MASTER EN SCIENCES INFORMATIQUES

Implémentation d'un éditeur BPMN au sein d'un outil de métamodélisation

Simon, Arnaud

Award date:
2014

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2013–2014

Implémentation d'un éditeur BPMN au sein d'un outil de métamodélisation

Arnaud SIMON

Maître de stage : Vincent ENGLEBERT

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Vincent ENGLEBERT

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Abstract

Modeling languages are the reference to specify and configure information systems. To match an application domain, the *domain-specific languages (DSL)* are customized and updated with changes in the domain. That is why it is necessary to have an adequate envirronement for the development and the maintenance of such languages to facilitate their evolution.

In this thesis, an approach to the implementation of a popular modeling language, BPMN 2.0, in a metamodeling tool, MetaDone. Specifically, we will implement the BPMN 2.0 metamodel within MetaDone and define the concrete syntax using the Grasyla 2 language. Next, a review of the MetaDone implementation approach and a review of the BPMN 2.0 language will be presented.

Résumé

Les langages de modélisation sont la référence pour spécifier et configurer les systèmes d'informations. Afin de correspondre à un domaine d'application, ces *domain-specific languages (DSL)* sont personnalisés et sont mis à jour selon l'évolution du domaine. C'est pourquoi il est nécessaire d'avoir un environnement adéquat permettant le développement et la maintenance de tels langages afin de faciliter leur évolution.

Dans ce mémoire, une approche permettant l'implémentation d'un langage de modélisation connu, BPMN, au sein d'un outil de métamodélisation, MetaDone. Concrètement, nous implémenterons le métamodèle de BPMN au sein de MetaDone et définirons sa syntaxe concrète à l'aide du langage Grasyla 2. Ensuite, une critique concernant l'approche de l'implémentation proposée par MetaDone ainsi qu'une critique sur le langage BPMN seront présentées.

Remerciements

En préambule à ce mémoire, je souhaitais adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à l'élaboration de ce mémoire ainsi qu'à la réussite de ce cursus universitaire.

Je tiens à remercier sincèrement le professeur Vincent Englebert, qui, en tant que Promoteur et Maître de Stage, s'est toujours montré à l'écoute et très disponible tout au long de la réalisation de ce mémoire, ainsi que pour l'inspiration, l'aide, les multiples corrections et le temps qu'il a bien voulu me consacrer et sans qui ce mémoire n'aurait jamais vu le jour.

Je remercie mes parents pour leur contribution, leur soutien, leur patience et leurs sacrifices sans qui la réalisation de ce travail aurait été impossible.

Ensuite, je tiens à remercier Axel, Keyvin, Pol et Florent pour l'ambiance qu'ils ont apportée pendant la durée de mon stage.

Enfin, j'adresse mes plus sincères remerciements à tous mes proches et amis, qui m'ont toujours soutenu et encouragé au cours de la réalisation de ce mémoire.

Table des matières

1 Introduction **7**
- 1.1 Contexte . . . 7
  - 1.1.1 CASE . . . 7
  - 1.1.2 DSL . . . 7
  - 1.1.3 CASE Tools . . . 8
  - 1.1.4 MetaCASE . . . 8
- 1.2 Objectifs . . . 9
- 1.3 Problématique de référence . . . 10
- 1.4 Plan du mémoire . . . 10

2 État de l'art **13**
- 2.1 Les metaCASE . . . 13
  - 2.1.1 Introduction . . . 13
  - 2.1.2 Les DSML . . . 14
  - 2.1.3 Logiciels existants . . . 14
  - 2.1.4 Constatations globales . . . 19
- 2.2 Modélisation de processus business . . . 19
  - 2.2.1 Introduction . . . 19
  - 2.2.2 Langages existants . . . 20

3 BPMN **27**
- 3.1 Description générale . . . 27
  - 3.1.1 En général . . . 28
  - 3.1.2 Histoire de BPMN . . . 28
  - 3.1.3 Utilisation . . . 30
  - 3.1.4 Analyse de l'efficacité cognitive . . . 30
- 3.2 Inventaire des constructions . . . 31
  - 3.2.1 Liste des éléments . . . 31
  - 3.2.2 Constructions spécifiques (Challenges) . . . 36
- 3.3 Métamodèle . . . 38
  - 3.3.1 Description . . . 38
  - 3.3.2 Critique . . . 38
- 3.4 Comparaison des outils existants . . . 41
  - 3.4.1 Introduction . . . 41
  - 3.4.2 Constatations communes . . . 41
  - 3.4.3 Présentation des outils . . . 42

3.4.4 Conclusion . . . 46
3.5 Sémantique de BPMN . . . 46
3.5.1 Instanciation et terminaison d'un processus . . . 47
3.5.2 Les activités . . . 47
3.5.3 Les portes . . . 48
3.5.4 Les événements . . . 48

4 MetaDone **49**
4.1 Base de données . . . 49
4.1.1 MeTaL1 . . . 49
4.1.2 MeTaL2 . . . 50
4.2 Grasyla 2 . . . 51
4.2.1 Script Grasyla . . . 51
4.2.2 Exemple . . . 53
4.3 Architecture . . . 53

5 Implémentation **55**
5.1 Architecture du plug-in . . . 55
5.2 Métamodèle . . . 55
5.3 Grasyla . . . 58
5.3.1 Construction du fichier . . . 58
5.3.2 Points importants . . . 59
5.4 Métriques . . . 63
5.5 Modélisation de l'exemple de référence . . . 64
5.6 Illustrations . . . 64

6 Approche prospective **69**
6.1 Discussions sur l'éditeur . . . 69
6.2 Embryon de simulateur (animation de BPMN) . . . 70
6.2.1 Contexte . . . 70
6.2.2 Principe . . . 70
6.2.3 Grasyla . . . 72
6.2.4 Conclusion . . . 72
6.3 Critique de BPMN . . . 72
6.4 Critique de MetaDone . . . 73
6.4.1 Grasyla 2 . . . 74
6.5 Future work . . . 77

7 Conclusion **79**

Chapitre 1

Introduction

1.1 Contexte

1.1.1 CASE

Le développement de systèmes logiciels est une tâche complexe. L'ingénierie du logiciel est une discipline qui définit des processus de développement afin de permettre une construction d'un logiciel d'une manière fiable, à moindres coûts et dans les délais impartis [IL97]. Des modèles tels que le modèle en cascade ou le modèle en spirale proposent de structurer le développement en une série de phases bien déterminées. L'automatisation de ces méthodes est l'objet d'une discipline appelée le "Computer Aided Software Engineering", plus connue sous l'abréviation "CASE".

L'apparition de cette discipline remonte aux années 80. Chikofsky définit le CASE comme étant une technologie d'intégration de la production qui lie les méthodes et les outils dans des environnements commercialement viables et efficaces [CR88]. Beaucoup de chercheurs, dont Chikofsky [CR88] et McClure [McC88], ont affirmé que le CASE peut réduire substantiellement le coût de développement des logiciels, standardiser les spécifications et augmenter la qualité des systèmes d'information. Nous verrons que cette vision comprend plusieurs problèmes importants.

1.1.2 DSL

Le développement logiciel doit donc être un processus encadré. Une manière largement utilisée est l'utilisation de méthodologies. Selon Smolander [SLTM91], une méthode est un ensemble de concepts qui détermine ce qui est perçu, un ensemble de conventions qui gouvernent comment la perception est représentée et communiquée, et un ensemble de lignes directrices qui établissent comment les représentations sont dérivées ou transformées. Un produit important d'une méthodologie est un modèle. Selon Perry [PK91], un modèle est une simple représentation d'un système qui fournit une abstraction sur certains détails non essentiels et met l'accent sur les composants essentiels du modèle. Cependant, ce concept de composant essentiel varie selon plusieurs choses : le type de profil d'utilisateur (manager ou développeur) ou encore l'étape concernée du processus de développement (analyse et implémentation). Le langage (formalisme) dans lequel le modèle est représenté doit pouvoir s'adapter à ces contraintes. Un langage adapté est appelé un "Domain-Specific Language" (DSL). Une caractéristique essentielle de ces DSL est que leur réalisation est un long proces-

sus qui requiert plusieurs propositions, corrections et itérations afin de suivre l'évolution de leur domaine d'application [Eng00]. Cette constatation va fortement influencer l'utilisation des outils CASE (CASE Tools) comme nous le verrons à la section suivante.

