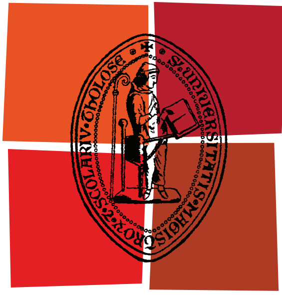


THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Réseaux, Télécommunications, Systèmes et Architecture - RTSA

Présentée et soutenue par :
Suzy Hélène Germaine TEMATE NGAFFO

le : lundi 12 novembre 2012

Titre :

Des langages de modélisation dédiés aux environnements de méta-modélisation dédiés

---

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
IRIT - Institut de Recherche en Informatique de Toulouse

Directeur(s) de Thèse :
Daniel Hagimont
Laurent Brotto

Rapporteurs :
Pr. Michel Riveill
Pr. Franck Barbier

Membre(s) du jury :
Pr. Noël De Palma
Mcf Daniel Hagimont
Pr Laurent Brotto

À mon loubh et à notre princesse...

Avant de vouloir qu'un logiciel soit
réutilisable, il faudrait d'abord qu'il ait
été utilisable.
Ralph Johnson

Je tiens à remercier

Monsieur Noël De Palma, Professeur à l'université Joseph Fourrier de Grenoble qui m'a fait l'honneur de présider mon jury de thèse,

Monsieur Franck Barbier, Professeur à l'université de Pau et des pays de l'Adour et Monsieur Michel Riveill, Professeur à l'université de Nice Sophia Antipolis qui ont accepté de rapporter cette thèse. Ils ont évalué mon travail d'une manière approfondie et m'ont fait des remarques constructives,

J'exprime ma profonde gratitude à Monsieur Daniel Hagimont, Professeur à l'Institut National Polytechnique de Toulouse et à Monsieur Laurent Broto, Maître de Conférence à l'Institut National Polytechnique de Toulouse qui m'ont encadré durant ces année de dur labeur. Merci pour leurs conseils avisés, leur pédagogie et le temps qu'ils m'ont consacré pour que je puisse mener cette thèse à son terme.

Au delà du jury, je tiens à remercier tous ceux qui ont permis à ces travaux d'aboutir, par leurs conseils, leurs contributions et leurs encouragements.

- Merci à Monsieur Michel Daydé, directeur de l'Institut de Recherche en Informatique de Toulouse pour son soutien.
- Merci à tous les membres de l'équipe Sépia pour leur accueil. Je remercie plus particulièrement tous les doctorants Sépia du site IRIT/ENSEEIHT (Larissa, Aeiman, Alain, Sun, Esso, Momo et Brice) pour les moments très agréables et toutes les blagues (même les ratées) que nous avons partagés. Vos encouragements plus particulièrement durant la phase de rédaction m'ont été d'une grande aide.
- Merci à Sylvie Armengaud et à Sylvie Eichen pour les services rendus. Leur travail a grandement contribué à faciliter le mien.
- Merci à tous les amis et amies que j'ai en dehors du cadre professionnel (Épifanie, Valérie, Nélia, Paulin, Filo, Flora, Fotina, Clara, William, Marie-Ange et les autres qui m'excuseront de ne pas les nommer). J'ai pu apprécié votre amitié durant ces années et j'ai toujours pu compter sur votre soutien.

Enfin, je garde une place toute particulière pour ma famille, pour tout ce qu'ils ont pu (et continuent à) m'apporter. Merci à mes beaux-parents pour leur soutien et pour avoir si souvent gardé ma fille pour me permettre d'aller travailler. Merci à mes frères et sœurs pour leur grand soutien et leurs encouragements malgré la distance. Merci à mon mari Laurent pour m'avoir encouragée et avoir toujours cru en moi. Merci pour sa patience et pour sa présence à mes côtés. Merci à ma fille Sarah pour la tendresse innocente qu'elle m'apporte quand je me sens débordée. Enfin, merci à mes parents pour m'avoir fourni toutes les bonnes conditions pour réussir mes études et pour m'avoir apporté, en plus de leur amour, le goût de la réussite.

Les langages dédiés (DSL) sont de plus en plus utilisés parce qu'ils permettent aux utilisateurs qui ne sont pas des experts en programmation d'exprimer des solutions avec des langages simples qui capturent l'expertise de leur domaine. C'est encore plus vrai pour les langages dédiés graphiques (DSML) qui ont un niveau d'abstraction plus élevé que les langages dédiés de programmation.

Implémenter un DSML revient généralement à fournir un éditeur dédié qui permette aux utilisateurs de manipuler les abstractions de leur domaine (d'instancier le langage). Les expériences ont montré que l'implémentation d'un tel éditeur dédié graphique est coûteuse en termes de temps et de ressources humaines. Nous constatons que la plupart des plates-formes permettant de construire ce type d'éditeur (EMF/GMF, DSL Tools, Obeo Designer, ...) sont génériques. Elles essayent d'adresser le maximum de domaines possibles, ce qui les rend complexes et inadaptées à des cas d'utilisation spécifiques. Si la spécialisation aux domaines a été un succès pour les langages, pourquoi ne pas l'appliquer aux plates-formes de construction d'éditeurs ?

Cela reviendrait à concevoir pour un domaine donné, une plate-forme permettant de construire facilement des éditeurs dédiés pour ce domaine. Cette plate-forme n'aurait pas les défauts d'une plate-forme totalement générique parce qu'elle serait restreinte au domaine ciblé. Ce type de plate-forme spécifique à un domaine, nous l'appelons Domain Specific Modeling Framework (DSMF).

Le principal inconvénient d'un DSMF est qu'on ne peut l'utiliser que dans le cadre du domaine pour lequel il a été conçu. Cela implique qu'il faille construire un DSMF par domaine et c'est une solution coûteuse. Toutefois, nous pensons que cette approche sur les DSMF peut être généralisée afin d'adresser un grand nombre de domaines. Cette thèse a donc consisté à concevoir et à implanter un environnement qui permet de construire des DSMF de façon modulaire.

Domain Specific Languages (DSLs) are increasingly used in many fields as they allow users to express strategies without being programming experts. This is particularly true for graphical DSLs called Domain Specific Modeling Languages (DSMLs) which are more intuitive than programming DSLs.

Implementing a DSML means providing a specific editor which allow users to express the language's constructions (instantiate the language). Many experiments showed that implementing specific graphical editors is much manpower consuming. Our analysis is that most frameworks for building such editors (e.g. EMF/GMF) are generic, i.e. aim at fulfilling the requirements of any field, which leads to increased complexity and costs a lot in terms of development time. If domain specialization was successful for languages, why don't we apply it to frameworks ? Specializing such a framework according to the constraints of a domain would allow keeping the definition of a specific editor simple, while fulfilling the requirements of the considered domain.

Domain specific frameworks for building DSML editors in specific application fields is a promising approach. Such a framework does not have the limits of generic frameworks because it is restricted to a particular domain. It is more intuitive and simpler to use as it only proposes abstraction of the domain for building DSMLs. We call this type of framework Domain Specific Modeling Framework (DSMF). For example, if we consider the component domain, there are several DSMLs in this domain which share the same layout requirements. We implemented a DSMF for this family of DSMLs. This DSMF is specialized according to the constraints and layout requirements of the component domain (Components, connectors, Bindings, . . .). This specialization allows simple and rapid generation of specific editors devoted to component-based architectures.

The principal drawback of a DSMF is its restricted scope to one specific domain. This approach requires to develop one DSMF per domain and the development cost can be significant. A solution may be to generalise the DSMF approach in order to address many application fields. We designed a Generic framework for building DSMFs in a modular way. This thesis is based on the implementation of this framework.

Table des matières

I Cadre des travaux **3**

1 Administration autonome **5**
1.1 Généralités 5
1.2 Définition 6
1.3 Fonctionnement 6
1.4 Avantages 8
1.5 Synthèse 8

2 TUNe **11**
2.1 Cas d'utilisation 11
2.2 Historique 12
2.3 Principe général 12
2.3.1 *Architecture Description Langage* (ADL) 13
2.3.2 *Wrapping Description Langage* (WDL) 14
2.3.3 *Reconfiguration Description Language* (RDL) 15

3 Expérimentations de TUNe **16**
3.1 Applications à large échelle : le cas de DIET 16
3.2 Applications virtualisées 18
3.3 Synthèse 19

II Problématique **22**

4 Langages dédiés **24**
4.1 Besoin de langages dédiés 24
4.2 Définition 25
4.3 Ingénierie Dirigée par les Modèles 27
4.3.1 Approches existantes 28
4.4 Conception d'un DSML 32
4.4.1 Syntaxe 33
4.4.2 Sémantique 34
4.5 Synthèse 35

5 Limites des DSML et problématique **37**

5.1 Coût de développement 37
5.2 Limitations liées à la délimitation du domaine 38
5.3 Limitations liées aux outils de méta-modélisation 39
5.3.1 Générique Vs. Spécifique 40
5.4 Synthèse 43

6 État de l'art **44**
6.1 Introduction 44
6.2 Ce que fournit un environnement de méta-modélisation 45
6.3 Critères d'évaluation 47
6.4 Implantation de MDA : *Eclipse Modeling Project* (EMP) 49
6.4.1 EMF 49
6.4.2 GEF 50
6.4.3 GMF 50
6.4.4 Spécification de la syntaxe abstraite 51
6.4.5 Spécification de la syntaxe concrète 52
6.4.6 Spécification de la sémantique 54
6.4.7 Génération de l'éditeur 54
6.4.8 Synthèse 54
6.5 Autre implantation de MDA : DiaMeta 55
6.5.1 Spécification de la syntaxe abstraite 57
6.5.2 Spécification de la syntaxe concrète 57
6.5.3 Spécification de la sémantique 57
6.5.4 Génération de l'éditeur 58
6.5.5 Synthèse 58
6.6 Implantation des usines à logiciel : VMSDK 58
6.6.1 Spécification de la syntaxe abstraite 59
6.6.2 Spécification de la syntaxe concrète 60
6.6.3 Spécification de la sémantique 61
6.6.4 Génération de l'éditeur 61
6.6.5 Synthèse 61
6.7 Implantation de MIC : GME 62
6.7.1 Spécification de la syntaxe abstraite 64
6.7.2 Spécification de la syntaxe concrète 64
6.7.3 Spécification de la sémantique 64
6.7.4 Génération de l'éditeur 65
6.7.5 Synthèse 65
6.8 Autre implantation de MIC : MetaEdit+ 66
6.8.1 Spécification de la syntaxe abstraite 67
6.8.2 Spécification de la syntaxe concrète 67
6.8.3 Spécification de la sémantique 68
6.8.4 Génération de l'éditeur 68

6.8.5 Synthèse . . . 69
6.9 Synthèse générale . . . 69

III Contributions **73**

7 Orientations générales **75**
7.1 Vers la spécialisation des outils de méta-modélisation . . . 75
7.1.1 Rôle des utilisateurs . . . 75
7.1.2 Utilité des assistants . . . 76
7.1.3 Vers des outils spécifiques . . . 76
7.1.4 Coût des outils de méta-modélisation spécifiques . . . 76
7.2 Notre approche . . . 77
7.2.1 Exigences d'un outil de méta-méta-modélisation . . . 80

8 Description de GyTUNe **82**
8.1 Introduction . . . 82
8.2 Choix d'implantation . . . 82
8.2.1 Modèle à composant Fractal . . . 84
8.3 Conception de GyTUNe . . . 87
8.3.1 Séparation entre fonctionnalités génériques et fonctionnalités spécifiques . . . 88
8.3.2 Définition des concepts au niveau M3 . . . 88
8.4 Description de l'architecture de GyTUNe . . . 89
8.4.1 *Common* . . . 90
8.4.2 *GyTUNe specific* . . . 95
8.5 Fonctionnement général . . . 96
8.5.1 userM3 : spécification de la syntaxe abstraite . . . 96
8.5.2 userM3 : spécification de la syntaxe concrète . . . 98
8.5.3 userM3 : spécification de la sémantique . . . 101
8.5.4 UserM2 : Spécification de la syntaxe abstraite . . . 102
8.5.5 userM2 : spécification de la syntaxe concrète . . . 103
8.5.6 userM2 : spécification de la sémantique . . . 104
8.5.7 userM2 : Génération de l'outillage . . . 104
8.5.8 userM1 : définition d'un modèle . . . 105

9 Validation **107**
9.1 Introduction . . . 107
9.2 Cas d'utilisation : architectures à composants . . . 107
9.2.1 Création de l'outil de méta-modélisation . . . 108
9.3 Cas d'utilisation : gestion de projets . . . 115
9.3.1 Création de l'outil de méta-modélisation . . . 117
9.4 Synthèse . . . 125

10 Conclusion et perspectives **127**
10.1 Conclusion . . . 127
10.2 Perspectives . . . 128
10.2.1 Perspectives à court terme . . . 129
10.2.2 Perspectives à long terme . . . 130

Introduction

Les logiciels aujourd'hui sont devenus de plus en plus complexes à administrer. Cette complexité est apparue avec le nombre sans cesse croissant de fonctionnalités requises, la généralisation des environnements répartis, et plus particulièrement avec l'utilisation de grilles de calcul à grande échelle et d'applications multi-tiers. Cette complexité est devenue telle qu'aujourd'hui une administration effectuée essentiellement par des humains est trop coûteuse, source d'erreurs et pas assez réactive. Une solution consiste à utiliser un logiciel d'administration autonome permettant d'effectuer des tâches d'administration telles que le déploiement, la réparation ou encore l'optimisation en limitant les interventions humaines.
Nous avons conçu et développé un tel logiciel que nous appelons TUNe. TUNe utilise des langages qui reposent sur UML pour définir ses politiques d'administration. Les expériences réalisées avec TUNe ont montré l'inadéquation des langages d'administration de TUNe lorsque l'on aborde des environnements logiciels très différents. Par exemple, les langages de TUNe n'étaient pas adaptés pour l'administration des applications à large échelle ou encore pour des applications virtualisées. Ces expériences ont également mis en évidence que les langages d'administration doivent êtres adaptés en fonction de la facette d'administration et du domaine d'application. Ce type de langage spécifique à un domaine est appelé langage dédié.

Dans ce contexte du projet TUNe, nous nous sommes intéressé aux outils permettant de définir des langages d'administration dédiés, afin de faciliter la définition des politiques pour les administrateurs. Pour fournir de tels outils, nous nous sommes tournés vers les plates-formes existantes de méta-modélisation qui permettent d'implanter des DSL graphiques communément appelés *Domain Specific Modeling Language*. Ces plates-formes se sont toutes révélées complexes d'utilisation. Elles requièrent de l'utilisateur une trop grande expertise en programmation, expertise qu'il n'a pas forcément. Les outils de méta-modélisation existants essayent d'être les plus génériques possible et d'adresser tous les domaines applicatifs, mais plus ils sont génériques plus leur complexité augmente. Nous pensons que pour être efficaces ces outils de méta-modélisation permettant d'implanter des DSMLs doivent être eux-mêmes spécifiques à un domaine. Spécialiser ainsi un outil en fonction d'un domaine permettrait de ne pas tomber dans les travers des outils génériques et d'avoir un outil plus simple d'utilisation, parce que restreint. Nous appelons un tel outil spécifique à un domaine *Domain Specific Modeling Framework* (DSMF).

Le principal inconvénient d'un DSMF est qu'on ne peut l'utiliser que dans le cadre du domaine pour lequel il a été conçu. Cela implique qu'il faille construire un DSMF par domaine et c'est une solution coûteuse. Toutefois, nous pensons que cette approche sur les DSMF peut être généralisée afin d'adresser un grand nombre de domaines. Nous proposons dans cette thèse, la conception et le développement d'une plate-forme générique permettant de créer des DSMFs de façon modulaire. Une telle plate-forme permet de réduire le coût de production d'un outil de méta-modélisation dédié et ce faisant, favoriserait l'adoption des DSMLs à large échelle. Notre plate-forme appelée GyTUNe repose sur le modèle à composant fractal et permet de définir les outils de méta-modélisation sous

forme d'architecture à composants. Chaque composant peut alors être adapté, configuré ou remplacé en fonction des spécificités d'un domaine et on peut ainsi créer un nouvel outil de méta-modélisation sans avoir besoin de le développer entièrement.

Ce document est organisé de la manière suivante :

- **Première partie : Cadre des travaux**
  Cette partie présente le contexte du projet qui a motivé ces travaux. Elle est composée des chapitres 1, 2 et 3. Dans le chapitre 1 nous présentons le contexte scientifique de cette thèse qui est l'administration autonome. Le chapitre 2 présente le système d'administration autonome TUNe. Le chapitre 3 présente les expérimentations menées avec TUNe, expérimentations qui ont soulevé le besoin de langages dédiés et motivé ces travaux.
- **Deuxième partie : Problématique**
  Cette partie présente la problématique de cette thèse et un état de l'art autour de cette problématique. Elle regroupe les chapitres 4, 5 et 6. Le chapitre 4 présente les langages dédiés et les avantages qu'ils offrent. Le chapitre 5 présente les limites des langages dédiés. Le chapitre 6 présente un état de l'art des différents outils de méta-modélisation existants. Nous avons testé chacun de ces outils dans le but d'implanter des langages dédiés pour TUNe. Nous décrivons chaque outil et son mode de fonctionnement.
- **Troisième partie : Contribution**
  Cette partie présente la contribution de cette thèse. Elle regroupe les chapitres 7, 8 et 9. Dans le chapitre 7 nous présentons l'idée de cette thèse qui consiste premièrement à spécialiser les outils de méta-modélisation permettant d'implanter des DSMLs afin de les rendre plus accessibles. Deuxièmement, nous proposons la mise en œuvre d'une plate-forme générique qui permet de générer à moindre coût des outils de méta-modélisation dédiés. Dans le chapitre 8 nous décrivons l'implantation de cette plate-forme générique dédiée à la création d'outils de méta-modélisation dédiés. Nous motivons nos choix d'implantation, décrivons l'architecture générale de la plate-forme et son utilisation. Le chapitre 9 présente deux cas d'utilisation de notre plate-forme dans deux domaines différents qui illustrent la validité de notre approche.

Première partie

Cadre des travaux

Chapitre 1

Administration autonome

Dans ce chapitre nous présentons le contexte de nos travaux qui est l'administration autonome. Nous menons depuis quelques années un projet qui vise la construction d'un système autonome pour l'administration de grandes infrastructures logicielles. L'étude des langages d'administration dans ce système a soulevé une problématique plus générale autour des langages dédiés, et cette problématique fait l'objet de cette thèse.

1.1 Généralités

Ces dernières années, on a pu observer une complexité sans cesse croissante des environnements logiciels. Les raisons de cette augmentation sont diverses et variées. Premièrement, les logiciels de nos jours sont construits pour être les plus complets possible et fournir un maximum de fonctionnalités, ils deviennent donc très sophistiqués, mais aussi très difficiles d'utilisation. Deuxièmement, ces logiciels sont très souvent composés de plusieurs tiers ou nécessitent d'être déployés dans des environnements distribués ce qui introduit une nouvelle dimension de difficulté. Troisièmement, l'augmentation du nombre d'utilisateurs ainsi que la multiplication de leurs besoins sont autant de facteurs supplémentaires qui contribuent à rendre les systèmes informatiques encore plus difficiles à appréhender. Toute cette complexité induit un coût considérable dans l'administration des systèmes informatiques. Il faut toujours plus d'administrateurs pour y faire face. Mais l'administration de ces systèmes opérée uniquement par des hommes a montré ses limites, d'abord parce qu'elle n'est pas suffisamment réactive, mais aussi parce qu'elle est source d'erreurs et de pannes. Il devient donc aujourd'hui nécessaire de s'affranchir le plus possible de l'intervention humaine dans les tâches d'administration afin de limiter le coût d'exploitation de ces systèmes. D'où l'idée intéressante de développer des systèmes d'administration qui seraient automatiques et plus abstraits afin de simplifier les tâches d'administration et de réduire les interventions humaines. Nous présentons dans la suite cette approche d'administration dite *autonome* qui consiste à administrer un système informatique à l'aide d'un autre système informatique.

1.2 Définition

Administrer un système informatique consiste à mettre en œuvre un ensemble de tâches ou fonctions relatives à la gestion de ce système. [Berrayana 2006] donne une vue d'ensemble de ces tâches d'administration qui sont multiples :

- Configuration et déploiement : la première tâche de l'administration consiste à configurer et à déployer le système. Cela veut dire choisir et assembler les composants du système et les placer dans un environnement matériel ;
- Réparation de pannes : l'occurrence d'une panne (matérielle, réseau ou logicielle) peut faire basculer le système dans un état d'erreur et le rendre indisponible pour les besoins des usagers. Le système n'est alors plus à même d'assurer le minimum de services requis par ses utilisateurs. Il est donc important de le réparer de façon à le ramener à une configuration qu'il avait avant la panne ;
- Gestion de performance : Le système doit assurer une certaine qualité de service. Il doit, par conséquent être adapté aux différentes variations de performances et aux pics de charge ;
- Gestion des ressources : Nous entendons par ressources toutes les ressources mémoires, CPU, disques, réseau et machines nécessaires au fonctionnement du système. La gestion des ressources qui a un impact important sur les performances doit tenir compte de la qualité de service ;
- Gestion de la sécurité : Le système doit être capable de parer à toute attaque malveillante.

Les *systèmes d'administration autonome* sont des systèmes capables d'effectuer des tâches d'administration de manière autonome, sans requérir d'intervention humaine. Le système d'administration autonome fournit un support pour le déploiement et la configuration des applications. Il fournit également un support pour la supervision de l'environnement administré et permet de définir des réactions face à des événements comme des pannes ou des surcharges afin de reconfigurer les applications administrées de façon autonome. Toutefois, la mise en œuvre de chacune de ces tâches d'administration s'effectue conformément à une *politique d'administration* définie par un opérateur humain. Les politiques d'administration définissent comment chacune des tâches d'administration doit être effectuée. Pour l'administrateur humain, les tâches d'administration prennent alors une nouvelle forme, qui consiste à définir des politiques d'administration de haut niveau. Ces politiques seront interprétées et mises en application automatiquement par le système d'administration autonome. Nous verrons par la suite que pour rendre l'administration plus efficace, il est important que ces politiques d'administration soient définies en fonction du système à administrer.

1.3 Fonctionnement

Le schéma de fonctionnement d'un système d'administration autonome représenté par la Figure 1.1, suit essentiellement trois étapes [Taton 2008].

1. *L'observation*, qui permet d'évaluer l'état du système administré et de collecter les données qui caractérisent son comportement ;
2. En fonction des observations et de l'état courant du système administré, le système d'administration autonome peut prendre des décisions conformément aux *politiques d'administration* définies par l'administrateur humain ;
3. *Action*, qui correspond à la mise en œuvre effective des opérations d'administration.

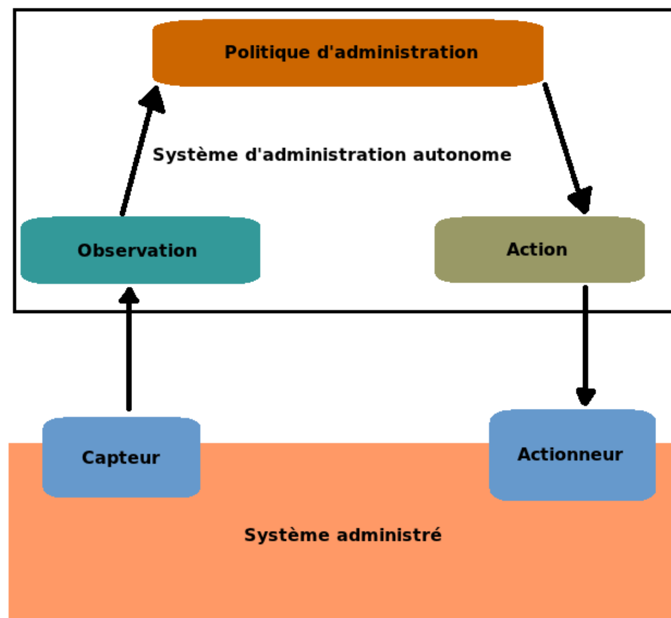


FIGURE 1.1 – Organisation d'un système d'administration autonome

Le système d'administration autonome communique avec l'élément administré grâce aux *capteur*s et aux *actionneur*s. Les capteurs et les actionneurs se servent des interfaces implantées par le système administré pour observer son état et pour le modifier.
De nombreux travaux se sont appuyés sur un modèle à composants pour concevoir un système autonome. Un composant est une unité de structuration qui a des interfaces permettant de modéliser des services fournis et des services requis. Le principe général d'un système d'administration basé sur des composants est d'encapsuler les éléments administrés dans des composants et d'administrer l'environnement logiciel comme une architecture à composant. L'administrateur bénéficie alors des atouts de ce modèle à composants à savoir l'encapsulation, les outils de déploiement et les interfaces de reconfiguration, qui lui permettent d'implanter des procédures d'administration autonome. Cette approche offre une vision uniforme de l'environnement logiciel et matériel à administrer.

Il existe plusieurs solutions d'administration autonome basées sur des modèles à composant :

- **ArchStudio** [Oreizy et al. 1998] : C'est une approche basée sur le modèle à composant C2 [Taylor et al. 1995]. Elle permet l'auto-administration d'applications écrites en C2. ArchStudio met un accent sur la reconfiguration et le maintien de la cohérence ;
- **O. kephart** [Hanson et al. 2004] : O. Kephart propose une approche originale qui s'appuie sur des composants autonomes et un système qui s'auto administre grâce à ces différents composants autonomes ;
- **Rainbow** [Garlan et al. 2004] : Rainbow est un canevas dédié à la construction de systèmes auto-adaptables. Il utilise un modèle de composants hiérarchiques pour la représentation explicite de l'architecture du système administré ;
- **Jade** [Bouchenak et al. 2005] et **TUNe** [Broto et al. 2008] : Deux solutions d'administration autonome basées sur le modèle à composant fractal.
- ...

Les travaux présentés ici se situent dans la même veine que le projet TUNe, projet que nous présentons un peu plus en détail dans la section suivante.

1.4 Avantages

L'administration autonome offre plusieurs avantages :

- **Réduction du nombre d'erreurs et de pannes** : le système d'administration autonome dirige les opérations d'administration, réduisant ainsi les erreurs et les pannes introduites par des opérateurs humains ;
- **Système plus réactif** : le système d'administration maintient une surveillance étroite et continue de l'intégralité du système administré et est ainsi capable de réagir promptement. L'intervention assurée par l'administrateur humain implique des interventions évaluées en minutes ou heures. Mais l'utilisation d'un système d'administration autonome fait passer le temps d'intervention à la seconde ou milliseconde dans certains cas, le système est alors beaucoup plus réactif ;
- **Économie de ressources** : Les administrateurs sont remplacés par le système d'administration et il y a alors économie de ressources humaines. De plus en utilisant un système d'administration autonome, les ressources informatiques peuvent êtres allouées à la demande pour encore plus d'économie.

1.5 Synthèse

L'administration autonome est une approche prometteuse pour l'administration des systèmes complexes. Tous les avantages qu'elle présente justifient que le monde industriel et celui de la recherche s'y intéressent. Cette approche permet d'optimiser l'administration des environnements logiciels. L'administrateur humain est libéré de ces tâches automatisées, souvent laborieuses et répétitives, et peut désormais mettre à profit son expertise en

se concentrant sur des tâches d'administration de plus haut niveau. Ces dernières incluent principalement la définition des politiques d'administration. Toutes les opérations d'administration reposent sur la définition de ces politiques. Nous allons montrer que cette définition doit sans cesse être adaptée en fonction du système administré. Pour ce faire, nous nous appuyons sur les expérimentations menées dans le cadre du projet TUNe dans lequel j'ai effectué ma thèse.

Chapitre 2

TUNe

Les travaux décrits dans ce document se situent dans le contexte d'un projet qui vise à concevoir et implanter un système d'administration autonome appelé TUNe [Broto et al. 2008]. Nous avons constaté quelques lacunes au niveau des formalismes utilisés dans TUNe pour l'administration. La problématique de cette thèse est issue de ces constats, elle met en évidence le besoin de politiques d'administration adaptables. Dans ce chapitre nous présentons l'historique de TUNe et ses principes fondamentaux. Le chapitre suivant présentera les différentes expériences réalisées avec TUNe ainsi que les limitations que ces expériences ont mises en évidence.

2.1 Cas d'utilisation

À des fins d'illustration, les exemples utilisés dans cette section se réfèrent à une application JEE [Microsystems 200x]. Il s'agit d'une application web dynamique organisée suivant une architecture n-tiers (Figure 2.1). Elle est composée de trois serveurs :

- Apache [Apache 200x] : le serveur web qui reçoit les requêtes clientes. Lorsqu'il reçoit une requête pour une page statique, il retourne directement cette page au client. Mais lorsque la page demandée nécessite d'être générée dynamiquement, il l'envoie au serveur d'applications qui se charge de la générer. Dans certains cas, un répartiteur de charge (HA-Proxy par exemple) peut être associé à un ou plusieurs serveurs apaches afin de traiter une grande quantité de requête ;
- Tomcat [Tomcat 200x] : le serveur d'application qui à la demande d'Apache, se charge de la construction d'une page web à la volée qui est renvoyée au client. Apache et Tomcat communiquent via des connecteurs (par exemple ModJK [ModJK 200x] pour Apache et AJP [Berg 2008] pour Tomcat) ;
- MySQL [MySQL 200x] et MySQL-Proxy : le serveur de base de données qui permet de relier le serveur d'applications à plusieurs sources de données MySQL via un connecteur JDBC. En cas de besoin, le serveur d'applications fait appel au ser-

veur de données MySQL pour le retrait ou la sauvegarde de données venant des clients web.

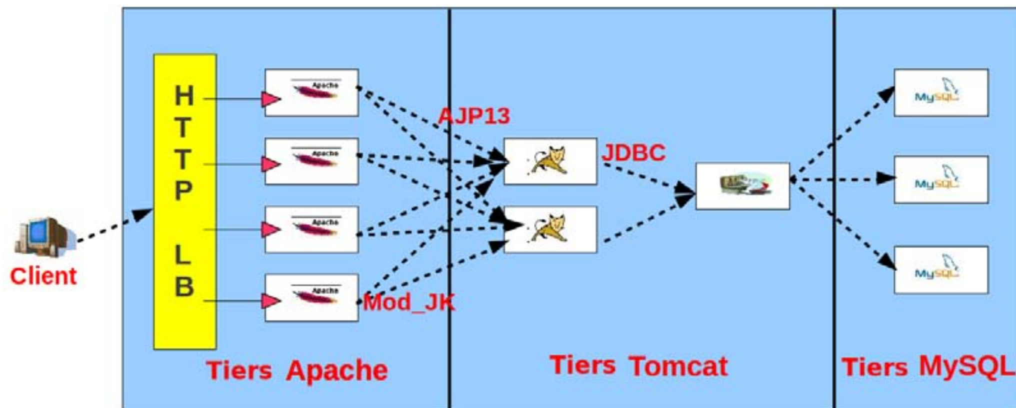


FIGURE 2.1 – Exemple d'une architecture JEE

2.2 Historique

TUNe se situe dans la continuité du projet Jade [Bouchenak et al. 2005]. Jade est une approche d'administration autonome développée en 2004 à l'INRIA à Grenoble, qui permet de déployer des applications patrimoniales et de les reconfigurer. Une application patrimoniale est une application ayant des interfaces d'administration spécifiques et n'ayant pas été conçue pour être administrée de façon autonome. Jade fournit un support pour l'administration de ces applications dans un environnement réparti et prend également en compte la supervision de l'environnement matériel administré.
Ce système d'administration est basé sur le modèle à composant Fractal [Bruneton et al. 2006] et plus précisément sur Julia, l'implémentation Java de ce modèle. Une des limitations majeures de cette approche est qu'elle nécessite que l'utilisateur ait des compétences en Fractal pour implanter les interfaces non fonctionnelles des composants et des compétences en Java pour implanter les politiques d'administration (configuration, reconfiguration ...). L'administrateur se retrouve en position de développeur alors qu'il n'en a pas forcément les compétences. Le projet TUNe qui fait suite à Jade a pour objectif d'élever le niveau d'abstraction de Jade et être plus accessible aux utilisateurs. Ainsi, TUNe fournit des langages de plus haut niveau pour l'expression des politiques d'administration. Ces langages reposent sur le formalisme UML qui est un standard beaucoup plus connu des utilisateurs. Dans la section suivante, nous présentons ces langages et leur utilisation dans TUNe.

2.3 Principe général

Tout comme Jade, TUNe est système d'administration autonome qui repose sur le modèle à composant fractal. TUNe a été conçu pour masquer les détails de Fractal dans Jade et s'appuie sur des profils UML pour la définition des politiques d'administration. Les langages fournis dans TUNe permettent de décrire les applications administrées, l'environnement matériel et les politiques de reconfiguration dynamique sans que l'administrateur ait besoin de programmer. Les sections suivantes présentent ces langages et leur utilisation dans TUNe. Pour une présentation détaillée de l'implantation et du fonctionnement de TUNe le lecteur peut se reporter à la thèse [Broto 2008].

2.3.1 *Architecture Description Langage* (ADL)

TUNe utilise le diagramme de classe UML pour décrire l'environnement logiciel ainsi que l'environnement matériel du système à administrer. TUNe peut ainsi tirer avantages des caractéristiques du diagramme de classe telles que l'instanciation, les multiplicités et la navigabilité (Cf problème 1 section 3.1). Cette description est appelée *Architecture Description Language* (ADL).

Pour l'environnement logiciel, chaque type d'application (serveur Web, BD ...) est représenté par une classe UML et les attributs assignés à cette classe décrivent les propriétés de ce type d'application. En plus des attributs propres aux applications, TUNe impose des attributs propres à son fonctionnement interne (par exemple les attributs *legacyFile* et *host-family* présentés dans la Figure 2.2). Chacune des classes représentées à la Figure 2.2(a) décrit un des logiciels qui constituent notre application JEE à administrer. Elles ont toutes un attribut *legacyFile* qui indique à TUNe l'archive contenant les binaires du logiciel concerné. Par contre, l'attribut *port* de la classe *Apache* par exemple représente une propriété propre au serveur web Apache en l'occurrence elle désigne son port d'écoute. L'ADL permet également de décrire les connexions entre les différents logiciels. Ces connexions sont représentées par des liaisons entre les classes représentant les logiciels concernés.
De façon générale la description de l'ADL est la base du processus d'administration dans TUNe parce que de sa définition dépend celle des autres langages.
Concernant l'environnement matériel, TUNe le considère comme étant un ensemble de groupes de machines (*clusters*). Chaque *cluster* est constitué d'un ensemble de machines homogènes (en termes d'OS, système de fichiers ...). Dans la description de l'environnement matériel, chaque *cluster* est représenté par une classe, tous les attributs assignés à cette classe sont imposés par TUNe et l'administrateur ne peut en rajouter. TUNe se sert de ces attributs pour le déploiement effectif des applications sur le *cluster*. L'attribut *nodefile* de la classe *Cluster* par exemple (Figure 2.2(b)) indique un nom de fichier contenant la liste des machines constituant le *cluster*. Quant à l'attribut *allocator-policy*, il désigne la politique d'allocation des logiciels sur les machines du *cluster*. Contrairement aux connexions de l'environnement logiciel, les connexions entre les *clusters* sont

des relations d'héritage d'attributs. Ces connexions servent uniquement à factoriser la description des attributs.

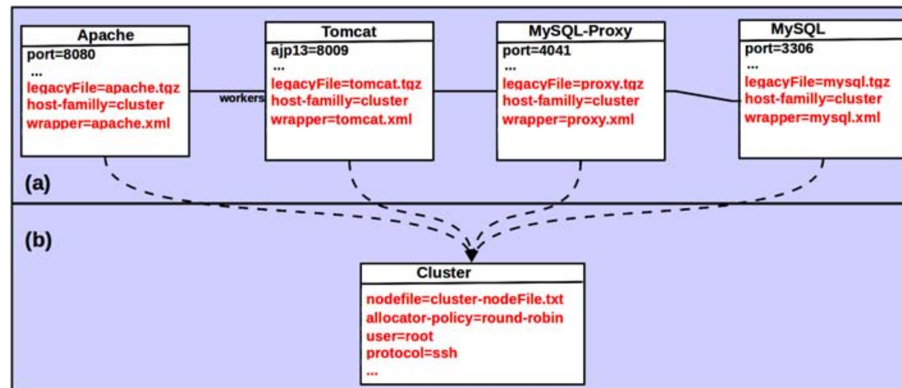


FIGURE 2.2 – Exemple d'ADL d'une application JEE

2.3.2 *Wrapping Description Langage* (WDL)

L'ADL permet une description structurelle de l'environnement logiciel et matériel administré. Pour décrire les opérations d'administration, TUNe propose un langage basé sur le formalisme XML appelé *Wrapping Description Langage* (WDL). Pour chaque type de logiciel décrit dans l'ADL, l'administrateur définit dans un fichier XML (que nous appellerons *wrapper*) l'ensemble des méthodes pouvant êtres appelées sur les instances logicielles de ce type. Une méthode de *wrapper* définit : (1) **la méthode** Java à appeler, c'est une méthode propre au logiciel auquel est associé le *wrapper* ; (2) **les paramètres** à lui donner, qui sont issus des attributs de l'ADL.
Par exemple pour le *wrapper* du logiciel *Apache* présenté à la Figure 2.3, définit une méthode *start* permettant de démarrer une instance d'*Apache*. Cette méthode prend en paramètres deux attributs issus de l'ADL, le numéro de port d'Apache (*"$port"*), et le numéro de port du connecteur AJP du serveur *Tomcat* lié à *Apache*.
Un *wrapper* permet ainsi de spécifier des actions d'administration de façon plus abstraite. L'utilisateur a juste à spécifier la méthode à appeler et les paramètres requis.

```
<wrapper name='Apache'>

    <method name="start" key="J2EEWrapping" method="apacheManager" >

        <param value="start" />
        <param value="$port" />
        <param value="$workers.ajp13" />

    </method>
    ...
</wrapper>
```

FIGURE 2.3 – Wrapper du logiciel Apache

Ces méthodes sont exécutées si elles apparaissent dans une des politiques de reconfiguration définies par l'administrateur. La section suivante présente comment les poli-

tiques de reconfiguration sont décrites et comment les fonctions des *wrappers* sont mises en œuvre.

2.3.3 *Reconfiguration Description Language* (RDL)

Pour définir les politiques de reconfiguration, TUNe propose un langage appelé *Reconfiguration Description Language* (RDL). Ce langage s'inspire des diagrammes états transitions d'UML et permet de faire des automates à états finis. Les reconfigurations sont déclenchées par l'apparition d'un événement dans l'environnement administré. À un type d'événement correspond un automate de reconfiguration. Un état de l'automate désigne une action à effectuer et les transitions déterminent l'ordre d'exécution des différentes actions. Il existe deux types d'actions possibles : la modification des attributs, la modification de l'architecture (ajout et suppression de logiciel) et l'appel des méthodes décrites dans les *wrappers*. La politique de reconfiguration dépend de l'application administrée. Le nombre d'automates requis sera fonction du nombre de types d'événements attendus par l'administrateur. Toutefois, deux automates particuliers sont requis par TUNe : un automate pour le démarrage et un automate pour l'arrêt de chaque logiciel administré. La Figue 2.4 montre un exemple d'automate décrivant la configuration et le démarrage d'une application JEE. On note que les serveurs peuvent êtres configurés parallèlement tandis que le démarrage se fait suivant un ordre donné (MySQL puis MySQL-Proxy puis Tomcat et enfin Apache).

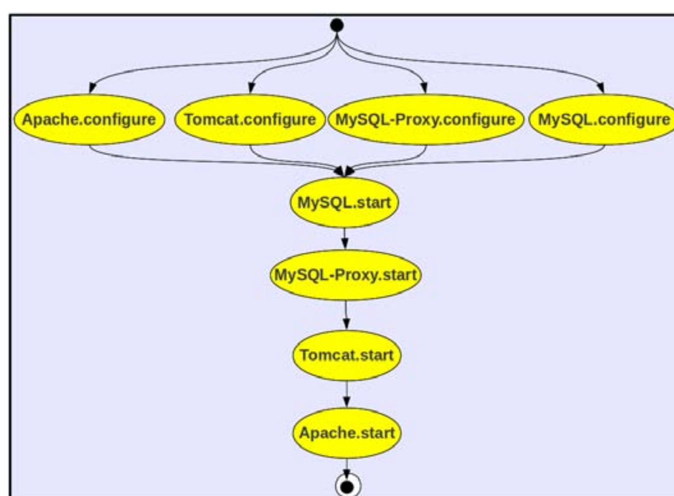


FIGURE 2.4 – Configuration et démarrage d'une application JEE

Depuis sa mise en œuvre, cette conception de TUNe a fait l'objet de plusieurs expérimentations dans divers domaines applicatifs. Ces expérimentations ont permis de montrer l'efficacité de TUNe mais elles ont aussi mis en évidence des failles dans sa conception. Le chapitre suivant présente quelques-unes de ces expérimentations ainsi que les limitations de TUNe notamment en termes de politiques d'administration inadaptées.

Chapitre 3

Expérimentations de TUNe

Les expérimentations réalisées avec le système TUNe couvrent plusieurs domaines d'application. Nous décrivons dans cette section deux d'entre elles, à savoir l'utilisation de TUNe pour l'administration d'une application large échelle et pour l'administration d'une application virtualisée. Il s'agit pour nous de montrer d'une part l'efficacité de TUNe et d'autre part de souligner les problèmes qu'ont soulevés ces expérimentations. Pour chaque problème nous présenterons le cas échéant la solution apportée. Toutes ces expérimentations soulèvent le besoin d'adapter les langages d'administration en fonction du contexte applicatif. Ce besoin d'adaptation des langages qui peut être généralisé au-delà de l'administration autonome, à toute l'ingénierie du logiciel, est la base de la problématique de cette thèse.

3.1 Applications à large échelle : le cas de DIET

DIET [DIET 200x] est un ordonnanceur composé de trois types de serveur appelés *agents* et organisé de façon arborescente. À la racine de l'arbre se trouvent les *Master Agents* (MA). Les MA reçoivent les demandes venant des clients et les orientent vers les serveurs de calculs les plus adaptés. Ces serveurs de calcul appelés *Server Daemon* (SeD) sont situés aux feuilles de l'arbre. Le SeD reçoit la requête du client, effectue les calculs requis et restitue les résultats au client. Lorsqu'il y a trop de SeD, pour éviter que les MA ne saturent, des serveurs appelés *Local Agent* (LA) sont introduis entre les MA et les SeD. Les LA font remonter aux MA des informations de monitoring qui leur permettent d'orienter la requête du client vers les SeD adéquats. En fonction de la taille de l'application, plusieurs niveaux de LA peuvent être introduits entre les MA et les SeD. La figure 3.1 résume l'architecture de cette application. Elle peut être constituée de milliers de serveurs lorsque l'environnement matériel le permet (cas de grid5000 [Badia and Nussbaum 2011]). L'utilisation de TUNe pour l'administration de cette application dans un contexte large échelle a montré quelques limites.

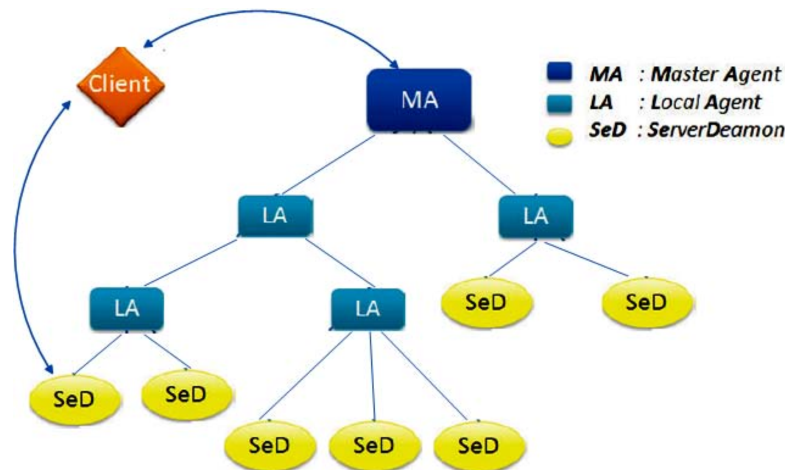


FIGURE 3.1 – Architecture de DIET

Problème 1 : la première limite concerne la description de l'ADL de l'application DIET. TUNe a été conçu pour l'administration des applications réparties de type client-serveur telles que JEE qui ne sont constituées que de quelques serveurs. De ce fait, les langages d'administration que propose TUNe répondent parfaitement aux besoins d'administration de ce type d'applications. Notamment, l'ADL permet de décrire entièrement chaque tiers et les différentes connexions entre eux. Mais quand il s'agit de décrire une application large échelle telle que DIET qui peut être composée de milliers de serveurs, cette description devient fastidieuse voir impossible. En effet dans sa version initiale, TUNe ne proposait qu'un ADL en *extension* c'est-à-dire une description explicite de chacun des logiciels qui composent l'application à administrer. Décrire ainsi des centaines de serveurs de calcul et les différentes connexions entre eux a mis en évidence l'inadéquation de l'ADL de TUNe pour les applications large échelle.
Pour résoudre ce problème, nous avons modifié l'ADL de TUNe afin qu'il permette une description plus intuitive de l'architecture à administrer. Nous l'appelons description en *intension*. Elle consiste à décrire non plus des logiciels, mais des types de logiciels. L'utilisation de cardinalités sur les liaisons (Figure 3.2) permet de spécifier un nombre minimum et un nombre maximum d'instances possible pour un type de logiciel. De plus, l'attribut *initial* permet à l'administrateur de spécifier le nombre d'instances voulues au démarrage pour un type de logiciel. Dans notre exemple DIET présenté Figure 3.2, cette description en intension va permettre d'instancier cinq LA et 100 SeD au démarrage.

