Une approche formelle de l'interopérabilité pour une famille de langages dédiés

Ali Abou Dib

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par *l'Université Toulouse III - Paul Sabatier*
Discipline ou spécialité : Informatique

Présentée et soutenue par ***Ali Abou Dib***
Le 18 décembre 2009

Une approche formelle de l'interopérabilité pour une famille de langages dédiés

---

JURY

Jean-Philippe Babau, Professeur, Université de Bretagne Occidentale
Henri Basson, Professeur, Université du Littoral Côte d'Opale
Louis Féraud, Professeur, Université Toulouse III
Ileana Ober, Maître de conférences, Université Toulouse III
Christian Percebois, Professeur, Université Toulouse III
Ervann Poupart, Ingénieur CNES, Toulouse

Ecole doctorale : EDMITT
Unité de recherche : IRIT
Directeur de Thèse : Christian Percebois
Encadrement : Ileana Ober
Rapporteurs : Jean-Philippe Babau et Henri Basson

A la mémoire de mon père,
A toute ma famille,

Ma très sincère gratitude va à Monsieur le Professeur Christian PERCEBOIS pour la direction de mes travaux de thèse et son initiation avec bienveillance à la recherche. Il a su avec succès me transmettre son dynamisme. Je remercie également Madame Ileana OBER pour son aide scientifique et morale qu'elle m'a apportée durant toutes ces années. Je les remercie profondément pour leurs suggestions et recommandations. Ce fut un honneur de travailler avec eux surtout leurs qualités humaines et leurs compétences scientifiques.

Je remercie également Monsieur le Professeur Louis FERAUD qui m'a m'aidé à franchir des difficultés majeures rencontrées durant mes travaux. Ses remarques éclairées m'ont notamment permis d'améliorer ce manuscrit. Je le remercie d'avoir accepté la présidence de ce jury.

Je remercie les rapporteurs de cette thèse, Monsieur le Professeur Henri BASSON et Monsieur le Professeur Jean-Philippe BABAU, pour l'intérêt qu'ils ont porté à mon travail. Merci également à Monsieur Erwann POUPART pour leur participation au jury.

Mes pensées s'adressent aussi à mon frère Hussein. Sans lui, je n'aurais jamais eu la possibilité de poursuivre ainsi mes études. Merci à ma mère, à mes sœurs et à mes frères pour leur confiance, leurs encouragements sans faille et leur assistance aussi bien matérielle que morale.

Je remercie aussi tous les membres de l'équipe MACAO, qui m'ont toujours encouragé et aidé.

Enfin, je remercie mes amis pour leur présence et leur fidélité.

Résumé : Dans cette thèse, nous proposons une méthode rigoureuse, formellement fondée pour traiter de l'interopérabilité d'une famille de langages dédiés (DSL) issus d'un même domaine métier. A partir de la sémantique de chacun des DSL, notre démarche construit, par un calcul de co-limite sur des spécifications algébriques, un langage qui unifie les concepts de la famille. L'approche se caractérise notamment par la capacité à traduire automatiquement le code d'un DSL vers le langage unificateur. Un autre bénéfice réside dans la preuve qu'une propriété sur un langage de la famille se décline, par construction, vers l'environnement unifié. La mise en œuvre de la démarche a été outillée ; elle s'appuie principalement sur le logiciel Specware de Kestrel et l'assistant de preuve Isabelle.

Abstract : In this thesis, we introduce a rigorous formally founded method to address the interoperability of a family of domain specific languages (DSLs). Our approach targets DSLs from the same business domain and consists in constructing a unifying language. The unification is obtained by using a categorical approach. For this, we use the category of algebraic specifications of each language in the family. On this category, we apply the colimit on algebraic specifications of each DSL to generate their unifying. This approach allows us to obtain a set of translators from each family member to the unifying language. Moreover, properties established in the context of a language of the family are transferred to the unifier. The approach experimented in the context of the Specware software for categorical computations and of the Isabelle proof assistant.

Table des matières

Introduction

Le développement de systèmes fait intervenir plusieurs langages, plusieurs environnements de développement, plusieurs techniques, etc. Cette hétérogénéité induit la question de l'interopérabilité, récurrente en informatique, qui se pose à différents niveaux et dans des contextes variés : entre différents logiciels, entre différents langages de développement ou de modélisation, entre des abstractions différentes, etc.

Le travail présenté dans cette thèse a pour source un problème concret : l'interopérabilité d'une famille de langages dédiés au domaine spatial. Pour résoudre ce problème, une première approche a été mise en œuvre, consistant à méta-modéliser chacun des langages. La difficulté réside alors à faire coopérer ces méta-modèles. En effet, la portée des résultats d'une telle approche est limitée par le fait qu'elle ne prend pas en compte les aspects sémantiques, les méta-modèles étant par nature des descriptions structurelles. Par conséquent, l'outillage est forcément limité à des transformations de modèles et ne couvre pas le comportement des composants spécifiés avec ces méta-modèles.

Pour pallier les difficultés sémantiques exposées ci-dessus, nous avons décidé d'infléchir le cadre conceptuel vers la modélisation orientée domaine et de traiter en tant que tels les «*Domain Specific Languages*» (DSL) qui caractérisent une famille. L'utilisation de ces langages spécifiques à un secteur d'activités a connu un essor ces dernières années, notamment en réaction aux formalismes généralistes tels UML et SysML.

L’intérêt des DSL réside donc dans leur simplicité et leur relation forte à un domaine d’application. Le revers de la médaille est que des experts différents d’un même domaine soient tentés de créer des DSL variés pour exprimer des besoins similaires sur un champ sémantique identique conduisant ainsi à une famille de DSL. Par exemple, dans le domaine spatial, la définition de procédures opérationnelles pour mettre en œuvre un ensemble de services sur des éléments embarqués (satellites ou segments-sol) se fait en Pluto à l’ESA, en Stol à la Nasa et en Elisa chez EADS. Cette approche polyglotte d’un même domaine conduit à des problèmes importants d’interopérabilité lorsque les experts de ce domaine doivent collaborer.

La sémantique des DSL est un point délicat. Elle est souvent donnée par des compilateurs ou des générateurs implantés dans des langages généralistes rendant celle-ci inadéquate à la génération automatique de

traitement des DSL. Ce constat nous a conduits à proposer une méthode rigoureuse, formellement fondée pour traiter sémantiquement d'une telle interopérabilité en construisant un langage unificateur de la famille. Décrire la sémantique d'un langage peut se faire de diverses manières en utilisant par exemple une approche axiomatique, opérationnelle, dénotationnelle, etc. Notre choix s'est porté sur les spécifications algébriques. Ce choix est motivé essentiellement par trois considérations.

Tout d'abord, les sémantiques des DSL sont relativement concises et les décrire par des spécifications algébriques est raisonnable de par l'usage de la logique équationnelle qui sous-tend ce formalisme. Ensuite, via la théorie des catégories, la somme amalgamée de spécifications est formellement descriptible et fournit le langage unificateur. Enfin, le choix des spécifications algébriques nous permet de fonder l'outillage de l'approche via les environnements Specware et Isabelle.

Outre la rigueur, cette stratégie de traitement d'une famille de DSL offre d'autres avantages. Ainsi, l'expression algébrique des sémantiques conduit à la définition de traducteurs de chaque membre de la famille vers le langage unificateur. Un autre bénéfice de l'utilisation de spécifications algébriques dans un cadre catégorique réside dans la preuve de propriétés. Une propriété donnée sous forme d'un invariant dans un membre de la famille se propage, par construction, au sein du langage unificateur.

Après avoir mis en évidence le besoin d'interopérabilité pour les DSL issus d'un même domaine d'expertise, nous exploitons le cadre théoriquement fondé des spécifications algébriques pour décrire la syntaxe et la sémantique d'un DSL et considérons ces spécifications algébriques comme les objets d'une catégorie. La construction du langage unificateur de la famille par co-limite s'accompagne d'une préservation des propriétés qui caractérisent les différents langages de la famille, tant d'un point de vue des langages que d'un point de vue des composants.

Ce mémoire se divise en 4 chapitres. Dans le premier chapitre, nous soulignons le problème de l'interopérabilité d'une famille de DSL et montrons qu'il se traduit par l'impossibilité de faire émerger un langage commun à un même secteur d'activités. La difficulté est essentiellement d'ordre sémantique, ce qui nous a conduits à analyser les différentes notations sémantiques au chapitre 2. Nous détaillons au chapitre 3 notre approche fondée sur la catégorie des spécifications algébriques. Ce formalisme permet de définir la sémantique des différents DSL de la famille et d'obtenir par co-limite un langage

unificateur. La démarche est illustrée sur les procédures opérationnelles du domaine spatial où plusieurs langages relativement indépendants ont été définis par les agences spatiales. Au chapitre 4, nous concluons cette étude par une évaluation de l'approche et adressons quelques développements qui resteraient à élaborer.

Chapitre I

1-Interopérabilité et langages dédiés

Le problème de l'interopérabilité est connu dans des nombreux domaines scientifiques. En informatique, il s'agit de composants qui fonctionnent ensemble mais qui n'ont pas été conçus pour celà, pas exemple parce qu'ils sont écrits dans des langages différents ou parce qu'ils utilisent des représentations de données différentes. Ces composants peuvent être des outils logiciels ou des composants matériels qui peuvent communiquer entre eux. Dans notre étude, on ne s'intéresse qu'à l'interopérabilité au niveau des composants logiciels, plus particulièrement à l'interopérabilité au niveau des langages de programmation.

Théoriquement, la solution au problème de l'interopérabilité de langages de programmation consiste à trouver une stratégie afin de faire communiquer des logiciels, ou des composants, différents. Plusieurs stratégies peuvent être envisagées pour résoudre un tel problème : utiliser un composant intermédiaire qui assure la communication, transformer les spécifications de deux logiciels, ou composants, en deux autres nouveaux, qui interagissent plus facilement. Dans cette thèse, on s'intéresse à un sous-ensemble du problème de l'interopérabilité des langages de programmation : celui de l'interopérabilité des langages dédiés à un même domaine.

L'essor des langages dédiés (DSL) rend nécessaire l'étude de leur interopérabilité. Le problème de l'interopérabilité se retrouve simplifié lorsqu'on fait référence à des langages dédiés à un même domaine, notamment parce que ces langages ont des propriétés similaires et partagent des fonctionnalités proches. Ainsi, par l'abstraction des concepts communs des DSL d'un domaine, on peut envisager d'arriver à un langage unificateur de DSL du domaine.

Ce chapitre présente un survol de l'état de l'art de deux des notions les plus importantes par rapport à notre travail de thèse. Nous allons regarder de plus près les notions d'interopérabilité et les langages dédiés.

Dans une première partie, nous présentons en détail quelques approches développées afin de résoudre le problème de

l'interopérabilité de logiciels. On s'intéressera de plus près à l'interopérabilité des langages de programmation.

La deuxième partie est consacrée aux langages dédiés. Nous allons les définir, les caractériser et regarder de plus près leurs propriétés. Par la suite, nous présenterons un processus de conception de DSL, ce qui nous permettra d'introduire la notion de famille de langages dédiés.

1.1-L'interopérabilité dans l'informatique

La notion d'interopérabilité est connue dans différents domaines scientifiques, ce qui fait que ses définitions abondent sans converger. Le dictionnaire généraliste Webster la définit par « *l'habilité d'un système à utiliser les parties d'un autre système* ».

Dans un dictionnaire généraliste Mediadico [1] l'interopérabilité est définie comme « *la capacité de plusieurs systèmes, unités ou organismes dont les structures, les procédures et les relations respectives autorisent une aide mutuelle qui les rend aptes à opérer de conserve* ». Une autre définition intéressante est fournie par IEEE [2], qui voit l'interopérabilité comme la capacité de deux ou plusieurs systèmes ou composants d'échanger des informations et d'utiliser ces informations échangées *(« Ability for two (or more) systems or components to exchange information and to use the information that has been exchanged »).*

Enfin, une autre définition de l'interopérabilité est présentée dans le livre blanc de l'Association Européenne de l'Industrie des Technologies de l'Informatique et des Communications (EICTA) [3]. Il définit l'interopérabilité en informatique comme : « *la capacité de deux ou plusieurs réseaux, systèmes, dispositifs, applications ou composants d'échanger de l'information entre eux et d'utiliser l'information échangée* ».

Dans ces différentes définitions on retrouve l'existence de plusieurs composants dont chacun a des ressources et des informations à partager ou à communiquer avec les autres composantes. L'interopérabilité revient à trouver les média qui permettent cette communication tout en gardant une certaine cohérence sémantique des données échangées.

Notons la proximité des notions d'interopérabilité et de compatibilité. La compatibilité est une relation binaire entre deux composants qui peut être définie de la façon suivante : supposons que *x* et *y* soient deux composants informatiques matériels ou logiciels. Ils sont

compatibles si et seulement si les constructeurs de *x* permettent de travailler ensemble avec ceux de *y*.

Contrairement à la comparaison qui est une notion binaire, l'interopérabilité est une notion générale. En plus, l'interopérabilité des composants *x* et *y* peut être réalisée grâce à des composants externes, ce que ne les rend pas compatibles.

L'interopérabilité est un problème récurrent en informatique. On peut parler d'interopérabilité au niveau logiciel et matériel. Au niveau logiciel, l'interopérabilité représente la capacité de faire travailler ensemble différents composants logiciels, alors qu'au niveau matériel on se réfère à l'interopérabilité des composants matériels, qui peut éventuellement aussi entraîner des différences au niveau logiciel.

D'autres travaux, tels que [4], classent le problème de l'interopérabilité en trois groupes : l'interopérabilité au niveau matériel, au niveau logiciel et au niveau entreprise. Le niveau entreprise concerne l'interopérabilité logicielle et matérielle survenue au sein de techniques utilisées par une même entreprise [5].

1.1.1- Interopérabilité au niveau matériel

L'interopérabilité au niveau matériel est le problème le plus ancien dans le monde de la communication entre les composants informatiques ; elle apparaît de plus en plus cruciale dans le domaine des réseaux informatiques.

Ce type d'interopérabilité concerne la communication entre les ordinateurs, dans un réseau informatique ou avec le réseau global [6] [7] [8]. La présence des protocoles de communication et des architectures de réseaux peuvent alourdir ce problème. Les frontières entre la compatibilité au niveau matériel et logiciel sont très minces, notamment à cause du fait que des parties logicielles peuvent interférer avec la communication des composants matériels

Notre travail ne concerne pas cette forme de l'interopérabilité.

1.1.2- Interopérabilité au niveau logiciel

Au niveau logiciel, l'interopérabilité concerne des composants logiciels qui doivent travailler ensemble. Notre travail s'intéresse à l'interopérabilité au niveau des langages de programmation. Il s'agit de faire coopérer les applications/composants écrit(e)s dans des langages

de programmation différents, ou de les réutiliser au sein d'une nouvelle application.

Des travaux importants [9] [10] ont été entrepris, notamment par l'OMG [11], durant les années 90, afin de résoudre le problème d'interopérabilité entre les logiciels. Beaucoup de ces travaux ont été basés sur des techniques *middelware* comme CORBA (Common Object Request Broker Architecture) [9]. Il s'agit de définir un niveau intermédiaire (appelé *middleware)* qui prend en charge le dialogue entre les logiciels. La technique CORBA a été initiée par différents éditeurs de logiciels regroupés au sein de l'Object Management Group (OMG). CORBA définit une architecture logicielle, permettant le développement avec des objets différents, qui peuvent être écrits dans des langages de programmation distincts, et qui peuvent être assemblés dans des applications complètes.

Des approches plus classiques [12] sont basées sur l'utilisation de messages pour passer les informations entre les applications comme nous allons le voir à la section 1.3.1. D'autres comme [13] sont basées sur l'idée d'un méta-langage dans lequel les langages doivent respecter les normes de l'environnement pendant la phase de développement ; cette approche fera l'objet de la section 1.3.5.

Dans notre travail, on s'intéresse exclusivement à l'interopérabilité au niveau des langages de programmation.

1.1.3-Interopérabilité au niveau des entreprises

Un troisième genre d'interopérabilité concerne les composants matériels et logiciels qui doivent coopérer au sein des applications d'une même entreprise. Ce niveau d'interopérabilité est incontournable pour assurer la pérennité économique de l'entreprise. A ce stade, l'interopérabilité peut être vue comme *« la capacité des entreprises à structurer, formaliser et présenter leurs connaissances et savoir-faire, afin d'être en mesure de les échanger ou de les partager »*. Le problème d'interopérabilité des entreprises passe souvent par celui de ses systèmes d'information, étant donné la forte dépendance entre les entreprises et leurs systèmes d'information [5].

Une autre vue de l'interopérabilité au niveau des entreprises est fournie par [14]. D'après ces auteurs, l'interopérabilité des entreprises est l’aptitude des systèmes à pouvoir travailler ensemble sans effort particulier pour les utilisateurs de ces systèmes. Dans cette vision, issue du réseau d’excellence européen INTEROP [15], l’interopérabilité se traduit par la capacité des entreprises – hors

intervenants humains – à supporter, de manière transparente pour les utilisateurs, les contraintes et conséquences des besoins d'intégration.

Dans le cadre de cette thèse, nous nous intéressons à l'interopérabilité des logiciels, sans prendre en compté le côté entreprise.

1.2-Langages dédiés

Avec l'augmentation des besoins humains, les systèmes informatiques deviennent de plus en plus complexes et sophistiqués. Les applications doivent se recomposer et se reconstruire en permanence. Elles sont souvent développées à partir de plusieurs autres applications moins complexes visant des problèmes plus petits et plus précis.

Plusieurs langages de programmation peuvent être utilisés pour décrire ces applications afin de répondre aux différents besoins. Les langages de programmation utilisés au quotidien sont souvent d'ordre généraliste, comme C [16], C++ [17], Java [18], etc. Ils peuvent servir à résoudre de vastes types de problèmes de nature différente.

A côté des langages généralistes, d'autres, plus spécifiques et moins complexes ont été développés pour résoudre des problèmes précis et visent un métier a priori déterminé. Il s'agit des langages dédiés, en anglais Domain Specific Languages (DSL). Signalons qu'à l'origine il s'agissait souvent de langages généralistes comme Fortran [19], Lisp [20] ou Cobol [21], qui ont évolués et se sont spécialisés aux besoins spécifiques d'une classe de problèmes à résoudre, un type de machine, ou à d'autres spécificités des composants informatiques.

1.2.1-Définition

On dit d'un langage qu'il est dédié à un domaine s'il contient des constructions spécifiques à ce domaine, s'il permet à l'utilisateur de manipuler des concepts situés à un niveau d'abstraction supérieur et si par sa simplicité, il permet l'accès aux experts du domaine (même s'ils ne sont pas informaticiens). Selon [22], les DSL sont des langages de programmation restreints à un domaine précis pour résoudre des problèmes déterminés dans un contexte particulier. Ces langages se caractérisent par leur adéquation à prendre en compte la spécification des besoins au plus près du niveau d'abstraction du domaine traité. Ils proposent en général une notation appropriée et des constructions spécifiques au domaine, ce qui permet d'accroître la lisibilité, la

concision et la sûreté des programmes ainsi que l'accessibilité au langage pour des experts non-informaticiens [23].

Les DSL sont souvent des langages peu complexes, souvent déclaratifs, qui sont moins expressifs qu'un langage généraliste. Ils sont aussi appelés [24] « *little language* » ou « *macro language* ».

Prenons l'exemple des langages dédiés utilisés dans les systèmes de gestion de données. Plusieurs langages de programmation ont été développés spécifiquement afin de traiter et manipuler des informations. Ces langages sont utilisés pour décrire des applications, sous forme de requêtes afin d'extraire, d'ajouter ou de modifier les informations. Ces langages, comme SQL [25], Oracle [26] etc., sont bien définis sur ce domaine et servent à réaliser des tâches particulières, ce qui correspond à la définition d'un DSL.

Au travers de son interprète de commandes (shell), l'environnement Unix constitue un autre exemple de DSL. L'utilisateur peut aisément gérer ses données et ses processus grâce notamment aux commandes de manipulation de fichiers (cp, chmod, make, etc.) incluant redirections (stdin, stdout, etc.) et tubes (pipes), aux commandes de gestion des processus (kill, ps, nice, etc.) et aux outils de développement (éditeurs, compilateurs, metteurs au point, etc.).

D'autres langages ont été développés dans de domaines différents comme le langage XML [27], le langage RISLA spécifique au domaine des produits financiers [28], les langages des systèmes téléphoniques [29], les pilotes de périphérique [22], etc.

1.2.2-Propriétés des langages dédiés

L'utilisation de langages dédiés n'est pas spécifique à un domaine ou à un champ unique d'applications. Plusieurs exemples ont été signalés dans le paragraphe précédent qui montrent la diversité des domaines concernés par les DSL. La plupart des utilisateurs font appel à ces langages parce qu'ils présentent des avantages par rapport aux autres langages plus généralistes. Le plus important est le fait qu'un langage dédié permet d'utiliser des constructions qui suivent de près les abstractions du domaine. Sa sémantique donne l'impression aux utilisateurs qu'ils travaillent directement avec les concepts du domaine. Ainsi, les spécifications obtenues, représentent souvent à la fois la conception, la spécification, et la documentation du système. Dans une thèse de doctorat [22] sur le développement des pilotes, l'auteur souligne l'intérêt des langages dédiés dans une telle démarche.

Dans les paragraphes suivants, nous allons présenter certains des avantages offerts par l'utilisation d'un langage dédié.

1.2.2.1- Facilités offertes pour la spécification

Un DSL provient souvent d'une abstraction de haut niveau suivie d'une phase de spécification des besoins du domaine métier. Les notations utilisées dans sa spécification sont inspirées de celles couramment utilisées dans le domaine métier considéré. Elles sont connues des experts du domaine qui peuvent ne pas être des programmeurs. Ce style de conception d'un langage est à la base de la construction d'un DSL, permettant d'augmenter la lisibilité et la concision des spécifications ainsi que l'accessibilité au langage.

De plus, grâce aux notations offrant souvent des terminologies usuelles, l'apprentissage du DSL, par un spécialiste du domaine, demande moins d'efforts d'une part, et d'autre part facilite la compréhension des spécifications.

Les langages dédiés sont souvent déclaratifs [30], ce qui permet aux programmeurs de se concentrer sur ce qu'ils doivent spécifier *(what to compute)*, et non sur comment le faire (*how to compute*). En plus, avec les DSL, les spécialistes d'un domaine métier ne sont plus obligés de maîtriser des langages de programmation. En effet, leur expérience dans un domaine métier peut s'avérer suffisante pour l'utilisation d'un DSL de ce domaine.

La programmation devient plus efficace puisque les erreurs sont détectées plus tôt dans le processus de production. En effet, cette propriété vient du fait qu'on utilise des abstractions spécifiques au domaine qui permettent de mieux analyser les programmes [22].

1.2.2.2- Facilités offertes pour la réutilisation

Un DSL est développé afin de réaliser des services spécifiques à un domaine métier. Ainsi, il offre un ensemble de fonctions réalisant ces services spécifiés via une syntaxe adaptée au domaine. Un DSL dispose, le plus souvent, d'une bibliothèque restant toujours disponible aux développeurs afin qu'ils puissent réutiliser systématiquement leurs fonctions. Cette bibliothèque représente une abstraction des fonctionnalités nécessaires pour le domaine métier couvert par le DSL.

Un DSL concentre les connaissances de son domaine et les concrétise

sous forme de fonctions spécifiques au domaine. Ces fonctions sont spécifiées de façon à ce qu'elles soient utilisables facilement à partir d'expertises du domaine sans avoir besoin d'approfondir les connaissances en programmation. On peut alors dire qu'à travers les fonctionnalités spécifiques au domaine et à travers les bibliothèques mises à disposition, un DSL offre une réutilisation de fonctionnalités qui couvre les besoins de la spécification abstraite jusqu'à la réutilisation de code.

1.2.2.3- Facilités offertes pour améliorer la qualité

La correction d'une spécification effectuée avec un langage dédié peut être assurée soit par construction, soit par vérification comme expliqué par la suite.

L'utilisation de langages dédiés engendre l'utilisation d'opérations spécifiques à un certain domaine. Le plus souvent, ces opérations se traduiraient en spécifications plus complexes dans un langage généraliste. Cela permet d'augmenter la fiabilité des spécifications effectuées dans le langage dédié, par rapport à d'autres spécifications équivalentes effectuées dans un langage généraliste.

Dans un papier sur le langage PLAN-P [31] dédié aux réseaux actifs, les auteurs montrent comment on peut assurer la correction d'un programme écrit en PLAN-P par rapport à des propriétés critiques et comment le fait d'utiliser ce langage dédié facilite cette tâche.

1.2.3-Langage dédié externe vs. langage dédié interne

Dans ce paragraphe, nous poursuivons le survol des classifications visant à préciser la notion de langage dédié.

Dans un article d'analyse des langages dédiés [30], Martin Fowler propose comment caractériser les langages dédiés selon leur domaine d'application et leur fonctionnalité au sein d'une application ; il introduit ainsi les langages dédiés internes et externes.

1.2.3.1- Langage dédié externe

On définit un langage dédié comme étant externe s'il est différent du langage essentiellement utilisé au sein d'une application

ou d'un domaine. Un bon exemple de langage dédié externe sont les fichiers de configuration du langage XML. En effet, en regardant un projet Java, les fichiers de configuration écrits en XML sont plus nombreux que ceux écrits en Java. Ces derniers sont plus difficiles à comprendre que ceux écrits en XML. Dans ce cas, XML est un langage dédié externe puisque c'est le langage Java qui est le langage essentiel dans un projet Java.

L'avantage de l'utilisation d'un langage dédié externe provient de la liberté du choix du langage de programmation, afin de répondre aux besoins de l'application. Le choix est limité seulement par la capacité d'écrire un transformateur capable d'analyser les fichiers de configuration et de produire des objets exécutables.

D'un autre côté, l'inconvénient d'utiliser un langage dédié externe est lié à la nécessité de construction de ses outils de transformation, qui exigent souvent un grand effort, malgré la faible complexité de la syntaxe du langage. En plus, pour des langages plus complexes, cette phase de construction nécessite un outillage plus lourd. Pour construire ces outils, dans le cas d'un langage généraliste, nous pouvons profiter de l'existence d'outils comme les générateurs, les analyseurs et les outils de compilation, ce qui rend la construction des outils dédiés moins complexe.

Un autre désavantage issu de l'utilisation d'un langage dédié externe (appelé *symbolic integration* dans [30]) vient du fait que le DSL n'est pas correctement lié au domaine métier. Les environnements du langage de base ne sont pas appropriés au langage dédié externe et les nouveaux environnements deviennent plus complexes et de plus en plus sophistiqués. Ainsi le problème de l'intégration symbolique devient plus difficile à résoudre.

Enfin, l'utilisation de langages dédiés externes engendre des problèmes supplémentaires comme la cacophonie des langages [30] due à la complexité de la lecture de l'ensemble des langages utilisés au sein d'une même application.

1.2.3.2- Langage dédié interne

Un langage dédié est dit interne s'il est défini à partir du langage essentiel d'une application ou d'un domaine métier. Un bon exemple de langage dédié interne est le langage Lisp [20], qui est souvent customisé pour des besoins spécifiques. On peut l'utiliser seul et sans la nécessité d'un autre langage complémentaire pour développer une application, tout en utilisant les outils spécifiques à

Lisp.

Avec un langage dédié interne, le problème de l'intégration du DSL dans le cadre du domaine ne se pose plus car le langage est construit sur la base du langage de base. De plus, on dispose directement des outils de manipulation et de transformation du langage dédié.

Un autre avantage de l'utilisation d'un DSL est qu'il n'exige pas la connaissance de notions spécifiques d'informatique. En effet, une bonne maîtrise du domaine métier est suffisante, dans la mesure où le langage avec lequel on écrit les spécifications est lui même suffisant.

L'inconvénient lié à l'utilisation d'un langage dédié interne vient du fait que la maîtrise des concepts du domaine est essentielle.

Pour conclure, le choix entre l'utilisation d'un langage dédié interne ou externe pour développer une application dépendra des besoins spécifiques et des coûts engendrés. Par exemple, avec un langage dédié externe, on impose l'utilisation d'un langage de programmation généraliste, ainsi la maîtrise du domaine métier n'est pas essentielle. Par contre, il faut introduire de nouveaux outils de manipulation et de compilation de ce langage, ce qui augmente d'autant le coût de production. D'un autre côté, un langage dédié interne ne nécessite pas de nouveaux outils, mais les développeurs doivent avoir une bonne connaissance du domaine.

1.2.4-Conception d'un langage dédié

Dans cette section, nous abordons la conception d'un langage dédié. Ce processus de conception demande une phase d'analyse, qui peut être une analyse de domaine ou une analyse de la famille de DSL. Les résultats obtenus avec ces méthodes sont utilisés comme des entrées par le processus de conception du DSL.

Étant donné qu'un langage DSL est lié directement à son domaine d'utilisation ou à un ensemble d'applications liées, une analyse de ces deux notions peut nous aider à bien identifier les besoins liés à la définition du langage dédié. On peut par la suite utiliser cette analyse dans notre approche pour résoudre le problème d'interopérabilité de ces DSL.

1.2.4.1- Analyse de domaine

L'objectif de l'analyse d'un domaine est d'identifier le type et la

forme de connaissances du domaine afin de définir le contenu et la forme d'éléments d'information qui peuvent être réutilisés dans la construction de logiciels de ce domaine. L'idée ici est de définir et d'analyser un domaine afin d'établir une méthodologie pour développer un nouveau langage dédié à ce domaine.

Le processus d'analyse de domaine est une activité clé dans la conception d'un langage dédié. Il consiste à déterminer les éléments réutilisables dans le domaine. Selon James Neighbors [32], le premier à avoir défini le terme d'analyse de domaine :

« *Une analyse de domaine tente d'identifier les objets, les opérations et les relations entre ce que les experts du domaine considèrent comme important pour ce domaine* » [32].

Dans sa méthode, Neighbors, ne s'intéresse qu'au résultat, sans se concentrer sur le processus d'analyse de domaine. Dans ce sens, McCain [33] propose une méthodologie qui intègre les résultats du processus de conception au sein d'un processus de développement des logiciels. Son approche [33] est construite sur selon trois étapes :

1. l'identification des éléments du domaine qui peuvent être réutilisables ;
2. l'abstraction ou la génération de ses composantes ;
3. la classification pour une réutilisation future.

D'autres approches permettent l'analyse d'un domaine, telles que celles d'Arango [34] et de Prieto-Diaz [35].

Arengo [34] définit sa méthode comme un processus qui permet d'identifier, d'extraire et de faire évaluer de l'information sur un domaine afin de la réutiliser pour construire des systèmes dans ce domaine. A partir des éléments identifiés d'un domaine, on extrait les opérations communes au domaine, comme le préconise aussi [35]. On constitue ainsi la bibliothèque des opérations communes spécifiques au domaine.

D'autres techniques, comme par exemple [36], permettent d'analyser un domaine et d'identifier ses éléments essentiels qui peuvent être réutilisés et intégrés au sein d'un processus de développement. Notons cependant, que, comme montré dans [36], les résultats de ces méthodes d'analyse d'un domaine sont en général insuffisants pour complètement caractériser un domaine et doivent souvent être complétés par l'analyse des familles de programmes/ applications spécifiques au domaine.

1.2.4.2- Analyse d'une famille de programmes

Lors de la définition d'un langage dédié, les méthodes d'analyse d'un domaine, comme celles présentées dans la section précédente, doivent être complétées par l'analyse des familles de programmes existants dans un domaine. Dans le paragraphe suivant, nous exposons quelques techniques qui analysent les points communs d'un domaine et leurs différences, à partir de l'étude des familles de programmes d'un langage.

1.2.4.3- Famille de programmes

Il y a deux décennies, Parnas [36] définissait une famille de programmes par :

« *Nous considérons qu'un ensemble de programmes constitue une famille, lorsqu'il est utile d'étudier les programmes de cet ensemble en regardant d'abord leurs propriétés communes puis en déterminant les particularités de chaque membre de la famille* ».

La définition d'une famille de programmes ressemble à celle d'un domaine donnée par Neighbors [32], présentée ci-dessus. La différence entre la définition d'un domaine et celle d'une famille de programmes est qu'une famille couvre un ensemble de programmes plus restreint, pour lequel il est aussi utile d'étudier les particularités de chaque membre de l'ensemble.

Parnas [36] propose également une méthode pour construire une *famille de programmes*, qui permet de réutiliser les points communs des membres d'une famille. Sa méthode est basée sur le raffinement successif : des programmes incomplets sont progressivement raffinés pour obtenir des programmes complets. D'après Parnas, la méthode permet de prendre un maximum de décisions de conception communes à une famille entière avant de considérer les éléments qui différencient ses membres [36].

Dans le même but de définir des familles de programmes, on peut citer la méthode *FAST (Family-Oriented Abstraction, Specification, and Translation)* [37] [38] [39]. FAST est un processus de génie logiciel qui permet de définir des familles et de développer des environnements pour générer les membres des familles. Dans FAST, on définit les points communs et les particularités de chaque programme de la famille. Cette méthode est composée de trois étapes qui servent à identifier la terminologie dans le cadre de la famille, les points communs entre les différents membres et les variations

existantes.

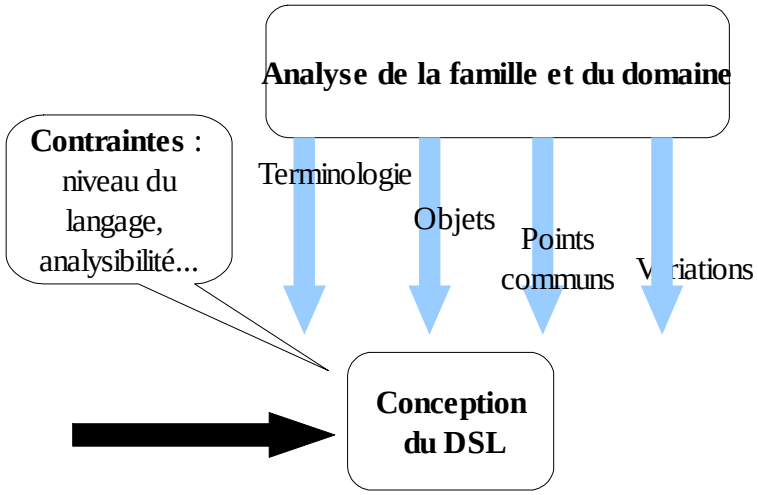

Figure 1: Conception d'un DSL (adaptée de [40])

A partir de ces éléments, [40] définit un processus de conception d'un langage dédié comme illustré dans la figure 1.

D'autres techniques et méthodes pour définir un DSL dans un domaine ont été proposées, comme par exemple la technique de Martin Fowler [30] basée sur le langage Workbench.

1.2.4.4- Famille de langages dédiés

Après avoir présenté différentes stratégies d'analyse d'un domaine métier et d'une famille de programmes, nous définissons ici la notion de famille de langages dédiés à un domaine métier. La notion de famille de langages dédiés sera largement utilisée dans le reste de ce mémoire.

Une famille de langages dédiés est un ensemble de langages dédiés au même domaine métier ; ces langages offrent les mêmes fonctionnalités, au même niveau d'abstraction. Malgré les équivalences existantes entre certains concepts de ces langages, ils sont distincts au niveau de la syntaxe et de la sémantique. Ces différences ont des raisons différentes, principalement de nature stratégique, économique ou politique, qui font que les différents langages proviennent souvent d'entreprises concurrentes. Les différences au sein des différents langages d'une famille génèrent des problèmes d'interopérabilité, que nous allons essayer de résoudre dans les chapitres suivants de ce mémoire.

Un exemple de famille de langages dédiés est celui des langages utilisés pour la gestion d'une base de données, comme le langage SQL [25] SQLplus [41], iSQL [42] et d'autres. Les différentes applications écrites en utilisant ces langages permettent de manipuler des tableaux de données, de stocker et de structurer ces informations, etc. Les applications spécifiées avec un de ces langages servent à atteindre le même résultat. Il existe néanmoins des différences syntaxiques et sémantiques entre ces langages, qui font qu'on a besoin de mécanismes spécifiques (le plus souvent des traducteurs) afin de pouvoir les utiliser ensemble.

La difficulté évidente qui résulte de l'existence de familles de langages est la babélisation générée par la diversité des langages d'une même famille. En fonction du domaine, les tentatives visant à imposer un langage universel ont échoué tant pour des raisons économiques que politiques. C'est le cas des langages définis pour la gestion des procédures opérationnelles, que nous allons utiliser comme base d'exemple du chapitre 3.

Pour une famille de DSL donnée, le défi d'interopérabilité revient à identifier les meilleures techniques qui permettront d'utiliser conjointement et de manière cohérente ses différents membres, tout en continuant à exploiter les environnements et les outils spécifiques à chacun.

1.3-Interopérabilité des logiciels

Nous détaillons dans la partie suivante quelques techniques employées pour résoudre le problème de l'interopérabilité de langages de programmation. En effet, diverses stratégies ont été développées afin de pouvoir utiliser conjointement des langages différents.

