p2) \wedge PorteToBat(p2) = b))$

spec JOURNAL $\stackrel{\text{re}}{=}$ JOURNAL **then**
pred $Etre\_dans : Carte \times Batiment \times Journal$

Nous définissons enfin deux prédicats testant l'un l'autorisation d'entrée, l'autre l'autorisation de sortie en regroupant les prédicats précédents. Ces prédicats sont localisés dans la même spécification nommée que la fonction d'envoi :

spec ENVOI $\stackrel{\text{re}}{=}$ ENVOI **and** JOURNAL **then**

$$\begin{array}{l} \textbf{pred } \mathit{Autorisation\_in}(c : \mathit{Carte}; p : \mathit{Porte}; j : \mathit{Journal}) \Leftrightarrow \\ \quad \mathit{EnCirculation}(c) \wedge \mathit{Is\_in}(p) \wedge \neg \exists\, b \bullet \mathit{Etre\_dans}(c, b, j) \\ \qquad \wedge\ \mathit{Autorise}(\mathit{Appartenir}(c), \mathit{PorteToBat}(p)); \\ \textbf{pred } \mathit{Autorisation\_out}(c : \mathit{Carte}; p : \mathit{Porte}; j : \mathit{Journal}) \Leftrightarrow \\ \quad \mathit{EnCirculation}(c) \wedge \mathit{Is\_out}(p) \wedge \mathit{Etre\_dans}(c, \mathit{PorteToBat}(p), j) \end{array}$$

Calcul des réponses du programme. Comme nous l'avons mentionné précédemment, les réponses du programme sont de deux sortes : soit il s'agit d'une réponse à un message reçu, soit d'une réponse à l'expiration d'un délai. Dans le premier cas, nous effectuons une analyse par cas sur l'ensemble des messages reçus.

Si cet ensemble contient un message de détection d'incendie, alors toutes les portes correspondant au bâtiment dans lequel se trouve ce capteur sont ouvertes. Si cet ensemble contient un message de lecture d'une carte, alors on teste si le prédicat d'autorisation d'entrée ou de sortie (selon le type de la porte) est vrai. Si oui, alors l'ensemble des messages envoyés contient un message de déblocage de la porte et d'allumage du voyant vert correspondant. Pour tester ce prédicat, on doit disposer du journal courant du système afin de vérifier que la personne n'est pas (resp. est) présente dans le bâtiment si elle veut y entrer (resp. sortir). Un observateur est ajouté à la structure des états pour fournir cette donnée. Lorsque la porte est débloquée, celle-ci rentre dans une section critique pendant laquelle aucune interaction autre que la réception du message d'ouverture de la porte n'est autorisée. Si la personne n'est pas autorisée à entrer ou à sortir, le voyant rouge correspondant est allumé. Pour gérer le délai de deux secondes d'allumage du voyant rouge, une autre forme de section critique est définie (dite de refus).

Pour gérer le passage d'une porte en section critique, un observateur contenant la liste des portes en section critique est défini. Une section critique est identifiée par une porte, une carte, le temps restant avant la fermeture de cette section critique, et le statut pour lequel cette section a été ouverte (cas d'attente du message d'ouverture, ou de refus). Dans la spécification nommée DOMAINE_CONTROLE, nous ajoutons une sorte pour le temps, ainsi que des constantes pour le temps imparti avant une ouverture ou pour un refus, ou pour identifier la fin du délai. Nous déclarons ensuite un type de données pour les statuts, puis pour les sections critiques en tant que type produit, pour pouvoir ajouter un observateur indiquant la liste courante des sections critiques ouvertes dans la spécification ETATS.

Si l'ensemble des messages reçus contient un message de fermeture d'une porte, alors la porte correspondante est rebloquée et la section critique correspondante est refermée. Comme il s'agit alors de la mise à jour d'un observateur, ce point sera traité à part. Dans le cas où l'ensemble contient un message d'ouverture, aucun message particulier n'est envoyé mais le journal est mis à jour. Comme dans le cas des sections critiques, il s'agit de la mise à jour d'un observateur, qui sera donc traité à part. Le cas des accusés de réception sera traité dans la phase de conception.

La deuxième sorte de réponses correspond à la fermeture d'une section critique. S'il s'agit de la fermeture d'une section critique d'ouverture, le programme envoi un message de reblocage de la porte et d'extinction du voyant vert. S'il s'agit de la fermeture d'une section critique de refus, le programme envoi un message d'extinction du voyant rouge.

La spécification de la fonction d'envoi utilise la structure ensembliste des messages reçus et envoyés, afin de ne pas imposer d'implémentation particulière. La spécification des listes est identique à celle du chapitre 2.

spec DOMAINE_CONTROLE $\stackrel{re}{=}$ DOMAINE_CONTROLE **then**
  sort *Time*
  ops $delai\_ouverture, delai\_refus, 0 : Time;$

spec ETATS $\stackrel{re}{=}$ ETATS **and** JOURNAL **then**
  free type $Status ::= Ok \mid Reject$
  free type $SecCri ::= < Porte : Porte, Carte : Carte, Delai : Time, Statut : Status >$
then LIST$[Elem \mapsto SecCri]$
then op $Obs\_Journal : Etats \to Journal;$
  $Obs\_SecCri : Etats \to List[SecCri]$
  $\bullet\ Obs\_Journal(Init) = Empty;$
  $\bullet\ Obs\_SecCri(Init) = Nil;$

spec ENVOI $\stackrel{re}{=}$ ENVOI **then**
  $\forall Rmsgs : Set[R\_Messages]; p : Porte; c : Carte; cap : Capteur; b : Batiment; j : Journal; q : Etats;$
  %% Message : incendie
  $\bullet\ Incendie(cap) \in Rmsgs$
    $\Rightarrow (\forall p : Porte \bullet PorteToBat(p) = CapteurToBat(cap)$
      $\Rightarrow Debloquer(p) \in Envoi(Rmsgs, q));$
  %% Message : lecture, autorisé
  $\bullet\ Lecture(p, c) \in Rmsgs \wedge$
    $(Autorisation\_in(c, p, Obs\_Journal(q)) \vee Autorisation\_out(c, p, Obs\_Journal(q)))$
      $\Rightarrow Allumer\_vert(p) \in Envoi(Rmsgs, q) \wedge Debloquer(p) \in Envoi(Rmsgs, q)$
  %% Message : lecture, refus
  $\bullet\ Lecture(p, c) \in Rmsgs \wedge$
    $\neg(Autorisation\_in(c, p, Obs\_Journal(q)) \vee Autorisation\_out(c, p, Obs\_Journal(q)))$
      $\Rightarrow Allumer\_rouge(p) \in Envoi(Rmsgs, q);$
  %% Message : fermeture
  $\bullet\ Fermeture(p) \in Rmsgs$
    $\Rightarrow Bloquer(p) \in Envoi(Rmsgs, q) \wedge Eteindre\_vert(p) \in Envoi(Rmsgs, q);$
  %% Délai ouverture expiré
  $\bullet\ \forall sc : SecCri \bullet (sc \in Obs\_SecCri(q) \wedge Delai(sc) = 0 \wedge Statut(sc) = Ok)$
    $\Rightarrow Bloquer(Porte(sc)) \in Envoi(Rmsgs, q) \wedge Eteindre\_vert(p) \in Envoi(i, q);$
  %% Délai refus expiré
  $\bullet\ \forall sc : SecCri \bullet (sc \in Obs\_SecCri(q) \wedge Delai(sc) = 0 \wedge Statut(sc) = Reject$
    $\Rightarrow Eteindre\_rouge(Porte(sc)) \in Envoi(Rmsgs, q);$

La structure des états est donc enrichie de deux observateurs durant cette partie du processus de spécification. Le premier observateur contient le journal courant du système, et le second la liste des portes en section critique. Ceci accrédite notre utilisation des redéfinitions pour la spécification de cette étude de cas. La propagation de la sémantique de la spécification des états permet de ne pas y faire référence de nouveau lors de la redéfinition de ENVOI. Des fonctions de transition partielles doivent donc être définies pour chacun de ces observateurs.

Mise à jour des observateurs. La boucle de contrôle principale effectue une transition d'un état à un autre tous les cycles, dont la durée correspond à une mise à jour de l'observateur de la date courante (fonction $Tick$). Pendant ce temps, le programme collecte les messages, détermine des réponses, et modifie la valeur de ses variables d'états. Concrètement, le journal courant est modifié lorsqu'un message d'ouverture d'une porte est présent, que l'on identifie avec le fait qu'une personne vient d'entrer dans un bâtiment. La mise à jour du journal est effectuée en retrouvant la section critique correspondante et en enregistrant le nouvel accès avec la carte utilisée, la porte concernée et la date de franchissement. Une opération datant un message en fonction de l'état courant est définie dans cette spécification. Cette opération est spécifiée en fonction de l'observateur de la date courante. Comme le système de transmission des messages est supposé parfait et instantané, la date donnée à un message reçu est identique à celle au moment de l'émission du message par l'unité physique, c'est à dire la date courante.

spec MAJ_JOURNAL = ETATS **and** INPUTS/OUTPUTS **then**
  ops $New\_Journal : Set[R\_Messages] \times Etats \rightarrow Journal;$
    $Dater : R\_Messages \times Etats \rightarrow Date;$
  $\forall Rmsgs : Set[R\_Messages]; p : Porte; q : Etats, rm : R\_Messages;$
  • $Dater(rm, q) = Obs\_DateCourante(q);$
  • $Ouverture(p) \in Rmsgs \wedge \exists\, sc : SecCri \bullet (sc \in Obs\_SecCri(q) \wedge Porte(sc) = p)$
    $\Rightarrow New\_Journal(Rmsgs, q) =$
      $Enregistrer(Carte(sc), p, Dater(Ouverture(p), q), Obs\_Journal(q))$

La liste des sections critiques est modifiée de trois façons : soit le délai imparti pour une section critique est écoulé, dans ce cas la section critique concernée est enlevée de la liste courante ; pour toutes celles restantes, leur délai est diminué de la durée d'un cycle ; de nouvelles sections sont ajoutées pour toute transaction avec un utilisateur. La diminution du délai dans les sections critiques est relié à la durée d'un cyle, que nous avons laissée abstraite pour l'instant. Nous définissons, dans une extension de la spécification du domaine de contrôle, une sorte $Time$ pour les durées munie d'une opération reflétant la transition d'un état à un autre. Nous surchargeons pour cela l'opération $Tick$ sur les dates en indiquant qu'un battement d'horloge sur la date courante diminue les délais de la même durée. Contrairement à un battement sur les dates qui «augmente» la date courante, un battement sur les durées diminue leur valeur. Le dernier axiome relie les deux utilisations de la fonction $Tick$. On obtient les deux spécifications suivantes :

spec DOMAINE_CONTROLE $\stackrel{\text{re}}{=}$ DOMAINE_CONTROLE **then**
  ops $Tick : Time \rightarrow Time;$
    $\_\_ - \_\_ : Date \times Date \rightarrow Date;$

$\_\_ - \_\_ : Time \times Time \rightarrow Time$
$\forall t : Time; d : Date$
• $Tick(t) < t$;
• $0 \leq Tick(t)$;
• $Tick(d) - d = t - Tick(t)$;

spec MAJ_SECCRI = ETATS **and** INPUTS/OUTPUTS **then**
op $New\_SecCri : Set[R\_Messages] \times Etats \rightarrow List[SecCri]$;
$\forall Rmsgs : Set[R\_Messages]; p : Porte; q : Etats; c : Carte$
%% Les sections critiques terminées sont éliminées
• $\forall sc : SecCri(sc \in Obs\_SecCri(q) \wedge Delai(sc) = 0)$
$\Rightarrow sc \notin New\_SecCri(Rmsgs, q)$;
%% Les autres sont conservées en diminuant le délai
• $sc : SecCri(sc \in Obs\_SecCri(q) \wedge Delai(sc) \neq 0)$
$\Rightarrow < Porte(sc), Carte(sc), Tick(Delai(sc)), Statut(sc) > \in New\_SecCri(Rmsgs, q)$;
%% De nouvelles sections sont ajoutées
• $lecture(p, c) \in Rmsgs$
$\wedge (Autorisation\_in(c, p, Obs\_Journal(q)) \vee Autorisation\_out(c, p, Obs\_Journal(q)))$
$\Rightarrow < p, c, delai\_ouverture, Ok > \in New\_SecCri(Rmsgs, q)$;
• $lecture(p, c) \in Rmsgs$
$\wedge \neg(Autorisation\_in(c, p, Obs\_Journal(q)) \vee Autorisation\_out(c, p, Obs\_Journal(q)))$
$\Rightarrow < p, c, delai\_refus, Reject > \in New\_SecCri(Rmsgs, q)$;

La spécification de la fonction de transition peut alors être ébauchée à partir de ces fonctions de mises à jour, en particulier celle concernant la date courante. L'application de la fonction de transition correspond au passage d'un cycle.

spec SYSTEME $\stackrel{\text{re}}{=}$ SYSTEME **and** MAJ_JOURNAL **and** MAJ_SECCRI **then**
$\forall q : Etats; Rmsgs : Set[R\_Messages]$
• $Obs\_DateCourante(T(Rmsgs, q)) = Tick(Obs\_DateCourante(q))$;
• $Obs\_Journal(T(Rmsgs, q)) = New\_Journal(Rmsgs, q)$;
• $Obs\_SecCri(T(Rmsgs, q)) = New\_SecCri(Rmsgs, q)$

La structure de la spécification à la fin de cette phase d'analyse est représentée dans la figure 5.1.