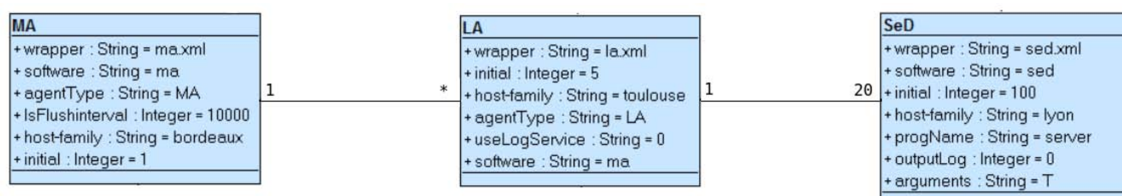


FIGURE 3.2 – Exemple d'ADL en intension d'une application DIET

Problème 2 : La deuxième limite concerne le déploiement des SeD. La performance de ces derniers est liée aux caractéristiques des machines sur lesquelles ils s'exécutent, car l'administrateur dispose de différentes versions d'installation de SeD en fonction du cluster où le SeD va être déployé. Mais TUNe ne permettait de spécifier qu'une seule archive pour un type de logiciel (cf l'attribut *legacyFile* Figure 2.2(a)). De plus, le choix de la version n'est possible que pendant le déploiement (c'est à dire une fois le cluster connu), il faudrait donc que l'administrateur puisse associer une version précise de SeD à ce moment-là.
Ce besoin applicatif nécessite la personnalisation du processus de déploiement, donc du langage associé dans TUNe.

Problème 3 : la troisième difficulté rencontrée durant l'administration de DIET concerne le caractère centralisé de TUNe. En effet, l'administration d'une application constituée de milliers de logiciels par une unique instance de TUNe devient impossible à cause des limites de la machine qui l'héberge. Par exemple, l'ouverture d'une connexion réseau entre la machine d'administration et tous les serveurs administrés est limitée par les capacités en termes de nombre de descripteurs de fichiers ouverts.
Pour résoudre ce problème, Toure [TOURE 2010] propose de hiérarchiser (mettre sous forme arborescente) l'administration des applications à large échelle. Il a créé plusieurs instances de TUNe, ainsi chaque instance s'occupe de l'administration d'une partie de l'application. Sans entrer dans les détails, la solution de Toure a nécessité la modification à la fois des langages d'administration et des mécanismes d'administration dans TUNe. Ainsi, le déploiement et les reconfigurations sont effectués de façon hiérarchique par plusieurs instances de TUNe.

Problème 4 : enfin, les automates de reconfiguration utilisés dans TUNe ne font intervenir que des états et des transitions. Ce type d'automates simples est adapté pour des petites applications. Pour des applications des applications à large échelle telle que DIET qui peut être déployé sur des milliers de nœuds, il serait judicieux de fournir des automates plus complets. De tels automates pourraient par exemple inclure des structures de contrôle qui permettraient de capturer au mieux toutes les actions de reconfiguration possibles. Pour intégrer cela, il faut rajouter de nouveaux langages dans TUNe.

3.2 Applications virtualisées

La virtualisation se définit comme étant l'ensemble des techniques matérielles ou logicielles qui permettent de faire fonctionner sur une seule machine physique plusieurs systèmes d'exploitation (OS invités) ou plusieurs applications, donnant l'illusion à l'utilisateur d'être en présence de plusieurs machines distinctes. Ces OS ou applications s'exécutent de machines dites virtuelles (*Virtual Machine*, VM). Une Machine virtuelle (*Virtual Machine*-VM) est la copie fidèle et isolée d'une machine physique. Cette copie est créée

par un logiciel (superviseur) et donne l'illusion d'être une véritable machine physique.
Nous avons expérimenté l'administration de VM avec TUNe dans le contexte du *Cloud Computing*. Le *Cloud Computing* est une plate-forme de mutualisation informatique fournissant aux entreprises des services à la demande avec l'illusion d'une infinité des ressources [Bakshi 2009]. L'unité d'allocation de ressources dans le *cloud* est la machine virtuelle, autrement dit, on alloue aux entreprises des portions de machines physiques sous forme de VM sur lesquelles ces entreprises peuvent déployer leurs applications (par exemple une application JEE). L'objectif de cette expérimentation consistait donc à administrer d'une part les VM fournies aux entreprises et d'autre part il fallait administrer les applications déployées sur ces VM. Le système TUNe s'est révélé être complètement inadapté pour l'administration des VM, son utilisation dans ce cadre a montré plusieurs limites.

Problème 1 : Les VM ont la particularité d'être des logiciels qui se comportent comme des machines physiques. Mais les langages et le cœur de TUNe n'ont pas été conçus pour prendre en compte cette double identité, ils différencient les machines des logiciels. Si la VM est considérée comme étant une machine alors TUNe ne pourra pas la configurer et l'administrer dynamiquement comme c'est le cas pour les logiciels. En effet TUNe déploie des logiciels sur des machines et ne prend pas en charge l'administration de ces machines. Si par contre la VM est considérée comme étant un logiciel, alors TUNe ne pourra pas déployer des logiciels sur cette VM, parce que TUNe ne prévoit pas le déploiement d'un logiciel sur un logiciel. La spécificité des VM soulève donc un problème : comment décrire et représenter les deux facettes d'une VM.
Pour résoudre ce problème, il a fallu modifier à la fois les langages d'administration et le cœur de TUNe afin de prendre en compte les spécificités des VM. L'ADL de TUNe a été modifié pour uniformiser la description des machines et des logiciels et pour cela il a fallu rajouter un autre langage.

Problème 2 : dans TUNe, l'installation d'un logiciel consiste à la copie du binaire correspondant sur une machine. Mais l'installation d'une VM ne se limite pas à la copie des binaires. En effet, déployer une VM nécessite le montage d'un système de fichier (NFS ou autre protocole). Pour intégrer ce type d'installation, il a fallu personnaliser la méthode de déploiement proposée par TUNe et donc modifier le langage de déploiement.

Le problème 2 a été résolu dans [Tchana 2011] sans support de langage et abordé dans [TOURE 2010] en adaptant le langage de déploiement.

3.3 Synthèse

Si l'on se réfère à [Taton 2008], on peut classer TUNe dans la catégorie des approches d'administration basées sur les langages. Ce type d'approche fournit en général

des langages pour la description des systèmes administrés, la gestion de leur déploiement ou de leur cycle de vie. La conception et l'implantation de TUNe sont réalisées autour de ces langages d'administration, mais les expériences réalisées avec TUNe ont montré l'inadéquation de ces langages dans certains cas. En effet la conception des langages d'administration de TUNe était initialement orientée pour l'administration des applications *clusterisées* de type client-serveur telle que JEE. Lors des expérimentations on a pu constater que lorsqu'on change de famille d'application (large échelle, machines virtuelles ...), ou de facette d'administration (gestion de performances, équilibrage de charge ...) les langages de TUNe deviennent inadaptés et nécessitent au mieux d'êtres modifiés et au pire n'existent pas et doivent êtres développés pour prendre en compte les spécificités de l'application à administrer.

Cette observation peut être généralisée à toutes les approches d'administration autonome basées sur des composants. Bien que toutes ces approches se soient affirmées comme étant pertinentes pour l'administration des systèmes complexes, elles partagent néanmoins un inconvénient commun qui est l'absence de politiques d'administration adaptables. Dans le cas de TUNe, tout le système est conçu autour de ces langages, mais les limites mises en évidence dans les différentes expérimentations montrent que les profils UML tels qu'utilisés dans TUNe sont insuffisants. De plus comme le souligne l'analyse [Combemale et al. 2008], les expérimentations réalisées avec TUNe ont révélé la variation des besoins d'administration en fonction des domaines d'application. Les langages d'administration doivent être spécifiques à la facette d'administration considérée et au domaine d'application. De tels langages qui sont conçus pour résoudre des problèmes d'un domaine particulier sont appelés langages dédiés (en anglais *Domain Specific Languages*, DSL).

La nouvelle orientation du projet TUNe est de proposer un système d'administration qui ne repose sur aucun langage ou domaine d'administration et de fournir au-dessus de ce système un ensemble d'outils permettant de définir les langages dédiés correspondants aux besoins précis des administrateurs. Ces langages dédiés pourraient êtres définis soit par les administrateurs eux-mêmes, soit en collaboration avec des experts langage qui spécialiseraient TUNe pour un contexte applicatif donné. La Figure 3.3 montre cette nouvelle orientation. La première partie de cette nouvelle approche est résolue dans [Tchana 2011] qui propose un système d'administration flexible et suffisamment générique pour supporter plusieurs domaines d'application (couche (1) Figure 3.3). Quant à la seconde partie (couche (2) de la Figure 3.3), elle constitue la motivation première de cette thèse. Nous voulons fournir une pile logicielle permettant de définir aisément des langages d'administration qui soient spécifiques à une facette et à un domaine d'application donnés. Ces langages d'administration, pour la plupart, doivent souvent être graphiques pour rester simples et expressifs. D'une façon plus générale, nous nous intéressons dans cette thèse à la définition et à l'implantation des langages dédiés graphiques dans l'ingénierie du logiciel.

La partie suivante présente plus en détail la problématique de cette thèse qui s'articule autour des langages dédiés graphiques. Nous allons définir la notion de langage dédié, présenter les enjeux de ce type de langage dans l'ingénierie du logiciel ainsi que leur implantation.

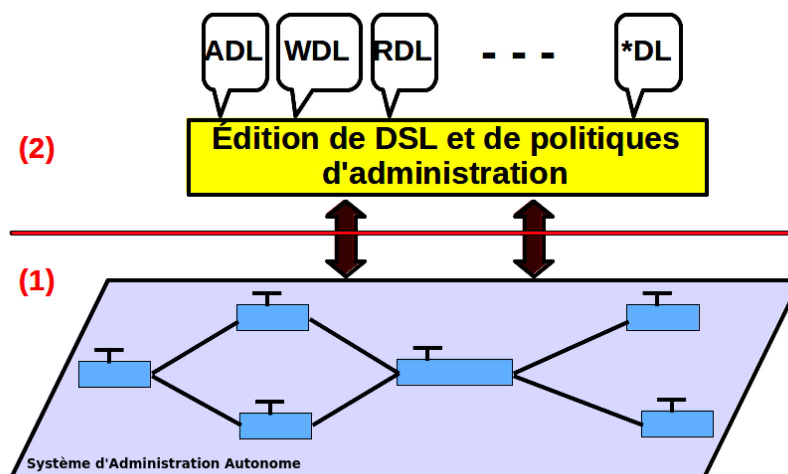


FIGURE 3.3 – Nouvelle orientation du projet TUNe

Deuxième partie

Problématique

Chapitre 4

Langages dédiés

La partie précédente décrit le contexte de nos travaux qui est l'administration autonome. Nous avons présenté le projet TUNe sur lequel nous travaillons, qui s'articule autour du développement d'un logiciel d'administration autonome. Les expérimentations menées ont montré que les langages d'administration utilisés dans TUNe devaient être adaptés en fonction du domaine applicatif. Adapter les langages en fonction d'un domaine revient à utiliser des langages dédiés. Notre idée consiste alors à utiliser des techniques éprouvées de l'Ingénierie Dirigée par les modèles pour implanter des langages dédiés dans TUNe.
Ce chapitre présente une introduction aux langages dédiés, ainsi qu'un aperçu des principaux avantages qu'ils offrent par rapport à une approche traditionnelle de développement logiciel. Nous décrivons ensuite les différentes parties d'un langage dédié. Enfin, nous présentons les principales limites de cette technologie, qui représentent parfois un frein à son utilisation dans un cadre industriel.

4.1 Besoin de langages dédiés

L'un des enjeux principaux de l'ingénierie du logiciel est d'augmenter la productivité et la fiabilité du développement logiciel. L'avancée la plus significative reste sans doute le passage des langages assembleur aux langages de programmation de troisième génération (e.g. BASIC). Depuis, malgré toutes les avancées dans les langages de programmation (objets, composants ...), aucun langage, aucune architecture ou nouvelle technologie ne semble avoir un impact aussi important sur la productivité [Jones 2006; dsm 2012]. L'élévation du niveau d'abstraction de l'assembleur aux langages de programmation de troisième génération est la principale cause du gain énorme de productivité observé alors. L'objectif de l'ingénierie du logiciel aujourd'hui est donc d'élever encore plus le niveau d'abstraction des langages de programmation pour augmenter la productivité de façon significative. Les langages dédiés (*Domain Specific Language* ou DSL [1]) sont un moyen

1. Dans ce document nous utiliserons de manière indifférente le terme *langage dédié* et DSL

reconnu pour atteindre cet objectif. En effet, en se concentrant sur un domaine ou un problème particulier, ils permettent d'élever le niveau d'abstraction. De par l'utilisation de concepts usuels du domaine, il devient plus facile de modéliser des solutions aux problèmes.

4.2 Définition

Comme le souligne [Kahlaoui 2011], il n'existe aujourd'hui aucune définition précise et établie de ce qu'est un langage dédié. La littérature fait état de définitions variées ([Fowler 2010], [Mernik et al. 2005], [Consel 2003], [van Deursen et al. 2000] ou encore [Deursen 1998]), chacune mettant l'accent sur un ou plusieurs aspects de ces langages. Ces différences de perception affectent même leurs appellations. Ainsi, les DSL sont également appelés " *Little Languages* ", " *Micro Languages*", "*Domain Modeling Languages* - DML", "*Domain Specific Modeling Languages* DSML", " *Domain Specific Visual Languages* - DSVL ", "*Domain Specific Visual Modeling Languages* - DSVML", "*Domain Specific Embedded Languages* - DSEL", ... Les différentes définitions varient en fonction du domaine applicatif et de la communauté d'utilisateurs visée, mais aussi en fonction de la définition de ce qu'est un domaine applicatif et du degré de spécificité que doit avoir le DSL.

Dans ce document nous entendons par langage dédié un langage conçu sur mesure pour répondre aux besoins spécifiques d'un domaine particulier. Étant restreint à un domaine ou à un problème particulier, ce langage offre alors les bonnes abstractions de ce domaine à l'utilisateur. Les concepts manipulés par le langage correspondent directement aux connaissances et aux concepts des experts du domaine. L'utilisateur peut donc penser les solutions directement dans les termes de son domaine et ne se pose plus la question de "*comment*" faire (les détails d'implantation), mais plutôt "*quoi*" faire (la logique du domaine). Le DSL permet ainsi à la fois d'élever progressivement le niveau d'abstraction, mais aussi de prendre en compte toutes les spécificités du domaine d'application. L'objectif final étant de fournir aux utilisateurs des outils dédiés à leurs métiers ou à la tâche à effectuer. L'utilisation des langages dédiés permet de faciliter de manière significative la construction de systèmes logiciels pour les domaines auxquels ils sont dédiés [Bentley 1986] [J. Karna 2009] et d'augmenter la productivité logicielle [Thibault 1998].

Les principales caractéristiques d'un DSL sont leur concision, ainsi que des notations et des abstractions appropriées à un domaine. Ils s'opposent donc en ce sens aux langages dits généralistes (*General Purpose Languages* ou GPL) tels que Java, C++ ou UML. Contrairement aux langages généralistes qui permettent d'adresser une large classe de problèmes, les langages dédiés sont plus petits et se focalisent sur un domaine précis et donc sur un ensemble fini de problèmes. Ce contexte d'étude particulier permet de spécialiser le processus de développement, de supprimer les détails d'implémentation propices aux erreurs et de diminuer la complexité de la solution. En amenant ainsi la programmation plus près du domaine de problèmes, les langages dédiés facilitent la validation des

solutions et permettent aux erreurs d'être détectées plus tôt dans le processus de conception.

Les exemples de langages dédiés sont nombreux. Nous pouvons citer le langage des expressions régulières pour la manipulation des chaînes de caractères, le langage SQL pour les requêtes de base de données, LaTex pour la mise en forme de documents ou encore ant pour la compilation d'applications Java. Toutefois, l'usage des langages dédiés ne se limite pas au domaine informatique, il existe également des langages dédiés dans le domaine de la chimie moléculaire [Murray-Rust and Rzepa 1999], des produits financiers [Arnold et al. 1995] ou encore de la robotique [Pfeiffer Jr 1997].

Dans le contexte informatique, on distingue deux types de langages dédiés : les langages dédiés de programmation et les langages de modélisation. On parle aussi de façon assez réductive de DSL textuels et de DSL graphiques. En réalité dans la littérature, il n'y a pas de définition précise et communément admise permettant de faire une distinction claire entre un langage de modélisation et un langage de programmation. Cette distinction n'est faite que de façon informelle en considérant les utilisations historiques de chacun des deux langages. Il se trouve que la majorité des langages de programmation sont textuels alors que la plupart des langages de modélisation sont graphiques. Mais faire d'emblée cette supposition serait erroné. Il existe des langages de programmation graphiques tels que celui proposé par [Czarnecki and Eisenecker 2000] et à l'inverse tout langage textuel n'est pas forcément un langage de programmation (exemple XML).

En fait, la frontière entre la modélisation et la programmation de solutions est juste une question de niveau d'abstraction. Les langages dédiés de programmation sont de bas niveau et nécessitent des compétences en programmation. Les langages dédiés de modélisation (ou encore *Domain Specific Modelling Language*, DSML [2]) possèdent de fortes abstractions et sont généralement centrés sur l'utilisateur et la modélisation des solutions. Les DSML nécessitent des connaissances dans le domaine, mais pas d'expertise en programmation. Le tableau 4.1 présente des exemples de langages dédiés de programmation et de modélisation dans différents domaines [Latry 2007].

Même s'il y a toujours des cas pour lesquels les DSLs dits *textuels* restent les mieux adaptés, la simplicité d'utilisation et l'expressivité des DSML demeurent des atouts majeurs permettant de fournir aux experts du domaine des solutions de haut niveau. De façon pratique nous admettons dans la suite de ce document qu'un langage dédié de modélisation ou DSML réfère à un DSL graphique.

Les DSML sont un des thèmes centraux de l'Ingénierie Dirigée par les Modèles (IDM ou *Model Driven Engineering*-MDE). L'IDM est une forme d'ingénierie dans laquelle tout ou partie d'une application est générée à partir de modèles. Les enjeux de l'IDM sont l'automatisation et la fiabilisation de la production logicielle dans le but de réduire les coûts de développement et d'améliorer la productivité. Nous présentons dans la section suivante les concepts centraux de cette approche et les relations qui existent entre ces

2. Dans ce document nous utiliserons de façon indifférente *langage dédié de modélisation* et DSML

Domaine	**Langage Dédié de Programmation**	**Langage Dédié de Modélisation, DSML**
Création d'interfaces graphiques	SWUL (*SWing User interface Language*)	CookSwing, SwiXML, SwingML, JellySwing
Description de matériel	VHDL (*Very high speed integrated circuit Hardware Description Language*)	TDML (*Timing Diagrams Editor and Analysis*), LogSim, Matlab
Filtrage de courrier électronique	Sieve	Mulberry, Ingo, AvelSieve, SmartSieve, WebSieve
Modélisation de la réalité virtuelle	VRML (*Virtual Reality Markup Language*)	STEP (*Scripting Language for Embodied Agents*), XSTEP
Modélisation de processus métier	BPEL (*Business Process Execution Language*)	ActiveBPEL, WS-CDL (*Web Services Choreography Description Language*), BPML (*Business Process Modeling Language*)

TABLE 4.1 – Exemples de langages dédiés de programmation et de modélisation

concepts. Cette présentation vise avant tout à préciser la terminologie employée dans la suite du document et est adaptée de la description de [Muller 2006].

4.3 Ingénierie Dirigée par les Modèles

L'Ingénierie Dirigée par les Modèles est une approche émergente du développement logiciel. Le principe fondateur de cette démarche est l'utilisation des modèles comme base de toutes les réflexions dans les processus de conception et de développement logiciel. Cette section définit les principales notions de cette approche que nous utilisons dans la suite de ce document.

Modèles

La première notion importante est la notion de modèle. Quelle que soit la discipline scientifique considérée, un modèle est une abstraction d'un système construite dans un but précis. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé. On dit alors que le modèle *représente* le système. Un modèle est une abstraction dans la mesure où il contient un ensemble restreint d'informations sur un système. Il est construit dans un but précis dans la mesure où les informations qu'il contient sont choisies pour être pertinentes vis-à- vis de l'utilisation qui sera faite du modèle.
À titre d'exemple, une carte routière est un modèle d'une zone géographique, conçu pour la circulation automobile dans cette zone. Pour être utilisable par un automobiliste, une

carte routière ne doit inclure qu'une information très synthétique et spécifique sur la zone cartographiée : une carte à l'échelle 1, bien que très précise serait inutilisable.

Méta-modèle : langage de modélisation
Afin de rendre un modèle utilisable, il est nécessaire de préciser et de spécifier le langage dans lequel le modèle est exprimé. Ce langage de modélisation est appelé le méta-modèle. Le méta-modèle définit ce qu'il est possible de modéliser et la relation entre les concepts manipulés dans le modèle. Il détermine la terminologie à utiliser pour définir des modèles.
Dans le cas d'une carte routière, le méta-modèle utilisé est donné par la légende qui précise, entre autres, l'échelle de la carte et la signification des différents symboles utilisés. De manière pratique, un modèle n'a aucun sens sans son méta-modèle.

Méta-méta-modèle : langage de méta-modélisation
De la même manière qu'il est nécessaire d'avoir un méta-modèle pour interpréter un modèle, pour pouvoir interpréter un méta-modèle il faut disposer d'une description du langage dans lequel il est écrit : un méta-modèle pour les méta-modèles. C'est naturellement que l'on désigne ce méta-modèle particulier par le terme de *méta-méta-modèle*. En pratique, un méta-méta-modèle détermine le paradigme utilisé dans les méta-modèles et les méta-modèles sont construits à partir de lui. De ce fait, le choix d'un méta-méta-modèle est un choix important. À titre d'exemple, on peut citer EBNF [Garshol 2008] qui permet de définir des grammaires qui jouent le rôle de méta-modèles et dont les mots jouent le rôle de modèles.
Afin de se soustraire au problème de la définition des méta-méta-modèles (et ainsi éviter d'avoir à définir une infinité d'abstractions), l'idée généralement retenue est de concevoir les méta-méta-modèles de sorte qu'ils soient auto-descriptifs, c'est-à-dire capables de se définir eux-mêmes. Dans le cas des grammaires, on spécifie ainsi la syntaxe de EBNF par une grammaire.
La Figure 4.1, adaptée de [Fleurey 2006], résume les différents niveaux de modélisation (M0 : "le monde réel", M1 : modèle, M2 : méta-modèle et M3 : méta-méta-modèle), ainsi que les relations qui existent entre les différents concepts. Par exemple, elle montre qu'un modèle est exprimé dans un langage de modélisation et est conforme au méta-modèle qui représente ce langage.

4.3.1 Approches existantes

L'ingénierie Dirigée par les Modèles est une démarche générique qui regroupe plusieurs mises en œuvre partageant une vision similaire de ce que doit être le développement logiciel. L'IDM peut être vue comme une famille d'approches dont le principe fondateur est l'utilisation des modèles comme élément central dans le processus de développement logiciel.
Il existe plusieurs implémentations de l'IDM qu'on peut regrouper en deux grandes fa-

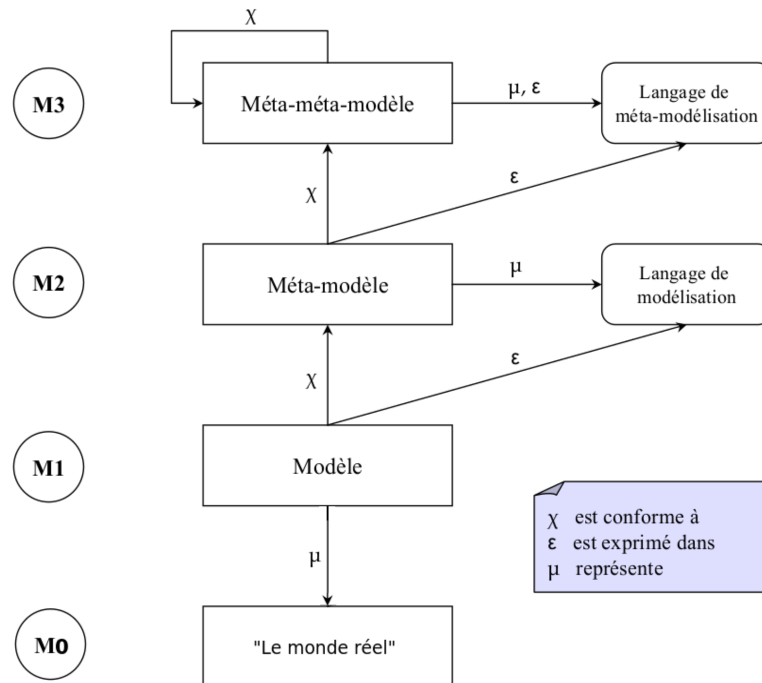


FIGURE 4.1 – Niveaux de modélisation

milles : l'approche d'Architecture Dirigée par les Modèles (*Model Driven Architecture*, MDA) [Miller and Mukerji 2003] et les approches de modélisation spécifique à un domaine (Domain Specific Modeling, DSM) comme le MIC (*Model Integrated Computing*) [Sztipanovits and Karsai 1997][Biegl 1995] ou encore les usines à logiciel (*Software factories*) [Greenfield and Short 2004]. Cette section propose un aperçu de ces approches.

Architecture Dirigée par les Modèles

L'architecture Dirigée par les Modèles est une démarche de développement proposée par l'organisme de standardisation OMG dans les années 2000. L'idée de base du MDA est de séparer les spécifications fonctionnelles d'un système des spécifications de son implantation sur une plate-forme donnée. Pour cela, le MDA définit une architecture de spécifications structurée en modèles indépendants des plates-formes (*Platform Indepandant Models* ou PIM) et en modèles spécifiques (*Platform Specific Models* ou PSM). Il s'agit donc de construire des modèles d'exécution (PIM) de systèmes de façon à prendre en compte tous les besoins fonctionnels, mais à demeurer complètement indépendants de toutes plates-formes, langages de programmation ou autres aspects liés à l'implantation. Afin de cibler des environnements d'exécution spécifiques, il faut concevoir plusieurs modèles spécifiques aux plates-formes (PSM) qui décrivent la manière dont les modèles de base sont implantés sur chaque plate-forme. L'approche MDA permet ainsi de réaliser le même modèle sur plusieurs plates-formes grâce à des projections normalisées. La mise en œuvre du MDA est donc entièrement basée sur les modèles et leurs transformations.
L'initiative MDA de l'OMG a donné lieu à une importante normalisation des technolo-

gies pour la modélisation. La proposition initiale était d'utiliser le langage UML (*Unified Modeling Language*) et ses différentes vues comme unique langage de modélisation. Cependant, il a fallu rapidement ajouter la possibilité d'étendre le langage UML, par exemple en créant des profils, afin de permettre d'exprimer de nouvelles notions. Ces extensions devenant de plus en plus importantes, la communauté MDA a progressivement délaissé UML au profit d'approches utilisant des langages spécifiques. Le langage de méta-modélisation MOF (*Meta-Objet Facilities*) [omg 2006] a été créé pour permettre la spécification de méta-modèles, c'est-à-dire de langages de modélisation. EMOF (*Essential MOF*) et CMOF (*Complete MOF*) sont deux évolutions de MOF. Comme leurs noms l'indiquent, la différence entre ces deux langages réside dans le nombre de concepts qu'ils contiennent. CMOF contient un ensemble de concepts plus important que EMOF qui lui ne contient que les concepts essentiels à la méta-modélisation.
Il faut noter que L'OMG fournit des spécifications et pas leur implantation. Ces spécifications sont ensuite implantées par des entreprises privées ou de groupes open sources. Cependant, la conformité de ces implantations n'est pas toujours stricte et peut dériver légèrement des spécifications de l'OMG.

Modélisation Spécifique au Domaine (DSM)

Les approches de modélisation spécifiques à un domaine mettent en avant une méthodologie de développement centrée sur la notion de domaine. Le concept central de la modélisation spécifique au domaine réside dans la notion de DSML. L'approche consiste à rassembler la connaissance du domaine d'application dans un langage de modélisation dédié. Les modèles construits à partir de ce langage ne doivent plus être uniquement contemplatifs, mais productifs (exécutables) pour permettre de générer le code d'une application en partie ou de manière complète. Du fait de sa portée restreinte, il est beaucoup plus facile de définir un langage de modélisation spécifique qu'un langage généraliste comme UML. En effet, l'intérêt des DSML ne se situe plus au niveau d'une modélisation généraliste, mais au niveau de la résolution de problèmes simples et restreints. Chaque problème est différent, d'une organisation à une autre, d'un produit à un autre, ou d'un système à un autre. En traduisant les concepts et les règles d'un domaine directement dans un langage de modélisation spécifique à ce domaine, il devient possible de fournir des outils adaptés aux besoins de l'organisation et des utilisateurs. L'approche tout entière devient alors spécifique, ce qui permet une réduction des cycles de développement. Un autre point important est que pour adapter exactement le langage de modélisation aux besoins des experts du domaine, il est indispensable que l'approche soit mise en œuvre par les experts du domaine eux-mêmes. Ce faisant, le temps nécessaire aux développeurs pour apprendre le langage de modélisation est réduit, car ils choisissent eux-mêmes les termes et les concepts du langage. De plus, il devient plus facile de faire évoluer le langage de modélisation en réponse aux changements dans le domaine. La démarche DSM se veut modeste, en spécialisant le processus à un domaine. Ceci rend la modélisation plus réalisable et concrète, entraînant de meilleurs résultats en termes de productivité. Afin de mener à bien cette démarche, l'approche implique donc de disposer d'outils ou d'environnements de modélisation simples et puissants, qui permettent de concevoir ces DSMLs. Il

existe plusieurs approches de modélisation spécifiques les deux principales étant le MIC et les usines à logiciel.

***Model Integrated Computing* :**
Les travaux autour du *Model-Integated Computing* (MIC) proposent, dès le milieu des années 90, une vision du génie logiciel dans laquelle les artefacts de base sont des modèles spécifiques au domaine d'application considéré. Initialement, le MIC est conçu pour le développement de systèmes embarqués complexes [Karsai et al. 2003]. La méthodologie proposée décompose le développement logiciel en deux grandes phases. La première phase est réalisée par des ingénieurs logiciels et systèmes. Elle consiste en une analyse du domaine d'application ayant pour but de produire un environnement de modélisation spécifique à ce domaine. Au cours de cette phase, il faut choisir les paradigmes de modélisation et définir formellement les langages de modélisation qui seront utilisés. À partir de ces éléments, un ensemble de méta-outils sont utilisés pour générer un environnement spécifique au domaine. Cet environnement généré a l'avantage d'être utilisable directement par des ingénieurs du domaine. La seconde phase consiste à modéliser l'application à réaliser en utilisant l'environnement généré. Cette phase est réalisée uniquement par des ingénieurs du domaine et permet de synthétiser automatiquement le logiciel.

***Software Factories* :**
Les usines à logiciels (*software factories*) sont une approche proposée par Microsoft pour le développement du logiciel. Le terme d'usine logicielle fait référence à une industrialisation du développement logiciel et s'explique par l'analogie entre le processus de développement proposé et une chaîne de montage industrielle classique. En effet, dans l'industrie classique :

- Une usine fabrique une seule famille de produits. Si l'on considère par exemple l'industrie automobile, une chaîne de montage ne fabrique généralement qu'un seul type de voiture avec différentes combinaisons d'options.
- Les ouvriers sont relativement spécialisés. Il n'est pas rare de trouver des ouvriers ayant des compétences sur plusieurs postes de la chaîne de montage, mais il est très rare qu'un ouvrier ait toutes les compétences depuis l'assemblage jusqu'à la peinture des véhicules.
- Les outils utilisés sont très spécialisés et fortement automatisés. Les outils utilisés sur une chaîne de montage sont conçus uniquement pour cette chaîne de montage ce qui permet d'atteindre des degrés d'automatisation importants.
- Toutes les pièces assemblées ne sont pas fabriquées sur place. Une chaîne de montage automobile ne fait généralement qu'un assemblage de pièces préfabriquées à un autre endroit.

L'idée des usines logicielles est d'adapter au développement logiciel ces caractéristiques qui ont fait leur preuve pour la production de masse de familles de produits matériels. Les deux premiers points correspondent à la spécialisation des fournisseurs de logiciels et des développeurs à l'intérieur des équipes de développement. Le troisième point correspond à l'utilisation d'outils de génie logiciel spécifiques au domaine d'application, c'est-à-dire

de langages, d'assistants et transformations spécifiques. Le dernier point correspond à la réutilisation de composants logiciels sur étagères.
Les efforts sont orientés vers le développement de familles de produits partageant des caractéristiques communes plutôt que vers le développement de produits séparés. Concrètement cela consiste à configurer les plates-formes de développement logiciel extensibles, avec des actifs fondamentaux comme les DSLs, les patrons et les frameworks, afin d'optimiser le développement et la maintenance de certains types d'applications. Le mécanisme d'extension est réalisé en se basant sur deux concepts : les schémas (*factory schema*) qui définissent et organisent les artefacts utilisés pour construire et maintenir le système, et les modèles (*templates*) qui forment une capacité de production pour la famille de produits en configurant les outils, les processus et les autres actifs fondamentaux.
Les usines à logiciels reposent fortement sur les langages dédiés pour soutenir l'automatisation [Greenfield and Short 2004]. Ces langages sont utilisés pour exprimer différents aspects des membres de la famille à développer avec l'usine. Généralement plusieurs DSLs sont nécessaires pour décrire une application. Chaque DSL (ou ensemble de DSLs) correspond à un point de vue donné de l'application.

La Figure 4.2 présente une vue d'ensemble de ces différentes approches de l'Ingénierie Dirigée par les modèles.

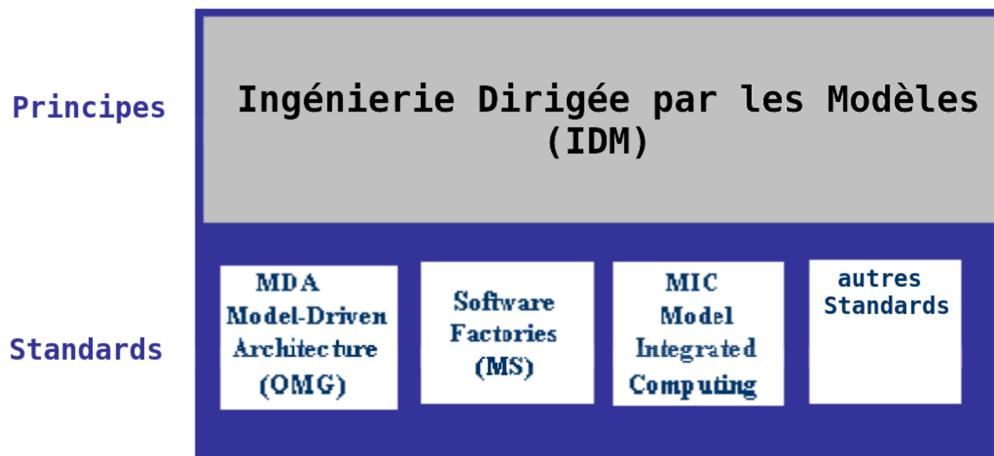


FIGURE 4.2 – Ingénierie Dirigée par les Modèles

4.4 Conception d'un DSML

De façon générale, qu'il soit textuel ou graphique, un langage est caractérisé par sa syntaxe et sa sémantique. La syntaxe décrit les différentes constructions du langage et les règles d'agencement de ces constructions tandis que la sémantique permet de donner un sens à chacune des constructions du langage. La conception d'un DSML consiste alors essentiellement à définir sa syntaxe et sa sémantique. Les sections suivantes présentent ces différentes notions.

4.4.1 Syntaxe

Dans le contexte informatique, on distingue deux types de syntaxe : syntaxe abstraite et syntaxe concrète.

Syntaxe Abstraite

La syntaxe abstraite définit de façon structurelle les concepts, les relations et les règles de grammaire caractérisant le DSML. Les concepts représentent les types d'éléments manipulés par le DSML alors que les relations et les règles de grammaire définissent comment ces éléments peuvent être combinés pour former des modèles valides [Greenfield and Short 2004]. La pertinence des concepts définis dépend fortement du niveau d'abstraction souhaité. Abstraire un système consiste à en extraire les caractéristiques essentielles par rapport à un type d'utilisateur et à un domaine particulier. Ce faisant, l'abstraction permet non seulement de cacher les détails non essentiels à l'utilisateur, mais également de s'appuyer sur des connaissances métiers pour résoudre des problèmes récurrents de ce domaine.
Selon [Tolvanen 2006], la partie la plus importante dans le processus de conception d'un DSML est de trouver les bonnes abstractions. Afin de ne garder que les concepts qui sont pertinents au DSML, il faut soigneusement déterminer le niveau d'abstraction. Travailler au juste niveau d'abstraction est crucial pour le succès du DSML car il affecte directement son expressivité. Un niveau trop abstrait fait qu'un DSML ne pourra pas fournir suffisamment d'informations dans ses modèles. Un niveau d'abstraction trop détaillé, par contre, peut mener à un DSML compliqué, confus et difficilement compréhensible par les utilisateurs [Kahlaoui 2011]. D'une manière générale, le niveau d'abstraction d'un système est déterminé par la finalité de son utilisation [Kramer 2007]. Plus cette finalité est claire et explicite, plus il est facile de l'abstraire.
Deux approches sont principalement utilisées pour la définition de la syntaxe abstraite [Greenfield and Short 2004] : la grammaire non-contextuelle (ou grammaire formelle e.g. *Backus-Naur Form/Backus Normal Form*-BNFs), utilisée surtout pour la définition des langages textuels et la méta-modélisation utilisée pour la définition des DSMLs. Cette distinction est tout simplement d'ordre pratique, on constate que l'approche de la grammaire non-contextuelle est mieux adaptée à la description des langages textuels puisqu'elle est bien optimisée pour le traitement de texte, alors que l'approche de méta-modélisation se prête mieux à la description des langages graphiques de modélisation. Dans le contexte de l'IDM, la syntaxe abstraite est placée au cœur de la description d'un DSML. Elle est donc généralement décrite en premier et sert de base pour spécifier la syntaxe concrète que nous définissons dans la section suivante.
Dans l'exemple des cartes routières, la syntaxe abstraite qui correspond au méta-modèle inclura des concepts tels que *Route*, *Autoroute*, *Échangeur*, *Distance* ... ainsi que les relations entre ces concepts par exemple une relation d'héritage entre le concept *Route* et le concept *Autoroute* pour signifier qu'une autoroute est une route particulière.

Syntaxe concrète

Un DSML n'a qu'une seule syntaxe abstraite, mais peut avoir plusieurs syntaxes concrètes. La syntaxe concrète fournit à l'utilisateur un ou plusieurs formalismes graphiques et/ou textuels, pour manipuler les concepts de la syntaxe abstraite et ainsi en créer des *instances*. L'instance ainsi obtenue sera conforme à la structure définie dans la syntaxe abstraite. La syntaxe concrète textuelle est utilisée, principalement, pour les DSL dits « de programmation » alors que la syntaxe concrète graphique est plutôt réservée aux DSML.
La syntaxe concrète graphique décrit les éléments graphiques à utiliser pour créer des modèles du DSL et détermine leurs dispositions géométriques. Elle consiste en une interface utilisateur graphique permettant de manipuler (spécifier, ajouter, retirer, lier ...) les concepts du DSL. Chaque élément représentable du langage est associé à un symbole graphique. L'avantage principal d'une syntaxe visuelle est sa capacité à exprimer l'information sous une forme intuitive et facile à comprendre [Clark et al. 2004].
La Figure 4.3 montre un exemple de carte routière avec la représentation généralement utilisée.

FIGURE 4.3 – Syntaxe concrète pour des cartes routières

Dans la suite de ce document, nous entendons par DSML un langage de modélisation ayant une syntaxe concrète graphique. Les travaux que nous présentons s'articulent autour de ces DSML, notamment sur l'implantation de la syntaxe concrète graphique.

4.4.2 Sémantique

Généralement, la syntaxe abstraite ne comporte pas assez d'information pour définir le sens des constructions du DSML. Des informations supplémentaires sont nécessaires afin de pouvoir déterminer ce sens. Le but de la sémantique est donc de décrire le sens des constructions du langage. On distingue deux types de sémantique : la sémantique statique et la sémantique dynamique (ou comportementale).

La sémantique statique se concentre sur la signification d'un modèle avant son exécution (par exemple pour la vérification des types), elle peut être exprimée sur la syntaxe abstraite, soit à l'aide du langage de méta-modélisation lui-même (par exemple les multiplicités - une *Autoroute* peut avoir plusieurs *Échangeurs*) soit en la complétant de contraintes

exprimées à l'aide d'un langage comme OCL (invariant, pré- ou post-condition - le *Numéro* d'un *Échangeur* doit toujours être positif).

La sémantique dynamique quant à elle s'intéresse au comportement des modèles à l'exécution. Elle permet d'exécuter un modèle en vue de l'animer, le simuler ou le vérifier dynamiquement. La définition de cette sémantique comportementale se rapporte à la définition d'un *mappage* sémantique (*Semantic Mapping*, SM) vers un domaine sémantique (*Semantic Domain*, SD) [Harel and Rumpe 2000]. Le domaine sémantique définit l'ensemble des significations possibles qui peuvent être accordées aux différents modèles du DSML. Le *mappage* sémantique établit le lien entre les éléments de la syntaxe et le sens qu'on leur donne.
La sémantique comportementale des DSMLs est généralement décrite de manière ad-hoc et nécessite beaucoup d'effort [Combemale 2008]. L'un des défis actuels de l'IDM consiste à donner les moyens d'exprimer la sémantique comportementale de façon précise. Cette problématique fait l'objet de nombreux travaux tels que [alain Muller et al. 2005] et [Clark et al. 2008], mais cet aspect des DSMLs ne fait pas partie de nos travaux.
La Figure 4.4 résume l'essentiel de la conception d'un DSML. Rappelons qu'un outil de méta-modélisation permet d'implanter un DSML, et que pour construire des modèles conformes au DSML il faut créer un éditeur graphique (un outil de modélisation) à partir de la syntaxe abstraite du DSML. Cet éditeur matérialise la syntaxe concrète.

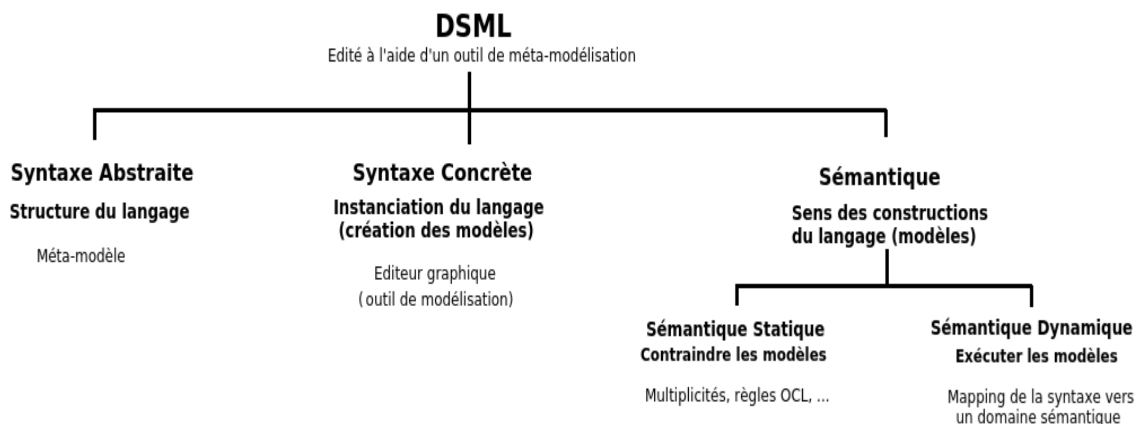


FIGURE 4.4 – Conception d'un DSML

4.5 Synthèse

Dans le cadre de TUNe l'utilisation des DSML va permettre d'adapter les langages d'administration en fonction du contexte applicatif et ainsi de faciliter l'administration autonome des applications. Cependant, sans des outils appropriés, les DSML peuvent être difficiles à mettre en œuvre et coûteux à concevoir et à implanter. Le chapitre suivant présente les limites des DSML notamment en termes d'outils.

Chapitre 5

Limites des DSML et problématique

Les langages de modélisation dédiés sont réputés pour leur expressivité et leur fort niveau d'abstraction. Ils sont utilisés dans différents domaines pour permettre aux experts du domaine de concevoir des solutions en utilisant des concepts et des notations qui leur sont familiers. Or, il reste encore des contraintes qui renforcent l'hésitation des organisations envers cette technologie et empêchent son adoption à plus grande échelle. Nous présentons les principales limites dans les sections suivantes.
Le développement d'un DSML nécessite à la fois la connaissance du domaine, mais aussi une expertise en développement de langages. Peu de personnes possèdent cette double compétence [dsm 2012]. De ce fait, la décision de développer un DSML est souvent écartée par les organismes industriels. Comme nous allons le montrer dans les sections suivantes, il existe aujourd'hui plusieurs freins à l'acceptation des langages de modélisation dédiés, notamment liés au coût de développement du DSML ou encore à la délimitation du domaine. Mais la limitation la plus importante est certainement celle liée aux outils de méta-modélisation souvent jugés peu intuitifs, trop complexes ou pas assez matures. Nous nous sommes heurtés à cette complexité des outils de méta-modélisation dans notre tentative d'implanter des langages dédiés dans TUNe. De même, les retours d'expériences de nos partenaires industriels soulignent que le temps d'apprentissage pour la prise en main des outils de méta-modélisation existants peut être très élevé. Chez Airbus, ils sont convaincus du gain de productivité que pourraient leur apporter les langages dédiés et s'y essayent, mais cela relève encore du domaine de la recherche.