Parmi ces stratégies, nous allons présenter celle basée sur l'échange de messages entre deux programmes, une autre basée sur l'utilisation d'une couche ou d'un langage intermédiaire, une basée sur l'utilisation des langages multi-paradigmes, ainsi qu'une stratégie fondée sur la traduction vers un langage intermédiaire.

1.3.1-Communication par messages

Une méthode simple et assez utilisée pour faire interopérer deux applications se base sur l'échange de messages.

Cette solution inclut, évidement, l'utilisation d'un format d'encodage des données communes, afin de permettre une homogénéité au niveau des représentations des données. Ceci est possible en faisant appel à des standards de représentation des données, tel que l'ASN.1 [43].

Pour utiliser cette méthode afin de résoudre le problème d'interopérabilité, il faut mettre au point en même temps que les mécanismes de communication, des canaux de communication. Plusieurs sortes de canaux peuvent être utilisés selon le contexte et l'environnement de travail.

XML permet aussi de spécifier le format d'encodage ou la structure des informations utilisées dans l'échange des messages.

1.3.2-Communication via un logiciel médiateur

Une autre approche pour faire interopérer des logiciels se base sur l'idée d'un logiciel intermédiaire qui prend en charge la

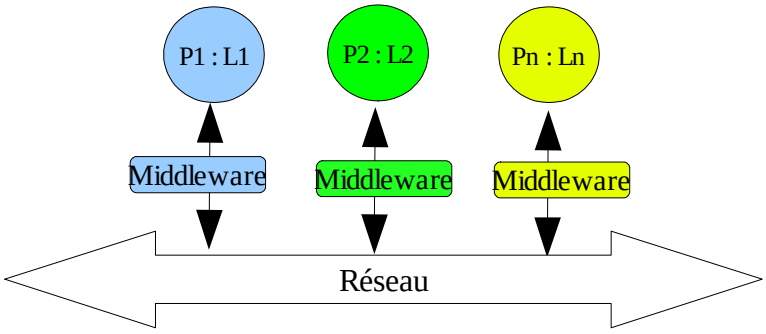

Figure 2: Logiciel médiateur à plusieurs logiciels

communication entre les différents logiciels. La figure 2 illustre la place du médiateur pour des programmes (P) développés en différents langages (L). Ce logiciel est connu sous le nom de *middleware (intergiciel).*

Un logiciel médiateur possède des fonctionnalités spécifiques au besoin d'interopérabilité : il synchronise la communication des données, permet de stocker des informations et même d'exécuter des services utiles pour les logiciels interopérants. Un SGBD « Système de Gestion de Bases de Données» [44] peut être considéré comme un logiciel médiateur entre plusieurs programmes qui peuvent accéder ensemble aux informations des bases de données [45]. En effet, dans un système d'information, plusieurs logiciels ont accès à une même base, et cet accès multiple est géré par le SGBD, malgré la diversité de

ces logiciels. On peut ainsi considérer que le SGBD est un logiciel médiateur.

L'utilisation d'un logiciel médiateur exige l'existence d'un modèle de données commun aux données et aux services de communication entre les logiciels.

Il existe plusieurs exemples de logiciels médiateurs utilisés dans l'informatique, tels CORBA [9] et COM [46].

Par exemple, CORBA propose un modèle ouvert générique pour réaliser l'interopérabilité, pour lequel plusieurs implémentations sont disponibles. Du fait de sa grande généralité, son langage intermédiaire est très complexe, afin de permettre l'intégration de différents langages. Ainsi, une implémentation dans un objet ORB (*Object Request Broker*) dans le modèle de CORBA inclut plusieurs services qui ne sont pas utilisés dans la même application.

Du fait de l'existence de différentes implémentations de CORBA, se pose la question de leur interopérabilité.

On note aussi que la conversion de données selon le modèle d'un langage de description d'interface (LDI) est une étape difficile puisque le modèle de données LDI manque souvent de sémantique claire par rapport aux langages de programmation [47].

1.3.3-Traduction dans un langage intermédiaire

Cette approche consiste à résoudre le problème de l'interopérabilité via la traduction vers un même langage. Les spécifications considérées peuvent être écrites en utilisant un nouveau langage de programmation, ou ce dernier peut être l'un des langages que l'on veut faire interopérer.

Une fois que les différentes parties sont réécrites dans un même langage, cette approche ressemble à la construction d'une application modulaire spécifiée avec un seul langage de programmation, où chaque modèle est traduit dans un code assembleur qui est intégré par l'éditeur de liens d'une application exécutable. La différence avec cette approche est que le rôle de l'assembleur est joué ici par une machine abstraite ou un langage de haut niveau et non pas par un vrai langage d'assemblage.

Par exemple le langage CIL (*Common Intermediate Language)* [48] de Microsoft est un langage intermédiaire conçu pour être partagé par tous les compilateurs basés sur .NET [49]. Un autre exemple est le

langage JVM (*Java Virtual Machine Language)* [49] utilisé par les compilateurs ayant pour cible la machine virtuelle Java.

D'autres approches pour résoudre le problème de l'intéropérabilité, fixent un modèle de données commun qui sera utilisé par les différents composants indépendamment de leur langage de base. Une telle approche est mise en oeuvre par la plateforme .NET [48]. Notons que dans ce cas, lors de l'ajout d'un composant utilisant un nouveau langage de base, il faudra s'assurer qu'il soit compatible avec le modèle de base utilisé, sinon il faudra prévoir des traducteurs. L'inconvénient de cette approche est qu'il est difficile d'intégrer un langage sans lien avec la plateforme.

Pour conclure, l'approche basée sur l'utilisation d'un modèle de données commun offre une solution au problème de l'interopérabilité des composants hétérogènes.

1.3.4-Langage multi-paradigme

Les langages de programmation peuvent être classés selon la notion de paradigme [50] [51]. Un paradigme caractérise un style de programmation et traite de la manière dont les solutions aux problèmes sont formulées. Un paradigme de programmation fournit et détermine la vue qu'à le développeur de son problème et des solutions possibles. Voici quelques exemples de paradigmes en informatique :

- paradigme de la programmation structurée – visant à structurer les programmes impératifs ;
- paradigme de la programmation par objets – où les programmeurs voient un programme comme un ensemble d'objets communiquants ;
- paradigme de la programmation fonctionnelle – où un programme est vu comme un ensemble d'évaluations fonctionnelles sans états ;
- paradigme de la programmation en logique – consistant à exprimer les problèmes et les algorithmes en termes de prédicats.

Notons qu'un même langage de programmation peut être utilisable avec des paradigmes différents. Un tel langage s'appelle langage multi-paradigme. C'est le cas par exemple de C++ [17]. En effet, le langage C++ peut être utilisé dans une programmation structurée, dans une programmation par objets et dans une programmation générique.

Similairement, le langage Python [52] supporte deux paradigmes de programmation : impératif et par objets.

T. Budd [51] introduit la notion de programmation multi-paradigme. Il s'agit d'envisager un cadre permettant de faire travailler ensemble des applications spécifiées dans des paradigmes différents.

Les langages multi-paradigmes offrent une solution au besoin généré par la programmation multi-paradigme. En effet, cette solution consiste à utiliser, ou concevoir, des langages de programmation qui peuvent supporter plusieurs paradigmes de programmation et diverses formes de raisonnement. Dans les paragraphes suivants, nous considérons l'exemple des langages multi-paradigmes Leda [51] et OZ [53]. Il s'agit d'exemples de langages offrant un bon support pour l'utilisation de plusieurs paradigmes [4].

Selon son auteur, l'objectif du langage Leda [51] est d'étudier dans quelle mesure on peut mixer plusieurs paradigmes au sein d'un seul langage. Le résultat est plutôt positif, car Leda supporte à la fois les paradigmes impératif, par objets, logique et fonctionnel, comme on le montre dans [4].

Un autre exemple de langage multi-paradigme est le langage OZ [53], qui inclut des concepts correspondant à la plupart des paradigmes de programmation : impérative, orienté objets, logique, fonctionnel, distribué et concurrent. En ce qui concerne le support pour la programmation, comme souligné en [4], OZ présente comme inconvénient le fait que le développeur doit raisonner, théoriquement, selon chaque paradigme via une traduction dans un modèle général.

Lors de l'utilisation d'un tel langage, certains problèmes sont faciles à résoudre comme celui de la conversion de données grâce à l'unicité du cadre de programmation. Par contre, du fait que ces langages ne permettent pas d'intégrer de nouveau langage ou de réutiliser des applications écrites dans un autre langage, cette solution reste limitée. Notons aussi que l'implémentation de ces langages est moins efficace que celle des langages plus classiques comme C ou C++.

1.3.5-Interopérabilité à travers un méta-langage

En partant du constat que les différents composants écrits dans des langages différents n'ont aucune visibilité réciproque, Gybels [54] propose une approche basée sur l'existence d'un méta-langage commun aux différents langages. Ce méta-langage permet aux différents langages d'avoir une certaine visibilité réciproque au niveau

des concepts.

La méthode proposée par Gybels [54] vise la « *symbiose linguistique* » (c'est le terme que les auteurs utilisent pour l'interopérabilité de composants décrits en utilisant des langages différents) et consiste à utiliser un langage comme Agora [55], afin de permettre l'utilisation de langages différents. Les auteurs proposent de construire dans chacun des deux langages une image réduite de l'autre langage. L'image d'un objet représente un modèle qu'on peut voir comme une interface permettant d'exécuter toutes les opérations autorisées dans l'objet initial avec un changement syntaxique. Ainsi dans [54], les auteurs proposent un étude de cas basée sur l'utilisation conjointe de Java et Smalltalk [56], en passant par Agora.

Dans la même lignée, les auteurs proposent une étude de cas visant l'interopérabilité des langages SOUL [57] et Smalltalk [56], qui appartiennent à des paradigmes de programmation différents.

D'autres travaux, dans le cadre du projet FLINT [58], visent l'interopérabilité et sont basés sur l'idée de traduction des langages utilisés vers un langage intermédiaire unique. Ce langage intermédiaire n'est pas choisi au hasard : il doit être le plus riche possible afin de supporter une grande variété d'objets dans un cadre unique. La difficulté ici revient à trouver un langage de programmation dans lequel l'on peut plonger tous les langages.

1.3.6-Interopérabilité des lignes de produits logiciels

Les lignes de produits logiciels transposent les chaînes de production au monde du logiciel. Une ligne de produit représente un ensemble de systèmes qui partagent des propriétés communes satisfaisant des besoins spécifiques pour le domaine visé. Son objectif principal est d'augmenter la productivité et de diminuer le temps de réalisation des logiciels. Cet objectif peut être atteint en développant simultanément plusieurs logiciels dédiés au même domaine ou des logiciels paramétrés, à la place de les développer séparément.

L'ingénierie des lignes de produits représente une approche qui consiste à regrouper les méthodes, outils et techniques de développement et à maintenir une collection de systèmes logiciels similaires appartenant à un domaine particulier.

Les difficultés liées à cette approche résident dans la conception d’une

architecture permettant de définir plusieurs produits en mettant en évidence leurs points communs, mais aussi leurs différences.

Dans la littérature, [59] et [60] proposent de distinguer dans le développement des lignes de produits relevant de l' « ingénierie du domaine » et de l' « ingénierie de l'application ». L'ingénierie du domaine consiste à développer et construire les « *assets* » qui seront réutilisés pour la construction de produits ; il s'agit de développer pour la réutilisation. L'ingénierie de l'application consiste à utiliser les « *assets* » pour la construction d'un produit ou d'une application particulière après cristallisation des points de variation ; il s'agit d'un développement par la réutilisation.

Dans la littérature, le développement des lignes de produits se fait en utilisant des techniques différentes :

- Les techniques de compilation qui permettent la dérivation d'un produit pendant la phase de compilation. Des exemples de ces techniques sont la compilation conditionnelle et le chargement de bibliothèques.
- Les techniques liées aux langages de programmation : les langages à objets ont apporté quelques techniques utiles pour implémenter la variabilité, comme par exemples le polymorphisme, la surcharge et la liaison dynamique.
- Des approches de programmation : des approches récentes de génie logiciel peuvent être utilisées pour implémenter et gérer la variabilité dans les systèmes. [61] [62] [63] proposent d'utiliser les aspects [64] pour la gestion de la variabilité dans les lignes de produits.
- Des approches basées sur la modélisation des lignes de produits en UML [65], ou en s'appuyant sur les mécanismes d'extension d'UML [59] [66].

Il faut noter dans ce contexte l'initiative de standardisation lancée par l'OMG en septembre 2009, qui vise à standardiser un langage commun pour spécifier la variabilité (CVL : *Common Variability Language*) [67]. Ce langage de modélisation doit avoir les mécanismes de gestion de la variabilité, permettant de gérer la variabilité au sein des lignes de produits et des langages dédiés proches.

Actuellement, le groupe de travail constitué afin de répondre à cet RFP travaille sur la spécification de la variabilité en utilisant un modèle complet des fonctionnalités attendues [68] [69] effectué sur la base des *features diagrams* [70].

Dans [71], une étude montre l’automatisation de procédures de génération du code à partir d'un langage de contrôle de la gestion des trains (TCL : *Train Control Language*), en utilisant le même langage de spécification de la variabilité que [69].

Pour conclure ce chapitre, nous avons vu qu'il existe un nombre important de techniques et approches qui essaient de résoudre le problème de l'interopérabilité. Dans le reste de cette thèse, nous nous situons dans la même lignée en essayant de trouver une solution au problème de l'interopérabilité au sein d'une famille de langages dédiés. Comme nous allons le voir, nous nous concentrons sur les techniques qui permettent de continuer d'utiliser les langages d'origine, en essayant aussi de nous concentrer sur l'unification des langages appartenant à la famille.

Chapitre II

2- Sémantique des langages et spécification algébrique

La syntaxe d'un langage de programmation décrit ses expressions valides sans en préciser le sens. Le mot sémantique qui est relatif au sens (du grec *sêmantikos*) fournit une signification à cette description textuelle. Étant donné que la syntaxe est un domaine bien défini et bien formalisé qui ne s'intéresse qu'aux propriétés structurelles et grammaticales du langage, une sémantique est nécessaire pour savoir comment exploiter le langage, quel type de problème il peut résoudre et comment un programme peut répondre aux exigences d'une application. Tous ces points relatifs à la programmation proprement dite ne sont perceptibles qu'au travers de la définition d'une sémantique.

La sémantique d'un langage de programmation est composée de la sémantique statique et de la sémantique dynamique. La sémantique statique s'intéresse au respect des règles grammaticales non directement exprimables par la grammaire comme le contrôle du bon usage des types ou les règles de transmission des paramètres à un sous-programme ; elle ne fait référence qu'au texte source du programme et se détermine à la compilation. La sémantique dynamique représente la partie la plus significative dans le sens où elle se rattache directement à l'exécution de ce même texte source en donnant une signification à chacune de ses instructions ou en précisant la valeur retournée par l'évaluation d'une expression à partir de données d'entrée. Pour un langage de programmation donné, la sémantique permet de relier les règles syntaxiques aux règles comportementales.

La sémantique d'un langage de programmation peut s'écrire sous différentes formes parmi lesquelles quatre approches sont généralement exploitées [72] : il s'agit des sémantiques opérationnelle, dénotationnelle, axiomatique et algébrique. La sémantique opérationnelle ne s'intéresse pas seulement au résultat de l'exécution, mais aussi à la façon dont s'exécute le programme. La sémantique dénotationnelle traduit les constructions syntaxiques du langage en fonctions mathématiques au travers de dénotations, alors que la sémantique axiomatique vérifie les propriétés du programme sur la base d'assertions avant et après l'exécution d'une instruction. La

sémantique algébrique qui découle de travaux sur les types abstraits algébriques, traduit les non terminaux d'une grammaire en types et fournit les équations sémantiques qui associent à un programme un terme conforme au type abstrait décrivant le langage ; ce mode de spécification a l'avantage de disposer d'outils et de se prêter à la définition de propriétés pouvant être démontrées rigoureusement.

2.1- Les différentes sémantiques

Afin d'illustrer les différentes sémantiques auxquelles nous nous sommes intéressés, nous considérons un même langage impératif, le langage *L*. Ce langage est doté de structures de contrôle conventionnelles (séquence, sélection et répétition) et se caractérise par une instruction de mise à jour de ses variables (affectation). Ses règles syntaxiques au format EBNF sont fournies à la figure 3 :

```
programme ::= "programme" nom_de_programme glossaire? instructions
glossaire ::= "glossaire" ( déclaration_de_variable ";" )+ ";"
déclaration_de_variable ::= nom_de_variable "<" type ">"
type ::= "Entier" | "Booléen"
instructions ::= "début" instruction+ "fin"
instruction ::= séquence | sélection | répétition | affectation
séquence ::= instruction+
sélection ::= "si" expression "alors" instruction
             ( "sinon" instruction )? "fin si" ";"
répétition ::= "tantque" expression "faire" instruction
             "fin tantque" ";"
affectation ::= variable "<-" expression ";"
expression ::= opérande | opération | "(" expression ")"
opération ::= opérande opérateur opérande
opérande ::= constante | nom_de_variable
opérateur ::= "+" | "-" | "*" | "/" | "=" | "/=" |
              "<" | ">" | "<=" | ">=" |  "et" | "ou"
```

Figure 3: Syntaxe EBNF du langage L

Dans tout ce qui suit, et afin de comparer les différents formalismes, nous posons :

E : expression,
G : glossaire,
I : instruction,
N : identificateur de programme,
V : variable.

2.1.1-La sémantique opérationnelle

La sémantique opérationnelle d'un langage de programmation décrit la signification d'un programme valide du langage en exécutant ses instructions en termes de séquences d'états sur une machine réelle ou simulée [73]. En d'autres termes, la sémantique opérationnelle associe à un programme la description d'un système de transitions d'états indépendant de l'architecture de la machine cible et de la stratégie d'implémentation.

Pour une instruction donnée, la sémantique opérationnelle établit une relation entre l'état obtenu avant et après exécution de l'instruction. Soient *env* et *env'* deux états et *I* une instruction telle que $< I / env > \rightarrow env'$, la sémantique opérationnelle exprime une relation de transition entre l'état *env* avant l'exécution de *I* et l'état *env'* après avoir exécuté *I*. Il peut s'agir d'un état *env'* dans le cas où l'exécution est achevée, ou $< I' / env' >$, un autre état *env'* avec une autre instruction *I'* sinon.

Pour écrire la sémantique opérationnelle d'un langage, il faut disposer d'une machine réelle ou simulée pour définir les effets d'un programme sur cette machine. Le passage vers cette machine se fait via un interprète du langage qui réagit aux changements de code. Le procédé revient donc à proposer une implémentation du langage.

Nous donnons ci-dessous la sémantique opérationnelle du langage *L* :

Notations :

< P / env > : état de l'interprète
P : programme à exécuter
env : environnement (liste de triplets (Variable, Type, Valeur))

$\perp$: valeur indéfinie

Opérations :

insérer : Variable x Type x Valeur x Environnement → Environnement
modifier : Variable x Type x Valeur x Environnement → Environnement
estTypeEntier : Variable x Environnement → Booléen
estTypeBooléen : Variable x Environnement → Booléen
expressionEntière : Expression x Environnement → Booléen
expressionBooléenne : Expression x Environnement → Booléen
évalExpression : Expression x Environnement → Valeur

Transitions :

programme

< **programme** N **glossaire** G **début** I **fin** / env > →
{ < G / env > ; < I / env > }

< **programme** N **début** I **fin** / env > →
{ < I / env > }

glossaire

< V <Entier> ; G / env > →
{ < G / insérer (V, entier, ⊥, env) > }

< V <Booléen> ; G / env > →
{ < G / insérer (V, booléen, ⊥ , env) > }

séquence

< I1 ; I2 / env > →
{ < I1 / env > ; < I2 / env > }

sélection

< **si** C **alors** I1 ; **sinon** I2 ; **fin si** ; I / env > →
{ **si** expressionBooléenne (C, env) **alors**
soit val = évalExpression (C, env)
si val /= ⊥ **alors**
si val **alors**
< I1 ; I / env >
sinon
< I2 ; I / env >
fin si

sinon
< **erreur** (« valeur indéfinie ») ; I / env >
sinon
< **erreur** (« erreur de type ») ; I / env >
fin si }

< **si** C **alors** I1 ; **fin si** ; I / env > →
{ **si** expressionBooléenne (C, env) **alors**
soit val = évalExpression (C, env)
si val /= ⊥ **alors**
si val **alors**
< I1 ; I / env >
sinon
< I / env >
fin si
sinon
< **erreur** (« valeur indéfinie ») ; I / env >
fin si
sinon
< **erreur** (« erreur de type ») ; I / env >
fin si }

<u>répétition</u>

< **tantque** C **condition faire** I1 ; **fin tantque** ; I / env > →
{ **si** expressionBooléenne (C, env) **alors**
soit val = évalExpression (C, env)
si val /= ⊥ **alors**
si val **alors**
< I1 ; **tantque** C **faire** I1 ; **fin tantque** ; I / env
>
sinon
< I / env >
fin si
sinon
< **erreur** (« valeur indéfinie ») ; I / env >
fin si
sinon
< **erreur** (« erreur de type ») ; I / env >

fin si }

affectation

< V <- E ; I / env > →
{ **soit** val = évalExpression (E, env)
si val /= ⊥ **alors**
si estTypeEntier (V, env) **alors**
si expressionEntière (E, env) **alors**
< I / modifier (V, entier, val, env) >
sinon
< **erreur** (« erreur de type ») ; I / env >
fin si
sinon
si estTypeBooléen (V, env) **alors**
si expressionBooléenne (E, env) **alors**
< I / modifier (V, booléen, val, env) >
sinon
< **erreur** (« erreur de type ») ; I / env >
fin si
sinon
< **erreur** (« variable non déclarée ») ; I / env >
fin si
fin si
sinon
< **erreur** (« valeur indéfinie ») ; I / env >
fin si }

La sémantique opérationnelle facilite la construction de l'interprète ou du compilateur et est intuitive. Néanmoins, elle n'est pas très formelle. De plus, on ne peut pas prouver que deux programmes sont équivalents d'où la difficulté de construire un compilateur optimisé.

Soient par exemple les deux programmes ci-dessous *P1* et *P2* équivalents en termes de résultats produits à l'issue de l'exécution, formés des mêmes instructions, mais dans un ordre différent :

```
x <- x + 1 ; y <- y + 2 ;
```

```
y <- y + 2 ; x <- x + 1 ;
```

Ces deux programmes conduisent au même résultat. Selon la sémantique opérationnelle, ils peuvent ne pas être équivalents au plan intentionnel car les systèmes de transition diffèrent. Ainsi, l'état *env'*, résultat de l'exécution < *x* <- *x + 1*, *env* > de *P1* est différent de l'état *env''*, résultat de l'exécution < *y* <- *y + 2*, *env* > de *P2*.

```
Liste = spec
type a
type Liste a = | Nil | Cons a Liste a
```

En résumé, la sémantique opérationnelle revient à fournir une méthode d'implémentation pour le langage. Il est ainsi parfois difficile de définir rigoureusement la machine abstraite supportant les transitions d'états. De plus, la mise en oeuvre d'un interprète induit un jeu de données pour connaître un résultat d'exécution. On notera que le formalisme ne permet pas naturellement la mise en place d'un système de preuve des programmes.

2.1.2-La sémantique dénotationnelle

La sémantique dénotationnelle a été proposée par Christopher Strachey et Dana Scott [74] [75] en 1970. Elle est plus structurée que la sémantique opérationnelle [76] et reste la sémantique la plus riche. Son principe associe à chaque entité du programme un objet mathématique, le plus souvent une fonction du λ-calcul, appelée dénotation qui attache les instances de ces entités à l'objet mathématique [77]. Une fonction prendra, par exemple, l'état de la mémoire en tant qu'attribut.

Définir la sémantique dénotationnelle d'un langage de programmation consiste à définir les objets dénotationnels ; elle est compositionnelle car elle définit un programme en utilisant les définitions qui le constituent. La difficulté que l'on rencontre le plus souvent pour développer ce type de sémantique est d'identifier ces objets et les fonctions pour ces objets, mais aussi de créer des correspondances entre eux.

Un exemple dans le contexte des expressions arithmétiques est donné par la fonction *A* ci-dessous :

$$A[n] = \lambda\sigma.n$$
$$A[x] = \lambda\sigma.\text{flookup}\ (\sigma, x)$$
$$A[\text{plus}\ (x, y)] = \lambda\sigma.\ A[x](\sigma) + A[y](\sigma)$$

La fonction *A* a pour profil *Expression* → *Environnement* → *Z*. Elle considère donc une expression et un environnement et restitue une valeur sur *Z*. Cette notation est connue en sémantique, par exemple sous la forme de l'écriture $\lambda\sigma.n$ qui signifie que l'on associe un environnement σ à une expression *n*.

La fonction *flookup*, fonction partielle sur les variables de profil *Environnement* → *Variables* → *Z* présente la valeur de la variable *x* dans l'environnement σ.

Les valeurs du domaine sont quelquefois indéterminées comme dans le cas d'un calcul. Ce qui conduit à définir les domaines avec un ordre partiel où la relation d'ordre est définie par « moins défini que ». A celà s'ajoute un élément indéfini noté $\perp$ et, en lieu et place de définir un ensemble *A*, on introduit $A_\perp$ qui signifie $A \cup \{\perp\}$ [78]. Ainsi la fonction *flookup* devient une fonction totale de profil *Environnement* → *Variables* → $Z_\perp$ et la fonction dénotationnelle devient :

$$A[n] = \lambda\sigma.n$$
$$A[x] = \lambda\sigma.\text{flookup}(\sigma, x)$$
$$A[\text{plus } (x, y)] = \lambda\sigma.\ \text{si } A[x](\sigma) = \perp \text{ ou } A[y](\sigma) = \perp \text{ alors } \perp \text{ sinon } A[x](\sigma) + A[y](\sigma)$$

La sémantique dénotationnelle du langage *L* s'exprime par :

Notations :

Fonction S : sémantique d'une instruction
Fonction D : sémantique d'une déclaration
Fonction E : sémantique d'une expression

$\perp$: valeur indéfinie

Opérations :

S : Instruction x Environnement x Mémoire → Mémoire
D : Déclaration x Environnement → Environnement
E : Expression x Environnement x Mémoire → Valeur

typeVariable : Variable → Type
typeExpression : Expression x Environnement → Type
allouer : Environnement → Adresse
e : Variable → Adresse

m : Adresse → Valeur

Conventions :

e [x / y] : Variable → Adresse
fournit l'adresse y si la variable est x, sinon fournit ce que fournit e
m [x / y] : Adresse → Valeur
fournit la valeur y si l'adresse est x, sinon fournit ce que fournit m

Dénotations :

programme

S (**programme** N **glossaire** G **début** I **fin**, env, mem) = S (I, env', mem)
où env' = D (G, env)

S (**programme** N **début** I **fin**, env, mem) = S (I, env, mem)

variables

D (V <Entier> ; G, env) = D (G, env')
où env' = e [V / allouer (env)]

D (V <Booléen> ; G, env) = D (G, env')
où env' = e [V / allouer (env)]

séquence

S (I1 ; I2, env, mem) = S (I2, env, mem')
où mem' = S (I1, env, mem)

sélection

```
S (si C alors I1 ; sinon I2 ; fin si, env, mem) =
        si typeExpression (C, env) = Booléen alors
                si v /= ⊥ alors
                        si v alors
                                S (I1, env, mem)
                        sinon
                                S (I2, env, mem)
                        fin si
                sinon
                        ⊥
                fin si
```

sinon
⊥
fin si
où v = E (C, env, mem)

S (**si** C **alors** I ; **fin si**, env, mem) =
si typeExpression (C, env) = Booléen **alors**
si v /= ⊥ **alors**
si v **alors**
S (I, env, mem)
sinon
mem
fin si
sinon
⊥
fin si
sinon
⊥
fin si
où v = E (C, env, mem)

<u>répétition</u>

S (**tantque** C **faire** I ; **fin tantque**, env, mem) =
si typeExpression (C, env) = Booléen **alors**
si v /= ⊥ **alors**
si v **alors**
S (**tantque** C **faire** I ; **fin tantque**, env, mem')
où mem' = S (I, env, mem)
sinon
mem
fin si
sinon
⊥
fin si
sinon
⊥
fin si
où v = E (C, env, mem)

<u>affectation</u>

S (V <- E, env, mem) =
si v /= ⊥ **alors**

si typeVariable (V) = Entier **alors**
si typeExpression (E, env) = Entier **alors**
m [e (V) / E (E, env, mem)]
sinon
⊥
fin si
sinon
si typeVariable (V) = booléen **alors**
si typeExpression (E, env) = booléen **alors**
m [e (V) / E (E, env, mem)]
sinon
⊥
fin si
sinon
⊥
fin si
fin si
sinon
⊥
fin si
où v = E (C, env, mem)

Contrairement à la notation opérationnelle, cette sémantique n'impose pas de contrainte quant à l'implémentation du langage car une dénotation incarne un mode de calcul ne spécifiant pas la manière dont il sera traduit en machine.

La dénotation d'une construction du langage peut cependant s'avérer relativement lourde à écrire. Pour obtenir le résultat d'une exécution, on exploite le plus souvent le caractère applicatif des fonctions en fournissant l'environnement de calcul initial à la fonction associée au programme.

On notera que la représentation de la valeur sémantique d'un programme par une λ-expression est parfois considérée moins riche qu'une représentation par le biais d'une machine abstraite, car elle ne fournit qu'une vue globale de la transformation des données en résultats, alors qu'une machine abstraite marque les différentes étapes de transformation [79].

2.1.3-La sémantique axiomatique

La spécification axiomatique fait référence aux travaux de C.A.R. Hoare [80] et E.W. Dijkstra [81]. Il s’agit d’une approche

basée sur la logique mathématique définie par une méthode de preuve de la validité d'un programme. La sémantique axiomatique considère un programme comme un transformateur de propriétés logiques. Elle s'attache à démontrer qu'un programme est valide ; la preuve de validité existe si le programme est correct. Cette preuve consiste à associer à chaque construction du programme, que l'on peut considérer comme des « commandes du langage », deux attributs : une proposition appelée pré-condition la précède et une autre appelée post-condition la succède [82].

L'enjeu est de trouver la sémantique axiomatique la plus fine possible afin d'avoir le résultat le plus fiable à la sortie du programme. La pré-condition doit être la moins restrictive possible, car elle garantit, pour construire la preuve du programme, la validité de la post-condition en « remontant » de la fin du programme vers son début. La sémantique axiomatique ne constitue pas le meilleur moyen pour définir la sémantique d'un langage de programmation, car elle est relativement complexe et qu'elle connaît des limites, notamment pour spécifier certaines instructions comme le branchement inconditionnel *aller à* (*go to).*

Les propriétés de la sémantique axiomatique sont en général formalisées par la logique d'Hoare [80]. Il s'agit de triplets de la forme *{p} I {q}* où *p* et *q* sont des propriétés logiques sur le programme : *p* représente la pré-condition et *q* la post-condition, alors que *I* est une commande du programme, comme par exemple l'instruction d'affectation. L'assertion *p*, qui doit être vérifiée avant l'exécution de l'instruction *I*, conduit à l'assertion *q* qui devra être vérifiée après l'exécution de *I*. Soit *x <- a* une affectation et *q* sa post-condition, alors sa pré-condition *p* est calculée par l'axiome *{ p = q [X/a] }* qui exprime que *p* est calculée de la même façon que *q* en remplaçant *X* par *a* dans la commande.

Nous décrivons dans ce qui suit la sémantique axiomatique du langage *L*. Aux assertions s'ajoutent des règles de déduction telles que chacune transforme une conjonction de prémisses *p1, p2, ..., pn* en une conclusion *p*. Comme précédemment, les énoncés sémantiques sont précédés de la notation nécessaire à leur compréhension.

Notations :

axiome : { p } I { q }

règle de déduction : $\frac{p1, p2, \ldots, pn}{q}$

substitution de y à x dans le prédicat p : p[x/y]

Règles de déduction :

programme

$$\frac{\{ p \}\ G\ \{ q \},\ \{ q \}\ I\ \{ r \}}{\{ p \}\ \textbf{programme}\ N\ \textbf{glossaire}\ G\ \textbf{début}\ I\ \textbf{fin}\ \{ r \}}$$

$$\frac{\{ p \}\ I\ \{ q \}}{\{ p \}\ \textbf{programme}\ N\ \textbf{début}\ I\ \textbf{fin}\ \{ q \}}$$

glossaire

$$\frac{\{ p[V/V'] \}\ G\ \{ q[V/V'] \}}{\{ p \}\ V\ \langle \text{Entier} \rangle\ ;\ G\ ;\ \{ q \}}$$

$$\frac{\{ p[V/V'] \}\ G\ \{ q[V/V'] \}}{\{ p \}\ V\ \langle \text{Booléen} \rangle\ ;\ G\ ;\ \{ q \}}$$

séquence

$$\frac{\{ p \}\ I1\ \{ q \},\ \{ q \}\ I2\ \{ r \}}{\{ p \}\ I1\ ;\ I2\ ;\ \{ r \}}$$

sélection

$$\frac{\{ p \text{ et } C \}\ I1\ \{ q \},\ \{ p \text{ et non } C \}\ I2\ \{ q \}}{\{ p \}\ \textbf{si}\ C\ \textbf{alors}\ I1\ ;\ \textbf{sinon}\ I2\ ;\ \textbf{fin si}\ ;\ \{ q \}}$$

$$\{ p \text{ et } C \}\ I\ \{ q \},\ \{ p \Rightarrow q \}$$

$$\frac{}{\{ p \}\ \textbf{si}\ C\ \textbf{alors}\ I\ ;\ \textbf{fin si}\ ;\ \{ q \}}$$

répétition

$$\frac{\{ p \text{ et } C \}\ I\ \{ p \},\ \{ p \text{ et non } C \Rightarrow q \}}{\{ p \}\ \textbf{tantque}\ C\ \textbf{faire}\ I\ ;\ \textbf{fin tantque}\ ;\ \{ q \}}$$

affectation

$$\{ p[V / E] \}\ V \leftarrow E\ ;\ \{ p \}$$

Cette approche a été largement exploitée entre autres pour la preuve de programmes. Par exemple, la règle de déduction de la répétition met en évidence un invariant définissant intrinsèquement le calcul réalisé par la boucle. L'approche permet donc de s'abstraire des détails d'implémentation du langage dans le sens où elle ne s'attarde que sur les transformations des prédicats associées aux constructions du langage.

Dès lors, la définition de la sémantique n'est pas explicite car elle impose de définir ces différentes transformations par la théorie axiomatique. Pour obtenir toutes les propriétés souhaitées du langage, le système formel doit être complet et imposer un grand nombre de conditions ; ce n'est qu'à ce prix qu'il sera en mesure d'exprimer toute la sémantique du langage [79].

2.1.4-La sémantique algébrique

La sémantique algébrique identifie les entités et opérations d'un langage, puis décrit ses propriétés par des axiomes dans une logique donnée [83]. Ce formalisme a été fréquemment exploité pour définir des types abstraits, en établissant via les algèbres, un modèle de la spécification [84]. C'est ainsi qu'un type abstrait algébrique caractérise une structure de données par des propriétés opérationnelles, indépendamment d'une implémentation spécifique, à l'inverse d'une sémantique dénotationnelle qui dépend d'un modèle particulier de mémoire.

La sémantique algébrique d'un langage de programmation est définie à travers une classe de machines abstraites, dans lesquelles on décrit l'effet du programme sur ces machines [85]. Elle collecte un ensemble de sortes d'objets et décrit des opérations sur ces ensembles en rapport

aux constructions du langage ; cet aspect syntaxique est complété par un ensemble d'axiomes donnant la sémantique des opérations.

Pour le langage *L* dont la sémantique algébrique est fournie ci-dessous, cela revient notamment à définir les types abstraits *VariableEntière*, *VariableBooléenne* (qu'on suppose identique au type *VariableEntière* aux noms des sortes près) et *Instruction*, puis à exploiter ces types abstraits au travers d'équations sémantiques décrivant le comportement des constructions de *L*.