1.1.3 CASE Tools

Les CASE Tools sont les outils qui vont mettre en œuvre de manière informatisée la discipline CASE. Les CASE Tools peuvent être classés en deux catégories [IL97] : les outils "Front End" qui sont utilisés durant l'analyse et le design, et les outils "Back End" qui sont utilisés durant l'implémentation et les tests. Englebert [Eng00] identifie cinq dimensions si l'on considère un CASE Tool comme une boîte noire :

- Les CASE Tools peuvent stocker et gérer de l'information.
- Les CASE Tools peuvent afficher de l'information (sous forme de graphes, de tables).
- Les CASE Tools peuvent exécuter des tâches (analyse, génération, transformation).
- Les CASE Tools peuvent interagir avec les utilisateurs (interface utilisateur).
- Les CASE Tools peuvent communiquer ensemble (groupe de travail).

Les CASE Tools peuvent être d'une grande aide dans la réalisation de tâches encombrantes pour un informaticien comme la génération de code ou des transformations de schémas. Selon ter Hofstede [THV96], ces tâches sont fastidieuses, consommatrices de temps, et même impraticables, si leur utilisation n'est pas prise en charge par des outils automatisés.

Cependant, plusieurs auteurs dont certains cités ci-après constatent que les CASE Tools comportent d'importants problèmes.

Isazadeh [IL97] constate les CASE tools sont complexes, nécessitent beaucoup de main d'oeuvre, et sont extrêmement coûteux à produire et à adopter. Ils supportent un nombre fixe de méthodologies, mais les organisations de développement logiciel changent dynamiquement leurs méthodologies. De plus, ces méthodologies sont fixées et sont difficilement modifiables au travers de ces outils. Cette approche traditionnelle est une source de grandes difficultés.

Selon Englebert [Eng00], la personnalisation est un mot-clé essentiel pour augmenter l'utilisation et l'adéquation des CASE Tools avec la pratique. Il identifie trois manques importants dans la capacité de personnalisation des CASE Tools : l'ensemble des représentations graphiques est souvent fixé, la base de données est trop rigide et leur architecture n'est pas flexible.

Marttiin [MRTL93] affirme que les développeurs ont tendance à se diriger vers les CASE Tools qui sont capables de fournir un support pour plusieurs différentes méthodologies.

Afin d'atténuer ces importants problèmes dans la conception des CASE Tools, des chercheurs ont conçu de nouveaux outils, les metaCASE.

1.1.4 MetaCASE

La première conférence sur les metaCASE "First International Conference on MetaCASE" a eu lieu en 1995 à Sunderland (UK). Lors de cette conférence, Alan Gillies affirmait que les metaCASE échoueraient si les facilités offertes par ces metaCASE ne compensent pas les raisons pour lesquelles les CASE Tools ne sont pas adoptés.

Selon Englebert [Eng00], les metaCASE ont deux architectures possibles : a) le metaCASE se comporte comme un compilateur et génère des programmes prêts à compiler (ToolBuilder [Ald91], Kogge [ESU97]) et b) le metaCASE interprète les spécifications pour émuler le

CASE Tool (MetaEdit+ [TR03], Dome [EK00], GME [LMB+01], MetaDone [Eng00]). Dans ce mémoire, nous nous intéresserons principalement à ces derniers.

Un metaCASE doit donc permettre, via ses mécanismes et ses langages, la personnalisation d'un CASE Tool. D'après Isazadeh [IL97], ce processus peut prendre juste quelques heures, ce qui est plus court et moins coûteux que l'approche traditionnelle de personnalisation par le vendeur d'un CASE Tool.

Les premiers metaCASE étaient des environnements basés sur du texte comme Plexsys [KK84] ou MetaPlex [CN89]. Ces environnements séparaient déjà la définition du langage (niveau méta) et l'actuelle information du système (niveau instance) qui utilise le langage définit au niveau méta [KS96]. Par la suite, l'évolution des interfaces graphique a poussé la création de metaCASE basés sur un environnement graphique. En effet, un modèle graphique est souvent plus représentatif des concepts qu'il illustre, du point de vue humain, plutôt que son équivalent sous forme de texte. Cette nouvelle génération de metaCASE implique de devoir définir la syntaxe concrète en plus de la spécification du langage.

Dans ce mémoire, nous étudierons l'implémentation d'un langage de modélisation standard au sein d'un metaCASE, afin d'en tirer des conclusions. Le langage choisi est BPMN 2.0, particulièrement connu pour sa grande expressivité et la diversité des constructions possibles. Le metaCASE qui supportera l'implémentation sera MetaDone, un logiciel développé à l'Université de Namur. Ce travail implique de nombreux enjeux et objectifs qui sont décrits dans la section suivante.

23 [CN89] 24

1.2 Objectifs

Dans le contexte particulier décrit aux points précédents, l'objectif principal du mémoire est d'implémenter BPMN 2.0 au sein de MetaDone. Trois aspects distincts sont à aborder : l'implémentation des constructions BPMN et de leurs représentations graphiques dans un metaCASE, la critique de cette implémentation et en retirer une perspective d'évolution de MetaDone.

Concernant l'implémentation des constructions, il faut tout d'abord établir la faisabilité d'implémentation d'un langage aussi complet que BPMN. Les langages tels que les réseaux de Petri ou URN, déjà implémentés dans MetaDone, pourront fournir une base de travail car ils ont des concepts en commun avec BPMN. Ensuite, le but sera de produire un éditeur graphique comprenant un maximum de constructions possible en respectant les règles définies dans le standard BPMN de l'OMG. Nous pourrons ainsi évaluer la capacité de MetaDone à gérer un langage complexe en respectant à la fois son métamodèle et sa syntaxe concrète. De plus, il serait intéressant d'intégrer un embryon de simulateur BPMN pouvant illustrer le déroulement d'une exécution de processus.

Du point de vue de la critique de l'implémentation de BPMN dans MetaDone, il faudra comparer les éditeurs BPMN populaires afin de relever leurs forces et faiblesses, pour illustrer l'avantage d'utiliser un metaCASE par rapport à une solution dédiée uniquement à BPMN. A travers cela, il sera intéressant de montrer les avantages et faiblesses de MetaDone par rapport à ses concurrents.

Concernant la perspective d'évolution, d'une part des pistes d'amélioration seront à suggérer afin de donner des lignes directrices pour une future version de MetaDone.

1.3 Problématique de référence

Afin d'illustrer les concepts de ce travail, cette section décrit un exemple de référence fictif illustrant les différentes difficultés d'implémentation de BPMN (voir figure 1.1).

Cet exemple illustre le passage et le traitement d'une commande de plusieurs produits entre un client et une entreprise. L'entreprise est constituée d'un service de facturation, d'un service responsable des ventes ainsi qu'un service dédié à l'exécution des commandes. Le processus est le suivant. Le client envoie sa commande au service des ventes. Cette commande est ensuite entrée dans le système informatique de l'entreprise. Si la commande est erronée, elle est annulée. Dans le cas positif, elle est transmise au service d'exécution. Le service d'exécution vérifie la disponibilité de chaque produit constituant la commande. Si chaque produit est disponible en stock, le compte client est mis à jour avec le détail de la commande, ce qui termine le processus. Dans le cas où un produit est indisponible, la commande est annulée. De plus, chaque 1er du mois, le service de facturation génère un relevé concernant l'ensemble des comptes clients de l'entreprise.