5.4.4. Phase de conception

Durant la phase de conception, nous allons compléter la description du comportement du programme, en incorporant le traitement des accusés de réception, et en étudiant les comportements possibles dans les cas limites. Les accusés de réception sont envoyés soit par les unités physiques lors de la réception d'un message provenant du programme, soit par le programme lors de la réception d'un message provenant des différents capteurs. Il sera intéressant de traiter le cas où ces accusés de réception ne sont pas correctement transmis, afin de détecter d'éventuels problèmes techniques.

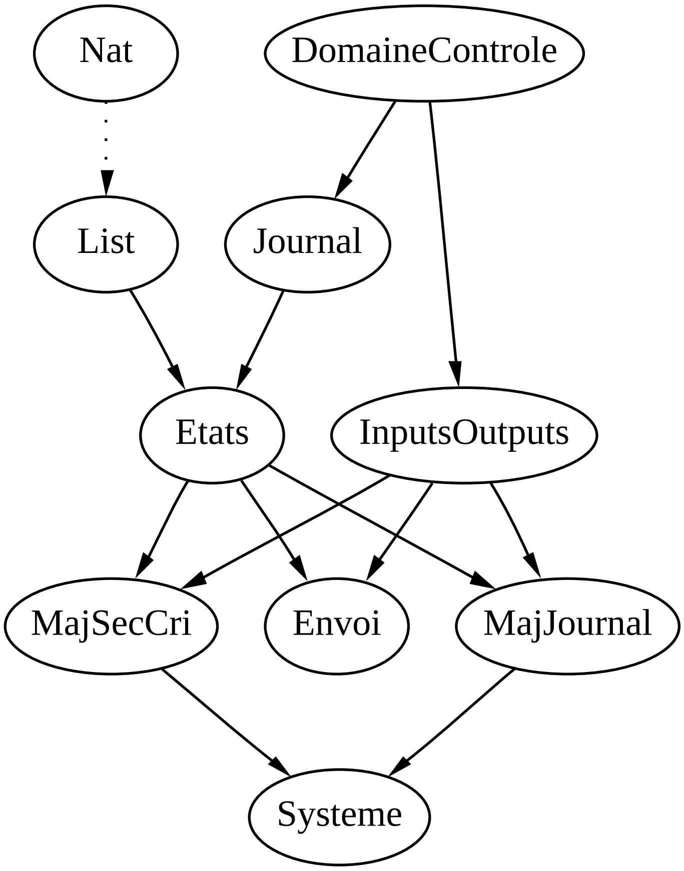

FIG. 5.1.: Phase d'analyse - Structure de la spécification

Le comportement idéal des utilisateurs est décrit dans le paragraphe B.3.3 ainsi que dans la section de questions/réponses. Cependant, certaines actions sont néanmoins possibles, que ce soit par malveillance ou mauvaise utilisation. Par exemple, le programme peut détecter l'ouverture d'une porte sans que celle-ci soit débloquée auparavant, ou bien qu'une porte ne soit pas refermée après son ouverture. De plus, des équipements matériels peuvent être endommagés, nuisant ainsi au bon fonctionnement du système. Le programme doit pouvoir réagir à ces comportements. Dans certains cas, il peut ordonner des actions correctives particulières, comme dans le cas d'une détection d'incendie où toutes les portes d'un bâtiment sont ouvertes. Mais en général, il ne sera pas possible d'agir automatiquement. Dans ce cas, soit le cadre de cette étude de cas est modifiée, soit l'information n'est que stockée dans le journal en tant que problème technique.

Après avoir traité ces situations, nous transformerons la structure des états en un ensemble structuré de variables d'état, puis nous décrirons la fonction de transition à partir des différentes fonctions de mise à jour. Cette transformation de la spécification nous permettra enfin de proposer un découpage architectural pour cette étude de cas.

Envoi des accusés de réception. Pour déterminer les accusés de réception de messages provenant des capteurs, il suffit de parcourir l'ensemble des messages reçus puis d'identifier ceux qui ne sont pas eux-mêmes des accusés de réception et enfin d'envoyer l'accusé correspondant. Pour formaliser cette action, nous introduisons une fonction évaluant l'identificateur du capteur associé à un message. Cette opération est ajoutée à la spécification des messages, puis la spécification Envoi est étendue par l'axiome formalisant cette action :

spec $\textsc{Inputs/Outputs} \stackrel{\text{re}}{=} \textsc{Inputs/Outputs}$ **then**
  ops $\mathit{RMessToPhysUnit} : \mathit{R\_Messages} \rightarrow \mathit{PhysicalUnit};$
    $\mathit{SMessToPhysUnit} : \mathit{S\_Messages} \rightarrow \mathit{PhysicalUnit}$

spec $\textsc{Envoi} \stackrel{\text{re}}{=} \textsc{Envoi}$ **then**
  $\forall \mathit{Rmsgs} : \mathit{Set}[\mathit{R\_Message}]; \mathit{PU} : \mathit{PhysicalUnit}; q : \mathit{Etats}; \mathit{rm} : \mathit{R\_Messages}$
  $\bullet\ \mathit{rm} \in \mathit{Rmsgs} \wedge \mathit{rm} \neq \mathit{Okr}(\mathit{PU}) \Rightarrow \mathit{Oks}(\mathit{RMessToPhysUnit}(\mathit{rm})) \in \mathit{Envoi}(\mathit{Rmsgs}, q)$

Détection d'une intrusion. Nous appelons intrusion le fait de recevoir un message d'ouverture d'une porte sans qu'aucune carte n'ait été lue pour cela. Une extension du cahier des charges propose (section B.5) de relier le système à un poste de gardiennage pour intervenir dans ce cas. Cela modifie le cadre, en étendant les actuateurs disponibles. Nous supposerons que ce poste de gardiennage visualise l'évolution du système par le biais du journal des accès. Nous proposons alors comme action corrective le fait d'ajouter un constructeur à ce type afin de mettre en évidence cette intrusion. Pour formaliser la détection, nous utilisons la liste des sections critiques de l'état courant. Lorsqu'une carte est lue et que l'autorisation est accordée, une section critique d'ouverture est créée. C'est le seul cas de démarrage d'une section critique d'ouverture. Donc si un message d'ouverture d'une porte est reçu et qu'aucune section critique n'est ouverte pour la même porte, alors une détection d'intrusion est ajoutée au journal.

spec $\textsc{Journal} \stackrel{\text{re}}{=} \textsc{Journal}$ **then**

$$\textbf{op} \quad \mathit{Intrusion} : \mathit{Porte} \times \mathit{Date} \times \mathit{Journal} \rightarrow \mathit{Journal}$$

$$\begin{array}{l}
\textbf{spec}\ \textsc{Maj_Journal} \stackrel{\text{re}}{=} \textsc{Maj_Journal}\ \textbf{then} \\
\quad \forall i : \mathit{Set}[\mathit{R_Messages}]; ; p : \mathit{Porte}; q : \mathit{Etats}; sc : \mathit{SecCri} \\
\quad \bullet\ \mathit{Ouverture}(p) \in i \wedge \neg(\exists\, sc : \mathit{SecCri} \bullet (sc \in \mathit{Obs_SecCri}(q) \wedge \mathit{Porte}(sc) = p) \\
\qquad\quad \Rightarrow \mathit{New_Journal}(i, q) = \\
\qquad\qquad \mathit{Intrusion}(\mathit{Porte}(sc), \mathit{Dater}(\mathit{Ouverture}(p), q), \mathit{Obs_Journal}(q))
\end{array}$$

Détection de défaillances. La détection des défaillances des équipements du système (ou du médium de communication) provient en général de la comparaison entre les messages reçus et la prévision de certains événements dans le système. Si ces événements ne sont pas perçus au moment où ils devraient l'être, le programme infère qu'un erreur matérielle a dû se produire. Il est donc nécessaire d'introduire des variables d'états spécifiques traduisant ces prévisions. Une fois ces défaillances détectées, le programme dispose de plusieurs moyens d'action. Par exemple, son comportement peut être modifié lors de la réception de messages en provenance des équipements supposés défaillants. Un mécanisme doit être ensuite disponible pour permettre le retour à un état normal, utilisant des messages spécifiques. Ce principe est utilisé dans le système décrit dans [Abr96b]. Dans notre cas, les messages échangés entre le programme et les équipements ne comprennent pas un tel mécanisme. À moins d'étendre considérablement le cahier des charges par de nombreuses hypothèses, nous ne pouvons suivre cette voie. En revanche, nous pouvons noter dans le journal des accès qu'une défaillance a été détectée. À charge d'un intervenant humain de réparer cet équipement dans les plus brefs délais, sans que le comportement du programme ne soit modifié entre-temps concernant cet équipement.

Pour formaliser cette détection, nous utilisons les accusés de réception que doivent envoyer les équipements après avoir reçu un message du programme. Comme nous supposons le médium de communication parfait et instantané, la prévision consiste à attendre le retour d'un accusé de réception provenant d'une unité physique au cycle immédiatement suivant l'envoi du message à cette unité. Nous étendons la structure des états par un observateur sur le dernier ensemble de messages envoyés par le programme, auquel on soustrait les accusés de réception vers les équipements afin de ne pas boucler indéfiniment. Ainsi, la détection d'une défaillance se fait en comparant le dernier ensemble de messages reçus avec cet observateur. Celui-ci est mis à jour avec l'ensemble de messages déterminé par la fonction d'envoi. Une fonction intermédiaire, *CollectSAccuse*, regroupe les accusés de réception envoyés aux unités afin de les soustraire lors de la mise à jour de l'observateur de messages envoyés.

$$\begin{array}{l}
\textbf{spec}\ \textsc{Etats} \stackrel{\text{re}}{=} \textsc{Etats}\ \textbf{and}\ \textsc{Inputs/Outputs}\ \textbf{then} \\
\quad \textbf{op} \quad \mathit{Obs_SMess} : \mathit{Etats} \rightarrow \mathit{Set}[\mathit{S_Messages}]; \\
\quad \bullet\ \mathit{Obs_SMess}(\mathit{Init}) = \mathit{EmptySet};
\end{array}$$

$$\begin{array}{l}
\textbf{spec}\ \textsc{Maj_SMess} = \textsc{Envoi}\ \textbf{then} \\
\quad \textbf{ops} \quad \mathit{CollectSAccuse} : \mathit{Set}[\mathit{S_Message}] \rightarrow \mathit{Set}[\mathit{S_Message}]; \\
\qquad\quad\ \ \mathit{New_SMess} : \mathit{Set}[\mathit{R_Message}] \times \mathit{Etats} \rightarrow \mathit{Set}[\mathit{S_Message}];
\end{array}$$

$\forall Rmsgs : Set[R\_Messages]; Smsgs : Set[S\_Messages]; rm : R\_Messages;$
$sm : S\_Messages; q : Etats; PU : PhysicalUnit;$
- $CollectSAccuse(EmptySet) = EmptySet;$
- $sm = Oks(PU) \Rightarrow CollectSAccuse(sm, Smsgs) = CollectSAccuse(Smsgs);$
- $sm \neq Oks(PU) \Rightarrow CollectSAccuse(sm, Smsgs) = sm + CollectSAccuse(Smsgs);$

%% Mise à jour des messages envoyes
- $New\_SMess(Rmsgs, q) = Envoi(Rmsgs, q) \setminus CollectSAccuse(Envoi(Rmsgs, q))$

L'action corrective consistant simplement à noter dans le journal la détection d'une défaillance, cette formalisation se trouve dans la spécification de la mise à jour du journal. Une opération est ajoutée dans la spécification du journal pour pouvoir signaler une défaillance, et une constante représentant la date courante est ajoutée dans la spécification contenant les aspects temporels. En effet la date de détection exacte d'une défaillance étant inconnue par manque d'informations, la date courante est utilisée pour approximer cette date. Une fonction auxiliaire *ErreurAccuse* compile l'ensemble des identificateurs d'unités physiques pour lesquelles soit un accusé attendu n'a pas été reçu, soit un accusé non attendu a été reçu. Cette fonction est ensuite utilisée lors de la mise à jour du journal des accès.