Types :

variable entière

type VariableEntière

opérations

varEntière : → VariableEntière
estDéclarée : VariableEntière → Booléen
valeur : VariableEntière → Entier
égal : VariableEntière x VariableEntière → Booléen

préconditions

valeur (v) **ssi** estDéclarée (v)
égal (v1, v2) **ssi** estDéclarée (v1) **et** estDéclarée (v2)

axiomes

non (estDéclarée (varEntière))
égal (v1, v2) = (valeur (v1) = valeur (v2))

fin VariableEntière

instructions

type Instruction

opérations

dclVarEntière : VariableEntière → Instruction
dclVarBooléenne : VariableBooléenne → Instruction
séquence : Instruction x Instruction → Instruction
sélection : Booléen x Instruction x Instruction → Instruction
sélection : Booléen x Instruction → Instruction
répétition : Booléen x Instruction → Instruction
affecterVarEntière : VariableEntière x Entier → Instruction
affecterVarBooléenne : VariableBooléenne x Booléen → Instruction

fin Instruction

Equations :

T_x : termes de type *X*

S : G U I → $T_{Instruction}$
VE : V → $T_{VariableEntière}$
VB : V → $T_{VariableBooléenne}$
E : E → $T_{booléen}$

S (**programme** N **glossaire** G **début** I **fin**) = séquence (S (G), S (I))

S (**programme** N **début** I **fin**) = S (I)

S (V <Entier>) = dclVarEntière (V)

S (V <Booléen>) = dclVarBooléenne (V)

S (I1 ; I2) = séquence (S (I1), S (I2))

S (**si** C **alors** I1 ; **sinon** I2 ; **fin si**) = sélection (S (C), S (I1), S (I2))

S (**si** C **alors** I ; **fin si**) = sélection (S (C), S (I))

S (**tantque** C **faire** I ; **fin tantque**) = répétition (S (C), S (I))

S (V <- E) = affecterVarEntière (VE (V), E (E))

S (V <- E) = affecterVarBooléenne (VB (V), E (E))

Définitions :

éval : Instruction x Environnement → Environnement

éval (dclVarEntière (v), E) = éval (substituer (estDéclarée (v), vrai), E)

éval (dclVarBooléenne (v), E) = éval (substituer (estDéclarée (v), vrai), E)

éval (séquence (i1, i2), E) = éval (i2, éval (i1, E))

éval (sélection (vrai, i1, i2), E) = éval (i1, E)

éval (sélection (faux, i1, i2), E) = éval (i2, E)

éval (sélection (vrai, i), E) = éval (i, E)

éval (répétition (vrai, i), E) = éval (répétition (vrai, i), éval (i, E))

éval (répétition (faux, i), E) = E

éval (affecterVarEntière (v, val), E) = éval (substituer (valeur (v), val), E)

éval (affecterVarBooléenne (v, val), E) = éval (substituer (valeur (v), val), E)

L'abstraction des concepts du langage par le biais de types conduit à une formulation assez systématique de la sémantique. A l'instar de la sémantique dénotationnelle qui associe une formule à un programme, le programme se traduit ici en un terme dont la signification est donnée par les axiomes des types abstraits. Les méthodes dénotationnelle et algébrique se rejoignent sur ce point. Le formalisme algébrique s'en distingue par le cadre formel qu'il propose et sa capacité à déduire de nouvelles propriétés à partir des axiomes et de la logique sous-jacente à la réécriture, équationnelle ici.

2.1.5-Relation entre les sémantiques

Ces quatre formalismes, et d'autres [86], servent à spécifier de façon formelle la sémantique d'un langage de programmation. Elles utilisent des approches distinctes ; néanmoins, elles ne sont pas complètement indépendantes les unes des autres. D’après la théorie de l'interprétation abstraite, on peut classer de manière hiérarchique les sémantiques selon des niveaux d'abstraction différents [87]. En effet, une sémantique *S1* est l'abstraction d'une autre *S2,* si et seulement si deux programmes équivalents en *S2* sont aussi équivalents en *S1*. Ce qu'on peut résumer par la proposition suivante en énonçant pour deux programmes *P1* et *P2* : « *P1 et P2 équivalents syntaxiquement le sont opérationnellement, P1 et P2 équivalents opérationnellement le sont dénotationnellement, P1 et P2 équivalents dénotationellement le sont axiomatiquement.* »

Selon le contexte, l'utilisation d'une sémantique spécifique apporte plus de bénéfices à certaines applications qu’à d’autres. Par exemple, la sémantique dénotationnelle aide à valider les compilateurs, concevoir un langage et joue un rôle dans son apprentissage.

L’utilisation de la sémantique opérationnelle est idéale pour définir un compilateur, et la sémantique axiomatique constitue la référence des traitements à appliquer aux programmes.

Dans cette thèse, nous nous intéressons aux spécifications algébriques fondées sur la logique équationnelle, qui présentent l'avantage de décrire assez simplement et de manière quasi-automatique les spécificités d'un langage de programmation et qui offrent en outre des outils pour le faire rigoureusement. De plus, l'intérêt d'une spécification algébrique réside dans son aspect à la fois dénotationnel au travers des algèbres servant de modèles à la spécification, axiomatique par l'intermédiaire des axiomes associés aux opérations de la spécification et opérationnel lorsqu'on considère ces mêmes axiomes comme des règles de réécriture pour évaluer symboliquement un terme correspondant à un programme [88].

Dans la partie qui suit, nous définissons plus en détail ce qu’est une spécification algébrique, ses utilisations et comment on l'exploite pour spécifier un langage de programmation. Nous décrivons en particulier quelles sont les étapes nécessaires pour traduire un langage de programmation en une spécification algébrique. On essayera de modéliser ce passage par une méthodologie permettant d'obtenir la signature et les axiomes d'une spécification algébrique, à partir d'une syntaxe modélisée par une grammaire au format EBNF.

2.2-Les spécifications algébriques

Spécifier algébriquement correspond une approche formelle qui permet de définir rigoureusement des structures de données, tout comme des programmes ou des fragments de programme, en se basant sur des concepts de logique mathématique [89]. Une spécification algébrique se compose d'une signature et d’axiomes. La signature sert à décrire les termes que l'on utilise et les axiomes spécifient le comportement d'expressions constituées par les termes. Un ensemble de théorèmes peut être déduit des axiomes par la logique équationnelle sous-jacente.

Historiquement, les spécifications algébriques s'inspirent de deux domaines : l'algèbre universelle en mathématique [90] et les types abstraits en génie logiciel [84]. Un type abstrait a pour objectif de formaliser des données au travers de sortes et d'opérations. Il met en jeu un ensemble de domaines de valeurs et des fonctionnalités sur ces domaines, via des opérations. On modélise ainsi un type abstrait par une algèbre multi-sorte, définie sur une signature. Les opérations de la

signature correspondent à celles qui caractérisent le type abstrait.

Des travaux importants sur les spécifications algébriques ont été réalisés par le groupe ADJ (J. Goguen, J. Thatcher, E. Wagner et J. Wright) [89]. Il s'agit d'un groupe du département de recherche d'IBM s'intéressant à l'étude d'algèbres et de méthodes catégoriques en informatique. Dans les années 70, nous pouvons signaler d'autres travaux comme ceux de S. Zilles et J. Guttag [90]. Avant que les spécifications algébriques ne puissent raisonner selon différents types de logique, elles exploitaient essentiellement la logique équationnelle fondée sur une substitution de termes égaux : si deux termes sont égaux, alors on peut les interchanger sans changer la validité de la conclusion. Au fil du temps, elle s'est enrichie d'autres logiques comme celle avec contraintes ou avec conditions [91].

2.2.1-Signature

On définit une signature Σ d'une algèbre par les données suivantes :

● Un ensemble de sortes *S* défini par un ensemble de noms ayant pour objectif de présenter les différents types de données. Cet ensemble fait également référence aux symboles utilisés pour décrire les opérations associées.

● Un ensemble de profils d'opération *O* défini sur l'ensemble *S* tel que, chaque opération possède un domaine et un co-domaine composés chacun de plusieurs sortes appartenant à *S*.

A titre d'exemple, pour les opérations classiques en arithmétique, *O* est un ensemble de $S^* \times S$ indexé, tel que $O = \{O_{w,s} \mid w \in S^*, s \in S\}$ où *w* correspond à l'arité de l'opération et *s* à son ensemble de sortes.

Deux types d'opération sont à distinguer dans une signature : les constructeurs servant à construire des termes de la sorte étudiée et les observateurs s'appliquant à une certaine sorte d'arguments pour retourner une observation d'une autre sorte.

2.2.2-Algèbre

Pour définir une algèbre, il faut disposer d'une signature $\Sigma = (S, O)$. Deux types d'algèbre sont à distinguer selon leur signature : mono-sorte ou homogène dans les cas où les opérateurs de la signature manipulent un seul type et multi-sorte ou hétérogène si l'on introduit

plusieurs types. Des applications en informatique surtout en génie logiciel et d'autres en mathématique, comme ceux sur les espaces vectoriels, exigent la manipulation de plusieurs types de données, ce qui nécessite un mixage de plusieurs sortes pour les opérateurs d'une signature. Dans ce cas précis, on définit une algèbre multi-sorte.

Pour une signature Σ, on définit une Σ-algèbre A par :

● un ensemble $A(s)$ pour chaque $s \in A$,

● une correspondance $A(f) : A(w) \rightarrow A(s)$, pour chaque $f \in O_{w,s}$, $s \in S$ et $w \in S^*$. Pour w défini par $w = s_1. s_2. ... . s_n$, $A(w)= A(s_1) \times A(s_2) \times ... \times A(s_n)$. Si A est la chaîne vide, alors $A(w)= \{\theta\}$.

2.2.3-Spécification algébrique

On définit une spécification algébrique par son ensemble de sortes et de profils d'opération (ce qui constitue une signature Σ) et un ensemble E d'axiomes sur les opérations. Pour une spécification $p = (\Sigma, E)$, on dit qu'une Σ-algèbre A est p-algèbre si A satisfait tous les axiomes de E.

Une spécification algébrique permet de spécifier rigoureusement un problème par l'abstraction de ses fonctionnalités et l'utilisation d'opérations purement fonctionnelles. On utilise la signature pour décrire la partie syntaxique, en représentant les éléments des données par des sortes et les fonctionnalités par des opérations sur les sortes. Pour décrire la sémantique du problème, il faut formaliser les comportements de ces opérations ; c'est le rôle des axiomes.

Plusieurs langages pour décrire des spécifications algébriques ont été définis, comme LARCH [92], OBJ [93], PLUSS [94], ASI [95], Specware [96], etc.

Nous explicitons ci-dessous en Specware quelques exemples de spécification algébrique.

1) *Spécification du type abstrait Couple*

Pour définir la spécification d'un couple, il faut introduire deux types génériques, *a* et *b*, qui, combinés forment le type générique *Couple (a, b)*. Trois opérations sont nécessaires pour manipuler un couple : un constructeur *unCouple* et deux observateurs *premier* et *second* pour obtenir respectivement le premier et le deuxième élément

du couple.

Le mot-clé *spec* introduit la spécification, ici de nom *Couple ;* elle prendra fin à la rencontre du mot-clé *endspec*. Sont ensuite listés les types utilisés, comme *a*, *b* et *Couple (a, b)*. Les opérations sont définies l'une après l'autre et précédées par le mot-clé *op*. Le mot-clé *fa* d'un axiome correspond au quantificateur universel *pour tout* (*for all*).

```
Couple = spec
type a
type b
type Couple (a, b)
op unCouple : a * b -> Couple (a, b)
op premier : Couple (a, b) -> a
op  second : Couple (a, b) -> b
axiom p1 is fa (x : a, y : b)
      premier (unCouple(x, y)) = x
axiom p2 is fa (x : a, y : b)
      second (unCouple(x ,y)) = y
endspec
```

2) Spécification d'une liste

La spécification algébrique d'une structure de données est une technique largement connue. Voici le code Specware de la spécification d'une liste. Nous introduisons tout d'abord la définition du type *Liste* d'éléments de type *a* :

```
op nil : Liste a
op cons : a * Liste a  -> Liste a
op longueur : Liste a -> Nat
op tête  : Liste a -> a
op queue : Liste a -> Liste a
op concat : Liste a * Liste a -> Liste a
```

Pour définir cette spécification, on introduit deux sortes *a* et *Liste a*, qui sont des types génériques. *Liste a* peut spécifier, par un élément de type *a*, n'importe quelle sorte comme élément de la liste. Ici on définit donc une liste générique ou polymorphe, avec un élément *a* pouvant prendre toutes les formes possibles.

Nous décrivons maintenant les opérations applicables à une liste :

Cette seconde partie de la signature identifie les opérations

nécessaires. On trouve des opérations constructives comme *cons* et *concat*. Cette dernière a pour objectif de juxtaposer les éléments de deux listes. L'observateur *tête* retourne l'élément en début de la liste, à condition qu'elle ne soit pas vide.

Au travers des axiomes du type *Liste*, nous donnons la sémantique des opérations précédemment définies. Ils s'écrivent :

```
axiom p1 is nil = Nil
axiom p2 is fa (l : Liste a, a : a)
      cons (a, l) = Cons (a, l)
axiom p3 is fa (l : Liste a)
      longueur (Nil) = 0
axiom p4 is fa (l : Liste a, a : a)
      longueur (Cons (a, l)) = longueur (l) + 1
axiom p5 is fa (l : Liste a, a : a)
      tête (Cons (a, l)) = a
axiom p6 is fa (l : Liste a, a : a)
      queue (Cons (a, l)) = l
axiom p7 is fa (l : Liste a)
      concat (Nil, l) = l
axiom p8 is fa (l1 : Liste a, l2 : Liste a, a : a)
      concat (Cons (a, l1),l2) =
      cons (a, (concat (l1, l2)))
endspec
```

Les axiomes sont nécessaires pour décrire le comportement des opérations. Autrement dit, ils indiquent la sémantique de la structure *Liste*. Par exemple, pour expliquer l'effet de l'opération *tête*, il convient de se référer à l'axiome *p5*.

2.2.4-Théorèmes et propriétés

La spécification algébrique modélise une application via une signature et des axiomes. Elle décrit un ensemble de fonctionnalités en termes d'opérations interprétées par les axiomes. D'un point de vue logique, on pourra déduire des théorèmes, c'est-à-dire les prouver, en se basant sur les axiomes définis. En effet, dans le cas général, la spécification algébrique consiste en une théorie équationnelle du premier ordre. Une signature adjointe à un ensemble de variables nous permet de construire des composants logiques, qui constituent un cadre formel d'où l'on peut établir des preuves de théorèmes, par une équation de la forme $t = t'$.

Au sein d'une application, les fonctionnalités de base constituent la matière essentielle d'une signature. Cet ensemble d'opérations n'est pas suffisant pour décrire des fonctionnalités complémentaires, qui peuvent se déduire des fonctions de base. Ces fonctionnalités s'expriment dans notre cadre par des théorèmes.

2.2.5-Preuve d'un théorème

Un théorème décrit une propriété déduite des axiomes. On peut démontrer un théorème selon plusieurs stratégies de déduction. Des outils basés sur des techniques de preuve peuvent être utilisés durant ce processus de démonstration, comme l'assistant de preuve Isabelle [97] connecté à Specware et basé sur la logique HOL. Des explications plus détaillées sur ces outils seront fournies dans le chapitre suivant.

Soit par exemple le mécanisme de concaténation de plusieurs listes. L'opération de concaténation est associative : la concaténation de trois listes fournit la même liste résultat, indépendamment de l'ordre d'exécution des concaténations élémentaires de deux listes. Pour formaliser cette propriété, on écrit un théorème *t1* formalisé comme suit en Specware :

```
theorem t1 is
fa (l1 : Liste a, l2 : Liste a, l3 : Liste a)
      concat (concat (l1, l2), l3) =
      concat(l1, concat (l2,l3))
```

Voici un autre exemple exprimant une propriété des listes : la longueur de la concaténation de deux listes est égale à la somme des longueurs des deux listes. Cette propriété s'exprime par le théorème *t2* défini en Specware comme suit :

```
theorem t2 is
fa (l1 : Liste a , l2 : Liste a)
      longueur (concat (l1, l2)) =
      longueur (l1) + longueur (l2)
```

La différence entre un théorème et un axiome est qu'un axiome est défini par construction comme une proposition valide, en tant que composante intrinsèque à la spécification. Par contre un théorème est

une proposition déduite des axiomes ; il n'est pas valide tant qu'il n'a pas été démontré. Le théorème *t2* a besoin d'être démontré, tout comme *t1*. La démonstration du théorème *t2* est basée sur l'induction structurelle en utilisant les axiomes qui définissent la fonctionnalité de la concaténation.

Plusieurs méthodes formelles peuvent aider à prouver la validité d'un théorème, au travers notamment des mécanismes de déduction et d'induction. Pour démontrer le théorème *t2*, on utilise une logique d'induction qui se propose de chercher des lois générales, à partir de l'observation de faits particuliers.

Par induction sur la longueur de la liste *l1*, on peut démontrer la validité du théorème *t2*. Supposons *longueur (l1) = n*.

Pour n = 0, c'est le cas d'une liste vide. On a alors :

d'une part *longueur (concat (Nil, l2)) = longueur (l2)*

et d'autre part *longueur (Nil) + longueur (l2)*

= 0 + *longueur (l2)*

= *longueur (l2)*

Pour *n + k*, selon la logique d'induction, on suppose que *t2* est vérifié pour une longueur donnée *n*, et on le démontre pour *n + k* tel que *k < n*. Pour toute liste *l2*, si *longueur (l1) = n* alors

longueur (concat (l1, l2)) = longueur (l1) + longueur (l2).

Soit *l1'* , tel que *l1' = concat (l1'$_a$, l1'$_b$)*, une liste de longueur *n + k* où *longueur (l1'$_a$) = n* et *longueur (l1'$_b$) = k*, il faut démontrer :

longueur (concat (l1', l2)) = longueur (l1') + longueur (l2).

D'après l'hypothèse d'induction, on a :

longueur (l1')

= *longueur (concat (l1'$_a$, l1'$_b$))*

= *longueur (l1'$_a$) + longueur(l1'$_b$)*

= *n + k*

Le membre droit de l'équation définissant le théorème *t2* est égal à *n + k + longueur (l2)*. Il reste à démontrer que son membre gauche est aussi égal à *n + k + longueur (l2).*

longueur (concat (l1', l2))

= longueur (concat (concat (l1'$_a$, l1'$_b$), l2))

= longueur (concat (l1'$_a$, l1'$_b$)) + longueur (l2)

= longueur (l1') + longueur (l2)

= n + k + longueur (l2)

On peut déduire, selon la logique d'induction, que le théorème *t2* est valide. Dans notre raisonnement, on utilise les axiomes de base et les théorèmes existants à condition qu'ils aient été préalablement validés.

2.3-Spécification algébrique et langage de programmation

Un langage de programmation est décrit par deux composantes : sa syntaxe et sa sémantique. La syntaxe décrit les constructions autorisées par le langage, c'est-à-dire énonce les règles qu'il faut respecter pour écrire un programme dans le langage. La sémantique définit l'ensemble des comportements des constructions syntaxiques du langage ; elle en précise donc sa signification. Par exemple, deux langages différents peuvent avoir la même règle syntaxique, mais avec deux sémantiques différentes, ce qui implique des comportements différents pour une même syntaxe. A l'inverse, un même comportement pourra se décrire selon plusieurs syntaxes.

Pour un langage de programmation donné, sa syntaxe est relativement aisée à définir car des formalismes bien connus existent comme les règles de production par exemple. Le plus difficile concerne sa sémantique. Parmi les multiples approches, nous nous intéressons aux spécifications algébriques afin de définir la sémantique d'un DSL.

En effet, la spécification algébrique d'un DSL est maîtrisable compte tenu de la taille a priori limitée de ce type de langage. De plus, l'existence d'outils aide à manipuler des spécifications algébriques et à formaliser des spécifications au sein d'environnements dédiés, comme ASF+SDF [98] ou Specware [96]. D'autre part, il existe une relation étroite entre spécification algébrique et sémantique d'un langage de programmation, dès lors qu'un langage de programmation peut être vu comme un ensemble de sortes et d'opérations appliquées à ces sortes.

2.3.1-Signature

Dans un premier temps, il faut identifier les domaines de valeurs manipulés par le langage de programmation. Ces domaines définissent des sortes ou types, comme les entiers ou les booléens. Ces ensembles de sortes identifient une partie de la signature de la spécification algébrique associée au langage. D'autres types sont nécessaires ; ils résultent des entités manipulées par les constructions syntaxiques du langage, comme les déclarations, les instructions ou les sous-programmes. Ces constructions sont en fait un assemblage d'autres constructions qui permettent d'en composer d'autres à leur tour. Par exemple, dans le langage HTML, un programme est composé d'une tête et d'un corps ; à leur tour, ils introduisent d'autres composants. Au niveau de l'exécution, la manipulation du corps ne ressemble pas à celle de la tête.

La figure 4 illustre ce processus d'identification de la signature d'une spécification algébrique découlant directement de la grammaire d'un langage de programmation.

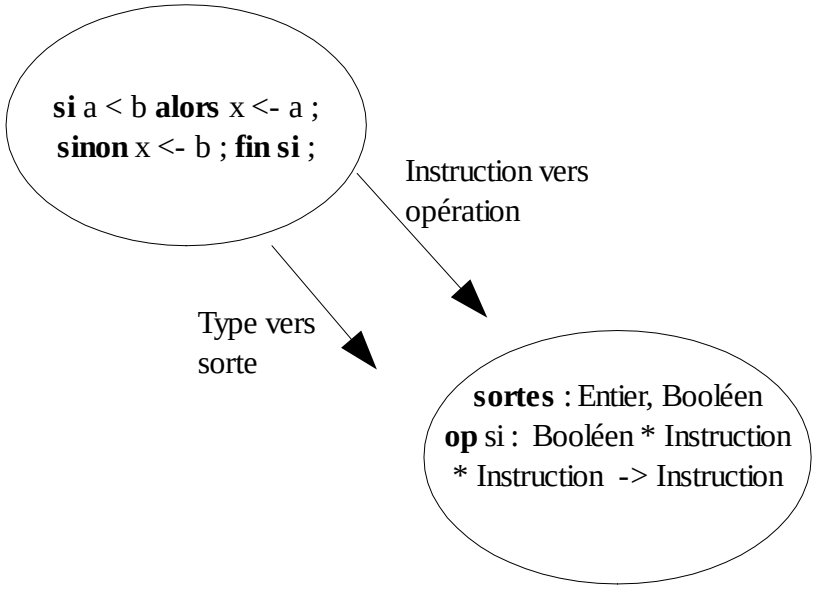

Figure 4: Du langage vers une signature

Les instructions qui manipulent des sortes, comme par exemple les instructions de contrôle, se déclinent ainsi en termes d'opérations au sein de la signature algébrique du langage. Ces opérations considèrent

des sortes hétérogènes. Par exemple, l'instruction de sélection du langage *L* de la forme *si ... alors ... sinon ...* a un domaine égal au produit cartésien *Booléen x Instruction x Instruction* et un co-domaine égal à *Instruction*. Au regard de la syntaxe, cela revient à considérer l'instruction de sélection *si* comme une opération sur les sortes *Booléen* et *Instruction*. D'autres instructions aidant à structurer et à ordonner l'exécution d'un programme introduiront des sortes spécifiques, comme par exemple *Instuctions* dans le cas d'une séquence d'instructions.

Plus généralement, à tout non terminal de la grammaire correspond une sorte et à chaque terme associé à une règle de production *i* correspond un opérateur r_i. Ainsi, si l'on considère la règle de production L_i mettant en jeu les non terminaux L_{i1}, L_{i2} ..., L_{in}, on introduit dans la signature associée au langage un opérateur r_i de profil $r_i : L_{i1} \times L_{i2} \times ... \times L_{in} \rightarrow L_i$.

2.3.2-Axiomes

Après transformation des types et des instructions du langage en signature d'une spécification algébrique, il reste à expliciter la sémantique des opérations, ce qui permet d'associer une sémantique au langage de programmation. Les axiomes traduiront l'effet des constructions du langage de programmation sur une machine abstraite. Ces axiomes permettront par la suite de valider un théorème, et donc de caractériser une propriété du langage.

De façon générale, les règles de grammaire ainsi traduites constituent des opérations qui remplacent les non terminaux du membre droit par leurs arguments ; le résultat est alors un mot engendré par le non terminal du membre gauche. Au regard de l'opération r_i, ce constat conduit à interpréter un terme de la forme r_i, muni au moins des paramètres t_{i1}, ti_2, ..., t_{in} mis en évidence par le profil de l'opération. C'est ici qu'il s'agit d'écrire, en s'appuyant sur les types abstraits précédemment définis, les axiomes chargés d'interpréter de tels termes.

2.3.3-Application : spécification du langage *L*

Pour décrire la sémantique du langage *L*, on commence par considérer sa syntaxe abstraite décrite à la figure 5, car la syntaxe concrète qui ajoute, pour l'utilisateur, du sucre syntaxique aux constructions du langage présente peu d'intérêt d'un point de vue

```
programme ::= nom_de_programme glossaire? instructions
glossaire ::= (déclaration_de_variable)+
déclaration_de_variable ::= nom_de_variable  type
instructions ::=  instruction+
instruction ::= séquence | sélection | répétition | affectation
séquence::= instruction+
sélection::= expression  instruction (instruction)?
répétition::=  expression  instruction
affectation ::= variable  expression
expression ::= opérande | opération |  expression
opération::= opérande opérateur opérande
opérande::= constante | nom_de_variable
```

Figure 5: Syntaxe abstraite du langage L

sémantique. Par exemple, considérant une sélection, on peut en définir plusieurs syntaxes concrètes pour une même syntaxe abstraite mettant ici en jeu une expression (la condition de la sélection) et deux instructions (les deux branches d'exécution de la sélection).

L'intérêt d'utiliser la syntaxe abstraite résulte de sa simplicité et de sa lisibilité, mais surtout, en ce qui nous concerne, de la facilité avec laquelle on peut migrer vers la signature d'une spécification algébrique. Par la suite, on pourra ainsi décrire aisément le profil d'une opération, en déterminant son domaine et son co-domaine.

A la figure 6, nous fournissons ainsi la signature du langage *L* considérant sa syntaxe abstraite. Cette signature réutilise des sortes que l'on peut considérer comme prédéfinies, par exemple les sortes *Entier* et *Booléen* pour l'opération *type* en charge de produire un terme de sorte *Type*. Les autres sortes de la spécification découlent directement des concepts manipulés par le langage comme *Variable, Instruction, Expression*... qui constitueront autant de noms de domaine ou de co-domaine pour définir d'autres profils comme *déclarationVar, séquence, sélection, répétition, affectation, expression, variable, opérande*...

```
programme : NomProgramme * Glossaire * Instructions -> Programme
programme : NomProgramme * Instructions -> Programme
glossaire : ( NomVariable )+ -> Glossaire
déclarationVar: NomVariable * Type -> Variable
type : Entier -> Type
type : Booléen -> Type
instructions : Instruction+ -> Instructions
séquence : Instruction+ -> Instruction
sélection : Expression * Instruction -> Instruction
sélection : Expression * Instruction * Instruction  -> Instruction
répétition : Expression * Instruction  -> Instruction
affectation : Variable * Expression -> Instruction
expression : Opérande -> Expression
expression : Opération -> Expression
expression : Expression -> Expression
opération :Opérande * Opérateur * Opérande -> Opération
opérande : Constante -> Opérande
opérande : NomVariable -> Opérande
```

Figure 6: Signature du langage L

La figure 7 définit certaines de ces sortes du langage *L* en Specware. Le type *séquence* y est défini comme une liste d'instructions afin d'interpréter le fait que l'opération de même nom doit traiter plusieurs instructions.

Une fois la signature de notre langage *L* déterminée, nous pouvons passer à l'étape suivante qui consiste à définir formellement les opérations nécessaires afin de décrire les fonctionnalités du langage. Il s'agit donc de définir entre autres les opérations suivantes : *sélection*, *répétition*, *séquence*, *instruction*... mais aussi dans ce cas précis l'opération d'*affectation qui* manipule la sorte *Variable* en changeant les valeurs stockées en mémoire.

La mise en œuvre fonctionnelle d'une machine abstraite est donc nécessaire pour définir la sémantique du langage *L*. En effet, la sémantique d'un langage de programmation se doit d'expliquer les effets des constructeurs du langage, et en particulier dans ce cas précis les opérations ayant un effet sur cette machine abstraite.

Les fonctionnalités du langage *L* sont réduites à des calculs logiques

```
L = spec
type Expression
type Instruction
type Séquence = List Instruction
```

Figure 7: Les sortes de base du langage L

ou arithmétiques sur un ensemble de variables. Dans ce cas, la machine abstraite du langage *L* peut se présenter sous la forme d'un type *Environnement* qui devient dès lors un élément de l'ensemble des types de la spécification algébrique *L*.

Les opérations de base de *L* ont des effets sur cet environnement. Certaines consultent les valeurs des variables, d'autres les modifient. On peut abstraire le résultat de l'exécution d'un programme *L* en un environnement terminal, modifié au fur et à mesure par le déroulement des instructions du programme. Selon *L*, un environnement est composé d'un ensemble de couples (*variable, valeur)*, pour lequel on associe à chaque *variable* une *valeur* unique instantanée. Cette valeur est manipulée et modifiée par les opérations de la spécification algébrique *L*.

La figure 8 définit en Specware les types nécessaires à la représentation d'un environnement d'exécution. Pour cela, on introduit les types *Variable* pour typer les variables qui sont définies et manipulées par un programme et le type *Valeur* pour typer la valeur d'une variable. Dès lors, un environnement est une liste de couples (*variable*, *valeur*) permettant de typer tous les contextes d'exécution d'un programme *L*.

```
type Variable
type Valeur
type Couple = Variable * Valeur
type Environnement = List Couple
```

Figure 8: La sorte Environnement

A celà s'ajouteront des opérations classiques de manipulation de l'environnement (consultation, adjonction, suppression et modification) afin de traduire l'effet des instructions, notamment celle d'affectation mettant à jour la valeur d'une variable. Il convient dès

lors de rajouter ce type *Environnement* en domaine et co-domaine de chacune des opérations de *L* afin de décrire sa sémantique, comme illustré à la figure 9. A titre d'exemple, l'opération é*valSélection* aura en charge l'évaluation d'une sélection lorsque celle-ci nécessitera d'être exécutée par le programme. De façon similaire, é*valInstruction* doit capter la modification entraînée par son exécution et donc produire un nouvel environnement à partir d'un environnement existant. En d'autres termes, ces opérations traduisent l'effet des instructions *L* sur la machine abstraite, et donc définissent formellement *L*.

```
op évalSélection : Expression * Instruction *
     Instruction * Environnement -> Environnement
op évalRépétition : Expression * Instruction
     * Environnement -> Environnement
op évalExpression: Expression * Environnement -> Booléen
op évalInstruction : Instruction * Environnement ->
     Environnement
op évalSéquence : Séquence * Environnement ->
     Environnement
```

Figure 9: Les opérations de la spécification algébrique de L

Nous sommes maintenant en mesure de définir cette sémantique au travers des axiomes de la spécification. Il convient donc de définir un ou plusieurs axiomes décrivant chacune des opérations d'évaluation précédemment définies.

L'évaluation d'une séquence, spécifiée par l'axiome *axEvalSéquence*, se traduit par l'évaluation, dans l'ordre, de chacune des instructions qui la constituent. De même l'axiome *axEvalRépétition* définit le comportement d'une répétition, comme illustré à la figure 10. L'idée est d'exécuter

```
axiom axEvalRépétition is fa
(exp : Expression, i : Instruction, env : Environnement)
 (case évalExpression (exp, env) of
     | true ->  évalRépétition (exp, i, env) =
     évalRépétition (exp, i, évalInstruction (i, env))
     | false -> évalRépétition (exp, i, env) = env))
endspec
```

Figure 10: L'axiome de la répétition

l'instruction du corps de la boucle itérativement, en produisant à chaque itération un nouvel environnement.

```
op évalExpression : Expression * Environnement-> Valeur
```

Figure 11 : Le profil de l'opération évalExpression

L'opération é*valExpression* dont le profil est spécifié à la figure 11 est rendue nécessaire pour fournir la valeur d'une expression. Cette opération est définie *in fine* sur les sortes *Entier* et *Booléen*. Notons qu'une valeur de sorte *Booléen* (*vrai* ou *faux*) est syntaxiquement imposée pour les instructions de choix de *L* comme la sélection ou la répétition.

```
op évalAffectation : Variable * Expression  *
     Environnement -> Environnement
```

Figure 12 : Le profil de l'opération évalAffectation

L'opération é*valAffectation* prend en domaine une variable, une expression et bien sûr, un environnement comme indiqué par la figure 12. Après exécution de cette instruction, l'environnement est mis à jour comme en témoigne la sorte de même nom en co-domaine.

```
axiom axEvalAffectation is fa
(env : Environnement, v : Variable, exp : Expression)
      (case env of
       | [] ->  évalAffectation (v, exp, env) = env
       | p::q ->
            if (project 1 p) = v then
                  (project 2 p)
                  = évalExpression (exp, env)
            else évalAffectation (v, exp, env
                  = évalAffectation (v, exp, q))
```

Figure 13: L'axiome de l'affectation

L'axiome de la figure 13 exprime le comportement de cette opération d'affectation. L'environnement est parcouru en recherchant la variable en membre gauche de l'affectation. Lorsque cette variable a été identifiée, la fonction é*valExpression* est invoquée afin de déterminer sa nouvelle valeur qui remplacera l'ancienne. Dans ce code, nous faisons l'hypothèse que l'ensemble des variables du programme sont

déjà stockées au sein de l'environnement, tâche assignée à l'interprétation d'un glossaire *L* qui impose que toutes les variables du programme soient déclarées. D'autre part, les fonctions Specware *project* permettent l'accès à une des composantes d'un type produit.

De plus, la notation [] Specware correspond ici au symbole fonctionnel *Nil* des listes et *p::q* permet de filtrer en une seule notation à la fois la tête *p* et son reste *q*.

Dans ce chapitre, nous avons donc montré comment spécifier algébriquement la sémantique d'un langage de programmation. A partir de la syntaxe abstraite du langage, nous avons été en mesure d'isoler aisément et de manière quasi-automatique les sortes et les opérations manipulées par le langage, ce qui nous a permis de définir la signature du langage. La sémantique a pu ensuite être fournie par le biais d'axiomes décrivant le comportement attendu, en l'occurrence l'effet des instructions sur une machine virtuelle limitée dans notre cas à un ensemble de couples (variable, valeur) modélisant la mémoire. C'est cette approche que nous préconisons en tant que base de travail au niveau de chacun des DSL à faire interopérer.

Chapitre III

3-Construction formelle du langage unificateur à un domaine

Nous proposons dans ce chapitre une approche qui vise à résoudre le problème de l'interopérabilité des DSL. Nous limitons notre étude à un ensemble de DSL appartenant à une famille dédiée à un domaine métier. L'interopérabilité des langages de programmation pose le problème de la communication entre ces langages ; nous essayons dans ce travail de trouver les média nécessaires pour établir cette communication.

Notre idée consiste à identifier un langage unificateur de la famille des DSL. Nous proposons un environnement pour développer un langage unificateur d'une famille de DSL dans lequel nous déterminerons les outils et les technologies nécessaires à son établissement.

Notre stratégie est basée sur l'abstraction des composantes des langages d'une famille vers ce langage unificateur en utilisant des techniques basées sur des théories mathématiques. La fonction *pushout* (ou somme amalgamée) de la théorie des catégories permet de combiner les composantes de plusieurs objets au sein d'un nouvel objet créé par la fonction elle-même. Cette fonction sert de modèle à notre démarche afin d'atteindre notre objectif de façon systématique. De plus, la spécification algébrique, méthode formelle de spécification, sera utilisée pour modéliser la sémantique d'un DSL.

Dans ce chapitre, nous mettons l'accent sur notre approche pour résoudre le problème de l'interopérabilité. Nous expliquons notre idée, la stratégie utilisée pour développer le langage unificateur et les techniques utilisées. Nous commençons par situer le contexte en déterminant la place de notre problème par rapport au problème de l'interopérabilité qui est vaste et très général. Ensuite, nous détaillons notre approche en exposant sa partie théorique et en explicitant les techniques qu'elle nécessite. La dernière partie du chapitre sera consacrée à une étude expérimentale afin de valider l'idée théorique : nous considérons deux fragments DSLα *et DSL*β inspirés d'un langage du domaine spatial sur lesquels nous appliquons notre démarche et nous générons leur langage unificateur DSL$\alpha\beta$. Specware, outil de manipulation de spécifications, nous permet de calculer la fonction *pushout* de construction du langage DSL$\alpha\beta$.

3.1-Interopérabilité

3.1.1-Interopérabilité dans un domaine métier

Nous nous intéressons à l'utilisation combinée de différents langages dédiés. Plusieurs stratégies sont traditionnellement employées pour permettre l'interopérabilité dans un tel contexte. Dans le cas de l'interopérabilité des langages, la technique la plus utilisée consiste à définir un langage intermédiaire jouant le rôle d'un pivot entre les langages et utilisé comme un langage de référence pour les différents applicatifs. Nous avons besoin également de traducteurs bidirectionnels depuis les différents langages qu'on souhaite faire communiquer ensemble et envers ce langage pivot.

L'avantage de l'utilisation d'un langage pivot évite d'avoir des traducteurs pour chaque paire de langages. Aussi, en fonction du contexte de l'utilisation, on peut établir des propriétés dans le contexte du langage pivot. Un de ses inconvénients est qu'il peut aboutir le plus souvent à un langage plus ou moins proche des langages qu'il est censé fédérer.