5.1 Coût de développement

La première barrière à l'adoption des DSMLs est celle du coût de développement. Développer entièrement un DSML ainsi que les outils associés pour l'édition des modèles est une tâche difficile et coûteuse dont la rentabilité est difficilement prouvée à l'avance. Le coût de développement d'un langage dédié inclut non seulement la conception, l'implémentation et la maintenance, mais aussi le coût d'apprentissage des utilisateurs du DSML.

Même si les bénéfices des langages dédiés ont souvent été observés dans la pratique et ont été quantifiés par différentes études [Herndon and Berzins 1988][Batory et al. 1994][Gray and Karsai 2003], il demeure néanmoins très difficile de le démontrer à priori. À ce constat vient s'ajouter l'échec de l'adoption dans le monde industriel des outils issus du monde de la recherche tels que GME [Davis 2003], DiaMeta [Minas 2012] ... souvent jugés peu matures.

5.2 Limitations liées à la délimitation du domaine

La difficulté à définir un langage dédié est due en grande partie à l'imprécision de la définition de domaine. Le terme « domaine » est tellement surchargé que cela devient impossible de couvrir toutes les définitions données. Chaque communauté a développé sa propre perception du concept de domaine et a formulé sa propre définition. Plus que cela, il est très fréquent de trouver plusieurs définitions au sein de la même communauté [Kahlaoui 2011]. Par exemple, dans la communauté de la réutilisation logicielle un domaine est un "ensemble de systèmes", mais dans la communauté de l'Intelligence Artificielle un domaine est plutôt défini en tant que "*Real World*". Lorsque le domaine est défini en tant que "Real World", les experts du domaine sont généralement les praticiens alors que s'il est défini en tant "qu'ensemble de systèmes", ce sont les ingénieurs qui ont développé les systèmes du domaine qui sont considérés comme les experts de domaine [Simos et al. 1996]. Il existe plusieurs autres catégorisations des domaines : vertical versus horizontal, natif versus novateur, et diffusé versus encapsulé [Simos et al. 1996]. Toutes ces distinctions illustrent bien toutes les variations qu'on peut avoir dans la perception du concept de domaine.

Afin de bien délimiter la portée du DSML, il faut définir précisément son domaine. Après tout, un DSML est un langage spécialisé conçu pour répondre aux besoins d'un problème particulier dans un domaine donné. La définition claire et précise du problème à résoudre aide à mieux tracer les frontières de la portée du DSML. Une bonne délimitation du domaine permet alors de fournir un DSML avec un degré adéquat de spécificité. Cette caractéristique détermine le degré de précision avec lequel le langage dédié définit les besoins des utilisateurs dans ce domaine. Un DSML avec un degré élevé de spécificité offre des constructions et des fonctionnalités qui concordent avec les tâches quotidiennes des experts métier. Ces derniers doivent être à même de comprendre les concepts du langage et de manipuler par eux-mêmes ces modèles sans avoir besoin de connaissances particulières en programmation. En revanche, et étant donné que la majorité des domaines changent et évoluent continuellement, le maintien de la spécificité des DSML associés devient un véritable défi. Cela exige du DSML d'avoir une forte capacité d'adaptation et d'évolution afin de pouvoir s'accommoder facilement des changements et des transformations du domaine. La difficulté est que le DSL doit répondre aux changements de son domaine tout en conservant son expressivité et son fort niveau d'abstraction. Il doit également assurer la compatibilité ascendante ce qui signifie que la nouvelle version du DSML doit continuer son support pour les modèles développés avec les versions précédentes.
Selon le domaine, il peut également apparaître intéressant de fournir plusieurs DSL conçus pour différentes catégories d'utilisateurs, ou encore, il peut arriver que plusieurs DSL

soient nécessaires pour spécifier complètement une application ou un domaine complexe. Cette situation pose le problème central de la composition des langages dédiés au sein d'un domaine, qui est aujourd'hui une thématique de recherche importante autour des langages dédiés [White et al. 2009][Barroca et al. 2009][Clarke 2001].

5.3 Limitations liées aux outils de méta-modélisation

Un outil de méta-modélisation est un environnement permettant de concevoir des DSMLs et de générer des outils pour instancier ces langages. Concrètement, comme le présente la Figure 5.1, un outil de méta-modélisation permet d'éditer la syntaxe abstraite d'un DSML sous forme de méta-modèle, de spécifier les représentations graphiques des éléments du méta-modèle et de préciser le sens des modèles du DSML. À partir de ces spécifications, l'outil de méta-modélisation permet de générer un environnement de modélisation ou éditeur de modèles qui va permettre d'instancier le langage (de créer des modèles).

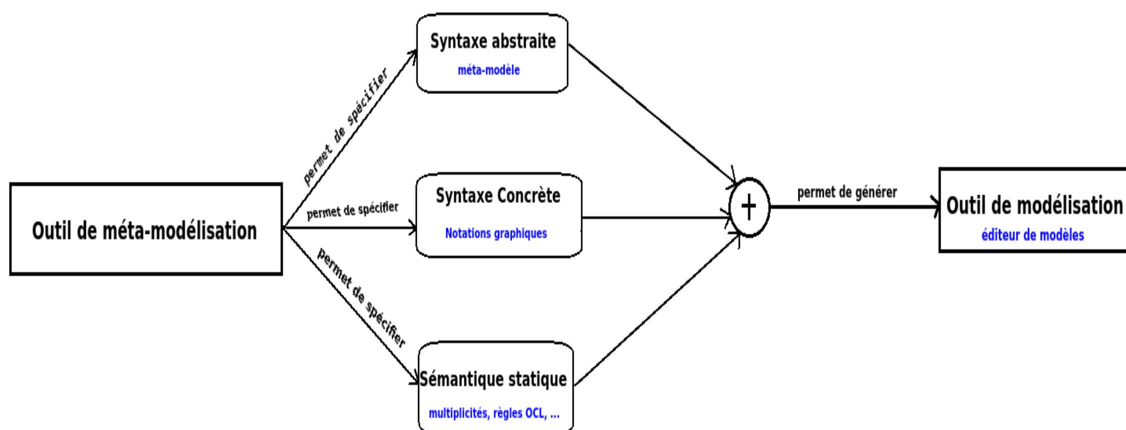


FIGURE 5.1 – Outil de méta-modélisation

Il existe aujourd'hui de nombreux outils de méta-modélisation qui permettent de créer et d'outiller les DSMLs, mais l'adoption à large échelle de ces outils peine à se faire à cause du décalage observé entre leur efficience et les attentes des utilisateurs. Ces outils peuvent êtres scindés en deux grandes catégories :

- **Outils génériques** : ce sont des frameworks de méta-modélisation qui permettent de concevoir et d'implanter des DSMLs quelque soit le domaine d'application. Le langage de méta-modélisation qu'offre ce type d'outil fournit des concepts suffisamment génériques pour permettre la conception de langages dédiés pour quasiment tous les domaines.
- **Outils spécifiques** : ce sont des frameworks de méta-modélisation qui se restreignent à un domaine ou à une famille de DSMLs et permettent la conception et l'implantation de DSMLs uniquement à l'intérieur de ce domaine. Le langage de méta-

modélisation de l'outil spécifique fournit des concepts propres au domaine ciblé. Ces concepts ont par exemple des types particuliers ou des propriétés prédéfinies.

Chacune de ces solutions présente des avantages et des inconvénients, mais selon qu'un outil est générique ou spécifique il peut être plus ou moins adapté pour résoudre un problème et plus ou moins complexe d'utilisation.

5.3.1 Générique Vs. Spécifique

Généralités

Que ce soit langages généralistes vs. langages spécifiques, UML vs. DSML, XML vs. XHTML, RDBMS Tables vs. Metatables, SOAP vs. REST, Maven vs. Ant (vs. Scripts) . . . , le débat est le même à savoir : quelle est la meilleure solution entre l'approche générique et l'approche spécifique ? Cette opposition générique vs. spécifique n'est pas nouvelle et existe dans toutes les branches de la science et de l'ingénierie. À cette question [Tilkov 2009] préconise de répondre "*it depends*" parce que ça dépend du problème à résoudre et que trouver la meilleure solution c'est arriver à faire un bon compromis entre les deux approches. Il détermine un certain nombre de critères permettant de faire ce choix : existence de support (disponibilité), coût de développement, coût d'apprentissage, performance lors du développement, performances à l'exécution, les compétences et les connaissances requises, l'étendue de la communauté d'utilisateurs . . .
Les outils génériques coûtent peu cher en termes de développement, parce qu'on se sert directement de la plate-forme existante. De plus, ces outils couvrent en général un large spectre d'application et permettent de résoudre plusieurs types de problème et sont donc généralement très flexibles. Néanmoins, même si ces outils présentent plusieurs avantages, il est déconseillé de toujours opter pour une solution générique et paramétrable ("over-engineering" ou "gold-plating") parce que le coût peut être plus important que les bénéfices escomptés [Wagner and Deissenboeck 2008]. Les outils génériques ne coûtent pas cher en termes de développement, mais sont généralement plus complexes d'utilisation que des outils spécifiques.

> *"Some programmers, when faced with a problem, turn to a generic solution ... now they have two problems."* ([Tilkov 2009] [1])

> *"Weak methods are general approaches that may be applied to many types of problems. . . . a weak method, like a monkey wrench, is designed to adjust to a multiplicity of problems, but solve none of them optimally."* [Vessey and Glass 1998]

1. Adapté de : *"Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems"*. (Jamie Zawinski)

En effet pour être suffisamment génériques ces outils ne doivent pas avoir un niveau d'abstraction élevé. Leur faible niveau d'abstraction leur permet certes d'être flexibles, mais entraîne également une plus grande complexité d'utilisation. L'outil générique adresse plusieurs domaines, mais aucun de façon optimale à cause de son manque de précision. Ce manque de précision empêche de construire des solutions avec le niveau d'expressivité attendu. Un outil difficile à utiliser fait perdre du temps et n'est pas exploité pleinement parce que sa complexité peut dissuader l'utilisateur d'aller vers des fonctionnalités avancées [Bevan and Macleod 1994]. De plus pour adresser un maximum de problèmes, les outils génériques intègrent de multiples fonctionnalités dont l'utilisateur en fonction de son problème n'a pas forcément besoin ce qui contribue à les rendre plus complexes et moins intuitifs. Un outil peu intuitif augmente le temps d'apprentissage ce qui influe sur la productivité.

Les outils de méta-modélisation spécifiques quant à eux sont connus pour être simples d'utilisation parce qu'ils se concentrent sur un ensemble restreint de problèmes.

> *"Strong methods are those designed to address a specific type of problem ...A strong method, like a specific size of wrench, is designed to fit and do an optimal job on one kind of problem ;"* [Vessey and Glass 1998]

Un outil spécifique couvre en général un spectre plus restreint un niveau d'abstraction élevé. L'abstraction vise à diminuer la complexité globale de l'outil en assurant l'essentiel. Cette restriction à un ensemble restreint de problèmes augmente leur expressivité et les rend plus intuitifs. Le temps d'apprentissage est alors réduit et les performances globales accrues. En revanche, un outil spécifique a une portée réduite, il est moins flexible et difficilement extensible. Il faut donc souvent en utiliser plusieurs dans le processus de développement, mais l'utilisation de plusieurs outils spécifiques soulève d'autres problèmes tels que la gestion de l'interopérabilité entre ces outils.

Le tableau 5.1 résume ces différentes caractéristiques.

Outils	**Avantages**	**Inconvénients**
Générique	grande portée, flexibilité, disponibilité, communauté d'utilisateurs étendue	complexité d'utilisation, manque d'expressivité, manque de précision, temps d'apprentissage élevé, nécessite de coder
Spécifique	petit, intuitif, performant, productif	coût, portée restreinte, variation du degré de spécificité, pas d'interopérabilité entre les outils

TABLE 5.1 – Générique versus Spécifique

Cas des outils de méta-modélisation

Si l'on se rapporte à la méta-modélisation, les outils génériques sont caractérisés par :

- **Faible niveau d'abstraction** : un outil de méta-modélisation générique pour adresser tous les domaines applicatifs doit fournir des abstractions communes à tous ces domaines. Le niveau d'abstraction est ainsi tiré vers le bas pour répondre à tous ces domaines. Le langage de méta-modélisation fourni propose des concepts très génériques, très souvent des classes et des liaisons. À ces concepts génériques n'est associé aucune sémantique particulière, on pourra donc ainsi représenter n'importe quel élément du langage spécifique par une classe et n'importe quelle relation par une liaison. Il revient alors à l'utilisateur de programmer toutes les spécificités de son langage qui ne sont pas capturées par ces concepts génériques. Concrètement l'outil de méta-modélisation permet généralement de générer une première version du code source du DSML, code que l'utilisateur doit ensuite compléter en utilisant un langage de programmation généraliste tel que Java ou C++. L'utilisateur se retrouve donc à devoir programmer sa solution spécifique alors qu'il n'a pas forcément les compétences d'un expert en programmation.
- **Librairie graphique générale** : pour spécifier la représentation graphique d'un élément du langage, un outil de méta-modélisation générique va fournir à l'utilisateur des figures basiques telles que des rectangles, des cercles, des traits ou des icônes. Ce type de représentation graphique se prête bien à la construction de diagrammes ayant une structure en graphe dans laquelle chaque élément graphique a une correspondance 1-1 avec un concept du méta-modèle. Mais lorsque la syntaxe concrète dévie de la syntaxe abstraite, les icônes et les figures basiques deviennent insuffisantes. L'utilisateur se retrouve à devoir programmer lui même les représentations et le comportement graphique qu'il souhaite attribuer aux modèles. Mais la programmation d'un éditeur graphique nécessite une grande expertise dans le langage de programmation utilisé ce qui rend cette tâche très complexe. Dans la pratique, les représentations graphiques fournies par défaut par l'outil de méta-modélisation générique sont rarement satisfaisantes pour construire des solutions spécifiques ou alors uniquement pour des cas très simples.
- **Construction de l'outillage complexe** : avec le faible niveau d'abstraction, c'est tout le processus de création d'un éditeur graphique qui devient complexe. Ce processus consiste généralement en trois étapes : spécification de la syntaxe abstraite, spécification de la syntaxe concrète et spécification de la correspondance entre les éléments de la syntaxe concrète et ceux de la syntaxe abstraite. Les outils de méta-modélisation génériques proposent généralement un assistant pour chacune de ces étapes. L'assistant fourni fonctionne bien pour des cas très généraux et masque en quelque sorte le faible niveau d'abstraction de l'outil. Mais quand il s'agit implanter des solutions vraiment spécifiques (ce qui est quand même l'objectif de ces outils), l'utilisateur doit lui même programmer sa solution.

Les outils de méta-modélisation spécifiques quant à eux sont caractérisés par :

- **Haut niveau d'abstraction** : Un outil de méta-modélisation spécifique à un domaine ne fournit que les abstractions du domaine ciblé. Le langage de méta-modélisation qu'il propose est de ce fait spécialisé et fournit des concepts ayant des propriétés ou des attributs prédéfinis tirés du domaine cible. Plutôt que de fournir des concepts généraux tels que *classe* et *liaison*, il fournira par exemple les concepts *Composant*, *Connecteur* et *Binding* si le domaine ciblé est celui des composants. Un expert du domaine comprendra alors rapidement à quoi correspond chaque concept du langage de méta-modélisation et saura d'emblée comment les assembler pour mettre en œuvre son langage spécifique.
- **notations et sémantique graphiques spécialisées** : l'outil spécifique fournit la notation graphique standard généralement utilisée dans le domaine. La sémantique du domaine est déjà quelque peu implantée dans chacun des concepts et la spécification de la syntaxe concrète se fait alors de façon directe.
- **Construction de l'outillage simple** : Avec la spécialisation de l'outil, c'est tout le processus de création d'un éditeur graphique qui se trouve simplifié. La spécification de la syntaxe abstraite et de la syntaxe concrète se fait en utilisant des concepts très abstraits et l'utilisateur a rarement besoin de programmer lui même sa solution.

Dans la pratique, les outils de méta-modélisation sont en majorité génériques et sont donc assez complexes d'utilisation et c'est cette complexité qui constitue le frein majeur à la vulgarisation des DSMLs. La solution généralement utilisée par les industriels est de développer leurs propres outils spécifiques qui sont alors des outils propriétaires développés entièrement par l'entreprise pour ses besoins. Cette solution qui consiste à développer soi-même son outil spécifique est très coûteuse et n'est pas accessible à tous.

5.4 Synthèse

Les langages dédiés sont un moyen reconnu pour permettre aux experts d'un domaine de concevoir des applications sans être experts en programmation. Restreints à un domaine, ils permettent de factoriser toute la connaissance relative à ce domaine dans le langage en utilisant un haut niveau d'abstraction. Cette factorisation permet d'améliorer la productivité du développement logiciel.
Dans ce chapitre nous avons discuté du concept de langage dédié de modélisation, de la notion de domaine et de spécificité. Nous avons mis en avant dans ce chapitre que, même si les langages dédiés présentent de nombreux avantages en termes de productivité, il reste de nombreuses limitations qui freinent leur adoption à plus grande échelle. La principale limitation est liée aux outils de méta-modélisation qui sont très souvent inadaptés aux besoins des utilisateurs. Ces outils sont pour la plupart trop génériques et donc complexes d'utilisation.
Les DSMLs ouvrent donc des perspectives intéressantes, mais sans outils appropriés, ils peuvent apparaître difficiles à mettre en œuvre, coûteux à concevoir et à implanter. La partie suivante donne un état de l'art des principaux frameworks de méta-modélisation existants avec pour chacun de ces outils une description du degré de spécificité. Nous

avons expérimenté tous ces outils, notre objectif étant alors de trouver la façon la plus adéquate d'implanter des DSML dans TUNe.

Chapitre 6

État de l'art

6.1 Introduction

Dans cette partie, nous présentons un état de l'art des différents outils de méta-modélisation existants. Ces outils sont des environnements de développement destinés, entre autres, à la création et à la modélisation de DSMLs. Ils permettent ainsi de définir explicitement un méta-modèle (la syntaxe abstraite du DSML), et de générer ou de programmer tous les outils spécifiques nécessaires, allant des éditeurs graphiques aux générateurs de code. De plus, afin de guider et restreindre les constructions du langage au sein du domaine, ils permettent la mise en place de règles (par exemple, des règles OCL). Ces règles contraignent l'utilisation du langage de modélisation en définissant les relations autorisées entre les différents concepts et la manière de les manipuler. Elles permettent ainsi de réduire l'espace de conception de solutions et assurent que les constructions du langage sont correctes vis-à-vis du domaine.

Avec l'avènement de l'IDM de plus en plus d'outils de méta-modélisation sont créés. Ces outils de méta-modélisation peuvent être commerciaux ou issus du monde de la recherche. Nous avons testé plusieurs de ces outils et avons choisi de ne présenter dans cet état de l'art qu'un sous-ensemble d'entre eux que nous avons jugé suffisamment représentatif de ce qui existe. Pour tester chacun de ces outils, il a d'abord fallu en faire l'acquisition, certains sont des outils commerciaux et donc payants et d'autres, issus du monde de la recherche, doivent être obtenus auprès des équipes qui les développent. Selon l'outil l'installation et la mise en fonctionnement se sont faites plus ou moins aisément avec plus ou moins de contraintes. Par exemple, certains d'entre eux ne fonctionnent que sur Windows. Enfin, chacun de ces outils a nécessité un certain temps d'apprentissage pour comprendre comment il fonctionne et comment l'utiliser, la difficulté étant qu'une fois sorti du tutoriel de base, il n'y a très souvent plus aucune documentation disponible. Ces outils diffèrent principalement par le langage de méta-modélisation qu'ils utilisent et qui est la base de tout l'environnement.

Nous avons utilisé ces outils dans notre contexte d'administration autonome, notre but étant d'implanter des DSML dans TUNe. Les contraintes de nos langages étaient issues du domaine des composants et se résument en trois principales notions :

– Composant : c'est une unité de structuration souvent représentée par une boite et qui possède des propriétés. La représentation graphique d'un composant est presque toujours une boite ou un rectangle.
– Connecteur : C'est une interface de connexions liée à un composant. Graphiquement, les points d'attache des connecteurs sur les composants peuvent avoir des positions variables comme le présente la Figure 6.1.
– Liaison : permet de relier des connecteurs entre eux. Graphiquement une liaison peut avoir une forme variable (en pointillé, avec des flèches ou des carrés aux extrémités ...). Une liaison se positionne sur un point précis du connecteur.
– Composition : un composant peut être composé d'autres composants, cette composition est généralement représentée par des boites à l'intérieur les unes des autres comme le présente la figure 6.1.

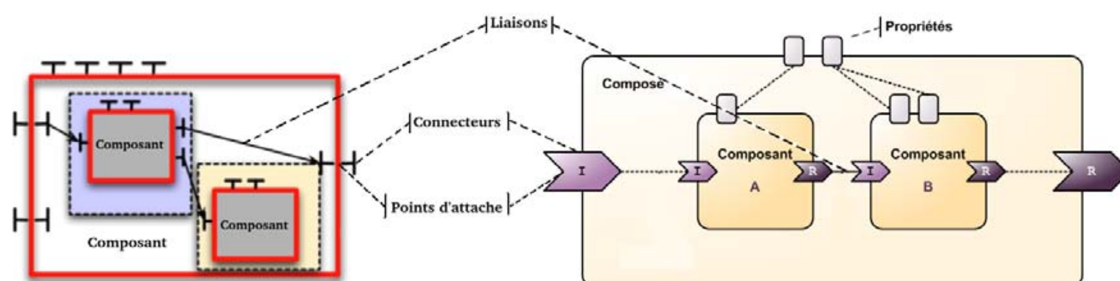


FIGURE 6.1 – Modèles à composants

6.2 Ce que fournit un environnement de méta-modélisation

Un environnement de méta-modélisation permet de définir un DSML et d'en générer un éditeur permettant d'exprimer des modèles conformes au langage. Ce processus se fait généralement en quatre étapes :

– **Spécification de la syntaxe abstraite** : de façon générale, l'outil doit permettre de spécifier des objets et des relations, de spécifier les propriétés de ces objets, propriétés pour lesquelles il doit fournir un support pour les types de base (Booléen, String ...). Enfin, l'outil doit permettre de spécifier des règles structurelles permettant de décrire comment les objets pourront être reliés les uns aux autres. Cette spécification est appelée méta-modèle. Il faut que le langage de méta-modélisation de l'outil soit suffisamment complet pour que la structure du langage puisse être exprimée complètement avec ce méta-modèle. Par exemple, il est utile d'avoir un concept plus global que les objets et les relations, ce concept représente un ensemble d'objets et de relations. Cela permet de structurer le méta-modèle qui est alors plus

simple à gérer. L'absence d'un tel concept serait équivalente à écrire un programme dans un seul fichier, ce qui fonctionne bien uniquement pour des petits programmes. De même, le langage de méta-modélisation doit offrir différents types de relations telles que la composition, l'héritage ..., qui permettent de structurer le langage. Il est également utile d'avoir des relations n-aires et pas que des relations binaires parce qu'il est possible qu'un langage nécessite une relation faisant intervenir plus de deux objets.

– **Spécification de la syntaxe concrète** : à cette étape, l'outil doit permettre de spécifier toutes les notations graphiques qui seront utilisées dans l'éditeur de modèle. À chaque concept de la syntaxe abstraite sera associé un symbole graphique qui permettra de représenter le concept associé au niveau de l'éditeur de modèles. De la même façon que le langage de méta-modélisation délimite ce qu'il est possible d'exprimer dans la syntaxe abstraite, l'ensemble des notations graphiques que permet l'outil délimite ce qu'il est possible d'exprimer dans la syntaxe concrète. Cette partie est assez critique parce que d'elle dépend l'expressivité de l'éditeur de modèles généré. Le défi ici c'est que les symboles graphiques doivent évoquer les concepts du domaine qu'ils représentent. Cela nécessite de pouvoir combiner librement tous les éléments graphiques de bases à savoir lignes, courbes, texte ... pour faire des symboles qui soient spécifiques au domaine. Dans la syntaxe concrète on ne spécifie pas que les symboles,mais également comment ces symboles vont êtres positionnés à l'écran, comment les déplacer, comment les situer les uns par rapport aux autres ... Tout cela est fait en fonction du domaine. Par exemple, la relation de composition entre deux objets, dans notre contexte sera représentée comme à la Figure 6.1, mais on peut également faire le choix de ne représenter que les composants de premier niveau. Dans le cas des relations, la spécification de la syntaxe concrète présente quelques spécificités. Le symbole d'une relation est souvent une liaison dont il faut alors spécifier la forme, la position sur l'objet source et la position sur la cible. Il faut spécifier comment la liaison va tourner autour des objets qu'elle relie lorsqu'ils sont déplacés, sachant qu'elle doit être positionnée à des points prédéfinis. Par exemple dans notre contexte, des connecteurs doivent êtres reliés comme présenté à la Figure 6.2(a) , les relier comme à la Figure 6.2(b) ou comme à la Figure 6.2(c) n'aurait aucun sens pour nous. Enfin, il faut également spécifier la syntaxe concrète pour du texte (labels, propriétés des objets, noms des liaisons ...), il faut spécifier où positionner le texte, comment l'afficher (à l'horizontale, à la verticale, incliné), gérer sa longueur quand il doit être affiché sur un objet ou une liaison, spécifier la police la fonte .... La spécification de la syntaxe concrète nécessite un temps très significatif lorsqu'elle doit être entièrement programmée par l'utilisateur.

– **Spécification de la sémantique** : En plus des règles structurelles spécifiées dans la syntaxe abstraite, il faut que l'outil offre un langage tel que OCL permettant d'exprimer des règles non structurelles. Cela contribue à définir le langage de façon précise.
– **Génération de l'éditeur de modèles** : à partir de la définition de la syntaxe et de la définition de la sémantique, l'outil doit permettre de générer automatiquement

l'éditeur de modèles permettant d'instancier le langage. Ce qui signifie qu'une fois le DSML défini, l'outil de méta-modélisation doit permettre de créer l'éditeur dédié à ce langage sans que l'utilisateur ait besoin de le programmer.

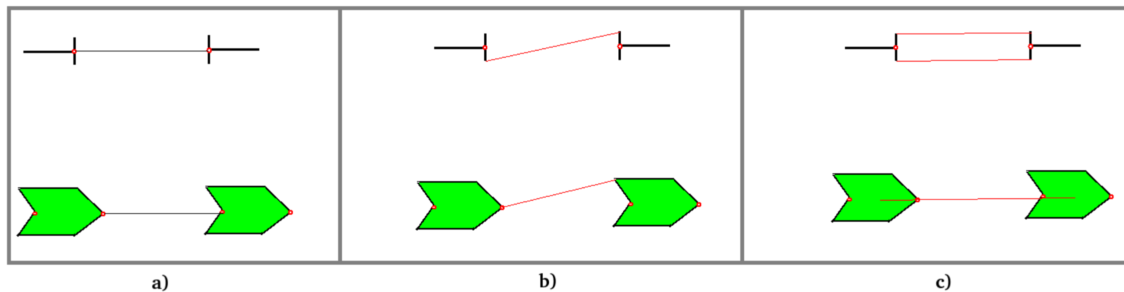


FIGURE 6.2 – Liaisons spécifiques

6.3 Critères d'évaluation

Nos travaux portent essentiellement sur la syntaxe (abstraite et concrète) des DSMLs. Nous n'évaluerons donc pas les différents outils en fonction de la sémantique, par contre nous précisons à chaque fois si l'outil permet la gestion de la sémantique (exécution des modèles) ou pas. Nous présentons chacun des outils en fonction de l'approche qu'il utilise.

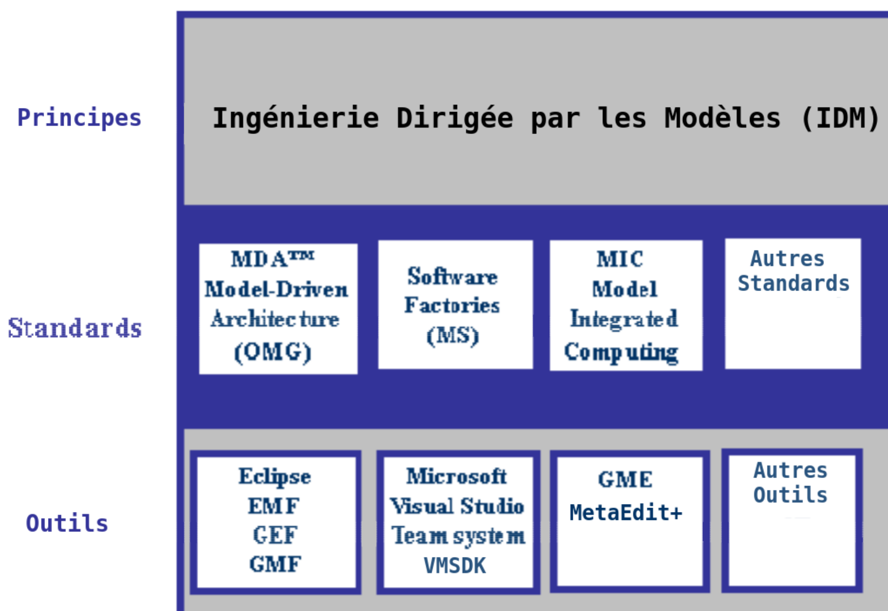


FIGURE 6.3 – Outils de l'IDM

Comme présenté à la Figure 6.3, l'approche MDA de l'OMG est implantée dans différents projets de modélisation de la plate-forme Eclipse (*Eclipse Modeling Project*-EMP).

Microsoft a implanté son approche sur les usines à logiciel dans l'environnement *Visual Studio SDK* (VMSDK). L'approche MIC quant à elle est implantée dans des environnements tels que GME et Metadit+.

Nous évaluons le degré de spécificité et de généricité de chacun des outils en fonction de plusieurs critères permettant de les ranger dans la catégorie des outils de méta-modélisation spécifiques ou celle des outils de méta-modélisation génériques. Ces critères sont les suivants :

- **Syntaxe abstraite** : le langage de méta-modélisation fourni est-il
  - standard : est-ce un standard connu ou un langage propriétaire ? L'utilisation d'un standard connu réduit le temps d'apprentissage nécessaire pour appréhender le langage de méta-modélisation. La standardisation facilite également l'interopérabilité entre les différents outils.
  - interopérable : peut-on échanger les méta-modèles avec d'autres outils ? Le développement d'un logiciel fait souvent intervenir plusieurs outils différents, il est donc important que l'outil de méta-modélisation fonctionne bien avec les outils utilisés en amont ou en aval.
  - complet : les concepts fournis par le langage de méta-modélisation sont-ils suffisamment génériques pour permettre de décrire la structure de tout DSML de façon précise ? Par exemple, peut-on faire des sous-modèles ? permet-il des relations n-aires ?
  - intuitif : est-il simple d'utilisation, les concepts qu'il offre sont-ils suffisamment évocateurs ? Par exemple, si un langage de méta-modélisation offre des concepts tels que "*classe*" et "attribut", ces concepts trouveront d'emblée un écho chez un utilisateur habitué à la programmation orientée objet.
- **Syntaxe concrète**
  - modularité : la syntaxe concrète est-elle séparée de la syntaxe abstraite ? Dans le domaine de la méta-modélisation il est communément admis que la syntaxe abstraite doit être séparée de la syntaxe concrète. Bien que la syntaxe concrète permette d'implémenter un éditeur dédié, elle ne donne pas plus de précision au sens du langage et elle peut même polluer la syntaxe abstraite en rendant son interprétation plus difficile.
  - assistant graphique : l'outil fournit-il un assistant graphique permettant de définir les notations graphiques du langage sans avoir besoin de les programmer ?
- **Sémantique**
  - langage de contraintes : l'outil offre-t-il un langage permettant d'exprimer les contraintes non structurelles ?
  - sémantique dynamique : l'outil offre-t-il un support pour l'exécution des modèles ?
- **Génération de l'éditeur**
  - Automatique : la génération de l'éditeur se fait elle de façon automatique ? C'est à dire, une fois le langage défini (Syntaxe et sémantique), la création d'un éditeur permettant d'instancier ce langage nécessite-t-elle de la programmation ou pas ?
  - Évolutivité : Lorsqu'on modifie le langage est-ce que cela nécessite de reprendre tout le processus de génération de l'éditeur depuis le début ? La rétrocompatibilité des modèles existants est-elle assurée ?

  - utilisabilité de l'éditeur : l'éditeur offre-t-il toutes les fonctionnalités de base nécessaires à la modélisation (sauvegarde/chargement, création, zoom ...)
- **Utilisabilité**
  - documentation disponible : y a-t-il suffisamment de documentation permettant de comprendre le fonctionnement de l'outil ?
  - flexibilité : l'outil permet-il à l'utilisateur de parvenir à ses fins quelque soit les spécificités de son langage ? En d'autres termes, est-ce que l'utilisateur a la possibilité d'exprimer tout type de langage ou est-ce qu'il peut être limité par l'outil.
  - complexité : l'outil est-il complexe d'utilisation ? est-il difficile d'implanter un langage dédié avec cet outil ? Un outil basé sur le code est très générique, mais complexe d'utilisation. Un outil plus abstrait (plus automatique) ne permet pas de tout faire, il a donc une portée restreinte, mais par contre il est plus simple d'utilisation.

Parmi tous ces critères, ceux qui apparaissent comme étant les plus pertinents pour nos travaux sont : (1) la présence ou non d'un assistant graphique permettant de simplifier la spécification de la syntaxe concrète. La portée de cet assistant graphique, à savoir quel type de notations graphiques il fournit. (2) tous les critères relatifs à l'utilisabilité de l'outil de méta-modélisation.

6.4 Implantation de MDA : *Eclipse Modeling Project* (EMP)

IBM a implanté sa vision de MDA dans l'*Eclipse Modeling Project* (EMP). EMP est un projet open source qui regroupe un ensemble de sous projets chacun centré sur un aspect particulier de MDA (méta-modélisation, transformation des modèles ...). L'objectif de ce projet est de créer un ensemble unifié de frameworks, d'outils et de normes pour prendre en charge le développement dirigé par les modèles. Trois de ces projets permettent principalement de concevoir des DSMLs et de générer des éditeurs dédiés, il s'agit de EMF (*Eclipse Modeling Framework*), GEF (Graphical Editing Framework) et GMF (Graphical Modeling Framework).

6.4.1 EMF

EMF est un framework de méta-modélisation et de génération de code qui permet de spécifier des méta-modèles et de gérer ses instances (création, édition, sauvegarde ...). Pour spécifier un méta-modèle, EMF utilise le langage de méta-modélisation Ecore défini par IBM. Ecore est complet et simple d'utilisation. Il utilise la syntaxe graphique de UML qui est relativement bien connue et de ce fait est assez intuitif. Les concepts qu'il fournit (Packages, Classes, associations, attributs ...) sont assez génériques pour convenir à tous les domaines. La persistance du méta-modèle est effectuée en XMI (*XML Metadata Interchange*) qui est un des standards de l'OMG ce qui favorise l'interopérabilité avec d'autres outils de méta-modélisation. EMF n'est pas une implantation stricte des spécifications

de l'OMG. En effet, Ecore dérive légèrement de la spécification de l'OMG pour EMOF. Toutefois, Ecore est compatible avec EMOF et EMF supporte l'import-export des modèles EMOF. EMF peut également prendre en entrée plusieurs autres formats de méta-modèles (UML, Java Annoté, Schéma XML, XMI), il peut donc être compatible avec d'autres outils utilisés en amont. De façon générale, la plupart des utilisateurs trouvent l'environnement de méta-modélisation d'EMF simple d'utilisation [Pelechano et al. 2006].

À partir de la spécification d'un méta-modèle (fichier .ecore), EMF génère le code permettant de gérer la structure du méta-modèle sous forme de classes et d'interfaces Java. Ces classes peuvent êtres utilisées pour générer un éditeur de modèles basique. La faiblesse de EMF en ce qui concerne les DSMLs, est que l'objectif premier de ce framework est de générer du code pour accélérer le développement logiciel. EMF ne fournit pas de support pour la syntaxe concrète. Un éditeur fourni permet de créer des modèles uniquement sous forme arborescente ce qui n'est pas très expressif.

6.4.2 GEF

GEF fournit un support pour la construction d'éditeurs graphiques au-dessus de EMF. GEF utilise une architecture MVC (Modèle Vue Contrôleur) où le Modèle peut être un méta-modèle EMF. Le Contrôleur s'appelle *EditPart* et est chargé de mettre à jour l'éditeur en fonction des changements du méta-modèle. Pour chaque classe Java représentant un élément du méta-modèle EMF, l'utilisateur doit créer la classe *EditPart* correspondante. Chaque *EditPart* a une figure qui est la représentation graphique de l'élément correspondant dans le méta-modèle. Cette représentation graphique est implantée avec le Framework *Draw2D*, un Framework de dessin 2D basé sur SWT qui est un *toolkit* pour développer des interfaces graphiques en Java. La partie Vue de GEF (qui est la syntaxe concrète) correspond alors à un éditeur permettant de manipuler ces figures.
En fait, définir un symbole graphique avec GEF et implanter son comportement graphique tel que défini à la section 6.2 revient à écrire tout le code Java de ce symbole (chaque ligne et chaque courbe) ainsi que le code relatif à son comportement graphique. Cela offre certes beaucoup de liberté et la possibilité de construction d'éditeurs graphiques pour presque tous les domaines, mais le fait de devoir tout coder à la main peut rapidement devenir très contraignant et c'est pourquoi GMF a été créé.

6.4.3 GMF

GMF a été créé pour pallier les difficultés d'utilisation de GEF et aussi pour faire le pont entre EMF et GEF. GMF repose sur GEF et utilise toutes les caractéristiques de EMF pour concevoir des éditeurs graphiques. GMF est constitué de deux composantes principales : *GMF Tooling* qui consiste en un ensemble d'éditeurs permettant de créer et d'éditer la syntaxe abstraite d'un DSML, ses notations graphiques ainsi que sa sémantique. Le plug-in généré à partir de ces spécifications repose sur le composant *GMF Runtime*. *GMF*

Runtime fournit plusieurs fonctionnalités que le développeur devrait implanter lui-même s'il ne l'utilisait pas (par exemple l'impression, l'exportation des images, la barre d'outil . . .) ; la création d'éditeurs graphiques sous GMF est faite avec *GMF Tooling* et l'environnement d'exécution est *GMF Runtime*.
Pour implanter un éditeur graphique avec GMF le développeur doit d'abord spécifier la syntaxe abstraite de son langage sous EMF sous forme de méta-modèle Ecore. Ensuite, il doit construire plusieurs méta-modèles intermédiaires pour spécifier la syntaxe visuelle (syntaxe concrète) du langage. Cela est fait dans le but de dissocier la syntaxe abstraite de la syntaxe concrète. Tous ces modèles intermédiaires sont ensuite utilisés pour générer le plug-in Eclipse qui implante l'éditeur graphique dédié.

Le processus de développement d'un DSML en utilisant ces outils de EMP est le suivant :

6.4.4 Spécification de la syntaxe abstraite

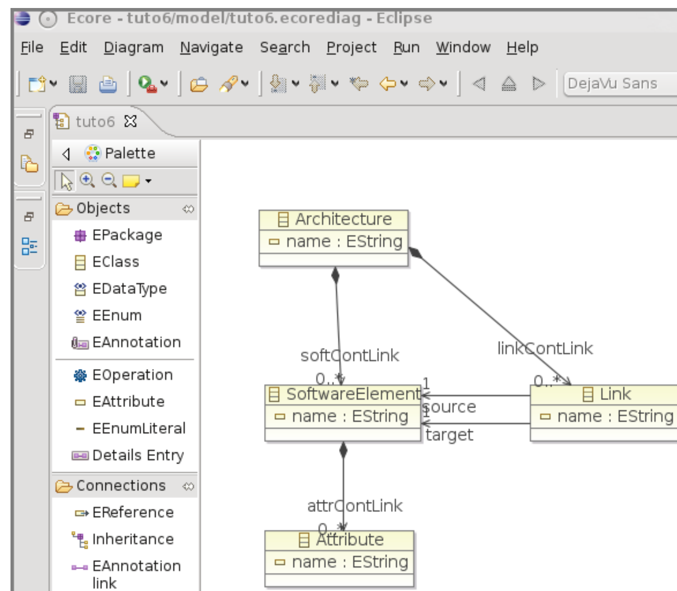


FIGURE 6.4 – exemple de domain model

Cela consiste en la création du méta-modèle définissant les concepts du DSML, leurs propriétés et leurs relations. Il est appelé *domain model* et est défini en Ecore. Il peut être créé soit par l'éditeur standard EMF ou par l'éditeur graphique *Ecore Tools* qui est un composant du projet EMFT (*Eclipse Modeling Framework Technology*) permettant de manipuler graphiquement des modèles Ecore.
Un exemple de *domain model* pour un DSML de définition d'architectures logicielles (ADL) est présenté à la Figure 6.4. Dans cet exemple, le méta-modèle représente une *Architecture* composée de logiciels (*SoftwareElements*). Les logiciels sont reliés entre eux par des liaisons (*Links*) et chacun d'eux possède une liste d'attributs (*Attributes*).

6.4.5 Spécification de la syntaxe concrète

La définition de la syntaxe concrète est séparée de celle de la syntaxe abstraite. Elle se fait à travers plusieurs modèles différents.

Définition des notations graphiques : Spécifier la syntaxe concrète revient à spécifier la représentation graphique qui sera associée aux différents concepts du langage. Pour cela, il faut créer un modèle appelé *graph model* qui va définir les figures qui seront utilisées dans l'éditeur à générer. La Figure 6.5 présente un exemple de *graph model* qui définit une figure qui est un rectangle arrondi. Cette figure sera associée plus tard à un élément du *domain model*.

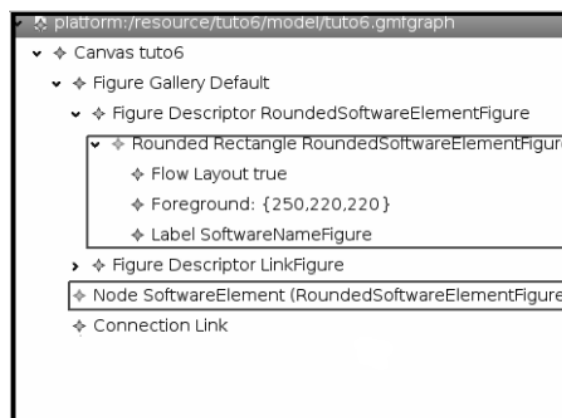


FIGURE 6.5 – Graph model

GMF fournit un assistant graphique qui propose une liste prédéfinie de notations graphiques. Dans notre exemple, comme on peut le voir sur la Figure 6.5, nous avons choisi d'utiliser un *RoundedRectangle*. Le problème qui se pose est que cette liste de notations graphiques est très limitée et consiste essentiellement en des rectangles et des liaisons qui ne sont satisfaisants que pour des cas triviaux. Il est par exemple impossible de spécifier quels seront les points d'ancrage des liaisons sur leur source et leur cible. Pour le faire, il faut nécessairement retourner dans GEF et le coder à la main. Mais comme nous l'avons déjà écrit, cela peut être très contraignant d'autant plus qu'il faut avoir une bonne expertise de comment fonctionne EMF et GEF et de la façon dont ils sont combinés [Amyot et al. 2006].

Description des outils : Cela consiste à définir dans un modèle appelé *tooling model*, les outils qui seront disponibles dans la palette de l'éditeur. Par exemple un outil de création qui permettra de créer une nouvelle instance d'un concept (Figure 6.6).

Définition de la correspondance entre les notations graphiques et des concepts du langage : Enfin, il faut créer un *mapping model* qui permet de spécifier la correspon-

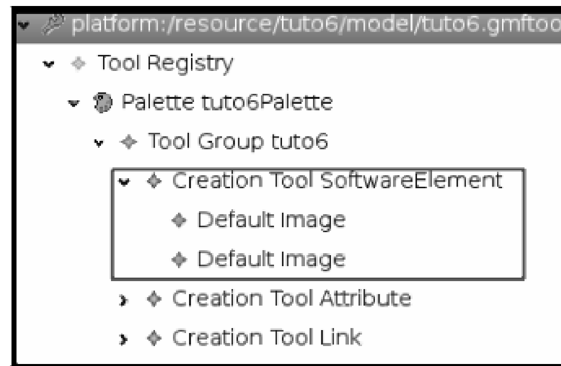


FIGURE 6.6 – Tooling model

dance entre les figures du *graph model*, les éléments de la syntaxe abstraite et les outils du *tooling model*. Par exemple dans le *mapping model* de la Figure 6.7, le rectangle arrondi de la Figure 6.5 et l'outil de création de la Figure 6.6 sont associés au concept *SoftwareElement* du *domain model*. Dans l'éditeur graphique généré, on se servira donc de cet outil de création pour instancier un *SoftwareElement*, et le *SoftwareElement* sera représenté par un rectangle arrondi.