Cet exemple de référence a été modélisé à l'aide d'un éditeur BPMN. Il se trouve à la figure suivante : 1.1.

1.4 Plan du mémoire

La structure du mémoire est définie ci-après. Tout d'abord, l'état de l'art apportera un aperçu des metaCASE ainsi que des langages de modélisation de workflow. Ensuite, MetaDone et BPMN seront expliqués chacun dans un chapitre distinct afin de mettre en place les concepts nécessaires à la compréhension de l'implémentation. Cette implémentation de BPMN au sein de MetaDone sera exposée dans le chapitre suivant. Elle sera séparée en deux sections complémentaires, d'une part l'implémentation du métamodèle et d'autre part l'implémentation de la syntaxe concrète. Enfin, le dernier chapitre comprendra une critique globale du travail. Cette critique globale sera une réflexion sur l'ensemble du travail. Tout d'abord, BPMN sera critiqué et une amélioration de la syntaxe graphique sera implémentée au sein de l'éditeur. Nous verrons que BPMN pourra constituer une base pour des futurs DSL, et que l'adaptation de BPMN à travers MetaDone présente de nombreux avantages par rapport à l'adaptation d'un outil dédié à BPMN. Ensuite, une appréciation des avantages et faiblesses de MetaDone sera décrite. Elle aboutira à plusieurs pistes d'améliorations à envisager pour une future version de ce logiciel.

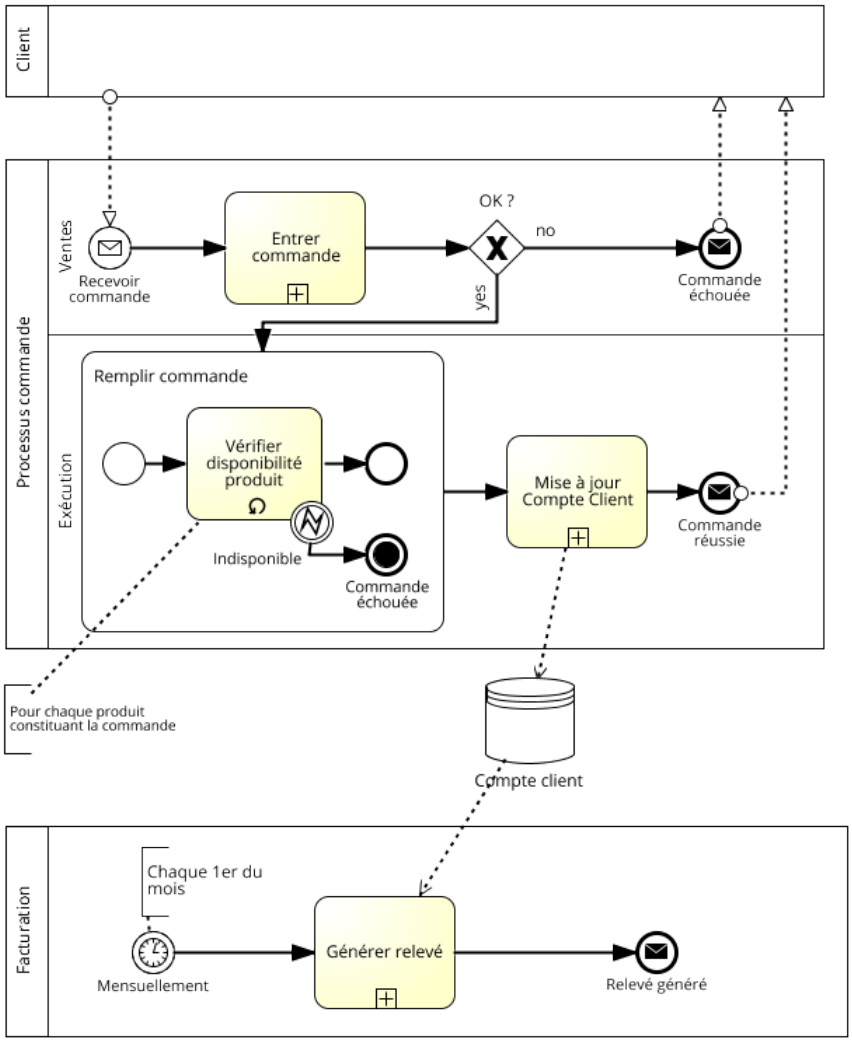

FIGURE 1.1 – BPMN : Exemple contextuel

Chapitre 2

État de l'art

2.1 Les metaCASE

2.1.1 Introduction

Cette section se base sur l'article [Met04].

La plupart des metaCASES sont basés sur une architecture à trois niveaux, contrairement au *CASE Tools* qui sont basés sur deux niveaux. Ils fournissent une couche supplémentaire permettant d'abstraire certains concepts. Le niveau le plus bas est celui des modèles, créés par l'utilisateur final. Le niveau intermédiaire contient le modèle de la méthode appelé le métamodèle. Il inclut les concepts et les règles de construction pour une méthode donnée. Il peut représenter des concepts par des classes ainsi que les relations d'héritages entre ces classes. La représentation du métamodèle est associée avec une syntaxe concrète qui fournit la représentation graphique des éléments du langage. Par exemple, la notation BPMN propose un métamodèle que nous décrirons en détail dans la section 3.3. Ce niveau est codé dans un *CASE Tool* est n'est pas modifiable. Le dernier niveau est celui du méta-métamodèle, implémenté dans les metaCASE, qui permet de définir un métamodèle. Un méta-métamodèle est conçu pour être générique afin de pouvoir exprimer toutes les constructions possibles d'un métamodèle.

D'un point de vue pratique, l'utilisateur a la possibilité de créer un métamodèle qui peut être basé sur un langage existant et potentiellement le personnaliser selon ses besoins. Le metaCASE va interpréter ce métamodèle pour émuler un *CASE Tool* souvent de manière graphique. Un grand avantage des metaCASE est que cette méthode est plus rapide que développer un *CASE Tool* "from scratch". De plus, il sera plus rapide de faire évoluer la méthode, aussi appelée le langage (DSML voir 2.1.2), au court du temps afin de suivre l'évolution du domaine d'activité.

Rossi [RK99] identifie 9 exigences pour la conception d'un metaCASE : 1) La simplicité de l'architecture du système et le métamodèle associé ainsi que la facilité d'utilisation. 2) La puissance d'expression du métamodèle doit être maximisée et être simple dans le même temps. 3) L'identification d'unités ou modèles fins. 4) La capacité à modéliser des objets complexes. 5) La possibilité d'utiliser partiellement les métamodèles et modèles implémentés. 6) Le support de la continuité des données. 7) Le dynamisme du type et instance de niveaux. 8) Plusieurs niveaux de modification 9) Un support multi-utilisateur.

2.1.2 Les DSML

Les "Domain-specific modelling language", DSML, ont deux buts selon Kelly [KT08]. Premièrement, augmenter le niveau d'abstraction à travers la programmation d'une spécification de solution dans un langage qui utilise directement les concepts et les règles d'un domaine spécifique. Deuxièmement, générer un produit final dans un langage de programmation choisi ou une autre forme depuis ces spécifications haut niveau.

Ces langages ont, pour Luoma [LKT04], une influence claire sur la productivité : "DSM had a clear productivity influence due to is higher level abstraction : it required less modeling work, which could often be carried out by personnel with little or no programming exprerience."

2.1.3 Logiciels existants

GME

Cette section se base sur les articles [LMB+01] et [GME08].

GME, "Generic Modeling Environnment", est un metaCASE fonctionnant sous Windows développé à l'Université Vanderbilt. Il permet la spécification de métamodèles décrivant un langage de modélisation afin de générer un environnement spécifique pour ce langage. Les modèles créés dans cet environnement sont stockés dans une base de données de modèles.