Le point de départ de notre travail vient d'une étude proposée au sein du projet RNTL DOMINO (DOMaINes et prOcessus méthodologique)[1] par le Centre National d'Etudes Spatiales (CNES). En effet, plusieurs langages de programmation dans le domaine spatial dédiés à la programmation des services de commande/contrôle d'un satellite ont été développés pour des raisons distinctes : techniques, sociologiques, économiques, etc. Ces langages comme Pluto, Elisa, Stol et d'autres sont différents syntaxiquement et sémantiquement bien qu'ils aient été développés pour servir le même type de tâche et offrir les mêmes fonctionnalités. Cette approche polyglotte d'un même domaine conduit à des problèmes importants d'interopérabilité lorsque les experts de ce domaine doivent collaborer.

Plusieurs DSL peuvent appartenir au même domaine métier, avec des caractéristiques fonctionnelles communes et des syntaxes différentes. L'ensemble de ces langages définit en soi la notion de famille de DSL dédiée à un domaine précis. Les langages d'une famille se caractérisent par leur adéquation à prendre en compte spécifiquement les besoins au plus près du domaine traité. Ils proposent en général

1Contrat ANR-06-TLOG-011

une notation appropriée et des constructions spécifiques au domaine, ce qui permet d'accroître la lisibilité, la concision et la sûreté des programmes ainsi que l'accessibilité au langage par des experts non-informaticiens.

Nous ne traitons pas le problème de l'interopérabilité dans un contexte général, mais nous essayons d'introduire des paramètres pour l'aborder dans un contexte spécifique. Nous essayons de systématiser une approche capable de résoudre le problème de l'interopérabilité dans une famille de DSL dédiée à un domaine métier. La difficulté évidente qui s'en suit est la babélisation générée par la diversité des langages d'une même famille. Pour une famille de DSL donnée, le défi d'interopérabilité revient à identifier les meilleures techniques qui permettront d'utiliser conjointement et de manière cohérente ses différents membres, tout en continuant à exploiter les environnements et les outils spécifiques à chacun.

Pour garantir cette interopérabilité, nous avons initialement mis en oeuvre une première approche [99] [100]. Celle-ci consistait à résoudre cette question en définissant chacun des langages par un méta-modèle. La difficulté consiste ici à faire coopérer ces méta-modèles. En effet, la portée des résultats d'une telle approche est limitée par le fait qu'elle ne prend pas en compte les aspects sémantiques, les méta-modèles étant par nature des descriptions structurelles. Par conséquent, l'outillage est forcément limité à des transformations de modèles et ne couvre pas le comportement des composants spécifiés avec ces méta-modèles. Pour pallier les difficultés sémantiques exposées ci-dessus, nous avons décidé d'infléchir le cadre conceptuel vers la modélisation orientée domaine et de traiter en tant que tels les DSL qui caractérisent une famille [101].

3.1.2-Famille de DSL

L'utilisation des langages dédiés à un domaine ou DSL a connu un essor important ces dernières années. Ces langages dont la syntaxe et la sémantique sont spécifiques offrent une expression suffisante pour qu'un non spécialiste de l'informatique, expert dans le domaine métier, puisse directement les exploiter.

L'intérêt des DSL réside donc dans leur simplicité et leur relation forte avec le domaine d'application. Le revers de la médaille est que, compte tenu de ces caractéristiques, des experts différents d'un même domaine soient tentés de créer des DSL distincts pour exprimer des

besoins similaires sur un champ sémantique identique conduisant ainsi à une famille de DSL. Les langages des procédures opérationnelles du domaine spatial en sont un exemple.

Une famille de DSL se définit comme un ensemble de langages dédiés au même type de problème. Ils offrent sur le domaine les mêmes fonctionnalités, au même niveau d'abstraction, avec cependant des différences syntaxiques et sémantiques entre les membres de la famille. Dans ce contexte, le problème de l'interopérabilité est d'identifier les meilleures techniques permettant d'utiliser les différents langages de manière cohérente. Un de nos objectifs est aussi de permettre aux experts de continuer à utiliser les environnements et les outils de chacun des langages.

Un des avantages d'un DSL par rapport à un langage généraliste réside dans le nombre en général limité des ses constructions. En effet, un DSL possède un nombre restreint de constructions grâce à son domaine d'application spécifique, ce qui facilite la tâche de construction de notre langage unificateur. Il s'agit dès lors de tenter de rassembler au sein d'un unique langage les concepts communs à la famille, en cherchant à s'abstraire des problèmes syntaxiques et idiomatiques qui nuisent au dialogue entre experts.

Dans le cas d'une famille de DSL, le langage unificateur doit constituer le référentiel du domaine et synthétiser les concepts communs mais suffisants pour déployer les fonctionnalités du domaine. En effet, les DSL d'une famille ont été développés pour réaliser un ensemble de tâches similaires au domaine. Dans le domaine spatial par exemple, le langage unificateur des procédures opérationnelles doit être en mesure de déployer des services à base de commandes/contrôles pour guider les satellites. On en déduit que le langage unificateur est lui-même un DSL de la famille qu'il est censé fédérer.

3.2-Définition de notre approche

3.2.1-Synoptique

Nous définissons de manière synthétique l'approche développée pour résoudre le problème de l'interopérabilité dans une famille de DSL. Plusieurs méthodes ont été développées pour résoudre ce problème, comme celle du langage pivot pour passer d'un langage ou celle de la définition d'un méta-modèle commun [102].

Notre stratégie se fonde sur l'idée de définir formellement un langage unificateur des DSL d'un domaine métier, en identifiant de manière plus ou moins automatique les concepts communs aux différents langages, comme illustré à la figure 14. Ce DSL unifié devra couvrir les fonctionnalités de chacun des DSL de la famille.

En considérant les DSL comme des objets d'une catégorie, nous avons été en mesure de partager des caractéristiques entre ces objets. En particulier, le calcul de la co-limite de la théorie des catégories [103] [104] [105] nous permet de calculer un nouvel objet à partir de plusieurs objets sources. Un nouvel objet est ainsi créé sans redondance des structures des objets qui participent au calcul, tout en partageant les concepts communs. L'originalité de nos travaux consiste à exploiter ce calcul de co-limite ; dans le cas de deux objets, il porte le nom de *pushout* ou somme amalgamée et correspond, grossièrement, à « l'intersection sémantique » des deux DSL initiaux [23]. L'analogie entre ce calcul et l'objectif d'interopérabilité nous paraît la meilleure réponse au problème posé.

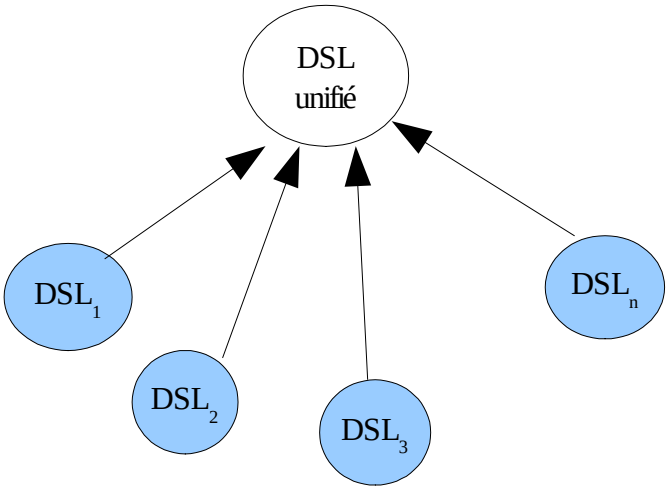

Figure 14: Factorisation des fonctionnalités communes aux DSL d'un même domaine métier

La théorie des catégories [103] [104] [105] permet d'étudier les structures et les relations entre ces structures sans tenir compte de leurs propriétés internes. Une catégorie est composée d'objets et de morphismes entre ces objets. Pour appliquer la co-limite aux DSL, nous avons à définir une catégorie spécifique, exploitant des objets que sont les DSL et des morphismes entre ces objets, à savoir des dépendances syntaxiques et sémantiques entre les DSL.

La spécification algébrique détaillée au chapitre précédent nous apporte une réponse quant à la catégorie à exploiter concrètement. Il s'agit de celle utilisée pour modéliser la sémantique d'un DSL, à savoir la catégorie des spécifications algébriques. Dans ce contexte, un objet est une spécification algébrique et un morphisme se traduit par un morphisme entre deux spécifications algébriques. L'application de la co-limite dans ce contexte identifie une nouvelle spécification algébrique (le DSL unificateur) et les morphismes qu'elle définit expriment des liens entre les spécifications algébriques (les liens de correspondance entre les DSL de la famille vis-à-vis du DSL commun).

Dans les paragraphes qui suivent, nous introduisons l'outillage nécessaire pour valider notre approche ainsi que l'environnement visant à construire le langage unificateur en question.

3.2.2-Techniques et mécanismes utilisés

Systématiser notre approche exige l'utilisation de techniques et d'outils. Dans ce paragraphe, nous présentons les techniques utilisées et nous situons le rôle et l'utilité de chacune dans notre démarche.

3.2.3-Les spécifications algébriques

La première a trait à la spécification formelle de la sémantique de chacun des DSL de la famille. Comme souligné au chapitre précédent, le formalisme des spécifications algébriques se focalise sur la sémantique au travers d'axiomes. Dans le contexte des DSL, ce formalisme paraît bien adapté et des démarches et outils existent pour aider à une telle modélisation. Les spécifications formelles constituent ainsi un domaine largement étudié et les premiers travaux qui nous concernent directement remontent à Goguen au début des années 80 [83]. Pour la mise en œuvre informatique, il convient de trouver un environnement logiciel adapté. Plutôt que d’utiliser des langages algébriques généralistes tels que OBJ [106] ou CASL [107], ou des langages spécifiques comme Community [108], notre choix s’est porté vers Specware [96], un DSL pour les spécifications algébriques.

Le formalisme retenu a une forte capacité à exprimer la sémantique d'un DSL. Il ne vise pas un objectif précis en lien avec la programmation, comme cela pourrait être le cas avec la sémantique axiomatique qui s'intéresse à la vérification et la validation de programmes, tout comme la sémantique opérationnelle idéale pour

définir un compilateur et justifier le comportement des programmes. Il est cependant important de noter que d'autres formalismes pourraient être exploités sans nuire à la généralité de l'approche ; dans ce cadre, il conviendrait d'étendre notre proposition aux institutions de Goguen et Burstall [109] afin de répondre à l'hétérogénéité des formalismes de spécification, comme suggéré par Diaconescu [110]. Dans un souci de validation, nous avons limité notre étude à l'exploitation d'un seul formalisme pour exprimer la sémantique des DSL.

3.2.4-La théorie des catégories

La deuxième technique se base sur la théorie des catégories [103]. Celle-ci est exploitée par les informaticiens depuis longtemps. C'est un langage mathématique universel destiné à étudier les structures et les relations entre ces structures sans tenir compte de leurs caractéristiques internes. La « catégorie formelle » des DSL que nous considérons met en jeu des morphismes de spécifications algébriques qui constituent par essence des relations de dépendance entre spécifications algébriques, comme par exemple celles de composition et de raffinement.

De plus, un certain nombre de résultats connus de la théorie des catégories pourront être exploités, comme par exemple les conditions d'existence d'un *pushout* en ce qui nous concerne. L'émergence d'environnements liés à ce formalisme nous a conforté dans l'idée de pouvoir outiller et automatiser en partie la démarche.

De plus, l'approche algébrique permet de mettre en place relativement facilement les morphismes entre DSL. En effet, la fonction *pushout* nécessite, comme on le verra ultérieurement, la définition d'un DSL en tant qu'amorce au calcul et la mise en place de morphismes connectant les deux DSL à unifier avec l'amorce. Établir ces éléments catégoriques peut en partie s'automatiser.

C'est ainsi que nos expérimentations ont pu être validées au travers de l'environnement Specware [111] [112] [113] [114]. Specware est un environnement de développement logiciel permettant aux concepteurs de spécifier formellement la fonctionnalité de leur système en termes de spécifications algébriques et de produire, via des étapes de raffinement, la traduction de cette spécification vers du code exécutable correct. Il soutient l'idée qu'un système complexe peut être spécifié en composant des spécifications de systèmes plus simples, de telle sorte que les points de décision soient contrôlés et prouvés. Specware introduit la notion de morphisme de spécifications

algébriques et autorise le calcul de leur co-limite, point qui nous intéresse plus particulièrement. L'environnement intègre une passerelle vers des assistants de preuve, tel Isabelle [97], afin d'établir un théorème à partir des axiomes d'une spécification algébrique. Un composant Specware incluant un énoncé de théorème est ainsi traduit en un code Isabelle afin de bénéficier de l'environnement interactif du démonstrateur.

3.2.5-Bilan technique

L'utilisation conjointe des spécifications algébriques et des catégories nous permet d'appliquer les fonctions des catégories aux DSL de la famille. La co-limite a ainsi en charge de créer un nouvel objet contenant les caractéristiques de plusieurs objets sources. D'autres techniques de preuve basées sur la logique du premier ordre et sur la logique HOL seront également utilisées dans le contexte des spécifications algébriques afin de dégager des propriétés relatives aux DSL que l'on souhaite fusionner, mais aussi celles du DSL commun construit par *pushout*. Au final, nous visons ainsi à offrir un environnement de développement ainsi qu'une démarche méthodologique qui puisse s'appliquer à différents domaines métiers.

Dans le paragraphe suivant, nous mettons l'accent sur la théorie des catégories et son utilité dans notre approche.

3.3-Théorie des catégories

3.3.1-Historique

En 1940, S. Mac Lane et S. Eilenberg [103] ont imaginé la théorie des catégories, une branche des mathématiques permettant de décrire des objets de manière générale et abstraite. Les mathématiciens ont eu des difficultés pour découvrir son intérêt, mais avec le temps elle est devenue essentielle surtout en topologie et en algèbre. L'exploitation de cette théorie en informatique date du début des années 80 dans le domaine de la théorie des types. Ainsi, il existe des relations étroites entre la théorie des catégories et celles des types [115].

Plusieurs ouvrages expliquent la théorie des catégories comme celui de S. Mac Lane [103] qui s'adresse plutôt à des mathématiciens.

D'autres ouvrages ont un point de vue plus informatique comme le livre de M. Barr et C. Wells [116] ou celui de J.L. Fiadero [105] qui en souligne l'intérêt pour le génie logiciel.

3.3.2-Définition

La théorie des catégories est une théorie mathématique permettant d'établir les caractéristiques d'une structure dans la perspective des relations entre objets qui la composent [117] ; elle ne caractérise pas ces objets via leurs propriétés internes [118]. Contrairement à la théorie des ensembles qui repose sur un seul concept (l'ensemble) et qui étudie cette structure en s'intéressant à ses éléments, la théorie des catégories établit des liens entre objets.

Une catégorie se compose d'un ensemble d'objets (points) et de morphismes (flèches) reliant les objets. Sa capacité à s'abstraire de la représentation interne des objets permet de déduire des propriétés sur les objets et les morphismes.

3.3.3-Définition formelle d'une catégorie

Une catégorie C est formée à partir d'une collection d'objets O_c et d'une collection de morphismes M_c telles que :

1. Chaque morphisme est défini sur un domaine (source) et un co-domaine (cible) constitués d'objets de la catégorie. Pour un morphisme $f : X \rightarrow Y$ défini de X vers Y, f a pour domaine X et Y pour co-domaine, ce que l'on note respectivement par $Dom\ (f) = X$ et $Cod\ (f) = Y$.

2. Pour chaque couple de morphismes $f : X \rightarrow Y$ et $g : Y \rightarrow Z$ tel que $Cod\ (f) = Dom\ (g)$, on garantit par construction sa composition notée ';', ce que l'on traduit par $(f ; g) : X \rightarrow Z$, soit encore $Dom\ (f ; g) = Dom\ (f)$ et $Cod\ (f ; g) = Cod\ (g)$.

3. La loi de composition ';' est associative. Ainsi, pour f, g et h tels que $f : X \rightarrow Y$, $g : Y \rightarrow Z$ et $h : Z \rightarrow T$, on a $(f ; g) ; h = f ; (g ; h)$.

4. L'identité $id_x : X \rightarrow X$ existe pour chaque objet X. Elle satisfait $g ; id_x = g$ et $id_x ; f = f$ pour chaque morphisme $g : Y \rightarrow X$ et $f : X \rightarrow Y$.

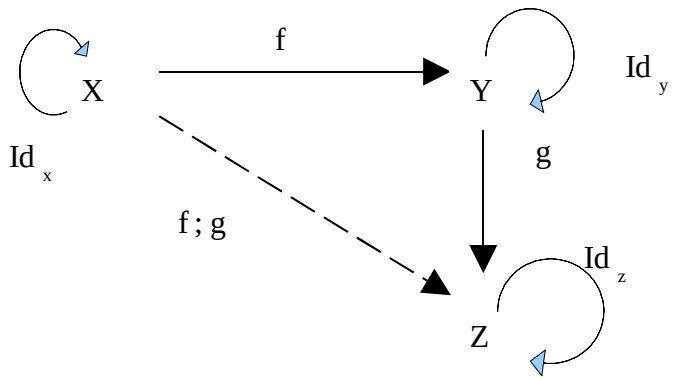

Figure 15: Catégorie modélisée par un graphe orienté

Supposons *C* une catégorie composée de trois objets *X*, *Y* et *Z*, et deux morphismes *f* et *g* tels que $f : X \rightarrow Y$ et $g : Y \rightarrow Z$. Comme le montre sa représentation à la figure 15, nous pouvons modéliser cette catégorie par un graphe orienté dans lequel les sommets sont les objets et les morphismes sont représentés par des arcs.

Pour définir une catégorie, il convient de déterminer ses objets et ses morphismes. De plus, il faut s'assurer de la satisfaction des propriétés sur les morphismes. Considérons par exemple la catégorie des

ensembles. Dans cette catégorie, les objets sont des ensembles et les morphismes correspondent à des relations entre leurs éléments, comme cela est l'usage en théorie ensembliste. La figure 16 représente une telle catégorie composée des objets *A*, *B* et *C* et des morphismes (relations) *R1* et *R2*. Chaque objet modélise un ensemble d'éléments reliés par des équations mathématiques. Ces dernières sont définies par les relations suivantes :

- Le morphisme *R1* entre *A* et *B* est défini de la façon suivante : pour $x \in A$, $y \in B$, $x\ R1\ y$ si et seulement si $y = 2 * x$.
- Le morphisme *R2* entre *B* et *C* est défini de la façon suivante :pour $x \in B$, $y \in$ C, $x\ R2\ y$ si et seulement si $y = 2 + x$.

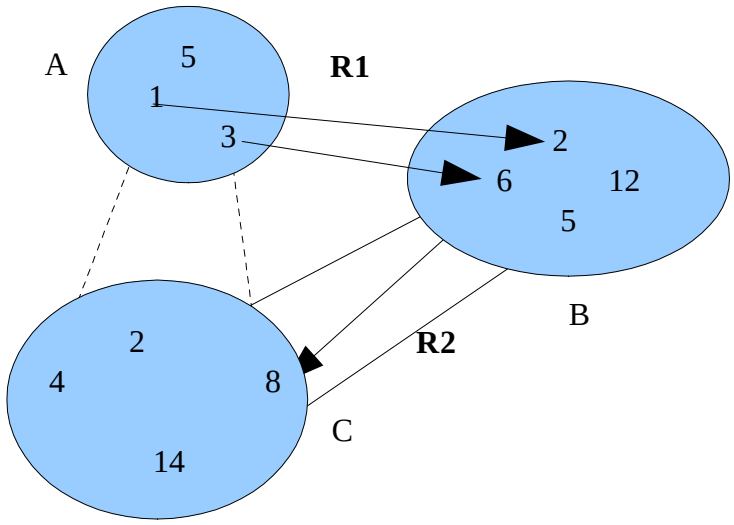

Figure 16: Exemple de catégorie dans la catégorie des ensembles

L'exemple de la figure 16 définit bien une catégorie. La loi de composition existe ; en effet, pour $x \in A$, $y \in B$ et $z \in C$, on doit être en mesure de définir une relation *R3* telle que $x\ R3\ z$ si $x\ R1\ y$ et $y\ R2\ z$. La relation *R3* est telle que $x\ R3\ z$ si et seulement si $z = 2 * x + 2$; en effet, comme $z = y + 2$ par *R2*, le remplacement de *y* par sa valeur $2 * x$ avec *R1* conduit à $z = 2 * x + 2$. Nous déduisons de plus l'existence d'une relation réflexive sur chaque élément de cette catégorie, à savoir la fonction identité sur chacun des éléments d'un ensemble. Cette relation représente l'élément neutre par rapport à la loi de composition des morphismes.

Un autre exemple de catégorie est celui de la figure 17. Il concerne la catégorie des figures géométriques définies dans le plan. Elle a comme objets tous les dessins d'un polygone et les morphismes sont des relations de translations géométriques qui conservent la distance entre les points d'un polygone. Un morphisme peut être vu comme une opération de transformation permettant d'obtenir un autre polygone, orienté différemment dans le plan.

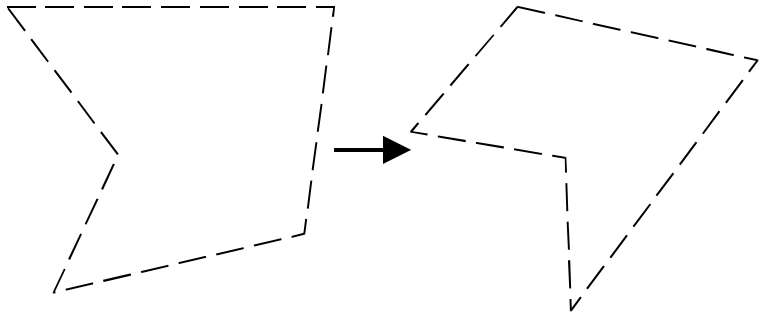

Figure 17: Exemple de morphisme dans la catégorie des polygones

3.3.4-La catégorie des spécifications algébriques

Afin d'exploiter les concepts de la théorie des catégories dans notre travail, et d'abstraire par un calcul de co-limite les fonctionnalités conjointes à une famille de DSL, nous introduisons dans les paragraphes suivants la catégorie des spécifications algébriques.

Dans la catégorie des spécifications algébriques, un objet est une spécification algébrique associée à un DSL ; c'est donc une signature et un ensemble d'axiomes. D'autre part, les morphismes sont des morphismes de spécifications algébriques, ce qui leur confère un certain nombre de caractéristiques comme cela sera précisé ultérieurement.

Un morphisme de spécifications algébriques met en relation deux spécifications. Il est tel qu'il préserve au sein de la cible les propriétés de la spécification source : tous les axiomes de la source sont valides et se traduisent par des théorèmes dans la cible. Il s'agit

donc d'une mise en correspondance des sortes et des opérations, mais aussi des axiomes, de deux spécifications.

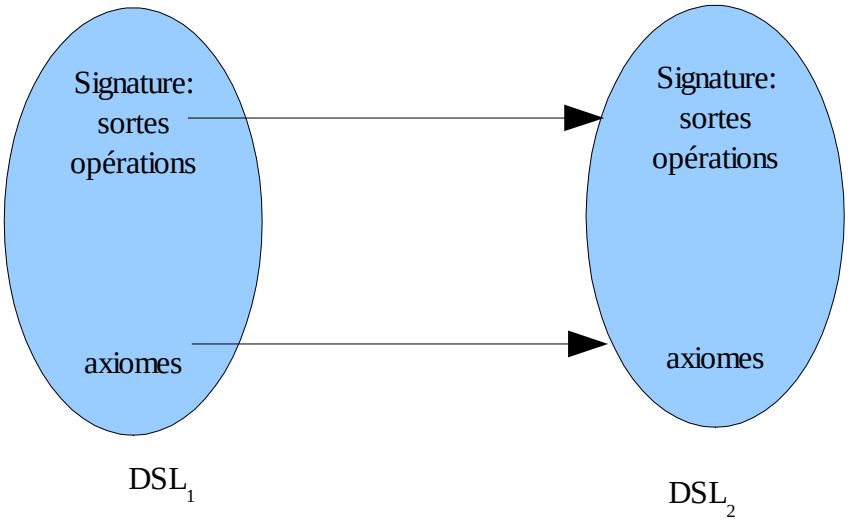

Figure 18: Morphisme de spécifications algébriques

Le morphisme que l'on se doit d'établir entre deux spécifications algébriques implique dès lors de mettre en correspondance les signatures et les axiomes comme illustré à la figure 18. Un tel morphisme dépendra de la structure interne d'une spécification algébrique. Ainsi, initialement, il s'agira de mettre en relation les sortes et les profils des opérations des deux signatures. Dans le cas des sortes, cette relation s'apparente la plupart du temps à un simple renommage ; pour les profils, l'association sera telle qu'elle préserve les domaines et co-domaines de chacune des opérations, au renommage près de sortes.

Dans le cas des axiomes, les contraintes sont plus fortes. Elles sont telles qu'un axiome de la source est un théorème déductible des axiomes de la cible. Si *f* est un morphisme de signatures défini entre deux spécifications algébriques de source *spec1* et de cible *spec2*, *f* peut être qualifié de morphisme de spécifications algébriques si tout axiome *a* de *specl* est un théorème *t* de *spec2*, c'est-à-dire peut se déduire des axiomes de *spec2* en considérant le morphisme de signatures entre *spec1* et *spec2*. Cette contrainte se traduit par la nécessité d'établir la preuve des théorèmes de *spec2* et fait référence en informatique à un assistant de preuve, comme par exemple Isabelle complémentaire à Specware.

Explorons plus en détails les objets et les morphismes manipulés par notre catégorie des spécifications algébriques. Pour cela, considérons l'exemple du morphisme algébrique d'un monoïde (*E*, *, *e*) vers les entiers relatifs.

```
Monoïde = spec
type E
op e : -> E
op f : E * E -> E
axiom a1 is fa (x : E) f (e, x) = x
axiom a2 is fa (x : E) f (x, e) = x
axiom a3 is fa (x : E, y : E, z : E)
      f (f (x, y), z) = f (x, f (y, z))
endspec
```

Figure 19: Spécification algébrique d'un monoïde

On peut tout d'abord caractériser la signature d'un monoïde (*E*, *, *e*) par une sorte *E*, par son élément neutre *e* de profil $e : \rightarrow E$ et par sa loi de composition interne * de profil $* : E\ E \rightarrow E$.

Un morphisme de signatures établit une correspondance symbolique des sortes *S* de deux signatures respectant les profils des opérations mises en jeu. Sachant que l'ensemble des entiers naturels $\mathbb{N}$ muni de l'addition est un monoïde, on établit, ici par un simple «renommage», un morphisme de signatures des monoïdes vers les entiers naturels en associant la sorte *E* à la sorte *Nat* des entiers naturels, la constante *e* à la constante *0* (élément neutre de l'addition), et l'opérateur * à l'opérateur + des entiers naturels de profil + : $Nat\ Nat \rightarrow Nat$.

Une signature est exclusivement syntaxique ; elle doit s'accompagner

```
MonoïdeEntiers = spec
type Nat
op zero : -> Nat
op  plus : Nat * Nat -> Nat

axiom a1 is fa (x : Nat)  plus (zero, x) = x
axiom a2 is fa (x : Nat)  plus (zero, e) =  e
axiom a3 is fa (x : Nat, y : Nat, z : Nat)
       plus (plus (x, y), z) = plus (x, plus (y, z))
endspec
```

Figure 20: Spécification algébrique (incomplète) des entiers

d'axiomes pour définir la sémantique des opérations et tendre vers une spécification algébrique. Un morphisme de spécifications algébriques correspond dès lors au morphisme des signatures correspondantes tel que la traduction de toute équation de la spécification source, par le morphisme des deux signatures, appartienne à la théorie équationnelle de la spécification cible. Sachant que la structure algébrique d'un groupe est celle d'un monoïde dont tous les éléments sont inversibles, on peut établir un morphisme de spécifications algébriques d'un monoïde (*E, *, e*) vers les entiers relatifs ($\mathbb{Z}$, +, *0*) en mettant en

```
MonoïdeEntiers = spec
type Nat
op zero : -> Nat
op  plus : Nat * Nat -> Nat

axiom a1 is fa (x : Nat)  plus (zero, x) = x
axiom a2 is fa (x : Nat)  plus (zero, e) =  e
axiom a3 is fa (x : Nat, y : Nat, z : Nat)
       plus (plus (x, y), z) = plus (x, plus (y, z))
endspec
```

Figure 21: Spécification algébrique (incomplète) des entiers

correspondance les sortes *E* et *Rel* (pour entiers relatifs) et les opérateurs *, *e*, + et 0 comme précédemment et en s'assurant que la traduction des équations relatives au monoïde, à savoir *e * x = x et x * e = x* pour l'inverse *et (x * y) * z = x * (y * z)* pour l'associativité, sont valides dans les entiers relatifs, à savoir *0 + x = x, x + 0 = x* et *(x+ y) + z = x + (y + z).*

```
MonoïdeVersEntiersRelatifs =
      morphim Monoïde -> EntiersRelatifs
      {E +-> Rel, e +-> zero, f +-> plus}
```

Figure 22: Morphisme d'un monoïde vers les entiers relatifs

En Specware, ces spécifications se déclinent conformément au code des figures 19, 21 et 22. Pour des raisons syntaxiques, la loi de composition interne * d'un monoïde est ici désignée par le symbole fonctionnel *f*. La notation *x +-> y* établit le morphisme entre la sorte ou l'opérateur *x* et son homologue *y* ; ce lien est étendu aux profils et axiomes exploitant *x* et *y.*

3.3.5-Opérations catégoriques

Nous présentons maintenant les propriétés fonctionnelles de la théorie des catégories qui intéressent notre étude ainsi que quelques applications de l'opération *pushout* considérée comme un cas particulier de co-limite. Par rapport à une approche mathématique qui s'intéresse essentiellement à l'existence et aux propriétés du *pushout*, nous souhaitons porter nos efforts vers une démarche constructive plus ou moins automatique visant à l'élaboration du langage unificateur.

Plusieurs opérations existent sur les catégories. Parmi ces opérations, certaines sont qualifiées d'universelles car elles s'appliquent à tous les objets ; d'autres sont d'ordre supérieur car elles prennent en compte une catégorie dont les objets sont eux-même des catégories. Pour les premières, on identifie les opérations *pushout, colimit, pullback...* pour les secondes, on parlera par exemple de foncteur (*functor)* pour exprimer un morphisme de catégories et on pourra introduire la catégorie des catégories *Cat* où les objets sont des catégories et les morphismes des foncteurs. Pour notre étude, seul le premier groupe d'opérations est pertinent et notamment l'opération *pushout*.

3.3.6-L'opération *pushout*

Elle joue un rôle essentiel dans les travaux des informaticiens grâce à ses fortes potentialités car elle permet notamment une mise en facteur de spécificités communes à deux objets. Avant de détailler cette opération, nous introduisons et définissons le vocabulaire catégorique nécessaire à sa compréhension.

De nombreuses propriétés catégoriques se représentent par un diagramme. Il s'agit d'une figure composée de sommets et de flèches les reliant. Chaque sommet représente un objet et les flèches sont étiquetés par des morphismes reliant les objets.

Un diagramme d'une catégorie *C* commute si pour toute paire de sommets *X* et *Y,* tous les chemins de *X* à *Y* déterminent une même et unique flèche dans *C*. On affirme ainsi que chaque chemin définit une flèche entre deux objets et que toutes ces flèches fournissent le même résultat. C’est cet aspect de la théorie des catégories qui constitue la pierre angulaire de notre étude. On doit noter que le résultat est construit « à un isomorphisme près », c'est-à-dire sans se préoccuper des détails de représentation des objets, comme par exemple l’ordre des paramètres d’une fonction pour un isomorphisme de types en

programmation.

Soient par exemple *X, Y, Z* et *T* les quatre sommets d'un diagramme *X-Y-Z-T* et *f1, f2, g1* et *g2* les flèches telles que *f1 : X→ Y, f2 : X →Z, g1 : Y→ T* et *g2 : Z→ T.* Le diagramme *X-Y-Z-T* commute si *f1 ; g1 = f2 ; g2*. La figure 23 illustre cette commutativité ; on en déduit que la transmission des propriétés de *X* vers *T* peut considérer soit le chemin du triangle supérieur *X-Y-T* du diagramme, soit le chemin *X-Z-T* du triangle inférieur.

Soient trois objets *X*, *Y* et *Z* munis des deux morphismes *f1* et *f2* définis de *X* vers *Y* et de *X* vers *Z* respectivement. Le résultat du *pushout*, illustré par la figure 24, des deux morphismes *f1* et *f2* est un

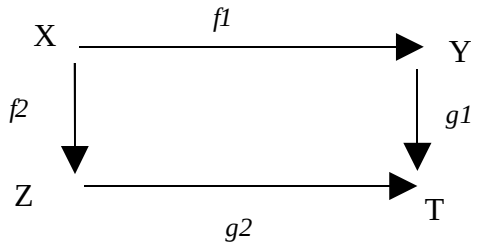

Figure 23: Commutativité du diagramme X-Y-Z-T

objet *T* relié par deux morphismes *g1* et *g2* tels que le diagramme *X-Y-T-Z* commute c'est-à-dire *f1 ; g1 = f2 ; g2*. Notons que *X* est le point de recollement du *pushout* (l'amorce du *pushout*).

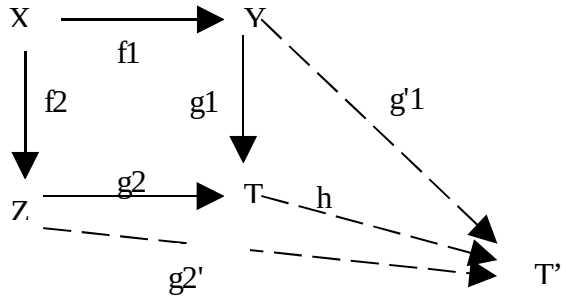

Figure24: L'opération pushout

De plus, pour chaque *T'* candidat potentiel relié à *Z* et à *Y* respectivement par *g'1* et *g'2*, il existe un morphisme *h* unique entre *T*

et *T'* tel que *g1 ; h = g'1* et *g2 ; h = g'2*. Ainsi, *T* est l'objet initial dans la catégorie de tous les candidats potentiels *T'*. Rappelons qu'un objet *o* d'une catégorie *C* est dit initial, si pour tout objet *X* de *C*, il existe une unique flèche de *o* vers *X*.

Lorsque cette opération est généralisée à plusieurs objets et à plusieurs morphismes, on parle de co-limite. Considérant un diagramme composé d'une collection d'objets *Xi* et de morphismes *fi*, sa co-limite est un objet initial *Z* et une famille de morphismes *gi* telle que pour tout *g1 : X1→ Z* et *g2 : X2→ Z*, on vérifie *g1 = f12 ; g2* pour *f12 : X1→ X2*.

3.3.7-Exemples

Commençons par un exemple simple : *pushout* dans la catégorie des ensembles. Dans cet univers, *pushout* exprime le partage des images des deux morphismes ayant une pré-image commune ; cet ensemble résultat est ensuite complété par l'union des éléments qui ne sont pas mis en relation par les morphismes. Son effet est illustré par le schéma de la figure 25.

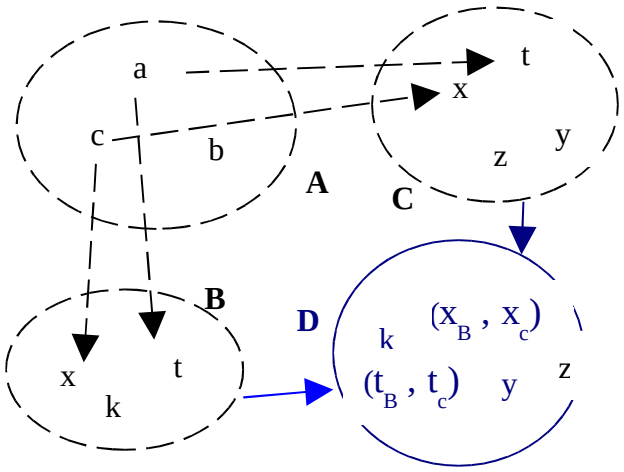

Figure 25: Pushout dans la catégorie des ensembles

Poursuivons avec un exemple plus informatique et qui concerne la relation d'héritage de la programmation par objets qui fait référence à la notion de partage de connaissances. Selon ce point de vue, imaginons une catégorie formée des classes *X*, *Y1*, *Y2* et *Z* pour les objets et des relations d'héritage *Y1* → *X* (*X* hérite de *Y1*), *Y2* → *X*,

$Z \rightarrow Y1$ et $Z \rightarrow Y2$ pour les morphismes étiquetés respectivement *f1*, *f2*, *g1* et *g2*. Selon la sémantique choisie de l'héritage, la classe *X* répète ou partage les attributs et méthodes de *Z*, ancêtres des classes *Y1* et *Y2*. Ainsi, si *f1 ; g1 = f2 ; g2*, le diagramme *X-Y1-Y2-Z* commute et la classe *X* ne possède qu'un seul exemplaire des attributs et méthodes de *Z* ; dans le cas contraire, les données de *Z* sont dupliquées.

En C++ par exemple, les deux sémantiques cohabitent avec, soit le mode duplication des données membres de *Z* dans la classe descendante *X*, soit la déclaration explicite de *Z* en tant que classe de base virtuelle (mot-clé *virtual*) des classes dérivées qui sont responsables de l'héritage répété, *Y1* et *Y2*, pour ne disposer en *X* que d'une seule copie de *Z* [119].