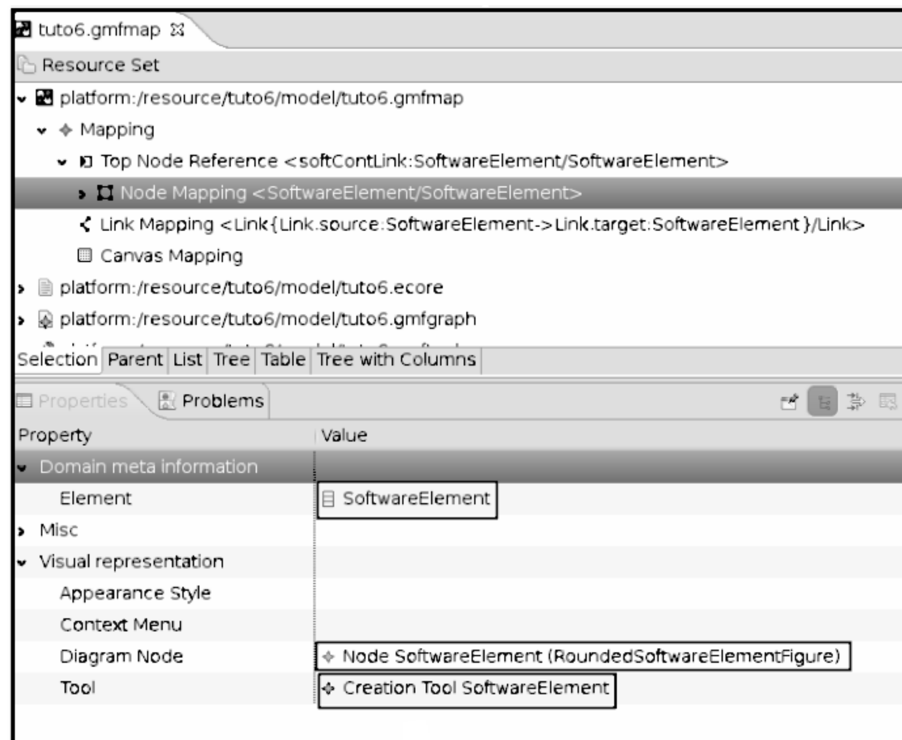


FIGURE 6.7 – exemple de mapping model

6.4.6 Spécification de la sémantique

Le projet MDT (*Model Development Tools*) d'Eclipse fournit une implémentation du standard OCL de l'OMG pour les modèles EMF. Ce langage de contraintes permet d'exprimer toutes les contraintes non structurelles pour assurer des modèles cohérents et précis. Notons aussi que les solutions EMF, GEF et GMF ne fournissent aucun support pour l'exécution des modèles. Pour cela il faut recourir à des solutions telles que Kermeta [Drey and Vojtisek 2006] qui est également basé sur Eclipse et sur EMF.

6.4.7 Génération de l'éditeur

Une fois les quatre modèles définis, ils sont combinés dans un cinquième modèle appelé *generator model*. Ce *generator model* regroupe les quatre premiers et permet de générer le code Java d'un plug-in d'Eclipse qui implémente l'éditeur. L'assistant graphique que propose GMF permet de générer une première version des modèles intermédiaires (*graphical*, *tooling* and *mapping*) à partir du *domain model*. Ensuite, ces modèles générés peuvent être raffinés. Quant à la création du *generator model* elle est faite de façon automatique de même que la génération de l'éditeur. Mais si on modifie même légèrement le méta-modèle il faut alors répéter tout le processus de génération du code et il y a toujours un risque important que des erreurs se glissent dans l'un des modèles intermédiaires. Dans ce cas, soit GMF renvoie des erreurs de bas niveau difficiles à résoudre pour un novice, soit il génère un code erroné.

L'éditeur graphique généré est suffisamment robuste et simple d'utilisation avec de nombreuses fonctionnalités fournies (zoom, sauvegarde, palette d'outils, exportation d'images ...). L'éditeur graphique obtenu dans notre exemple est présenté à la Figure 6.8, nous y décrivons une architecture JEE basique qui comprend un serveur Apache, un serveur Tomcat et un serveur Mysql.

6.4.8 Synthèse

Les solutions EMF, GEF et GMF ont fait l'objet de plusieurs autres évaluations dans le cadre d'expériences ou d'études comparatives sur les outils de méta-modélisation [Kelly 2004][Pelechano et al. 2006][Amyot et al. 2006][Evans et al. 2009][Ozgur 2009][El Kouhen et al. 2012]. Conformément à notre analyse, toutes les études s'accordent à dire que GMF est assez robuste, mais complexe d'utilisation. Il faut beaucoup d'effort pour apprendre le fonctionnement de GMF et comprendre comment générer un éditeur. De plus, le manque de documentation ne favorise pas une prise en main rapide de l'outil d'où un temps d'apprentissage élevé. Plusieurs outils basés sur Eclipse ont été développés dans le but de pallier la complexité d'utilisation de EMF/GEF/GMF. Par exemple Eclipse Graphiti [eclipse 2012] qui fournit une API au dessus de GEF pour masquer la complexité de GEF et Draw2D. Graphiti est en effet moins complexe que GEF parce qu'il est plus

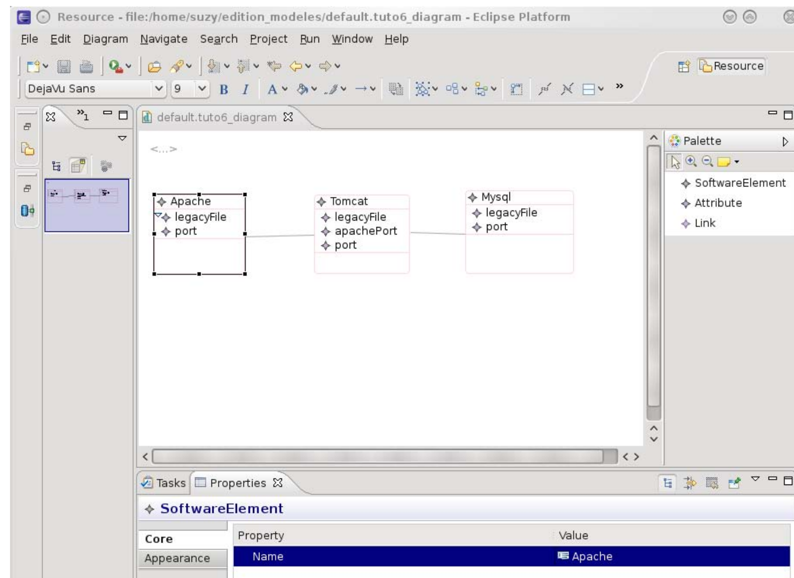


FIGURE 6.8 – Éditeur généré

abstrait, l'utilisateur a juste besoin de connaître EMF pour l'utiliser. Graphiti est également mieux documenté, mais en élevant ainsi le niveau d'abstraction de GEF, il devient aussi beaucoup moins flexible. On peut également citer Eugenia [Kolovos et al. 2009] qui propose d'annoter le méta-modèle EMF avec des éléments de la syntaxe concrète pour générer plus facilement les modèles intermédiaires de GMF (*graph*, *tooling*, *mapping*). Eugenia tout comme graphiti est plus simple d'utilisation parce que plus abstrait, mais aussi moins flexible. Enfin, le framework commercial Obeo Designer [Company 200x] utilise la notion de *point de vue* pour fournir à l'utilisateur un ensemble de représentations graphiques centrées sur un problème particulier. Obeo Designer est un outil commercial assez robuste et facile d'utilisation comparé à la solution EMF/GEF/GMF mais il est également beaucoup moins flexible que cette dernière.

La solution EMF/GEF/GMF peut être considérée comme étant une solution générique. C'est un framework qui permet à l'utilisateur de programmer tout le comportement graphique et toutes les notations graphiques spécifiques qu'il souhaite. Cela laisse beaucoup de liberté pour construire des éditeurs graphiques bien adaptés aux besoins des experts du domaine. Toutefois, cette généricité requiert une grande expertise en programmation et au final, la quantité de code nécessaire peut décourager les développeurs.

6.5 Autre implantation de MDA : DiaMeta

Le projet DiaMeta [Minas 2012] développé à l'université de Bundeswehr München permet de créer des éditeurs graphiques dédiés *stand-alone* [1] à partir d'un méta-modèle. Ce projet est une extension du projet DiaGen [Minas 2012] qui permet de générer un

1. Application à part entière à la différence d'un plug-in

Syntaxe abstraite - Langage de méta-modélisation	
Standard	oui, Ecore est conforme aux spécifications de l'OMG
Interopérable	oui, les standards de l'OMG favorisent l'interopérabilité
Complet	non, concepts suffisamment génériques, mais pas de relation n-aire et pas de notion de sous modèle
Intuitif	oui, sa syntaxe graphique est similaire à celle de UML qui est connue
Syntaxe concrète	
Modularité	oui, la syntaxe concrète est séparée de la syntaxe abstraite
Assistant graphique	oui, mais très limité
Sémantique	
Langage de contraintes	OCL
Sémantique dynamique	non
Génération de l'éditeur	
Automatique	oui
Évolutivité	non
Utilisabilité de l'éditeur généré	grande
Utilisabilité de l'outil	
Documentation	faible
Flexibilité	grande [1]
Complexité	générique, mais très complexe d'utilisation

[1] Moyennant une expertise en programmation

TABLE 6.1 – Résumé de EMP

éditeur à partir d'une grammaire basée sur des hypergraphes [2]. L'environnement de DiaMeta consiste en deux parties : l'*editor framework* et le *DiaMeta DESIGNER*. L'*editor framework* est constitué d'un ensemble de classes Java qui implantent les fonctionnalités génériques de l'éditeur nécessaires à l'édition et l'analyse des modèles, alors que le *DiaMeta DESIGNER* permet de spécifier la syntaxe concrète d'un DSML. Pour créer un éditeur dédié pour un DSML, le développeur doit premièrement spécifier la syntaxe abstraite du DSML sous forme de méta-modèle ; et deuxièmement, spécifier sa syntaxe concrète ce qui se traduit par la définition des représentations graphiques des concepts du DSML, la définition des règles d'analyse syntaxique des modèles et enfin la définition des interactions entre les concepts du DSML.

2. Objets mathématiques généralisant la notion de graphe dans le sens où les arêtes ne relient plus un ou deux sommets, mais un nombre quelconque de sommets

6.5.1 Spécification de la syntaxe abstraite

DiaMeta utilise l'environnement EMF pour définir le méta-modèle et générer son code source. Il repose donc également sur les standards de l'OMG. Pour notre exemple, la syntaxe abstraite restera la même que celle présentée à la Figure 6.4.

6.5.2 Spécification de la syntaxe concrète

Le développeur utilise *DiaMeta DESIGNER* (Figure 6.9) pour spécifier les représentations graphiques des éléments du méta-modèle (fichier spec.dds). *DiaMeta DESIGNER* génère le code Java correspondant à cette spécification. Ce code associé au code généré par EMF et à celui de l'*editor framework* implémente un éditeur dédié pour le DSML.
Le DiaMeta DESIGNER fournit une liste prédéfinie de représentations graphiques, essentiellement des figures de base telles que des cercles, des rectangles, des ellipses . . . Si l'utilisateur souhaite avoir une représentation graphique plus élaborée alors il doit la programmer lui même. La spécification de la syntaxe concrète avec DiaMeta est d'assez bas niveau. Par exemple pour une liaison, l'utilisateur doit spécifier à la main comment calculer les coordonnées de l'extrémité de la liaison sur la cible. Pour notre exemple d'ADL nous avons dû spécifier où placer les *SoftwareElements* quand on les instancie (leurs coordonnées), comment les déplacer (calcul des nouvelles coordonnées), lorsqu'on clique sur deux *SoftwareElements* pour les relier où doit se positionner la liaison . . .

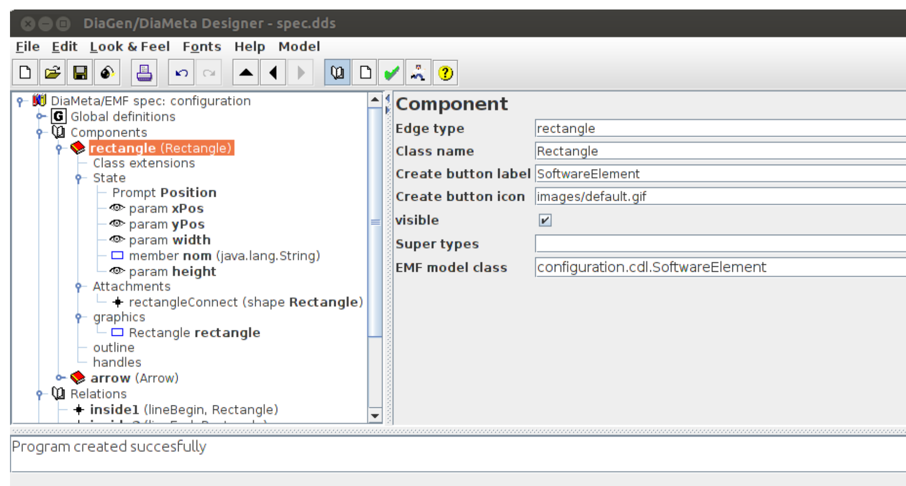


FIGURE 6.9 – DiaMeta DESIGNER

6.5.3 Spécification de la sémantique

DiaMeta n'offre pas de véritable langage de contraintes permettant d'exprimer les contraintes non structurelles, mais en revanche il permet l'analyse et la simulation des modèles.

6.5.4 Génération de l'éditeur

Une fois le DSML défini, la génération de l'éditeur se fait de façon automatique. L'éditeur généré peut être très robuste et très expressif moyennant une maîtrise de Java. L'un des principaux avantages de DiaMeta c'est que l'éditeur généré permet un mode d'édition en *main libre* (*free-hand editing*), c'est-à- dire que pour créer un modèle, l'utilisateur peut agencer les concepts du langage et les positionner à l'écran comme il veut. Cela implique que les modèles incorrects sont parfaitement admis et juste signalés à l'utilisateur. Ce mode d'édition est pratique et permet d'améliorer la flexibilité et la créativité de l'utilisateur. L'éditeur généré pour notre exemple est représenté à la Figure 6.10.

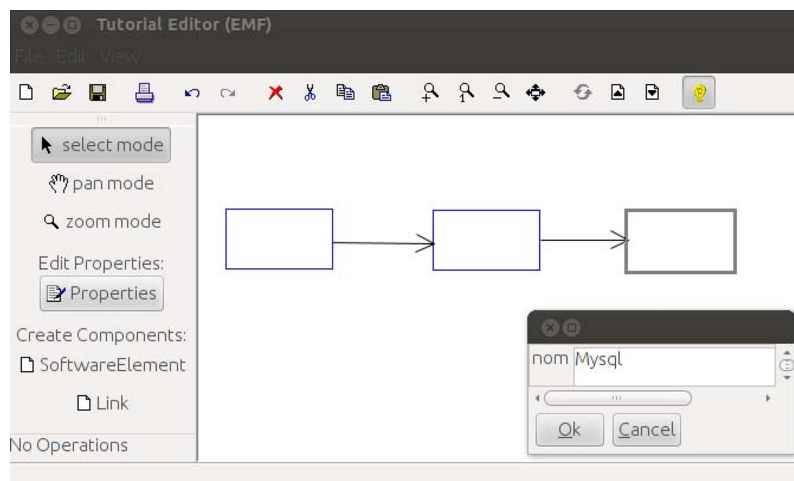


FIGURE 6.10 – Éditeur généré avec DiaMeta

6.5.5 Synthèse

DiaMeta est un outil très générique certes plus abstrait que DiaGen, mais qui nécessite tout de même beaucoup d'effort programmation de la part de l'utilisateur. Cet outil offre donc une grande liberté à l'utilisateur qui peut alors implanter toutes les spécificités voulues, mais cette liberté a un coût qui est une plus grande complexité d'utilisation.

6.6 Implantation des usines à logiciel : VMSDK

En 2004, Microsoft a implanté son approche sur les usines à logiciel dans l'environnement de développement *Visual Studio SDK*. *Visual Studio SDK* est dédié à la personnalisation de l'IDE *Visual Studio* de Microsoft (qui repose largement sur .NET) pour des besoins ou des domaines spécifiques [Bézivin et al. 2005]. La partie en charge des DSMLs dans cet environnement était connue sous le nom de *DSL Tools*, mais en 2010 elle a été renommée en *Visualization and Modeling SDK* (VMSDK) et est désormais livrée avec

Syntaxe abstraite - Langage de méta-modélisation	
Standard	oui, Ecore est conforme aux spécifications de l'OMG
Interopérable	oui, les standards de l'OMG favorisent l'interopérabilité
Complet	non, concepts suffisamment génériques, mais pas de relation n-aire et pas de notion de sous modèle
Intuitif	oui, sa syntaxe graphique est similaire à celle de UML qui est connue.
Syntaxe concrète	
Modularité	oui, la syntaxe concrète est séparée de la syntaxe abstraite
Assistant graphique	oui, mais très limité
Sémantique	
Langage de contraintes	non
Sémantique dynamique	oui, analyse et simulation des modèles
Génération de l'éditeur	
Automatique	oui
Évolutivité	non
Utilisabilité de l'éditeur généré	grande
Utilisabilité de l'outil	
Documentation	faible
Flexibilité	grande [1]
Complexité	générique, mais très complexe d'utilisation

[1] Moyennant une expertise en programmation

TABLE 6.2 – Résumé de DiaMeta

Visual Studio. VMSDK est un kit de développement qui permet de créer, à partir d'une modélisation des concepts métiers (méta-modèle), un environnement de conception personnalisé dans *Visual Studio*.
Cet environnement fournit un assistant (***Project Wizard***) pour la création, l'édition, la visualisation et l'utilisation des DSMLs. La création d'un environnement dédié suit les mêmes étapes que pour EMP.

6.6.1 Spécification de la syntaxe abstraite

C'est la définition de la structure du langage dédié sous forme de méta-modèle constitué de classes et de liaisons. Dans VMSDK, un méta-modèle est toujours créé à partir de *templates* de langages tel que les diagrammes de classes, les modèles à composants, l'exécution de tâches ou encore un langage minimal. Il est impossible de créer un méta-modèle initialement vide. Le méta-modèle est appelé *domain model*. Le ***graphical designer*** permet de définir le *domain model* en utilisant une notation propriétaire. Cette nota-

tion propriétaire qui de plus est moins intuitive que celle d'Ecore [Pelechano et al. 2006] ne favorise pas l'interopérabilité avec d'autres outils. Toutefois, les concepts fournis pour la méta-modélisation, *domainClass* pour classe et *domainRelationShip* pour liaison sont des classiques. La définition du méta-modèle se fait dans un fichier.dsl et la sauvegarde est faite dans un format XML propriétaire. La figure 6.11 présente notre exemple de DSML pour les architectures logicielles, conçues avec le *graphical designer* de VMSDK.

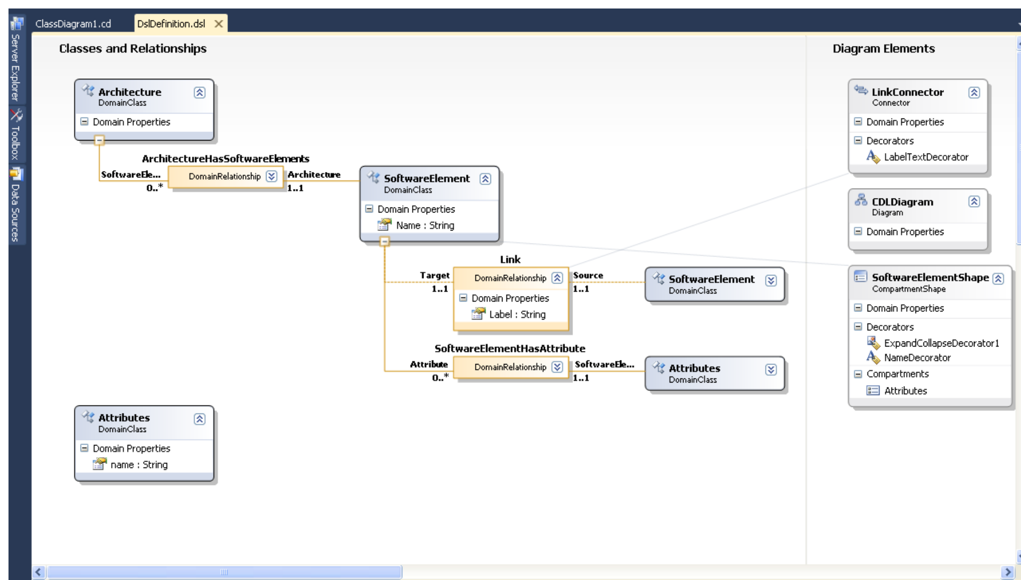


FIGURE 6.11 – Graphical Designer

6.6.2 Spécification de la syntaxe concrète

Spécifier la syntaxe concrète revient à définir les notations graphiques et les faire correspondre à des éléments de la syntaxe abstraite.

Définition des notations graphiques :
Dans le *domain model* on définit des formes et des connecteurs (partie *Diagram Elements* sur la Figure 6.11). Cette notation graphique est sauvegardée dans un fichier XML séparé (.dsl.diagram). VMSDK offre une liste prédéfinie de notations graphiques telles que des rectangles, des ellipses ou des icônes. La définition de la notation graphique y est beaucoup plus simple que dans GMF parce que le *graphical designer* est assez intuitif. Toutefois si la notation graphique souhaitée diverge de celle qui est fournie par défaut (par exemple, l'utilisateur souhaite avoir une liaison spéciale avec des carrés aux extrémités), alors il doit la programmer lui même en c# et cela devient plus compliqué.

Définition de la correspondance entre notation graphique et concepts du langage :
On associe les formes définies aux classes définies et les connecteurs aux liaisons. Sur

la Figure 6.11 cette association est matérialisée par les liens entre la partie *Classes and Relationships* et la partie *Diagram Elements*.

6.6.3 Spécification de la sémantique

VMSDK comme son nom l'indique insiste sur l'aspect visualisation et modélisation et ne fournit aucun support pour l'analyse des modèles et aucun langage pour l'expression des contraintes non structurelles.

6.6.4 Génération de l'éditeur

Une fois le *domain model* défini, on génère automatiquement le code (.cs file) associé. Le *Project Wizard* fournit un ensemble de générateurs de code qui prennent en entrée la définition du *domain model* et produisent le code source du langage. L'environnement dédié va être généré à partir de ce code source, tout ceci est fait de façon transparente à l'utilisateur. Ces générateurs de code permettent également de valider le *domain model* et le cas échéant font remonter des erreurs. Le code rajouté manuellement par l'utilisateur est séparé du code généré pour empêcher qu'une nouvelle génération de code n'affecte le code rajouté et ne crée des problèmes de synchronisation. La Figure 6.12 présente l'environnement dédié généré pour notre exemple. L'environnement de modélisation généré est robuste et bénéficie de toutes les fonctionnalités implantées dans *Visual Studio*.

6.6.5 Synthèse

VMSDK est considéré comme étant un environnement robuste et simple d'utilisation [Ozgur 2009] toutefois le temps d'apprentissage de l'outil est élevé pour ceux qui n'ont jamais utilisé *Visual studio*. Ceci est accentué par le fait qu'il est difficile de trouver une bonne documentation. Cette solution peut être considérée comme étant une solution à la fois spécifique et générique. VMSDK peut être considéré comme un environnement spécifique d'abord à cause de son langage de méta-modélisation qui est spécifique. Mais aussi parce que tant qu'on ne diverge pas trop des *templates* fournis l'outil est assez abstrait et intuitif. Lorsqu'il s'agit d'implanter des solutions vraiment spécifiques, cela nécessite du code et une bonne compréhension des mécanismes sous-jacents. VMSDK peut également être vu comme un environnement générique dès lors qu'on veut implanter un DSML avec des spécificités graphiques non prises en charge, parce que l'utilisateur peut alors les programmer lui même.

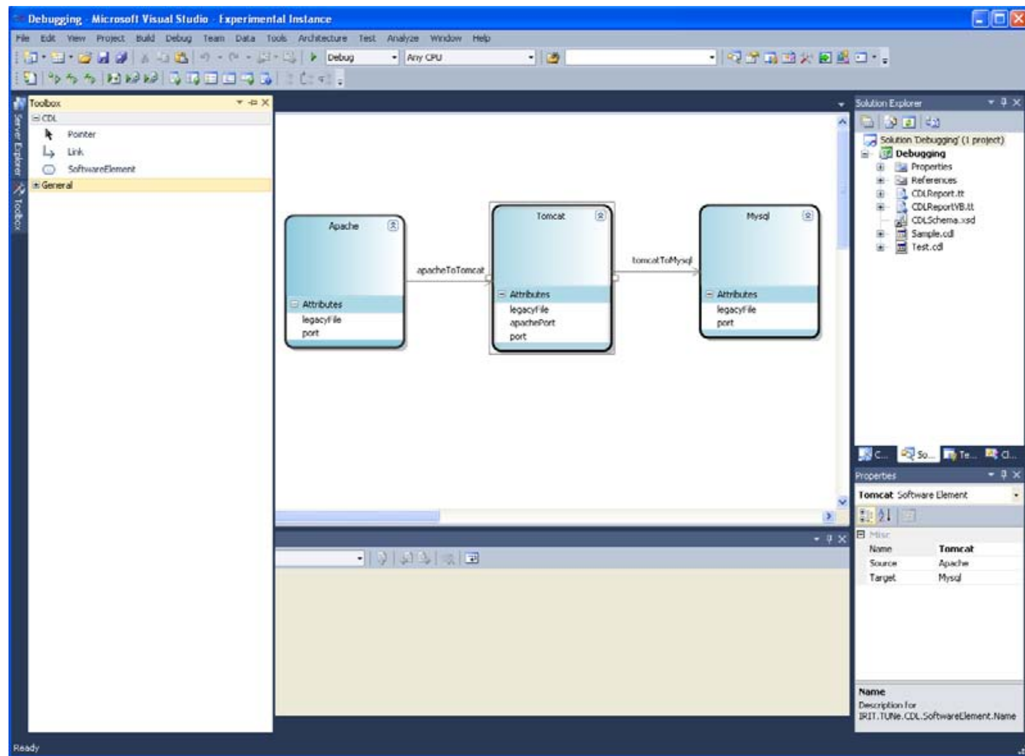


FIGURE 6.12 – Environnement généré

6.7 Implantation de MIC : GME

Le *Generic Modeling Environment* (GME) [Davis 2003] est un outil de méta-modélisation configurable développé à l'institut ISIS (*Institute for Software Integrated Systems*) de l'université de Vanderbilt et qui permet de créer des environnements de modélisation dédiés. GME est une solution issue du monde de la recherche qui utilise l'approche MIC et a une architecture basée sur la technologie COM (*Component Object Model*) de Microsoft.
Dans GME, un DSML est décrit sous forme de paradigme. C'est essentiellement un méta-modèle qui permet de créer un environnement de modélisation dédié. On configure GME en spécifiant un méta-modèle qui définit la famille des modèles pouvant êtres créés en utilisant cet environnement de modélisation. La Figure 6.13 présente le paradigme défini pour notre exemple de DSML pour des architectures logicielles.
Un paradigme contient la définition de la syntaxe, de la sémantique ainsi que de la représentation du domaine - quels concepts vont êtres utilisés pour construire les modèles, quelles sont les relations entre ces différents concepts, comment sont-ils organisés et représentés, et enfin quelles sont les règles qui dirigent la construction des modèles [ISIS 2012].

Syntaxe abstraite - Langage de méta-modélisation	
Standard	non, langage de méta-modélisation propriétaire
Interopérable	non, format propriétaire
Complet	oui
Intuitif	non
Syntaxe concrète	
Modularité	oui, la syntaxe concrète est séparée de la syntaxe abstraite
Assistant graphique	oui, mais très limité
Sémantique	
Langage de contraintes	non
Sémantique dynamique	non
Génération de l'éditeur	
Automatique	oui
Évolutivité	non, si on change juste le nom d'un méta-modèle, les modèles créés avant sont supprimés.
Utilisabilité de l'éditeur généré	grande
Utilisabilité de l'outil	
Documentation	faible
Flexibilité	grande [1]
Complexité	générique, mais complexe d'utilisation

[1] Moyennant une expertise en programmation

TABLE 6.3 – Résumé de VMSDK

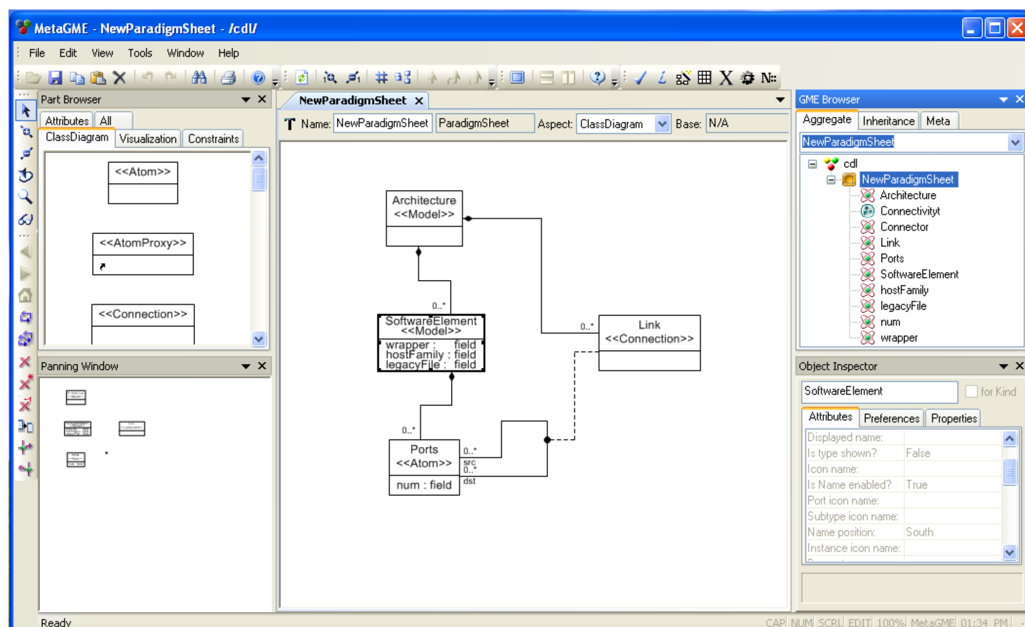


FIGURE 6.13 – Paradigme GME

6.7.1 Spécification de la syntaxe abstraite

Ici on définit la structure du langage dédié sous forme de méta-modèle. Le méta-méta-modèle de GME offre un ensemble de concepts génériques tels que *Atom* (Objet élémentaire), *Model* (qui peut contenir d'autres objets et d'autres structures), *Connection* (relation entre deux objets dans un modèle), *Reference*, *Attribute*, Set (similaire à l'agrégation en UML), *First Class Object* (FCO), *Folders*, *Roles*, *Constraints* et *Aspects* (fournit un contrôle primaire pour la visibilité). Ces concepts sont utilisés pour définir les DSMLs. Le choix des concepts les plus appropriés constitue la décision la plus critique dans la conception. Les *Model*s sont similaires aux classes Java ; ils peuvent être instanciés. Quand un *Model* est créé dans GME, il devient un type (classe). Il peut être sous-typé et instancié autant de fois que l'utilisateur le souhaite [Ledeczi et al. 2001]. Ce concept facilite la réutilisation et la maintenance des modèles parce que tout changement dans un type est automatiquement propagé dans l'arborescence. Cela permet également de créer des librairies de type de modèles qui peuvent êtres utilisées pour différentes applications dans un domaine donné. Pour notre paradigme de la Figure 6.13 nous avons utilisé deux concepts *Model* (*Architecture* et *SoftwareElement*), un *Atom* (*Ports*), une *Connection* (*Link*), un *Aspect* (*Connectivity* pour contrôler la visibilité des éléments de l'Architecture dans l'environnement à créer) et des Attributs.

6.7.2 Spécification de la syntaxe concrète

GME ne sépare pas la syntaxe abstraite de la syntaxe concrète. En ce qui concerne la syntaxe concrète, GME permet de spécifier la représentation graphique à l'aide d'un objet COM appelé *decorator*. Cela permet (avec beaucoup de limites) d'associer des figures et des symboles aux concepts du méta-modèle. GME offre également une interface C++ de plus haut niveau appelée *Builder Object Network*, qui est plus simple à utiliser que les *decorators* COM, mais qui est aussi plus limitée. Il nous a été difficile tout comme pour [Amyot et al. 2006] et [El Kouhen et al. 2012] de personnaliser les figures et les liaisons utilisées. Par exemple, faire apparaître la liste des attributs d'un logiciel à l'intérieur de sa représentation nécessite du code, de même que représenter une liaison par une figure ou personnaliser le type de liaison. De façon générale, la représentation graphique est définie sous forme d'icône (*Bitmap Image*) et il est difficile de créer des figures particulières.

6.7.3 Spécification de la sémantique

GME n'offre pas de support pour l'exécution des modèles, par contre des contraintes OCL peuvent êtres ajoutées au méta-modèle pour augmenter sa précision. GME est l'un des seuls outils de méta-modélisation à permettre la vérification au niveau modèle des contraintes OCL définies dans le méta-modèle.

6.7.4 Génération de l'éditeur

Une fois le méta-modèle créé et les *decorators* définis, le DSML peut être enregistré dans GME comme un nouveau *paradigm* et utilisé comme un outil de modélisation dédié comme le montre la Figure 6.14. L'éditeur généré fournit plusieurs fonctionnalités : chargement/sauvegarde (en XMI), *undo/redo*, *drag/drop* pour la création des éléments du modèle, validation des contraintes OCL et des multiplicités du méta-modèle, impression, zoom, propriétés ... Lorsqu'on modifie le méta-modèle, la rétrocompatibilité est facilement mise en œuvre.

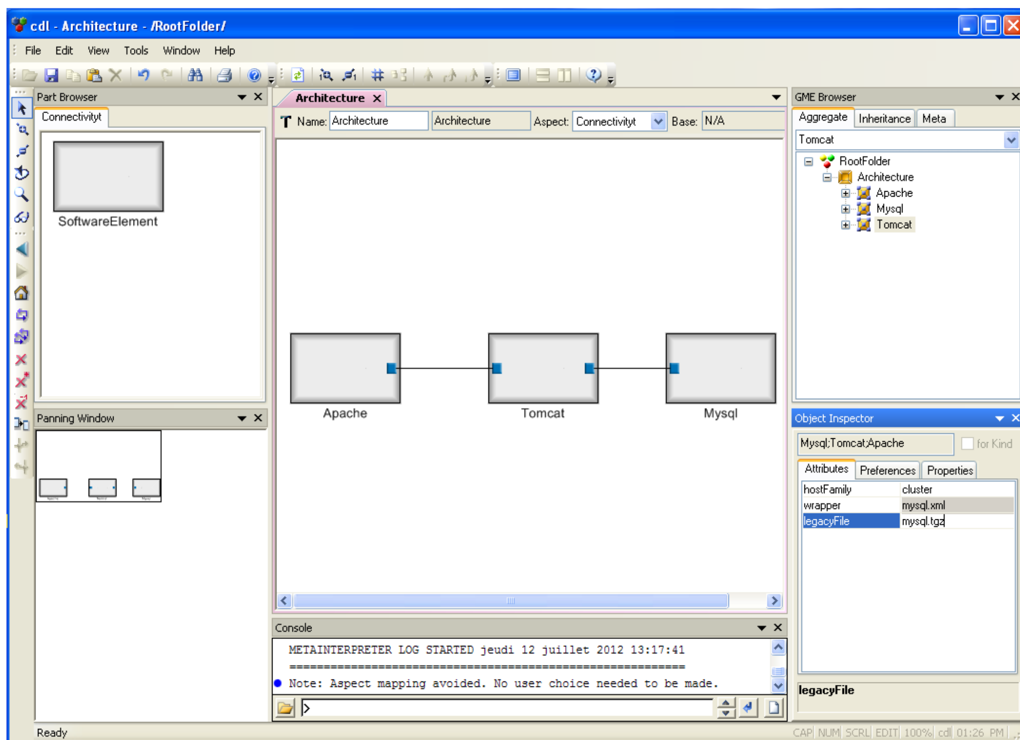


FIGURE 6.14 – Éditeur créé avec GME

6.7.5 Synthèse

GME nécessite beaucoup d'effort en termes d'apprentissage pour comprendre son langage de méta-modélisation et son fonctionnement. Toutefois, par rapport à GMF et VM-SDK le projet est assez bien documenté. GME était à la base un projet dédié au traitement de signal et il est très adapté pour la définition de la notation graphique dans ce domaine. De ce point de vue GME peut donc être considéré comme un outil spécifique simple d'utilisation à l'intérieur de ce domaine. Mais parce qu'il permet également de coder tout type de représentation graphique moyennant des compétences de programmation en C++, il peut également être considéré comme un outil générique complexe d'utilisation, mais flexible.

Syntaxe abstraite - Langage de méta-modélisation	
Standard	non, langage de méta-modélisation propriétaire
Interopérable	non
Complet	oui
Intuitif	non
Syntaxe concrète	
Modularité	non, pas de séparation entre la syntaxe concrète et la syntaxe abstraite
Assistant graphique	non
Sémantique	
Langage de contraintes	OCL
Sémantique dynamique	non
Génération de l'éditeur	
Automatique	oui
Évolutivité	oui
Utilisabilité de l'éditeur généré	grande
Utilisabilité de l'outil	
Documentation	assez bien documenté
Flexibilité	grande [1]
Complexité	Générique, mais complexe d'utilisation

[1] Moyennant une expertise en programmation

TABLE 6.4 – Résumé de GME

6.8 Autre implantation de MIC : MetaEdit+

Dans le domaine des outils de méta-modélisation, MetaEdit+ est l'une des solutions commerciales les plus utilisées. MetaEdit+ est un framework de méta-modélisation créé par la compagnie MetaCase [MetaCase 2012] qui permet de créer des environnements de modélisation dédiés.
Le processus de construction d'un environnement de modélisation dans MetaEdit+ est simple. D'abord, on définit le langage sous forme de méta-modèle et sa syntaxe concrète avec *MetaEdit+ Workbench* et ensuite on utilise l'éditeur produit dans *MetaEdit+ Modeler*.

MetaEdit+ Workbench : Destiné essentiellement aux experts de domaines et aux architectes, cet outil offre un environnement intégré pour la définition et l'implémentation des DSMLs. L'environnement intègre un ensemble d'outils qui répondent aux besoins les plus fréquents en matière de méta-modélisation. On y trouve, entre autres, un éditeur d'objets pour la définition des concepts, un éditeur de propriétés, un éditeur de relations et un éditeur de contraintes. L'environnement offre également un éditeur de symboles qui sert à définir la syntaxe concrète du DSML et un éditeur de générateur pour le dévelop-

pement des générateurs de code. La définition du langage est stockée sous forme d'un méta-modèle dans le référentiel de MetaEdit+ (*MetaEdit+ repository*).

MetaEdit+ Modeler : *MetaEdit+ Modeler* est un outil qui offre un environnement d'utilisation pour les DSMLs définis avec *MetaEdit+ Workbench*. Il permet à partir du DSML extrait du référentiel d'offrir automatiquement toutes les fonctionnalités d'un environnement de modélisation. Les fonctionnalités offertes par cet environnement comportent, entre autres, les éditeurs graphiques, les explorateurs de modèles et les générateurs. L'environnement permet également de vérifier les modèles du DSML et d'établir des liens entre eux.

6.8.1 Spécification de la syntaxe abstraite

Tout comme GME, MetaEdit+ est basé sur un langage de méta-modélisation propriétaire qui est le langage GOPPRR. GOPPRR est une abréviation des types de base que fournit ce langage à savoir *Graph*, *Object*, *Port*, *Property*, *Relationship* et *Role*. Le concept de *Graph* est similaire au concept *Model* de GME et représente un ensemble d'objets avec des relations ayant des rôles spécifiques. Un *Graph* est la structure de plus haut niveau dans un méta-modèle, il définit un langage ou un type de diagrammes tels que le diagramme de classe ou le diagramme d'états transitions. L'*Object* est le concept de base du langage. Les *Object*s sont reliés entre eux par des *Relationship*s, et chaque *Object* impliqué dans une relation a un *Role* particulier. Le GOPPRR présente un avantage intéressant qui est de favoriser la réutilisation et la maintenance des modèles. Toute modification faite dans un *Graph* est propagée dans l'arborescence et un *Graph* peut être réutilisé dans un autre *Graph* ou dans un autre projet. Dans notre exemple représenté à la Figure 6.15, tous les concepts de notre méta-modèles (*SoftwareElement*, *Attributes*, *Dependency*, ...) sont contenus dans le Graph *Architecture*.

6.8.2 Spécification de la syntaxe concrète

MetaEdit+ comme GME ne sépare pas la définition de la syntaxe abstraite et de la syntaxe concrète, l'utilisateur définit tout son langage en GOPPRR et l'enregistre dans le référentiel de MetaEdit+. MetaEdit+ offre un ensemble de représentations graphiques basiques comme les autres outils, mais pour des notations spécifiques l'utilisateur peut utiliser l'éditeur de symboles du *MetaEdit+ Workbench* qui permet de dessiner la représentation graphique souhaitée. Même si l'éditeur de symboles ne permet pas de spécifier toutes les représentations graphiques possibles, il offre tout de même un large panel de possibilités.

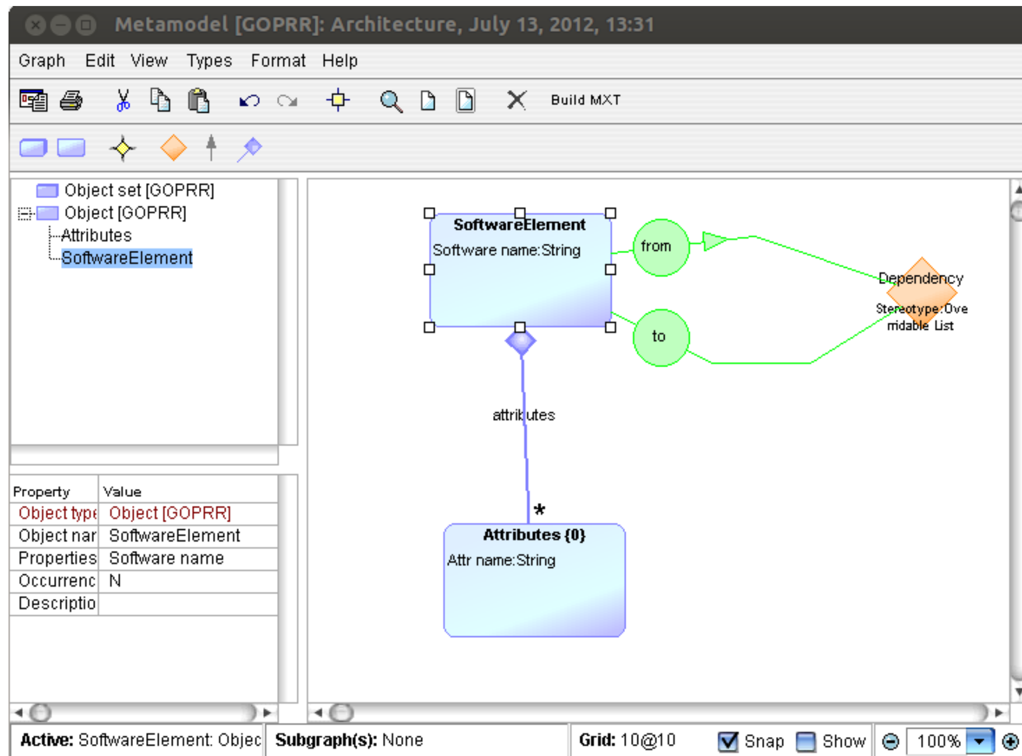


FIGURE 6.15 – Exemple de méta-modèle défini avec GOPPRR

6.8.3 Spécification de la sémantique

MetaEdit+ n'offre pas de langage de contraintes tel que OCL, l'utilisateur peut tout de même exprimer un ensemble très réduit de contraintes non structurelles au niveau du méta-modèle. MetaEdit+ permet par contre la simulation et la transformation de modèles. Il inclut une API qui fournit une interface pour l'écriture de scénarios et la simulation des modèles.