Les concepts de modélisations sont définis selon un vocabulaire spécifique à GME. Ils sont illustrés à la figure 2.1. Un *Project* contient un ensemble de *Folders*. Ces *folders* sont des conteneurs qui organisent les modèles comme sur un système de fichiers. Les *Models*, *Atoms*, *References*, *Connections* et *Set* sont des objets de première classe qui peuvent avoir des *Attributes*. Les *Atoms* sont les objets de base. Un exemple d'*Atom* est une porte AND ou XOR. Les *Models* sont les objets composés de *FCO* par l'intermédiaire de *Roles*. Les *Aspects* permettent un contrôle sur les objets graphiques composant un modèle. Ils permettent d'afficher ou cacher un groupe d'objets au sein d'un modèle. Les relations entre objets sont représentées par les *Connections*. Les connections entre objets sont établies à l'aide de ports contrairement à d'autres MetaCASE comme MetaEdit+ qui indique une relation entre deux objets avec une certaine cardinalité. Les ports sont des entités internes aux entités de base d'un métamodèle définit dans GME. Les *References* sont assimilables à un pointeur vers un objet existant. Les *Connections* et les *Références* sont des relations binaires. Afin de construire des relations vers plusieurs objets, l'objet *Set* est prévu et défini un ensemble d'objets.

Les sémantiques statiques des langages de modélisation sont spécifiées avec des contraintes OCL (Object Constraint Language). La représentation visuelle (syntaxe concrète) est assimilée à un attribut de l'objet. Un objet peut donc avoir un attribut de type String décrivant le chemin vers le symbole adéquat.

L'architecture de GME est basée sur des composants. Plusieurs composants gèrent le stockage selon plusieurs format comme MS Repository (basé sur SQL Server) et un format de fichier propriétaire. Le composant `Core` implémente les blocs principaux de l'outil comme les concepts de modélisation. Deux composants utilisent les services du `Core` : le `Meta` et le `MGA`. Le `Meta` défini les paradigmes de modélisation tandis que le `MGA` implémente les concepts de modélisation de GME pour le paradigme donnés. L'architecture est extensible par l'ajout d'add-ons pour étendre les fonctionnalités de GME. Par exemple, le manager (interpréteur) des contraintes OCL est un add-on à part entière. Aussi, la partie visualisation et la partie stockage des modèles sont deux composants séparés par une interface accessible à

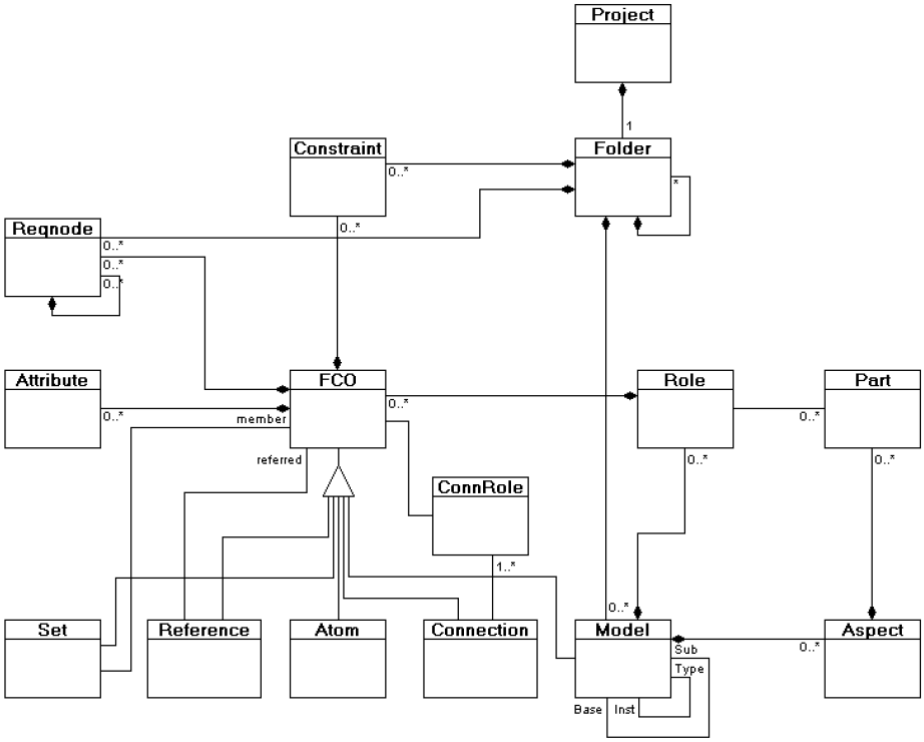

FIGURE 2.1 – GME : Concepts de modélisation

d'autres composants. Grâce à cette conception, plusieurs composants peuvent être remplacés facilement selon l'environnement et les besoins liés à l'utilisation de GME.

Dome

Cette section se base sur les articles [EK00] et [LMB+01].

Domain Modeling Environment, DOME, est un metaCASE permettant de synthétiser rapidement des outils basés sur une notation graphique. Un outil (éditeur) est défini par un modèle de métamodélisation appelé DOME Tool Specification (DTS). Un DTS peut être compilé et interprété par MetaDOME, le compilateur de DOME. MetaDOME génère du code Smalltalk intégré avec l'environnement de développement CINCOM VisualWorks Smalltalk. Smalltalk est un langage de programmation orienté objet disposant d'un environnement de développement intégré complètement graphique [Wik14a].

L'outil DTS a deux composants : une notation graphique (DSTL) permettant de spécifier les objets (classes, propriétés, contraintes et attributs visuels) d'un langage graphique et un langage textuel basé sur Scheme permettant l'implémentation de comportements spéciaux. Ce dernier est appelé "Alter".

Le langage DSTL supporte les concepts suivants. Un *Model* consiste en un réseau de graphes. Les modèles peuvent être hiérarchiques, c'est-à-dire qu'un modèle peut contenir plusieurs modèles. Un *Graph* est un ensemble de composants gouverné par les contraintes imposées par un métamodèle. Un *Component* est un objet de base d'un graphe (noeud). Il possède une *Interface* qui consiste en un ensemble de connections qui peuvent être liées aux

interfaces des autres *Component*. Enfin, un *Archetype* est un composant réutilisable défini dans une partie spécifique d'un modèle. Il peut être réutilisé par l'intermédiaire de références au sein du modèle.

Alter assure la transformation des modèles en code, documentation ou analyses au sein de DOME. Alter est un langage basé sur le standard SCHEME comportant plusieurs extensions. Ces extensions permettent la navigation dans un modèle, la manipulation d'une interface utilisateur, la capacité d'appel de procédures et un support de programmation orienté objet. Il permet un accès complet à la base de données des modèles en lecture, modification et création d'entités.

Cependant, DOME a plusieurs limitations. Le langage Alter devrait supporter l'exécution des modèles mais DOME ne dispose pas des fonctionnalités suffisamment robustes pour la supporter. Par rapport au stockage, DOME stocke les modèles dans des fichiers ce qui implique des problèmes lors de la séparation de modèles en petites parties éditables ou en édition collaborative par équipe.

MetaDone

MetaDone est à l'origine un logiciel développé par V. Englebert [Eng00], professeur à la faculté d'informatique à l'Université de Namur. Ce logiciel, qui est le support de l'implémentation de ce travail, sera détaillé dans le chapitre 4.

MetaEdit+

Cette section se base sur les articles [TR03] et [Met14b].

MetaEdit+ est un logiciel de métamodélisation très populaire développé par la société MetaCase. Il fournit une suite d'outils permettant de définir les concepts d'un langage, les règles, les symboles et de générer le code relatif à un modèle conçu par l'utilisateur. L'interface de MetaEdit+ est basée sur une série de boites de dialogue.

MetaEdit+ se base sur la structure d'un métamodèle pour générer un outil de modélisation. La représentation des entités est réalisée avec le langage Smalltalk, utilisé aussi dans DOME (section 2.1.3).