Ainsi pour le code C++ ci-dessous de la figure 26 :

```
class X {…}

 class  Y1 : virtual public    X   {…}

 class  Y2 : virtual public    X   {…}

 class  Z  : public Y1, public Y2  {…}
```

Figure 26: Mécanisme d'héritage virtuel en C++

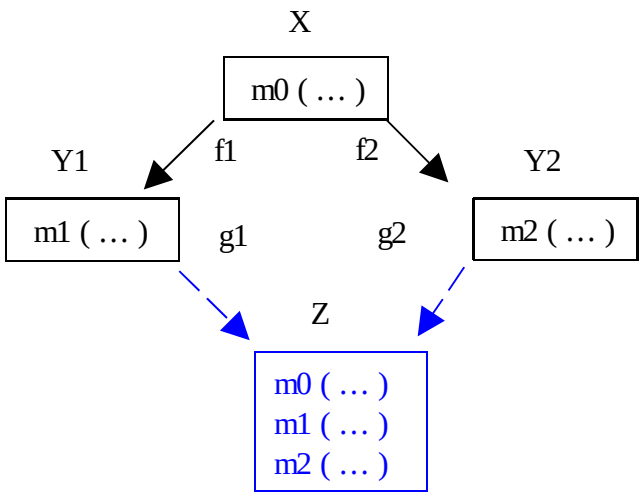

Figure 27: Construction de Z par pushout

le *pushout* construit l'objet *Z*, comme spécifié par la figure 27. Le code de *Z* provient exclusivement d'*Y1* et de *Y2*, selon les morphismes d'inclusion. La classe *Z* contient les méthodes d'*Y1* et celles d'*Y2*, sans redondance au point de recollement *X*. De plus, *Z* contient les méthodes spécifiques à *Y1* et à *Y2*.

Le dernier exemple montre comment l'opération *pushout* permet simplement d'établir une relation d'ordre partiel à partir des relations d'antisymétrie d'une part et de réflexivité et de transitivité d'autre part. En Specware, on considère pour cela séparément la spécification *specP1* de la figure 29 munie d'une opération (*op*) antisymétrique (*axiom*) et la spécification *specP2* de la figure 31 munie d'axiomes (*op* et *axiom*) exprimant la réflexivité et la transitivité. Reste ensuite à spécifier l'amorce de la figure 28 et les morphismes des figures 30 et 32 qui en découlent afin de pouvoir appliquer la somme amalgamée des deux spécifications qui définira une relation d'ordre.

```
specP0 = spec
type Z
op r0 : Z * Z -> Boolean
endspec
```

Figure 28: La spécification P0

```
specP1 = spec
type X
op r1 : X * X -> Boolean
axiom antisymétrie is fa (x : X, y : X)
      r1(y, x)&& r1(x, y) => x = y
endspec
```

Figure 29: La spécification P1

```
morphismP0_P1= morphism
specP0 -> specP1 {
      Z +-> X,
      r0 +-> r1 }
```

Figure 30: Le morphisme P0 vers P1

```
specP2 = spec
type Y
op r2 : Y * Y -> Boolean
axiom réflexivité is fa (y : Y) r2(y, y)
axiom transitivité is fa (x : Y, y : Y, z : Y)
      r2(x, y)&& r2(y, z)  =>  r2(x, z)
endspec
```

Figure 31: La spécification P2

```
morphismP0_P2 = morphism
specP0 -> specP2 {
      Z +-> Y,
      r0 +-> r2 }
```

Figure 32: Le morphisme P0 vers P2

Le *pushout* produit la relation d'ordre partiel de code décrit la figure 33. Conformément aux morphismes, les sortes correspondantes *X*, *Y* et *Z* sont confondues en une seule sorte. Il en est de même pour les opérations *r0*, *r1* et *r2*. Ces correspondances permettent de transférer les axiomes de *specP1* et de *specP2* vers la spécification résultante *pushout* tout en conservant leur validité.

```
spec
type {Z,X,Y}
op  {r0,r1,r2 } : X * X -> Boolean

axiom antisymétrie is  fa (x : X, y : X)
      r1(y, x) && r1(x, y) => x = y
axiom reflexivité is fa(x : Y)
       r2(x, x)
axiom transivité is fa(x : Y, y : Y, z : Y)
      r2(x, y) && r2(y, z) => r2(x, z)
endspec
```

Figure 33: Code de la relation d'ordre résultat du pushout

3.4-Construction du DSL unifié

Tout comme la catégorie des spécifications algébriques a été exploitée avec succès dans un contexte industriel [114] [120], nous souhaitons profiter de ce cadre formel pour construire notre DSL unificateur. Les spécifications algébriques ont en effet été exploitées afin de spécifier et développer des systèmes logiciels dédiés à la conception de composants avioniques faisant intervenir outre le logiciel, des aspects

relevant de la mécanique, de la thermique, de facteurs humains, etc. [114].

Le DSL unificateur pourra être établi par application de la fonction *pushout* sur des spécifications algébriques. Pour mettre en oeuvre ce mécanisme, nous avons besoin d'aborder deux points essentiels : la construction de l'amorce du *pushout* et la définition des morphismes à mettre en place entre l'amorce et la spécification de chaque objet, comme explicité à la figure 34.

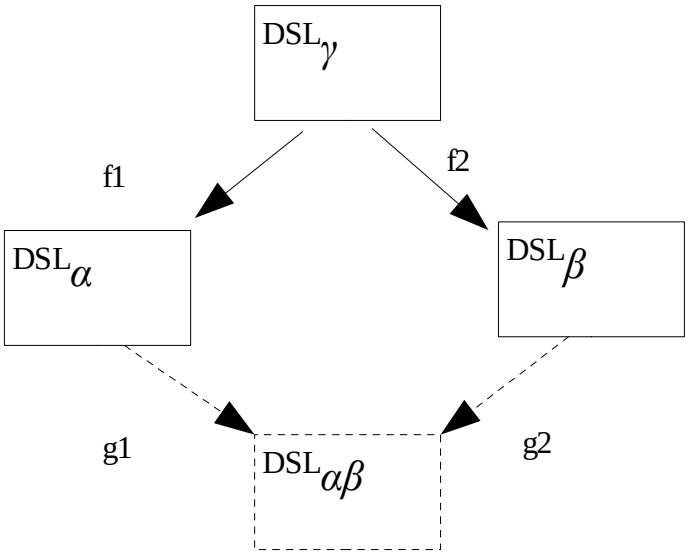

Figure 34: L'opération pushout appliquée aux DSL

Ainsi, soient *DSLα* et *DSLβ* deux spécifications de deux DSL, *DSLγ* la spécification de l'amorce commune à *DSLα* et *DSLβ, f1* et *f2* les morphismes définis entre l'amorce *DSLγ* et *DSLα* d'une part, et *DSLγ* et *DSLβ* d'autre part. Le *pushout* appliqué aux morphismes *f1* et *f2* produit le DSL *DSLαβ*, la spécification algébrique attendue, mais aussi les morphismes *g1* et *g2* qui prolongent ceux relatifs à *DSLα* et à *DSLβ*.

La spécification *DSLαβ* a pour signature celle de l'amorce *DSLγ* complétée par les sortes et les profils des opérations spécifiques à *DSLα* et à *DSLβ*. Le même raisonnement est applicable aux axiomes du résultat *DSLαβ*. On peut ainsi intuitivement paraphraser ce comportement en indiquant qu'il s'agit là « de la plus petite union » des concepts qui composent les DSL *α* et *β*.

Les éléments catégoriques à définir sont donc le DSL *γ* et les deux

morphismes *f1* et *f2*. Leur élaboration exige des efforts afin d'établir la correspondance entre les concepts d'α et de β. Notons que la démarche permet aussi de déterminer, par construction, les morphismes *g1* et *g2*. Ces morphismes joueront le rôle de traducteurs de composants écrits en α et en β. Ils sont à la base d'une conséquence importante caractérisant notre approche ; c'est ainsi que les experts d'un domaine pourront continuer à exploiter leurs outils pour leurs développements, ici les DSL α et β.

Mises à part les spécifications initiales α et β, les autres éléments γ, *f1* et *f2* du diagramme catégorique ne sont pas connus. Il convient donc d'être en mesure de construire l'amorce γ et les morphismes *f1* et *f2* qui en résulteront. Ce point fait l'objet des paragraphes suivants.

3.4.1-Construction de l'amorce et des morphismes

Intuitivement, le *pushout* est une construction qui consiste à coller deux objets de la catégorie le long d'un sous-graphe défini par un troisième objet que nous avons nommé amorce. Cette amorce établit la correspondance entre les concepts identiques aux deux objets. Dans le cas des spécifications de DSL d'une famille, l'amorce constitue le premier niveau d'abstraction de la famille. Il contient la spécification des sortes et opérations communes. La correspondance entre les concepts de l'amorce et ceux des langages à unifier est de fait interprétée via les morphismes du *pushout*.

Plus formellement, la spécification de l'amorce s'obtient à partir d'une relation d'équivalence entre les symboles $DSL\alpha$ et $DSL\beta$ préservant la propriété de prouvabilité d'un théorème de la source $DSL\gamma$ pour les théories cibles $DSL\alpha$ et $DSL\beta$. Considérant les deux morphismes $f1 : DSL\gamma \rightarrow DSL\alpha$ et $f2 : DSL\gamma \rightarrow DSL\beta$, le couple de symbole $(s,t) \in DSL\alpha \times DSL\beta$ est équivalent au symbole $u \in DSL\gamma$ ssi $f1\ (u) = s$ et $f2\ (u) = t$ et si toute équation $(s/u)E$ où u est substitué à s dans E de $DSL\alpha$ (respectivement $DSL\beta$) est équivalente à l'un des axiomes de $DSL\alpha$ (respectivement $DSL\beta$). Les équivalences des sortes, opérations et axiomes de $DSL\alpha$ et $DSL\beta$ ainsi définies par cette relation forment $DSL\gamma$, et les morphismes *f1* et *f2* sont, par construction, des morphismes de spécification.

La figure 35 résume le processus dans le cas déjà présenté de l'élaboration d'un ordre partiel à partir de relations qui le caractérisent.

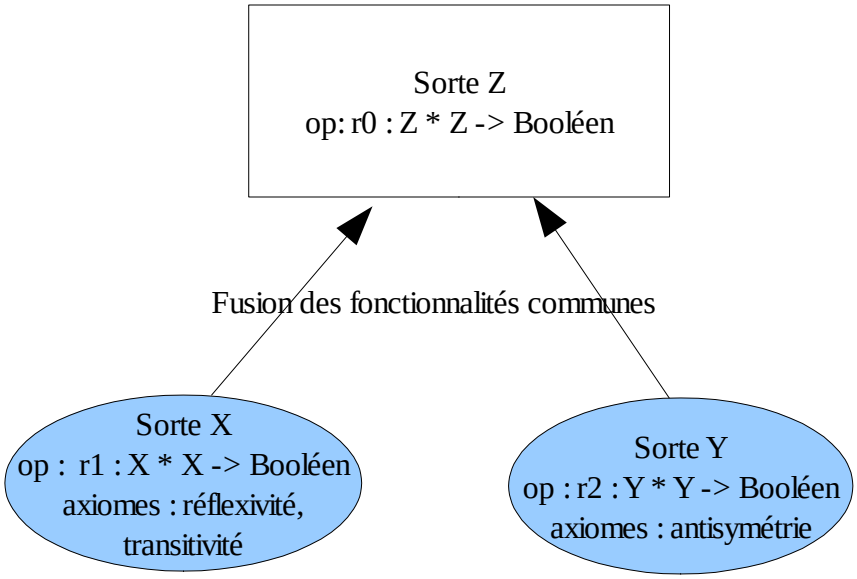

Figure 35: Construction de l'amorce

Les morphismes dépendent de l'amorce et des deux objets à fusionner. Il apparaît clairement que les morphismes se déduisent directement de l'élaboration de l'amorce. Les morphismes doivent être des morphismes de spécifications algébriques.

3.4.2- L'objet *pushout*

Une fois l'amorce *DSLγ* et ses morphismes *f1* et *f2* mis en place, il reste à appliquer l'opération *pushout* pour « coller » les deux DSL *DSLα* et *DSLβ*. Le *DSLαβ pushout* obtenu est une sorte « d'union minimale » des spécifications d'*α* et de *β* qui inclut la spécification factorisée *γ*, soit ses sortes, ses opérations et ses axiomes communs au renommage des sortes près, mais aussi les sortes, opérations et axiomes spécifiques à *α* et à *β*.

Les éléments spécifiques ne sont pas liés aux morphismes puisqu'ils n'ont pas de correspondance dans l'amorce. En effet, soit *e1* un élément du DSL *α* lié par le morphisme *f1* tel que *e1* n'a pas d'équivalent au sein du DSL *β*, alors, par définition, *e1* a un correspondant dans l'amorce *e0* tel que *f1 (e0) = e1*. Puisque *e0* appartient à l'amorce, *f2 (e0)* existe et appartient à *β* et vérifie *f2 (e0) = e2*, ce qui montre que l'image *e2* de *e0* par *f2* existe dans *β*. On en déduit que *e1* possède un correspondant en *β*, ce qui en contradiction avec l'hypothèse.

En résumé, la figure 36 montre un exemple d'éléments fusionnés comme *(α1, β3)* en provenance de *γ1* et des éléments spécifiques comme *α2 ou β2*.

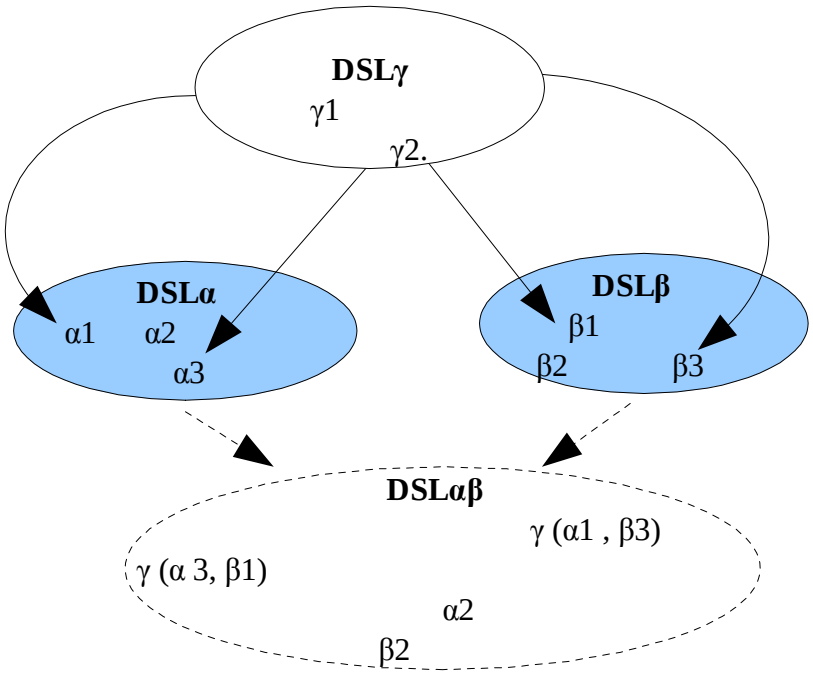

Figure 36: Construction du DSL *αβ*

De surcroît, le *pushout* détermine les morphismes qui permettent de transiter tout code d'un composant écrit en *α* ou en *β* en un code équivalent en *αβ*. Cette caractéristique, fort séduisante, nous permet d'entrevoir une conséquence de la démarche concernant l'interopérabilité des composants dès lors que le langage *αβ* peut constituer le pivot des différents langages de la famille. Ce point sera développé au chapitre suivant relatif à l'évaluation de l'approche.

3.4.3-Validation

L'objectif de ce paragraphe est de valider notre approche au travers d'une étude de cas, développée dans le cadre du projet RNTL DOMINO conjointement avec le CNES (Centre National d'Etudes Spatiales). Elle a pour source un problème concret : l'interopérabilité d'une famille de langages dédiés au développement de procédures opérationnelles du domaine spatial.

Les procédures opérationnelles définissent des processus mis en œuvre dans les centres de contrôle spatiaux afin d'effectuer un ensemble de services sur des éléments embarqués. Les directives de commande/contrôle décrivant ces services sont regroupées en opérations et s'expriment au travers de langages dits langages de procédures opérationnelles dont l'exploitation dépend le plus souvent de l'agence concernée. Comme par exemple le langage Pluto pour l'ESA, Elisa utilisé par EADS/ASTRIUM ou Stol développé par la NASA. Tous ces DSL partagent cependant le même caractère impératif ; ils intègrent des primitives d'ordonnancement des activités d'une opération et gèrent des télécommandes et des télémesures.

3.4.3.1- Les DSL *α et β*

Nous définissons dans ce qui suit deux fragments appelés *DSLα* et *β* inspirés de la norme Pluto [121]. Les deux DSL ont en commun la notion de procédure correspondante à une mission à réaliser caractérisée par un objectif spécifique à atteindre. Une procédure se décompose en trois parties distinctes : une précondition qui spécifie ses conditions d'activation, un corps constitué d'étapes conformes à la planification des activités de la mission et une postcondition qui garantit le bon achèvement de la procédure.

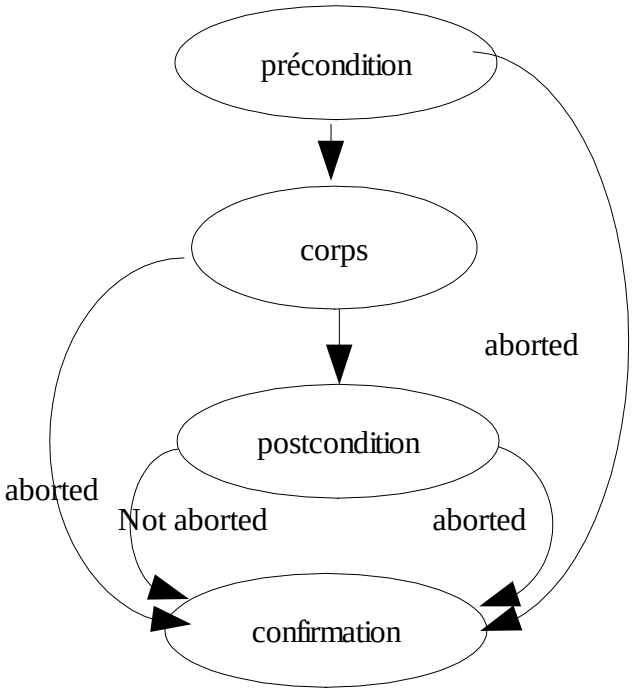

Figure 37: Flot d'exécution d'une procédure α ou β

Le flot d'exécution d'une procédure est initié par l'évaluation de sa précondition qui conduit à un échec de la procédure (*aborted*) dès lors que le contexte initial d'exécution n'est pas conforme à la situation attendue par la procédure. Dans le cas contraire, les différentes étapes du corps de la procédure sont exécutées séquentiellement. La procédure peut également échouer si sa postcondition n'est pas remplie (*aborted*). A l'issue de ces traitements, une confirmation est engagée avec l'expert afin de retranscrire le bon achèvement ou non de la procédure. La figure 37 schématise le flot d'exécution d'une procédure.

La précondition d'une procédure α ou β s'exprime par une conjonction de relations binaires que les variables du programme doivent vérifier vis-à-vis d'un environnement établi antérieurement au déroulement de la procédure. Le DSL α y ajoute la consommation d'événements externes (*wait*) selon une politique FIFO ; l'exécution de la procédure (*aborted*) est abandonnée chaque fois que l'absence d'occurrence d'un événement prévu est constatée. La procédure β filtre des télémesures liées à la survie du satellite et évalue les cas d'anomalies (*contingency*). Les figures 38 et 39 illustrent les données considérées par chacun des langages pour évaluer une précondition.

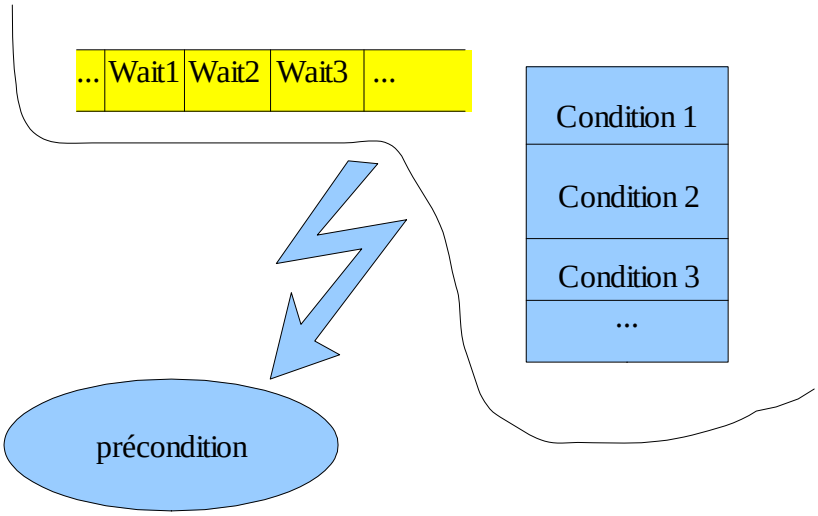

Figure 38: Précondition du DSL α

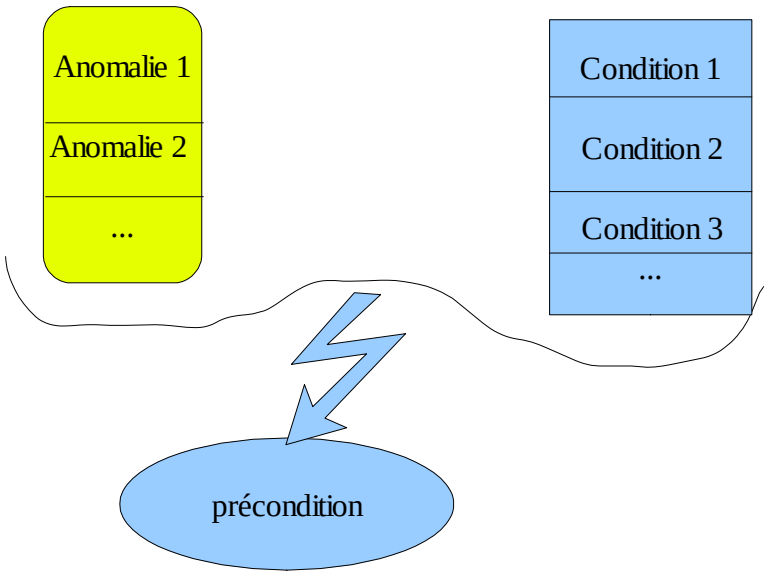

Figure 39: Précondition du DSL β

3.4.3.2- Construction du DSLαβ

```
alpha = spec
type  Status = | aborted |completed
type Event
type Checks  = List (Expression) * List (Event)
type Block
type Environment = Context * Queue
type Expression
type Context = List (Expression)
type Queue = List (Event)

op evalBlock : Checks *  Block * Environment  ->  Status * Environment
op evalBlockBody : Block * Environment ->   Status * Environment
op checkVariables : Checks * Environment -> Boolean
op checkExpressions : List (Expression) * Context -> Boolean
op waitStatements : List (Event) * Queue -> Boolean
```

Figure 40: Signature du DSL α

```
axiom a1 is fa (c : Checks, b : Block, e : Environment)
      ~ (checkVariables (c, e)) =>
      evalBlock (c, b, e) = (aborted, e)

axiom a2 is fa (c : Checks, b : Block, e : Environment)
      checkVariables (c, e) =>
      evalBlock (c, b, e) = evalBlockBody (b, e)

axiom a3 is fa (c : Checks, e : Environment)
      checkVariables (c, e) = checkExpressions (c.1, e.1)
      && waitStatements (c.2, e.2)

axiom a4 is fa (le : List (Expression), c : Context)
      checkExpressions (le, c) =
      (case le of
      | [] -> true
      | hd::tl -> member (hd, c) && checkExpressions (tl, c))

axiom a5 is fa (le : List (Event), q : Queue)
      waitStatements (le, q) =
      (case le of
            | [] -> true
            | hd::tl -> (case q of
                  | [] -> false
                  | q1::others -> if q1 = hd then
                  waitStatements (tl, others) else false))
```

Figure 41: Axiomes du DSL α

```
beta = spec
type  Status= | aborted |completed
type Conditions  = List (Expression) * List (Event)
type Statements
type Environment  = Context * Queue
type Expression
type Event
type Context = List (Expression)
type Queue = List (Event)
type ContingencyStatus = |nominal |warning | error

op evalModule : Conditions* Statements   * Environment
      ->  Status * Environment
op evalStatements : Statements *  Environment ->  Status
      * Environment
op evalPreconditionBody : Conditions * Environment
      -> Boolean
op evalBooleanExpressions : List (Expression) * Context
      -> Boolean
op evalContingency : Queue -> Boolean
op contingencyStatus : Event -> ContingencyStatus
```

Figure 42: Signature du DSL β

Conformément au mythe de la Tour de Babel, les programmeurs α et β sont censés parler la même et unique langue. En réalité, chaque groupe de spécialistes retranscrit dans son jargon les concepts de procédure, de précondition, d'événement, d'anomalie, etc. En pratique, une procédure est un bloc en α (Block) et un module en β (Statements), une précondition s'assimile à des tests de valeurs de variables pour un programmeur α (checkVariables) et prend la forme d'expressions booléennes en β (evalPreconditionBody). Parfois, un même vocable véhicule des sémantiques distinctes selon l'origine du groupe (Body, Expression et Statement par exemple).Conformément au mythe de la Tour de Babel, les programmeurs α et β sont censés parler la même et unique langue. En réalité, chaque groupe de spécialistes retranscrit dans son jargon les concepts de procédure, de précondition, d'événement, d'anomalie, etc. En pratique, une procédure est un bloc en α (Block) et un module en β (Statements), une précondition s'assimile à des tests de valeurs de variables pour un programmeur α (checkVariables) et prend la forme d'expressions booléennes en β (evalPreconditionBody). Parfois, un même vocable véhicule des sémantiques distinctes selon l'origine du groupe (Body, Expression et Statement par exemple).

```
axiom b1 is fa
(lc : Conditions, s : Statements, e : Environment)
      ~( evalPreconditionBody (lc, e)) =>
      evalModule (lc, s, e)   = (aborted, e
axiom b2 is fa
(lc : Conditions, s : Statements, e : Environment)
      evalPreconditionBody (lc, e) =>
      evalModule (lc, s, e) = evalStatements (s, e)
axiom b3 is fa (lc : Conditions, e : Environment)
      evalPreconditionBody (lc, e) =
      evalBooleanExpressions  (lc.1, e.1)
      && evalContingency (e.2)
axiom b4 is fa (le : List (Expression), c : Context)
       evalBooleanExpressions (le, c) =
      (case le of
            | [] -> true
            | hd::tl -> member (hd, c) &&
            evalBooleanExpressions (tl, c))
axiom b5 is fa (q : Queue)
       evalContingency (q) =
      (case q of
            | [] -> false
            | hd::tl ->  contingencyStatus (hd) = nominal)
```

Figure 43: Axiomes du DSL β

Dans les figures 40 et 42, nous introduisons les signatures partielles des deux DSL *α* et *β* relativement à l'interprétation d'une précondition. Outre les sortes associées aux entités syntaxiques citées précédemment, une procédure est dotée d'un contexte d'exécution (*Environment*) défini par une liste de valeurs de variables (*Context*) et par une file d'événements à consommer (*Queue*). Ainsi, les types *Environment = Context * Queue*, *Context = List (Expression)* et *Queue = List (Event)* ont trait aux machines abstraites *α* et *β* que nous avons voulues similaires. La validité de la précondition d'une procédure est codée dans les deux cas par le terme *aborted* ou *completed* du type énumératif *Status*. Par souci de lisibilité et de mise en correspondance des concepts, le code Specware des deux spécifications est comparable ligne à ligne.

L'axiome *a1* de la figure 41 relatif au DSL*α* exprime que le terme *evalBlock (c, b, e)* est évalué à *aborted* si la précondition *checkVariables (c, e)* d'un bloc *b* n'est pas valide. Dans le cas

contraire, l'évaluation du bloc d'instructions est engagée comme le stipule l'axiome *a2*. L'axiome *a3* s'assure notamment de l'adéquation du contexte *c* du bloc aux valeurs attendues des variables de la précondition. Dans ces axiomes, la notation *fa* correspond à « *for all* » et l'accès à une composante *i* d'un couple s'obtient par simple projection, notée par le suffixe « *i* ».

```
gamma = spec
type Status = | aborted |completed
type Event
type Conditions
type Steps
type Environment
type Expression
type Context = List (Expression)
type Queue

op evalProcedure : Conditions* Steps * Environment
      ->  Status * Environment
op evalMainBody : Steps *  Environment -> Status
      * Environment
op evalPrecondition : Conditions * Environment
      -> Boolean
op evalExpressions : List (Expression) * Context
      -> Boolean

axiom g1 is fa
(lc : Conditions, ls : Steps, e : Environment)
      ~ (evalPrecondition (lc, e)) =>
      evalProcedure (lc, ls, e) = (aborted, e)

axiom g2 is fa
(lc : Conditions, ls : Steps, e : Environment)
      evalPrecondition (lc, e) =>
      evalProcedure (lc, ls, e) = evalMainBody (ls, e)

axiom g3 is fa (le : List (Expression), c : Context)
      evalExpressions (le, c) =
      (case le of
            | [] -> true
            | hd::tl -> member (hd, c) &&
                  evalExpressions (tl, c))
```

Figure 44: Spécification du DSL γ

On retrouve à la figure 43 sensiblement le même comportement pour le DSL *β*, sauf en ce qui concerne l'évaluation d'une précondition qui met en œuvre, en lieu et place d'une attente d'événements (*wait*), un test de télémesure du satellite (*contingency*). Comme pour les

signatures des figures 40 et 42, les axiomes *a1* et *a2* sont comparables aux noms près à leurs homologues *b1* et *b2* ; seuls *a3* et *b3* divergent : *waitStatements* pour le DSL α et *evalContingency* pour le DSL β.

Il reste désormais à «factoriser» les DSL α et β en établissant deux morphismes de spécifications algébriques : d'une spécification amorce baptisée DSL γ vers la spécification algébrique DSL α, et de cette même amorce γ vers le DSL β. Cela revient à identifier les concepts correspondants en termes de sortes, profils et axiomes en α et β, et à les encapsuler en γ.

Par exemple, les concepts de bloc en α et de module en β coïncident car les profils des opérateurs *evalBlock* et *evalModule* sont identiques aux noms des sortes près (morphismes de signatures) et que les axiomes *a1* et *b1* sont transposables d'une théorie à l'autre (morphismes de spécifications algébriques). Nous appliquons la même stratégie pour les opérateurs *checkVariables* et *evalBlockBody* d'α vis-à-vis des opérateurs *evalPreconditionBody* et *evalStatements* de β, ce qui conduit à confondre les axiomes *a2* et *b2*.

Le point de divergence des deux spécifications a trait à la prise en compte de la consommation des événements attendus par une précondition : *waitStatements* pour l'un et *evalContingency* pour l'autre. Les profils de ces deux opérations diffèrent et par voie de conséquence les axiomes *a3* et *b3* qui les exploitent. Les équivalences des sortes, opérateurs et axiomes n'étant pas établies vis-à-vis de cette fonctionnalité, il ne peut y avoir unification de cette seconde partie de la conjonction en γ. On notera par contre que la première condition relative au contrôle des valeurs des variables est commune ; ce qui permet de fusionner ce sous-ensemble de la conjonction, à savoir *checkExpressions* et *evalBooleanExpressions*, ce qui conduit à l'axiome *g3* de la spécification γ.

Les figures 44 et 45 décrivent respectivement l'amorce *DSL*γ ainsi construite et les deux morphismes *DSL*$\gamma \rightarrow$ *DSL*α et *DSL*$\gamma \rightarrow$ *DSL* β qui en découlent. En Specware, rappelons que la notation *x* +-> *y* établit le morphisme entre la sorte ou l'opérateur *x* et son homologue *y* ; ce lien est étendu aux profils et axiomes exploitant *x* et *y*. On notera à la figure 45 qu'il n'est pas nécessaire de mettre en correspondance les sortes *Conditions* de γ et de β par le morphisme $\gamma \rightarrow \beta$ puisque ce nom est emprunté par les deux spécifications.

Notons que Specware fournit un moyen empirique d'identifier ces deux morphismes par le biais de l'opération *translate* qui retranscrit une spécification algébrique aux noms des sortes et opérations près.

Dans notre cas, les morphismes $\gamma \rightarrow \alpha$ et $\gamma \rightarrow \beta$ de la figure 45 sont

relatifs au processus de construction du langage *pushout αβ* qui en résulte, ce qui implique un même domaine γ pour les deux morphismes.

La spécification γ incluant par définition des axiomes, il convient de s'assurer, lors de l'établissement du morphisme $\gamma \rightarrow \alpha$ (respectivement $\gamma \rightarrow \beta$), que chacun des axiomes de la source γ,

```
gamma_to_alpha = morphism gamma -> alpha
      {Conditions +> Checks,
      Steps +> Block,
      evalProcedure +> evalBlock,
      evalPrecondition +> checkVariables,
      evalMainBody +> evalBlockBody,
      evalExpressions +> checkExpressions}
gamma_to_beta = morphism gamma -> beta
      {Steps +> Statements,
      evalProcedure +> evalModule,
      evalPrecondition +> evalPreconditionBody,
      evalMainBody +> evalStatements,
      evalExpressions +> evalBooleanExpressions}
```

Figure 45: Morphismes $\gamma \rightarrow \alpha$ et $\gamma \rightarrow \beta$

traduits en remplaçant tout élément de la cible α (respectivement β) par son image, soit un théorème de la théorie cible α (respectivement β). De cette exigence découle par construction la spécification algébrique du point de recollement γ spécifiée à la figure 44 contenant non seulement les sortes et opérateurs communs à α et β, mais aussi les axiomes partagés.

Tous les concepts sont désormais mis en place pour que Specware calcule la somme amalgamée des deux DSL α et β. Cette spécification, décrite en partie par le code de la figure 46, énumère les sortes, les opérateurs et les axiomes qui unifient ceux d'α et β, mais inclut aussi ceux qui leur sont spécifiques. La partie commune agrège les axiomes communs *a1* et *b1* en *g1*, *a2* et *b2* en *g2*, *a4* et *b4* en *g3* ; elle comprend aussi tous les éléments communs non spécifiés par les morphismes, comme par exemple les sortes *Expression*, *Event* et *Environment* pour notre exemple. A cette partie commune sont rajoutés les éléments de spécification spécifiques comme les axiomes *a3* et *b3*. Le code complet de cette étude de cas est fourni en annexe.

On notera que le calcul Specware conserve la trace des morphismes en assimilant les noms de sortes et d'opérateurs comme dans la notation {*Checks*, *Conditions*} et qu'un même raffinement de sorte en α et β

```
spec
type {Checks, Conditions} = List (Expression) * List
(Event)
type {Block, Statements, Steps}
…
op {evalBlock, evalModule, evalProcedure} :
Conditions * Steps * Environment -> Status * Environment
op waitStatements : List (Event) * Queue -> Boolean
op evalContingency : Queue -> Boolean

axiom g1 is fa (lc : Conditions, ls : Steps, e :
Environment)
      ~ (evalPrecondition (lc, e)) =>
      evalProcedure (lc, ls, e) = (aborted, e)

axiom g3 is …

axiom a3 is fa (c : Checks, e : Environment)
checkVariables (c, e) =
      checkExpressions (c.1, e.1) &&
      waitStatements (c.2, e.2)
…
axiom b3 is fa (lc : Conditions, e : Environment)
      evalPreconditionBody (lc, e) =
      evalBooleanExpressions (lc.1, e.1)&&
      evalContingency (e.2)
…
endspec
```

Figure 46: Spécification du DSL pushout αβ

est factorisé en *αβ*, comme pour *{Checks, Conditions} = List (Expression) * List (Event).*

3.5-Conclusion

Nous avons décrit dans ce chapitre une approche pour construire le DSL minimal offrant toutes les fonctionnalités sémantiques des membres d'une famille de DSL et par là même, mis en place une interopérabilité au sein de cette famille. La méthode choisie ici se fonde sur des constructions théoriques telles que la sémantique algébrique et la théorie des catégories qui assurent un socle formel solide à la démarche.

Cette étude a été validée en fin de chapitre en cherchant à fusionner deux DSL α et β relevant des procédures opérationnelles du domaine spatial. Il est à noter que notre proposition ne se cantonne pas à une

étude formelle mais qu'elle inclut aussi une mise en œuvre logicielle via l'environnement Specware et l'écriture des deux spécifications algébriques associées aux deux DSL d'origine. C'est ainsi que l'opération *pushout* des deux morphismes de spécifications algébriques projetant une amorce commune γ vers les spécifications α et β nous a permis de fusionner les fonctionnalités des deux DSL sans redondance et d'obtenir automatiquement la spécification $\alpha\beta$.