spec JOURNAL $\stackrel{re}{=}$ JOURNAL **then**
op $Defaillance : PhysicalUnit \times Date \times Journal \rightarrow Journal$

spec ERREUR_ACCUSE = LIST[ETATS **with** $Elem \mapsto PhysicalUnit$] **then**
op $ErreurAccuse : Set[S\_Message] \times Set[R\_Message] \rightarrow List[PhysicalUnit]$
$\forall Rmsgs : Set[R\_Messages]; Smsgs : Set[S\_Messages];$
$sm : S\_Messages; rm : R\_Messages; PU : PhysicalUnit;$
- $ErreurAccuse(EmptySet, EmptySet) = Nil;$
- $rm = Okr(PU) \Rightarrow$
  $ErreurAccuse(EmptySet, (rm + Rmsgs)) =$
  $Cons(PU, ErreurAccuse(EmptySet, Rmsgs));$
- $rm \neq Okr(PU) \Rightarrow$
  $ErreurAccuse(EmptySet, (rm + Rmsgs)) = ErreurAccuse(EmptySet, Rmsgs);$
- $Okr(SMessToPhysUnit(sm)) \in Rmsgs \Rightarrow$
  $ErreurAccuse((sm + Smsgs), Rmsgs) =$
  $Cons(SMessToPhysUnit(sm), ErreurAccuse(Smsgs, Rmsgs));$
- $Okr(SMessToPhysUnit(sm)) \notin Rmsgs \Rightarrow$
  $ErreurAccuse((sm + Smsgs), Rmsgs) = ErreurAccuse(Smsgs, Rmsgs);$

spec MAJ_JOURNAL $\stackrel{re}{=}$ MAJ_JOURNAL **and** ERREUR_ACCUSE **then**
$\forall Rmsgs : Set[R\_Messages]; q : State; PU : PhysicslUnit$
- $\neg Is\_Empty(ErreurAccuse(Obs\_SMess(q), Rmsgs)) \Rightarrow$
  $\forall PU : PhysicalUnit \bullet PU \in ErreurAccuse(Obs\_SMess(q), Rmsgs) \Rightarrow$
  $New\_Journal(Rmsgs, q) =$
  $Defaillance(PU, Obs\_DateCourante(q), Obs\_Journal(q))$

Transformation de la structure des états. Arrivé à ce stade du processus de spécification, nous pouvons considérer que les observateurs nécessaires ont tous été identifiés précédemment et que les seuls raffinements à effectuer vont concerner les algorithmes utilisés pour les opérations ou bien préciser les configurations des bâtiments et des groupes.[2] Nous pouvons donc procéder à la transformation de la structure des états sous la forme d'un ensemble structuré de variables d'états. Comme nous l'avons explicité dans la section 5.3, les variables d'états sont issues des observateurs. La boucle de contrôle est ensuite formalisée à partir des différentes opérations de mise à jour.

spec $\textsc{Etats} \stackrel{re}{=} \textsc{Etats}$ **then**
    free type $Etats ::= <\_\_, \_\_, \_\_>$
        $(jou : Journal; seccri : List[SecCri]; smess : Set[S\_Messages], datecourante : Date)$
    $\forall q : Etats$
    • $Obs\_DateCourante(q) = datecourante(q);$
    • $Obs\_Journal(q) = jou(q);$
    • $Obs\_SecCri(q) = seccri(q);$
    • $Obs\_SMess(q) = SMess(q);$

spec $\textsc{Systeme} \stackrel{re}{=} \textsc{Systeme}$ **and** $\textsc{Maj\_SMess}$ **then**
%% Boucle de controle
    • $T(Rmsgs, q) =< New\_Journal(Rmsgs, q); New\_SecCri(Rmsgs, q);$
        $New\_SMess(Rmsgs, q), Tick(datecourante(q) >$
then free {
    pred $Reach : States$
    $\forall\, q, q' : State; m : Messages$
    • $Reach(Init)$
    • $(Reach(q) \wedge q \xrightarrow{m} q') \;\Rightarrow\; Reach(q')$ **}**

La structure de la spécification à la fin de cette phase de conception est représentée dans la figure 5.2. Comme le montre cette figure, la possibilité de redéfinir la sémantique d'une spécification nommée permet de conserver une structure simple de la spécification en suivant un schéma de spécification global clair. Néanmoins, il est difficile alors, sans outil spécifique, de reconstruire la version courante de chaque spécification nommée. Nous reproduisons dans l'annexe C la spécification complète de cette étude de cas en éliminant les redéfinitions afin de montrer les versions finales de chaque spécification nommée telle qu'elle pourrait être proposée par un outil.

[2] Une extension du cahier des charges indique que cette configuration pourrait être modifiée en cours de fonctionnement. Cette extension impliquerait de stocker la configuration courante dans la structure des états et de disposer de messages permettant de modifier cette configuration. Nous ne traitons pas cette extension.

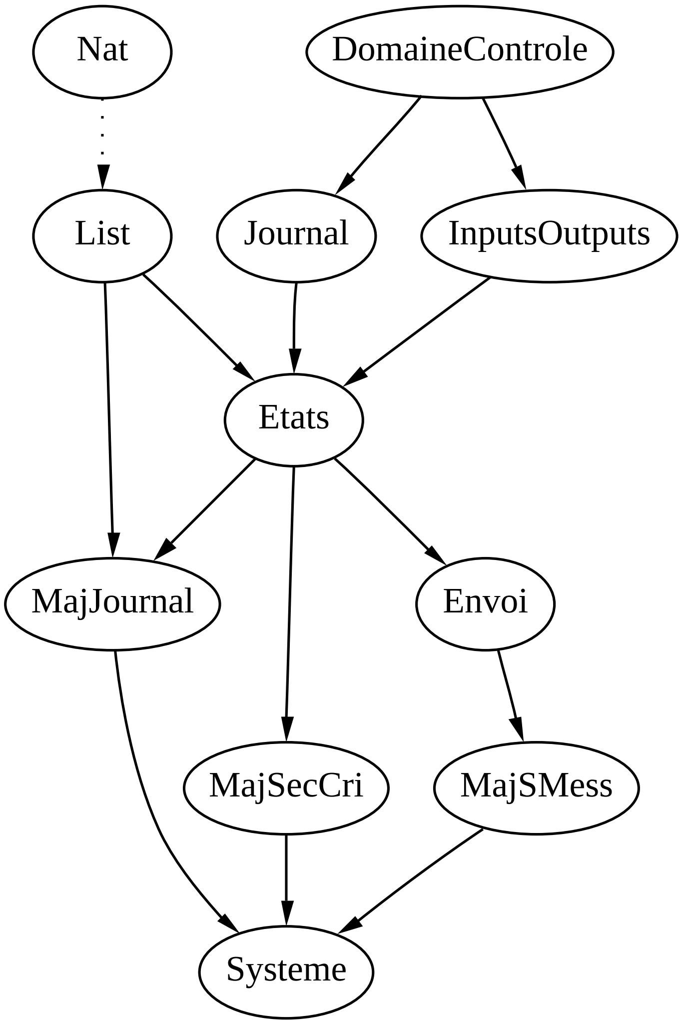

FIG. 5.2.: Phase de conception - Structure de la spécification
Spécification algébrique et prototypage du "controle d'accès" en LPG. Didier Bert Dans "Approches formelles pour l'aide au développement de logiciels" TSI, vol 20, Numéro 7, pages 849-873, 2001
@bert, author = Didier Bert, title = Spécification algébrique et prototypage du Controle d'accès en LPG, booktitle = Approches formelles pour l'aide au développement de logiciels, OPTcrossref = , OPTkey = , pages = 142-157, year = 2000, OPTeditor = , OPTvolume = , OPTnumber = , OPTseries = , OPTaddress = , OPTmonth = , OPTorganization = , publisher = LSR-IMAG, OPTnote = , OPTannote =

5.4.5. Phase de découpage

Comme dans l'exemple de l'analyse lexicale, le but de cette partie est de construire une spécification architecturale bien formée prenant en compte les différentes contraintes méthodologiques évoquées dans la section 2.3 : choisir un style de découpage, modifier le moins possible la structure de la spécification.

Nous avons choisi, pendant l'identification d'une structure préliminaire, d'associer un nom de spécification à chaque sorte. Au moment du découpage, ce choix n'apparait pas pratique, puisqu'alors certaines unités ne contiennent que peu d'informations. Au contraire, nous allons regrouper dans une première unité *BasicAlg* tous les types de base que l'on a introduit. La spécification associée à cette unité est JOURNAL qui contient toutes celles qui nous intéressent. Le type des états est placé dans une unité distincte *StatesAlg* qui étend cette unité. Nous n'associons pas d'unité spécifique aux listes ou aux ensembles, car ces types de données sont en général déjà implémentés dans un langage de programmation (les ensembles pouvant se voir comme des listes sans répétitions). La fonction d'envoi est associée à une unité spécifique *EnvoiAlg*. Les mises à jour des différentes variables d'états sont toutes regroupées dans une même unité *MajAlg*. La spécification associée est l'union des spécifications pour chaque mise à jour. Notons que cette union ne représente pas une union au niveau des modèles, mais seulement au niveau des spécifications. La dernière unité consiste en la spécification du système, qui correspond également à la reconstruction des différentes unités du programme. On obtient la spécification architecturale suivante :

arch spec CONTROLEACCES =
    units
        BasicAlg : JOURNAL ;
        StatesAlg : ETATS **given** *BasicAlg* ;
        EnvoiAlg : ENVOI **given** *StatesAlg* ;
        MajAlg : MAJSMESS **and** MAJSECCRI **and** MAJJOURNAL **given** *EnvoiAlg* ;
        SystemeAlg : SYSTEME **given** *MajAlg* ;
    result *SystemeAlg*
end

Contrairement à ce qui s'était passé au moment de la construction d'une spécification architecturale pour le cas de l'analyse lexicale, il suffit ici de choisir comment répartir les spécifications nommées dans les unités, en faisant très peu de modifications sur l'organisation de la spécification. Ici, on aurait pu vouloir associer les listes à une unité spécifique, auquel cas il aurait fallu déplacer la déclaration du type des sections critiques pour pouvoir instancier cette unité. Chaque unité peut maintenant être développée indépendamment les unes des autres.

Nous pouvons comparer ce résultat avec la spécification de [Ber01]. Le langage de spécification (LPG) utilisé par Didier Bert permet de découper la spécification globale en fichiers indépendants qui sont ensuite assemblés. Ces fichiers peuvent être assimilés aux unités dans notre approche. L'architecture proposée par le découpage présenté dans cet article est proche de celui que nous venons de présenter, ce qui permet de consolider notre approche en montrant que l'évolution de la structure préliminaire vers la structure de la spécification architecturale est cohérente avec celle que l'on aurait pu déduire par d'autres approches. Notre approche ne complique donc pas le processus de développement des spécifications.

6. Conclusion

Dans le développement d'un système informatique complexe, l'utilisation d'une technique formelle s'avère particulièrement utile pour assurer une certaine confiance dans les parties logicielles critiques. Utiliser une technique formelle signifie d'abord développer un modèle pour chacune de ces parties critiques, puis grâce à la sémantique formelle de ces modèles vérifier un certain nombre de propriétés sur ces modèles.

Dans cette thèse, nous nous sommes principalement intéressés à la première partie : l'écriture d'un modèle. Dans la famille des techniques algébriques, cette première partie consiste à écrire une spécification de chaque partie critique à l'aide de symboles représentant des informations du cahier des charges, dont on contraint le comportement par des axiomes de la logique du premier ordre. Nous avons essayé de nous placer du point de vue de l'utilisateur du langage CASL pour écrire une telle spécification. De son point de vue, ce langage lui propose des outils, des notations, une sémantique, lui permettant d'exprimer les informations importantes du cahier des charges. En observant le processus d'écriture suivi par un utilisateur lors de tout développement un peu complexe, nous avons mis en évidence des caractéristiques de ce processus. Ces caractéristiques sont l'incrémentalité, c'est-à-dire le fait de procéder étapes par étapes, et l'itérativité, le fait de revenir plusieurs fois sur la spécification d'un même point.

Ces deux caractéristiques sont bien connues et ont donné lieu à de nombreux travaux afin de les prendre en compte et de leur donner un sens précis. Pour permettre une approche incrémentale, des opérateurs de spécifications ont été construits permettant de combiner plusieurs morceaux de spécification en une seule spécification. La présence des opérateurs dans un langage comme CASL permet de traduire une partie du cahier des charges à la fois, puis de relier les différentes spécifications obtenues pour former une spécification globale. La caractéristique itérative apparait lorsque la traduction de ces parties ne comporte pas tous les détails, mais seulement les informations pertinentes pour la vérification des propriétés voulues. Les détails sont alors introduits au fur et à mesure, tout en conservant la validité de ces propriétés. Ceci est le principe du développement par raffinements successifs. Dans le langage CASL, la formalisation de cette caractéristique se situe au niveau de la sémantique des spécifications, comme une inclusion des classes de modèles. La seule notation proposée dans ce langage pour indiquer que deux spécifications sont dans une relation de raffinement est d'indiquer un morphisme entre leur signature. Or, dans la plupart des cas, ce que l'utilisateur veut faire lors d'une itération est de compléter la spécification par de nouvelles informations. Cette complétion peut être un raffinement dans le sens classique du terme ou seulement un ajout d'informations au même niveau d'abstraction. Il peut également s'agir d'une modification, pour changer un profil ou un axiome par exemple.