6.8.4 Génération de l'éditeur

Une fois le méta-modèle défini et enregistré dans le référentiel de Metaedit+, il peut être utilisé comme langage de modélisation comme le montre la Figure 6.16. L'environnement de modélisation produit présente plusieurs fonctionnalités comme le chargement/-sauvegarde des modèles dans un format XML adapté, *undo/redo*, impression, exportation des modèles au format bitmap, GIF, PNG et PICT, import/export de la représentation graphique à partir/vers SVG, *zoom*, explorateur de modèles, éditeur de symboles pour dessiner des représentations graphiques, un outil pour la définition des règles et des contraintes .... L'utilisateur peut également choisir de représenter ces modèles sous forme de table, de diagramme ou de matrice sans avoir à régénérer l'environnement. Lorsque le méta-modèle est modifié, les modèles sont immédiatement mis à jour. Pour générer du code à partir des modèles édités, MetaEdit+ propose un éditeur de générateurs interactif per-

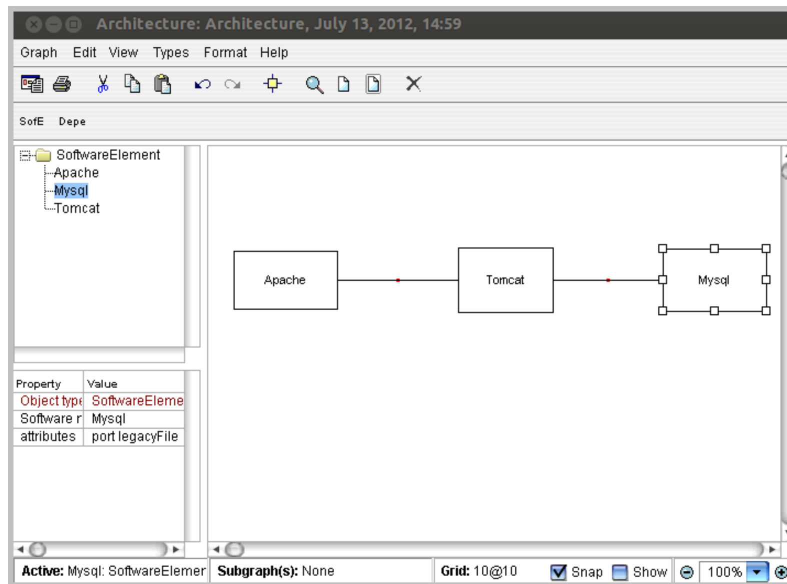


FIGURE 6.16 – Éditeur généré avec MetaEdit+

mettant le développement, le débogage et le test de générateurs. Il permet à l'utilisateur de visualiser, d'éditer ou d'exécuter les générateurs de code disponibles. Ces générateurs sont écrits dans un langage dédié qui permet de manipuler les informations des modèles et de générer le code.

6.8.5 Synthèse

MetaEdit+ simple d'utilisation et différentes études le classe parmi les meilleurs frameworks en termes simplicité d'utilisation. Toutefois, MetaEdit+ nécessite comme les autres outils un temps d'apprentissage qui peut être significatif d'autant plus que le langage GOPPRR n'est pas un standard connu. Cet outil peut être considéré comme étant un outil spécifique d'abord parce que MetaEdit+ utilise un langage de méta-modélisation spécifique. Mais aussi parce que bien qu'il soit simple d'utilisation, il est limité en termes de sémantique statique et de notations graphiques. Si l'éditeur de symboles permet de spécifier un grand nombre de notations graphiques, ces notations sont surtout adaptées dans le cas des diagrammes ayant une structure en graphe.

6.9 Synthèse générale

Les principaux critères de ces outils que nous venons de présenter sont résumés dans le tableau 6.6.

Syntaxe abstraite - Langage de méta-modélisation	
Standard	non, langage propriétaire
Interopérable	non
Complet	oui
Intuitif	non
Syntaxe concrète	
Modularité	non, pas de séparation entre la syntaxe concrète et la syntaxe abstraite
Assistant graphique	oui, éditeur de symboles
Sémantique	
Langage de contraintes	non
Sémantique dynamique	oui, simulation des modèles
Génération de l'éditeur	
Automatique	oui
Évolutivité	oui
Utilisabilité de l'éditeur généré	grande
Utilisabilité de l'outil	
Documentation	assez bien documenté
Flexibilité	grande [1]
Complexité	relativement simple d'utilisation

[1] Moyennant une expertise en programmation

TABLE 6.5 – Résumé de MetaEdit+

Critères	EMP	DiaMeta	VMSDK	GME	MetaEdit+
Assistant graphique	oui, mais très limité	oui, mais très limité	oui, mais très limité	non	oui, éditeur de symboles
Flexibilité	large panel de DSML [1]	large panel de DSML [1]	large panel de DSML [1]	large panel de DSML [1]	large panel de DSML
Complexité	complexe d'utilisation	complexe d'utilisation	complexe d'utilisation	complexe d'utilisation	relativement simple d'utilisation

[1] Moyennant une expertise en programmation

TABLE 6.6 – Vue d'ensemble

On remarque qu'il n'y a pas vraiment d'outil idéal, tous ces outils sont génériques et nécessitent beaucoup d'efforts en programmation. MetaEdit+ apparaît comme étant l'outil qui offre le meilleur rapport qualité/coût. En effet c'est celui qui est le plus abstrait et donc le plus intuitif. Mais l'absence de support pour la sémantique statique et les limites de son éditeur de symbole font qu'il n'est pas possible d'implanter tout type de DSML avec précision. De façon générale, la plupart de ces outils offrent des assistants qui per-

mettent d'automatiser le processus de génération des éditeurs dédiés. Mais ces assistants ne sont efficaces que pour des cas très généralistes et au final ne sont pas satisfaisants dans la plupart des cas. L'utilisateur se retrouve donc très souvent obligé d'implanter les spécificités de son langage en les programmant lui même. Le gain de productivité escompté se trouve ainsi réduit par le surcoût de programmation nécessaire.
Comme le souligne [Evans et al. 2009], les outils de méta-modélisation devraient tenir compte du fait qu'un DSML est une solution métier et non une solution technique. Les solutions existantes requièrent de l'utilisateur une expertise en langage et une expertise en programmation. C'est pourquoi les DSMLs sont aujourd'hui implantés majoritairement par des experts techniques et non par les experts du domaine concerné. Il apparaît donc comme étant contradictoire que l'objectif majeur d'un DSML qui est de fournir des solutions spécifiques pour un domaine particulier ne soit accessible qu'à des programmeurs expérimentés. Tout cela freine l'adoption des DSMLs à large échelle, alors que bien définis un DSML entraîne un gain de productivité considérable.
Dans la partie suivante, nous présentons notre approche qui préconise la spécialisation des outils de méta-modélisation pour les rendre plus accessibles.

Troisième partie

Contributions

Chapitre 7

Orientations générales

Dans les parties précédentes, nous avons montré notre besoin de langages dédiés pour l'administration autonome. Pour implanter ces langages, nous nous sommes tournés vers les techniques éprouvées de l'IDM telles que la méta-modélisation qui permet de spécifier des DSML et de créer des outils pour les instancier. Mais les outils de méta-modélisation existants se sont révélés être génériques et trop complexes d'utilisation. Leur complexité diminue le gain de productivité escompté et freine l'utilisation des DSML à plus large échelle. Une solution serait d'utiliser des outils spécifiques par domaines, ceux-ci étant plus abstraits et plus simples d'utilisation. Mais développer un outil de méta-modélisation spécifique par domaine coûte cher. Dans ce chapitre nous présentons une approche qui permettrait de tirer avantage à la fois du générique et du spécifique pour fournir des outils de méta-modélisation moins complexes.

7.1 Vers la spécialisation des outils de méta-modélisation

7.1.1 Rôle des utilisateurs

Il ressort de notre expérience que les outils de méta-modélisation existants sont génériques et complexes d'utilisation parce qu'ils reposent en majorité sur des API de bas niveau et nécessitent beaucoup de développement. Ils requièrent de l'utilisateur de nombreuses compétences, notamment la connaissance du domaine, des compétences en développement de langages, des compétences en développement d'outils qui peuvent également nécessiter des compétences d'ergonomiste . . . Mais très peu d'utilisateurs ont toutes ces compétences. Dans la pratique l'utilisateur est souvent un ingénieur qui travaille pour une compagnie qui souhaite augmenter sa productivité en utilisant des DSML. Cet ingénieur a généralement de bonnes connaissances de la compagnie et donc du domaine, il maîtrise les différentes terminologies utilisées, les concepts et les règles du domaine qui ont même quelquefois été implantés par lui, mais il n'est pas forcément expert en développement de langages et d'outils. Il lui est donc imposé de jouer plusieurs rôles pour lesquels

il n'a pas forcément les compétences, il mettra alors énormément de temps pour implanter son DSML et le résultat obtenu sera très souvent bien en deçà de celui escompté.

7.1.2 Utilité des assistants

Les DSML étant des solutions métiers et non techniques, il devrait être possible de les implanter sans que l'utilisateur ait besoin de programmer tout l'outillage associé. Les concepteurs de tous les outils de méta-modélisation génériques ont bien conscience de ce problème c'est pourquoi ils fournissent pour la plupart des assistants permettant d'implanter certains types de DSML sans que l'utilisateur ait besoin de les programmer. Mais il est rare que le DSML voulu par l'utilisateur soit complètement pris en charge par l'assistant et l'utilisateur doit donc souvent le programmer lui même.

L'idée de proposer un assistant est judicieuse, mais pour que cette solution soit satisfaisante il faudrait fournir un assistant qui prenne en charge tout type de DSML ce qui est clairement impossible si on veut garder un niveau d'abstraction élevé.

7.1.3 Vers des outils spécifiques

L'assistant intervient généralement au moment de la définition de la syntaxe concrète, mais en fait c'est le processus dans son ensemble qui doit être spécialisé pour être plus productif. C'est-à-dire que de la spécification de la syntaxe abstraite à la génération de l'outillage, l'utilisateur doit manipuler des concepts issus de son domaine. Cela équivaudrait à spécialiser un outil de méta-modélisation en fonction d'un domaine particulier ou en fonction d'un certain type de DSML.

Un outil de méta-modélisation spécifique a l'avantage d'offrir d'emblée les abstractions du domaine ciblé et permet alors de masquer tous les détails d'implémentation des DSML. En élevant ainsi le niveau d'abstraction, cet outil devient plus intuitif et la productivité en est accrue. De plus, un outil de méta-modélisation spécifique intègre déjà les contraintes du domaine auquel il est dédié, contraintes qui seront alors automatiquement incorporées dans les outils de modélisation qu'il va générer. Les solutions crées à l'aide de ces outils de modélisation seront ainsi optimisées. Par exemple, le risque de générer un code erroné sera nettement réduit. De façon générale, les bénéfices escomptés en spécialisant un outil de méta-modélisation sont les mêmes que ceux qui ont fait le succès des langages dédiés. Mais, même si les outils de méta-modélisation présentent de nombreux avantages, ils ont tout de même un inconvénient majeur qui est leur coût de production.

7.1.4 Coût des outils de méta-modélisation spécifiques

Développer un outil de méta-modélisation spécifique à une famille de problèmes ou à un domaine coûte cher et il n'y a que de grandes compagnies qui se le permettent. Ces

grandes compagnies ont suffisamment de ressources humaines et matérielles pour s'offrir un outil de méta-modélisation complètement dédié à leurs besoins internes en langages spécifiques. Ces outils développés en interne ne sont alors disponibles qu'au sein de la compagnie. Mais il n'y a pas que les grosses compagnies qui ont besoin de langages spécifiques et les utilisateurs aux besoins plus modestes sont d'ailleurs les plus nombreux. Pour un utilisateur qui voudrait juste implanter un DSML, la solution de l'outil de méta-modélisation spécifique apparaît comme étant inaccessible parce que trop coûteuse. Ce type d'utilisateur se tournera donc naturellement vers des outils génériques et sera confronté à leur complexité d'utilisation.

En conclusion, les outils génériques bien que flexibles sont trop complexes d'utilisation. La généricité nécessite un faible niveau d'abstraction et qui dit faible niveau d'abstraction dit complexité accrue. Des outils spécifiques offriraient un meilleur rendement, mais développer un outil spécifique par domaine coûte cher. Notre contribution consiste donc à trouver et à implanter un bon compromis entre outils de méta-modélisation génériques, mais complexes et outils de méta-modélisation spécifiques, mais coûteux, l'objectif final étant de permettre l'implémentation des DSML de façon simple.

7.2 Notre approche

Nous sommes arrivés à la conclusion qu'il faut spécialiser les outils de méta-modélisation tout comme l'ont été les langages. Cette spécialisation est nécessaire si on veut automatiser et donc simplifier de façon significative la mise en œuvre des DSML.

L'approche que nous proposons vise à concevoir une plate-forme générique permettant de construire des outils de méta-modélisation spécifiques. Cette approche résulte de deux constats simples :

- Un langage de méta-modélisation est un langage et peut donc être spécialisé pour un domaine ;
- Un outil de méta-modélisation est la syntaxe concrète d'un langage de méta-modélisation, de la même façon qu'un outil de modélisation est la syntaxe concrète d'un DSML.

Partant de ces deux constats on peut déduire qu'un outil de méta-modélisation dédié peut être généré à partir d'un langage de méta-modélisation dédié, de la même façon qu'un outil de modélisation est généré à partir d'un DSML. Il faudrait alors permettre de définir les langages de méta-modélisation en fonction des spécificités des domaines et d'en générer des outils de méta-modélisation spécifiques. Cela permettrait de tirer profit de tous les avantages qui ont fait le succès des DSML, mais cette fois-ci ils sont rapportés aux outils.

Le domaine ciblé par un langage de méta-modélisation pourra être soit un domaine technique dit horizontal tel que la persistance, les interfaces utilisateurs, la communication ou les transactions ou alors un domaine métier dit vertical tel que les télécommunications, les banques, la robotique, les assurances ou le commerce.

Prenons l'exemple du domaine des composants qui est un domaine horizontal. Il serait intéressant pour nous d'avoir un outil de méta-modélisation dédié au domaine des composants. Pour cela il faudrait définir un langage de méta-modélisation qui intégrerait tous les concepts spécifiques au domaine des composants, concepts que nous avons brièvement présentés à la section 6.1. On pourrait alors implanter dans ce langage la syntaxe et la sémantique issue de ce domaine (sens de la relation de composition entre composants, comportement graphique d'un connecteur ...). À partir d'un tel langage, on pourrait générer un outil de méta-modélisation qui permettrait d'implanter des DSML tels que notre ADL ou notre CDL (Cf. section 2.3).

Nous pouvons également envisager un langage de méta-modélisation spécifique à un domaine vertical. Par exemple, un outil de méta-modélisation qui serait dédié à l'événementiel et qui permettrait de définir des DSML pour la gestion de projets. Le langage de méta-modélisation de cet outil fournirait des concepts spécifiques tels que des tâches, des événements ou encore des ressources. Un tel outil de méta-modélisation permettrait par exemple d'implanter des outils de modélisation pour la construction de diagrammes tels que les diagrammes de Gantt ou de Pert généralement utilisés par des responsables de projets.

Pour permettre la définition de tels langages de méta-modélisation spécifiques, nous proposons de rajouter un niveau dans la pile des outils. Cela équivaut à fournir un framework de *méta-méta-modélisation* qui permettrait de définir des langages de méta-modélisations dédiés et de générer à partir de ces langages des outils de méta-modélisation spécifiques. C'est en quelque sorte la fameuse architecture de modélisation à quatre niveaux que nous rapportons ainsi aux outils (Figure 7.1). Mais contrairement à l'architecture des modèles, ici il n'y a pas de relations de conformité entre les différents niveaux d'outil.

Comme le présente la Figure 7.1, au niveau M1 on retrouve les outils de modélisation qui fournissent un support aux langages de modélisation et permettent d'exprimer des modèles. Au niveau M2 nous avons les outils de méta-modélisation tels que GMF, GME, MetaEdit+ ... qui fournissent un support au langage de méta-modélisation et permettent d'exprimer des DSML. À partir de ces DSML sont générés des outils de modélisation spécifiques. Nous proposons de rajouter un niveau M3 dans la hiérarchie de ces outils. Au niveau M3 on aurait un outil de *méta-méta-modélisation* qui permettrait d'exprimer des langages de méta-modélisation ou encore des méta-méta-modèles. À partir de la syntaxe et de la sémantique de ces méta-méta-modèles, on pourrait générer un outil de méta-modélisation spécifique à un domaine particulier. L'outil de méta-modélisation spécifique pourra ainsi facilement évoluer en fonction des changements des contraintes du domaine.

Quand nous écrivons outil de *méta-méta-modélisation* cela ne signifie pas que cet outil devrait nécessairement utiliser des méta-modèles pour définir les langages de méta-modélisation dédiés, mais c'est simplement pour signifier que cet outil se situe au-dessus des outils de méta-modélisation.

L'outil de méta-méta-modélisation doit être générique pour permettre d'exprimer des langages de méta-modélisation pour un maximum de domaines. Il doit donc fournir des

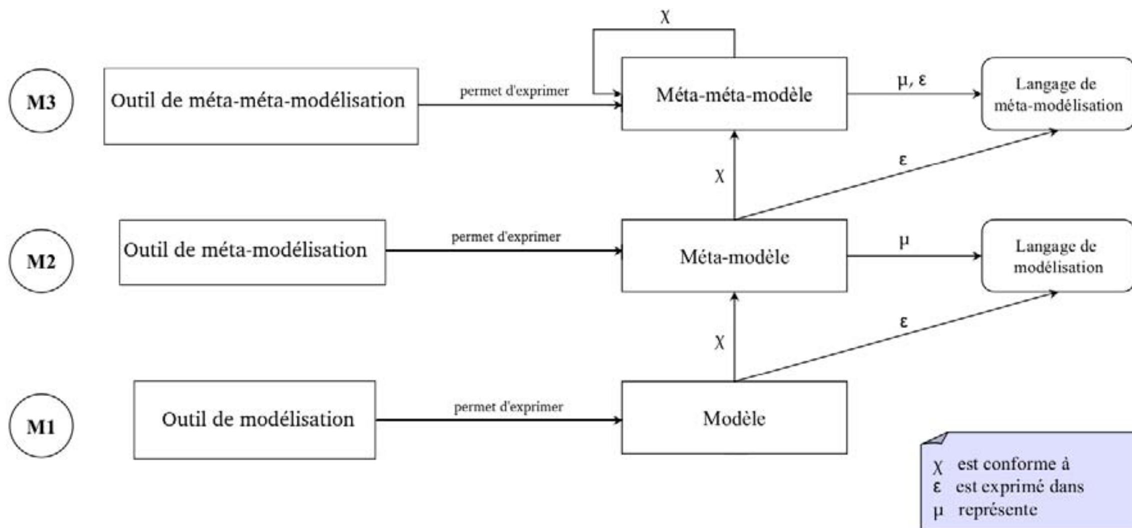


FIGURE 7.1 – Architecture des outils de modélisation

abstractions suffisamment généralistes pour convenir à tous ces domaines et sera de ce fait d'assez bas niveau. En outre, plus on monte dans les différents niveaux d'outils plus la complexité en termes de maintenance et le besoin d'expertise en développement croissent. C'est pourquoi l'outil de méta-méta-modélisation sera destiné aux experts en développement. Ces experts en développement qui sont peu nombreux vont encapsuler toute leur expertise dans la conception des outils de méta-modélisation spécifiques pour faciliter et améliorer le travail des experts du domaine qui implantent les DSML. Le coût de production des outils de modélisation est donc déplacé au niveau M3 où seront capitalisées les compétences des experts en développement. Le travail de ces derniers permettra d'améliorer le rendement au niveau de l'outil de méta-modélisation et simplifiera le travail des experts du domaine. Autrement dit, le travail de quelques un au niveau M3 (experts en développement) permettra aux autres qui sont plus nombreux (experts du domaine) de se concentrer uniquement sur ce qu'ils savent faire.

Cette approche qui est résumée par la figure 7.2 présente un double avantage. Elle permet de tirer profit à la fois du générique et du spécifique. En effet, une plate-forme de méta-méta-modélisation générique va offrir un niveau d'abstraction adéquat à l'expert en développement qui pourra alors implanter des langages de méta-modélisation spécifiques à des domaines particuliers. À partir des langages de méta-modélisation dédiés, des outils de méta-modélisation spécifiques vont être générés pour les experts du domaine qui pourront alors implanter dans les DSML toute leur expertise du domaine et la génération de l'outillage sera plus automatisée. Chaque expert travaillera ainsi au niveau qui lui correspond. Au niveau M3 on bénéficiera donc de tous les avantages du générique pour permettre de faire du spécifique au niveau M2.

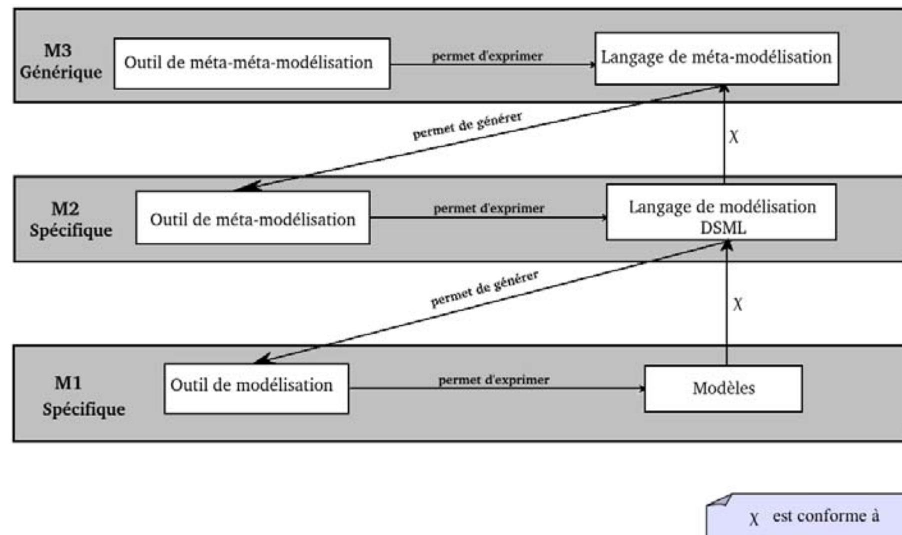


FIGURE 7.2 – Notre approche

7.2.1 Exigences d'un outil de méta-méta-modélisation

Pour qu'un outil de méta-modélisation puisse être généré, il faut spécifier la syntaxe et la sémantique du langage de méta-modélisation qui le définit. Pour éviter de polluer la syntaxe abstraite, il faut qu'elle soit séparée de la syntaxe concrète et de la sémantique ce qui permet également de séparer les préoccupations. L'outil de méta-méta-modélisation doit donc permettre de définir des langages de méta-modélisation de façon modulaire.

En plus de la modularité, un outil de méta-méta-modélisation doit favoriser l'évolutivité et l'adaptabilité des langages de méta-modélisation. En effet pour pouvoir adresser un maximum de domaines, l'outil doit être adaptable et puisque les contraintes d'un domaine peuvent évoluer, il faudrait également que cet outil permette l'évolution des langages de méta-modélisation.

Dans les outils de méta-modélisation génériques, la syntaxe concrète du niveau M1 est définie au niveau M2. Dans notre approche la syntaxe concrète du niveau M1 sera définie au niveau M3. Cela consistera à assigner un comportement graphique par défaut aux outils de modélisation qui seront générés. Ce comportement graphique tiré du domaine ciblé sera commun à tous les outils de modélisation créés pour ce domaine. Toutefois, la syntaxe concrète par défaut doit pouvoir être modifiable en fonction des besoins des utilisateurs.

Enfin, pour réduire les coûts de développement, il faudrait que l'outil de méta-méta-modélisation permette la réutilisation totale ou partielle d'un langage de méta-modélisation déjà défini afin de faciliter la création d'un nouveau.

Chapitre 8

Description de GyTUNe

8.1 Introduction

Nous avons conçu et développé une plate-forme générique de méta-méta-modélisation appelée GyTUNe dont l'objectif est de permettre la création des outils de méta-modélisation spécialisés en fonction des domaines. GyTUNe permet de créer des environnements de méta-modélisation dédiés en fonction des domaines ou en fonction d'une famille de problèmes. Nous présentons dans cette partie les spécifications et les choix d'implantation de cette plate-forme ainsi que son fonctionnement général.

8.2 Choix d'implantation

L'objectif de GyTUNe est de permettre la création d'outils de méta-modélisation dédiés en utilisant des méthodes de développement logicielles qui ne seraient pas trop coûteuses. Les spécifications de GyTUNe sont les suivantes :

- **Généricité** : GyTUNe doit permettre de construire des outils de méta-modélisation pour quasiment tous les domaines et doit donc être une plate-forme générique. Cela implique que son niveau d'abstraction ne doit pas être très élevé pour permettre d'exprimer toutes les spécificités des différents domaines.
- **Modularité** : De même que pour les langages de modélisation, il convient d'utiliser dans GyTUNe des méthodes de développement qui permettent de séparer la syntaxe concrète de la syntaxe abstraite et de la sémantique d'un langage de méta-modélisation. Concevoir GyTUNe de façon modulaire faciliterait cette séparation et permettrait de façon plus générale une meilleure séparation des préoccupations.
- **Adaptabilité** : nous désignons par adaptabilité la capacité d'ajuster un environnement de méta-modélisation précédemment créé à l'aide de GyTUNe afin d'en dériver un second ayant des spécificités différentes. Cela permettrait de réduire les coûts

de production. De plus, un bon langage dédié doit être évolutif afin de s'adapter à l'évolution des contraintes du domaine ciblé. Cela implique d'utiliser des formalismes promouvant l'adaptabilité.
– **ré-utilisabilité** : De même que l'exigence d'adaptabilité, l'exigence de ré-utilisabilité a pour objectif de réduire les coûts de développements nécessaires à la création d'un environnement de méta-modélisation dédié. GyTUNe doit fournir des formalismes qui favorisent la réutilisation afin que tout ou partie d'un langage déjà défini puisse être réutilisé afin d'en créer un nouveau.

Nous avons trouvé dans notre contexte une approche de développement qui satisfait bien à toutes ces exigences, il s'agit de la programmation basée sur les composants. C'est au même titre que la programmation orientée objet un paradigme de programmation pour la conception et l'implémentation de systèmes logiciels. Elle se situe dans la continuité de la programmation orientée objet et offre une meilleure structuration, non plus au sein du code, mais au niveau de l'architecture générale du logiciel. Cette approche de développement vise à fournir des briques logicielles de base configurables et adaptables pour permettre la construction d'une application par composition. Ces briques qui sont appelées composants sont reliées entre elles par des *liaisons* pour former une architecture logicielle. Un composant constitue donc une unité de structuration, qui remplit une fonction spécifique et peut être assemblée avec d'autres composants. Cet assemblage repose sur un modèle à composants qui fournit des services non fonctionnels permettant aux composants de passer des contrats entre eux. Ces contrats consistent à définir de façon explicite des services fournis et des services requis par chaque composant.
Les composants logiciels présentent plusieurs caractéristiques qui répondent aux spécifications que nous avons pour notre framework de méta-méta-modélisation :

– **Modularité** : un composant est une unité de structuration autonome. Contrairement à l'approche objet, les dépendances entre les composants sont vraiment réduites et se limitent aux services requis par les différents composants. Un composant peut alors être déployé ou exécuté indépendamment des autres. Pour satisfaire à notre exigence de modularité, il suffirait alors de décomposer notre framework en plusieurs composants ayant une granularité adéquate pour permettre une meilleure séparation des préoccupations. De plus, on peut séparer la syntaxe concrète d'un langage de sa syntaxe abstraite et de sa sémantique en les définissant dans des composants distincts.
– **Adaptabilité** : les modèles à composant permettent explicitement de remplacer un composant par un autre ou de changer les liaisons entre les composants. Il serait donc par exemple possible de remplacer le composant qui encapsule la syntaxe concrète d'un langage de méta-modélisation par un autre et de générer ensuite un environnement de méta-modélisation différent.
– **Ré-utilisabilité** : un composant est réutilisable, il définit tous les services qu'il offre et ceux qu'il requière de façon contractuelle. Il propose ses services et ses règles de fonctionnement de telle sorte qu'ils soit ré-utilisables dans un assemblage différent.

Nous avons donc choisi d'implanter notre framework de méta-méta-modélisation à l'aide de composants parce que cette méthode de programmation satisfait bien nos exi-

gences. De plus, nous avons déjà de nombreuses expériences autour de la programmation par composant, expériences qui nous ont d'ailleurs amenés à choisir le modèle à composant fractal pour la mise en œuvre de notre approche.

8.2.1 Modèle à composant Fractal

Fractal [Bruneton et al. 2006] est un modèle de composants conçu conjointement par France Telecom R&D et l'INRIA. C'est une spécification de modèle de composant récursif et réflexif. C'est-à-dire qu'un composant peut être composé de sous-composants et qu'un composant peut s'inspecter et se modifier (cela permet de contrôler les composants au déploiement et à l'exécution). De plus, ce modèle est hautement flexible et extensible. Flexible parce que tout est optionnel, et extensible parce que l'on peut personnaliser les capacités de contrôle de chacun des composants.
Ce modèle est destiné à la construction de systèmes logiciels adaptables. Il possède une spécification [Merle and Stefani 2008] et de multiples implémentations dont Julia son implémentation en Java que nous utilisons et qui est celle de référence. Le modèle définit la notion de composants partagés, facilitant ainsi la représentation de ressources.
Le modèle Fractal impose peu de restrictions et peut être utilisé selon divers niveaux de conformité. Il n'est pas lié à un langage de programmation.

Éléments du modèle

Composant :
Comme représenté sur la Figure 8.1, un composant Fractal possède un contenu et une membrane. Le contenu implémente les fonctionnalités offertes par le composant et la membrane implémente ses capacités de contrôle, permettant, par exemple, d'administrer celui-ci. Un composant qui ne contient aucun autre composant est dit primitif ; dans le cas contraire, il est appelé composite (par exemple le composant ClientServeur de la Figure 8.2(a)). Un composant peut être partagé, en appartenant à plusieurs composites à la fois.
Un composant possède un type, permettant notamment de vérifier que deux composants peuvent êtres reliés. Le système de type proposé par la spécification (celui-ci pouvant être modifié) définit le type d'un composant à partir de l'ensemble des types des interfaces métier de ce composant.

Interface :
Une interface est un point d'accès au composant. Un composant Fractal peut posséder des interfaces fonctionnelles (dites métiers) et des interfaces non fonctionnelles (dites de contrôle). Le système de type, proposé dans la spécification Fractal, définit le type d'une interface à partir d'une signature, correspondant au nom du type d'une interface langage. Les interfaces sont dites de type serveur lorsqu'elles décrivent un service fourni et de type

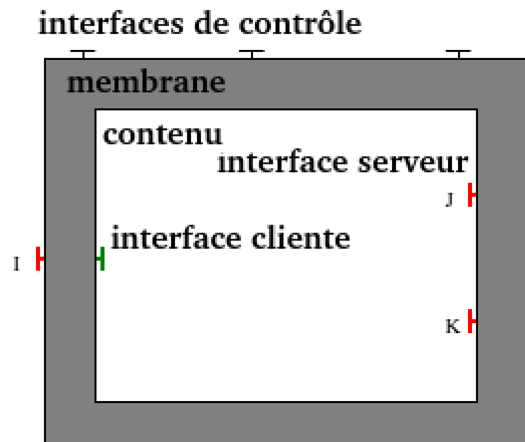


FIGURE 8.1 – Interfaces internes d'un composant composite Fractal.

client lorsqu'elles décrivent un service requis. Fonctionnalités et capacités de contrôle sont ainsi décrites par des interfaces de type serveur.

Liaison :
Les composants sont liés les uns aux autres via leurs interfaces fonctionnelles. Fractal est un modèle de composant typé, c'est-à-dire qu'une interface serveur doit être du même type que l'interface cliente qui se lie à elle. Ces connexions entre interfaces sont nommées *binding*.
Un composant Fractal a un cycle de vie. C'est-à-dire qu'un composant a un état d'exécution. Il peut être dans l'état *stopped* (arrêté) ou *started* (en marche). Une application doit avoir tous ses composants en état de marche pour pouvoir fonctionner.

Contrôle des composants

La spécification Fractal définit six contrôleurs, pouvant être intégrés, ou non, à la membrane du composant.

Contrôle des attributs : Le contrôleur d'attributs donne accès en lecture et en écriture aux propriétés configurables d'un composant. Un composant peut fournir une interface ***AttributeController*** pour fournir aux autres composants les méthodes *get* et *set* de ses attributs.

Contrôle des liaisons : Une interface de type client peut être liée à l'interface serveur d'un autre composant à l'aide du contrôleur de liaison appelé ***BindingController***. Ce dernier permet également de suivre et détruire une liaison.

Contrôle de contenu : un composant composite fournit l'interface ***ContentController***, pour lister, ajouter ou supprimer des sous-composants.

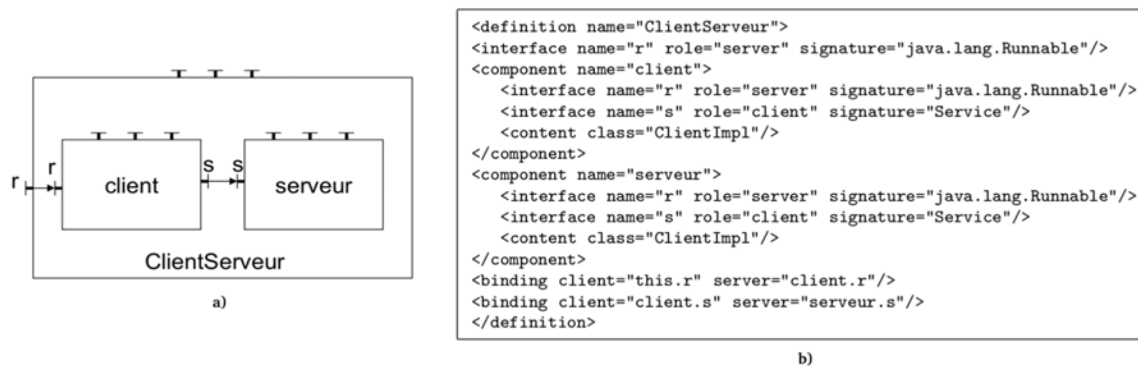
Contrôle de cycle de vie : un composant fournit l'interface ***LifeCycleController*** pour permettre le contrôle explicite du cycle de vie pour faciliter les opérations de reconfiguration. Ce contrôleur de cycle de vie permet de changer l'état d'exécution d'un composant. La spécification définit un cycle de vie de base à deux états, correspondant soit à un composant arrêté, soit à un composant démarré. Ce cycle de vie peut être librement étendu.

Contrôle de nom : L'interface ***NameController*** permet d'associer un nom à un composant.

Contrôle du composant super : L'interface ***SuperController*** permet de connaître les composants composites auxquels appartient un composant.

Fractal ADL

Fractal fournit un langage de description d'architectures basé sur XML. Les balises de base sont les balises <interface>, <component> et <binding> mais le langage est extensible et d'autres balises peuvent êtres définies. La figure 8.2(a) présente la définition ADL du composant de la figure 8.2(b). Nous distinguons un composant composite *ClientServeur* et deux sous-composants client et serveur. L'interface cliente du composant *client* est explicitement liée à l'interface serveur s du composant *serveur*.




```
<definition name="ClientServeur">
<interface name="r" role="server" signature="java.lang.Runnable"/>
<component name="client">
   <interface name="r" role="server" signature="java.lang.Runnable"/>
   <interface name="s" role="client" signature="Service"/>
   <content class="ClientImpl"/>
</component>
<component name="serveur">
   <interface name="r" role="server" signature="java.lang.Runnable"/>
   <interface name="s" role="client" signature="Service"/>
   <content class="ClientImpl"/>
</component>
<binding client="this.r" server="client.r"/>
<binding client="client.s" server="serveur.s"/>
</definition>
```

FIGURE 8.2 – (a) Composant *ClientServeur*. (b) Description en ADL

Julia, une implémentation de Fractal

Julia est une implémentation en Java du modèle de composant Fractal. Dans ce modèle les composants sont constitués d'un contenu et d'une membrane. Le contenu est constitué soit d'un ensemble d'objets Java (composant primitif), soit de sous-composants (composant composite). La membrane renferme un ensemble de contrôleurs et d'intercepteurs. Un intercepteur permet d'intercepter les appels entrants et sortants sur les interfaces fonctionnelles. Les contrôleurs/intercepteurs permettent au composant de s'inspecter (introspection), de se modifier (intercession), d'intercepter les appels de méthodes métier, entrants ou sortants, et de gérer des propriétés non fonctionnelles.

La Figure 8.3(b) correspond à la structure dans Julia du composant Fractal de la Figure 8.3(a). Dans Julia un composant est représenté par différents objets Java. On peut séparer ces objets en 3 groupes distincts :

- Les objets qui implantent les interfaces du composant (Interfaces de contrôle et interfaces fonctionnelles sur la Figure 8.3(b)). Il y a un objet par interface du composant ;
- Les objets qui implantent les contrôleurs du composant (Contrôleurs et intercepteurs sur la Figure 8.3(b)). Un objet contrôleur peut implanter aucune, une ou plusieurs interfaces de contrôle ;
- Les objets qui implantent le contenu du composant dans le cas d'un composant primitif. (Pas présents sur la Figure 8.3(b)).

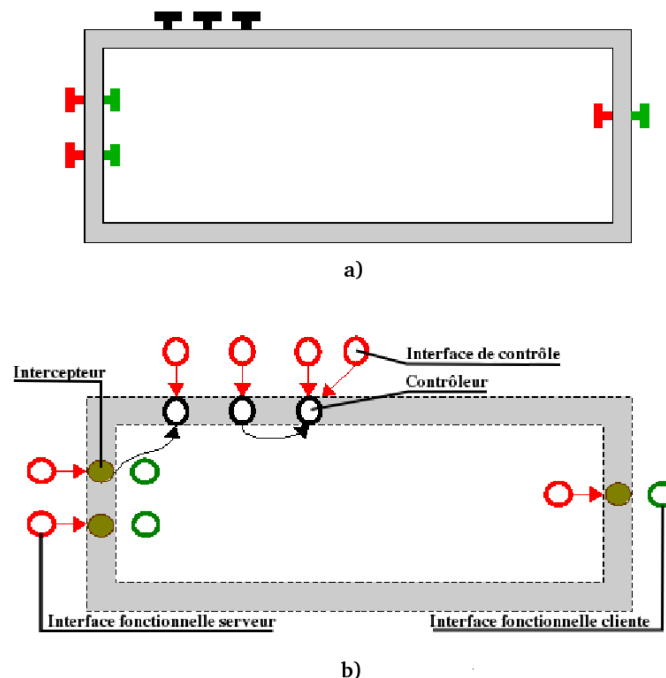


FIGURE 8.3 – (a) Modèle. (b) Implémentation en Julia.

8.3 Conception de GyTUNe

Notre objectif est de créer une plate-forme générique permettant de créer des outils de méta-modélisation spécifiques et adaptables. Pour cela nous voulons permettre aux experts en développement d'implanter des langages de méta-modélisation dédiés que nous utiliserons pour générer des environnements de méta-modélisation dédiés. Cette idée implique alors plusieurs choix de conception que nous détaillons dans cette section.

8.3.1 Séparation entre fonctionnalités génériques et fonctionnalités spécifiques

Notre plate-forme pour être générique ne doit être liée à aucun domaine en particulier. C'est pourquoi il faut la scinder en deux parties. Une partie spécifique aux domaines où les experts en développement implanteront toutes les spécificités escomptées dans les outils de méta-modélisation qu'ils veulent créer ; une deuxième partie dé corrélée de tout domaine qui doit fournir toutes les fonctionnalités génériques qui seront communes à tous les environnements de méta-modélisation créés par les experts en développement. Pour garantir la généricité de notre plate-forme, il est donc important que ces deux parties soient bien séparées.

8.3.2 Définition des concepts au niveau M3

Au niveau M3, quand on définit un langage de méta-modélisation il faut énumérer l'ensemble des concepts de ce langage. Ces concepts pourront être utilisés au niveau M2 pour construire des méta-modèles. Certains concepts utilisés dans un méta-modèle au niveau M2 auront une représentation graphique pour le niveau M1 et d'autres non. Reprenons notre exemple du domaine des composants que nous restreignons à trois concepts à savoir composant, connecteur et un troisième concept que nous appellerons collage. Le concept collage permet de relier un connecteur et un composant au niveau M2. Dans cet exemple, il faudrait définir au niveau M3 les concepts *composant*, *collage* et *connecteur*. Cela est représenté à la Figure 8.4 par la première ligne (niveau M3). Ces concepts sont présents au niveau M2 où ils permettent de construire des méta-modèles Au niveau M2 de notre Figure 8.4 nous avons par exemple construit un méta-modèle avec une instance de *composant* (*unComposant*) qui est liée à une instance de *connecteur* (*unConnecteur*) à l'aide d'une instance de *collage* (*unCollage*). *unComposant* a une représentation graphique pour le niveau M1 qui est un carré. De même, *unConnecteur* a une représentation graphique pour le niveau M1 qui est un "T" renversé (⊣). Mais, *unCollage* n'a pas de représentation graphique pour le niveau M1, il permettra simplement de coller *unConnecteur* sur *unComposant* comme on peut le voir au niveau M1 de la Figure 8.4.

Quand on définit un langage de méta-modélisation (au niveau M3), on spécifie non seulement la liste des concepts, mais aussi leur syntaxe concrète. La syntaxe concrète définit comment ces concepts sont représentés au niveau M2. Dans tous les outils de méta-modélisation existants, la représentation graphique du niveau M1 est définie au niveau M2. L'idée de faire des outils de méta-modélisation spécifiques revient à définir la représentation graphique du niveau M1 (complète ou dans sa plus grande partie) lorsqu'on définit le langage de méta-modélisation c'est-à-dire au niveau M3. Ainsi, la création d'un outil de modélisation est simplifiée puisqu'on ne doit plus définir la syntaxe concrète du niveau M1 au niveau M2, car cela a été fait lors de la construction de l'outil de méta-modélisation (au niveau M3).

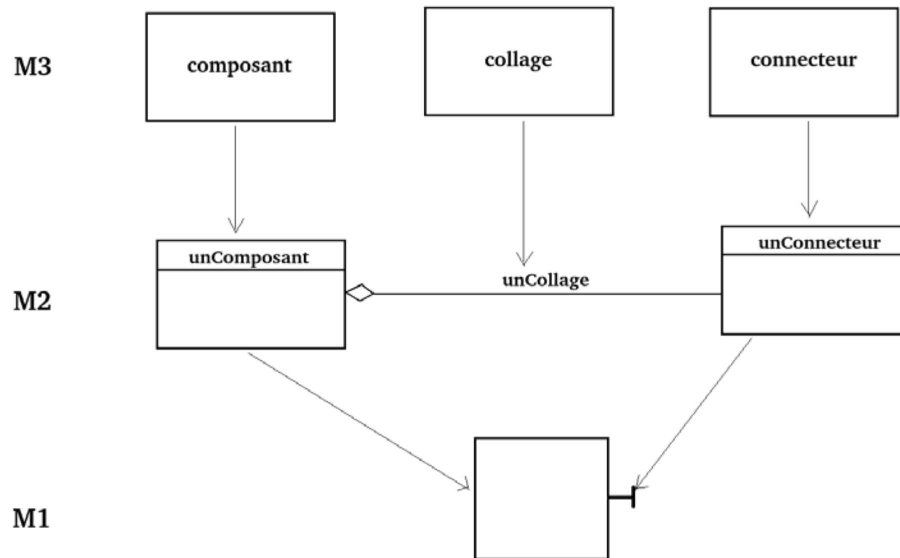


FIGURE 8.4 – domaine des composants simplifié

Il faut donc retenir que lorsqu'on définit un concept au niveau M3, on donne :
- sa syntaxe concrète pour le niveau M2 ;
- sa syntaxe concrète pour le niveau M1 s'il doit y être représenté.

8.4 Description de l'architecture de GyTUNe

GyTUNe repose sur le modèle à composant Fractal et est subdivisé en deux grandes parties. La première partie qui est générique fournit un ensemble de composants qui implantent des fonctionnalités nécessaires à tous les environnements de méta-modélisation telles que la gestion des projets ou encore la gestion des différentes zones d'édition de modèles (que nous appelons éditeurs) . . . Ces composants requièrent des services leur permettant d'implanter dans l'environnement de méta-modélisation toutes les spécificités liées au domaine ciblé. Ces services requis sont implantés par l'utilisateur dans la deuxième partie qui est spécifique à un domaine. L'utilisateur implante ces services sous forme de composants définis suivant des types précis imposés par GyTUNe. Les services requis consistent essentiellement en trois points qui sont : connaître quels sont les concepts du domaine à manipuler dans l'environnement de méta-modélisation, quelles représentations leur assigner et enfin savoir comment ils doivent être manipulés (leur comportement) dans l'environnement. La partie générique se sert alors de la partie spécifique pour générer un environnement de méta-modélisation dédié à un domaine.
L'architecture de GyTUNe représentée à la Figure 8.5 donne une vue d'ensemble des différents composants qui interviennent dans la construction d'un environnement de méta-modélisation dédié. Nous détaillerons dans la suite le rôle de chacun de ces composants.