Chapitre IV

4-Évaluation de l'approche

Les chapitres précédents ont détaillé notre approche pour résoudre le problème de l'interopérabilité entre plusieurs langages d'une même famille. Notre approche se base sur l'utilisation de techniques catégoriques appliquées aux spécifications algébriques de langages. Ces techniques nous permettent d'obtenir de façon automatique la spécification algébrique d'un langage unificateur, ainsi que des morphismes de traduction nous permettant le passage depuis les langages initiaux vers ce langage pivot.

Dans ce chapitre, nous essayons d'évaluer notre approche. Nous commençons par analyser trois points :

- l'impact de notre approche sur certains résultats de vérification établis au sein des langages initiaux ;
- l'effet sur l'interopérabilité des composants spécifiés dans les langages originaux ;
- l'effort requis par une bonne définition du langage amorce.

En premier lieu, nous analysons la possibilité de conserver les propriétés écrites dans un DSL, sous la forme de théorèmes, lors du passage au langage unificateur. Nous allons montrer que notre approche nous permet de conserver les propriétés d'un langage, et de les transférer vers le langage unifié automatiquement en utilisant les morphismes de traduction. Un exemple vient appuyer notre raisonnement et montre, qu'un théorème écrit dans le langage DSLα, reste valide dans le contexte de l'unificateur DSL$\alpha\beta$.

Nous amenons ensuite notre discours vers le niveau de spécification concrète des composants, dans le cas où celle-ci est effectuée dans un des langages de la famille. Dans cette partie, nous analysons comment on peut obtenir l'interopérabilité en exploitant le langage unificateur proposé par notre approche.

Troisièmement, nous proposons une méthodologie pour construire le langage unificateur de deux DSL. En plus de cette méthodologie, une heuristique est définie afin d'automatiser la tâche de construction de l'amorce du *pushout*. Notons que cette méthodologie a été validée dans le contexte de notre cas d'étude sur les langages DSLα et DSLβ.

Enfin, ce chapitre se termine par les avantages et les inconvénients de notre approche.

4.1-Propriétés sur les langages

Dans le contexte de la définition d'un langage on peut établir des propriétés dérivant de sa définition. Ces propriétés sont satisfaites indépendamment de ses utilisations.

D'autres propriétés peuvent aussi être définies au niveau des spécifications de programmes ou de composants effectuées en utilisant un langage.

À ce stade, nous faisons la distinction entre les propriétés établies au niveau de la définition du langage et des autres établies au niveau de l'application.

Afin d'expliquer la différence entre ces deux types de propriétés, nous prenons comme exemple les deux propriétés suivantes d'un langage langage $\mathcal{L}$ des procédures opérationnelles :

- « *P1* (propriété langage) : dans le langage $\mathcal{L}$, on n'arrive jamais à exécuter le corps d'une procédure si la pré-condition associée n'est pas satisfaite ».
- « *P2* (propriété programme) : dans une spécification concrète $\mathcal{S}$ du programme *P* écrit en $\mathcal{L}$, la boucle *B* n'est jamais exécutée plus de 3 fois ».

La propriété *P1* fait référence seulement à la définition du langage $\mathcal{L}$. De ce fait, si elle est établie, elle sera satisfaite par toutes les spécifications qui exploitent $\mathcal{L}$.

La situation est différente dans le cas de la deuxième propriété qui dépend non seulement de la définition du langage $\mathcal{L}$, mais aussi de la spécification $\mathcal{S}$. Une telle propriété, si elle est établie, sera satisfaite seulement par $\mathcal{S}$.

Par conséquent, *P1* est aussi une propriété sur la spécification $\mathcal{S}$, par contre *P2* n'est pas une propriété sur $\mathcal{L}$. Pour commencer, nous nous concentrons sur les propriétés au niveau langage. Ces idées ont aussi été présentées dans le papier [122].

4.1.1-Transposition de propriétés

Dans le contexte des langages décrits en termes de spécifications algébriques, on peut définir des propriétés, qui décrivent des relations qui existent entre des constructions du langage. Ces propriétés se présentent sous la forme de prédicats logiques dans le contexte de la spécification algébrique.

Prouver une propriété dans ce contexte revient à démontrer que le prédicat qui lui correspond est un théorème.

Soit $\mathcal{P}$, une propriété sur un langage dédié $\mathcal{L}$, exprimée sous forme équationnelle, comme un prédicat.

Si ce prédicat est un axiome du langage $\mathcal{L}$, alors il est considéré valide sans besoin d'une preuve.

Si le prédicat n'est pas un axiome, il peut être déductible des axiomes du langage $\mathcal{L}$. Dans ce cas, une preuve est exigée afin de démontrer la validité de la propriété $\mathcal{P}$. Une manière courante de prouver une propriété, consiste à montrer que le prédicat lui correspondant représente un théorème, donc, qu'il est déductible de l'ensemble des axiomes qui caractérisent le langage $\mathcal{L}$.

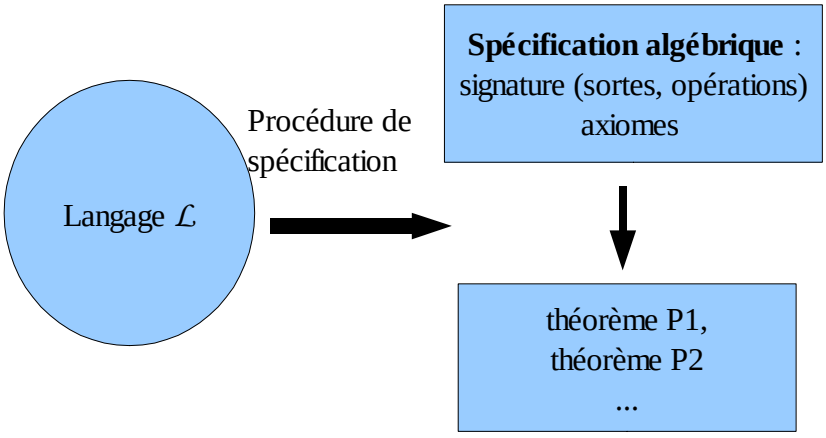

Figure 47: La spécification d'un langage de programmation

Les prédicats qui ne peuvent être prouvés correspondent à des propriétés dont la validité n'a pu être établie. Ainsi, dans le cas de la spécification algébrique d'un langage, les propriétés établies se présentent sous la forme de théorèmes.

Dans la catégorie des spécifications algébriques, comme détaillé en [123], le *pushout* transporte ces propriétés dans le contexte unificateur en tant que théorèmes.

On conclut ainsi, que le *pushout* permet d'un côté de traduire la propriété et d'un autre qu'elle sera un théorème du nouveau contexte. En effet, un morphisme de spécifications algébriques obtenu comme résultat du *pushout*, permet de transposer un théorème de son domaine (qui correspond à la spécification d'un langage initial) vers un théorème du co-domaine (qui correspond à la spécification du langage unificateur qui vient d'être calculée).

Dans le développement logiciel, les techniques de vérification sont principalement utilisées dans des domaines critiques, particulièrement lorsque la certification est souhaitée. Il est intéressant de noter que la certification impose en effet de lourdes contraintes par rapport à la technique de validation utilisée. En particulier, la réutilisation des propriétés établies uniquement sur la base de résultats mathématiques n'est pas toujours considérée comme acceptable. Le processus de certification peut ainsi exiger de fournir la preuve qu'une propriété $\mathcal{P}$ conserve son statut de théorème dans le contexte unifié. Dans ce contexte, il est intéressant de souligner que reproduire la preuve des propriétés originelles dans le nouveau contexte est fortement facilité par l'expérience déjà acquise au niveau du langage initial. Dans le cas où le théorème $\mathcal{P}$ a été prouvé en utilisant un assistant de preuve, tels Isabelle [112] ou Snark [124] [125], prouver le théorème dans le contexte unifié bénéficie directement de l'expérience gagnée lors de la preuve de $\mathcal{P}$. Concrètement, cela veut dire qu'on va probablement utiliser les mêmes tactiques de preuve et des lemmes équivalents.

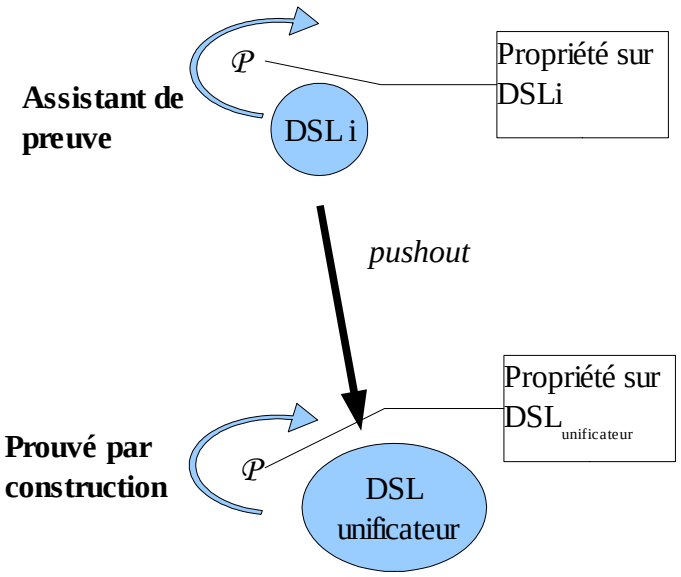

Figure 48: Transfert d'une propriété vers le langage unificateur

Dans la Figure 48, nous illustrons le transfert de propriétés. Nous considérons pour cela une propriété $\mathcal{P}$ prouvée à l'aide d'un assistant de preuve dans le contexte de DSLi, un des langages de la famille.

L'application *pushout* nous permet non seulement d'obtenir la spécification algébrique du langage unificateur, mais aussi de transposer notre propriété dans ce nouveau contexte.

Dans [109], les auteurs montrent que les propositions correspondent à des théorèmes établis dans le contexte des objets qui participent à la somme amalgamée, sont transformées en propositions qui restent des théorèmes dans le contexte amalgamé. Ces techniques sont disponibles grâces aux morphismes de spécifications qui maintiennent la preuve dans la spécification du co-domaine.

Ainsi, dans le contexte du DSL unificateur, $\mathcal{P}$ reste un théorème.

Ce résultat nous permet de réutiliser dans le contexte unifié des propriétés établies dans le contexte des langages de la famille. Il n'est pas nécessaire de prouver une propriété transférée vers l'unificateur.

4.1.2-Outils de preuve

Le résultat énoncé dans la section précédente part de l'hypothèse qu'on est arrivé à montrer la validité d'une propriété. Cette étape peut s'avérer difficile et fait souvent appel à des assistants de preuve.

Même si cela dépasse un peu la portée de cette thèse, nous allons nous arrêter un peu sur ces outils, afin de mieux expliquer comment ils s'utilisent et comment ils s'intègrent à l'ensemble des outils que nous utilisons.

Pour prouver la validité d'une propriété, on peut faire appel à des assistants de preuve. Leur rôle en ce qui nous concerne est d'aider à prouver les propriétés sur un DSL. Ces propriétés sont écrites sous la forme de propositions logiques. Les prouver consiste à montrer qu'il s'agit de théorèmes, i.e. qu'elles sont déductibles de l'ensemble des axiomes.

Plusieurs assistants de preuve sont disponibles pour prouver un théorème écrit sous la forme d'une proposition logique. Comme nous l'avons signalé précédemment, pour expérimenter notre approche, nous avons utilisé l'outil Specware qui permet (entre autres) de calculer des *pushouts* de spécifications algébriques.

L'outil Specware intègre une passerelle vers des assistants de preuve. En effet, il dispose d'une interface qui nous permet de transformer automatiquement les codes d'une spécification algébrique, écrites en Specware, en code intégré dans l'assistant de preuve Isabelle. Ce qui facilite l'utilisation conjointe des outils Specware et Isabelle.

Isabelle est un assistant de preuve qui offre la possibilité de spécifier un système et de le vérifier. Dans la partie expérimentale, nous avons utilisé cet assistant doté notamment de la logique HOL (*Higher-Order Logic*) [126] [97]. Ce démonstrateur nous a permis de démontrer les propriétés dans notre contexte.

La nouvelle version de Specware 4.2.2 rend disponible une passerelle vers l'assistant de preuve Isabelle, ce qui facilite l'utilisation de cet outil. Cette interface permet à Specware de se décharger des preuves de théorèmes. Son rôle essentiel est de traduire les composants de Specware en composants Isabelle, afin de bénéficier de l'environnement de preuve de ce dernier.

La traduction du Specware vers Isabelle est en partie une transformation implicite. La traduction spécifie la translation des sortes, des opérations, et des axiomes. Cela nous permet de démontrer

un théorème sous Isabelle en conservant le code existant Specware.

Il est à noter que la traduction n'est pas toujours systématique. Quelques fois, par exemple dans le cas des fonctions récursives, l'utilisateur est obligé d'intervenir pour modifier ou ajouter quelques lignes de commandes, afin de rendre la théorie translatée valide sous Isabelle, notamment en ce qui concerne l'arrêt de la récursivité [127] [128]. Malgré cette étape parfois fastidieuse, la passerelle Specware permet de spécifier les tactiques de preuve.

4.1.3-Exemple

Pour illustrer la possibilité d'exploiter le transfert de propriétés établies au niveau des langages d'une famille vers le langage unificateur, nous nous concentrons sur les langages DSLα et DSLβ définis dans le chapitre précédent.

Dans cet exemple, on considère une propriété du DSLα. Nous allons exprimer cette propriété comme une proposition et nous allons montrer qu'il s'agit d'un théorème sur l'ensemble d'axiomes du DSLα.

Ensuite, en appliquant *pushout*, nous montrerons que l'on retrouve cette propriété transposée dans le contexte du langage unifié DSL$\alpha\beta$. Les résultats de la théorie des catégories nous assurent que cette nouvelle propriété est un théorème dans le contexte unificateur. Cependant, pour des raisons d'illustration, nous allons utiliser l'assistant de preuve Isabelle pour s'assurer de sa validité.

Considérons à titre d'exemple la propriété $\mathcal{P}$ dans le contexte du langage DSLα :

« *si les événements à vérifier par waitStatements sont en plus grand nombre que la taille de la file d'attente Queue de la procédure, la précondition checkVariables échoue et le statut de la procédure est positionné à aborted* ».

Cette propriété est exprimée en DSLα par le biais des théorèmes *wait* et *check* de la figure 49.

Dans notre expérimentation, ce résultat a pu être établi à l'aide de l'assistant de preuve Isabelle, disponible grâce au pont qui a été développé entre Specware et Isabelle. Ainsi, on a pu prouver que les propositions *wait* et *check* constituent des théorèmes dans le contexte du DSLα.

Lors du passage vers le contexte du langage unifié DSL$\alpha\beta$, ces

```
theorem wait is fa (le : List(Event), q : Queue)
 length (le) >length (q) => ~ (waitStatements (le, q))

theorem check is fa (le : List(Event), q : Queue)
(fa(l : List(Expression), c : Context )
  length (le) > length (q) =>
     ~(checkVariables ((l,le),(c, q))))
```

Figure 49: Théorèmes wait et check d'une précondition DSLα

propriétés s'expriment exactement de la même façon. A noter qu'à cause de la variation de l'ensemble des axiomes, la validité de cette proposition est remise en cause. Cependant, en prenant en compte la synonymie des termes *checkVariables* et *evalPrecondition* établie par le morphisme du DSL amorce vers le DSLα, la propriété $\mathcal{P}$ du langage DSLα est prouvée en DSLαβ, selon les mêmes stratégies de preuve.

Similairement, tout théorème dans la spécification algébrique associée aux DSLα ou DSLβ est automatiquement préservée par construction dans le DSLαβ *pushout*.

4.2-Interopérabilité des composants

Dans ce paragraphe, nous passons du raisonnement au niveau de la définition d'un langage vers le niveau de ses composants. En effet, jusqu'à présent, nous avons toujours considéré les définitions de langages, à travers leurs spécifications algébriques. En se rapportant à la hiérarchie des niveaux d'abstraction spécifiques à l'ingénierie des modèles [129], nous avons toujours oeuvré jusqu'à présent au niveau M2. Dans ce paragraphe, nous allons analyser comment les résultats d'unification obtenus peuvent être exploités au niveau applicatif des spécifications concrètes de composants, donc au niveau M1. Ce travail a été exposé dans le papier [130].

4.2.1-Intégration de composants hétérogènes

Dans cette section, on s'intéresse à l'interopérabilité de composants écrits dans des langages hétérogènes.

Dans le cas de composants écrits dans un langage unique, plusieurs approches nous permettent d'assurer l'interopérabilité, à travers des modèles de composants, par exemple qui utilisent une

machine virtuelle commune [131], qui collaborent exclusivement à travers des échanges de messages, etc. La question de l'interopérabilité des composants devient encore plus complexe pour des composants écrits en utilisant plusieurs langages.

En effet, lorsque les composants sont décrits dans un même langage, une fonctionnalité d'un composant peut être appelée à partir d'un autre composant, sans la nécessité de nouveaux outils ou de couches intermédiaires, par exemple en utilisant toujours le même compilateur.

Dans le cas des composants hétérogènes, les composants écrits dans différents langages doivent être soit traduits dans un nouveau langage, soit transiter par une couche intermédiaire, car il n'existe plus la possibilité d'avoir un compilateur unique.

Notre idée pour assurer l'interopérabilité de ces composants est de profiter de l'approche d'unification des langages au niveau sémantique et d'appliquer les morphismes obtenus vers l'unificateur comme des traducteurs de spécifications.

Ceci résout le problème de l'interopérabilité généré par l'hétérogénéité des spécifications. A partir du moment où les différentes spécifications sont décrites dans un même langage, on applique le mécanisme de composition qui est le plus adapté, sans se préoccuper de l'hétérogénéité. Cela peut passer par l'existence d'un environnement commun, par des échanges de services spécifiés au niveau des interfaces, etc.

Notre élément clé pour gérer l'interopérabilité des composants hétérogènes réside dans les morphismes de traduction. Notons que ces morphismes sont générés automatiquement par la fonction *pushout*, en plus de la traduction vers l'unificateur d'un langage appartenant à la famille.

Dans ce qui suit, nous formalisons cette idée : on note un composant par le symbole C muni de deux indices : un premier indice (indice de symbole) décrit son langage de développement et le second (sa puissance) décrit son numéro.

Soit un ensemble de m composants, spécifiés en utilisant n langages dédiés où n est la cardinalité d'une famille. Dans ce cas, le nombre de composants est obligatoirement plus grand que celui des langages ($m \geq n$) :

$$C^{1}_{DSL_{k_1}}, \cdots, C^{m}_{DSL_{k_m}}, \quad \text{avec } k_1, \ldots, k_m \in \{1, \ldots, n\} \tag{1}$$

Nous nous appuyons sur les morphismes permettant le passage depuis les différents langages dédiés de la famille vers le langage unificateur. Ces morphismes sont obtenus en utilisant l'approche détaillée à la section antérieure et nous permettent de traduire les composants de (1) dans le langage unificateur. Nous obtenons ainsi un ensemble de composants exprimés dans ce même langage :

$$C'^{1}_{DSL^{\mu}}, C'^{2}_{DSL^{\mu}}, \quad \cdots, C'^{m}_{DSL^{\mu}} \tag{2}$$

En (2), on note par DSL^{μ} le langage unificateur de la famille {*DSL1*, *DSL2*, …, *DSLn*} et chaque composant est obtenu en appliquant le morphisme de traduction *gj* depuis *DSLj* vers $DSL\mu$ obtenu de manière automatique.

$$C'^{i}_{DSL^{\mu}} = g_j\left(C^{i}_{DSLj}\right) \quad \text{avec } i \leq m \text{ et } j \leq n \tag{3}$$

Nous arrivons ainsi à un ensemble de composants décrits en utilisant le même langage, ce qui permet leur utilisation conjointe.

Dans un article de référence de 1999 [113], Schneider et Nierstrasz présentent une classification des différentes techniques permettant l'utilisation conjointe de différents composants. Dans notre approche, nous nous concentrons sur les problèmes d'interopérabilité des composants issus de différents langages. Dès lors que nos composants hétérogènes ont été traduits dans un même langage unificateur, les techniques conventionnelles de composition ne s'avèrent pas nécessaires.

De plus, il nous semble important de laisser les utilisateurs libres de leurs choix par rapport à ces techniques, qui ont chacune leurs avantages et inconvénients qui dépendent d'autres critères que celui de leur langue d'origine. Si l'on considère le contexte des procédures opérationnelles, la composition peut effectivement faire intervenir des langages de script [113], mais ce choix fait

aussi intervenir d'autres aspects que la nature hétérogène des composants.

4.2.2-Exemple

Dans ce paragraphe nous illustrons notre solution pour l'interopérabilité de composants, en considérant un exemple. Soient deux composants écrits respectivement dans les langages DSLα et DSLβ. L'interopérabilité est accomplie si nous pouvons utiliser des services (appeler une méthode, ou autre) d'un composant à partir d'un autre.

Considérons trois composants *P1*, *P2* et *P3* décrits en DSLα et DSLβ. Comme dans nos langages l'unité de base est la procédure, nous allons parler en termes de procédures.

Dans la figure 50, nous définissons en langage α la procédure *P1* `ReplacesEquipements`. Cette procédure est composée d'une précondition et d'un corps principal *main*. *P1* prévoit deux types de communication en deux modes différents représentés par les lignes de commande (3) et (6).

```
(1) Procedure ReplacesEquipments
  (2) Precondition
    (3) Wait RightTemperature
  (4) End precondition
  (5) Main
    (6) Initiate and confirm SetHT with voltage :=2000v
  (7) End main
(8) End Procedure
```

Figure 50: La procédure P1 spécifiée en DSLα

1. A la ligne (3), la procédure attend la réception du message `RightTemperature`.
2. A la ligne (6), la procédure invoque la procédure *SetHT*.

La Figure 51 donne la description d'une procédure *P2* écrite en DSLα. Dans cette procédure on envoie à la ligne (6) le message *RightTemperature*, qui pourra être intercepté par la procédure *P1*.

```
(1) Procedure MonitorTemperature
      (2) Precondition ExternalTemperature > 60
      (3) End precondition
(4)   Main
(5)       Raise event RightTemperature
(6)   End main
(7) End procedure
```

Figure 51: La procédure P2 en langage α

La figure 52 décrit le composant *P3*, cette fois développé dans le langage DSLβ. Ce composant offre le service *SetHT* demandé par la procédure *P1*.

On constate que ces trois procédures sont décrites en utilisant deux langages différents. Pour assurer leur interopérabilité, on a besoin :

1. d'utiliser un mécanisme de composition qui permet des échanges entre les différents composants ;
2. de résoudre le problème généré par leur hétérogénéité, car même s'ils sont similaires, ces langages sont distincts, comme spécifié au chapitre 3.

```
(1) Procedure SetHT With voltage, equipment
 (2) preconditionBody Htoff, on equipment
      End preconditionBody
(3) Statement
 (4) Initiate and confirm equipment on
 (5) Initiate and confirm connect WithVoltage
(6) End statement
 (7) postcondition HT on (Voltage) on equipment
      End postcondition
(8) End procedure
```

Figure 52: La procédure P3 en langage β

En ce qui concerne le premier point relatif au mécanisme de composition, une manière de permettre les échanges entre des composants différents, dans le cas de notre exemple, est de faire appel à une machine abstraite commune. En effet, si l'on suppose que les procédures *P1* et *P2* utilisent le même environnement de communication, le message `RightTemperature` généré par *P2*, arrivera dans la file d'attente de *P1*, et pourra être utilisé par la procédure *P1*.

Le deuxième point fait référence à la partie plus intéressante par rapport à notre approche et concerne le fait que les trois procédures sont décrites en utilisant deux langages différents. C'est ainsi le cas de la communication nécessaire entre *P1* et *P3* pour l'appel *SetHT.*

La procédure *P3* est composée d'une clause `preconditionBody`, de `statements` et d'une `postcondition`. Deux variables sont définies et initialisées dans *P3* afin de manipuler des équipements du domaine, accompagnées par une postcondition qui contrôle le résultat obtenu. Ce qui pose problème pour effectuer cet appel est le fait qu'on ne dispose pas de compilateur commun et qu'il n'existe pas d'environnement d'exécution unique car les langages de base des deux spécifications sont distincts.

En utilisant notre approche de passage vers le contexte du langage unificateur, au niveau de chacun des composants, donc de *P1* et de *P3*, le problème de l'hétérogénéité ne se pose plus.

En passant au langage unificateur unique, nous pouvons traiter les différents composants comme s'ils étaient écrits dans un seul langage.

La construction d'un compilateur du langage unificateur peut raisonnablement s'envisager en utilisant ASF+SDF [98]. Cette phase de construction d'un compilateur prendra en entrée les morphismes de traduction obtenus en tant que résultats du *pushout*. Ces morphismes conservent en effet la trace des éléments définissant le code venant d'un DSL de famille. De plus, ces morphismes dans notre approche constituent autant d'outils en charge de traduire la relation entre un DSL et l'unificateur.

Pour la complétude de la discussion, notons qu'un autre problème peut provenir du format différent de représentation des données dans le contexte des différents langages. Dans notre cas, il s'agit de l'appel de la procédure *P3* à la ligne (6) de *P1.* Le paramètre de cet appel est de type `Voltage`, qui risque de ne pas être représenté de la même manière dans les deux contextes.

Notons que ce problème ne vient pas de l'hétérogénéité des spécifications des langages, mais des différences qui peuvent exister au niveau de la représentation des données dans les différents contextes. Ainsi, tout mécanisme de composition devra résoudre ce type de problème.

Une solution classique dans une telle situation est d'utiliser ces données en les représentant de façon standardisée, ou de prévoir des mécanismes pour passer à ces représentations. ASN.1 [43] propose

par exemple un standard pour la représentation des données.

En conclusion, l'utilisation des traducteurs vers le langage unificateur, obtenus à partir des morphismes de traduction, permet aux utilisateurs de conserver leurs environnements initiaux. Les morphismes de traduction prennent en charge la translation des codes de ces composants, vers des composants en un code unificateur standard, au sein d'un contexte unique.

4.3-Méthodologie de construction du langage unificateur

Dans cette section, nous proposons une démarche de processus de construction d'un langage unificateur de deux langages DSL Y_1 et Y_2. Nous allons mettre en évidence les étapes et les techniques à suivre pour obtenir le langage unificateur *Z*. Nous examinons dans quelle mesure les étapes que nous allons identifier sont automatiques ou automatisables.

En ordre chronologique, les différentes étapes à suivre sont :

1. la spécification formelle, en tant que spécification algébrique, des langages initiaux ;
2. la définition d'un langage amorce ;
3. la construction des morphismes initiaux ;
4. la génération du langage unificateur.

Avant de détailler ces étapes, nous nous intéressons de plus près à la construction de l'amorce. Dans cette étape il faut identifier et extraire les éléments de l'amorce des spécifications initiales. Ce travail ne peut pas s'accomplir de façon intuitive, ce qui exige une assistance à la construction.

4.3.1-Construction de l'amorce

Jusqu'à présent, nous avons considéré que le langage amorce était défini. Dans cette section, nous allons définir une heuristique de construction. Par rapport aux quatre étapes mentionnées auparavant, cette heuristique couvre la deuxième étape du processus de construction du langage unificateur.

Afin d'établir le langage unificateur, nous avons besoin d'un langage amorce. Intuitivement, c'est à ce niveau que l'on spécifie les parties communes des différents langages à unifier. L'amorce est alors une première abstraction de la famille. On peut objecter à cette vision que la famille de DSL à traiter n'est pas le résultat d'un tel raffinement qu'il faudrait fournir a posteriori.

Une autre approche permet de répondre à cette objection et de proposer une méthode constructive pour définir l'amorce : il s'agit de la méthode définie par Pavlovic et Smith [132] pour construire l'objet initial de spécifications algébriques dont Specware fait la somme amalgamée par un *pushout* catégorique. Cette appréhension de la définition de l'amorce paraît beaucoup plus intéressante dans la mesure où elle se fonde sur les termes théoriques et pratiques utilisés dans notre étude.

Soient Y_1 et Y_2 respectivement les spécifications algébriques de DSL_1 et DSL_2. Définir formellement l'amorce d'une spécification algébrique X, de Y_1 et Y_2, consiste à décrire une spécification commune à ces DSL. Elle s'obtient, à partir d'une relation d'équivalence entre les éléments des spécifications, tout en préservant la propriété de prouvabilité d'un théorème de la source X, pour les théories cibles Y_1 et Y_2. Ainsi, considérant $r_1 : X \rightarrow Y_1$ et $r_2 : X \rightarrow Y_2$, le couple de symboles $(s, t) \in Y_1 \times Y_2$ est équivalent au symbole $u \in X$ si et seulement si les trois conditions suivantes sont validées :

1. $r_1(u) = s$

2. $r_2(u) = t$

3. Toute équation $(s/u)E$ où u est substitué à s dans E de X (respectivement $(t/u)E$) est équivalente à l'un des axiomes de Y_1 (respectivement Y_2).

Cette définition pourra servir de mécanisme pour élaborer la spécification de l'amorce en déterminant ses éléments en provenance de Y_1 et Y_2. Ces éléments reprennent ceux des deux DSL de base, en respectant les propriétés 1, 2 et 3, énoncées ci-dessus. Le mécanisme consiste à filtrer les éléments des spécifications initiales de façon consécutive afin de trouver ceux qui valident la définition. L'application de ce mécanisme sur les spécifications Y_1 et Y_2 nous permet de construire l'amorce de façon itérative.

Il s'agit d'identifier initialement les sortes et les opérations de l'amorce. Le choix de sélectionner ces éléments génère un résultat

fusionnant les notions communes. Cette heuristique donne un fil conducteur afin de définir l'amorce ; dans la pratique, l'identification des éléments est principalement intuitive.

Identification du vocabulaire commun : La première partie du travail consiste à trouver le vocabulaire métier commun entre les deux DSL, et aussi les opérations appliquées à ce vocabulaire. Les opérations représentent généralement des services connus dans le domaine.

Ensuite, les éléments du vocabulaire deviennent des candidats de l'amorce. Étant donné qu'un vocabulaire métier est représenté par des sortes dans une spécification algébrique d'un langage DSL, il s'agit de choisir une sorte *s* de DSL_1 et de reconnaître sa correspondance *t* dans DSL_2 afin de tester la possibilité de l'ajouter dans l'amorce avec éventuellement ses opérations.

Formellement, pour déterminer si *s* doit être rajouté à l'amorce, il faut trouver un élément *t* appartenant à DSL_2, tel que le couple *(s, t)* vérifie les deux premiers critères de la méthode. La recherche de *t* est guidée par sa signification métier, comme nous avons signalé ci-dessus. En effet, chacune des sortes, quelle soit écrite en DSL_1 ou DSL_2, représente une notion connue par les spécialistes du domaine. Par exemple, la notion d'un message existe nécessairement dans les deux langages DSLα et DSLβ puisqu'ils doivent supporter des échanges de signaux. Cette information pourra nous aider à identifier, dans DSLβ, une sorte associée. En utilisant cette technique à partir de la sorte *Condition*, on pourra trouver par exemple dans DSLβ une sorte équivalente, puisque cette notion existe dans les deux langages.

Une fois que nous avons identifié l'élément *t* correspondant à *s*, nous pourrons projeter le couple (*s*, *t*) dans l'amorce sous la forme d'un nouvel élément *u* et étudier la possibilité de projeter aussi les services qui lui sont rattachés. L'ajout du couple *(s ,t)* dans l'amorce induit un premier pas vers les morphismes r_1 et r_2 à établir entre DSL_1 et DSL_2.

Opérations et axiomes : une fois identifié le vocabulaire commun, il nous reste à projeter les opérations et les axiomes communs vers l'amorce. La condition nécessaire pour qu'une opération *p* du DSL_1 de profil *op* : *Dom* $\rightarrow$ *Cod* puisse être transférée vers l'amorce est de trouver une opération *p'* correspondant à *p* en DSL_2, telle que les sortes de *Dom' et Cod' de p' : Dom'* $\rightarrow$ *Cod'* correspondent respectivement à celles de *Dom* et *Cod* par les morphismes r_1 et r_2.

Supposons par exemple, que les opérations *p* et *p'* soient définies

par les profils respectifs $p : a \times b \rightarrow c$ et $p' : x \times y \rightarrow z$, nous avons à vérifier que $r_1(a) = r_2(x)$, $r_1(b) = r_2(y)$ et $r_1(c) = r_2(z)$.

Une fois ces conditions satisfaites, nous pourrons rajouter l'opération p (ou p') vers l'amorce. Cette transformation doit être respectée par les axiomes liés à l'opération p, équivalents au sens des morphismes de spécifications algébriques à ceux de p'.

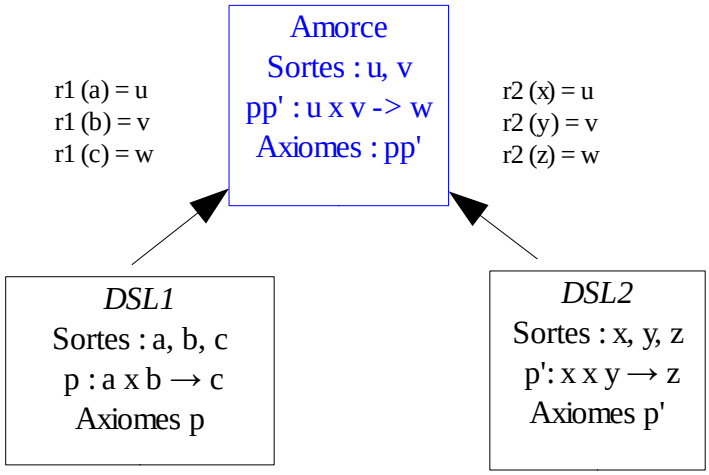

Figure 53: Les fonctions r_1 et r_2

La figure 53 schématise la construction de l'amorce de l'exemple. Les opérations p et p' donnent lieu à une seule opération pp'. Étant donné que l'égalité au sens des morphismes de spécifications algébriques existe entre p et p', leurs axiomes correspondants sont transférées vers l'amorce sans redondance. Notons que r_1 et r_2 sont bijectifs par construction sur leur domaine de définition ; il en résulte que les morphismes inverses existent : ce sont les morphismes initiaux du *pushout*.

Soulignons que le contexte de notre travail offre un avantage pour le processus de construction de l'amorce. En effet, le fait d'avoir un environnement métier unique pour les DSL à unifier nous donne des indications pour choisir les sortes à appareiller.

4.3.2-Obtention de l'amorce

Le contenu de l'amorce dans notre cas d'étude est bien établi via ce mécanisme. Dans le paragraphe qui suit, nous allons expliciter les étapes suivantes afin de construire l'amorce DSLγ à partir de DSLα et DSLβ.

Soit *Block* un élément du DSLα candidat à être intégré dans l'amorce. Il faut d'abord trouver un élément dans DSLβ ayant le même comportement au niveau métier. La notion *Statements* apparaît comme le meilleur candidat. Nous allons montrer que *Steps* est bien l'élément qui est associé au couple (*Block, Statements*).

```
Block ← Steps → Statements
```

Soit r_1 (*Block*) = *Steps* et r_2 (*Statements*) = *Steps*. Pour vérifier la condition (3) de la méthode proposée par Pavlovic et Smith, nous appliquons le procédé dans le sens inverse aux opérations et aux axiomes de DSLα et DSLβ, à condition qu'ils contiennent la sorte *Block* ou *Statements*.

D'autres sortes font aussi partie de l'amorce, et sont associées aux mêmes comportements fonctionnels dans les DSLα et DSLβ. Ces sortes ont les mêmes noms dans les deux langages, ce qui rend cette étape facile à réaliser. Dans les définitions suivantes, nous remplaçons dans (1) la sorte *Block* par *Step*. Les sortes *Environnement et Status* gardent leur nom et nous obtenons (2) qui est une autre opération, à savoir l'opération *evalMainBody* de l'amorce, en correspondance à que nous avons considéré.

```
Environnement ← Environnement → Environnement
Status ← Status → Status
```

```
op evalBlockBody : Block * Environnement -> Status *
                   Environnement (1)

op evalMainBody : Steps * Environnement -> Status *
                  Environnement (2)
```

Nous déduisons que l'opération (2), à la base de l'opération

evalMainBody, fait partie des opérations de l'amorce. Les axiomes reliés à l'opération *evalMainBody* sont eux aussi transférés vers l'amorce, après le changement de sortes indiqué ci-dessus.

Après la transformation des DSL α et β, nous obtenons deux candidats pour l'amorce : DSLγ-α et DSLγ-β. Ils contiennent un ensemble d'opérations et d'axiomes identiques portant les mêmes noms de paramètres et d'axiomes. L'intersection de ces deux versions donne l'amorce, suivie éventuellement d'une phase de renommage des opérations et des axiomes afin d'avoir une spécification plus homogène.

La mise en œuvre se fait avec le même outil Specware. Cette fois nous utilisons la fonctionnalité *translate*. Cette fonction de substitution modifie une spécification algébrique en remplaçant sa liste de sortes et d'opérations par une deuxième, tout en conservant les autres sortes inchangées.