L'absence d'une notation pour cette vision de l'aspect itératif porte particulièrement à préjudice lors de la phase de découpage architectural. Dans le cadre d'un processus de développement avec Casl, le découpage architectural consiste à identifier des parties de la spécification que l'on associe à des unités qui pourront ensuite être développées indépendamment les unes des autres et recomposées selon une structure précise. Si aucune information dans la structure de la spécification devant être découpée n'indique cet aspect itératif, la phase de découpage peut être problématique. Les informations relatives à la spécification d'une unité doivent être regroupées, modifiant la structure de la spécification jusqu'à ce point. Cette modification apporte un travail supplémentaire et, dans la plupart des cas, inutile. De plus la modification de l'organisation peut nuire à la revue de la spécification lors de sa validation. Enfin, dans certains cas, aucun découpage n'est possible à cause des choix de modélisation. Ce cas se présente notamment dans le cas des systèmes réactifs, lorsque le schéma de spécification de l'historique des échanges de messages est utilisé, comme c'est le cas pour la spécification de [BCPR95].

Nous avons alors proposé une extension du langage Casl permettant de donner une notation à l'aspect itératif. Cette extension consiste à permettre la redéfinition d'une spécification nommée et de propager la nouvelle définition dans la sémantique des autres spécifications nommées faisant référence à la spécification modifiée. La redéfinition est définie comme une constructeur de bibliothèques, par deux règles sémantiques préservant la structure du graphe de dépendance des noms de spécifications. La propagation de la nouvelle définition est effectuée dans les environnements contenant la signature et la classe de modèles de chaque spécification nommée de la bibliothèque courante. Nous avons étudié la possibilité de réutiliser les informations de ces environnements pour évaluer les nouvelles valeurs. Plutôt que de calculer complètement la mise à jour de la signature des spécifications faisant référence à la spécification redéfinie, une formule peut directement être appliquée. Nous avons montré que cette amélioration préserve la correction des bibliothèques.

Pour faciliter l'utilisation de cette nouvelle construction, un guide structurel peut être utilisé dès la phase d'analyse. Ce guide structurel permet de recenser des traits essentiels du programme et de les organiser en une structure préliminaire. Nous n'avons pas approfondi les techniques permettant d'identifier ces traits. Nous nous sommes basés sur les *Problem Frames*, un outil de structuration logique du cahier des charges. Ce guide préliminaire est ensuite utilisé pour construire la spécification des besoins de manière compatible avec un découpage architectural.

La technique résultante a été utilisée dans le cas particulier des systèmes réactifs. Certains aspects de ces systèmes peuvent être étudiés avec un langage de spécification algébrique. Nous avons présenté trois approches possibles, parmi d'autres, pour spécifier un tel système. Les deux premières sont classiques et proposent une version algébrique de modèles mathématiques couramment utilisés dans la vérification de ces systèmes. La troisième approche combine les avantages des deux précédentes quant à leur facilité pour suivre un processus incrémental et modulaire. Cette approche est basée sur la technique développée dans cette thèse : construction d'un guide préliminaire, puis écriture de la spécification en indiquant explicitement les itérations par la construction de redéfinition. Nous avons appliqué cette approche sur l'étude de cas du Contrôle d'Accès. Nous avons montré que cette approche était tout à fait compatible avec le développement incrémental et modulaire d'un système complexe et pouvait donc être appliquée à des études de cas plus difficiles.

Annexes

A. Syntaxe abstraite de CASL

La syntaxe abstraite est une notion centrale dans la définition d'un langage de spécification formel. Elle se situe entre les représentations concretes dans les documents, que l'on peut voir comme des signes sur le papier ou des images sur un écran, et les notions abstraites que sont la sémantique et les fonctions sémantiques utilisées pour déterminer la signification des signes en tant qu'objets sémantiques.

La syntaxe abstraite a comme objectif :

- d'identifier et donc nommer les entités syntaxiques abstraites ;
- de simplifier et d'unifier les concepts sous-jacents, en rassemblant des choses semblables et évitant les duplications superflues.

Il y a plusieurs manières de construire une syntaxe abstraite. Le choix d'une forme plutôt qu'une autre est une affaire de goût, prenant en compte les possibles conflits inhérent à trop de simplicité, et l'économie des définitions sémantiques.

La syntaxe abstraite se présente sous la forme d'un ensemble de règles de productions dans lequel chaque sorte d'entité est défini en fonction de ses constituants :

```
SOME-SORT       ::=  SUBSORT-1  |  ...  |  SUBSORT-n
```

ou bien en fonction de ses constructeurs muni de leurs arguments :

```
SOME-CONSTRUCT  ::=  some-construct COMPONENT-1 ... COMPONENT-n
```

Les règles de production forment une grammaire hors contexte. Les symboles non terminaux de la grammaire correspondent aux feuilles d'un arbre, et les symboles terminaux aux noeuds de cet arbre. La notation `COMPONENT*` indique la répétition de `COMPONENT` un nombre indéfini de fois ; `COMPONENT+` indique une répétition d'au moins une fois. Les conditions de bonne formation des spécifications ne sont pas déterminées par la grammaire, mais par les règles sémantiques données par ailleurs.

Les symboles non terminaux suivants correspondent à l'analyse lexicale d'une spécification et sont laissés sous-spécifiés dans la syntaxe abstraite : `WORDS`, `SIGNS`, `DOT-WORDS`, `PLACE`, `URL`, `PATH`, and `NUMBER`.

A.1. Spécifications de base

```
BASIC-SPEC       ::= basic-spec BASIC-ITEMS*
```

```
BASIC-ITEMS       ::= SIG-ITEMS | FREE-DATATYPE | SORT-GEN
                    | VAR-ITEMS | LOCAL-VAR-AXIOMS | AXIOM-ITEMS

SIG-ITEMS         ::= SORT-ITEMS | OP-ITEMS | PRED-ITEMS
                    | DATATYPE-ITEMS

SORT-ITEMS        ::= sort-items SORT-ITEM+
SORT-ITEM         ::= SORT-DECL

SORT-DECL         ::= sort-decl SORT+

OP-ITEMS          ::= op-items OP-ITEM+
OP-ITEM           ::= OP-DECL | OP-DEFN

OP-DECL           ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
OP-TYPE           ::= TOTAL-OP-TYPE | PARTIAL-OP-TYPE
TOTAL-OP-TYPE     ::= total-op-type SORTS SORT
PARTIAL-OP-TYPE   ::= partial-op-type SORTS SORT
SORTS             ::= sorts SORT*
OP-ATTR           ::= BINARY-OP-ATTR | UNIT-OP-ATTR
BINARY-OP-ATTR    ::= associative | commutative | idempotent
UNIT-OP-ATTR      ::= unit-op-attr TERM

OP-DEFN           ::= op-defn OP-NAME OP-HEAD TERM
OP-HEAD           ::= TOTAL-OP-HEAD | PARTIAL-OP-HEAD
TOTAL-OP-HEAD     ::= total-op-head ARG-DECL* SORT
PARTIAL-OP-HEAD   ::= partial-op-head ARG-DECL* SORT
ARG-DECL          ::= arg-decl VAR+ SORT

PRED-ITEMS        ::= pred-items PRED-ITEM+
PRED-ITEM         ::= PRED-DECL | PRED-DEFN

PRED-DECL         ::= pred-decl PRED-NAME+ PRED-TYPE
PRED-TYPE         ::= pred-type SORTS
PRED-DEFN         ::= pred-defn PRED-NAME PRED-HEAD FORMULA
PRED-HEAD         ::= pred-head ARG-DECL*

DATATYPE-ITEMS    ::= datatype-items DATATYPE-DECL+
DATATYPE-DECL     ::= datatype-decl SORT ALTERNATIVE+
ALTERNATIVE       ::= TOTAL-CONSTRUCT | PARTIAL-CONSTRUCT
TOTAL-CONSTRUCT   ::= total-construct   OP-NAME COMPONENTS*
PARTIAL-CONSTRUCT::= partial-construct OP-NAME COMPONENTS*
```

```
COMPONENTS       ::= TOTAL-SELECT | PARTIAL-SELECT | SORT
TOTAL-SELECT     ::= total-select   OP-NAME+ SORT
PARTIAL-SELECT   ::= partial-select OP-NAME+ SORT

FREE-DATATYPE    ::= free-datatype DATATYPE-ITEMS

SORT-GEN         ::= sort-gen SIG-ITEMS+

VAR-ITEMS        ::= var-items VAR-DECL+
VAR-DECL         ::= var-decl VAR+ SORT

LOCAL-VAR-AXIOMS ::= local-var-axioms VAR-DECL+ AXIOM+

AXIOM-ITEMS      ::= axiom-items AXIOM+

AXIOM            ::= FORMULA
FORMULA          ::= QUANTIFICATION | CONJUNCTION | DISJUNCTION
                   | IMPLICATION | EQUIVALENCE | NEGATION | ATOM
QUANTIFICATION   ::= quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER       ::= forall | exists | exists-uniquely
CONJUNCTION      ::= conjunction FORMULA+
DISJUNCTION      ::= disjunction FORMULA+
IMPLICATION      ::= implication FORMULA FORMULA
EQUIVALENCE      ::= equivalence FORMULA FORMULA
NEGATION         ::= negation FORMULA

ATOM             ::= TRUTH | PREDICATION | DEFINEDNESS
                   | EXISTL-EQUATION | STRONG-EQUATION
TRUTH            ::= true | false
PREDICATION      ::= predication PRED-SYMB TERMS
PRED-SYMB        ::= PRED-NAME | QUAL-PRED-NAME
QUAL-PRED-NAME   ::= qual-pred-name PRED-NAME PRED-TYPE
DEFINEDNESS      ::= definedness TERM
EXISTL-EQUATION  ::= existl-equation TERM TERM
STRONG-EQUATION  ::= strong-equation TERM TERM

TERMS            ::= terms TERM*
TERM             ::= SIMPLE-ID | QUAL-VAR | APPLICATION
                   | SORTED-TERM | CONDITIONAL
QUAL-VAR         ::= qual-var VAR SORT
APPLICATION      ::= application OP-SYMB TERMS
OP-SYMB          ::= OP-NAME | QUAL-OP-NAME
QUAL-OP-NAME     ::= qual-op-name OP-NAME OP-TYPE
```

```
SORTED-TERM      ::= sorted-term TERM SORT
CONDITIONAL      ::= conditional TERM FORMULA TERM

SORT             ::= TOKEN-ID
OP-NAME          ::= ID
PRED-NAME        ::= ID
VAR              ::= SIMPLE-ID

ID               ::= TOKEN-ID | MIXFIX-ID
TOKEN-ID         ::= TOKEN
MIXFIX-ID        ::= TOKEN-PLACES
TOKEN-PLACES     ::= token-places TOKEN-OR-PLACE+
TOKEN-OR-PLACE   ::= TOKEN | PLACE
TOKEN            ::= WORDS | SIGNS | DOT-WORDS
SIMPLE-ID        ::= WORDS
```

A.2. Sous-sortage

```
SORT-ITEM        ::= ... | SUBSORT-DECL | ISO-DECL | SUBSORT-DEFN

SUBSORT-DECL     ::= subsort-decl SORT+ SORT
ISO-DECL         ::= iso-decl SORT+
SUBSORT-DEFN     ::= subsort-defn SORT VAR SORT FORMULA

ALTERNATIVE      ::= ... | SUBSORTS
SUBSORTS         ::= subsorts SORT+

ATOM             ::= ... | MEMBERSHIP
MEMBERSHIP       ::= membership TERM SORT

TERM             ::= ... | CAST
CAST             ::= cast TERM SORT
```