Le logiciel supporte l'édition de modèle multi-utilisateurs. La base de données des DSL de MetaEdit+ fournit une gestion des versions. Lors d'une mise à jour, les changements sont répercutés sur les modèles des utilisateurs. De plus, les définitions des DSL peuvent être partagés selon des fichiers pouvant être importés par une autre instance de MetaEdit+. Une approche conservative est observée lors de la modification des métamodèles existants. Par exemple, si un concept est enlevé du métamodèle, la création d'instances de ce concept ne sera plus possible, mais les instances existantes ne sont pas supprimées des modèles.

Les générateurs de code sont définis en utilisant le Generator Editor basé sur MERL, un DSL textuel conçu pour transformer les modèles en texte. Un débugger permet de contrôler l'exécution d'un générateur en affichant le modèle, la règle de transformation appliquée ainsi que le code résultant. Il fournit la possibilité d'insérer des breakpoints ou encore d'exécuter les transformations pas à pas. Ce débugger est illustré à la figure 2.2 où le générateur transforme un modèle en code HTML. De plus, le code généré permet, au travers d'une fonctionnalité appelée "Live code", de basculer vers la partie du modèle correspondant à une portion de code via un clic de l'utilisateur.

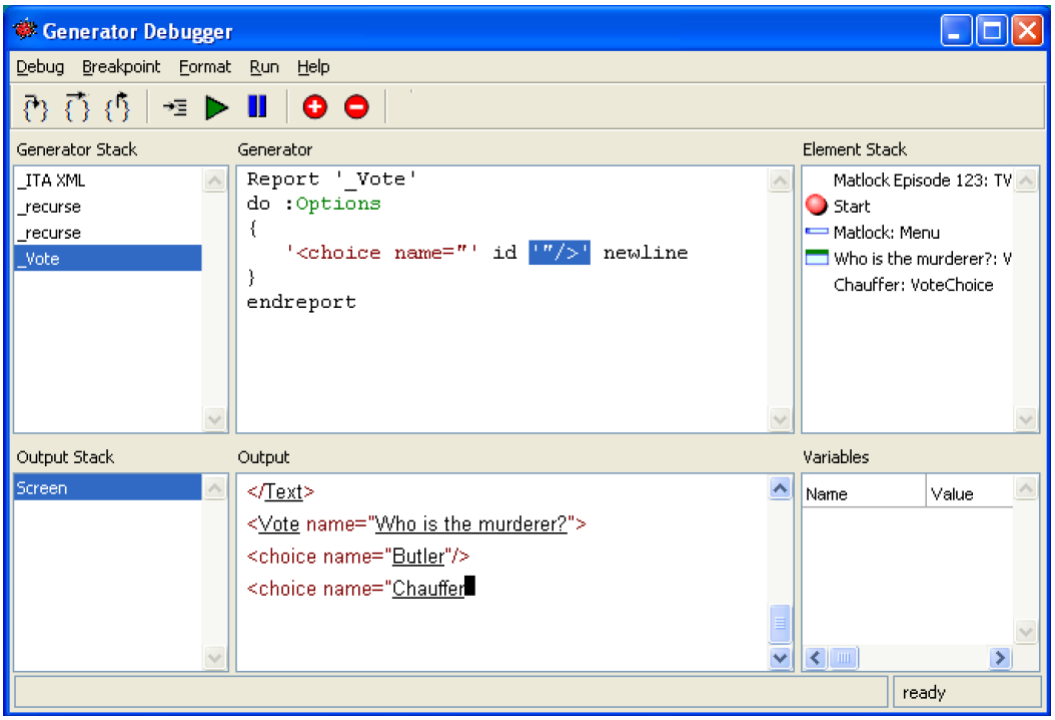

FIGURE 2.2 – MetaEdit+ : Generateur Debugger

OpenFlexo

[Ope14] Openflexo est un metaCASE capable de produire un éditeur graphique à partir d'une source d'information. Il utilise un langage dédié appelé "Flexo Modeling Language" afin de définir les perspectives spécifiques au domaine d'application. Aussi, il fournit un ensemble de mécanismes pour manipuler et représenter les informations.

OpenFlexo organise les informations dans une notion propre appelée "Flexo Concept" afin de les manipuler. Cette notion fait référence à différentes sources d'information telles que des modèles, métaobjets, représentation graphique, etc. Les relations entre ces "Flexo Concept" sont appelées des "Flexo Roles".

Un "Viewpoint" définit un ensemble de règles de construction qui permettent d'afficher une "View" particulière, une "View" étant une agrégation d'un ensemble de modèles.

Afin de spécifier les informations, OpenFlexo peut utiliser plusieurs technologies comme EMF [1], les ontologies OWL, XML et les feuilles de calcul Excel.

Openflexo propose un éditeur BPMN qui sera analysé dans la section 3.4.3.

ATOM 3

Cette section se base sur les articles [dLV02] et [ATo].

AToM^3, littéralement "A Tool for Multi-Formalism and Meta-Modelling", est un logiciel de métamodélisation pouvant utiliser et exprimer plusieurs formalismes différents. Il est développé au "Modelling, Simulation and Design Lab (MSDL)" à la faculté d'informatique de l'Université McGill.

1. EMF : Eclipse Modeling Framework

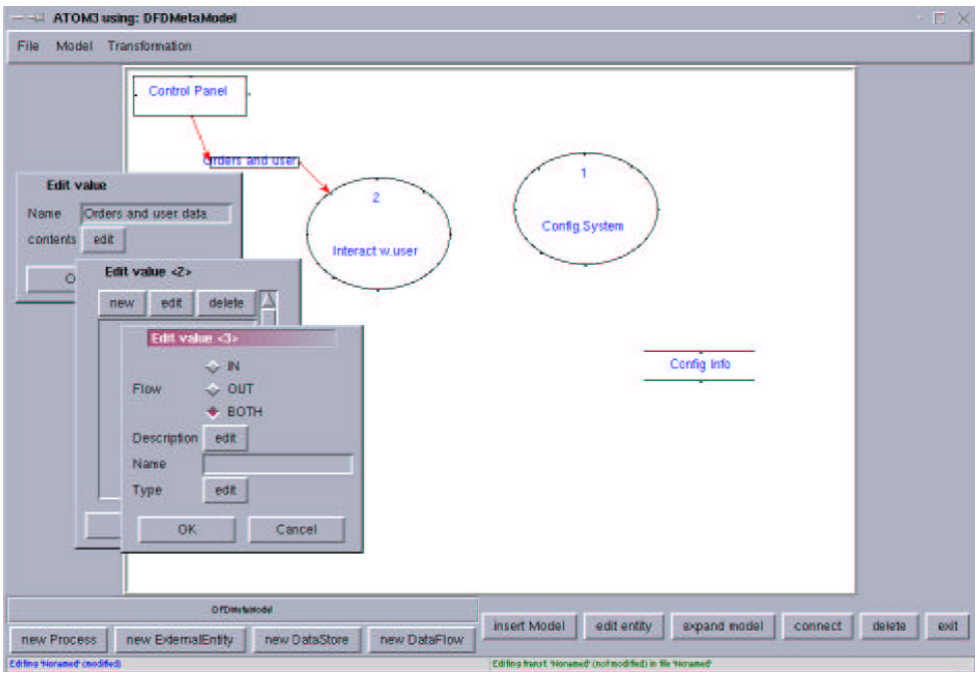

FIGURE 2.3 – ATOM3 : Editeur DFD

L'outil a une couche de métamodélisation qui lui permet de représenter différents formalismes graphiques. D'après une méta-spécification, composée d'un modèle entité-relation (ER) étendu avec des contraintes, $AToM^3$ génère un outil permettant d'éditer des modèles. Cette méta-spécification est composée d'un métamodèle exprimé à l'aide d'une notation graphique (ER) et de contraintes exprimées de manière textuelle. $AToM^3$ gère ces contraintes avec le langage OCL (Object Constraint Language) de la notation UML.