Dans le cas de la construction de l'amorce, l'application de la fonction *translate* sur le langage DSLα, produit une nouvelle spécification DSLγ-α. Cette dernière est le langage DSLα en modifiant les sortes par les images définies par la fonction r_1. La fonction *translate* considère la liste de changements signalés par le mot clé *by* de la spécification donnée à la figure 54. *L*e code complet de la translation du DSLα figure en annexe.

Nous appliquons la même fonction que *translate* sur DSLβ, et nous

```
translate spec ...
--Spécification de DSLα-----
endspec by {
      Checks +-> Conditions ,
      Block +-> Steps,
      evalBlock +-> evalProcedure,
      checkVariables +-> evalPrecondition,
      evalBlockBody +-> evalMainBody,
      checkExpressions +-> evalExpressions.
            }
```

Figure 54: Fragment du code de la translation du DSLα

obtenons une nouvelle spécification DSLγ-β, avec la liste des modifications données à la figure 55.

La comparaison du code de DSLγ-α et DSLγ-β prouve l'existence d'une partie commune de code. Cette partie constitue l'amorce des

```
translate spec ...
-- Spécification  du DSLβ----
endspec by  {
      Statements +-> Steps,
      evalModule +-> evalProcedure,
      evalPreconditionBody +-> evalPrecondition,
      evalStatements +-> evalMainBody,
      evalBooleanExpressions +-> evalExpressions.
            }
```

Figure 55: Fragment du code de la translation du DSLβ

deux DSL, telle que nous l'avons utilisée dans l'exemple du chapitre antérieur.

4.3.3-Démarche d'élaboration du DSL unificateur

Après l'introduction de l'heuristique pour construire l'amorce, nous présentons les étapes nécessaires afin de construire le langage unificateur d'une famille de DSL. Soient DSL_1 et DSL_2 deux DSL appartenant à la même famille, qu'on désire unifier.

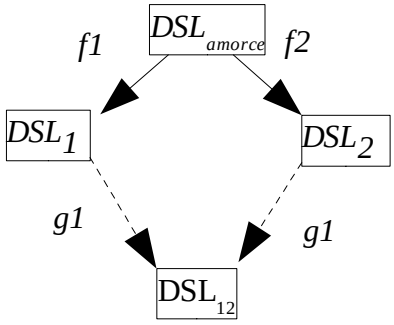

Figure 56: Pushout

Établir l'objet DSL_{12}, co-limite de $DSL_1 \leftarrow DSL_{amorce} \rightarrow DSL_2$, nécessite de spécifier algébriquement l'amorce et les deux morphismes de spécifications f_1 et f_2.

• DSL_1 et DSL_2 correspondent chacun à la spécification d'un DSL de la famille,

• DSL_{amorce} est la spécification commune des DSL_1 et DSL_2,

• f_1 et f_2 matérialisent formellement les points communs à DSL_{amorce}, DSL_1 et DSL_2,

• DSL_{12} est un DSL calculé qui unifie DSL_1 et DSL_2 conformément au points de recollement du DSL_{amorce},

• g_1 et g_2 plongent par morphismes les spécifications DSL_1 et DSL_2 dans DSL_{12}.

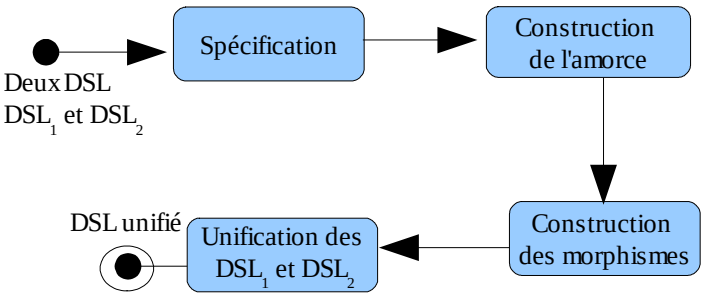

Figure 57: Étapes du processus de développement du langage unificateur

La description des ces quatre étapes de la démarche est la suivante:

Etape 1 - Spécification des DSL : Le point de départ de notre approche est donné par les langages DSL de la famille ; dans notre cas deux langages. Le processus doit initialement établir la sémantique algébrique des langages DSL_1 et DSL_2.

Cette étape n'est pas automatisable.

Etape 2 - Définition de l'amorce : Une fois écrites les spécifications algébriques des deux DSL, la deuxième étape consiste à définir formellement une spécification commune DSL_{amorce} aux deux DSL DSL_1 et DSL_2. Celle-ci, spécifiée algébriquement, est obtenue à partir d'une relation d'équivalence entre les symboles de DSL_1 et de DSL_2. Ces relations doivent préserver la propriété de prouvabilité d'un axiome de la source DSL_{amorce} transformé pour les théories cibles DSL_1 et DSL_2.

Cette étape a été largement détaillée dans le paragraphe précédent ; elle est basée sur des travaux de Pavlovic et Smith [132].

Etape3 - Définitions de morphismes initiaux : L'heuristique proposée pour la définition de l'amorce rend la construction des morphismes initiaux automatisable. En effet, ces morphismes mettent en correspondance les sortes, opérations et axiomes de l'amorce d'une part, et ceux des deux langages DSL_1 et DSL_2 d'autre part. Ces morphismes sont en fait les relations r_1 et r_2 définies à l'étape précédente ; par construction, ce sont des morphismes de spécifications.

Cette étape est automatique.

Etape 4 - Création du langage unificateur : Dans cette étape, nous avons tous les éléments nécessaires pour générer, par *pushout*, l'unificateur DSL_{12}. Ainsi, si $f_1 : DSL_{amorce} \rightarrow DSL_1$ et $f_2 : DSL_{amorce} \rightarrow DSL_2$ sont les morphismes de spécifications initiales, alors la spécification algébrique de DSL_{12}, résultat de la somme amalgamée de f_1 et f_2 a pour sortes et opérations celles qui sont communes à f_1 et à f_2 dans DSL_{amorce}, complétées par celles qui sont disjointes. Ses équations sont celles de DSL_1 et DSL_2 définies sur les sortes de DSL_{amorce} après traduction par les morphismes g_1 et g_2. Notons que les équations de l'amorce DSL_{amorce} sont déjà présentes dans DSL_1 et DSL_2 par les morphismes de spécifications f_1 et f_2.

Le calcul du DSL DSL_{12} est effectué par le logiciel Specware de Kestrel qui contient tous les mécanismes nécessaires à la définition et au traitement des entités catégoriques qui interviennent dans la mise en oeuvre de notre approche formelle. Le calcul est constructif et ne dépend que des liens établis initialement entre les spécifications d'origines. Cette co-limite existe pour tout diagramme fini, car la catégorie des spécifications algébriques équationnelle est co-complète [109].

Considérant la construction $DSL_1 \leftarrow DSL_{amorce} \rightarrow DSL_2$, Specware détermine la spécification du DSL_{12}, somme amalgamée de DSL_1 et DSL_2, et construit les morphismes *g1* et *g2*, qui, dans notre problématique expriment des traductions de code.

Étape	Spécification algébrique des DSL	Définition de l'amorce	Morphismes des spécifications initiales	Langage unificateur
Niveau d'assistance	Manuel	Assisté	Automatique	Automatique
Outillage	Assisté	Notre démarche/ Specware	Specware	Specware
Pérennité	Unique par langage	Unique pour un ensemble de langages	Unique pour un ensemble de langages	Unique

Tableau 1: Caractéristiques des étapes de la démarche

Le tableau 1 ci-dessus résume les propriétés de notre démarche de construction de l'unificateur. Il reprend les quatre étapes proposées et décrit pour chacune son niveau d'assistance, l'outillage préconisé et l'effort à déployer vis-à-vis de la famille.

Les étapes d'écriture des spécifications algébriques et de définition de l'amorce restent essentiellement manuelles par manque d'outils. Elles peuvent influencer les autres parties, éventuellement, le résultat lui-même.

L'étape de construction de l'amorce est assistée. Elle nécessite cependant pragmatisme et intuition. Elle doit prendre appui sur les concepts métiers. Notons que la construction de l'amorce et son contenu influencent directement le résultat de l'unification.

Une fois l'amorce définie, les morphismes initiaux peuvent être obtenus de façon automatique à l'aide de Specware.

La quatrième étape se fait de façon automatique par Specware, dès lors que les trois autres étapes ont été effectuées correctement.

Les deux dernières étapes sont totalement dépendantes de la construction de l'amorce et des langages initiaux. Pour un ensemble de langages donné et une amorce fixe, la troisième et la quatrième étape sont pérennes.

En résumé, notre démarche propose un calcul constructif du langage unificateur d'une famille, qui ne dépend que des liens établis initialement entre les spécifications d'origine. Une fois les

spécifications initiales construites, nous pouvons identifier les éléments composant l'amorce et définir les morphismes initiaux. De plus, cette co-limite existe pour tout diagramme fini.

Considérant à la fois la faisabilité théorique et pratique, nous avons choisi dans cette étude le cadre des spécifications algébriques pour construire une sémantique formelle d'une famille de DSL. L'utilisation du concept d'institution [109] pour décrire la sémantique et les propriétés a été envisagée. Rappelons qu'une institution fait intervenir une syntaxe (collection de signatures), des formules définies sur la signature, des modèles (ici des DSL) et une relation de satisfaction entre des formules et des modèles. En cas de changement de signatures, l'exigence sémantique impose la préservation de la relation de satisfaction pour les formules et modèles traduits.

L'intérêt de ce formalisme réside dans la possibilité de développer des concepts sur des spécifications indépendamment des systèmes logiques sous-jacents, la liaison se faisant par des morphismes catégoriques. Bien que fort riches et ayant fait l'objet de nombreux travaux jusqu'à ce jour [110] [133], nous n'avons pas retenu les institutions pour cette étude. D'une part, une telle stratégie aurait induit un fort déséquilibre entre la complexité des objets traités (les DSL) et le formalisme pour les traiter (les institutions) au sein d'une communauté, en général, peu encline aux méthodes formelles. D'autre part, la propriété fondamentale des institutions liée à la possibilité de s'appuyer sur plusieurs systèmes logiques n'aurait guère été utilisée : l'exploitation d'une seule logique est suffisante pour traiter des langages simples tels que les DSL, en l'occurrence la logique équationnelle.

4.4-Évaluation de notre approche

4.4.1-Discussion

Pourquoi une famille et pourquoi de DSL ?

Notre approche, telle que définie au chapitre 3, peut s'appliquer a priori à un ensemble arbitraire de langages. Cependant l'utilisation d'une famille de DSL est un des avantages de notre approche pour les raisons suivantes qui montrent l'intérêt de se limiter à un tel contexte.

Dans le cas de langages n'appartenant pas à la même famille, c'est-à-

dire dont les ensembles de primitives sont très éloignés, on ne peut identifier qu'une faible partie commune. Par conséquent, la partie à factoriser devient moins importante et peut être vide. Dans ce cas, le langage unificateur sera simplement une juxtaposition des concepts des différents langages. C'est pourquoi, nous préconisons de se restreindre à un ensemble de langages proches syntaxiquement et sémantiquement.

Dans le cas de langages dont les définitions sont plus sophistiquées et plus complexes que celles d'un DSL, la définition de la sémantique formelle devient une tâche ardue. D'un autre côté, la définition de l'amorce, ainsi que celle des morphismes initiaux s'alourdissent en complexité. Ainsi, nous considérons plus raisonnable de se limiter à des langages dont les spécifications algébriques sont réduites en taille, comme c'est le cas des langages dédiés.

Une direction de travail consisterait à étudier plus précisément dans quelle mesure l'éloignement des langages qu'on désire unifier ainsi que la taille de leur spécification algébrique affectent l'applicabilité de notre approche.

Taille et « qualité » de l'amorce

Dans cette section, nous nous posons la question de la taille et de la qualité de l'amorce.

Dans le cas extrême, on peut imaginer une amorce vide. Ce résultat correspond à une intersection vide des langages à unifier. Le raisonnement nous amène à déduire que les deux langages n'ont aucun concept en commun. Le langage unificateur est alors simplement l'union ensembliste des deux langages. Si des points communs existent mais n'ont pas été identifiés, le langage unificateur contiendra des concepts en doublons, et on ne pourra profiter des factorisations réalisées par le *pushout*.

La solution idéale consiste à avoir une amorce maximale. Cela correspond à la situation où on arrive à identifier toutes les intersections des concepts mis en jeu séparément. L'heuristique pour la définition de l'amorce propose une construction itérative, à partir d'une amorce vide. De façon itérative, nous ajoutons un élément après l'autre, selon notre méthodologie, jusqu'à obtenir une amorce maximale.

A noter qu'une amorce incluant tous les éléments d'un des deux langages signifie que le langage spécifié par l'amorce est inclus dans l'autre, à un morphisme près. Si les deux langages ont un contenu strictement identique à l'amorce, ils sont identiques.

4.4.2-Points forts de l'approche

Le problème de l'interopérabilité a fait l'objet de nombreux travaux depuis l'apparition des langages de programmation de haut niveau. En particulier, [134] [135], proposent une approche qui vise l'interopérabilité au sein des langages de procédures opérationnelles, à savoir le même support métier que celui utilisé dans notre travail. Dans leurs travaux, les auteurs exploitent des passerelles semi-automatiques entre les langages. Notre méthode se fonde sur des constructions théoriques, telles que la sémantique algébrique [118] et la théorie des catégories [104], ce qui assure un socle formel solide à la démarche. De plus, le calcul du langage unificateur se fait d'une façon automatique à partir de la spécification de l'amorce et des morphismes dits initiaux grâce à la fonctionnalité du *pushout* [103].

Notre approche présente l'avantage d'être plus flexible par rapport à l'introduction d'un nouveau langage dans la famille. En effet, lors d'un ajout de langage, nous avons besoin d'adapter l'amorce, et par la suite les morphismes en fonction des propriétés syntaxiques et sémantiques du nouveau langage. Dans notre contexte, un développeur travaille toujours dans son langage. L'apprentissage d'un nouveau langage ou la migration vers un nouvel environnement de développement ne sont pas nécessaires.

Notre proposition se base sur l'utilisation d'outils existants. L'approche a pu être validée sans développement logiciel spécifique. Des outils comme Specware [111] [96] [113], Isabelle [128] et ASF+SDF existent depuis longtemps et sont utilisés dans d'autres contextes. Nous avons pu les utiliser sans modification.

Les morphismes de traduction générés automatiquement peuvent donner lieu à des traducteurs de la famille vers l'unificateur. De par l'existence d'un langage unificateur, le nombre minimal de traducteurs est d'ordre $O(n)$, et non d'ordre $O(n^2)$, comme c'est le cas si on utilise des traducteurs entre chaque paire de langages [136] et [135].

Un des points forts de notre approche concerne la préservation des propriétés établies dans le contexte des DSL. La mise en place de vérifications formelles pour des DSL évolutifs est une difficulté soulevée dans [137], où les auteurs mettent en avant l'intérêt d'une vérification incrémentale.

L'intérêt des preuves de propriétés de DSL est important. Dans cette direction, notre souci a été de dépasser le cadre strictement formel pour donner l'outillage informatique assurant la continuité entre la théorie, l'implantation au moyen de Specware, et le lien vers l'assistant

de preuve Isabelle. La démarche présentée ici, offre donc toutes les garanties issues des méthodes formelles ainsi qu'un environnement logiciel certes relativement lourd mais qui permet d'en implanter toutes les étapes ; cette dernière considération, constitue sans doute un point fort de notre étude.

4.4.3-Limites de l'approche

La définition des DSL se fait le plus souvent à travers des grammaires BNF. La première étape du processus d'élaboration du langage unificateur consiste à décrire ces langage en termes de spécifications algébriques afin d'être capable de calculer le langage unificateur. Cette définition vient en plus du développement des compilateurs pour les langages et constitue le prix à payer pour mettre en œuvre notre approche. En effet, malgré l'existence d'outils qui peuvent faciliter une partie de cette définition formelle, cette étape reste indispensable.

La construction de l'amorce joue un rôle essentiel pour calculer le langage unificateur. Sa définition est indispensable et incontournable. Malgré l'heuristique que nous proposons, la définition de l'amorce peut s'avérer délicate et laborieuse.

Un grand effort a été dépensé afin d'expérimenter l'unification des langages DSLα et DSLβ, la difficulté principale provenant de la prise en main d'outils existants, principalement Specware pour modéliser la spécification et calculer le *pushout*, et Isabelle afin de prouver des propriétés et démontrer la validité de l'approche.

L'effort nécessaire est bien sûr capitalisable. Cependant les outils requis restent disparates. Pour une utilisation plus large de notre approche, il convient d'envisager des développements autour des interfaces de ces outils.

4.5-Travaux futurs

4.5.1-Étendre l'approche vers l'unification des spécifications

Notre approche permet d'unifier des définitions de langages. Une direction intéressante consisterait à unifier des spécifications

directement utilisées par les ingénieurs. Cela passerait fort probablement par l'utilisation d'un autre formalisme de spécification.

Pour chaque famille des DSL traitée, il conviendrait de s'interroger sur la logique la plus pertinente à employer compte tenu du domaine spécifique de ces DSL. Combinant systèmes logiques et spécifications algébriques, le concept d'institution de Goguen et Burstall [109] apparaît très pertinent ; la poursuite de cette étude en utilisant ce formalisme semble une voie très prometteuse à explorer.

En ce qui concerne nos travaux, les spécifications algébriques présentent l'avantage qu'il existe des résultats fondés montrant qu'elles forment une catégorie. Choisir un autre formalisme impose de conserver cette propriété.

Certains résultats [132] nous laissent confiants quant à l'existence d'un tel formalisme. Cependant un travail mathématique est nécessaire afin de trouver ce formalisme et prouver qu'il satisfait les bonnes propriétés. Dans notre papier [138], nous avons montré notre confiance dans le fait que les ASM (*Abstract State Machines*) [139] pourraient être utilisés dans ce contexte pour exécuter symboliquement un système à base de composants du domaine. Cependant, depuis, nous n'avons pas pu explorer plus en détail cette piste.

4.5.2-Réutilisation des propriétés établies au niveau M0

Dans le paragraphe 4.2, nous avons expliqué comment exploiter notre approche et notamment comment transférer une propriété établie dans le contexte d'un élément de la famille vers le langage unifié. Pour l'instant, cette réutilisation des propriétés concerne seulement les propriétés qui portent sur la définition du langage, comme c'est le cas des exemples de la section 4.1, qui représentent des propriétés qui doivent être satisfaites indépendamment d'une spécification concrète.

Une piste de travail consiste à se concentrer sur des propriétés établies au niveau applicatif, ou M0, donc qui dépendent des spécifications concrètes et non pas seulement de la définition du langage.

La figure 58 illustre les différences pouvant exister entre des propriétés de niveau M0 et de niveau M1. Par rapport à cette figure, nous préconisons comme travail futur d'étudier comment garantir la transition des propriétés d'un niveau applicatif d'une spécification (*p*

dans la figure) vers des propriétés dans le contexte unifié (*p'* dans cette figure).

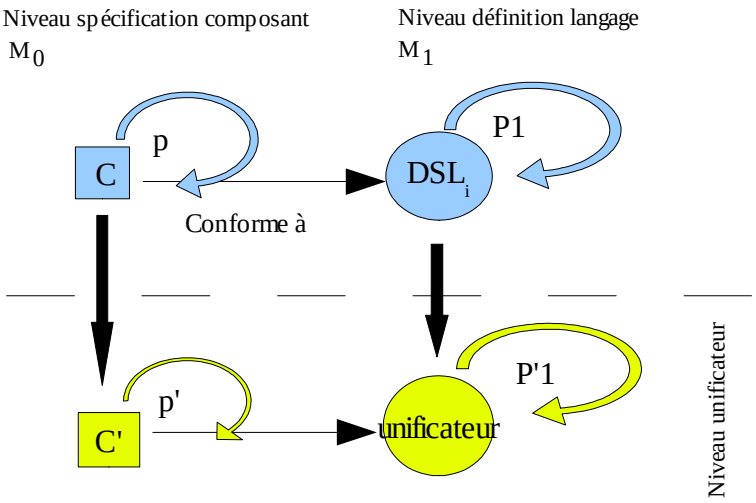

Figure 58: Propriétés au niveau des composants

Dans le papier ABZ [138], nous avons présenté plus en détail cette direction de travail. Ici le point clé consiste à trouver la meilleure façon de représenter les spécifications. Ceci pourrait nécessiter le passage des spécifications algébriques vers un autre formalisme, comme les ASM [139] par exemple, plus adéquat pour la partie exécution.

4.5.3-Famille de DSL et domaine métier

Dans notre approche, nous avons décrit une approche qui permet de résoudre le problème de l'interopérabilité au sein d'une famille de DSL. Afin de valider cette approche, nous avons considéré deux langages DSLα et DSLβ du domaine spatial. Dans l'environnement de l'outil Specware, nous avons appliqué notre approche, en définissant l'amorce et les morphismes initiaux, afin de générer automatiquement son langage unificateur.

Il serait pertinent de tester cette approche sur des familles diverses de DSL, dans lesquelles les paramètres de la famille jouent un rôle sur le résultat obtenu. Par exemple, la taille de la famille ou l'écart entre les spécifications des langages de la famille peuvent influencer la mise en

œuvre de notre approche par la complexité de la définition des sémantiques, du langage amorce et des morphismes initiaux.

4.5.4-Passage à l'échelle

Une autre direction intéressante consisterait à étudier comment notre approche réagit à l'utilisation d'ensembles de taille importante, ou de langages de taille importante.

Il s'agit en grande partie d'un travail expérimental. Afin de rendre les résultats plus significatifs, on peut imaginer de les combiner avec l'utilisation de métriques sur la complexité ou la taille d'un langage.

4.5.5-Utiliser le standard OMG sur la variabilité

Comme nous l'avons vu à la section 1.3.6, une initiative OMG récente vise à définir un langage de spécification de la variabilité au sein d'une ligne de produits ou d'une famille de langages. Des propositions sont attendues pour l'été 2010.

Une fois le langage de variabilité défini, il faudrait analyser l'impact de son existence sur notre approche. Il est probable qu'une telle existence faciliterait la définition du langage amorce, dont la définition pourrait se faire de façon quasi-automatique.

4.6-Travaux similaires

Le résultat majeur de notre travail concerne l’interopérabilité des langages de programmation. Cette dernière a fait l’objet de nombreux travaux.

Pour assurer l'interopérabilité de *n* langages on peut être amené à construire *n(n-1)* traducteurs. Cette complexité peut être réduite à *O(n)*, en utilisant un langage pivot.

Une autre réponse à l’interopérabilité est donnée par la dérivation automatique de traducteurs à partir d’une description syntaxo-sémantique appropriée. Dans cette catégorie, on peut trouver des environnements tels que ASF+SDF [98] et des réalisations plus récentes comme [135] avec qui nous partageons la problématique de l’interopérabilité des procédures opérationnelles [134]. Dans [135], les auteurs s’appuient sur la similarité entre les membres d’une famille de

langages pour construire semi-automatiquement un schéma de traduction entre eux. Ici aussi, il en résulte une famille de n^2 traducteurs centrés sur les aspects syntaxiques alors que notre approche traite au premier chef de la sémantique.

L'approche utilisée par [101] est très colorée IDM puisque elle repose sur le système AMMA [140] explicitement destiné à la coopération inter-DSL dans un contexte de (méta-) modélisation. Il est à noter toutefois que le système AMMA est plus large et a pour ambition de faire interopérer des DSL quelconques et non des DSL voisins comme nous le faisons. Il est à remarquer aussi que cette approche privilégie également le point de vue syntaxique par rapport à la sémantique.

Notre travail propose une solution pour l'unification d'une famille de langages dédiés. Il convient d'insister sur l'importance de ces deux aspects : en nous concentrant sur des langages dédiés, nous restreignons la complexité des langages à prendre en compte ; en nous focalisant sur les langages d'une même famille, nous visons à prendre en compte l'ensemble des notions communes aux langages à unifier. Nous sommes loin des approches d'unification généralistes qui essaient de marier des langages éloignés ou à grande portée, comme des approches de construction simultanée de modèles en UML et B [141], ou qui visent à intégrer dans un même contexte des modules de nature différente comme par exemple données et protocoles en [142].

Les DSL sont par essence des langages simples, permettant à un non-spécialiste de spécifier et de développer un logiciel. Cette caractéristique induit un nombre de constructeurs limités dans le langage qu'une sémantique algébrique est à même de décrire raisonnablement. La théorie des catégories est un outil formel relativement courant en génie logiciel depuis longtemps. Des travaux récents en ingénierie des modèles utilisent ce formalisme pour lier l'IDM et la synthèse de programmes par *Software Production Lines* [143].

Dans notre proposition, l'utilisation des catégories offre une certaine proximité avec [118]. Fondée sur des diagrammes catégoriques tels que *pushouts* ou co-limites, notre méthode nécessite l'existence d'un langage amorce. La définition de ce dernier conduit à un problème méthodologique qui peut recevoir plusieurs réponses, notamment celle définie par Pavlovic et Smith [132] pour construire l'objet initial de spécifications algébriques dont Specware fait la somme amalgamée par un *pushout* catégorique. Cette appréhension de la définition de l'amorce paraît beaucoup plus intéressante dans la mesure où elle se fonde sur les termes théoriques et pratiques utilisés dans notre étude.

Un des intérêts majeurs de notre approche concerne la preuve de propriétés. La mise en place de vérifications formelles pour des DSL évolutifs est une difficulté soulevée dans [137] où les auteurs mettent en avant l'intérêt d'une vérification incrémentale. Dans notre proposition, la construction de preuves et l'héritage de propriétés provenant des membres de la famille dans le DSL obtenu par co-limite est l'un des bénéfices du caractère formel de la construction. L'intérêt des preuves de propriétés de DSL est important surtout dans le cas où le monde du problème est celui des applications critiques ou embarquées comme celles qui illustrent cette contribution.

Conclusion et perspectives

Le résultat majeur de cette thèse concerne l'interopérabilité de langages dédiés issus d'un même domaine métier. Par rapport à une approche IDM conventionnelle qui consiste à méta-modéliser les différents langages de la famille afin de faciliter la définition d'un méta-modèle du domaine, notre objectif principal est de permettre la factorisation du savoir-faire métier sans introduire de nouveaux artéfacts de modélisation et d'y intégrer de manière semi-automatique la sémantique. Contrairement à une approche par méta-modélisation souvent dépourvue de sémantique explicite, le DSL unificateur que nous obtenons fédère les concepts sémantiquement communs.

Synthèse des travaux

Nous avons ainsi décrit une approche pour construire ce DSL minimal offrant toutes les fonctionnalités sémantiques des membres d'une même famille et par là même mis en place une interopérabilité au sein de la famille. La méthode choisie ici se fonde sur des constructions théoriques telles que la sémantique algébrique et la théorie des catégories qui assurent un socle formel solide à la démarche. Ce DSL se définit et s'obtient par un calcul de co-limite qui garantit, par construction, que ses sortes et opérations proviennent uniquement de celles des DSL mis en jeu par le diagramme catégorique et de celles qui coïncident à son point de recollement. Les constructions catégoriques par co-limites nous apportent également la possibilité de transmettre des propriétés d'un DSL vers le DSL unificateur.

Grâce aux propriétés de la co-limite, nous garantissons le même environnement de développement à un expert du domaine. En effet, la co-limite détermine aussi les morphismes entre les DSL mis en jeu et le DSL unificateur, ce qui nous permet de traduire automatiquement les codes sources d'un expert vers ce DSL commun. Considérant *n* langages sources, nous évitons ainsi la construction de $n(n-1)$ traducteurs croisés et ramenons la complexité du système à $O(n)$. L'intégration d'un nouveau langage de la famille est incrémentale, en définissant une nouvelle amorce et les morphismes vers le nouveau langage. D'autre part, grâce au langage unificateur incluant à la fois les notions communes aux DSL et leur spécificité, les composants peuvent interopérer.

Nous avons également défini les étapes de construction du DSL unificateur en indiquant pour chacune son niveau d'assistance ainsi

que l'outillage approprié. En nous concentrant sur des langages dédiés, nous restreignons la complexité des langages à prendre en compte ; en nous focalisant sur les langages d'une même famille, nous visons à prendre en compte l'ensemble des notions communes aux langages à unifier. Ces caractéristiques induisent un nombre de constructeurs limités pour un langage donné qu'une sémantique algébrique est à même de décrire raisonnablement, première étape de notre démarche. Les autres étapes résultent de la mise en place de morphismes nécessaires au *pushout* : la définition du point de recollement, appelé langage amorce, accompagné de morphismes vers les spécifications algébriques définies précédemment.

Nous suggérons une méthode constructive pour définir cette amorce : il s'agit de filtrer les éléments communs à deux spécifications par le biais d'une relation d'équivalence entre les sortes et les opérateurs préservant la propriété de prouvabilité d'un théorème de la source pour la théorie cible. Il n'est pas nécessaire pour cela de comparer exhaustivement deux à deux les symboles des deux spécifications, mais plutôt de superposer les concepts similaires véhiculés par les deux DSL, puis de s'assurer que les noms des sortes, les profils des opérateurs et les axiomes ainsi identifiés sont transposables d'une théorie à l'autre. La synonymie à mettre en évidence est facilitée par le fait que les profils des opérations de la spécification calquent, pour un même concept langagier, les règles de production de la grammaire.

Notre souci a été de dépasser le cadre strictement formel pour donner l'outillage informatique assurant la continuité entre la théorie, l'implantation au moyen de Specware pour les opérations catégoriques et le lien vers Isabelle pour la preuve de propriétés. La démarche présentée s'appuie à la fois sur une méthode formelle et sur un environnement logiciel certes relativement lourd, mais qui permet d'en implanter toutes les étapes ; cette dernière considération constitue sans doute le point fort de l'étude. C'est ainsi que nous avons validé notre approche pour deux fragments de DSL relevant des procédures opérationnelles du domaine spatial.

Perspectives

Certains points de l'étude restent cependant à améliorer. Les deux premières étapes du processus de développement consistent respectivement à écrire la spécification algébrique des langages et la définition d'une amorce commune à ces spécifications. L'écriture de spécifications reste un exercice conceptuel demandé aux experts du domaine ; ces derniers doivent être accompagnés pour établir l'amorce et les morphismes vers les spécifications sources. Cette tâche reste indispensable à notre approche. En complément de l'algorithme

d’appariement de deux spécifications que nous avons élaboré, il nous semble important d’outiller l’étape de construction de l’amorce par diverses heuristiques afin de la rendre plus automatique. De manière générale, si nous souhaitons dépasser le cadre expérimental de l’étude, il convient d’assister le plus possible les experts dans les différentes étapes d'obtention du langage unificateur à plusieurs DSL.

Il serait intéressant de poursuivre ce travail dans plusieurs directions. Pour chaque famille de DSL, il conviendrait de s’interroger sur la logique la plus pertinente à employer compte tenu de la spécificité du domaine. Combinant les systèmes logiques et les spécifications algébriques, le concept d’institution apparaît séduisant ; la poursuite de cette étude en utilisant ce formalisme semble une voie prometteuse à explorer. C'est ainsi que nous avons expérimenté d'autres formalismes que les spécifications algébriques, par exemple les ASM afin d'exécuter symboliquement un système à base de composants du domaine.

Une autre piste de travail consisterait à se concentrer sur des propriétés établies au niveau applicatif dépendant de spécifications concrètes et non uniquement de la définition d’un langage. Ce travail consiste à étudier comment faire interopérer les composants et transférer leurs propriétés dans le contexte unifié. Le point clé consiste à trouver la meilleure façon de faire coopérer les spécifications algébriques et les spécifications comportementales des composants ou bien d’identifier une catégorie dédiée aux composants.

Notons que certains travaux visent à spécifier un composant ou un programme par des propriétés dans une logique équationnelle, ce qui permet d'en extraire automatiquement sa sémantique algébrique. On retrouve ainsi une préoccupation similaire à la nôtre car elle tend à démontrer que les propriétés de la spécification sont des théorèmes dans la théorie du programme.

Enfin, une autre question relative à notre approche concerne son passage à l’échelle. Il serait utile de tester cette approche sur diverses familles de DSL, dans lesquelles les paramètres d’une famille peuvent avoir une incidence sur le résultat obtenu. Par exemple, la taille de la famille ou la distance entre les spécifications des langages peuvent influencer la mise en œuvre de notre approche. Dans le même ordre d’idée, se pose la question de l’ajout d’un nouveau membre à la famille. Lors d’un tel ajout, le langage unificateur doit naturellement être re-calculé, ce qui remet en cause la définition du langage amorce. Le cas particulier d'un *pushout* de deux morphismes devrait, dans notre contexte, pouvoir se généraliser aisément à un calcul de co-limite avec plusieurs morphismes.

La préoccupation récente de l'OMG concernant la variabilité de lignes de produits peut s'apparenter à la prise en compte des variabilités syntaxiques et sémantiques au sein d'une même famille. Il conviendrait dans ce cas de comparer finement les objectifs et les techniques mises en œuvre dans les deux situations, afin d'envisager un éventuel rapprochement.