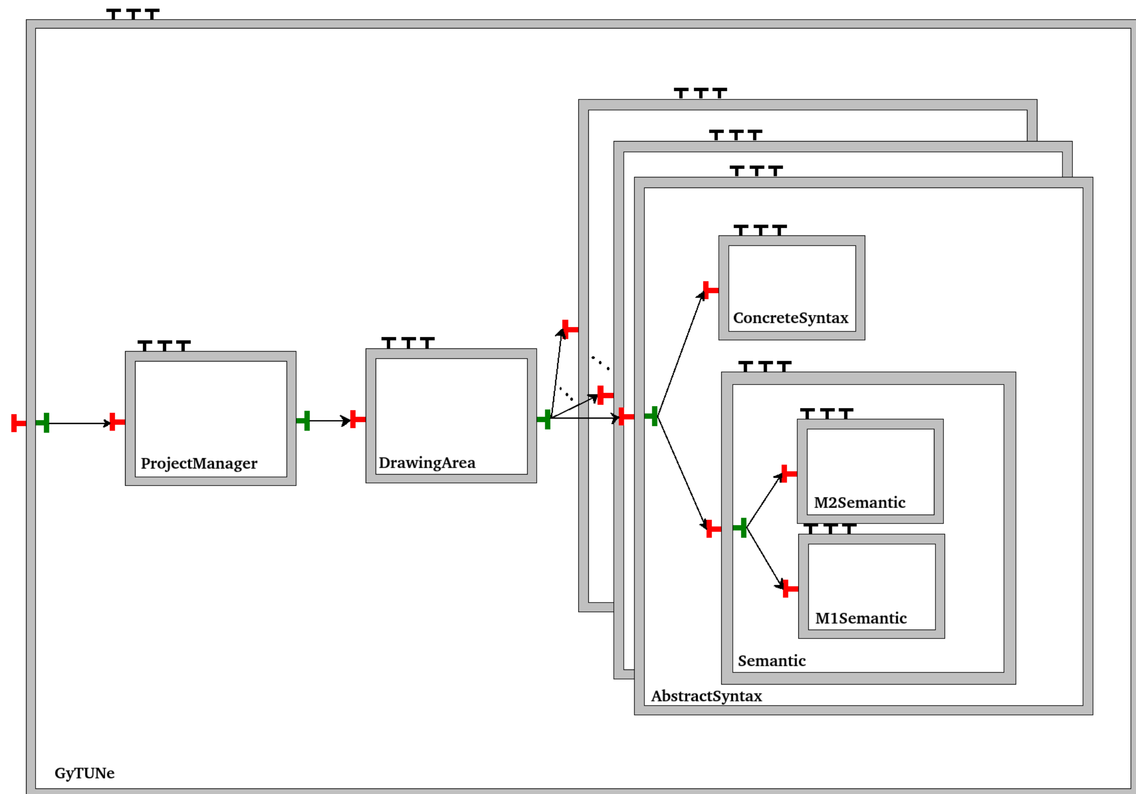


FIGURE 8.5 – Architecture de GyTUNe

La partie générique nous l'avons appelé ***Common*** et la partie spécifique nous l'appelons ***GyTUNe specific***. Nous allons présenter les différents services fournis par chacune de ces parties.

8.4.1 *Common*

C'est dans cette partie que nous implantons toutes les fonctionnalités qui seront communes aux différents environnements de méta-modélisation (gestions des projets, sauvegarde, sélection d'un élément dans un éditeur, déplacement d'un élément . . .). Ces fonctionnalités sont implantées dans les composants *ProjectManager* et *DrawingArea*. ***Common*** contient également la définition de tous les types de composants que l'utilisateur utilisera dans la partie spécifique, l'implémentation de toutes les interfaces de contrôle ainsi que des interfaces métiers que devront implémenter les composants définis dans la partie spécifiques.

ProjectManager

Le composant *ProjectManager* est le composant directement lié au composant principal GyTUNe qui permettra de démarrer l'application GyTUNe (Cf. Figure 8.5). C'est un

composant primitif de type *ProjectManagerType* (Figure 8.6) qui fournit une interface *r* de type *runnable* permettant de démarrer l'application. Le *projectManager* est chargé de tout ce qui est lié à la gestion des projets dans un environnement de méta-modélisation ainsi que de la création de tout l'environnement graphique (fenêtres, menus ...) autre que les zones d'édition des DSLs ou des modèles. Le *projectManager* va notamment implanter les fonctions de gestions de projets telles que la création d'un nouveau projet, sauvegarde/ouverture d'un projet, suppression de projets ...

```
<definition name="gytune.ProjectManagerType">
  <interface name="r" signature="java.lang.Runnable"
role="server"/>
  <interface name="d" signature="gytune.EditorItf" role="client"
cardinality="collection"/>
</definition>
```

FIGURE 8.6 – ProjectManagerType

DrawingArea

Le *DrawingArea* fait le lien entre la partie générique et la partie spécifique. Il implante toutes les fonctionnalités génériques des éditeurs de méta-modèles (niveau M2) et de modèles (niveau M1), et il requiert des services de la partie *GyTUNe specific* pour mettre en œuvre tout ce qui doit être spécialisé dans l'environnement de méta-modélisation. C'est un composant de type *DrawingAreaType* (Figure 8.7) qui permet de gérer les éditeurs de méta-modèles et de modèles. Un éditeur est la partie de l'environnement où l'on peut effectivement créer un DSL (éditeur de DSLs) ou créer un modèle (éditeur de modèles). Le *DrawingArea* fournit une interface *EditorItf* qui définit toutes les méthodes nécessaires à la gestion des éditeurs. Cette interface qui est représentée à la Figure 8.8 définit par exemple les méthodes *getEditor*, *findEditor* et *removeEditor* qui sont implantées dans le *DrawingArea* pour la création, la recherche et la suppression d'un éditeur. Le *DrawingArea* implante également plusieurs fonctionnalités graphiques nécessaires à un environnement d'édition de méta-modèles ou de modèles. Notamment la sélection d'un élément lorsqu'un l'utilisateur clique sur cet élément, le déplacement d'un élément dans la zone de dessin, la création d'une table de propriétés pour chaque élément ... Enfin, le *DrawingArea* implante dans chaque éditeur de méta-modèle, un éditeur de dessins permettant d'assigner une représentation graphique à un élément d'un méta-modèle. Représentation qui sera utilisée au niveau modèle si aucune représentation par défaut n'a pas été fournie (au niveau M3) par l'expert en développement pour le niveau M1.

```
<definition name="gytune.DrawingAreaType">
  <interface name="d" signature="gytune.EditorItf" role="server"/>
  <interface name="concept" signature="gytune.Concept"
cardinality="collection" role="client" />
</definition>
```

FIGURE 8.7 – DrawingAreaType

```
public interface EditorItf
{
    public JPanel getEditor();
    public ArrayList<Editor> getEditors();
    public void setEditors(ArrayList<Editor> editors);
    public Editor getEditor(String editorName);
    public ModelEditor getModelEditor(String name, Editor edsl);
    public JPanel getModelEditor(Editor edsl);
    public Editor getCurrentEditor();
    public void setCurrentEditor(Editor editor);
    public ModelEditor getCurrentModelEditor();
    public void setCurrentModelEditor(ModelEditor editor);
    public ArrayList<ModelEditor> getModelEditors();
    public void setModelEditors(ArrayList<ModelEditor> modelEditors);
    public Editor findEditor(String editorName);
    public ModelEditor findModelEditor(String modelEditorName);
    public String getName();
    public void setName(String name);
    public gytune.Element getElementAt(int x, int y);
    public void addElement(gytune.Element e);
    public void removeElement(gytune.Element e);
    public gytune.Element getModelElementAt(int x, int y);
    public void setCurrentEditorByName(String currentEditorName);
    public void notifyChange();
    public void removeEditor(Editor e);

}
```

FIGURE 8.8 – editor-itf

Nous pouvons résumer le rôle du *ProjectManager* et celui du *DrawingArea* ainsi : le *ProjectManager* construit un environnement permettant de gérer des projets qui font intervenir des éditeurs, ces éditeurs sont implantés par le composant *DrawingArea*, et le contenu de ces éditeurs ou encore l'ensemble des opérations spécifiques qu'il est possible d'effectuer dans ces éditeurs est défini par l'utilisateur dans la partie spécifique.

Définition des types de composants

Dans la partie *Common*, nous définissons les types des composants que l'expert en développement qui intervient au niveau M3 pourra créer. Concrètement un type correspond à un ensemble d'interfaces métiers que chaque composant de ce type doit posséder. Cela permet de savoir exactement quels sont les services fournis et requis par un composant. Les types de composants et les interfaces métiers qu'ils définissent sont liés à une partie précise d'un langage dédié.

Le type ***AbstractSyntaxType*** représenté à la Figure 8.9.

- **Rôle** : permet de définir des composants qui représentent les concepts du langage de méta-modélisation. L'ensemble de ces concepts constitue la syntaxe abstraite ;
- **Principale interface** : chaque composant de type *AbstractSyntaxType* fournit une interface serveur appelée *Concept* qui permet de le connecter au composant *DrawingArea*. C'est par cette interface que le *DrawingArea* accède aux différents concepts du langage et construit la palette de l'éditeur de méta-modélisation dédié correspondant.

```
<definition name="gytune.AbstractSyntaxType">
  <interface name="concept" signature="gytune.Concept"
role="server" cardinality="collection"/>
</definition>
```

FIGURE 8.9 – AbstractSyntaxType

Le type ***ConcreteSyntaxType***

- **Rôle** : permet de définir les composants qui implantent la syntaxe concrète du langage ;
- **Principale interface** : tous les composants de ce type fournissent une interface appelée *ConcreteItf* ;
- **Quelques méthodes** : l'interface *ConcreItf* est représentée à la Figure 8.10. Les méthodes définies dans cette interface permettent d'implanter une représentation et un comportement graphiques pour un concept. Premièrement, un concept peut être un objet ou une liaison (au sens UML). La méthode *isLink()* qui renvoie un booléen permet de savoir si un concept est une classe ou une liaison. Dans le cas où c'est une classe, c'est la méthode *getRepresentation()* qui sera appelée par le *DrawingArea* pour obtenir la représentation de ce concept au niveau méta-modèle. Cette méthode ne prend pas de paramètres et renvoi un *JPanel* que le *DrawingArea* pourra afficher et manipuler dans sa zone dessin. Dans le cas où le concept est une liaison, c'est la méthode *getLinkRepresentation* qui sera appelée par le *DrawingArea* pour obtenir la représentation du concept au niveau méta-modèle. Cette méthode prend en paramètre les coordonnées du point de départ de la liaison à dessiner, les coordonnées du point d'arrivée ainsi qu'une référence à l'objet *Graphics* utilisé par le *DrawingArea*. Référence que l'utilisateur peut alors utiliser pour dessiner sa liaison comme il le souhaite. Les méthodes *isModelLink()*, *getModelRepresentation()*, *getModelLinkRepresentation* font respectivement la même chose que *isLink()*, *getRepresentation()* et *getLinkRepresentation* mais pour le niveau M1. Si la méthode *getModelRepresentation()* n'est pas implantée dans la partie GyTUNe specific, l'utilisateur qui définit un méta-modèle pourra se servir de l'éditeur de dessins fourni par le *DrawingArea* pour assigner une représentation graphique pour le niveau M1. On voit ainsi que la syntaxe concrète d'un concept définit les représentations graphiques aux niveaux M2 et M1.

Le type ***SemanticType***

- **Rôle** : permet de définir les composants qui implantent la sémantique du langage ;
- **Principale interface** : Ces composants fournissent une interface appelée *SemanticItf* qui définit toutes les méthodes permettant de valider la création, la modification ou la suppression d'un élément dans un éditeur ;
- **Quelques méthodes** : l'interface *SemanticItf* est représentée à la Figure 8.11. Cette interface définit par exemple une méthode *isConnectable(Element from, Element to)* qui est appelée par le *DrawingArea* dans le cas où le concept qu'il manipule est une

```
public interface ConcreteItf
{
public JPanel getRepresentation();
public JDrawPanel getModelRepresentation(Element e);
public void getLinkRepresentation(int x, int y, int cx, int cy, Graphics g);
public void getModelLinkRepresentation(int x, int y, int cx, int cy, Graphics g);
public int getHeight();
public void setHeight(int height);
public int getWidth();
public void setWidth(int width);
public boolean isLink();
public boolean isModelLink();
public boolean isInChargeOfModelComputeClosestBorder();
public void computeModelClosestBorder(MyLink l);
public boolean drawWhileMoving();
public Vector<Attribut> getProperties();
public Vector<Attribut> getModelProperties();
public Hashtable<String, String[]> getPropertiesToDisplay();
}
```

FIGURE 8.10 – ConcreteItf

liaison. Cette méthode permet d'autoriser ou non cette liaison entre deux éléments *from* et *to*. L'interface *SemanticItf* définit également les méthodes *validateCreation*, *validateModification*, *validateRemove* qui sont appelées par le *DrawingArea* lorsqu'un concept est créé, modifié ou supprimé. Elles permettent de vérifier les contraintes liées à ce concept.

```
public interface SemanticItf
{
public boolean isConnectable(Element from, Element to);
public boolean isConnectable(gytune.Element myself, gytune.Element from,
gytune.Element to, Vector<gytune.Element> dslElements);
public boolean validateCreation(Element myself, Vector<Element> elements);
public boolean validateRemove(Element myself, Attribut attribut,
Vector<Element> elements);
public Object[] validateModification(Vector<Attribut> attributs,
Hashtable<String, String []> propertiesToDisplay);
public boolean validateRemove();
public OurButton makeTool(Element myself);
}
```

FIGURE 8.11 – SemanticItf

Toutes les interfaces métiers (*EditorItf*, *Concept*, *ConcreteItf*, *SemanticItf*) sont définies dans la partie *Common* de GyTUNe. Elles définissent un panel de méthodes qui représentent autant de possibilités d'adaptation de l'environnement de méta-modélisation. Le cas échéant, l'utilisateur peut rajouter des méthodes dans une interface et utiliser ces méthodes au niveau du *DrawingArea*.

Gestion des propriétés

Dans un modèle comme dans un méta-modèle, on a besoin de gérer les propriétés des concepts qu'on manipule. Ce sont des attributs à qui on peut attribuer des valeurs et qui

correspondent à des caractéristiques spécifiques. Le *DrawingArea* fournit un support pour éditer ces propriétés, mais également pour en rajouter. Ces propriétés peuvent être définies au niveau M3. Mais elles peuvent également être définies au niveau M2 pour les éléments des méta-modèles.

8.4.2 *GyTUNe specific*

C'est dans cette partie que l'utilisateur implante pour chacun des concepts de son langage, les méthodes définies dans *ConcreteItf* et *SemanticItf*. *GyTUNe specific* a été décomposé de telle sorte que la syntaxe abstraite, la syntaxe concrète et la sémantique du langage de méta-modélisation soient définies dans des composants distincts :

- ***AbstractSyntax*** : pour chaque concept du langage, l'utilisateur fournit un composant *AbstractSyntax*. Les composants *AbstractSyntax* sont de type *AbstractSyntaxType*. L'ensemble des composants *AbstractSyntax* constitue la syntaxe abstraite du langage. Un composant *AbstractSyntax* est un composant composite qui contient deux autres composants, *ConcreteSyntax* et *Semantic* qui décrivent respectivement la syntaxe concrète du langage et sa sémantique. Si nous reprenons l'exemple du langage de méta-modélisation dédié au domaine des composants, on aura un composant *AbstractSyntax* pour le concept *Composant*, un deuxième pour le concept *Connecteur*, un troisième pour la relation de *Composition*, un autre pour le concept de *Liaison* ...

- ***ConcreteSyntax*** : à chaque composant *abstractSyntax* est associé un composant primitif *ConcreteSyntax*. Les composants *ConcreteSyntax* sont de type *ConcreteSyntaxType* et ils implantent la représentation graphique au niveau de l'éditeur de méta-modèles (niveau M2). La représentation graphique du niveau M1 y est définie pour les concepts qui seront présents au niveau M1. C'est ainsi que nous spécialisons les outils de méta-modélisation en implantant directement la syntaxe concrète du niveau M1 dans l'outil. Toutefois, la représentation graphique du niveau M1 pourra être spécialisée au niveau M2 à l'aide de l'éditeur de dessins fourni par le composant *DrawingArea*.
  Par exemple au concept *Composant* on peut associer comme représentation graphique au niveau M2, un rectangle divisé en deux sections. La première section contiendra le nom du *Composant*. Dans la méthode *getRepresentation()* du composant *ConcreteSyntax* associé à ce concept sera donc implanté le code Java permettant de dessiner ce rectangle.

- ***Semantic*** : à chaque concept du langage est associé un composant composite *Semantic* de type *SemanticType*, qui définit la sémantique du concept associé. Le composant *Semantic* contient deux sous composants primitifs qui sont ***M2Semantic*** et ***M1Semantic***. *M2Semantic* implémente la sémantique du concept au niveau de l'éditeur de méta-modèles (niveau M2) et *M1Semantique* sa sémantique au niveau de l'éditeur de modèles (niveau M1).

> Par exemple, notre langage de méta-modélisation dédié au domaine des composants fournit une relation que nous avons appelée *collage* (Cf. section 8.3.2) et qui a une sémantique particulière. Premièrement, au niveau M2 *collage* ne peut exister qu'entre un *Composant* et un *Connecteur*. Deuxièmement, si au niveau M2 il existe une relation *collage* entre un composant *comp* et un connecteur *conn*, alors au niveau M1 cela se traduira par le fait qu'on peut "coller" une instance de *conn* sur une instance de *comp* (Cf. Figure 6.1, les connecteurs sont collés autour des composants). La première règle s'applique au niveau méta-modèle et sera donc implantée dans *M2Semantic*, la deuxième s'applique au niveau modèle et sera implantée dans *M1Semantic*.

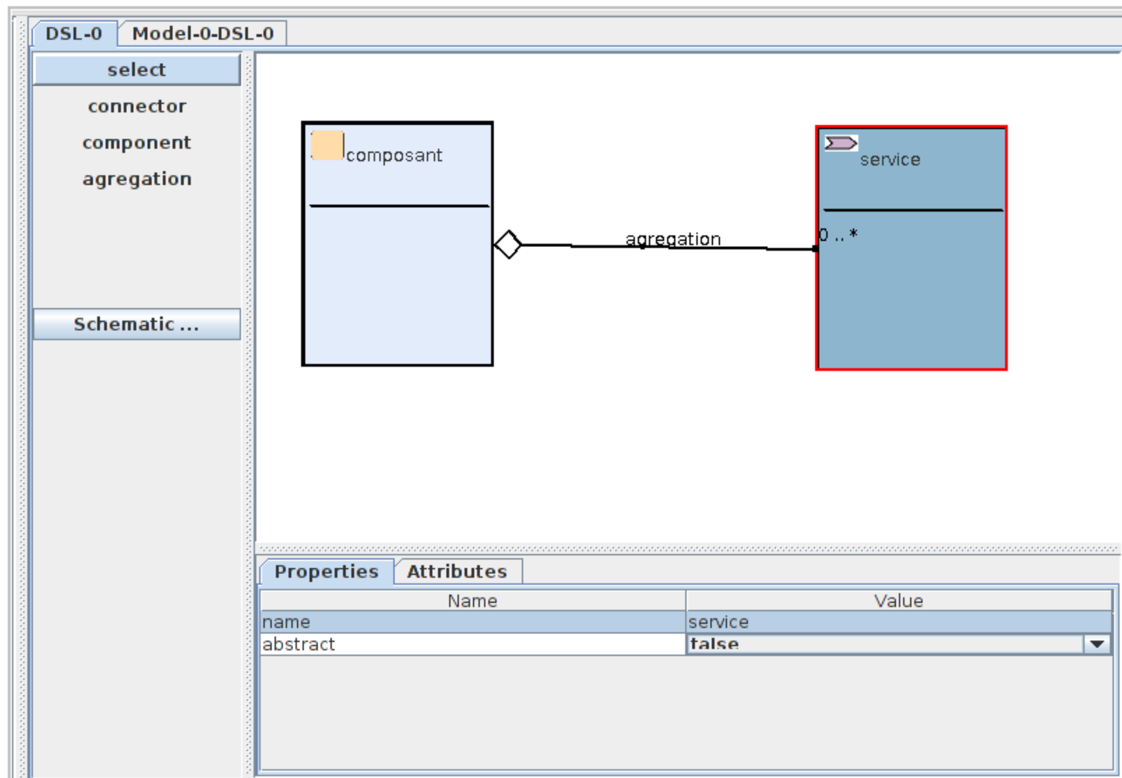
Nous permettons la définition de la syntaxe concrète du niveau M1 et du niveau M2 dans un seul composant. Mais en toute rigueur et pour plus de modularité, il aurait fallu définir ses deux syntaxes dans deux composants séparés comme nous l'avons fait pour la sémantique. L'architecture de GyTUNe va donc être modifiée dans ce sens, mais cette modification n'est pas prise en compte dans ce document.

8.5 Fonctionnement général

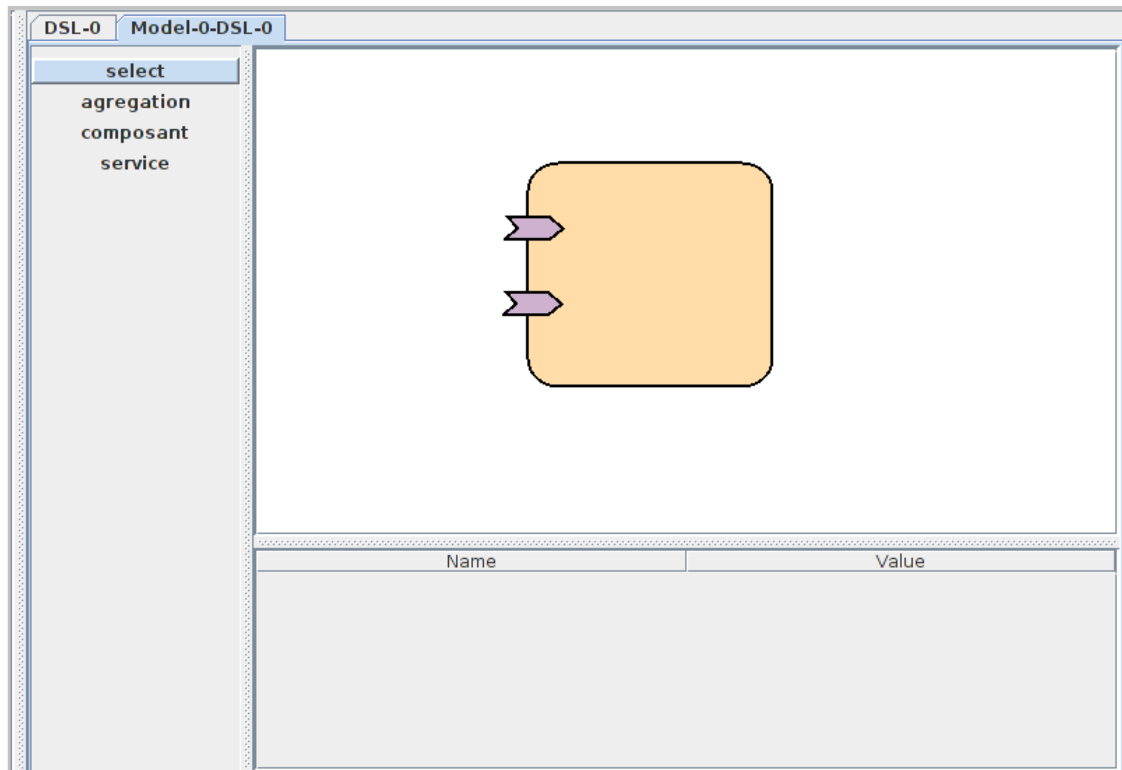
Nous avons trois profils d'utilisateur de GyTUNe : le premier utilisateur est celui qui construit un environnement de méta-modélisation dédié, nous l'appellerons userM3. Le second est celui qui utilise l'environnement de méta-modélisation créé par un userM3 pour définir des DSML, nous l'appellerons userM2. Le troisième est celui qui utilise un outil de modélisation généré par userM2, nous l'appellerons userM1. Les utilisateurs userM3 et userM2 spécifient leurs langages en trois étapes : spécification de la syntaxe abstraite, spécification de la syntaxe concrète et enfin spécification de la sémantique. Nous allons montrer le fonctionnement général de GyTUNe du point de vue de ces trois utilisateurs en faisant ressortir les mécanismes mis en œuvre par GyTUNe à chacune des actions effectuées par ces utilisateurs. Nous nous appuyons sur notre exemple du domaine des composants que nous allons restreindre pour la démonstration à trois concepts : *composant*, *connecteur* et *agrégation*. Le concept *agrégation* correspond au concept *collage* que nous définissons à la section 8.3.2. Le but étant que le userM2 arrive à méta-modéliser des langages dédiés au domaine des composants tels que le DSML du modèle à composants SCA simplifié présenté à la Figure 8.12(a) et que le userM1 arrive à faire des modèles tels que le modèle présenté à la Figure 8.12(b).

8.5.1 userM3 : spécification de la syntaxe abstraite

Le userM3 pour spécifier la syntaxe abstraite de son langage de méta-modélisation crée (décrit l'ADL fractal de) trois composants composites de type *AbstractSyntaxType* : *component*, *connector* et *agregation*. L'ensemble de ces trois composites représente sa



a)



b)

FIGURE 8.12 – (a) DSL SCA simplifié. (b) exemple de modèle SCA

syntaxe abstraite. Il relie ensuite chacun des composites à l'interface concept du *DrawingArea*.

Si le userM3 s'arrête à ce niveau et qu'il exécute l'application GyTUNe, le composant *DrawingArea* va parcourir tous les composants liés à son interface *concept* (*component*, *connector* et *agregation*). Pour chaque composant trouvé, il crée un *item* qui va encapsuler le composant et qui aura le même nom que celui-ci. Enfin, chaque *item* est rajouté dans la palette de l'éditeur représenté à la Figure 8.12(a). À ce stade le userM2 ne pourra rien faire avec cet outil parce que la syntaxe concrète et la sémantique de chacun des concepts n'ont pas encore été définies.

8.5.2 userM3 : spécification de la syntaxe concrète

Le userM3 pour spécifier la syntaxe concrète de son langage de méta-modélisation crée trois composants primitifs de type *ConcreteSyntaxType* : *componentSC*, *connectorSC* et *agregationSC*. Il les rajoutent respectivement dans *component*, *connector* et *agregation*. Les composants *componentSC*, *connectorSC* et *agregationSC* fournissent une interface *ConcreItf* puisqu'ils sont du type *ConcreteSyntaxType*. Les principales méthodes qui doivent être implantées dans *componentSC*, *connectorSC* et *agregationSC* sont :

- ***isLink()*** : permet de savoir si au niveau M2 le concept concerné est une liaison ou une classe (au sens UML) ;
- ***getRepresentation()*** : dans le cas où *isLink()* retourne *false*, *getRepresentation()* doit être implantée pour définir la représentation du concept concerné au niveau M2 ;
- ***getLinkRepresentation(int x, int y, int cx, int cy, Graphics g)*** : dans le cas où *isLink()* retourne *true*, *getLinkRepresentation()* doit être implantée pour définir la représentation du concept concerné au niveau M2 ;
- ***isModelLink()*** : qui va permettre de savoir si au niveau M1 le concept concerné est une liaison ou une classe ;
- ***getModelRepresentation(Element e)*** : dans le cas où *isModelLink()* retourne *false*, *getModelRepresentation()* doit être implantée pour définir la représentation par défaut du concept concerné au niveau M1 ;
- ***getModelLinkRepresentation(int x, int y, int cx, int cy, Graphics g)*** : dans le cas où *isModelLink()* retourne *true*, *getModelLinkRepresentation()* doit être implantée pour définir la représentation du concept concerné au niveau M1 ;
- ***isInChargeOfModelComputeClosestBorder()*** : il se pose le problème des points de connexion entre une liaison et un concept. Ces points de connexion doivent pouvoir être définis autant au niveau M2 qu'au niveau M1. Toutefois, la nécessite d'avoir des points de connexion prédéfinis est beaucoup moins importante lorsqu'on défini un méta-modèle. C'est pourquoi au niveau M2 nous avons choisi de définir les points de connexion par défaut [1]. Par contre pour le niveau M1, nous permettons à userM3 de définir des points de connexion pour des liaisons au niveau M1. Lorsque la méthode *isInChargeOfModelComputeClosestBorder* retourne *true*, cela signifie

1. Nous choisissons le points de liaison qui permettent d'avoir la liaison la plus courte

que la méthode suivante *computeModelClosestBorder* sera appelée pour le calcul des points de connexion. Sinon c'est le *DrawingArea* qui se chargera de trouver les points de connexion de cette liaison ;
– ***computeModelClosestBorder(MyLink l)*** : l représente une référence vers un objet de type *MyLink* qui est une structure de donnée permettant de représenter les liaisons d'un méta-modèle . Si *isInChargeOfModelComputeClosestBorder()* retourne true, userM3 doit implanter la méthode *computeModelClosestBorder(MyLink l)* qui va déterminer quel est le point de connexion sur la source et le point de connexion de la liaison sur la destination. Remarquons au passage que l'éditeur de dessin du *DrawingArea* permet de spécialiser la représentation d'un élément d'un méta-modèle et de définir des points de connexion sur ces représentations. Ces points pourront ensuite être utilisés par la méthode *computeModelClosestBorder*.

Syntaxe concrète de *component*

UserM3 implante les méthodes requises pour la syntaxe concrète du concept *component* de la façon suivante :
– ***isLink()*** : retourne *false*, un *component* ne sera pas une liaison au niveau M2.
– ***getRepresentation()*** : userM3 implante cette méthode de telle sorte qu'elle renvoie un JPanel dans lequel il a dessiné un rectangle (Cf. Figure 8.13). JPanel que le DrawingArea pourra afficher et manipuler dans sa zone de dessin ;
– ***isModelLink()*** : retourne *false*, un *component* ne sera pas une liaison au niveau M1 ;
– ***GetModelRepresentation()*** : n'est pas implantée, ce qui signifie que ce sera à userM2 d'assigner une représentation graphique (avec l'éditeur de dessin) pour le niveau M1 à chaque élément de type *component* de ses langages (Figure 8.14).

Syntaxe concrète de *connector*

UserM3 implante la syntaxe concrète d'un *connector* qui est exactement la même que celle d'un *component*. La seule différence étant la couleur du rectangle renvoyé par *getRepresentation()* pour ce concept au niveau M2.

Syntaxe concrète d'*agregation*

UserM3 implante les méthodes requises pour la syntaxe concrète du concept *agregation* de la façon suivante :
– ***isLink()*** : retourne *true*, une *agregation* sera une liaison au niveau M2.
– ***getLinkRepresentation(int x, int y, int cx, int cy, Graphics g)*** : dans cette méthode userM3 va utiliser la référence du Graphics utilisé par le *DrawingArea* qui lui est

```
public Vector<ElementToDraw> getSchema()
{
    v.add(new ElementToDraw(0,0,width,height,"rectangle",new Color(226,236,251)));
    v.add(new ElementToDraw(4,58,131,58,"line",new Color(0,0,0)));
    return v;
}

@Override
public JPanel getRepresentation()
{
    JPanel jp=(new JPanel()
    {
        private static final long serialVersionUID = 3970222525326355332L;

        public void paint(Graphics g)
        {
            Vector<ElementToDraw> v=getSchema();
            for(int i=0;i<v.size();i++)
            {
                ((ElementToDraw)v.get(i)).paintElement(g, 0);
            }
        }
    });
    jp.setSize(width, height);
    return jp;
}
```

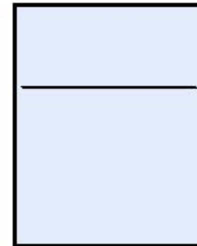


FIGURE 8.13 – Méthode *getRepresentation()* de *ComponentSc*

passé en paramètre pour dessiner une liaison allant du point (x,y) au point (cx, cy). Il dessine également à l'extrémité (cx, cy) un losange (symbole généralement utilisé pour les agrégations) ;

– ***isModelLink()*** : retourne *true*, une *agregation* sera une liaison au niveau M1 ;
– ***getModelLinkRepresentation(int x, int y, int cx, int cy, Graphics g)*** : n'est pas implantée parce qu'au niveau modèle une *agregation* n'aura pas de représentation graphique. Par contre, mettre une *agregation* entre deux éléments e1 et e2 au niveau modèle aura une incidence sur les coordonnées de ces éléments. Cette action est implantée dans *computeModelClosestBorder(MaLiaison l)* ;
– ***computeModelClosestBorder(MyLink l)*** : lorsque userM2 assigne une représentation graphique à un *component* ou à un *connector* à l'aide de l'éditeur de dessin du *DrawingArea*, il peut également définir des points de connexions qui pourront éventuellement être utilisés par des liaisons. La méthode *computeModelClosestBorder(MyLink l)* d'*agregationSC* va trouver les points de connexions les plus proches entre l'élément source et l'élément cible de la liaison l. Ensuite, elle attribue à ces deux points les mêmes coordonnées dans la zone de dessin du *DrawingArea*. Cela aura pour effet visuel de "coller" l'élément source de la liaison à sa cible ou inversement.

À chacun de ces trois composants de type *ConcreteSyntaxType*, userM3 ajoute des propriétés qui seront visibles dans la table des propriétés au niveau méta-modèle ou au niveau modèle. Par exemple une propriété "*name*" de type String, propriété qui permettra d'assigner un nom au concept associé au niveau méta-modèle et dont la valeur devra apparaître sur la représentation du concept. Une propriété "*abstract*" de type booléen qui permettra à userM3 d'autoriser ou non qu'un concept de type *component* ou *connector* présent au niveau M2 le soit au niveau M1.

8.5.3 userM3 : spécification de la sémantique

Le userM3 pour spécifier la sémantique de son langage de méta-modélisation va créer trois composants composites de type *SemanticType* : *componentSem*, *connectorSem* et *agregationSem*. Il crée ensuite pour chacun d'eux, deux composants primitifs : *M2Semantic* et *M1Semantic* qui vont implanter respectivement la sémantique du concept associé au niveau méta-modèle et sa sémantique au niveau modèle. On aura alors les composants *componentMMSem*, *componentMSem*, *connectorMMSem*, *connectorMSem*, *agregationMMSem* et *agregationMSem* qui fournissent tous une interface *SemanticItf*. Les principales méthodes de *SemanticItf* qui doivent être implantées par ces composants primitifs sont :

– ***isConnectable(Element from, Element to)*** : cette méthode est appelée sur les concepts de type liaison. Elle permet de vérifier si les éléments *from* et *to* peuvent être reliés par la liaison sur qui est appelée cette méthode ;
– ***validateCreation(Element myself, Vector<Element> elements)*** : permet à userM3 de spécifier des contraintes qui doivent être vérifiées à la création d'un concept ;
– ***validateModification(Vector<Attribut> attributs, Hashtable<String, String []> propertiesToDisplay)*** : permet à userM3 de spécifier des contraintes qui doivent être vérifiées à la modification d'un concept ;
– ***makeTool(Element myself)*** : permet à userM3 de spécifier si un concept présent au niveau M2 doit être présent au niveau M1 ou non. Si oui, cette méthode renvoie un *item* pour ce concept, *item* qui sera intégré dans la palette de l'éditeur de modèles. Comme nous le verrons dans le méta-modèle de Fractal dans le chapitre suivant, il y a des composants qui sont abstraits et qu'on ne désire pas avoir dans la palette de l'éditeur de modèles (niveau M1). userM3 peut contrôler cela par des propriétés.

Sémantique *component*

UserM3 implante les méthodes requises pour la sémantique du concept *component* au niveau méta-modèle (*M2Semantic*) de la façon suivante :

– ***validateCreation(Element myself, Vector<Element> elements)*** : dans cette méthode, userM3 vérifie par exemple l'unicité de la propriété *name* ;
– ***validateModification(Vector<Attribut> attributs, Hashtable<String, String []> propertiesToDisplay)*** : dans cette méthode, userM3 vérifie par exemple que les valeurs que userM2 assigne aux propriétés *name* et *abstract* respectent les types de ces propriétés ;
– ***makeTool(Element myself)*** : Dans cette méthode, userM3 retourne un *item* pour *myself* qui sera rajouté dans la palette de l'éditeur de modèles.

Au niveau modèle aucune sémantique particulière n'est implantée pour un *component*.

Sémantique *connector*

UserM3 implante la sémantique d'un *connector* qui est exactement la même que celle d'un *component*.

Sémantique *agregation*

UserM3 implante les méthodes requises pour la sémantique au niveau M2 du concept *agregation* :

– ***isConnectable(Element from, Element to)*** : au niveau M2, userM3 souhaite qu'une *agregation* ne soit autorisée qu'entre un élément de type *component* et un élément de type *connector*. Dans cette méthode userM3 vérifie que *from* et *to* soient bien un élément de type *component* pour l'un et un élément de type *connector* pour l'autre. Sinon la méthode renvoi false et la liaison n'est pas autorisée ;
– ***validateCreation(Element myself, Vector<Element> elements)*** : vérifie par exemple l'unicité de la propriété *name* dans le méta-modèle en cours d'édition ;
– ***validateModification(Vector<Attribut> attributs, Hashtable<String, String []> propertiesToDisplay)*** : UserM3 a ajouté une propriété *minFrom* et une propriété *maxFrom* au concept *agregation*. Ces propriétés représentent les cardinalités d'une *agregation*. Dans cette méthode userM3 vérifie que les valeurs que userM2 assigne aux propriétés *minFrom* et *maxFrom* sont positives ou égales à -1 ;
– ***makeTool(Element myself)*** : dans notre exemple, userM3 autorise qu'un item soit créé dans la palette de l'éditeur de modèles pour chaque *agregation* définie au niveau M2.

Au niveau M1 la sémantique d'une *agregation* consistera essentiellement en la vérification des valeurs des propriétés.

Nous allons maintenant dérouler le fil d'exécution pour que userM2 arrive à implanter le DSML décrit à la Figure 8.12(a).

8.5.4 UserM2 : Spécification de la syntaxe abstraite

Pour créer un élément *component* dans la zone de dessin de l'éditeur de méta-modèle, userM2 clique sur le bouton *component* dans la palette. Le *DrawingArea* enregistre alors le composant *component* encapsulé dans ce bouton comme étant le composant courant. Lorsque userM2 clique ensuite dans la zone de dessin, le *DrawingArea* utilise les API Fractal pour récupérer le sous composant *concreteSyntax* du composant courant. Dans notre cas ce sera le composant *componentSC*. Le *DrawingArea* récupère ensuite l'interface serveur *ConcreteItf* de *componentSC* et appelle sa méthode *isLink()* pour savoir si

userM2 veut créer une liaison ou une classe. *isLink()* de *componentSC* renvoie *false*. Le *DrawingArea* appelle alors la méthode *getRepresentation()* de *ComponentSC* qui lui renvoie un JPanel dans lequel est dessiné un rectangle. JPanel que le DrawingArea va alors ajouter à l'endroit où userM2 a cliqué dans la zone de dessin. Ensuite, le *DrawingArea* utilise les API Fractal pour récupérer le sous composant *semantic* du composant courant. Ce sera le composant *componentSem*. Il récupère ensuite le sous composant *M2Semantic* de *componentSem* et obtient le composant *componentMMSem*. Enfin, il récupère l'interface serveur *SemanticItf* de *componentMMSem* et appelle sa méthode *validateCreation* qui va vérifier la contrainte d'unicité de nom pour l'élément de type *component* nouvellement créé. userM2 attribue la valeur *"composant"* à la propriété *name* de l'élément de type *component* nouvellement créé.

UserM2 clique ensuite sur le bouton *connector*, et tout le processus précédent est répété, mais cette fois ci avec comme composant courant, le composant *connector*. userM2 attribue la valeur *"service"* à la propriété *name* de l'élément de type *connector* nouvellement créé. Le *DrawingArea* appelle alors la méthode *validateModification* de *connectorMMSem* qu'il a récupéré lors de la création du nouvel élément *connector*. *validateModification* de *connectorMMSem* va autoriser la valeur "*service*".

Lorsque userM2 clique maintenant sur le bouton *agregation* pour créer une *agregation* entre l'élément *composant* créé et l'élément *service*. Le même mécanisme s'opère, le *DrawingArea* récupère l'interface *ConcreteItf* d'*agregationSC* et appelle sa méthode *isLink()*. La méthode *isLink()* de *agregationSC* renvoyant *true*, le *DrawingArea* va alors attendre le deuxième clic de l'utilisateur parce que c'est une liaison que userM2 veut créer. Il récupère ensuite l'interface *SemanticItf* du composant *agregationMMSem* de la même façon que pour l'élément *composant*. Le *DrawingArea* récupère les éléments sur lesquels userM2 a cliqué dans la zone de dessin et les passent en paramètre de la méthode *isConnectable* d'*agregationMMSem*. Si *isConnectable* retourne true alors le *DrawingArea* appelle la méthode *getLinkRepresentation* d'*agregationSC* en lui passant en paramètre les coordonnées des points où l'utilisateur a cliqué ainsi que le *Graphics* permettant de dessiner dans la zone de dessin du *DrawingArea*. *getLinkRepresentation* d'*agregationSC* va donc dessiner une ligne allant du premier point au deuxième, ainsi qu'un losange au niveau du deuxième point.

Chaque élément créé au niveau M2 possède une référence vers l'interface *ConcreteItf* du sous composant de type *SyntaxConcreType* du composant à partir duquel il a été créé, ainsi qu'une référence vers l'interface *SemanticItf* de son sous composant *M2Semantic* et une référence vers l'interface *SemanticItf* de son sous composant *M1Semantic*.

8.5.5 userM2 : spécification de la syntaxe concrète

Pour assigner une représentation graphique à l'élément *composant* de son langage, userM2 sélectionne d'abord cet élément à l'aide du bouton *selection* de la palette. Ensuite, il clique sur le bouton *schematic* qui va ouvrir l'éditeur de dessin du *DrawingArea*, éditeur

représenté à la Figure 8.14. userM2 dessine ou importe l'image d'un carré arrondi et orange qui sera la représentation de tous les éléments de ce type au niveau modèle. Il y spécifie également deux points de connexion sur la face latérale gauche du carré. L'éditeur de dessin retourne un JPanel qui contient la représentation graphique spécifiée par userM2.

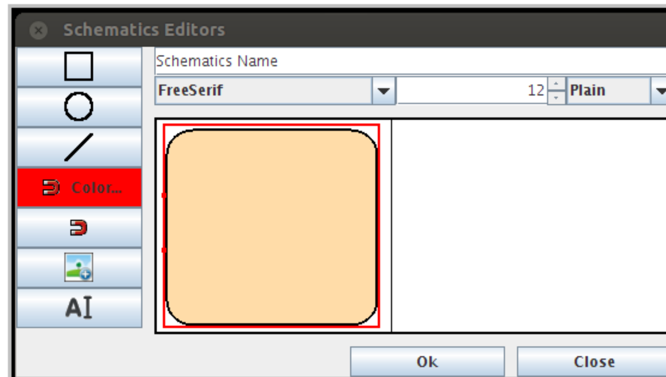


FIGURE 8.14 – Éditeur de dessin du *DrawingArea*

Pour assigner une représentation à l'élément *connector* userM2 procède de la même façon. Il assigne la représentation d'un *connector* SCA à l'élément *service* (Figure 8.12(b)).

8.5.6 userM2 : spécification de la sémantique

userM3 a assigné une sémantique par défaut pour le niveau M1 à chacun des éléments du langage de userM2, *a priori* userM2 ne devrait pas avoir besoin de la changer parce que cette sémantique est directement issue du domaine.

8.5.7 userM2 : Génération de l'outillage

Pour générer l'outillage du niveau M1, userM2 clique sur le menu *Model* puis sur le sous-menu *new* et enfin sélectionne le DSML qu'il a créé. Le *DrawingArea* crée alors un nouvel éditeur. Le *DrawingArea* parcourt la liste des éléments présents dans le langage défini par userM2 et pour chaque élément trouvé il appelle la méthode *makeTool(Element myself)* de l'interface *SemanticItf* du composant *M2Semantic* dont l'élément a la référence. Cette méthode va permettre de créer ou non un *item* pour l'élément concerné au niveau M1. Chacun des *item* encapsule l'élément du DSML qui a permis de le créer. Dans notre exemple la méthode *makeTool* de *componentMMSem* crée un *item* pour l'élément *composant* du DSML de userM2. La méthode *makeTool* de *connectorMMSem* crée un item pour l'élément *service*. La méthode *makeTool* de *agregationMMSem* crée également un item pour l'élément *agregation* du DSML de userM2.

8.5.8 userM1 : définition d'un modèle

Pour définir le modèle présenté à la Figure 8.12(b), userM1 clique sur le bouton *composant*, le *DrawingArea* enregistre l'élément *composant* du DSML de userM2 encapsulé dans ce bouton comme élément courant. Lorsque userM1 clique dans la zone de dessin, le *DrawingArea* récupère la référence de l'élément courant vers l'interface *concreteItf* qui dans ce cas est *componentSC*. Le *DrawingArea* appelle ensuite la méthode *isModelLink()* de *componentSC* qui retourne *false*. Le *DrawingArea* va examiner l'élément courant pour savoir si userM2 a assigné une représentation graphique à cet élément (avec l'éditeur de dessin) ou s'il faut appeler *getModelRepresentation()* pour obtenir la représentation par défaut de *componentSC*. Dans notre cas, userM2 a assigné une représentation graphique. Le *DrawingArea* récupère alors le JPanel qui contient cette représentation et l'affiche dans la zone de dessin. Le *DrawingArea* récupère ensuite la référence de l'élément courant vers l'interface *SemanticItf* du composant *M1Semantic*. Il appelle la méthode *validateCreation* que userM3 a implantée de telle sorte qu'elle retourne toujours *true* et donc autorise toujours la création de la nouvelle instance de *composant*.

Pour créer les deux instances du connecteur *service*, userM1 clique sur le bouton service et procède de la même façon que pour créer une instance de *composant*.