A.3. Spécifications structurées

```
SPEC             ::= BASIC-SPEC | TRANSLATION | REDUCTION
                   | UNION | EXTENSION | FREE-SPEC | LOCAL-SPEC
                   | CLOSED-SPEC | SPEC-INST

TRANSLATION      ::= translation SPEC RENAMING
RENAMING         ::= renaming  SYMB-MAP-ITEMS+

REDUCTION        ::= reduction SPEC RESTRICTION
RESTRICTION      ::= HIDE | REVEAL
HIDE             ::= hide   SYMB-ITEMS+
REVEAL           ::= reveal SYMB-MAP-ITEMS+

UNION            ::= union SPEC+
EXTENSION        ::= extension SPEC+
FREE-SPEC        ::= free-spec SPEC
LOCAL-SPEC       ::= local-spec SPEC SPEC
CLOSED-SPEC      ::= closed-spec SPEC

SPEC-DEFN        ::= spec-defn SPEC-NAME GENERICITY SPEC
GENERICITY       ::= genericity PARAMS IMPORTS
PARAMS           ::= params SPEC*
IMPORTS          ::= imports SPEC*

SPEC-INST        ::= spec-inst SPEC-NAME FIT-ARG*

FIT-ARG          ::= FIT-SPEC | FIT-VIEW
FIT-SPEC         ::= fit-spec SPEC SYMB-MAP-ITEMS*
FIT-VIEW         ::= fit-view VIEW-NAME FIT-ARG*

VIEW-DEFN        ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE
                               SYMB-MAP-ITEMS*
VIEW-TYPE        ::= view-type SPEC SPEC

SYMB-ITEMS       ::= symb-items     SYMB-KIND SYMB+
SYMB-MAP-ITEMS   ::= symb-map-items SYMB-KIND SYMB-OR-MAP+
SYMB-KIND        ::= implicit | sorts | ops | preds

SYMB             ::= ID | QUAL-ID
QUAL-ID          ::= qual-id ID TYPE
TYPE             ::= OP-TYPE | PRED-TYPE | SORT
SYMB-MAP         ::= symb-map SYMB SYMB
```

```
SYMB-OR-MAP      ::= SYMB | SYMB-MAP

SPEC-NAME        ::= SIMPLE-ID
VIEW-NAME        ::= SIMPLE-ID

TOKEN-ID         ::= ... | COMP-TOKEN-ID
MIXFIX-ID        ::= ... | COMP-MIXFIX-ID
COMP-TOKEN-ID    ::= comp-token-id TOKEN ID+
COMP-MIXFIX-ID   ::= comp-mixfix-id TOKEN-PLACES ID+ %
```

A.4. Spécifications architecturales

```
ARCH-SPEC-DEFN   ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC        ::= BASIC-ARCH-SPEC | ARCH-SPEC-NAME
BASIC-ARCH-SPEC  ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT

UNIT-DECL-DEFN   ::= UNIT-DECL | UNIT-DEFN
UNIT-DECL        ::= unit-decl UNIT-NAME UNIT-SPEC UNIT-IMPORTS
UNIT-IMPORTS     ::= unit-imports UNIT-TERM*
UNIT-DEFN        ::= unit-defn UNIT-NAME UNIT-EXPRESSION

UNIT-SPEC-DEFN   ::= unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC        ::= UNIT-TYPE | SPEC-NAME | ARCH-SPEC
                   | CLOSED-UNIT-SPEC
CLOSED-UNIT-SPEC ::= closed UNIT-SPEC
UNIT-TYPE        ::= unit-type SPEC* SPEC

RESULT-UNIT      ::= result-unit UNIT-EXPRESSION
UNIT-EXPRESSION  ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING     ::= unit-binding UNIT-NAME UNIT-SPEC
UNIT-TERM        ::= UNIT-REDUCTION | UNIT-TRANSLATION | AMALGAMATION
                   | LOCAL-UNIT | UNIT-APPL
UNIT-TRANSLATION ::= unit-translation UNIT-TERM RENAMING
UNIT-REDUCTION   ::= unit-reduction UNIT-TERM RESTRICTION
AMALGAMATION     ::= amalgamation UNIT-TERM+
LOCAL-UNIT       ::= local-unit UNIT-DEFN+ UNIT-TERM
UNIT-APPL        ::= unit-appl UNIT-NAME FIT-ARG-UNIT*
FIT-ARG-UNIT     ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*

ARCH-SPEC-NAME   ::= SIMPLE-ID
UNIT-NAME        ::= SIMPLE-ID %
```

A.5. Libraries de spécification

```
LIB-DEFN         ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM         ::= SPEC-DEFN | VIEW-DEFN
                   | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
                   | DOWNLOAD-ITEMS

DOWNLOAD-ITEMS   ::= download-items LIB-NAME ITEM-NAME-OR-MAP+
ITEM-NAME-OR-MAP ::= ITEM-NAME | ITEM-NAME-MAP
ITEM-NAME-MAP    ::= item-name-map ITEM-NAME ITEM-NAME
ITEM-NAME        ::= SIMPLE-ID

LIB-NAME         ::= LIB-ID | LIB-VERSION
LIB-ID           ::= DIRECT-LINK | INDIRECT-LINK
DIRECT-LINK      ::= direct-link URL
INDIRECT-LINK    ::= indirect-link PATH
LIB-VERSION      ::= lib-version LIB-ID VERSION
VERSION          ::= version NUMBER+
```

B. Cahier des charges du Contrôle d'accès

On souhaite gérer l'accès de personnes à un ensemble de bâtiments. Pour fixer les idées, les bâtiments seront ceux du pôle informatique du campus de Grenoble, et les personnes seront les étudiants, enseignants, chercheurs et personnels administratifs qui fréquentent ces locaux.

B.1. Politique de contrôle d'accès

L'accès à un bâtiment est accordé sur base de l'affiliation d'une personne à un groupe.

1. Un bâtiment est ouvert à certains groupes (par exemple, le bâtiment de l'UFR-IMA est ouvert aux étudiants de l'Université Joseph Fourier (UJF), aux enseignants et aux personnels administratifs de l'UJF).
2. Une personne appartient à un et un seul groupe (par exemple : Paul est un enseignant de l'ENSIMAG, Didier est un chercheur du CNRS, Yves est un enseignant de l'UJF, Rémy est un étudiant de l'UJF).

B.2. Enregistrement des accès

Tout accès est enregistré (personne, bâtiment, heure) tant en entrée qu'en sortie d'un bâtiment. Cet enregistrement est motivé par :

1. La possibilité de connaitre a posteriori qui était dans un bâtiment à un instant précis (par exemple : retrouver les étudiants présents lors d'un incident informatique sur le réseau étudiant).
2. La possibilité de connaitre la liste des personnes actuellement présentes dans un bâtiment (par exemple, en cas d'évacuation suite à un incendie).
3. La d'etection et la prévention de tentatives de fraudes :
   a) on ne peut sortir d'un bâtiment que si on y est préalablement entré ;
   b) on ne peut pas être présent simultanément dans plusieurs bâtiments.

B.3. Mise en oeuvre de la politique d'accès

B.3.1. Les équipements

Les équipements suivants sont installés :

1. Chaque personne dispose d'une carte magnétique dont l'identificateur est unique.
   a) La liste des cartes mises en circulation, ainsi que celle des cartes perdues ou volées sont enregistrées.
   b) L'identificateur d'une carte est constitué de 8 chiffres : les 6 premiers donnent le numéro de carte ; les deux derniers sont un code de détection d'erreur, à savoir le reste de la division du numéro de carte par 97. Ce code de détection d'erreur est destiné à empêcher l'utilisation de cartes défectueuses ou à détecter des erreurs de lecture.
2. Des lecteurs de cartes sont installés à chaque entrée/sortie de bâtiment.
3. L'ouverture des portes peut être bloquée/débloquée à distance.
4. Le passage ou l'ouverture de la porte est détecté
5. Un voyant vert indique l'état bloqué/débloqué de la porte.
6. Un voyant rouge indique le refus d'accès ou la fermeture (temporaire) de l'accès.

B.3.2. Le protocole d'accès

Le protocole d'accès est le suivant :

1. La personne qui souhaite entrer ou sortir d'un bâtiment insère sa carte dans le lecteur si le voyant n'est pas rouge.
2. Si le voyant est rouge, elle attend l'extinction du voyant, se présente plus tard à cet accès ou tente d'emprunter un autre accès.
3. Après lecture de la carte, si l'accès est autorisé, le voyant vert s'allume l'ouverture de la porte est débloquée et la personne dispose de 30" pour entrer avant reblocage de la porte.
4. Si l'utilisateur entre, son passage est détecté et le journal est mis à jour ; la porte est bloquée.
5. Si l'accès est refusé, le voyant rouge s'allume pendant deux secondes.

B.3.3. Usagers disciplinés

Les usagers sont instamment priés de respecter ce protocole. En particulier :

1. Il est interdit de permettre l'entrée ou la sortie d'une personne qui n'a pas inséré sa carte dans le lecteur ;

2. Une personne qui a inséré sa carte et re,cu le feu vert est tenue de franchir la porte ; cette règle peut paraitre en contradiction avec le blocage de la porte après 30" ; en fait, le blocage intervient si la personne réagit trop lentement au feu vert ;
3. Une fois, la porte franchie, il est obligatoire de la refermer avant le passage du suivant (ceci pour garantir qu'une seule personne a franchi la porte) ; cette obligation peut disparaitre si la porte qui barre l'accès est remplacée par un tourniquet qui garantit le passage d'une personne à la fois ;
4. Si deux personnes veulent emprunter le même accès, la deuxième personne attend que la première ait terminé la procédure avant d'introduire sa carte ;
5. Une fois la porte franchie, il est interdit de revenir sur ses pas.

B.4. Le système informatique

B.4.1. Mode opératoire

Le système informatique à spécifier et à développer contrôle les lecteurs de cartes, les portes et les voyants des bâtiments pour mettre en oeuvre la politique de contrôle d'accès. Son mode opératoire comprend trois phases :

1. La mise en route du système fait l'hypothèse (simplificatrice) que tous les bâtiments sont vides. Avant cette mise en route, tous les voyants sont éteints et toutes les portes sont bloquées (hypothèse simplificatrice).
2. En fonctionnement normal, le système contrôle les accès aux bâtiments. La détection de comportements frauduleux ne donne pas lieu à uneautre réaction que le maintien de la porte en position bloquée. La liste des utilisateurs, des groupes et l'affiliation d'un utilisateur à un groupe ne peuvent être modifiés.
3. L'arrêt de fonctionnement, essentiellement pour des raisons de sécurité (par exemple l'alarme incendie), provoque le déblocage de toutes les portes et la mise à vert des voyants pour tous les bâtiments.

B.4.2. Communication

La communication avec les périphériques du système informatique (lecteurs, portes, voyants) se fait par échange de messages sur un bus. Le format des messages est le suivant :

1. Les capteurs :
   a) `lecture(id_lecteur,id_carte)` signale la lecture d'une carte par un lecteur ;
   b) `ouverture(id_porte)` signale l'ouverture par un utilisateur (non identifié dans ce message) d'une porte ;
   c) `fermeture(id_porte)` signale la fermeture par un utilisateur (non identifié dans ce message) d'une porte ;

   d) `incendie(id_capteur)` signale un incendie détecté par un capteur.
2. Les portes et les voyants (les voyants sont associés aux lecteurs et non aux portes) :
   a) `bloquer(id_porte)` bloque la porte ;
   b) `debloquer(id_porte)` débloque la porte ;
   c) `allumer_voyant_rouge(id_lecteur)` allume le voyant rouge associé à un lecteur ;
   d) `allumer_voyant_vert(id_lecteur)` allume le voyant vert associé à un lecteur ;
   e) `eteindre_voyant_rouge(id_lecteur)` eteind le voyant rouge associé à un lecteur ;
   f) `eteindre_voyant_vert(id_lecteur)` eteind le voyant vert associé à un lecteur ;
3. Tout message donne lieu à un accusé de réception `ok(id)` :
   a) Le paramètre `id` identifie le capteur concerné (`id_lecteur, id_porte, id_capteur`), si l'accusé de réception est envoyé par le système de contrôle.
   b) Il identifie l'émetteur de l'accusé de réception si il est émis vers le système de contrôle par un actuateur (`id_porte, id_lecteur`).

Pour la construction d'un simulateur sous UNIX, on fera l'hypothèse que les messages destinés aux portes sont envoyés au stdin du simulateur et que les messages des capteurs sont re,cus sur le stdout. Il est possible que le simulateur demande lénvoi d'un message initial de synchronisation. Une version plus évoluée utilisera un simulateur par bâtiment.

B.5. Evolutions du cahier des charges

On peut envisager plusieurs améliorations du système, en particulier :

1. l'affiliation d'un utilisateur à plusieurs groupes ;
2. la gestion de plusieurs types de portes (certaines étant automatiques, d'autres correspondant aux portes des ascenseurs, ...)
3. l'accord d'accès sur base de l'appartenance à un groupe, mais aussi en fonction de l'heure (par exemple, les étudiants ne peuvent entrer qu'entre 7h et 22h).
4. une procédure d'initialisation du système et d'arrêt qui se déroule bâtiment par bâtiment ;
5. la possibilité de modifier les droits d'accès, la liste des utilisateurs, la liste des groupes et l'affiliation à un groupe pendant l'utilisation du système, en garantissant à un utilisateur qui est entré dans un bâtiment de pouvoir en sortir ;
6. la liaison du système avec un poste de gardiennage pour traiter les tentatives d'intrusion frauduleuse ;
7. la surveillance et le traitement des défaillances des équipements et des communications.

B.6. Questions/réponses

Suite aux premières tentatives de spécification, certaines questions ont été soulevées. Pour conserver l'énoncé original, elles sont traitées ci-après comme questions/réponses.