Afin d'exprimer la manipulation des modèles, l'outil utilise des grammaires graphiques. Elles sont composées de règles mappant un graphe vers un autre graphe après plusieurs itérations. Des sous-graphes sont identifiés dans le premier graphe et remplacés dans le deuxième graphe selon une règle. Les itérations se terminent lorsqu'il n'existe plus de règle à appliquer. Dans $AToM^3$, les règles sont ordonnées par une priorité donnée par l'utilisateur.

Le principal composant de $AToM^3$ est le "Processor". Il est responsable du chargement, de la sauvegarde, de la création et de la manipulation des modèles. D'après les méta-spécifications, ce composant génère des fichiers en Python qui permettent la manipulation des modèles.

Chaque entité du métamodèle peut être associée à une représentation graphique. Les attributs et contraintes relatives aux entités peuvent aussi être affichés avec la représentation. Chaque représentation graphique d'entité est composé de plusieurs formes graphiques référencées par un nom généré automatiquement. Ce nom donne accès à des méthodes permettant par exemple de changer la couleur de la forme ou de la cacher. La figure 2.3 illustre $AToM^3$ implémentant le formalisme Data Flow Diagram (DFD).

2.1.4 Constatations globales

Les metaCASE décrits dans ce chapitre ont plusieurs points communs mais ont aussi des approches différentes.

Tous les metaCASE analysés produisent un environnement d'édition à l'aide d'une définition d'un métamodèle.

La définition du métamodèle est différente selon les logiciels. MetaEdit+ choisit une approche tournée vers l'utilisateur en définissant un métamodèle par une succession de boîtes de dialogue. D'autres metaCASE comme MetaDone, GME ou DOME utilisent des langages de métamodélisation (MeTaL [Eng06], OCL (Object Constraint Language), DSLT [EK00]) afin de définir les métamodèles. Le premier est plus tourné vers l'utilisateur non-expert tandis que le second permet d'avoir une approche plus flexible destinée à un utilisateur plus avancé.

Les metaCASE ont une manière différente d'extraire et manipuler l'information des modèles. "MetaEdit+, via son système de générateur, permet la génération de code facile pour un utilisateur "de base". Ces générateurs ne permettent pas des fonctionnalités avancées que permettraient un langage de programmation" [LMB+01]. MetaDone permet de manipuler le modèle sous-jacent par l'ajout de plugins développés dans des langages compatibles avec la JVM comme Groovy ou Scala et l'utilisation d'une API d'accès au kernel, en ce compris le repository. Ce script a un accès global au modèle construit par l'utilisateur et peut effectuer des opérations complexes de manière transparente. GME représente les données en un réseau de données C++ grâce au "Builder Object Network". Ce réseau permet une définition d'API ayant un accès total au modèle. De plus, GME fournit des accès XML bidirectionnels pour chaque information du modèle et du métamodèle. DOME utilise le langage Alter afin de manipuler les modèles.

Plusieurs metaCASE (GME, MetaDone, MetaEdit, ...) ont une architecture prévue pour étendre leurs fonctionnalités sans modifier leur cœur.

2.2 Modélisation de processus business

2.2.1 Introduction

Cette section se base sur les articles [Wik14b] et [VDA98].

Un workflow est un anglicisme signifiant littéralement un "flux de travail". Il se caractérise par une représentation d'une série d'activités nécessaires à la réalisation d'un processus métier. Ces activités sont effectuées par des ressources et peuvent produire une information. Un flux, généralement représenté par des flèches, lie les activités entre-elles en indiquant l'ordre dans lequel elles doivent être effectuées.

Le "workflow management" est la gestion de ces flux de travail. Selon Van der Aalst [VDA98], le workflow management promet une nouvelle solution à un vieux problème : le contrôle, l'optimisation et le support des processus métier. Le support informatique permet de modéliser et d'exécuter les workflow afin de mieux les gérer.

Plusieurs langages, décrits à la section suivante, permettent de modéliser un workflow parfois de manière graphique. Chaque langage a une approche différente et permet de mettre en valeur différentes informations comme les informations transmises entre les activités ou encore les événements pouvant se produire durant un processus.

2.2.2 Langages existants

YAWL

YAWL, "An Other Workflow Language", est le fruit de l'université de technologie d'Eindhoven et de l'université de technologie du Queensland. Le point de départ a été les réseaux de Petri, qui ont été étendus avec des constructions pour gérer des instances multiples, une synchronisation avancée ainsi que des patterns d'annulation [vdAtH05, vdAADtH04]. Plus qu'étendre les réseaux de Petri, YAWL est vraiment un nouveau langage avec sa propre sémantique et est conçu spécifiquement pour la spécification de workflow. Pour cette notation, une spécification de workflow est un ensemble de processus formant une hiérarchie.

Chaque processus est constitué d'un ensemble de tâches pouvant être soit composites soit atomiques ainsi que des conditions, qui peuvent être assimilées à des places dans les réseaux de Petri [vdAtH05]. La figure 2.4 illustre les différents éléments de YAWL.

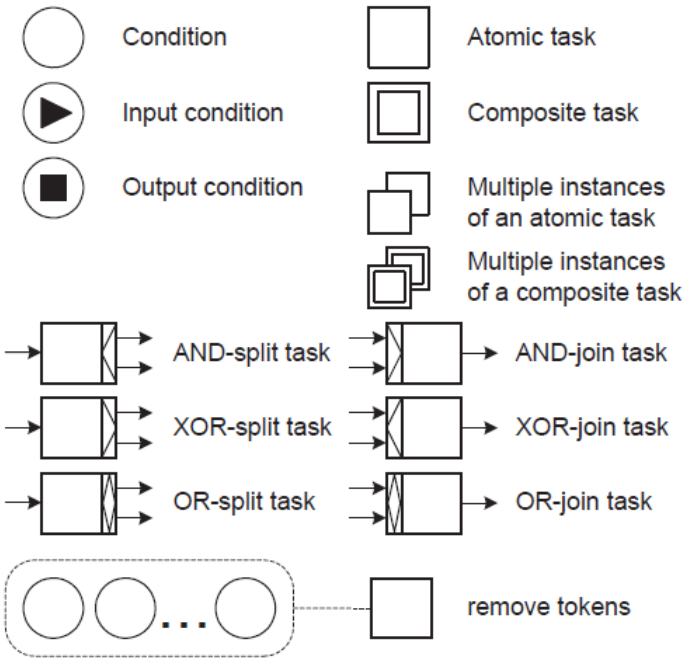

FIGURE 2.4 – Symboles YAWL

YAWL a une fondation formelle propre, ce qui rend les spécifications non ambiguës et permet une vérification automatisée possible [Yaw14]. Par rapport à BPMN, on remarque immédiatement que YAWL offre bien moins de constructions possibles, ce qui rend la notation plus simple. D'un autre point de vue, ce petit nombre d'éléments restreint l'expressivité de la notation. La précision doit alors être trouvée d'une manière différente via des commentaires ou des annotations. De plus, selon [Yaw14], les modèles BPMN peuvent être transformés en modèle YAWL pour être exécutés. YAWL a une implémentation fournie par la YAWL Foundation sur leur site `http://www.yawlfoundation.org/`.

UML

Les diagrammes d'activité UML sont un cas spécial des diagrammes d'état UML, qui sont tournés en représentation de machines à état [DtH01]. Ils sont représentés sous la forme d'un graphe où les nœuds sont connectés par des flèches, formant un workflow. "Les noeuds objet contiennent des données reçues en entrée et en sorties de noeuds exécutables, et se déplace à travers l'activité selon le sens du flux" [OMG11b] Les diagrammes d'activités peuvent être formés de plusieurs éléments de base indiqués ci-après. D'autres éléments, tels que les *Datastore* sont définis dans la spécification [OMG11b].