Pour relier les deux instances de *connector* créées à l'instance de *component*, userM1 clique sur le bouton *agregation*. Le *DrawingArea* enregistre l'élément *agregation* du DSML de userM2 comme l'élément courant. Lorsque userM1 clique dans la zone de dessin le *DrawingArea* récupère la référence de l'élément courant *agregation* vers l'interface *concreteItf* qui dans ce cas est *AgregationSC*. Le *DrawingArea* appelle ensuite la méthode *isModelLink()* de *AgregationSC* qui retourne *true*. Il vérifie que userM1 a bien cliqué sur un élément dans la zone de dessin, et attend le 2e clic de userM1. Après le 2e clic de userM1, le *DrawingArea* récupère la référence de l'élément courant vers l'interface *SemanticItf* du composant *M1Semantic*. Il appelle la méthode *isConnectable* de cette interface qui va vérifier que les deux éléments sur lesquels userM1 a cliqué sont bien de type *composant* pour l'un et *service* pour l'autre. Si oui elle retourne true et donc autorise la liaison. Lorsque la liaison est autorisée, le *DrawingArea* crée un nouvel élément liaison de type *agregation*, enregistre la source et la destination de cette liaison comme étant les deux éléments sur lesquels userM1 a cliqué. Puis *DrawingArea* appelle la méthode *computeModelClosestBorder* de *AgregationSC* qui cherche les deux points de connexion les plus proches sur les représentations graphiques de la source et de la cible de la liaison. Cette méthode va assigner les mêmes coordonnées dans la zone de dessin à ces points de connexion ce qui aura pour effet visuel de coller l'élément de type *service* à l'élément de type *composant*. Le *DrawingArea* appelle ensuite la méthode *getModelLinkRepresentation* de *AgregationSC* qui ne retourne rien parce que userM3 n'a pas implanté une représentation par défaut pour l'*agregation* au niveau M1. Enfin, pour créer le 2e élément *service* et le coller sur l'élément *component*, userM1 procède de la même façon que pour le premier et obtiens au final le modèle présenté à la Figure 8.12(b).

Avec le déroulement de ce fil d'exécution, on peut constater la totale indépendance entre la partie générique de GyTUNe et la partie spécifique. La partie spécifique implan-

tée par les userM3 peut évoluer et être adaptée sans que cela influe sur le comportement de la partie générique. Tant que les contrats entre les deux parties sont bien respectés c'est-à-dire que les composants définis dans la partie spécifique fournissent bien les interfaces *ConcreteItf* et *SemanticItf* alors la partie générique pourra créer un environnement de méta-modélisation et le spécialiser en fonction des services que lui fournit la partie spécifique.

Chapitre 9

Validation

9.1 Introduction

Notre objectif initial était d'être en mesure d'implanter des DSML pour la définition des politiques d'administration dans TUNe. Ces DSML pour TUNe sont tous dans le domaine des langages à composants.
Pour montrer la généricité de GyTUNe, nous allons présenter deux cas d'utilisation dans deux domaines différents. Le premier domaine est un domaine technique dit horizontal, il s'agit du domaine des composants. Nous allons utiliser GyTUNe pour créer un environnement de méta-modélisation dédié aux architectures à composants et nous montrerons comment implanter deux DSML de TUNe dans ce domaine. Le deuxième cas d'utilisation concernera un domaine métier dit vertical, il s'agit du domaine de la gestion de projets. Nous allons utiliser GyTUNe pour créer un environnement de méta-modélisation permettant de créer des DSMLs pour la construction de diagrammes temporels et nous montrerons comment implanter deux DSML dans ce domaine (Diagrammes de GANTT et de PERT utilisés par les responsables de projets).
Premièrement, nous allons présenter chaque domaine et les spécificités qu'un outil de méta-modélisation qui lui serait dédié devrait prendre en compte. Deuxièmement, nous présenterons, comment ces spécificités ont été implantées, et enfin nous présentons les environnements de méta-modélisation obtenus avec pour chacun des exemples de DSMLs créés.

9.2 Cas d'utilisation : architectures à composants

Nous avons déjà présenté le domaine des composants tout au long de ce document, nous allons rappeler ici ce qui le caractérise et les spécificités qu'un environnement de méta-modélisation qui lui serait dédié devrait intégrer. Ce domaine est caractérisé par les notions suivantes :

- **Composant** : c'est une unité de structuration souvent représentée par une boite et qui possède des propriétés. La représentation graphique d'un composant est souvent une boite ou un rectangle, mais peut être n'importe quelle figure ;
- **Connecteur** [1] c'est une interface de connexions liée à un composant. Graphiquement, la représentation d'un connecteur doit pouvoir être "collée" à celle du composant, et les points d'attache des connecteurs sur les composants sont prédéfinis et peuvent avoir des positions variables. Enfin, la représentation graphique d'un connecteur peut être n'importe quelle figure ;
- **Liaison** : permet de relier des connecteurs ou des composants (s'il n'y a pas de connecteur) entre eux afin de constituer une architecture logicielle. Graphiquement une liaison peut avoir une forme variable (en pointillé, avec des flèches ou des carrés aux extrémités . . .).

Nous allons décrire ce cas d'utilisation en nous appuyant sur l'exemple d'un méta-modèle du modèle à composant Fractal conçu avec l'environnement de méta-modélisation créé avec GyTUNe. Ce méta-modèle est présenté à la Figure 9.1(a), la Figure 9.1(b) présente un modèle qui lui est conforme.

9.2.1 Création de l'outil de méta-modélisation

La définition du langage de méta-modélisation suit les étapes vues au chapitre précédent à savoir : définition de la syntaxe abstraite, définition de la syntaxe concrète et enfin définition de la sémantique.

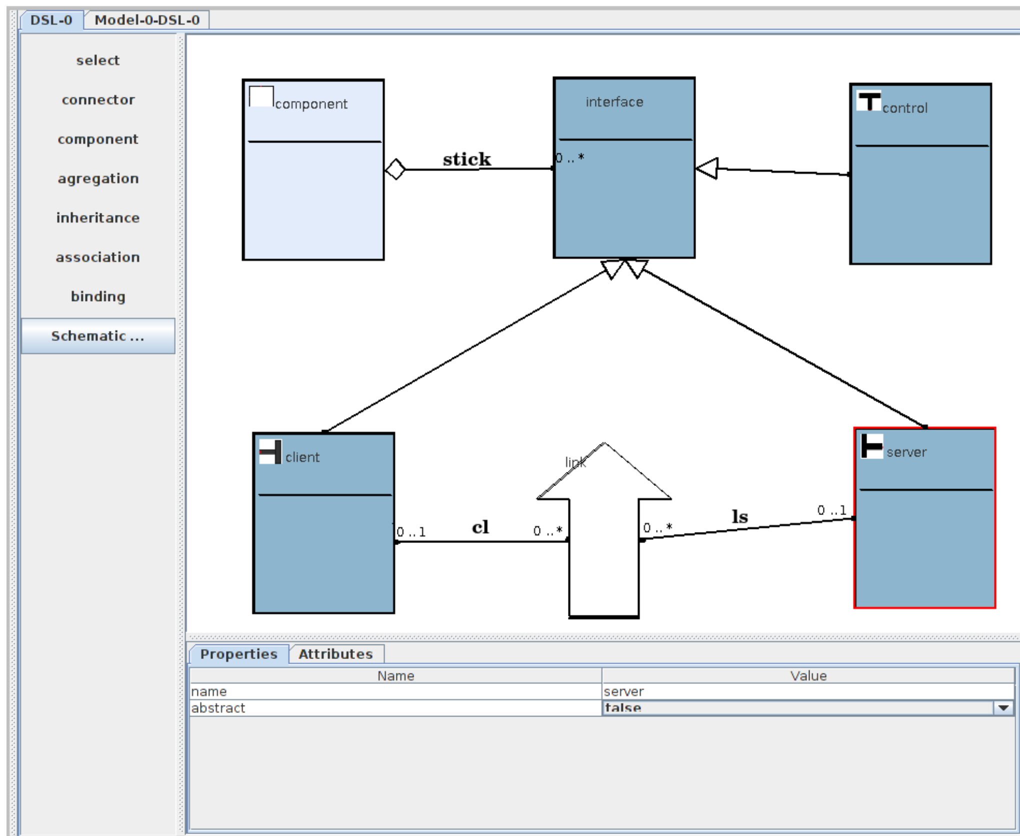
Définitions de la syntaxe abstraite

Ce langage définit trois concepts de base qui sont :

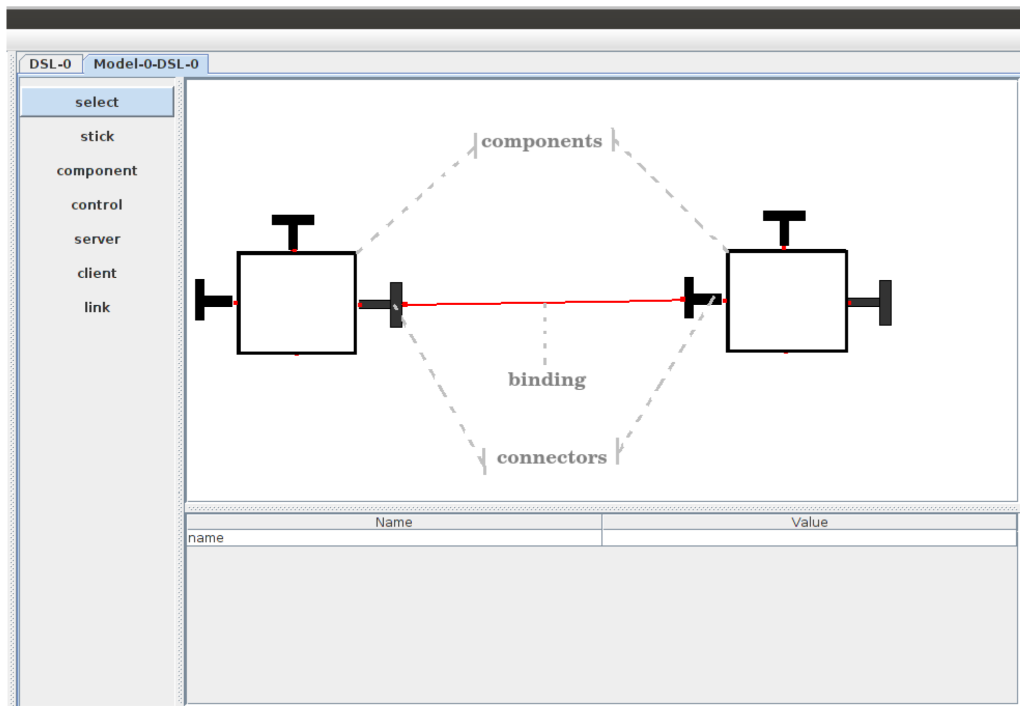
- *component* : qui représente le concept composant. L'élément *component* figurant dans le méta-modèle de Fractal présenté à la Figure 9.1(a) est un *component*, qui est représenté au niveau du modèle (Figure 9.1(b)) par une figure en l'occurrence ici un carré ;
- *connector* : qui représente le concept connecteur. Les éléments *interface*, *control*, *server*, *client* du méta-modèle de Fractal sont des *connector*s représentés au niveau du modèle par un "T" couché à droite ou à gauche (⊣ ou ⊢) ;
- *binding* : qui représente le concept liaison. L'élément *link* du méta-modèle de Fractal est un *binding* qui est représenté au niveau modèle par un fil [2] entre deux connecteurs.

1. connecteur est utilisé ici pour désigner une interface de composant. Connecteur est parfois utilisé dans la littérature pour désigner une liaison

2. Nous appellerons fil, un trait comme celui représenté à la Figure 9.1(b) et qui matérialise un *binding* entre deux *connectors*



a)



b)

FIGURE 9.1 – a) Méta-modèle de Fractal. b) exemple d'architecture Fractal

Ces concepts de base nous devons pouvoir les relier entre eux. Un *component* doit pouvoir être relié à un *connector*. De même, un *connector* doit pouvoir être lié à un *binding*. Les liaisons entre les *connector*s et les *component*s ont une sémantique différente des liaisons entre les *connectors* et les *bindings*. En effet, une liaison entre un *component* et un *connector* matérialise la relation *collage* que nous avons présentée à la section 8.3.2 du chapitre précédent. Cette liaison permet de coller un connecteur sur un composant au niveau M1. Tandis que, une liaison entre un *binding* et un *connector* traduit le fait qu'au niveau M1 un point d'attache du fil qui représente le *binding* se situe sur le *connector*. Nous avons défini deux concepts pour représenter ces deux types de liaison et un troisième concept qui permettra qu'un concept de base puisse hériter d'un autre. Il s'agit des trois concepts suivants :

- *association* : qui va permettre de spécifier la source et la destination d'un *binding*. Les éléments *cl* et *ls* du méta-modèle de Fractal sont des associations. L'*association* n'a pas de représentation au niveau modèle ;
- *agregation* : qui représente la relation entre un *component* et un *connector*. L'élément *stick* du méta-modèle de Fractal est une *agregation* permet au niveau modèle de "coller les *interfaces*" sur les *components* ;
- *inheritance* : qui va permettre qu'un concept puisse hériter d'un concept du même type. La liaison entre l'élément *interface* et l'élément *control* par exemple, est une *inheritance*. Ce sont les liaisons *inheritance* du méta-modèle qui permettent que la relation *stick puisse être appliquée sur control, server et client*.

La Figure 9.2 présente l'architecture Fractal obtenue après la création de tous ces concepts. Pour obtenir la représentation graphique de cette architecture, nous avons utilisé l'outil Fractal explorer qui est une console graphique d'administration d'applications Fractal.

Définition de la syntaxe concrète et définition de la sémantique

***component* :**

- **Syntaxe concrète** :
  - **propriétés** : un *component* a une propriété *name* de type String et une propriété *abstract* de type booléen qui seront présentes au niveau M2. Ces propriétés seront par exemple utilisées pour valider les contraintes définies dans la sémantique d'un *component* (l'utilisation de ces propriétés sera détaillée plus tard). Le userM2 pourra également définir des propriétés au niveau M2, propriétés qu'on retrouvera au niveau M1. Ces propriétés du niveau M1 pourraient notamment servir pour la génération de code, l'interprétation ou l'exécution des modèles mais cet aspect n'est pas abordé dans cette thèse ;
  - **représentation graphique au niveau M2** : la représentation graphique d'un *component* au niveau M2 est un rectangle divisé en deux sections (implanté dans *getRepresentation*). La valeur de sa propriété *name* est affichée dans la section supérieure ;

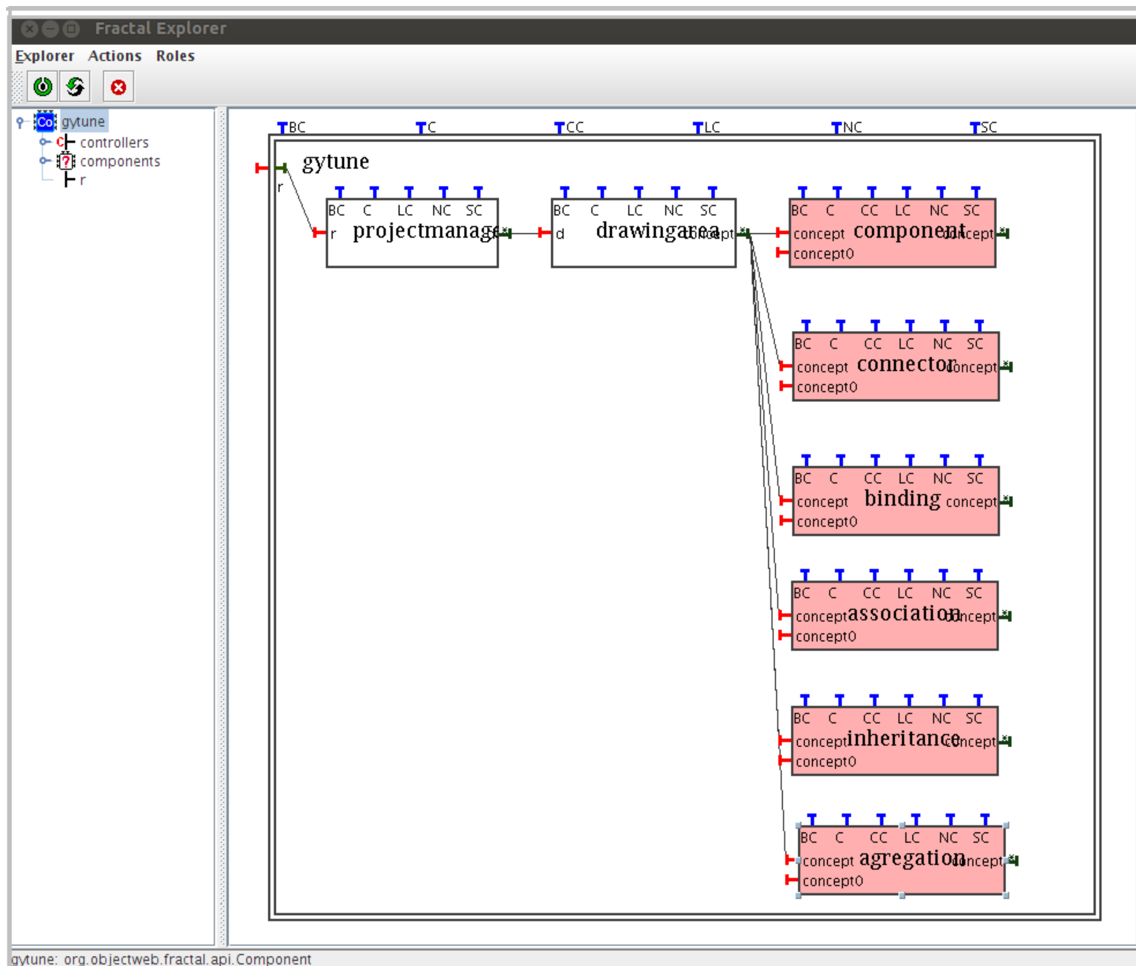


FIGURE 9.2 – Syntaxe abstraite

- **représentation graphique au niveau M1** : La représentation graphique d'un *component* au niveau M1 sera définie par userM2 à l'aide de l'éditeur de dessin du *DrawingArea* (par exemple un rectangle pour l'ADL de Fractal).

– **Sémantique** :
- **sémantique niveau M2** : la sémantique d'un *component* au niveau M2 va essentiellement consister en la vérification des valeurs de ses propriétés. Par exemple la vérification de l'unicité de la valeur de *name* ;
- **sémantique niveau M1** : un *component* n'a pas de sémantique pour le niveau M1.

***connector* :**

– **Syntaxe concrète** :
- **propriétés** : comme le concept *component*, un *connector* a une propriété *name* et une propriété *abstract* pour le niveau M2 ainsi qu'une propriété *name* pour le niveau M1 ;

  - **représentation graphique au niveau M2** : la représentation graphique d'un *connector* au niveau M2 est un rectangle comme pour *component* mais avec une couleur différente ;
  - **représentation graphique au niveau M1** : La représentation graphique d'un *connector* au niveau M1 sera définie par userM2 à l'aide de l'éditeur de dessin du *DrawingArea* (par exemple un ⊣ pour l'ADL Fractal).
- **Sémantique** :
  - **sémantique niveau M2** : essentiellement la vérification des valeurs de *name* et d'*abstract*. Par exemple, si la propriété *abstract* d'un *connector* au niveau M2 vaut *true* alors ce *connector* n'apparaîtra pas dans la palette au niveau M1. C'est le cas pour le *connector interface* du méta-modèle de Fractal (Figure 9.1) ;
  - **sémantique niveau M1** : pas de sémantique particulière.

***binding* :**

- **Syntaxe concrète** :
  - **propriétés** : comme le concept *component*, un *connector* a une propriété *name* et une propriété *abstract* pour le niveau M2 ainsi qu'une propriété *name* pour le niveau M1 ;
  - **représentation graphique au niveau M2** : la représentation graphique d'un *binding* au niveau M2 est un rectangle comme pour *component* et *connector* mais avec une couleur différente ;
  - **représentation graphique au niveau M1** : la représentation par défaut d'un *binding* au niveau M1 est un fil (Cf. Figure 9.1(b)). Celle-ci peut toutefois être spécialisée à l'aide de l'éditeur de dessin.
- **Sémantique** :
  - **sémantique niveau M2** : pas de sémantique particulière ;
  - **sémantique niveau M1** : au niveau M1, pour valider la création d'un *binding*, on vérifie qu'au niveau M2 ce *binding* a été prévu entre les concepts correspondants (vérification de la conformité du modèle avec le méta-modèle).

***agregation* :**

- **Syntaxe concrète** :
  - **propriétés** : le concept *agregation* a une propriété *minFrom* et une propriété *maxFrom* qui représentent les cardinalités d'une *agregation* au niveau M2 et qui permettent de limiter le nombre de connecteurs liés à un composant ;
  - **représentation graphique au niveau M2** : la représentation graphique d'une *agregation* au niveau M2 est un fil avec un losange au bout ;
  - **représentation graphique au niveau M1** : l'*agregation* n'a pas de représentation graphique au niveau M1, mais permet de coller un *connector* sur un *component* au niveau M1.
- **Sémantique** :

  - **sémantique niveau M2** : au niveau M2 une *agregation* n'est autorisée qu'entre un *connector* et un *component* ;
  - **sémantique niveau M1** : pour mettre une *agregation* entre un *connector* et un *component* au niveau M1, on vérifie qu'au niveau M2 cette *agregation* a été prévue entre ces éléments là (un méta-modèle peut spécifier qu'une instance d'un *connector* donné ne peut être collée que sur une instance d'un *component* donné. Comme dans l'exemple JEE qui va suivre).

***association* :**

- **Syntaxe concrète** :
  - **propriétés** : l'*association* a quatre propriétés qui définissent ses cardinalités sur sa source et sur sa destination. Ces cardinalités permettent de limiter le nombre de *binding* sur un *connector* ;
  - **représentation graphique au niveau M2** : la représentation graphique d'une *association* au niveau M2 est un fil ;
  - **représentation graphique au niveau M1** : l'*association* n'a pas de représentation graphique pour le niveau M1.
- **Sémantique** :
  - **sémantique niveau M2** : au niveau M2 une *association* n'est autorisée qu'entre un *connector* et un *binding* ou entre un *component* et un *binding* ;
  - **sémantique niveau M1** : pas de sémantique particulière.

***inheritance* :**

- **Syntaxe concrète** :
  - **propriétés** : une *inheritance* n'a pas de propriétés ;
  - **représentation graphique au niveau M2** : elle sera représentée au niveau M2 par un fil avec un triangle à une extrémité ;
  - **représentation graphique au niveau M1** : l'*inheritance* n'a pas de représentation au niveau M1.
- **Sémantique** :
  - **sémantique niveau M2** : une *inheritance* n'est autorisée qu'entre deux concepts de base et de même type (deux *components* ou deux *connectors* ou deux *bindings*). Lorsqu'un concept c1 hérite d'un concept c2, les attributs de c2 définis par userM2 sont ajoutés à c1. Au niveau modèle les liaisons autorisées pour c2 le seront également pour c1 ;
  - **sémantique niveau M1** : pas de sémantique particulière.

Exemple ADL JEE

Nous avons utilisé l'environnement créé ci-dessus pour implanter des DSML pour TUNe. Nous avons notamment implanté un langage de description d'application JEE. Ce langage présenté à la Figure 9.3 met en œuvre quatre composants *apache*, *tomcat*, *cjdbc*, *mysql*. À chacun de ces composants sont associé un ou deux connecteurs (le connecteur*modjk* pour *apache*, les connecteurs *ajp* et *dataSources* pour *tomcat*, *WAClients* et *backend* pour *cjdbc* et enfin le connecteur *dbclient* pour *mysql*). La Figure 9.4 représente

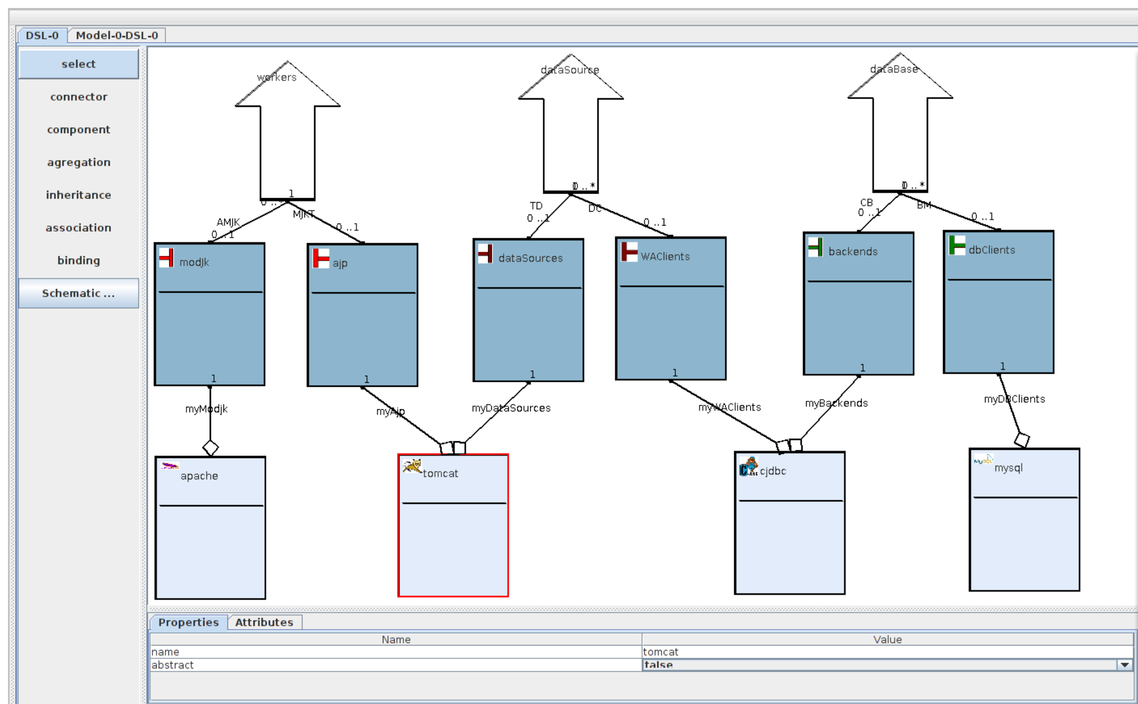


FIGURE 9.3 – ADL JEE

un modèle de cet ADL. Les cardinalités définies au niveau du méta-modèle n'autorisent qu'un seul connecteur *modjk* pour *apache*. Cette contrainte se vérifie bien au niveau de l'éditeur de modèles généré où il est impossible de coller deux *modjk* sur apache. Les connecteurs sont bien "collés" aux composants et les points de connexion des différents connecteurs sont bien ceux qui ont été définis par usserM2 sur la représentation d'un connecteur.

Exemple RDL

Le deuxième exemple est un langage pour la construction des diagrammes de reconfiguration pour TUNe. Ce langage dont le méta-modèle est décrit à la Figure 9.5 permet de construire des diagrammes états transitions. Ce langage définit un concept *etat* qui peut être un état *initial*, *final* ou *intermediaire*. Une transition est décrite par un binding qui permettra de relier les états entre eux. L'éditeur de modèles obtenus est présenté à la fi-

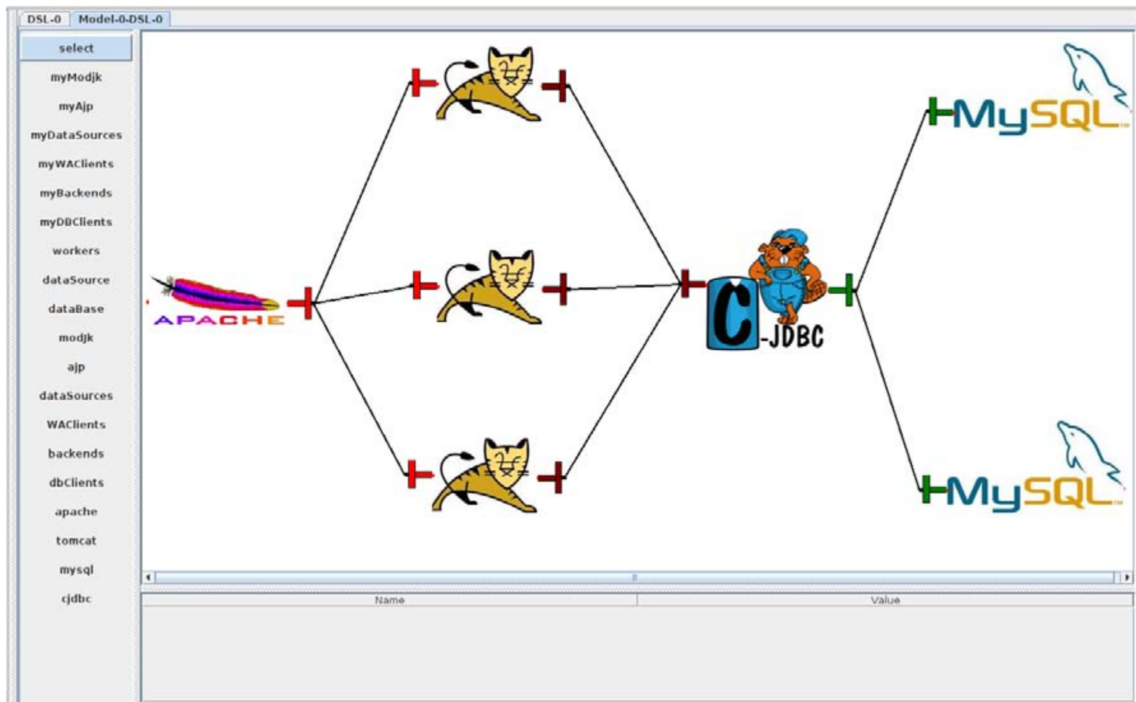


FIGURE 9.4 – modèle d'une application JEE

gure 9.6. Nous y avons défini un exemple de reconfiguration pour une application JEE. On peut y voir que la sémantique de l'héritage telle que définie dans l'environnement est bien respectée. Il est possible de lier un état *join* ou *fork* à un autre état parce que ces deux états héritent d'*intermediaire* qui lui peut être lié aux autres états.

9.3 Cas d'utilisation : gestion de projets

Le domaine ciblé par ce deuxième cas d'utilisation relève plus du domaine métier que technique, il s'agit du domaine de l'événementiel. Nous avons créé un environnement de méta-modélisation dédié à la gestion de projets. Cet environnement doit permettre de créer des outils de gestion de projets. Ces outils sont principalement de deux types : les outils de planification qui servent à établir le calendrier d'un projet et les outils de gestion comptables et de GRH, qui permettront principalement la gestion des ressources.

Les concepts qui caractérisent ce domaine sont les suivants :

- **Projet** : c'est l'ensemble des actions à entreprendre afin de répondre à un besoin défini dans des délais fixés. Un projet est ainsi une action temporaire comportant un début et une fin, mobilisant des ressources identifiées (humaines et matérielles) durant sa réalisation. Dans le cas d'un outil de planification, un projet est très souvent représenté avec une échelle de temps permettant de voir les différentes actions entreprises entre la date du début du projet et sa date de fin.

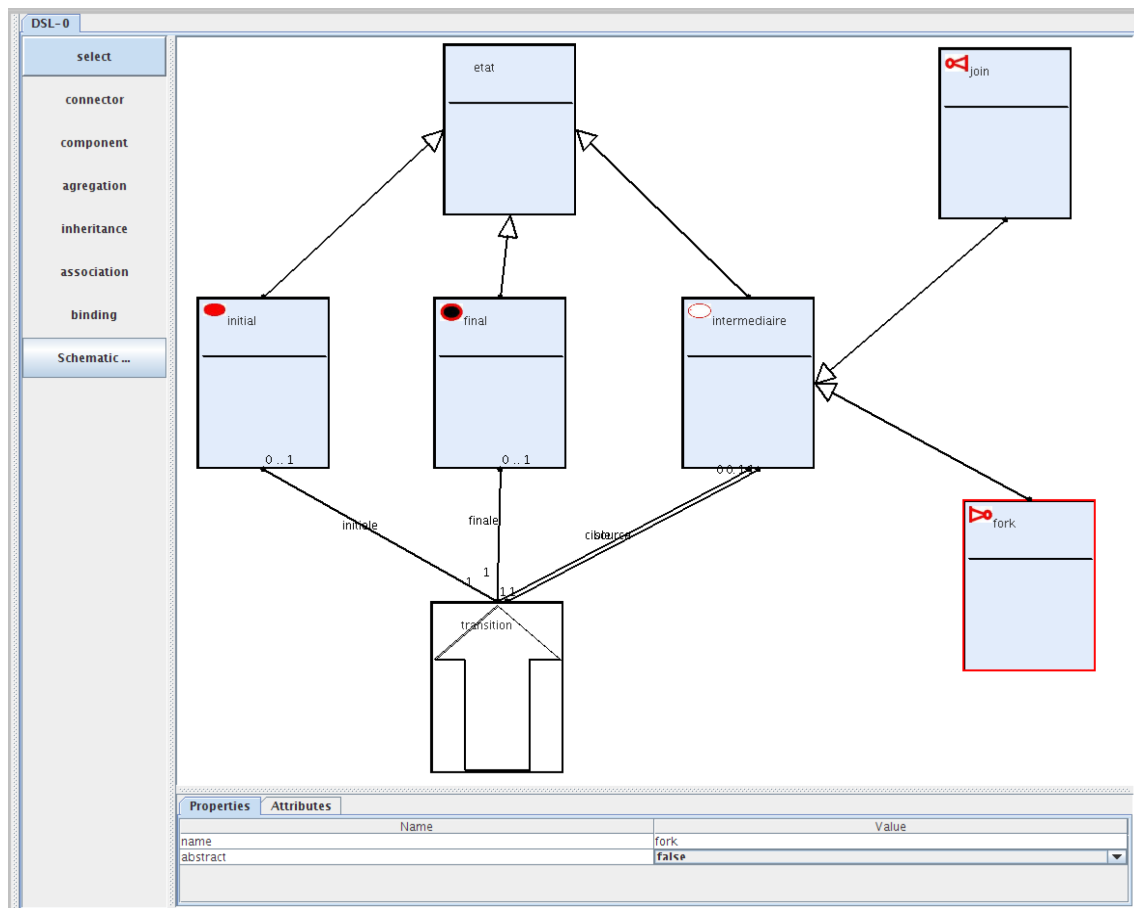


FIGURE 9.5 – *Reconfiguration Description Language*

- **Tâche** : les tâches sont les éléments de base du projet. Pour définir correctement les tâches, certaines conditions sont à observer :
  - une tâche décrit une action ou un événement, à entreprendre. Son libellé doit être clair et précis.
  - une tâche doit avoir des limites chronologiques bien définies. La durée est généralement exprimée en jours ouvrés, il s'agit de la durée attendue de la tâche telle qu'un expert sera en mesure de l'apprécier.
  - une tâche doit être associée à un responsable assumant la responsabilité de l'exécution.
- **Étape** : Il est possible de faire apparaître sur le planning des événements importants autre que les tâches elles-mêmes, constituant des points d'accroche pour le projet. Ces événements sont appelés étapes.
- **Ressource** : pour exécuter une tâche, il est nécessaire de faire appel à des ressources. On distingue deux types de ressources : les ressources matérielles et les ressources humaines. Aux ressources matérielles sont associées les notions de capacité, de coût et de disponibilité. Aux ressources humaines sont associées les notions de qualification, de coût, de disponibilité.

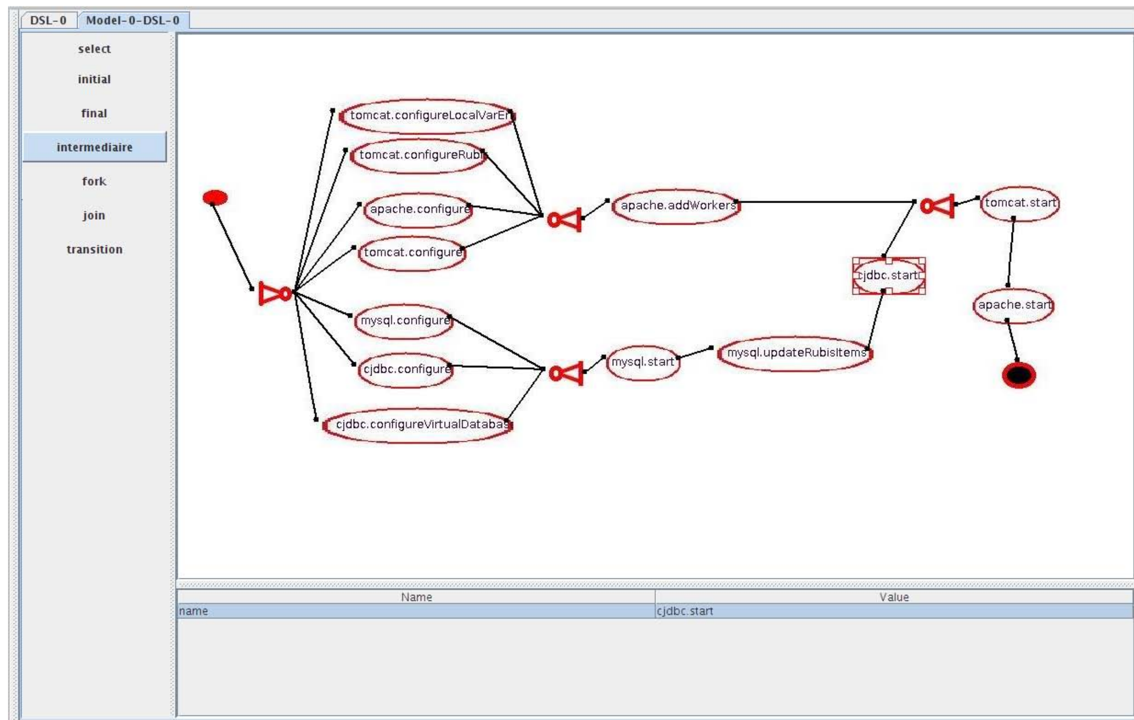


FIGURE 9.6 – Exemple de reconfiguration d'une application JEE

9.3.1 Création de l'outil de méta-modélisation

La définition du langage de méta-modélisation suit les étapes vues au chapitre précédent à savoir : définition de la syntaxe abstraite, définition de la syntaxe concrète et enfin définition de la sémantique.

Définition de la syntaxe abstraite

Ce langage définit trois concepts de base qui sont :

– *project* : va représenter le concept projet ;
– *task* : va représenter le concept tâche ;
– *step* : va représenter le concept étape ;
– *resource* : va représenter le concept ressource ;

Nous avons besoin de relier une tâche à des événements ou à un projet pour marquer les limites temporelles de cette tâche. Cette liaison est liée à la notion de temporalité et va infléchir l'enchaînement des tâches. De même, nous avons besoin d'une relation permettant d'associer une ressource ou un événement à un projet. Cette deuxième relation a une sémantique différente de la première, et est liée à la notion d'appartenance. Nous avons défini deux concepts pour représenter ces deux types de relation et un troisième

concept qui permettra qu'un concept de base puisse hériter d'un autre. Il s'agit des trois concepts suivants :

– *association* : permet de spécifier le début et la fin d'une tâche ;
– *agregation* : permet d'associer une ressource ou une étape à un projet ;
– *inheritance* : qui va permettre qu'un concept puisse hériter d'un concept du même type.

La Figure 9.7 présente l'architecture Fractal que nous obtenons après la création de tous ces concepts.

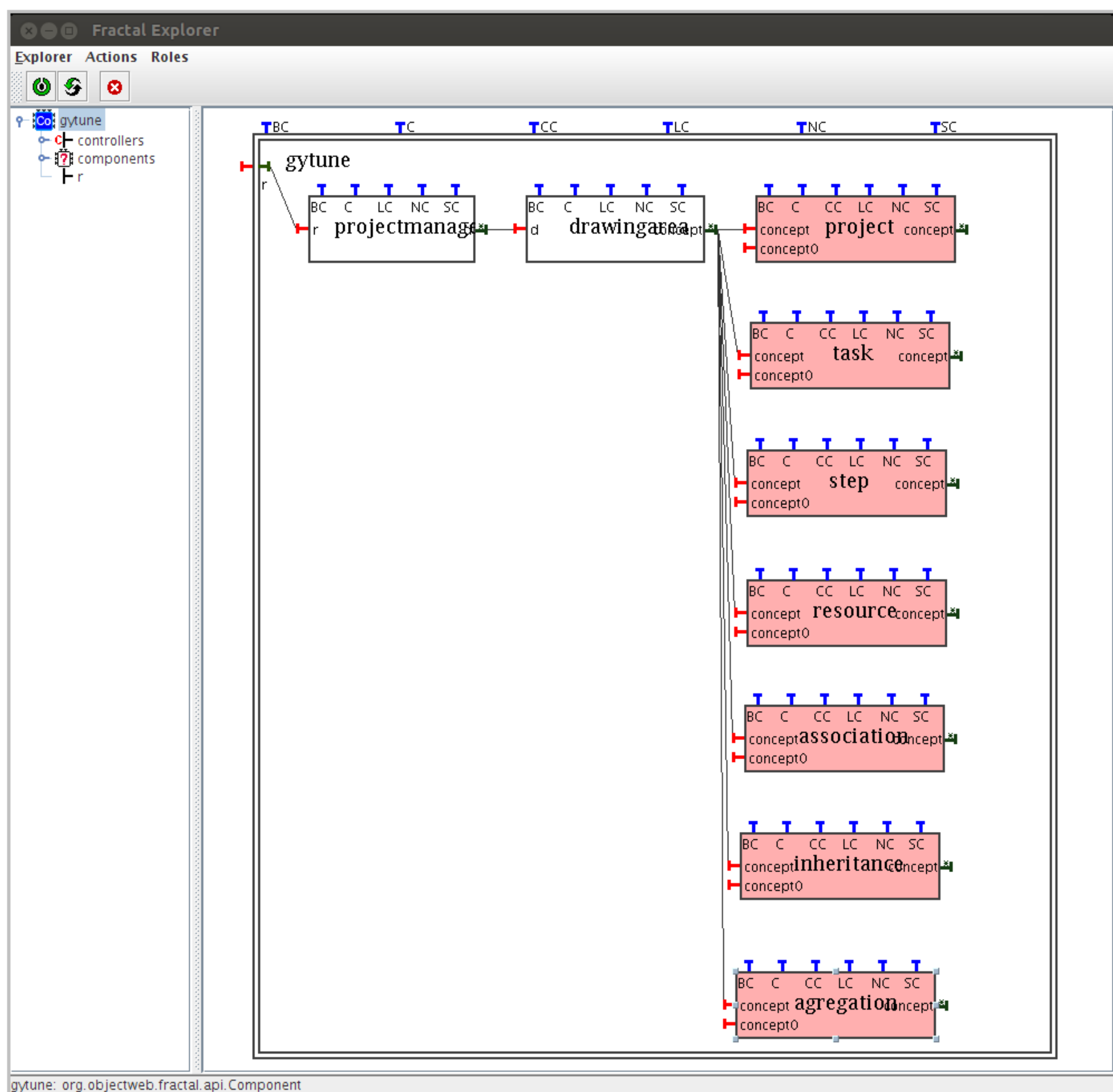


FIGURE 9.7 – Syntaxe abstraite

Définition de la syntaxe concrète et de la sémantique

***project* :**

- **Syntaxe concrète** :
  - **propriétés** : un *project* a une propriété *name* permettant de lui assigner un nom. Il a également une propriété *start* et une propriété *end* de type date qui permettront de définir le début et la fin d'un projet. Ces trois propriétés sont présentes au niveau M2 et au niveau M1 ;
  - **représentation graphique au niveau M2** : la représentation graphique d'un *project* au niveau M2 est un rectangle divisé en deux sections. La valeur de sa propriété *name* est affichée dans la section supérieure ;
  - **représentation graphique au niveau M1** : par défaut, au niveau M1 un project sera représenté par une échelle de temps allant de la date de début du project à sa date de fin. Toutefois, la représentation d'un projet pourra être spécialisée à l'aide de l'éditeur de dessins.
- **Sémantique** :
  - **sémantique niveau M2** : pas de sémantique particulière, juste la vérification des valeurs de ses propriétés ;
  - **sémantique niveau M1** : pas de sémantique particulière, juste la vérification des valeurs de ses propriétés.

***task* :**

- **Syntaxe concrète** :
  - **propriétés** : tout comme *project*, une *task* a les propriétés *name*, *start* et *end* qui seront présentes au niveau M1 et M2. En plus, au niveau M1, une *task* a une propriété *person in charge* permettant de lui assigner un responsable ;
  - **représentation graphique au niveau M2** : comme pour un *project*, la représentation graphique d'une *task* au niveau M2 est un rectangle, mais de couleur différente ;
  - **représentation graphique au niveau M1** : la représentation graphique par défaut d'une *task* au niveau M1 est un fil qui va du début de la *task* à sa fin. Cette représentation peut toutefois être spécialisée avec l'éditeur de dessins.
- **Sémantique** :
  - **sémantique niveau M2** : pas de sémantique particulière au niveau M2, juste la vérification des valeurs de ses propriétés ;
  - **sémantique niveau M1** : au niveau M1, pour valider la création d'une *task*, on vérifie qu'au niveau M2 cette *task* a été prévue entre les concepts correspondants (vérification de la conformité du modèle avec le méta-modèle).

***step* :**

- **Syntaxe concrète** :
  - **propriétés** : un *step* a une propriété *name* au niveau M2 et une propriété *name* au niveau M1 ;
  - **représentation graphique au niveau M2** : sa représentation graphique au niveau M2 est un rectangle comme pour un *project* ou une *task* mais de couleur différente ;
  - **représentation graphique au niveau M1** : La représentation graphique d'un *step* au niveau M1 sera définie par userM2 à l'aide de l'éditeur de dessin du *DrawingArea*.
- **Sémantique** :
  - **sémantique niveau M2** : pas de sémantique particulière, juste la vérification des valeurs des propriétés ;
  - **sémantique niveau M1** : pas de sémantique particulière, juste la vérification des valeurs des propriétés.

***resource* :**

- **Syntaxe concrète** :
  - **propriétés** : au niveau M2 une *resource* a une propriété *name*. Au niveau M1, elle a plusieurs propriétés telles que *type* dont la valeur peut être soit matériel soit humain, *disponibility*, *capacity*, *role* … ;
  - **représentation graphique au niveau M2** : sa représentation graphique au niveau M2 est un rectangle comme pour un *project* ou une *task* mais de couleur différente ; ;
  - **représentation graphique au niveau M1** : La représentation graphique d'un *step* au niveau M1 sera définie par userM2 à l'aide de l'éditeur de dessin du *DrawingArea*..
- **Sémantique** :
  - **sémantique niveau M2** : pas de sémantique particulière, juste la vérification des valeurs des propriétés ;
  - **sémantique niveau M1** : pas de sémantique particulière, juste la vérification des valeurs des propriétés.