Annexe

Langage α

```
alpha = spec

type Status= | aborted |completed
type Event
type Checks = List (Expression) * List (Event)
type Block
type Environment = Context * Queue
type Expression
type Context = List (Expression)
type Queue = List (Event)


op evalBlock :
      Checks * Block * Environment -> Status *
      Environment
op evalBlockBody :
      Block * Environment -> Status * Environment
op checkVariables : Checks * Environment -> Boolean
op checkExpressions :
      List (Expression) * Context -> Boolean
op  waitStatements  :  List  (Event)  *  Queue  ->  Boolean

axiom a1 is fa (c : Checks, b : Block, e : Environment)
      ~(checkVariables (c, e)) => evalBlock (c, b, e) =
      (aborted, e)

axiom a2 is fa (c : Checks, b : Block, e : Environment)
      checkVariables (c, e) => evalBlock (c, b, e) =
      evalBlockBody (b, e)

axiom a3 is fa (c : Checks, e : Environment)
      checkVariables (c, e) = checkExpressions (c.1,
e.1)  && waitStatements (c.2, e.2)

axiom a4 is fa (le : List (Expression), c : Context)
      checkExpressions (le, c) =
      (case le of
      | [] -> true
      | hd::tl -> member (hd, c) &&
                  checkExpressions (tl,   c))

axiom a5 is fa (le : List (Event), q : Queue)
      waitStatements (le, q) =
```

```
      (case le of
      | [] -> true
      | hd::tl -> (case q of
            | [] -> false
            | q1::others -> if q1 = hd then
waitStatements          (tl, others) else false))
endspec
```

Langage β

```
beta = spec

type Status= | aborted |completed
type Conditions = List (Expression) * List (Event)
type Statements
type Environment = Context * Queue
type Expression
type Event
type Context = List (Expression)
type Queue = List (Event)
type ContingencyStatus = |nominal |warning | error

op evalModule :
      Conditions * Statements * Environment -> Status *
      Environment
op evalStatements :
      Statements * Environment -> Status * Environment
op evalPreconditionBody :
      Conditions * Environment -> Boolean
op evalBooleanExpressions :
      List (Expression) * Context -> Boolean
op evalContingency : Queue -> Boolean
op contingencyStatus : Event -> ContingencyStatus


axiom b1 is fa (lc : Conditions, s : Statements, e :
Environment)
      ~( evalPreconditionBody (lc, e)) =>
      evalModule (lc,   s, e) = (aborted, e)

axiom b2 is fa (lc : Conditions, s : Statements, e :
Environment)
      evalPreconditionBody (lc, e) =>
      evalModule (lc, s,e) = evalStatements (s, e)

axiom b3 is fa (lc : Conditions, e : Environment)
      evalPreconditionBody (lc, e) =
      evalBooleanExpressions (lc.1, e.1) &&
      evalContingency (e.2)

axiom b4 is fa (le : List (Expression), c : Context)
```

```
      evalBooleanExpressions (le, c) =
      (case le of
      | [] -> true
      | hd::tl -> member (hd, c) &&
      evalBooleanExpressions  (tl, c))

axiom b5 is fa (q : Queue)
      evalContingency (q) =
      (case q of
      | [] -> false
      | hd::tl -> contingencyStatus (hd) = nominal)

endspec
```

Langage γ

```
gamma = spec

type Status= | aborted |completed
type Event
type Conditions
type Steps
type Environment
type Expression
type Context = List (Expression)
type Queue

op evalProcedure :
      Conditions * Steps * Environment -> Status *
      Environment
op evalMainBody :
      Steps * Environment -> Status * Environment
op evalPrecondition :
      Conditions * Environment -> Boolean
op evalExpressions :
      List (Expression) * Context -> Boolean

axiom g1 is fa (lc : Conditions, ls : Steps, e :
Environment)
      ~ (evalPrecondition (lc, e)) =>
      evalProcedure (lc, ls, e) = (aborted, e)

axiom g2 is fa (lc : Conditions, ls : Steps, e :
Environment)
      evalPrecondition (lc, e) =>
      evalProcedure (lc, ls, e) = evalMainBody (ls, e)

axiom g3 is fa (le : List (Expression), c : Context)
      evalExpressions (le, c) =
      (case le of
```

```
      | [] -> true
      | hd::tl -> member (hd, c) &&
      evalExpressions (tl, c))

endspec
```

Morphisme Gamma-Alpha

```
gamma_to_alpha = morphism gamma -> alpha
      {
      Conditions +> Checks,
      Steps +> Block,
      evalProcedure +> evalBlock,
      evalPrecondition +> checkVariables,
      evalMainBody +> evalBlockBody,
      evalExpressions +> checkExpressions
      }
```

Morphisme Gamma-Beta

```
gamma_to_beta = morphism gamma -> beta
      {
      Steps +> Statements,
      evalProcedure +> evalModule,
      evalPrecondition +> evalPreconditionBody,
      evalMainBody +> evalStatements,
      evalExpressions +> evalBooleanExpressions
      }
```

Langage unificateur αβ

```
spec

type Status = | aborted | completed
type Event
type {Statements, Block, Steps}
type Expression
type Context = List(Expression)

op {evalProcedure, evalBlock, evalModule} :
      Conditions * Statements * Environment -> Status *
Environment
op {evalMainBody, evalBlockBody, evalStatements} :
      Statements * Environment -> Status * Environment
```

```
op {evalPreconditionBody, checkVariables,
      evalPrecondition} : Conditions * Environment ->
      Boolean
op {evalBooleanExpressions, checkExpressions,
      evalExpressions} : List(Expression) * Context ->
      Boolean

axiom g1 is fa(lc : Conditions, ls : Steps, e :
Environment)
      ~(evalPrecondition(lc, e)) =>
      evalProcedure(lc, ls,   e) = (aborted, e)

axiom g2 is fa(lc : Conditions, ls : Steps, e :
Environment)
      evalPrecondition(lc, e) =>
      evalProcedure(lc, ls, e) = evalMainBody(ls, e)

axiom g3 is  fa(le : List(Expression), c : Context)
      evalExpressions(le, c)
      = (case le of
      [] -> true
      | hd :: tl -> member(hd, c) &&
      evalExpressions(tl, c))



type {Conditions, Checks} = List(Expression) *
List(Event)
type Environment = Context * Queue
type Queue = List(Event)

op waitStatements :
      List(Event) * Queue -> Boolean

axiom a1 is  fa(c : Checks, b : Block, e : Environment)
      ~(checkVariables(c, e)) =>
      evalBlock(c, b, e) = (aborted, e)

axiom a2 is  fa(c : Checks, b : Block, e : Environment)
      checkVariables(c, e) =>
      evalBlock(c, b, e) =    evalBlockBody(b, e)

axiom a3 is  fa(c : Checks, e : Environment)
      checkVariables(c, e) = checkExpressions(c.1, e.1)
      && waitStatements(c.2, e.2)

axiom a4 is  fa(le : List(Expression), c : Context)
```

```
      checkExpressions(le, c)= (case le of
      [] -> true
      | hd :: tl -> member(hd, c) &&
      checkExpressions(tl, c))

axiom a5 is fa(le : List(Event), q : Queue)
      waitStatements(le, q)  = (case le of
      [] -> true
      | hd :: tl ->
      (case q of
            Nil -> false
            | Cons (q1, others) ->
            if q1 = hd then waitStatements(tl, others)
            else false))

op evalContingency : Queue -> Boolean
op contingencyStatus : Event -> ContingencyStatus

type ContingencyStatus = | error | nominal | warning

axiom b1 is fa(lc : Conditions, s : Statements, e :
Environment)
      ~(evalPreconditionBody(lc, e)) =>
      evalModule(lc, s, e) = (aborted, e)

axiom b2 is fa(lc : Conditions, s : Statements, e :
Environment)
      evalPreconditionBody(lc, e)=>
      evalModule(lc, s, e)= evalStatements(s, e)

axiom b3 is fa(lc : Conditions, e : Environment)
      evalPreconditionBody(lc, e) =
      evalBooleanExpressions(lc.1, e.1)&&
      evalContingency(e.2)

axiom b4 is fa(le : List(Expression), c : Context)
      evalBooleanExpressions(le, c) = (case le  of
      [] -> true
      | hd :: tl -> member(hd, c) &&
      evalBooleanExpressions(tl, c))

axiom b5 is  fa(q : Queue)
      evalContingency q =(case q of
      Nil -> false
      | Cons (hd, tl) -> contingencyStatus hd = nominal)
```

Endspec

«Translate» du langage DSLα

```
translate spec

type Status= | aborted |completed
type Event
type Checks = List (Expression) * List (Event)
type Block
type Environment = Context * Queue
type Expression
type Context = List (Expression)
type Queue = List (Event)

op evalBlock :
      Checks* Block * Environment -> Status *
Environment
op evalBlockBody :
      Block * Environment -> Status * Environment
op checkVariables :
      Checks * Environment -> Boolean
op checkExpressions :
      List (Expression) * Context -> Boolean
op waitStatements :
      List (Event) * Queue -> Boolean

axiom a1 is fa (c : Checks, b : Block, e : Environment)
      ~ (checkVariables (c, e)) => evalBlock (c, b, e) =
      (aborted, e)

axiom a2 is fa (c : Checks, b : Block, e : Environment)
      checkVariables (c, e) => evalBlock (c, b, e) =
      evalBlockBody (b, e)

axiom a3 is fa (c : Checks, e : Environment)
      checkVariables (c, e) = checkExpressions (c.1,
e.1)  && waitStatements (c.2, e.2)

axiom a4 is fa (le : List (Expression), c : Context)
      checkExpressions (le, c) =  (case le of
      | [] -> true
      | hd::tl -> member (hd, c) &&
```

```
      checkExpressions  (tl,  c))

axiom a5 is fa (le : List (Event), q : Queue)
      waitStatements (le, q) =(case le of
      | [] -> true
      | hd::tl -> (case q of
            | [] -> false
            | q1::others ->
            if q1 = hd then waitStatements (tl, others)
            else false))

endspec by {
      Checks +-> Conditions,
      Block +-> Steps,
      evalBlock +-> evalProcedure,
      checkVariables +-> evalPrecondition,
      evalBlockBody +-> evalMainBody,
      checkExpressions +-> evalExpressions
      }
```

Langage DSLγ-α

```
spec

type Status = | aborted | completed
type Event
type Conditions = List(Expression) * List(Event)
type Steps
type Environment = Context * Queue
type Expression
type Context = List(Expression)
type Queue = List(Event)

op evalProcedure :
      Conditions * Steps * Environment -> Status *
      Environment
op evalMainBody :
      Steps * Environment -> Status * Environment
op evalPrecondition :
      Conditions * Environment -> Boolean
op evalExpressions :
      List(Expression) * Context -> Boolean
op waitStatements :
      List(Event) * Queue -> Boolean
```

```
axiom a1 is fa(c : Conditions, b : Steps, e :
Environment)
      ~(evalPrecondition(c, e)) =>
      evalProcedure(c, b, e) = (aborted, e)

axiom a2 is fa(c : Conditions, b : Steps, e :
Environment)      evalPrecondition(c, e) =>
      evalProcedure(c, b, e) =      evalMainBody(b, e)

axiom a3 is  fa(c : Conditions, e : Environment)
      evalPrecondition(c, e) =
      evalExpressions(c.1, e.1) && waitStatements(c.2,
      e.2)

axiom a4 is fa(le : List(Expression), c : Context)
      evalExpressions(le, c) = (case le of
      |[] -> true
      | hd :: tl -> member(hd, c) &&
      evalExpressions(tl,c))

axiom a5 is  fa(le : List(Event), q : Queue)
      waitStatements(le, q) = (case le of
      |[] -> true
      | hd :: tl ->  (case q of
            Nil -> false
            | Cons (q1, others) ->  if q1 = hd then
                  waitStatements(tl, others)
                  else false))

endspec
```

«*Translate*» de DSLβ

```
translate spec

type Status= | aborted |completed
type Conditions = List (Expression) * List (Event)
type Statements
type Environment = Context * Queue
type Expression
type Event
type Context = List (Expression)
type Queue = List (Event)
type ContingencyStatus = |nominal |warning | error
```

```
op evalModule :
      Conditions* Statements * Environment -> Status *
Environment
op evalStatements :
      Statements * Environment -> Status * Environment

op evalPreconditionBody :
      Conditions * Environment -> Boolean
op evalBooleanExpressions :
      List (Expression) * Context -> Boolean
op evalContingency :
      Queue -> Boolean
op contingencyStatus :
      Event -> ContingencyStatus

axiom b1 is fa (lc : Conditions, s : Statements, e :
Environment)
      ~( evalPreconditionBody (lc, e)) =>
      evalModule (lc, s, e) = (aborted, e)

axiom b2 is fa (lc : Conditions, s : Statements, e :
Environment)
      evalPreconditionBody (lc, e) =>
      evalModule (lc, s, e) = evalStatements (s, e)

axiom b3 is fa (lc : Conditions, e : Environment)
      evalPreconditionBody (lc, e) =
      evalBooleanExpressions (lc.1, e.1) &&
      evalContingency (e.2)

axiom b4 is fa (le : List (Expression), c : Context  )
      evalBooleanExpressions (le, c) =  (case le of
      | [] -> true
      | hd::tl -> member (hd, c) &&
evalBooleanExpressions  (tl, c))

axiom b5 is fa (q : Queue)
      evalContingency (q) =  (case q of
      | [] -> false
      | hd::tl -> contingencyStatus (hd) = nominal)

endspec by
      {
      Statements +-> Steps,
      evalModule +->evalProcedure,
      evalPreconditionBody +-> evalPrecondition,
      evalStatements+-> evalMainBody,
      evalBooleanExpressions+-> evalExpressions
      }
```

Langage DSLγ-β

```
spec

type Status = | aborted | completed
type Conditions = List(Expression) * List(Event)
type Steps
type Environment = Context * Queue
type Expression
type Event
type Context = List(Expression)
type Queue = List(Event)
type ContingencyStatus = | error | nominal | warning

op evalProcedure :
      Conditions * Steps * Environment -> Status *
      Environment
op evalMainBody :
      Steps * Environment -> Status * Environment
op evalPrecondition :
      Conditions * Environment -> Boolean
op evalExpressions :
      List(Expression) * Context -> Boolean
op evalContingency :
      Queue -> Boolean
op contingencyStatus :
      Event -> ContingencyStatus

axiom b1 is  fa(lc : Conditions, s : Steps, e :
Environment)
      ~(evalPrecondition(lc, e)) =>
      evalProcedure(lc, s, e) = (aborted, e)

axiom b2 is fa(lc : Conditions, s : Steps, e :
Environment)
      evalPrecondition(lc, e) =>
      evalProcedure(lc, s, e) = evalMainBody(s, e)

axiom b3 is fa(lc : Conditions, e : Environment)
      evalPrecondition(lc, e) =
      evalExpressions(lc.1, e.1)  &&
evalContingency(e.2)
```

```
axiom b4 is  fa(le : List(Expression), c : Context)
      evalExpressions(le, c) = (case le  of
      [] -> true
      | hd :: tl -> member(hd, c) &&
      evalExpressions(tl, c))

axiom b5 is fa(q : Queue)
      evalContingency q = (case q of
      Nil -> false
      | Cons(hd, tl) -> contingencyStatus hd = nominal)

endspec
```

Bibliographie

[1] MEDIADICO, http://www.mediadico.com/sommaire.asp

[2] IEEE standard computer dictionary : a compilation of IEEE standard computer glossaries, IEEE Computer Society Press, New York, NY, USA, 1991

[3] EICTA building digital europe, EICTA White Paper on Standardization and Interoperability, Brussels, Novembre 2006

[4] Gustavo A. Ospina Agudelo, Un cadre conceptuel pour l'interopérabilité des langages de programmation, Thèses doctorant, Université Catholique de Louvain, Belgique, 15 Février 2007

[5] Jihed Touzi, Aide à la conception de système d'information collaboratif, support de l'interopérabilité des entreprises, Thèse doctorant, Ecole des Mines d'Albi Carmaux, 9 Novembre 2007

[6] Kwang-Soon Choi, Young-Choong Park, Yang-Keun Ahn, Kwang-Mo Jung, Suk-I Hong, Design of Interoperable Module between Two Home Network Middlewares, IEEE Transactions on Consumer Electronics, Vol. 1, pages 314-318, 2005

[7] Tokunaga, E., Ishikawa, H., Morimoto, Y. and Nakajima, T, A Framework for Connecting Home Computing Middleware, In ICDCSW Conference, 2000

[8] Kyeong-Deok Moon, Young-Hee Lee, Chang-Eun Lee, and Young-Sung Son, Design of a universal middleware bridge for device interoperability in heterogeneous home network middleware, IEEE Trans. On CE, Vol. 51, Issue 1, pages 314-318, 2005

[9] OMG, Object Menagement Groupe, ptc/06-05-01, 2006

[10] A. Kaplan, J. Ridgway, and J.C. Wileden, Why IDLs are not ideal, In Proceedings of the 9th International Workshop on Software Specification and Design, pages 2–7, 1998

[11] OMG, The Object Management Group, http://www.omg.org/

[12] William Gropp, Ewing Lusk et Rajeev Thakur, Using MPI-2. Advanced Features of the Message-Passing Interfaces, MIT Press, 1999

[13] John Poole, Dan Chang, Douglas Tolbert, David Mellor, Common Warehouse Metamodel: An Introduction to the Standard for Data Warehouse Integration, 2001

[14] D. Konstantas, J.-P. Bourrières, M. Léonard, N. Boudjlida, Interoperability of Enterprise Software and Applications, Springer, Workshops of the INTEROP-ESA 2005 International Conference (EI2N), WSI, ISIDI and IEHENA 2005, 2005

[15] INTEROP, http://www.interop.com/

[16] Kernighan, W. Brian, Dennis M. Ritchie, The C Programming Language (1st ed.), Englewood Cliffs, NJ: Prentice Hall, 1978

[17] Stroustrup, Bjarne, The C++ Programming Language (Special Edition ed.), Addison-Wesley. p. 46

[18] Jon Byous, Java technology: The early years, Sun Developer Network, 1998

[19] ANSI, Fortran 77 standard X3.9-1978

[20] P. H. Winston et B. K. P. Horn, LISP, Addison Wesley, 1981

[21] Oliveira, Rui (2006), The Power of Cobol, BookSurge Publishing

[22] Laurent Réveillère, Approche langage au développement de pilotes de périphérique robustes, Thèse Doctorant, Renne 1, IFSIC/IRISA, 2001

[23] Erwann Poupart, Ali Abou Dib, Louis Féraud, Pierre Bazex, Ileana Ober, Christian Percebois, Thierry Millan, Vers une abstraction d'une famille de DSL, Dans : Génie Logiciel, GL & IS, Meudon - France, Vol. 81, pages 24-31, juin 2007.

[24] Elizabeth Bjarnason, APPLAB: A Laboratory for Application Languages, Proceedings of NWPER'96, Nordic Workshop on Programming Environment Reaserch, LU-CS-TR : 96-177, LUTEDX/(TECS-3073)/1-5, 1996

[25] CELKO Joe, SQL Avancé, International Thomson Publishing 1997

[26] Steven Feuerstein, Oracle PL/SQL Programming Guide to Oracle8i Features, O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472, 1999

[27] Bill Evjen, Kent Sharkey, Thiru Thangarathinam, Michael Kay, Alessandro Vernet, Sam Ferguson, refXML: Professional XML Programmer to Programmer, WROX 2007

[28] B.R.T. Arnold, A. Van Deursen et M. Res, An Algebraic specification of a language describing financial products, IEEE Workshop on formal Methods' Application in Software Engineering, pages 6-13, 1995

[29] N.K. Gupta, L. J. Jagadeesan, E. E. Koutsfios, et D. M. Weiss, Auditraw : Generating Audits the FAST Way, Proceedings of the 3rd IEEE International Symposium on Requirements Engineering, pages 188-197, 1997

[30] Martin Fowler, Language Workbenches: The Killer-App for Domain Specific Languages?, 12 juin 2005

[31] M. Hicks, P. Kakkar, J.T. Moore, C.A.Gunter, PLAN: A Packet Language for Active Networks, International Conference on Functional Programming(ICFP), Proceedings of the third ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland, US, pages 86-93, 1998

[32] James Neighbors, Software Construction using components, Thèse de doctorant, University of California, Irvine, 1980

[33] R. McCain, Reusable software component construction - A product-oriented paradigm, Computers in Aerospace Conference, 5th, Long Beach, CA, UNITED STATES, pages 125-135.1985

[34] G. Arango, Domain Analysis : From Art Form to Engineering Discipline, Proc. of the 5th International Workshop on Software Specifications and Design, pages 152-159, 1989

[35] Rubén Prieto-Diaz, Domain Analysis: An Introduction, ACM SIG SOFT Software

Engineering Notes,Vol. 15, Issue 2, pages 47-54, Anneé 1990

[36] David L. Parnas, On the design and development of program families, Software fundamentals : collected papers by David L. Parnas, pages: 193 - 213, 2001

[37] D.M. Weiss, Commonality Analysis: A Systematic Process for defining Families, Development and evolution of software architectures for product families, International ESPRIT ARES workshop No2, Las Palmas de Gran Canaria, ESPAGNE, Vol. 1429, pages. 214-222, 1998

[38] D.M. Weiss, Software Synthesis: The FAST Process, Proceedings ofthe International Conference on Computing in High Energy Physics, Rio de Janeiro, Brazil, September 1995

[39] Maarit Harsu, FAST product-line architecture process, Technical report 29, Institute of Software Systems, Tampere University of Technology, January 2002

[40] Scott Thibault, Domain-Specific Languages : Conception, Implementation, and Application, Thèse de doctorant, Univerity of Rennes 1, France, October 1998

[41] Jonathan Gennick, Oracle SQL*Plus: The Definitive Guide, O'Reilly, 1999

[42] Ken J Mcdonell, Instructional SQL (ISQL) user's guide, Technical report. Monash University. Dept. of Computer Science, 1988

[43] Olivier Dubuisson, ASN.1 - Communication between heterogeneous systems, Elsevier Science Ltd 2001

[44] Alexis Nédélec, Systèmes de Gestion de Bases de Données, Cours 2000/2001, Ecole d'ingénieurs de Brest

[45] Daniel Martin, Stockage et interopérabilité en XML, Expert Assermente en Informatique, FRANCE

[46] Tim Lindholm, Frank Yellin, The Java Virtual Machine Specification, Second Edition, Publisherd by Prentice Hall, part of the Java Series, 1999

[47] R Snodgrass, The interface description language : definition and use, Computer Science Press, Inc. New York, NY, USA, 1989

[48] Yannis Bres, Bernard Paul Serpette, Manuel Serrano, Compiling Scheme programs to .NET common Intermediate Language, Inria Sophia-Antipolis, 2004

[49] Don Symea, ILX: Extending the .NET Common IL for Functional Language Interoperability, Proc. BABEL Workshop on Multi-Language Infrastructure and Interoperability.ACM, 2001

[50] R.W. Floyed, The paradigms of programming, Springer India, in co-publication with Indian Academy of Sciences, Vol. 10, Number 5, 2005

[51] T. Budd, Multiparadigm Programming in Leda, 1st edition, Addision-Wesley Longman Publishing CO., Inc, Boston, MA, USA, 1994

[52] Sylvain Egger, infoscience, Ecole d'ingenierurs et d'architecture de Fribourg, Projet de diplôme, 2007

[53] P. Van Roy et S. Haridi, Concepts, Techniques, and Models of Computer Programming, Book, MIT Press, Cambridge, USA, 2004

[54] K. Gybels, R. Wuyts, S. Ducasse and M. D'hondt, Inter-language reflection: A conceptual model and its implementation, Computer Languages, Systems & Structures, Vol. 32, Issues 2-3, pages 109-124, 2006

[55] De Meuter, W. Agora, The story of the simplest MOP in the world. In Prototype-based Programming, Springer Verlag, 1998

[56] F. Rivard, SMALLTALK: A reflective language, dans Informal Proceedings of Reflection'96, 1996, pp. 21-38

[57] R. Wuyts, A logic meta-programming approach the co-evolution of object-oriented design and implementation, Thèse doctorant, Vrije University Brussel, 2001

[58] Valery Trifonov, Zhong Shao, Safe and principled language interopration, Springer-Verlag, Proceeding of the European Symposium on programming, Vol 1576, pages 128-146, 1999

[59] ESAPS, Engineering Software Architectures, Processes and Platforms for System-Families, Introduction to domain analysis., Délivrables du projet ESAPS, 2001. http://www.esi.es/esaps

[60] Northrop, A Framework for Software Product Line Practice –Version 3.0, Software EngineeringInstitute (SEI), 2002, http://www.sei.cmu.edu/pLdP/framework.html#framework_toc

[61] Anastapoulos, Implementing Product Line Variability, Technical report IESE report N°: 089.00/E, Franhofer IESE publication, 2000

[62] C. Atkinson, J. Bayer, and D. Muthig, Component-based product line development the KobrA approach, dans Proc. of the 1st Software Product Lines Conference (SPLC1), pages 289–309, 2000

[63] Griss.M L., Implementing Product-line Features by Composing Component Aspects, Dans Proceedings of the First Software Product Line Conference, P. Donohoe, pp. 271-288, 2000

[64] Kiczales, Aspect-Oriented Programming, Dans ECOOP'97 –Object Oriented Programming 11th European Conference, 1997

[65] Ziadi Tewfik, Helouët Loïc, Jezequel Jean-Marc, Modélisation de Lignes de Produits en UML, LMO 2003 Langages et modèles à objets, Vannes FRANCE,2003, vol. 9, no 1-2, pages 227-240

[66] M. Claub, Modeling variabilities with UML, GCSE 2001, Young Workshop, 2001

[67] OMG, Draft RFP for Common Variability Modeling (CVL), The OMG meeting in San Antonio, Texas, USA, 14-18 September 2009

[68] Jon Oldevik, Øystein Haugen, Birger Møller-Pedersen, Confluence in Domain-Independent Product Line Transformations, FASE 2009, Lecture Notes in Computer Science

5503, 34-48, Springer 2009

[69] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, Andreas Svendsen : Adding Standardized Variability to Domain Specific Languages., Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, 139-148, Proceedings. IEEE Computer Society, 2008

[70] Schobbens, P.-Y.; Heymans, P.; Trigaux, J.-C., Feature Diagrams: A Survey and a Formal Semantics, Requirements Engineering, 14th IEEE International Conference, pp.139-148, 11-15 Sept. 2006

[71] Andreas Svendsen, Gøran K. Olsen, Jan Endresen, Thomas Moen, Erik Carlson, Kjell-Joar Alme, Øystein Haugen, The Future of Train Signaling., MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings. Lecture Notes in Computer Science 5301, 128-142

[72] Olivier Ponsini, Des programmes impératifs vers la logique équationnelle pour la vérification, Thèse doctorant, Université de Nice - Sophia Antipolis, Laboratoire I3S, 24 Novembre 2005, Projets LANGAGES/COPRIN

[73] J. McCarthy, Towards a mathematical science of computation, http://www-formal.stanford.edu/jmc/Version étendue du papier IFIP 1962, 1996

[74] Scott, D., and C. Strachey, Towards a mathematical semantics for computer languages. , Proc. Symposium on Computers and Automata, Microwave Research Institute Symposia Series, Vol. 21. A Polytechnic Institute of Brooklyn, New York, pages 19-46, 1971

[75] Joseph E. Stoy, Denotational semantics : the Scott-Strachey approach to programming language theory, MIT Press, Cambridge, Massachusetts, 1977

[76] Isabelle Linden, Etude de langages de coordination: analyse d'expressivité, sémantiques et méthodologies de programmation, Rapport de rechercher, Institute of informatics, University of Namur, Namour, Belgium, FUNDP, 2005

[77] Jean Goubault-Larrecq, Sémantique dénotationnelle, Cours 2004 : LSV/CNRS UMR 8643 & ENS Cachan

[78] Yves Bertot, Sémantique des langages de programmation deuxième partie: sémantique dénotationnelle, Cours, Ecole Doctorale STIC, février 2008

[79] C. Pair, La formalisation des langages de programmation, Mathématiques et sciences humaines, Tome 34, pages 71-86, 1971

[80] C.A.R. Hoare, An Axiomatic Basis for computer programming, Communication of the ACM, Vol. 12, Issus 10, pages 576-580, 1969

[81] Edsger W. Dijkstra, A Discipline of programming, 1st edition, Prentice Hall PTR Upper Saddle River, NJ, USA, pages: 240, 1997

[82] Laurent Mounier et Gil Utard, Sémantique axiomatique des langages à parallelisme de données: automatisation de la preuve de programmes, INIST-CNRS, Cote INIST : RP 13479, Rapport LIP - 93-08, pages 48, 1993

[83] G. A. Goguen, G. Malcolm, Algebraic semantics of imperative programs, MIT Press, 1996

[84] G. A. Goguen, J.W. Thatcher, E.G. Wagner, J.B Wright, Abstract data types as initial algebras and the correctness of data representation, Current trends in programming methodology, pages 80-149, 1978

[85] Donald Sannella et Andrzej Tarlecki, Essential concepts of algebraic specification and program development, Formal Aspects of computing, Springer London, Vol. 9, Number 3, pages 229-269, 2005

[86] C. Fédèle, Construction automatisée des compilateurs : les systèmes CIGALE , Thèse de doctorant, Université de Nice - Sophia Antipolis, janvier 1991

[87] Didier Buchs, Sémantique élémentaire des langages, cours, Universite de Genève, 2008

[88] O. Ponsini, Des programmes impératifs vers la logique équationnelle pour la vérification, Thèse de doctorant, Université de Nice - Sophia Antipolis, janvier 2005

[89] Eric G. Wagner, Algebraic Specification: some old history and new thoughts, Nordic Journal of computing, Vol. 9, Issue 4, pages 373-404, 2002

[90] J.V. Guttag, Abstract data type and the development of data structure, Communication of the ACM,Vol. 20, Issus, pages 396-404, 1977

[91] H. Ehrig et B. Mahr, Fundamentals of Algebraic specification 1. Equations and Initial Semantics, EATCS Monographs on Theoretical computer Science, Springer-Verlag Vol. 6, Issue 21, 1985

[92] J.V. Guttag and J.J. Horming, LARCH: Languages and Tools for Formal Specification, Springer-Verlag New York, Inc. New York, NY, USA. pages 250, 1993

[93] K. Futatsugi, J. Goguen, J.P. Jounnaud, et J. Mesguer, Principles of OBJ2, Annual Symposium on Principles of Programming Languages, Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New Orleans, Louisiana, United States pages: 52 - 66, 1985

[94] Marie-Claude Gaudel, Structuring and Modularizing Algebraic specifications: the PLUSS specification language, evolutions and perspectives, Springer Berlin / Heidelberg, Vol. 577/1992, pages 1-18, 2006

[95] E Astesiano, M Wirsing, An introduction to ASL, The IFIP TC2/WG 2.1 Working Conference on Program specification and transformation, Germany, pages: 343 - 365, 1987

[96] Kestrel Development corporation and KestreL Technology LLC, Specware to Isabelle In Interface Manual, 2006-2009

[97] Tobias Nipkow, Lawrence C. Paulson et Markus Wenzel, A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Sciene (LNCS), Vol. 2283, pages 218, 2002

[98] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser and J. Visser, The ASF+SDF Meta-Environment: A Component-Based Language Development

Environment, Compiler Construction, LNCS, Vol. 2027/2001, pages 365-370, Springer Verlag, 2001

[99] Christian Percebois, Pierre Bazex, Arlette Sebatware, Hervé Leblanc, Ali Abou Dib, Ileana Ober, Louis Féraud., Modélisation en UML et OCL de langages dédiés, Dans : Journée KERMETA 2006, Rennes, France, 10-OCT-06

[100] Arlette Sebatware, Ali Abou Dib, Pierre Bazex, Christian Percebois, Vers la modélisation UML de la syntaxe et de la sémantique d'un langage de programmation., Dans : Séminaire Interopérabilité pour les procédures opérationnelles, IAS, Toulouse, France, 12-APR-06

[101] Ivan Kurtev, Jean Bézivin, Frédéric Jouault et Patrick Valduriez, Model-based DSL Frameworks, OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages. 602-616, 2006

[102] Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wasowski., Guided development with multiple domain-specific languages, In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, MoDELS, volume 4735 of Lecture Notes in Computer Science, pages 46–60. Springer, 2007

[103] S. Mac Lane, Category for the working mathematician(Second edition), Springer, Vol. 5 in the series Graduate Texts in Mathematics, 1998

[104] M.B. Smyth et G. D.Plotkin, The category-theoretic solution of recursive domain equations, SIAM J. Comput. Vol. 11, Issue 4, pages 761-783, 1982

[105] J.L. Fiadero, Categories for Software Engineering, Springer Berlin Heidelberg New York, 2005

[106] J.A. Goguen, G. Malcom, Software Engineering with OBJ: Algebraic Specification in Action, Kluwer Academic Publishers, 2000

[107] M. Bidoit, PD. Mosses, CASL User Manual: Introducing Common Algebraic Specification Language, LNCS 2900, Springer, 2004

[108] J.L. Fiadero, T. Mainbaum, Categorical Semantics of Parallel Programs Design, Science of Computer Programming 28 (2-3), pages. 111-138, 1997

[109] J.A. Goguen, R.M. Burstall, Institutions: Abstract Model Theory for Specification and Programming, Journal of the Association for Computing Machinery, vol. 39, no, pp. 95-146, 1992

[110] T. Mossakowski, J. Goguen, R. Diaconescu, A. Tarlecki, What is a Logic ?, Logica Universalis, J.-Y. Beziau (Ed.), pp. 113–1352006

[111] G. Hongge et J. Weyman, An approach to Automation of Fusion Using Specware, Proceedings of the Second International Conference on Information Fusion, pages 109-116, 1999

[112] Kestrel, Specware documentation, http://www.specware.org/doc.html

[113] Jean-Guy Schneider et Oscar Nierstrasz, Components, Scripts and Glue, Software

Architectures – Advances and Applications, pages 13- 25, 1999

[114] Keith Williamson, Michael Healy et Richard Barker, Industrial Applications of Software Synthesis via Category Theory - Case Stady Using Specware, Automated Software Engineering,Springer Netherlands, Vol. 8, Issue 1, pages: 7 - 30, 2001

[115] R.A.G. Seely, Locally cartesian closed categories and type theory. Math, Proc. Camb. Phil. Soc., 95 (33) : 33-48, 1984

[116] Michael Barr et Charles Wells, Category theory for Computing Science, Prentice-Hall International Series In Computer Science, pages 432, 1990

[117] Andrea Asperti et Giuseppe Longo, Categories, Types and Structures, Category Theory for the working computer scientist. M.I.T. Press, pages 1 - 300, 1991

[118] Catherine Oriat, Etude des spécifications modulaires : constructions de colimites finies, diagrammes, isomorphismes, Thèse doctorant, Laboratoire logicielle systeme et réseaux LSR-IMAG, Institue National Polytechnique de Grenoble, 1996

[119] J. Dumas, D. Duval, Vers une modélisation diagrammatique de la bibliothèque C++ d'algèbre linéaire LinBox, Actes des journées Langages et Modéles à Objets, LMO'06. Nîmes, Vol 12/HS, pages 117-132, 2006

[120] Douglas Smith, Composition by Colimit and Formal Software Development, Algebra, Meaning, and Computation (A Festschrift in honour of Prof. Joseph Goguen), LNCS4060 Vol. 4060, Page 317-332, 2006.

[121] ECSS-E-70-32, Space engineering, Grand systems and operations- procedure definition language, Auropean Cooperation For Space Standardization, ECSS, Noordwijk, the Netherlands, Draft 23, April 2004

[122] A. Abou Dib, L. Féraud, I. Ober, C. Percebois, « Towards a rigorous framework for dealing with domain specific language families, International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT 2008), Damascus, Syria, April 2008

[123] H. Ehrig et M. GroBe-Rhode et U. Wolter, On the Role of Category Theory in the Area of Algebraic Specification, The 11th Workshop on Specification of Abstract Data Types Joint with the 8th COMPASS Workshop on Recent Trends in Data Type Specification,LNCS, Vol. 1130,pages 17-48, 1995

[124] Mark E. Stickel, Richard J. Waldinger, Vinay K. Chaudhri, A Guide TO Snark, Technical Note Unassigned. AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, 2000

[125] Kenneth Baclawski, Mieczyslaw M. Kokar, Richard Waldinger et Paul A. Kogut, Consistency Checking of Semantic Web Ontologies , ISWC2002 : International Semantic Web Conference No1, ITALIE (09/06/2002),Vol. 2342, pages 454-459, 2002

[126] Makarius Wenzel, Clemens Ballarin, Florian Haftmann, Tobias Nipkow,larry Paulson, Sebastian Skalberg, The Isabelle/Isar Reference Manual, Part of the Isabelle2005 distribution, http://isabelle.in.tum.de/doc/isar-ref.pdf.

[127] Alexander Krauss, Defining Recursive Functions in Isabelle/HOL, Departmentof Informatics, Technische Universität München,2007, http://isabelle.in.tum.de/doc/functions.pdf, June 8,2008. efining Recursive Functions in Isabelle/HOL. Departmentof Informatics, Technische Universität München,2007, http://isabelle.in.tum.de/doc/functions.pdf, June 8,2008. A Tutorial, ISABELLE, Workshop, Bermen, Germany, July2007, Now part of the official Isabelle documentation

[128] Makarius Wenzel and Stefan Berghofer, The Isabelle System Manual, Documentation Isabelle, TU MÜnuchen, 19 April2009

[129] Jean-Marie Favre, Concepts fondamentaux de l'IDM - De l'Ancienne Egypte à l'Ingénierie des Langages, 2èmes Journées sur l'Ingénierie Dirigée par les Modèles, IDM06, pages 13-16, Lille, 26-28 juin2006

[130] I. Ober, A. Abou Dib, L. Féraud, C. Percebois, Towards interoperability in component based development with a family of DSLs, Proceedings of the 2nd European conference on Software Architecture,Springer-Verlag, LNCS, Papos, Chprus,Vol. 5292, pages 148-1631, 2008

[131] Frédéric Minne, Baudouin Le Charlier, Gustavo Ospina, Kim Mens, Etude de l'interopérabilité de deux langages de programmation basée sur la machine virtuelle de Java, Mémoire académique UCL, 2002-2003

[132] D. Pavlovic, D. R. Smith, Software Development by Refinement, 10th Anniversary Colloquium of UNU/IIST, LNCS 2757, pp. 267-286, 2002

[133] M. Aiguier, R. Diaconescu, Stratified institutions and elementary homomorphisms, Information Processing Letters, 103, pp. 5-13, 2007

[134] K. Mens, D. Ordonez Camacho, Using Annotated Grammars for the Automated Generation of Program Transformers, Actes des 3es Journées sur l'Ingénierie Dirigée par les Modèles (IDM2007). Eds. Antoine Beungard & Marc Pantel, pages 7–24, 2007

[135] D. Ordonez Camacho, K. Mens, M. Van Den Brand et J. Vinju, Automated Derivation of Translators From Annotated Grammars, Electronic Notes in Theoretical Computer Science, Vol. 164(2), pages 121-137,2006

[136] J. Van Der Brand, A. Van Deursen, J. Heering, HA. de Jong, M. de Jong, T. Kuipers, P. Klint, L. Moonen, PA. Olivier, J. Scheerder, JJ. Vinji, E. Visser, J. Visser, The ASF+SDF Meta-Environment: a Component-Based Language Development Environment

[137] J. Heering et M. Mernik, Domain-specific languages as key tools for ULSSIS engineering, International Conference on Software Engineering,Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems,Germany, pages 1-2, 2008

[138] Ileana Ober, Ali Abou Dib, Using ASM to achieve executability within a family of DSL, International Conference on ASM, B and Z, London, 17/09/2008-19/09/2008, Vol. LNCS, Egon Boerger, Michael Butler, Jonathan P. Bowen, Paul Boca (Eds.), Springer-Verlag, p. 354-374, septembre2008.

[139] Yuri Gurevich, Margus Veanes et Charles Wallace, Can Abstract State Machines Be

Useful in Language Theory?, Theoretical Computer Science 376 (2007) 17-29

[140] F. Jouault, J. Bézivin, C. Consel, I. Kurtev, and F. Latry., Building DSLs with AMMA/ATLa Case Study on SPL and CPL Telephony Languages, Proceedings of the 1st ECOOP Workshop on Domain-Specific Program Development (DSPD), Nantes, France, 2006

[141] D.D.O. Ossami, J. Souquières, J.-P. Jacquot, Opérations de construction de spécifications mult-vues UML et B, », Conférence AFADL'2004, Besançon, 2004

[142] A. Mota, A. Sampaio, Model-checking CSP-Z : Strategy, tool support and industrial application, Science of Computer Programming, Vol. 40, Issue 1, pages 59-96, May2001

[143] D. Batory, M. Azanza, J. Saraiva, The Objects and Arrows of computational Design, Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, LNCS Vol. 5301, Toulouse, France, pages 1-20, 2008