- Un rectangle contenant le diagramme
- Un header (en haut à gauche) qui identifie le diagramme
- Des activités, représentées par des rectangles arrondis
- Des flèches pour indiquer le sens du flux
- Des nœuds FORK et MERGE, représentés par des barres verticales
- Des points de choix (OR), représentés par des losanges verticaux
- Des cercles indiquant le début et la fin du processus

La figure 2.5 représente un diagramme d'activité nommé "processOrder". Il est formé d'une suite d'activités illustrant le passage d'une commande. On peut remarquer un point de choix ainsi que la séparation du flux en deux flux parallèles, où les activités se déroulent de manière concurrente. Les deux flux se rejoignent signifiant que la commande a été livrée et payée par le client. La commande est finalisée, mettant fin au processus.

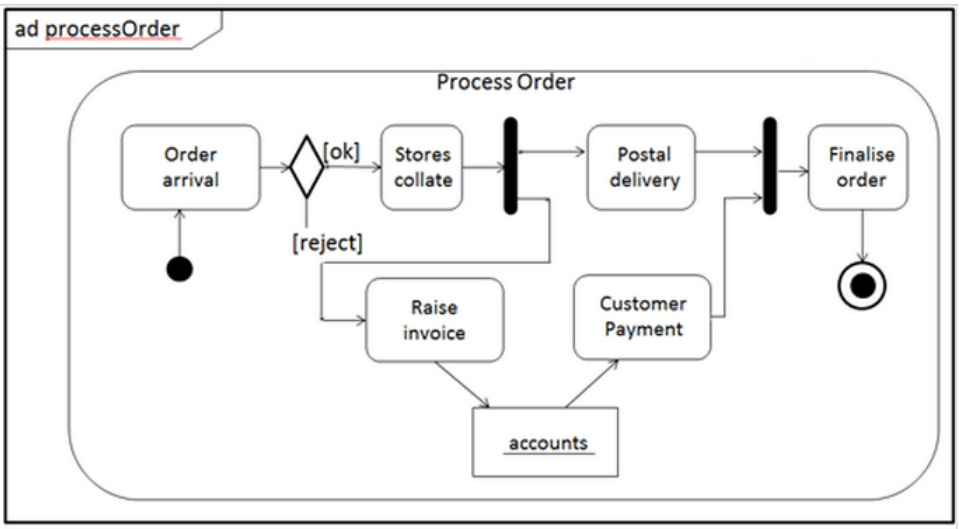

FIGURE 2.5 – Diagramme d'activité : Exemple

"Un diagramme d'activité composé de noeuds *FORK* et *MERGE* doit remplir certains critères pour être bien formé. Ces critères spécifient qu'il doit être possible de remplacer chaque noeud *FORK* par des *AND-states*" [DtH01] , les *AND-states* étant des noeuds particuliers des diagrammes d'état UML. Ce qui implique qu'en particulier, chaque noeud *FORK* doit avoir un noeud *MERGE* lui correspondant.

Comparaison avec BPMN Dans son article [Whi04], Stephen A.White (pour rappel étant le chef d'équipe conceptrice de BPMN 1.0) compare la notation BPMN avec les diagrammes

d'activité UML. Il ressort plusieurs points intéressants :

- Les deux notations proposent des solutions similaires pour les mêmes problèmes, révélant que les notations sont assez proches l'une de l'autre
- Les deux notations partagent des éléments communs dont les rectangles arrondis pour les activités ainsi que les diamants pour les points de choix
- Chaque diagramme BPMN peut être transformé en son équivalent en diagramme d'activité UML et vice-versa (à l'exception d'une construction particulière, le "Interleaved Parallel Routing pattern")
- Les deux notations ont chacune leur métamodèle défini dans la spécification

Les diagrammes BPMN et les diagrammes d'activités sont très proches dans leur aptitude à exprimer un processus. Cependant, BPMN 2.0 dispose de plus d'éléments graphiques qui peuvent faciliter la lecture de diagrammes plus complexes.

BPEL

BPEL, ou WS-BPEL, signifiant *Web Services Business Process Execution Language*, est un langage basé sur XML et étant un standard OASIS. Il est utilisé pour composer des *Web Services* au sein d'un processus exécutable. Il se base sur les définitions des *Web Services* (*WSDL*) et défini, sur base d'un diagramme, la manière dont les opérations doivent être séquencées [OAS07]. Un processus BPEL va produire un *Web Service*. C'est un langage d'orchestration, impliquant qu'une entité est responsable de l'exécution du processus mettant en œuvre plusieurs *Web Services*. En pratique, c'est le fichier BPEL, construit par un environnement de développement comme Eclipse [Ecl14]. La figure 2.6 représente un processus BPEL "main" pour l'exécution d'une commande. Le *Web Service* "Order" est accessible par un client. Le processus BPEL va d'abord recevoir la commande du client, puis demander un prix au fournisseur pour un produit particulier via le *Web Service* "getQuote". Le prix sera ensuite confirmé par un second *Web Service* du fournisseur qui terminera la commande par l'envoi de la facture au client. On remarque que trois *Web Service* ont été invoqués par ce processus BPEL.

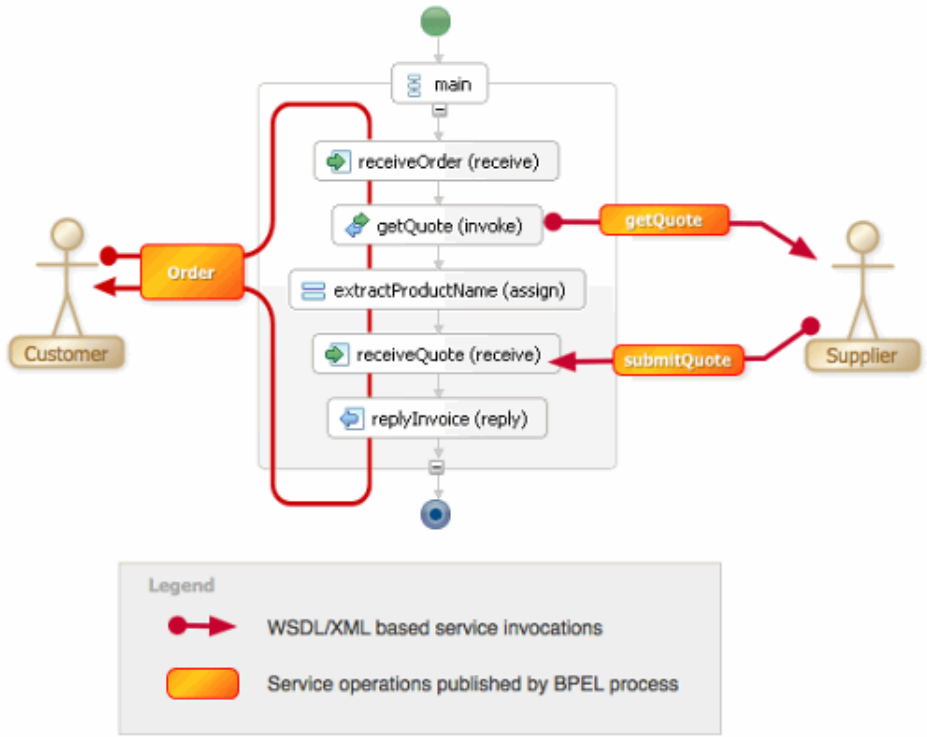

FIGURE 2.6 – BPEL : Exemple (http://www.amsimaging.com/blog/bid/138309/What-is-BPEL-and-Why-is-it-Important-for-your-Business)

Plusieurs articles ont été publiés concernant les relations entre BPEL et d'autres langages comme BPMN. Dans son article [Whi05], Stephen A.White présente un simple exemple de comment un diagramme BPMN peut être utilisé pour générer un processus BPEL.