***association* :**

- **Syntaxe concrète** :
  - **propriétés** : l'*association* a quatre propriétés qui définissent ses cardinalités sur sa source et sur sa destination. Ces cardinalités permettent de limiter le nombre de *task* sur un *step* ou sur un *project* ;
  - **représentation graphique au niveau M2** : la représentation graphique d'une *association* au niveau M2 est un fil ;
  - **représentation graphique au niveau M1** : pas de sémantique particulière.
- **Sémantique** :

  - **sémantique niveau M2** : au niveau M2 une *association* n'est autorisée qu'entre un *step* et une *task* ou entre un *project* et une *task* ;
  - **sémantique niveau M1** : l'*association* ne sera pas présente au niveau M1.

***agregation* :**

- **Syntaxe concrète** :
  - **propriétés** : le concept *agregation* a une propriété *minFrom* et une propriété *maxFrom* qui représentent les cardinalités d'une *agregation* au niveau M2 et qui permettent de limiter le nombre de connecteurs liés à un composant ;
  - **représentation graphique au niveau M2** : la représentation graphique d'une *agregation* au niveau M2 est un fil avec un losange au bout ;
  - **représentation graphique au niveau M1** : l'*agregation* n'a pas de représentation graphique au niveau M1, mais permet d'intégrer la représentation d'un *step* ou d'une *resource*, sur celle d'un *project* (la représentation du *step* ou de la *resource* est affichée sur celle du *project*).
- **Sémantique** :
  - **sémantique niveau M2** : au niveau M2 une *agregation* n'est autorisée que si sa cible est un *project*. Elle traduit la relation d'appartenance des concepts à un projet ;
  - **sémantique niveau M1** : pour mettre une *agregation* entre un *step* ou une *resource* et un *project* au niveau M1, on vérifie qu'au niveau M2 cette *agregation* a été prévue entre ces éléments là.

***inheritance* :**

- **Syntaxe concrète** :
  - **propriétés** : une *inheritance* n'a pas de propriétés ;
  - **représentation graphique au niveau M2** : elle sera représentée au niveau M2 par un fil avec un triangle à une extrémité ;
  - **représentation graphique au niveau M1** : l'*inheritance* n'a pas de représentation au niveau M1.
- **Sémantique** :
  - **sémantique niveau M2** : une *inheritance* n'est autorisée qu'entre deux concepts de base et de même type (deux *components* ou deux *connectors* ou deux *bindings*). Lorsqu'un concept c1 hérite d'un concept c2, les attributs de c2 définis par userM2 sont ajoutés à c1. Au niveau modèle les liaisons autorisées pour c2 le seront également pour c1 ;
  - **sémantique niveau M1** : pas de sémantique particulière.

Exemple diagramme de GANTT

Le diagramme de GANTT est un outil permettant de modéliser la planification de tâches nécessaires à la réalisation d'un projet. Il s'agit d'un outil inventé en 1917 par un américain Henry L. GANTT. Étant donné la relative facilitée de lecture des diagrammes GANTT, cet outil est utilisé par la quasi-totalité des chefs de projet dans tous les secteurs. Le diagramme GANTT représente un outil pour le chef de projet, permettant de représenter graphiquement l'avancement du projet, mais c'est également un bon moyen de communication entre les différents acteurs d'un projet.

La Figure 9.8 représente le méta-modèle du diagramme de GANTT conçu à l'aide de l'outil de méta-modélisation dédié à la gestion de projet que nous avons créé. Ce méta-modèle fait intervenir un concept *projet* qui permettra d'établir le calendrier de toutes les tâches à effectuer ; un concept *tâche*, qui représentera les tâches à effectuer. La durée d'exécution d'une tâche sera matérialisée par une barre horizontale, représentation que nous avons assignée à l'aide de l'éditeur de dessin du *DrawingArea*. Un second concept tâche appelé *précédence* permettra de représenter les liens de dépendance entre les tâches. Il est fréquent de matérialiser la *précédence* par des flèches, la flèche relie la tâche précédente à la tâche suivante. Un concept *step* appelé *jalon* va permettre de marquer les événements importants du projet.

La figure 9.9 présente un exemple de diagramme de GANTT conforme à ce langage. Ce diagramme détaille les étapes de la préparation d'une publication. La tâche de rédaction commence dès que la phase de préparation s'achève. La relecture commence quelque temps après le début de la rédaction, et se déroule en parallèle avec elle. La création des illustrations est liée à la tâche de rédaction. La tâche de vérification finale est liée à la création des illustrations, mais peut toutefois commencer avant la fin de cette tâche. L'impression ne peut démarrer qu'une fois la brochure finalisée. Enfin, la livraison du papier doit être terminée avant le début de l'impression.

Exemple graphe de PERT

La méthode de PERT, qui est l'acronyme de *program* (ou *project*) *evaluation and review technique*, ce qui signifie "technique d'évaluation et d'examen de programmes " ou " de projets ", a été créé en 1956 à la demande de la marine américaine, qui veut planifier la durée de son programme de missiles balistiques nucléaires Polaris. L'étude est réalisée par la société de conseil en stratégie Booz Allen Hamilton. Alors que le délai initial de ce programme – qui a fait intervenir 9000 sous-traitants et 250 fournisseurs – était de 7 ans, l'application de la technique PERT a permis de réduire ce délai à 4 ans. Contrairement à celle du GANTT, la méthode PERT s'attache surtout à mettre en évidence les liaisons qui existent entre les différentes tâches d'un projet et à définir le chemin dit "critique". Le graphe de PERT est composé d'étapes et de tâches (ou opérations). On représente les tâches par des flèches. La longueur des flèches n'a pas de signification ; il n'y a pas de proportionnalité dans le temps.

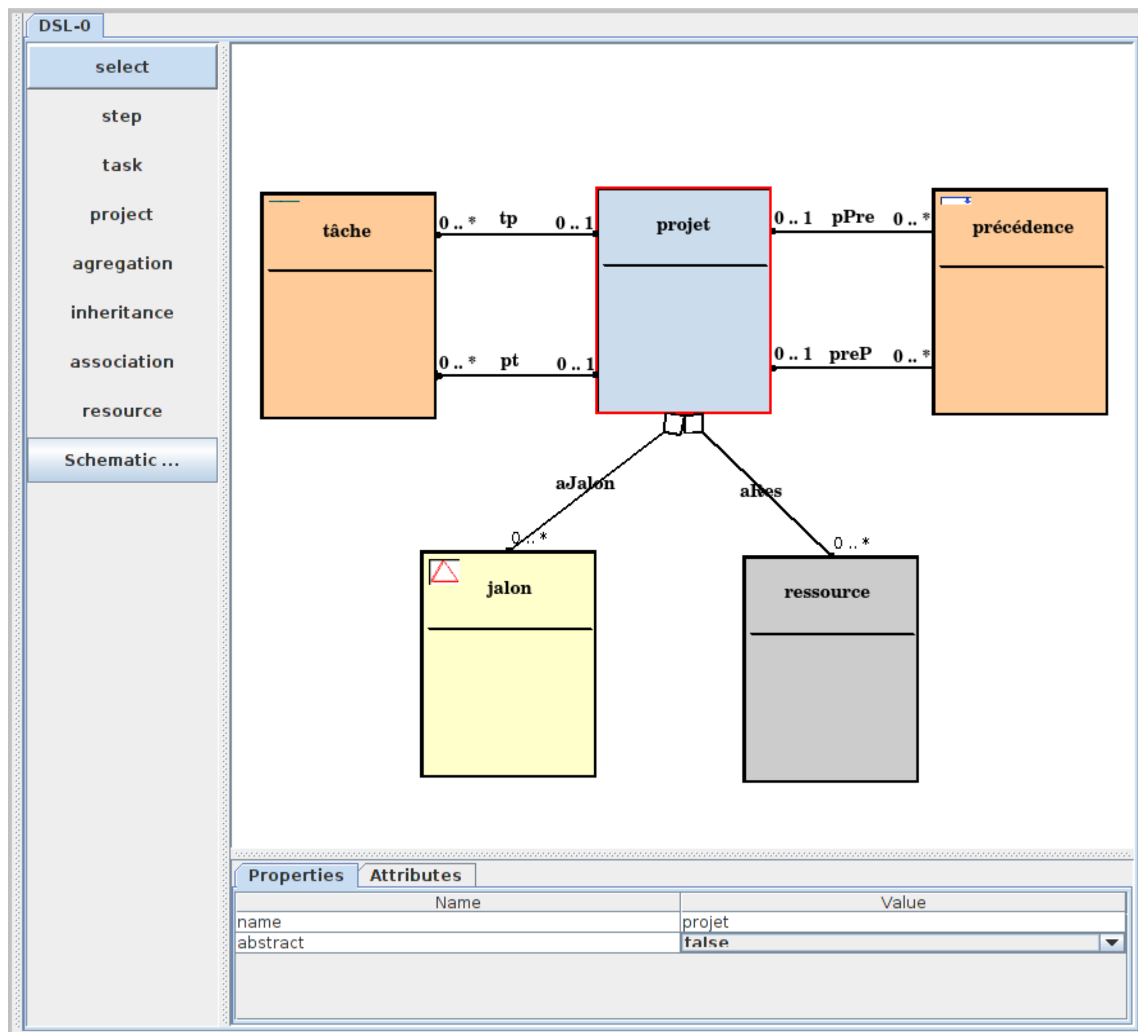


FIGURE 9.8 – Méta-modèle diagramme de gantt

La Figure 9.10 représente le méta-modèle des graphes de PERT conçu à l'aide de l'outil de méta-modélisation dédié à la gestion de projet que nous avons créé. Ce langage définit en fait des graphes de dépendance. Chaque tâche (*activité*) a une date de début et une date de fin et une tâche peut être *fictive*. L'*étape* représente le début ou la fin d'une ou de plusieurs tâches. Une *étape* a un numéro, une date de début au plus tôt et une date de fin au plus tard. On appelle donc diagramme de PERT l'ensemble des étapes et des tâches qui forment un projet. Les diagrammes créés auront pour but de déterminer le chemin critique qui conditionne la durée minimale d'un *projet*.

Un diagramme de PERT conforme à ce langage est présenté à la Figure 9.11. Les cercles représentent les étapes et les flèches représentent les tâches, les tâches fictives étant représentées par des flèches discontinues. À l'intérieur de chaque cercle : en haut le numéro de l'étape ; en bas à gauche la date de début au plus tôt qui est la date minimum à laquelle une étape peut être totalement validée ; en bas à droite la date de fin au plus tard qui est la date maximum à laquelle une étape doit être validée si l'on ne veut pas retarder

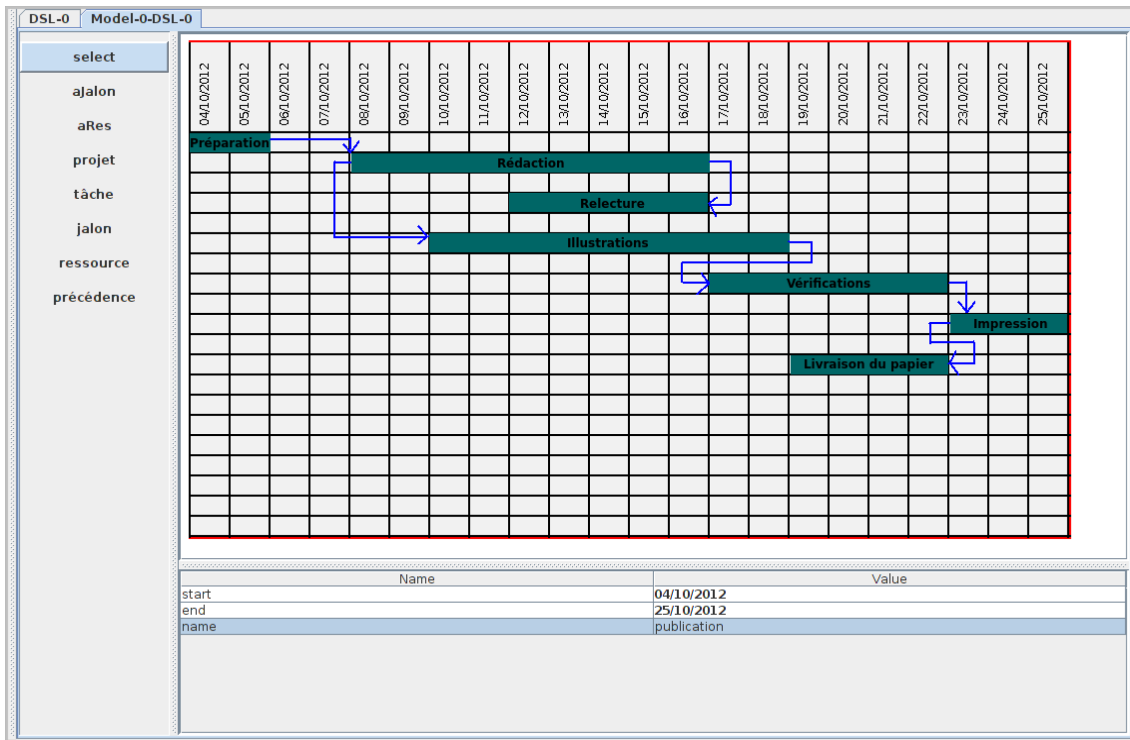


FIGURE 9.9 – Modèle diagramme de gantt

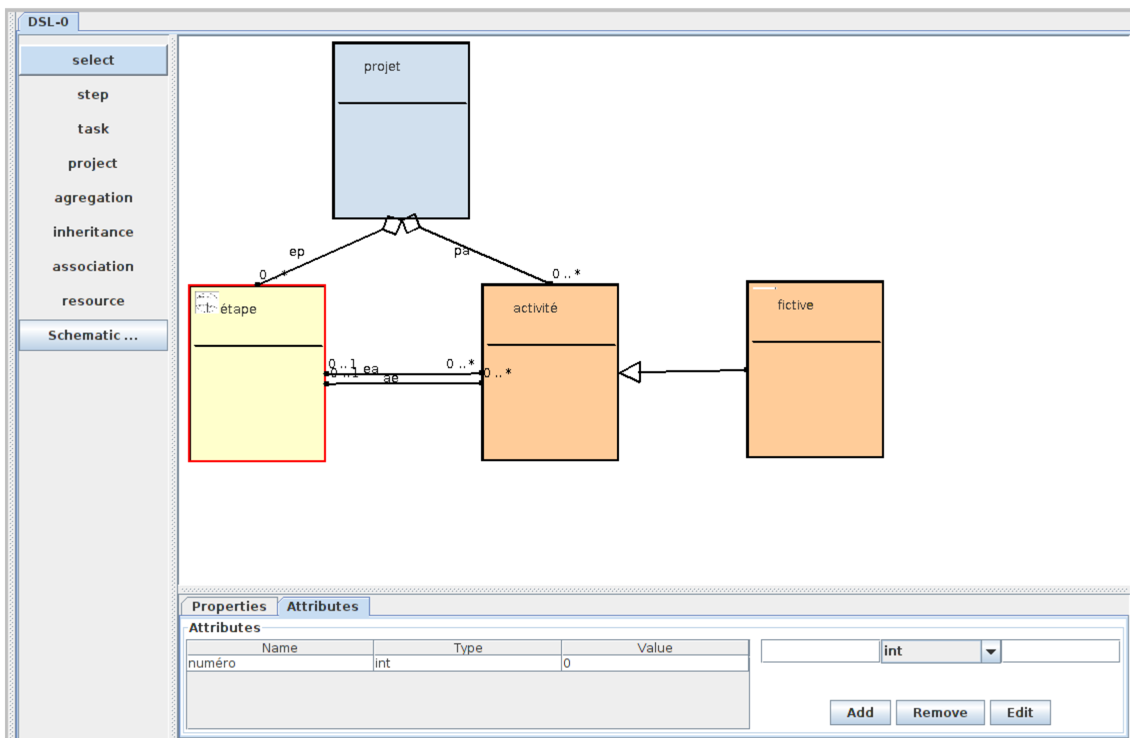


FIGURE 9.10 – Méta-modèle graphe de PERT

le projet. Le diagramme possède une seule étape de début et une seule étape de fin. Le diagramme se lit de la gauche vers la droite et les flèches sont orientées dans ce sens.

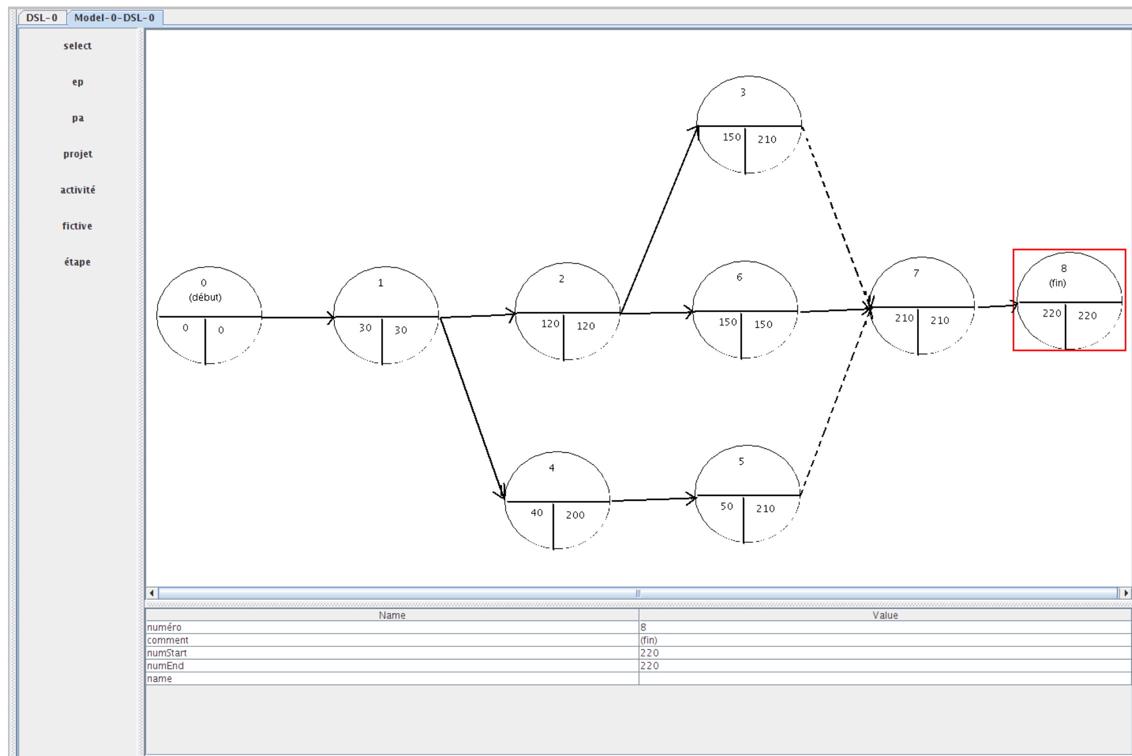


FIGURE 9.11 – Modèle graphe de PERT

9.4 Synthèse

Dans ce chapitre nous avons présenté deux cas d'utilisation de notre plate-forme GyTUNe. Ces deux cas d'utilisation ont montré qu'il est possible de concevoir une plate-forme générique qui permette de créer des outils de méta-modélisation en fonction du domaine d'application. À chaque cas d'utilisation, nous avons assigné une syntaxe et une sémantique spécifiques au langage de méta-modélisation. On obtient dans les deux cas un outil de méta-modélisation simple d'utilisation et suffisamment intuitif pour un utilisateur du domaine ciblé. Certes implanter la partie spécifique de GyTUNe nécessite quand même une bonne expertise en programmation, mais l'utilisateur aura dans l'ensemble beaucoup moins de fonctionnalités à implanter lui même parce que celles-ci sont prises en charge par GyTUNe. De plus, la réutilisation des composants déjà implantés ou de certaines de leurs méthodes contribue à réduire le coût de développement.
Pour donner quelques chiffres, nous avions entièrement développé un outil de méta-modélisation similaire en tout point à celui présenté dans notre premier cas d'utilisation. Cet environnement s'appelle yTUNe. Comme le présente le Tableau 9.1, pour développer yTUNe entièrement du début à la fin il nous aura fallu 13111 lignes de code . Tandis que développer le même outil avec exactement les mêmes fonctionnalités sous GyTUNe aura nécessité 3052 lignes de code (une bonne partie étant du code généré par Eclipse à partir des interfaces à implémenter). La partie *common* de GyTUNe aura donc fourni 80% des fonctionnalités de yTUNe. Cette expérience montre qu'il est possible de réduire forte-

Common	**8864**
GyTUNe spécifique au domaine des composants	**3052**
yTUNe	**13111**
GyTUNe spécifique à la gestion de projets	**3104**

TABLE 9.1 – Répartition des lignes de code

ment le coût de production d'un outil de méta-modélisation spécifique l'objectif final étant d'arriver à tirer pleinement profit des avantages des langages dédiés de modélisation.

Chapitre 10

Conclusion et perspectives

10.1 Conclusion

Nous menons depuis quelques années un projet qui vise la construction d'un système autonome pour l'administration de grandes infrastructures logicielles. Système que nous avons appelé TUNe. TUNe permet d'automatiser et ce faisant d'optimiser l'administration des environnements logiciels. L'administrateur humain est libéré de ces tâches automatisées souvent laborieuses et répétitives et peut désormais mettre à profit son expertise en se concentrant sur des tâches d'administration de plus haut niveau. Ces tâches de plus haut niveau incluent principalement la définition des politiques d'administration. TUNe fournit des langages permettant de définir ces politiques d'administration, mais les expériences réalisées ont montré l'inadéquation des langages de TUNe lorsqu'on élargit le spectre des applications visées. En effet la conception des langages d'administration de TUNe était initialement orientée vers l'administration des applications *clusterisées* de type client-serveur telle que JEE. Lors des expérimentations on a pu constater que lorsqu'on change de famille d'application (large échelle, machines virtuelles . . .), ou de facette d'administration (gestion de performances, équilibrage de charge . . .), les langages de TUNe deviennent inadaptés, il faut alors soit les modifier soit en créer des nouveaux. Ces expérimentations ont révélé la variation des besoins d'administration en fonction des domaines d'application. Les langages d'administration doivent être spécifiques à la facette d'administration considérée et au domaine d'application. De tels langages conçus pour résoudre des problèmes d'un domaine particulier sont appelés langages dédiés.
Nous avons donc orienté le projet TUNe afin de proposer un système d'administration qui ne repose sur aucun langage ou domaine d'administration et avons souhaité fournir au-dessus de ce système un ensemble d'outils permettant de définir les langages dédiés correspondants aux besoins précis des administrateurs. Notre orientation consiste alors à utiliser des techniques éprouvées de l'Ingénierie Dirigée par les Modèles (IDM) pour implanter des langages dédiés dans TUNe. Cela a constitué la motivation principale de cette thèse où nous nous intéressons de façon plus générale à la définition et à l'implantation des langages dédiés graphiques (communément appelés DSML) dans l'ingénierie du logiciel.

Les DSMLs sont un moyen reconnu pour permettre aux experts d'un domaine de concevoir des applications sans êtres experts en programmation. Restreints à un domaine, ils permettent de factoriser toute la connaissance relative à ce domaine dans le langage en utilisant un haut niveau d'abstraction. Cette factorisation permet d'améliorer la productivité du développement logiciel. La principale limitation qui freine l'adoption à grande échelle des DSMLs est le manque d'outils appropriés permettant de les implanter. En effet, les outils existants que nous appelons outils de méta-modélisation sont trop génériques et nécessitent une grande expertise en programmation pour arriver à implanter un DSML. Implanter un DSML revient généralement à fournir un éditeur dédié qui permette aux utilisateurs de manipuler les abstractions du domaine ciblé par le DSML (d'instancier le langage). La plupart des outils de méta-modélisation existants offrent des assistants permettant d'automatiser le processus de génération des éditeurs dédiés. Mais ces assistants ne sont efficaces que pour des cas très généralistes et au final ne sont pas satisfaisants dans la plupart des cas. Nous proposons donc dans cette thèse de spécialiser les outils de méta-modélisation en fonction des domaines, de la même façon que les langages pour être plus productifs sont spécialisés en fonction des domaines.
Mais développer un outil de méta-modélisation par domaine peut être coûteux et l'ampleur de la tâche peut très vite décourager ceux qui en ont besoin. Pour réduire ce coût, nous avons proposé une plate-forme générique permettant de créer des outils de méta-modélisation dédiés en fonction du domaine applicatif. Notre plate-forme appelée GyTUNe repose sur le modèle à composant fractal et permet de définir les outils de méta-modélisation sous forme d'architecture à composants. Chaque composant peut alors être adapté, configuré ou remplacé en fonction des spécificités d'un domaine et on peut ainsi créer un nouvel outil de méta-modélisation sans avoir besoin de le développer entièrement.
Nous avons appliqué GyTUNe à deux domaines. Un domaine technique qui est celui des architectures à composants et sur un domaine métier qui est celui de la gestion de projets. Dans chacun des cas d'utilisation, nous avons pu créer un outil de méta-modélisation qui intègre bien les spécificités du domaine ciblé. L'outil dédié aux architectures à composants nous a permis de définir des langages d'administration (ADL et RDL) pour une application JEE. Nous avons comparé cet outil à un autre (yTUNe) qui lui a été défini entièrement sans utiliser GyTUNe, les deux outils ayant exactement les mêmes fonctionnalités. L'outil créé à l'aide de GyTUNe aura nécessité cinq fois moins de lignes de code que yTUNe. L'outil dédié à la gestion de projets nous a permis de définir un langage pour la construction des diagrammes de GANTT ainsi qu'un langage pour la construction des diagrammes de PERT. Ces expériences nous ont permis de montrer qu'il est possible de réduire les coûts de production d'un outil de méta-modélisation dédié et ce faisant de favoriser l'utilisation des DSMLs pour une meilleure productivité.

10.2 Perspectives

GyTUNe est une plate-forme récente qui requiert des évolutions. Tout au long de la réalisation de cette thèse, nous avons identifié différents axes potentiels de prolongement de nos travaux. Ces axes peuvent êtres classés en deux catégories : les travaux réalisables

dans un futur proche, et ceux réalisables dans un futur plus lointain compte tenu de l'ampleur de la tâche.

10.2.1 Perspectives à court terme

Abstraire Fractal

Une des limitations de GyTUNe aujourd'hui est que les utilisateurs doivent connaître Fractal et plus précisément son implémentation en java. C'est une contrainte majeure parce que l'utilisateur même s'il doit être un expert en programmation n'aura pas forcément des compétences en Fractal. Une perspective à court terme consisterait par exemple à proposer des outils graphiques tels que *FractalGUI* qui permet de concevoir des architectures Fractal de façon graphique et d'en générer le squelette de code. De tels outils seraient plus expressifs et moins contraignants pour l'utilisateur.

Assistants au niveau M3

On peut se poser la question d'aller encore plus loin et de pousser le raisonnement jusqu'au niveau M3. On pourrait non seulement envisager de masquer Fractal mais de façon générale essayer de fournir des supports graphiques aux userM3 pour la création des outils de méta-modélisation dédiés. Si abstraire Fractal à l'aide des outils graphiques tels que *FractalGUI* relève du domaine du possible, permettre aux userM3 de définir entièrement leurs langages graphiquement nous semble moins réalisable.

Langage de contraintes plus évolué

Dans l'état actuel de GyTUNe, les contraintes qui doivent être vérifiées au niveau M2 et au niveau M1 sont définies en Java ce qui n'est pas très approprié. Il faudrait proposer un langage de contraintes tel que OCL qui permettrait de définir les contraintes de façon plus simple.

Interopérabilité

Dans l'état actuel, GyTUNe n'est pas interopérable avec d'autres outils de méta-modélisation. Il n'est par exemple pas possible d'importer ou d'exporter des méta-modèles ou des modèles créés avec un autre outil. Pour assurer l'interopérabilité avec d'autres outils de méta-modélisation, il faudrait que les outils créés à l'aide de GyTUNe utilisent au

maximum les standards existants. Pour que GyTUNe s'intègre parfaitement dans un processus de développement logiciel, il faudrait qu'il soit interopérable avec les outils utilisés en amont ou en aval dans le processus.

Génération de code

À partir des modèles créés au niveau M1, on pourrait générer un squelette de code comme le font les outils tels que Kermeta [Drey and Vojtisek 2006], ce code servirait alors de base à la simulation ou à l'exécution des modèles.

10.2.2 Perspectives à long terme

Composition des modèles

Selon le domaine, il peut être intéressant de fournir plusieurs DSLs conçus pour différentes catégories d'utilisateurs et il peut arriver que ces différents DSLs soient nécessaires pour spécifier complètement une application ou un domaine complexe, ces DSLs peuvent également partager des concepts. Il est donc important qu'une plate-forme telle que GyTUNe permette de créer des outils qui favorisent la composition des modèles. Le besoin de composition des langages fait l'objet d'importantes recherches autour des thématiques telles que les méga-modèles [Bézivin et al. 2004].

Tester GyTUNe sur un large panel de domaines différents

Pour que notre plate-forme soit suffisamment générique, il faudrait la faire tester par des utilisateurs d'un grand nombre de domaines différents. Les retours d'expériences permettraient d'affiner ou de grossir le degré de généricité de la partie *common* de GyTUNe. Ces tests/validation doivent se faire sur plusieurs années.

Étude de la rentabilité par domaine

Dans certains cas, créer un environnement de méta-modélisation peut être coûteux et pas rentable. En fait, avant de décider ou non de la création d'un environnement de méta-modélisation qui soit dédié à un domaine, il faudrait préalablement mener une étude permettant de déterminer si son degré de spécificité nécessite qu'il faille créer cet environnement, ou au contraire si utiliser un outil de méta-modélisation existant même complexe ne serait pas plus rentable. Une telle étude nécessite également une utilisation à grande échelle pouvant prendre plusieurs années.

Bibliographie

(2012). Dsm forum : Domain-specific modeling. http://www.dsmforum.org.

alain Muller, P., Fleurey, F., and marc Jézéquel, J. (2005). Weaving executability into object-oriented meta-languages. In *in : International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713 (2005*, pages 264–278. Springer.

Amyot, D., Farah, H., and Roy, J. F. (2006). Evaluation of development tools for domain-specific modeling languages. *System Analysis and Modeling : Language Profiles*, pages 183–197.

Apache (200x). Apache http server project. http://httpd.apache.org/.

Arnold, B., Deursen, A. V., and Res, M. (1995). An algebraic specification of a language for describing financial products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13. IEEE.

Badia, S. and Nussbaum, L. (2011). gLite sur Grid'5000 : vers une plate-forme d'expérimentation à taille réelle pour les grilles de production. In *Rencontres scientifiques France Grilles*, Lyon, France.

Bakshi, K. (2009). Cisco cloud computing - data center strategy, architecture, and solutions point of view white paper for u.s. public sector 1st edition. http ://www.cisco.com/web/strategy/docs/gov/CiscoCloudComputing_WP.pdf.

Barroca, B., Lucio, L., Buchs, D., Amaral, V., and Pedro, L. (2009). Dsl composition for model-based test generation. *ECEASST*, 21.

Batory, D., Thomas, J., and Sirkin, M. (1994). Reengineering a complex application using a scalable data structure compiler. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '94, pages 111–120, New York, NY, USA. ACM.

Bentley, J. (1986). Programming pearls : little languages. *Commun. ACM*, 29(8) :711–721.

Berg, A. (2008). Separate the static from the dynamic with Tomcat and Apache. *Linux J.*, 2008(165) :9.

Berrayana, T. A. (2006). *Apport des architectures à composants pour l'administration des intergiciels Étude de cas : JonasALaCarte, un serveur d'applications J2EE administrable*. Thèse de doctorat, Institut National Polytechnique de Grenoble.

Bevan, N. and Macleod, M. (1994). Usability measurement in context. *Behaviour and Information Technology*, 13 :132–145.

Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., and Piers, W. (2005). Bridging the MS/DSL tools and the eclipse modeling framework. In *Proceedings of the International Workshop on Software Factories at OOPSLA*.

Bézivin, J., Jouault, F., and Valduriez, P. (2004). On the need for megamodels. In *Proceedings of the OOPSLA/GPCE : Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

Biegl, C. (1995). Multigraph : an architecture for model-integrated computing. In *Proceedings of the 1st International Conference on Engineering of Complex Computer Systems*, ICECCS '95, pages 361–, Washington, DC, USA. IEEE Computer Society.

Bouchenak, S., Boyer, F., Krakowiak, S., Hagimont, D., Mos, A., Jean-Bernard, S., de Palma, N., and Quema, V. (2005). Architecture-based autonomous repair management : An application to j2ee clusters. In *SRDS '05 : Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, pages 13–24, Washington, DC, USA. IEEE Computer Society.

Broto, L. (2008). *Support langage et système pour l'administration autonome*. Thèse de doctorat, l'Université Toulouse 3.

Broto, L., Hagimont, D., Stolf, P., Depalma, N., and Temate, S. (2008). Autonomic management policy specification in Tune. In *Annual ACM Symposium on Applied Computing (SAC), Fortaleza, CearÃ¡, Brazil*. ACM.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The fractal component model and its support in java. In *Software - Practice and Experience, special issue on "Experiences with Auto-adaptive and Reconfigurable Systems", 36(11-12) :1257-1284*.

Clark, A., Sammut, P., and Willans, J. (2008). *Superlanguages : Developing Languages and Applications with XMF*. 1 edition.

Clark, T., Evans, A., Sammut, P., and Willans, J. (2004). *Applied Metamodelling - A Foundation for Language Driven Development*. Second edition.

Clarke, S. (2001). *Composition of Object-Oriented Software Design Models*. Thèse de doctorat, School of Computer Applications Dublin City University.

Combemale, B. (2008). *Approche de métamodélisation pour la simulation et la vérification de modèle*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, France.

Combemale, B., Broto, L., Crégut, X., Daydé, M., and Hagimont, D. (2008). Autonomic management policy specification : From uml to dsml. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 584–599, Berlin, Heidelberg. Springer-Verlag.

Company, O. M. D. (200x). Obeo Designer white paper. http://www.obeo.fr/resources/FicheProduit-OD.pdf.

Consel, C. (2003). From a program family to a domain-specific language. In *Domain-Specific Program Generation*, pages 19–29.

Czarnecki, K. and Eisenecker, U. (2000). *Generative programming : methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA.

Davis, J. (2003). Gme : the generic modeling environment. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 82–83, New York, NY, USA. ACM.

Deursen, A. V. (1998). Little languages : Little maintenance ? page 19.

DIET (200x). The distributed interactive engineering toolbox. http://graal.ens-lyon.fr/~diet/.

Drey, Z. and Vojtisek, D. (2006). *Kermeta EMF tutorial*.

eclipse (2012). Graphiti - a graphical tooling infrastructure. http://eclipse.org/graphiti/.

El Kouhen, A., Dumoulin, C., Gerard, S., and Boulet, P. (2012). Evaluation of Modeling Tools Adaptation. Technical report.

Evans, A., Fernández, M. A., and Mohagheghi, P. (2009). Experiences of developing a network modeling tool using the eclipse environment. In *ECMDA-FA '09 : Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, pages 301–312, Berlin, Heidelberg. Springer-Verlag.

Fleurey, F. (2006). *Langage et Méthode pour une ingenierie des modèles fiable*. Thèse de doctorat, Université de Rennes 1.

Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley Professional.

Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., and Steenkiste, P. (2004). Rainbow : Architecture-based self adaptation with reusable infrastructure. In *IEEE Computer, 37(10)*.

Garshol, L. M. (2008). Bnf and ebnf : What are they and how do they work ? http://www.garshol.priv.no/download/text/bnf.html.

Gray, J. and Karsai, G. (2003). An examination of dsls for concisely representing model traversals and transformations. In *36th Annual Hawaii International Conference on System Sciences (HICSS'03) - Track 9, p. 325a, January 06 - 09*.

Greenfield, J. and Short, K. (2004). *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, Indianapolis, IN.

Hanson, J. E., Whalley, I., Chess, D. M., and Kephart, J. O. (2004). An architectural approach to autonomic computing. In *ICAC '04 : Proceedings of the First International Conference on Autonomic Computing*, pages 2–9, Washington, DC, USA. IEEE Computer Society.

Harel, D. and Rumpe, B. (2000). Modeling languages : Syntax, semantics and all that stuff, part i : The basic stuff. Technical report, Jerusalem, Israel, Israel.

Herndon, Jr., R. M. and Berzins, V. A. (1988). The realizable benefits of a language prototyping language. *IEEE Trans. Softw. Eng.*, 14(6) :803–809.

ISIS (2012). Gme : Generic modeling environment. http://www.isis.vanderbilt.edu/Projects/gme/.

J. Karna, J-P. Tolvanen, S. K. (2009). Evaluating the use of domain-specific modeling in practice. In *Proceedings of the 9th OOPSLA workshop on Domain-Specific Modeling*.

Jones (2006). Software productivity research, programming languages table. http://www.spr.com.

Kahlaoui, A. (2011). *Méthode pour la définition des langages dédiés basée sur le méta-modèle ISO/IEC 24744*. Thèse de doctorat, Ecole Technologie Supérieure Montréal, Canada.

Karsai, G., Sztipanovits, J., Lédeczi, Á., and Bapty, T. (2003). Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1) :145–164.

Kelly, S. (2004). Comparison of eclipse EMF/GEF and MetaEdit+ for DSM. In *OOPSLA*.

Kolovos, D. S., Rose, L. M., Paige, R. F., and Polack, F. A. C. (2009). Raising the level of abstraction in the development of gmf-based graphical model editors. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, MISE '09, pages 13–19, Washington, DC, USA. IEEE Computer Society.

Kramer, J. (2007). Is abstraction the key to computing ? *Commun. ACM*, 50(4) :36–42.

Latry, F. (2007). *Approche langage au développement logiciel : Application au domaine des services de Téléphonie sur IP*. Thèse de doctorat, Université de Bordeaux 1.

Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001). The generic modeling environment. In *Workshop on Intelligent Signal Processing*.

Merle, P. and Stefani, J.-B. (2008). A formal specification of the Fractal component model in Alloy. Rapport de recherche RR-6721, INRIA.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4) :316–344.

MetaCase (2012). La modélisation métier avec metaedit+. http://www.metacase.com/fr/.

Microsystems, S. (200x). Java platform enterprise edition (j2ee). http://java.sun.com/j2ee/.

Miller, J. and Mukerji, J. (2003). Mda guide version 1.0.1. Technical report, Object Management Group (OMG).

Minas, M. (2012). The diagram editor generator. http://www.unibw.de/inf2/DiaGen/.

ModJK (200x). The apache tomcat connector. http://tomcat.apache.org/connectors-doc/.

Muller, P.-A. (2006). *De la modélisation objet des logiciels à la metamodélisation des langages informatiques*. Habilitation à diriger des recherches, Université de Rennes 1. 102 pages.

Murray-Rust, P. and Rzepa, H. S. (1999). Chemical markup, xml, and the world wide web. 1. basic principles. *Journal of Chemical Information and Computer Sciences*, 39(6) :928–942.

MySQL (200x). Mysql mysql web site. http://www.mysql.com/.

omg (2006). *Meta Object Facility (MOF) Core Specification Version 2.0*.

Oreizy, P., Medvidovic, N., and Taylor, R. N. (1998). Architecture-based runtime software evolution. In *ICSE '98 : Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA. IEEE Computer Society.

Ozgur, T. (2009). *Domain-Specific Modeling : A Practical Approach A comparison of Microsoft DSL Tools and Eclipse Modeling Frameworks in the context of Model-Driven Development*. LAP Lambert Academic Publishing, Germany.

Pelechano, V., Albert, M., Muñoz, J., and Cetina, C. (2006). Building tools for model driven development. comparing microsoft dsl tools and eclipse modeling plug-ins. In *DSDM*.

Pfeiffer Jr, J. J. (1997). A rule-based visual language for small robotic applications. In *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)*, VL '97, pages 162–, Washington, DC, USA. IEEE Computer Society.

Simos, M., Creps, D., Klingler, C., Levine, L., and Allemang, D. (1996). Organization domain modeling (ODM) guidebook version 2.0. Technical report, Software Engineering Institute / Carnegie Mellon University.

Sztipanovits, J. and Karsai, G. (1997). Model-integrated computing. *Computer*, 30(4) :110–111.

Taton, C. (2008). *Vers l'auto-optimisation dans les systèmes autonomes*. Thèse de doctorat, Institut National Polytechnique de Grenoble.

Taylor, R. N., Medvidovic, N., Anderson, K. M., E. James Whitehead, J., and Robbins, J. E. (1995). A component- and message-based architectural style for gui software. In *ICSE '95 : Proceedings of the 17th international conference on Software engineering*.

Tchana, A.-B. (2011). *Système d'administration autonome adaptable : application au cloud*. Thèse de doctorat, Institut National Polytechnique de Toulouse.

Thibault, S. (1998). *Langage Dédiés : Conception, Implémentation et Application*. Thèse de doctorat, Université de Rennes, France.

Tilkov, S. (2009). Thoughts on the generic vs. specific tradeoff. Accessed 20/06/2012 at : http://www.infoq.com/presentations/Generic-Specific-Tradeoffs-Stefan-Tilkov.

Tolvanen, J.-P. (2006). Domain-specific modeling : How to start defining your own language. Article accessible sur le site de l'entreprise à : http://www.devx.com/enterprise/Article/30550.

Tomcat (200x). The apache software foundation apache tomcat. http://tomcat.apache.org/.

TOURE, M. A. (2010). *Administration d'applications réparties à grande échelle*. Thèse de doctorat, Institut National Polytechnique de Toulouse.

van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35(6) :26–36.

Vessey, I. and Glass, R. (1998). Strong vs. weak approaches to systems development. *Commun. ACM*, 41(4) :99–102.

Wagner, S. and Deissenboeck, F. (2008). Abstractness, specificity, and complexity in software design. In *Proceedings of the 2nd international workshop on The role of abstraction in software engineering*, ROA '08, pages 35–42, New York, NY, USA. ACM.

White, J., Hill, J. H., Gray, J., Tambe, S., Gokhale, A. S., and Schmidt, D. C. (2009). Improving domain-specific language reuse with software product line techniques. *IEEE Softw.*, 26(4) :47–53.

Liste des tableaux

4.1 Exemples de langages dédiés de programmation et de modélisation 27

5.1 Générique versus Spécifique 41

6.1 Résumé de EMP 56
6.2 Résumé de DiaMeta 59
6.3 Résumé de VMSDK 63
6.4 Résumé de GME 66
6.5 Résumé de MetaEdit+ 70
6.6 Vue d'ensemble 70

9.1 Répartition des lignes de code 126

Table des figures

1.1 Organisation d'un système d'administration autonome 7

2.1 Exemple d'une architecture JEE 12
2.2 Exemple d'ADL d'une application JEE 14
2.3 Wrapper du logiciel Apache 14
2.4 Configuration et démarrage d'une application JEE 15

3.1 Architecture de DIET 17
3.2 Exemple d'ADL en intension d'une application DIET 17
3.3 Nouvelle orientation du projet TUNe 21

4.1 Niveaux de modélisation 29
4.2 Ingénierie Dirigée par les Modèles 32
4.3 Syntaxe concrète pour des cartes routières 34
4.4 Conception d'un DSML 35

5.1 Outil de méta-modélisation 39

6.1 Modèles à composants 45
6.2 Liaisons spécifiques 47
6.3 Outils de l'IDM 47
6.4 exemple de domain model 51
6.5 Graph model 52
6.6 Tooling model 53
6.7 exemple de mapping model 53
6.8 Éditeur généré 55
6.9 DiaMeta DESIGNER 57
6.10 Éditeur généré avec DiaMeta 58
6.11 Graphical Designer 60
6.12 Environnement généré 62
6.13 Paradigme GME 63
6.14 Éditeur créé avec GME 65
6.15 Exemple de méta-modèle défini avec GOPPRR 68
6.16 Éditeur généré avec MetaEdit+ 69

7.1 Architecture des outils de modélisation 79
7.2 Notre approche 80

8.1 Interfaces internes d'un composant composite Fractal. 85
8.2 (a) Composant *ClientServeur*. (b) Description en ADL 86

8.3 (a) Modèle. (b) Implémentation en Julia. 87
8.4 domaine des composants simplifié 89
8.5 Architecture de GyTUNe 90
8.6 ProjectManagerType 91
8.7 DrawingAreaType 91
8.8 editor-itf 92
8.9 AbstractSyntaxType 93
8.10 ConcreteItf 94
8.11 SemanticItf 94
8.12 (a) DSL SCA simplifié. (b) exemple de modèle SCA 97
8.13 Méthode *getRepresentation()* de *ComponentSc* 100
8.14 Éditeur de dessin du *DrawingArea* 104

9.1 a) Méta-modèle de Fractal. b) exemple d'architecture Fractal 109
9.2 Syntaxe abstraite 111
9.3 ADL JEE 114
9.4 modèle d'une application JEE 115
9.5 *Reconfiguration Description Language* 116
9.6 Exemple de reconfiguration d'une application JEE 117
9.7 Syntaxe abstraite 118
9.8 Méta-modèle diagramme de gantt 123
9.9 Modèle diagramme de gantt 124
9.10 Méta-modèle graphe de PERT 124
9.11 Modèle graphe de PERT 125