1. Dans 3.1, pour le point 6, il nous semble intéressant de distinguer ce cas de fermeture (temporaire) par un nom différent, par exemple, comdamnation, car ce nést pas tout à fait la notion de fermeture utilisée ensuite. (Gérard Padiou, 7 avril 1997)

   C'est exact.

2. Dans 3.2, il n'est pas clair qu'il existe deux voyants. Par exemple, dans le point 1, il faudrait remplacer "si le voyant n'est pas rouge" par "si le voyant rouge n'est pas allumé". C'est pareil dans le point 2, qui devrait plutôt commencer par "Si le voyant rouge est allumé. . . " . . . (Gérard Padiou, 7 avril 1997)

   Il existe deux voyants : un voyant rouge et un voyant vert. Chaque voyant peut être dans l'état allumé ou éteint.

3. Dans le point 3 de 3.2, ne pas utiliser le mot "entrer". En effet, l'usager entre OU sort. Utiliser le mot "passer" par exemple. D'ou, dans le point 4 suivant, le meme problème avec : "Si l'utilisateur entre...", il faudrait mettre "passe" . . . (Gérard Padiou, 7 avril 1997)

   C'est exact. Le système ne peut détecter qu'un passage.

4. De plus, on ne sait pas ce que deviennent les voyants. Il faudrait sans doute ajouter dans ce point 4 de 3.2, quelque chose du genre : "et le voyant vert s'éteint". (Gérard Padiou, 7 avril 1997)

   Cet aspect est spécifié à la section 3.1, au point 5. "Un voyant vert indique l'état bloqué/débloqué de la porte." Ceci signifie que le voyant vert est allumé quand la porte est débloquée et éteint quand la porte est bloquée.

5. Un problème plus important est la configuration des sas d'entrée/sortie : Pour schématiser, un sas est-il à sens unique et constitué d'un lecteur, de deux voyants et d'une porte OU à double sens cést-à-dire, un lecteur déntrée, un lecteur de sortie, 2 fois 2 voyants et UNE porte. Ce cas de figure est beaucoup plus compliqué à traiter. Il faut par exemple, assurer l'exclusion mutuelle entre une entrée et une sortie. Que choisit-on ? (Gérard Padiou, 7 avril 1997)

   On choisit la solution simple : une porte ne peut servir qu'en entrée ou (exclusif) en sortie.

6. La notion de groupe est devenue inutile compte-tenu de l'hypothèse : une personne appartient à un seul groupe. (Gérard Padiou, 7 avril 1997)

   Il est exact que l'on peut "compiler" ce niveau d'indirection et établir pour chaque bâtiment la liste des personnes autorisées. Cependant, cette notion de groupe fait partie des concepts de définition du problème.

7. Doit-on considérer les communications comme fiables ? Il y a des acquittements, ok(id) par exemple ; peuvent-ils se perdre ? (Gérard Padiou, 7 avril 1997)

   La défaillance des communications fait partie de l'évolution 7 de la section 5. Dans le problème de base, les communications et les équipements sont fiables. Les acquittements font partie d'un protocole (simple) de communication avec les équipements.

8. Dans le point 3 de la section 2, la phase "on ne peut sortir d'un bâtiment que si on y est préalablement rentré" mérite quelques clarifications. (réunion du 4 décembre)

   Il ne s'agit pas d'une "loi de la nature" ou d'un truisme. Cette phrase veut dire que quand quelqu'un veut sortir du bâtiment, on vérifie qu'il y est entré normalement, ce qui se traduit entre autres par léxistence d'un enregistrement dans le journal. Si la personne nést pas entrée suivant la procédure, on ne lui donne pas le droit de sortir : la porte reste bloquée (exactement de la même fa,con qu'une personne qui est déjà enregistrée dans un autre bâtiment n'a pas le droit d'entrer dans un deuxième bâtiment).

   D'aucuns s'inquiètent de la procédure à suivre pour faire sortir un fraudeur d'un bâtiment. Tel que le cahier des charges est décrit, il ne peut sortir que frauduleusement par la fenêtre, ou en accompagnant quelqu'un qui sort (par exemple le gardien), ou en déclenchant l'alarme incendie. . .

9. Dans le point 3 de la section 2, on précise qu'un individu ne peut être présent simultanément dans plusieurs bâtiments. Que se passe-t-il si il essaie d'entrer dans un bâtiment o'u il est déjà enregistré ? (réunion du 4 décembre)

   Le cahier des charges n'a pas prévu ce cas de figure. Il me semble que l'attitude la plus cohérente avec le cahier des charges est de lui interdire l'accès.

10. Le cahier des charges ne précise pas la conduite à tenir en cas de détection de fraude. (réunion du 4 décembre)

    En fait, il interdit seulement de laisser passer l'individu. Il ne précise pas si la tentative de fraude doit être signalée à un service de gardiennage, ou si la fraude doit être notée dans le journal.

11. Au point 2 de 4.1, on précise que l'affiliation d'un utilisateur à un groupe ne peut être modifiée. Par contre, on ne dit pas que le droit d'accès d'un groupe à un bâtiment ne peut être modifié. (réunion du 4 décembre)

    C'est un oubli. On ne peut modifier les droits d'accès d'un groupe à un bâtiment. C'est un point qu'il faut traiter dans le cadre des évolutions (point 5 de 5).

12. Le cahier des charges mentionne la condamnation des portes, mais n'est pas plus explicite à ce sujet. (réunion du 4 décembre)

    C'est exact. La condamnation des portes est une fonctionnalité mineure du système. Dans le simulateur, un message de condamnation de porte a été prévu.

13. Que doit faire le système si un utilisateur demande l'accès et qu'arrive un message de condamnation de la porte avant qu'il ne l'ait ouverte ? (réunion du 4 décembre)

    La condamnation des portes est une fonctionnalité mineure. Si l'utilisateur a respecté le protocole, c'est-à-dire qu'il a inséré sa carte alors que le voyant était éteint et qu'il a bien le droit d'accéder au bâtiment, la porte doit lui être ouverte. Elle sera condamnée après son passage.

14. Il n'est pas dit explicitement qu'un utilisateur dispose d'une et une seule carte. (rapport IRIT/97-33-R)

    C'est exact ! Cet aspect est passé sous silence par le cahier des charges. Tout ce que le cahier des charges garantit, cést que si un utilisateur a plusieurs cartes, elles ont des numéros différents. En résumé, le cahier des charges accepte une réalisation à plusieurs cartes par utilisateur, mais aussi une réalisation à carte unique.

15. La vivacité du système est incomplète : si l'accès est possible, le contrôleur doit agir (point 3 de 3.2), mais il n'est rien dit quand l'accès est refusé (le point 6 de 3.1 ne permet pas de conclure). (rapport IRIT/97-33-R)

    Ce n'est pas si clair car le point 5 de 3.2 impose que si l'accès est refusé, le voyant rouge s'allume pendant 2 secondes.

16. La vivacité des utilisateurs est aussi incomplète : une fois la carte passée, l'utilisateur doit bouger (soit rentrer, soit repartir), mais il n'est pas dit qu'il doit enlever sa carte. (rapport IRIT/97-33-R)

    C'est exact. En fait, les lecteurs de carte de notre bâtiment ne permettent pas d'y oublier la carte.

17. L'incendie est évoqué mais on ne sait pas ce que les utilisateurs ont le droit de faire dans ce cas-là. (rapport IRIT/97-33-R)

    A part passer sa carte dans un lecteur (interdit quand le voyant est allumé), les utilisateurs ont le droit d'ouvrir, de franchir et de fermer toutes les portes.

C. Spécification complète du contrôle d'accès

Nous proposons dans ce chapitre de reconstituer la spécifications complète de l'étude de cas du contrôle d'accès que nous avons traité dans le chapitre 5. Nous éliminons les redéfinitions pour ne présenter que les versions finales des spécifications de chaque partie.

spec $\textsc{Nat}$ =
  free type $Nat ::= 0 \mid Succ(Nat)$
  pred $\_\_ \leq \_\_ : Nat \times Nat$

spec $\textsc{List}$ [**sort** $Elem$] **given** $\textsc{Nat}$ =
  free type $List[Elem] ::= Nil \mid Cons(Hd? : Elem; Tl? : List[Elem])$
  op $Append : List[Elem] \times List[Elem] \rightarrow List[Elem];$
    $Length : List[Elem] \rightarrow Nat;$
    $Cancel : Elem \times List[Elem] \rightarrow List[Elem]$
  pred $\_\_ \in \_\_ : Elem \times List[Elem]$
  $\forall l, l' : List[Elem]; e, e' : Elem$
  • $Append(Nil, l') = l';$
  • $Append(Cons(e, l), l') = Cons(e, Append(l, l'));$
  • $Length(Nil) = 0;$
  • $Length(Cons(e, l)) = Succ(Length(l));$
  • $\neg\, e \in Nil;$
  • $e \in Cons(e', l) \Leftrightarrow e = e' \vee e \in l;$
  • $\neg\, e \in Cancel(e, l);$

end

spec $\textsc{DomaineControle}$ =
  sorts $Porte, Capteur, Batiment, Groupe, Carte, Date, Time;$
  free type $PhysicalUnit ::= sort\ Porte \mid sort\ Capteur;$
  ops $PorteToBat : Porte \rightarrow Batiment;$
    $CapteurToBat : Capteur \rightarrow Batiment;$
    $Appartenir : Carte \rightarrow ?\ Groupe;$
    $delaiouverture, delairefus, 0, 1 : Time;$
    $\_\_ - \_\_ : Time \times Time \rightarrow Time;$
    $\_\_ - \_\_ : date \times date \rightarrow Date;$

```
          Tick : Time → Time;
          Tick : Date → Date;
     preds  Autorise : Groupe × Batiment;
          EnCirculation : Carte;
          Valide : Carte;
          Isin, Isout : Porte;
     ∀c : Carte; p : Porte; t : Time; d : Date
     • ∀b : Batiment • ∃p : Porte • PorteToBat(p) = b;
     • ∀b : Batiment • ∃c : Capteur • CapteurToBat(c) = b;
     • Valide(c) => EnCirculation(c);
     • Isin(p) <=> ¬(Isout(p));
     • ∀b : Batiment • (∃p1 : Porte • Isin(p1) ∧ PorteToBat(p1) = b)
          ∧ (∃p2 : Porte • Isout(p2) ∧ PorteToBat(p2) = b);
     • 0 ≤ Tick(t);
     • Tick(t) < t;
     • Tick(d) − d = t − Tick(t);
end

spec SET[sort Elem] =
     generated type Set[Elem] ::= EmptySet|{__}(Elem)|__ + __(Elem; Set[Elem])
     ops  __ ∪ __ : Set[Elem] × Set[Elem] → Set[Elem];
          __ ∩ __ : Set[Elem] × Set[Elem] → Set[Elem];
          __\__ : Set[Elem] × Set[Elem] → Set[Elem];
     preds  __ ∈ __ : Elem × Set[Elem]
          __ ⊆ __ : Set[Elem] × Set[Elem]


spec INPUTS/OUTPUTS= DOMAINE_CONTROLE then
     free type R_Messages ::= Lecture(Porte, Carte) | Ouverture(Porte)
                    | Fermeture(Porte) | Incendie(Capteur) |Okr(PhysicalUnit);
     free type S_Messages ::= Bloquer(Porte) | Debloquer(Porte) |
               Allumer_rouge(Porte) | Allumer_vert(Porte) |
               Eteindre_rouge(Porte) | Eteindre_vert(Porte) | Oks(PhysicalUnit);
     ops  SMessToPhysUnit : SMessages → PhysicalUnit;
          RMessToPhysUnit : RMessages → PhysicalUnit
then SET[Elem ↦ R_Messages] and SET[Elem ↦ S_Messages]
end

spec JOURNAL = DOMAINECONTROLE then
     sort Journal
     ops  Empty : Journal;
          Enregistrer : Carte × Porte × Date × Journal → Journal;
```

$Intrusion : Porte \times Date \times Journal \to Journal;$
$Defaillance : PhysicalUnit \times Date \times Journal \to Journal;$
pred $Etredans : Carte \times Batiment \times Journal;$

end
spec ETATS = INPUTS/OUTPUTS **and** JOURNAL **then**
sort $Etats$
free type $Status ::= Ok \mid Reject$
free type $SecCri ::=< Porte : Porte, Carte : Carte, Delai : Time, Statut : Status >$
then LIST$[Elem \mapsto SecCri]$
then ops $Init : Etats;$
$ObsDateCourante : Etats \to Date;$
$ObsJournal : Etats \to Journal;$
$ObsSecCri : Etats \to List[SecCri];$
$ObsSMess : Etats \to Set[S\_Messages]$
then
free type $Etats ::= Etats(Journal; List[SecCri]; Set[S\_Messages]; Date)$
then $\forall q : Etats$
• $ObsJournal(Init) = Empty;$
• $ObsSecCri(Init) = Nil;$
• $ObsSMess(Init) = Empty;$
• $ObsJournal(q) = jou(q);$
• $ObsSecCri(q) = seccri(q);$
• $ObsSMess(q) = smess(q);$
• $ObsDateCourante(q) = date(q);$
end

spec ENVOI = ETATS **then**
op $Envoi : Set[R\_Messages] \times Etats \to Set[S\_Messages]$
pred $Autorisation\_in(c : Carte; p : Porte; j : Journal) \Leftrightarrow$
$EnCirculation(c) \wedge Is\_in(p) \wedge \neg Etre\_dans(c, PorteToBat(p), j)$
$\wedge Autorise(Appartenir(c), PorteToBat(p));$
pred $Autorisation\_out(c : Carte; p : Porte; j : Journal) \Leftrightarrow$
$EnCirculation(c) \wedge Is\_out(p) \wedge Etre\_dans(c, PorteToBat(p), j)$
$\wedge Autorise(Appartenir(c), PorteToBat(p))$
$\forall Rmsgs : Set[R\_Messages]; p : Porte; c : Carte; cap : Capteur; b : Batiment; j : Journal; q : Etats;$
%% Message : incendie
• $Incendie(cap) \in Rmsgs \Rightarrow (\forall p : Porte \bullet PorteToBat(p) = CapteurToBat(cap) \Rightarrow Debloquer(p) \in Envoi(Rmsgs, q));$
%% Message : lecture, autorise
• $Lecture(p, c) \in Rmsgs \wedge$
$(Autorisation\_in(c, p, Obs\_Journal(q)) \vee Autorisation\_out(c, p, Obs\_Journal(q)))$

$\Rightarrow Allumer\_vert(p) \in Envoi(Rmsgs, q) \wedge Debloquer(p) \in Envoi(Rmsgs, q)$

%% Message : lecture, refus

• $Lecture(p, c) \in Rmsgs \wedge$
$\neg(Autorisation\_in(c, p, Obs\_Journal(q)) \vee Autorisation\_out(c, p, Obs\_Journal(q)))$
$\Rightarrow Allumer\_rouge(p) \in Envoi(Rmsgs, q);$

%% Message : fermeture

• $Fermeture(p) \in Rmsgs$
$\Rightarrow Bloquer(p) \in Envoi(Rmsgs, q) \wedge Eteindre\_vert(p) \in Envoi(Rmsgs, q);$

%% Delai ouverture expire

• $\forall sc : SecCri \bullet (sc \in Obs\_SecCri(q) \wedge Delai(sc) = 0 \wedge Statut(sc) = Ok)$
$\Rightarrow Bloquer(Porte(sc)) \in Envoi(Rmsgs, q) \wedge Eteindre\_vert(p) \in Envoi(i, q);$

%% Delai refus expire

• $\forall sc : SecCri \bullet (sc \in Obs\_SecCri(q) \wedge Delai(sc) = 0 \wedge Statut(sc) = Reject$
$\Rightarrow Eteindre\_rouge(Porte(sc)) \in Envoi(Rmsgs, q);$

%% Accuse de reception

• $rm \in Rmsgs \wedge rm \neq Okr(PU) \Rightarrow Oks(RMessToPhysUnit(rm)) \in Envoi(Rmsgs, q)$

end

spec MAJ_JOURNAL = ETATS **then**

ops $New\_Journal : Set[R\_Messages] \times Etats \rightarrow Journal;$
$Dater : R\_Messages \times Etats \rightarrow Date;$
$ErreurAccuse : Set[S\_Message] \times Set[R\_Message] \rightarrow List[PhysicalUnit]$

$\forall Rmsgs : Set[R\_Messages]; p : Porte; q : Etats, rm : R\_Messages; Smsgs : Set[S\_Messages]; sm : S\_Messages; PU : PhysicalUnit$

• $Dater(rm, q) = Obs\_DateCourante(q);$

• $ErreurAccuse(EmptySet, EmptySet) = Nil;$

• $rm = Okr(PU) \Rightarrow$
$ErreurAccuse(EmptySet, (rm{+}Rmsgs)) = Cons(PU, ErreurAccuse(EmptySet, Rmsgs));$

• $rm \neq Okr(PU) \Rightarrow$
$ErreurAccuse(EmptySet, (rm + Rmsgs)) = ErreurAccuse(EmptySet, Rmsgs);$

• $Okr(SMessToPhysUnit(sm)) \in Rmsgs \Rightarrow$
$ErreurAccuse((sm + Smsgs), Rmsgs) =$
$Cons(SMessToPhysUnit(sm), ErreurAccuse(Smsgs, Rmsgs));$

• $Okr(SMessToPhysUnit(sm)) \notin Rmsgs \Rightarrow$
$ErreurAccuse((sm + Smsgs), Rmsgs) = ErreurAccuse(Smsgs, Rmsgs);$

• $Ouverture(p) \in Rmsgs \wedge \exists\, sc : SecCri \bullet (sc \in Obs\_SecCri(q) \wedge Porte(sc) = p$
$\Rightarrow New\_Journal(Rmsgs, q) =$
$Enregistrer(Carte(sc), p, Dater(Ouverture(p), q), Obs\_Journal(q));$

• $Ouverture(p) \in i \wedge \neg(\exists\, sc : SecCri \bullet (sc \in Obs\_SecCri(q) \wedge Porte(sc) = p)$
$\Rightarrow New\_Journal(i, q) = Intrusion(Porte(sc), Dater(Ouverture(p), q), Obs\_Journal(q));$

• $\neg Is\_Empty(ErreurAccuse(Obs\_SMess(q), Rmsgs)) \Rightarrow$
$\forall PU : PhysicalUnit\ .PU \in ErreurAccuse(Obs\_SMess(q), Rmsgs) \Rightarrow$
$New\_Journal(Rmsgs, q) =$

$Defaillance(PU, Obs\_DateCourante(q), Obs\_Journal(q))$

end

spec MAJ_SECCRI = ETATS **then**

op $New\_SecCri : Set[R\_Messages] \times Etats \rightarrow List[SecCri]$;

$\forall Rmsgs : Set[R\_Messages]; p : Porte; q : Etats; c : Carte$

%% Les sections critiques terminees sont eliminees

• $\forall sc : SecCri(sc \in Obs\_SecCri(q) \wedge Delai(sc) = 0)$
$\Rightarrow sc \notin New\_SecCri(Rmsgs, q)$;

%% Les autres sont conservees en diminuant le delai de 1 unite de temps

• $sc : SecCri(sc \in Obs\_SecCri(q) \wedge Delai(sc) \neq 0)$
$\Rightarrow < Porte(sc), Carte(sc), Tick(Delai(sc)), Statut(sc) > \in New\_SecCri(Rmsgs, q)$;

%% De nouvelles sections sont ajoutees

• $lecture(p, c) \in Rmsgs \wedge$
$(Autorisation\_in(c, p, Obs\_Journal(q)) \vee Autorisation\_out(c, p, Obs\_Journal(q)))$
$\Rightarrow < p, c, delai\_ouverture, Ok > \in New\_SecCri(Rmsgs, q)$;

• $lecture(p, c) \in Rmsgs \wedge$
$\neg(Autorisation\_in(c, p, Obs\_Journal(q)) \vee Autorisation\_out(c, p, Obs\_Journal(q)))$
$\Rightarrow < p, c, delai\_refus, Reject > \in New\_SecCri(Rmsgs, q)$;

end

spec MAJ_SMESS = ENVOI **then**

ops $CollectSAccuse : Set[S\_Message] \rightarrow Set[S\_Message]$;
$New\_SMess : Set[R\_Message] \times Etats \rightarrow Set[S\_Message]$;

$\forall Rmsgs : Set[R\_Messages]; Smsgs : Set[S\_Messages]; rm : R\_Messages;$
$sm : S\_Messages; q : Etats; PU : PhysicalUnit;$

• $CollectSAccuse(EmptySet) = EmptySet$;

• $sm = Oks(PU) \Rightarrow CollectSAccuse(sm, Smsgs) = CollectSAccuse(Smsgs)$;

• $sm \neq Oks(PU) \Rightarrow CollectSAccuse(sm, Smsgs) = sm + CollectSAccuse(Smsgs)$;

%% Mise à jour des messages envoyes

• $New\_SMess(Rmsgs, q) = Envoi(Rmsgs, q) \backslash CollectSAccuse(Envoi(Rmsgs, q))$

end

spec SYSTEME = MAJ_JOURNAL **and** MAJ_SECCRI **and** MAJ_SMESS **then**

op $T : Set[R\_Messages] \times Etats \rightarrow Etats$

$\forall Rmsgs : Set[R\_Messages]; q : Etats$

• $ObsDateCourante(T(Rmsgs, q)) = Tick(ObsDateCourante(q))$;

• $ObsJournal(T(Rmsgs, q)) = NewJournal(Rmsgs, q)$;

• $ObsSecCri(T(Rmsgs, q)) = NewSecCri(Rmsgs, q)$;

• $T(Rmsgs, q) =$
$< (NewJournal(Rmsgs, q), NewSecCri(Rmsgs, q), NewSMess(Rmsgs, q), Tick(date(q)) >$
;

end

Bibliographie

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings of SIGSOFT'93 : Foundations of Software Engineering*, Software Engineering Notes 18(5), pages 9–20. ACM Press, December 1993.

[ABK+02] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL : The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2) :153–196, 2002.

[Abr96a] J.-R. Abrial. *The B-Book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[Abr96b] Jean-Raymond Abrial. Steam-Boiler Control Specification Problem. In *Formal Methods for Industrial Applications*, volume 1165 of *LNCS*, pages 500–509. Springer-Verlag, 1996.

[ABR99] Egidio Astesiano, Manfred Broy, and Gianna Reggio. Algebraic Specification of Concurrent Systems. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, chapter 13. Springer, 1999.

[AG90] C. Ashworth and M. Goodland. *SSADM- A practical approach*. McGraw Hill, 1990.

[AG97] R. Allen and D. Garlan. A formal basis for architectural connections. *ACM Transactions on Software Engineering and Methodology*, pages 213–249, july 1997.

[AHMS99] S. Autexier, D. Hutter, H Mantel, and Axel Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In Bert D., Choppy C., and Mosses P.D., editors, *Recent Trends on Algebraic Development Techniques, WADT'99*, LNCS 1827, pages 73–88. Springer-Verlag, 1999.

[AM02] S. Autexier and T. Mossakowsi. Integrating HOLCASL into the Development Graph Manager MAYA. In *Proceedings FroCos'02*, LNCS 2303, pages 2–17. Springer-Verlag, 2002.

[AR00] Egidio Astesiano and Gianna Reggio. Formalism and Method. *Theoretical Computer Science*, 236(1-2) :3–34, 2000.

[Bau99] Hubert Baumeister. Methodological Considerations on the use of Extension and Union. CoFI Note M-1, 1999.

[BB00] Hubert Baumeister and Didier Bert. *Software Specification Methods : An Overview Using a Case Study*, chapter Algebraic Specification in CASL. Springer-Verlag, 2000. FACIT (Formal Approaches to Computing and Information Technology).

[BBC86] Gilles Bernot, Michel Bidoit, and Christine Choppy. Correctness Proofs for Abstract Implementations. In *Proceedings of 3rd Symposium on Theoretical Aspects of Computer Science*, LNCS 210, pages 236–251. Springer-Verlag, 1986.

[BBF+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools.* Springer, 2001.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. METEOR : A successful application of B in a large project. In *Proceedings of FM'99 : World Congress on Formal Methods*, LNCS 1709, pages 369–387. Springer-Verlag, 1999.

[BCPR95] M. Bidoit, C. Chevennier, C. Pellen, and J. Ryckbosh. An Algebraic Specification of the Steam-Boiler Control System. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications*, LNCS 1165, pages 79–108. Springer-Verlag, 1995.

[BDDG93] Ralph Betschko, Sabine Dick, Klaus Didrich, and Wolfgang Grieskamp. Formal Development of an Efficient Implementation of a Lexical Scanner within the KORSO Methodology Framework. Technical Report TR-93-30, Technische Universität Berlin, 1993.

[BEH+87] F.L. Baue, H. Ehler, B. Horsch, B. Möller, H. Partsch, O. Paukner, and P. Pepper. *The munich CIP project : The transformation system CIP-L*, volume 2. Springer-Verlag, LNCS 292, 1987.

[Ber01] Didier Bert. Spécification algébrique et prototypage du Controle d'accès en LPG. *Approches formelles pour l'aide au développement de logiciels,TSI*, 20(7) :849–873, 2001.

[BG77] Rod Burstall and Joseph Goguen. Putting Theories Together to Make Specifications. In R. Reddy, editor, *Proceedings 5th International Joint Conference on Artificial Intelligence*, volume 2, pages 1045–1058. William Kaufmann, 1977.

[Bid89] Michel Bidoit. *PLUSS, un langage pour le développement de spécifications algébriques modulaires*. PhD thesis, Universté Orsay, 1989.

[BKBdJ98] M.M. Bonsangue, J.N. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In

J. Carroll, G.B. Lamont, D. Oppenheim, K.M. George, and B. Bryant, editors, *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC '98)*, pages 159–168. ACM press, 1998.

[BKC97] Mario Bass, Rick Kazman, and Paul Clements. *Software Architecture in Practice.* Addison Wesley, 1997.

[BKK+98] P. Borovansky, C. Kirchner, H. Kirchner, P.E. Moreau, and C. Ringeissen. An overview of elan. In *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, volume 15. Electronic Notes in Theoretical Computer Science, 1998.

[Bla99] Benjamin Blanc. Modélisation et spécification d'architectures logicielles. Master's thesis, ENS Cachan, 1999.

[Boe88] B.W. Boehm. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5) :61–72, 1988.

[Bro96] Manfred Broy. Algebraic Specification of Reactive Systems. In M. Nivat and Martin Wirsing, editors, *Algebraic Methodology and Software Technology, 5th International Conference, AMAST '96*, pages 487–503. Springer-Verlag, 1996.

[BST02] Michel Bidoit, Don Sannella, and Andrzej Tarlecki. Architectural Specifications in CASL. *Formal Aspects of Computing*, 13(3-5) :252–273, 2002.

[CM98] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming.* Cambridge University Press, 1998.

[CMR99] Maura Cerioli, Till Mossakowski, and Horst Reichel. From Total Equational to Partial First-Order Logic. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, chapter 3. Springer, 1999.

[CoF01] CoFI Task Group on Semantics. CASL-Semantics (version 1.0.1), 2001. `http://www.brics.dk/Projects/CoFI/Documents/CASL/`.

[CR99] Christine Choppy and Gianna Reggio. Using CASL to Specify the Requirements and the Design : a Problem Specific Approach. In Bert D., Choppy C., and Mosses P.D., editors, *Recent Trends in Algebraic Development Techniques, WADT'99*, LNCS 1827, pages 106–125. Springer-Verlag, 1999.

[Dam98] Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, 192(2) :201–231, Feb 1998.

[EKMP82] Harmut Ehrig, Hans-Jörg Kreowski, Bernd Mahr, and Peter Padawitz. Algebraic Implementation of Abstract Data Types. *Theoretical Computer Science*, 20 :209–263, 1982.

[EM85] Hartmut Ehrig and Bernhard Mahr. Fundamentals of Algebraic Specifications : Equations and Initial Semantics. In *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[EM90] Hartmut Ehrig and Bernhard Mahr. Fundamentals of Algebraic Specifications 2 : Modules Specifications and Constraints. In *EATCS Monographs on Theoretical Computer Science*, volume 21. Springer, 1990.

[FJ90] John Fitzgerald and Cliff Jones. Modularizing the Formal Description of a Database System. In *Proceedings VDM'90*, LNCS 428, pages 198–210. Springer-Verlag, 1990.

[FL97] J.L. Fiadeiro and A. Lopes. Semantics of architectural connectors. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97*, volume 1214 of *LNCS*, pages 505–519. Springer-Verlag, 1997.

[Fle02] Emmanuel Fleury. *Automates temporisés avec mise à jour*. PhD thesis, ENS de Cachan, 2002.

[Flo95] C. Floyd. Theory and Practice of Software Development : Stages in Debate. In *Proceedings of TAPSOFT'95*, LNCS 915, pages 25–41. Springer-Verlag, 1995.

[FS97] M. Fowler and K. Scott. *UML Distilled. Applying the standard Object Modelling Language* . Addison-Wesley, 1997.

[GH93] J.V. Guttag and J.J. Horning. *Larch : Language and Tools for Formal Specifications*. Springer-Verlag, 1993.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software Architecture*. Addison-Wesley, 1995.

[GHM78] John Guttag, E. Horowitz, and D. Musser. Abstract Data Types and Software Validation. *Communications of the ACM*, 21(12) :1048–1064, 1978.

[GL95] Joseph Goguen and Luqi. Formal Methods and Social Context in Software Development. In *Proceedings of TAPSOFT'95*, LNCS 915, pages 62–81. Springer-Verlag, 1995.

[GP95] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. In *IEEE Transactions on Software Engineering*, volume 21, pages 269–274, April 1995.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[GTWW78] Joseph Goguen, J. Thatcher, E. Wagner, and J. Wright. An Initial Algebra Approach to the Specification Correctness, and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology*. Prentice-Hall, 1978.

[GWM+92] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ3. Technical Report SRI-CSL-9203, SRI International, 1992.

[HDRS99] J. Herbert, B. Dutertre, R. A. Riemenschneider, and V. Stavridou. A Formalization of Software Architecture. In *FM'99 – World Congress on Formal Methods in the Development of Computing Systems, Volume I*, number 1708 in Lecture Notes in Computer Science, pages 116–133, Toulouse, France, September 1999. Springer-Verlag.

[Het94] Rudolf Hettler. A Requirement Specification for a Lexical Analyser. Technical Report TU-I9409, Teknische Universität München, 1994.

[HK91] I. Houston and S. King. Cics project report : Experience and results from using Z. In *Proceedings of VDM'91 : Formal Development Methods*, LNCS 551, pages 588–596. Springer-Verlag, 1991.

[HKB93] Berthold Hoffmann and Bernd Krieg-Brückner, editors. *Program Development by Specification and Transformation, The PROSPECTRA Methodology, Language Family, and System.* LNCS 680. Springer-Verlag, 1993.

[Hoa72] C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica*, 1 :271–281, 1972.

[HSN95] Christine Hofmeister, Dilip Soni, and Robert Nord. Describing software architectures with UML. In *Proceedings First Workshop IFIP Conference on Software Architctures*, volume CMU-CS-TR-95-151. Carnegie-Mellon University, 1995.

[Hut00] Dieter Hutter. Management of Change in Structured Verification. In *Proceedings Automated Software Engineering (ASE)*, pages 23–34. IEEE, 2000.

[IW95] P. Inverardi and A.L. Wolf. Formal specification and formal analysis of software architecture using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4) :373–386, April 1995.

[Jac95a] Michael Jackson. *Software Requirements and Specification : A Lexicon of Practice, Principles and Prejudices.* Addison-Wesley, 1995.

[Jac95b] J. Jacky. Specifying a critical control system in Z. *IEEE Transactions on Software Engineering*, 21(2) :99–106, 1995.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1990.

[JZ95] Michael Jackson and Pamela Zave. Deriving Specifications from Requirements : an Example. In *Proceedings 17th International Conference on Software Engineering*, pages 15–24. ACM Press, 1995.

[Kru95] Philippe B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6) :42–50, 1995.

[KST97] Stefan Kahrs, Donald Sannella, and Andrzej Tarlecki. The definition of Extended ML : A gentle introduction. *Theo. Comp. Sc.*, 173 :445–484, 1997.

[Led96] Yves Ledru. Étude de cas : système de contrôle d'accès. http ://www.lsr-imag.fr/Les.Personnes/Yves.Ledru/Controle_acces/, 1996.

[LEW96] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types.* Wiley-Teubner, 1996.

[LKA+95] 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.

[MAH01] T. Mossakowsi, S. Autexier, and D. Hutter. Extending Development Graphs With Hiding. In David Sands, editor, *18th Fundamental Aspects Software Engineering*, LNCS 2029, pages 269–283. Springer-Verlag, 2001.

[Mea55] G.H. Mealy. a method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5) :1045–1079, 1955.

[MKKB98] T. Mossakowski, Kolyang, and B. Krieg-Brückner. Static semantics and theorem proving for CASL. In F. Parisi Presicce, editor, *Recent trends in algebraic development techniques.*, LNCS 1376, pages 333–348. Springer-Verlag, 1998.

[Mor97] Luc Moreau. A Syntactic Theory of Dynamic Binding. In *Proceedings TAPSOFT/FASE'97*, LNCS 1214, pages 727–741. Springer-Verlag, 1997.

[Mos99] Till Mossakowski. Two functional programming sublanguages of CASL. `http ://www.brics.dk/Projects/CoFI/Documents/CASL/Notes/L/L-9/`, 1999.

[Mos02] T. Mossakowski. Heterogeneous Development Graphs and Heterogeneous Borrowing. In *Proceedings FOSSACS'02*, LNCS 2303, pages 326–341. Springer-Verlag, 2002.

[Mos03] Till Mossakowski. Relating CASL with Other Specification Languages : the Institution Level. *Theoretical Computer Science*, 2003. To appear.

[MS95] S.P. Miller and M. Srivas. Formal verification of the AAMP5 microprocessor : a case study in the industrial use of formal methods. In *WIFT'95 : Workshop on Industrial-Strength Formal Specification Techniques*, pages 153–188. IEEE Computer Society, 1995.

[MTH90] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[ONS96] Fernando Orejas, Marisa Navarro, and Ana Sanchez. Algebraic Implementation of Abstract Data types : a Survey of Concepts and New Compositionnality Results. *Mathematical Structures in Computer Science*, (6) :33–67, 1996.

[Ore99] Fernando Orejas. Structuring and Modularity. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, chapter 6, pages 159–200. Springer, 1999.

[Pau94] L.C. Paulson. *Isabelle- A Generic Theorem Prover*. LNCS 828. Springer-Verlag, 1994.

[PCW85] David Parnas, Paul Clements, and David Weiss. The Modular Structure of Complex Systems. In *IEEE Transactions on Software Engineering*, volume SE-11, pages 259–266, 1985.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architecture. In *ACM SIGSOFT'92 : Symposium on Software Engineering*, October 1992.

[PW95] P. Pepper and M. Wirsing. A Method for the Development of Correct Software. In M. Broy and S. Jähnichen, editors, *KORSO : Correct Software by Formal Methods*, LNCS, pages 27–57. Springer-Verlag, 1995.

[RAC99] Gianna Reggio, Egidio Astesiano, and Christine Choppy. CASL-LTL. A CASL Extension for reactive systems. CoFI Note, 1999.

[SC97] M. Shaw and P. Clements. A field guide to boxology : Preliminary classification of architectural styles for software systems. In *COMPSAC97*. IEEE Computer, 1997.

[SD95] Jeanine Souquières and Robert Darimont. La description du développement des spécifications. *TSI*, 14(9), 1995.

[SG96] Mary Shaw and David Garlan. *Software Architecture. Perspectives on an emerging discipline*. Prentice-Hall, 1996.

[SH99] 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 ER'99*, LNCS 1728, pages 309–324. Springer-Verlag, 1999.

[SM02] L. Schröder and T. Mossakowski. HasCASL : Towards Integrated Specification and Development of Haskell Programs. In *Proceedings AMAST'02*, LNCS 2422, pages 99–116. Springer-Verlag, 2002.

[SMT+01] Lutz Schroeder, Till Mossakowski, Andrzej Tarlecki, Bartek Klin, and Piotr Hoffman. Semantics of Architectural Specifications in CASL. In *Proceedings FASE'02*, LNCS 2029, pages 253–268. Springer-Verlag, 2001.

[SNH95] Dilip Soni, Robert Nord, and Christine Hofmeister. Sofware architecture in industrial applications. In *Proceedings of the 17th ICSE*, pages 196–207, avril 1995.

[Som01] Ian Sommerville. *Software Engineering*. Addison Wesley, 6th edition, 2001.

[Spi92] J.M. Spivey. *The Z Notation, A Reference Manual*. Prentice-Hall, 1992.

[SST92] Donald Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward Formal Development of Programs from Algebraic Specifications : Parameterisation Revisited. *Acta Informatica*, 29(8) :689–736, 1992.

[ST88] Donald Sannella and Andrzej Tarlecki. Toward Formal Development of Programs from Algebraic Specifications : Implementation Revisited. *Acta Informatica*, 25 :233–281, 1988.

[ST99] Andrea Savigni and Francesco Tisato. Kaleidoscope : A Reference Architecture for Monitoring and Control Systems. In Patrick Donohoe, editor, *Proceedings of the first Working IFIP Conference on Software Architecture*, pages 369–388, 1999.

[SvK97] Sylvia Stuurman and Jan van Katwijk. Evaluation of software architecture for a control system : a case study. In *COORDINATION'97*, volume 1201 of *LNCS*, pages 157–171. Springer-Verlag, 1997.

[SW83] Donald Sannella and Martin Wirsing. A Kernel Language for Algebraic Specification and Implementation. In *Proceedings International Conference on Foundations of Computation Theory*, LNCS 158. Springer-Verlag, 1983.

[vdPHdJ99] Jaco van de Pol, Jozef Hooman, and Edwin de Jong. Requirements Specification and Analysis of Command and Control Systems. Technical report, CWI Amsterdam, 1999.

[Win95] Jeannette Wing. Hints to specifiers. Technical Report CMU-CS-95-118R, Carnegie-Mellon University, 1995.