EPC

La notation EPC "Event-driven Process Chain" est une technique de modélisation de processus métier, développée par August-Wilhelm Scheer au sein du framework ARIS ("Architecture of Integrated Information Systems") [S+92]. Les diagrammes EPC sont principalement utilisés pour configurer une implémentation d'un ERP [2]. "An Event-driven Process Chain (EPC) is an ordered graph of events and functions. It provides various connectors that allow alternative and parallel execution of processes. Furthermore it is specified by the usages of logical operators, such as OR, AND, and XOR. A major strength of EPC is claimed to be its simplicity and easy-to-understand notation. This makes EPC a widely acceptable technique to denote business processes." [TWTR06]

Un diagramme EPC est composé d'éléments tels que les évènements, des fonctions, des données en entrée et sortie, des connecteurs. Ces éléments peuvent trouver leur équivalent dans un autre diagramme de modélisation de processus métier comme BPMN (section 3) ou les diagrammes d'activité (section 2.2.2).

2. EPC : Enterprise Resource Planning

Réseaux de Petri

Les réseaux de Petri sont un outil de modélisation graphique et mathématique applicables à beaucoup de systèmes [Mur89]. Ils sont apparus en 1962, dans la thèse de Carl Adam Petri. Un réseau de Petri est composé de deux types de noeuds : les places et les transitions. Ces noeuds sont reliés par des arcs orientés. Une règle importante de conception étant qu'on ne peut pas relier deux places entre elles ni deux transitions entre elles. Les réseaux de Petri sont très connus et ont fait l'objet de milliers de publications dans le monde scientifique. Par exemple, Murata décrit les propriétés, analyse et applique les réseaux de Petri dans [Mur89] et K.Jensen étudie les réseaux de Petri colorés [Jen87].

IDEF0

"Integration DEFinition for Function Modeling" (IDEF0) est une technique de modélisation pour l'analyse, le développement, l'ingénierie et l'intégration de systèmes d'information et processus métier. Il est utilisé pour montrer le flux de données d'un cycle de vie de processus [Leo99]. En décembre 1993, le "Computer Systems Laboratory" du "National Institute of Standards and Technology (NIST)" ont défini IDEF0 comme un standard de la modélisation de fonction [IDE14]. IDEF0 est composé de deux éléments de base :

- Les fonctions, représentées par des boîtes rectangulaires
- Les données et objets, représentés par des flèches

Ces éléments sont illustrés à la figure 2.7. Les entrées d'une fonction (paramètres) sont indiquées venant de la gauche, les sorties (résultats) sortent du côté droit de la fonction. Une modélisation en utilisant IDEF0 implique de concevoir un diagramme pour chaque niveau de détail. Chaque diagramme étant le raffinement d'un autre, d'où il ressort une structure d'arbre. Plusieurs méthodologies supplémentaires telles que IDEF1, IDEF1X, IDEF3, IDEF4 ou IDEF5 ont été créées, chacune se spécialisant dans une représentation comme l'information, les données, les ontologies ou encore l'orienté objet. La figure 2.8 illustre un exemple de diagramme IDEF0.

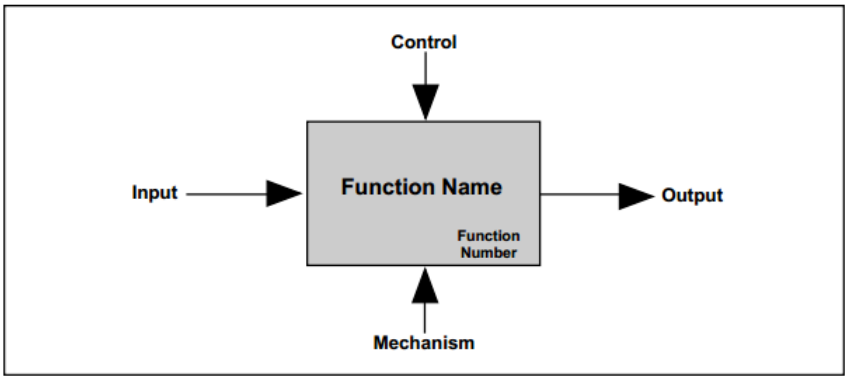

FIGURE 2.7 – IDEF0 : Format de base

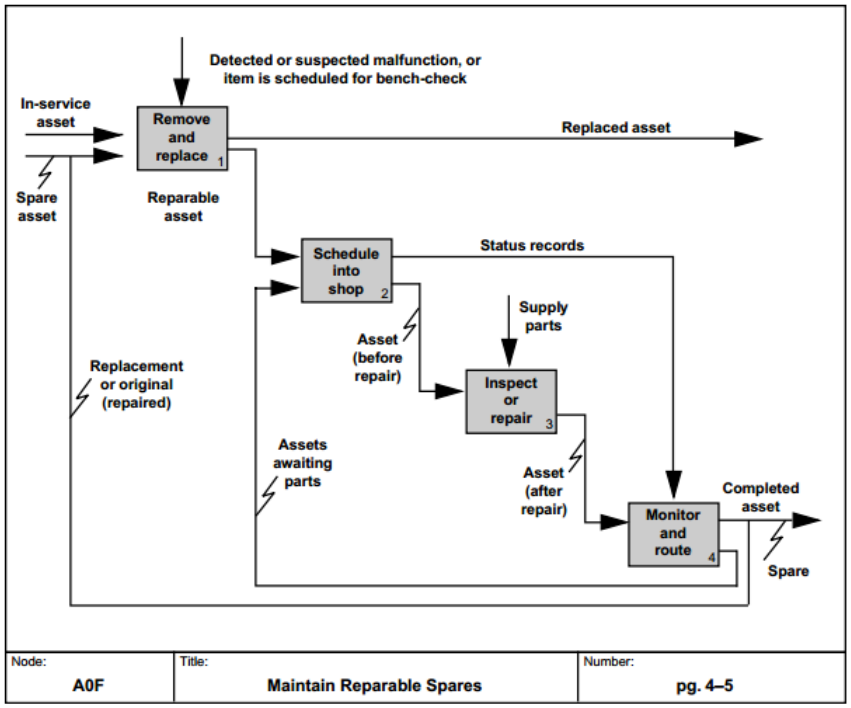

FIGURE 2.8 – IDEF0 : Exemple

Chapitre 3

BPMN

Ce chapitre est une présentation de BPMN 2.0. Afin de comprendre les difficultés d'implémentation de l'éditeur, il est nécessaire de présenter les différents concepts constituant la notation BPMN. Un modèle BPMN, appelé *Processus*, est composé d'un ensemble d'éléments disposés de manière à illustrer une série d'actions nécessaire à la réalisation d'un processus métier.

La première section présente une description générale de BPMN. La description des concepts de BPMN sera définie dans le métamodèle, présenté dans la section suivante. Ensuite, la syntaxe concrète de ces concepts sera recensée dans un inventaire des constructions. Dans le but d'implémentation de l'éditeur, une comparaison des outils d'édition BPMN est présentée dans la section 5. Une introduction à la sémantique de BPMN, nécessaire à la conception du simulateur, terminera ce chapitre.

3.1 Description générale

Afin de documenter les processus métiers, une manière appropriée est d'utiliser une notation, souvent graphique, pour les modéliser. Cependant, dans un monde vaste et en pleine évolution, une documentation doit pouvoir être lisible et compréhensible par tous les intervenants, du développeur au manager en passant par l'analyse. Une notation graphique adaptée est un moyen de casser cette frontière par un langage compréhensible par tous les maillons de la chaîne.

Selon plusieurs études, la documentation est une étape essentielle de la conception logicielle. La conception logicielle un processus complexe qui implique que les différents acteurs doivent documenter un maximum leurs interventions. Cependant, cette vision est souvent idéale et dans la pratique, le monde de l'industrie en général à plutôt tendance à négliger la documentation des processus métiers. Lorsque cette documentation est réalisée, elle n'est valable qu'au moment de sa rédaction et n'est pas mise à jour. De plus, selon B. Silver : "The familiar 80-20 rule sats that 80% of the costs, delays, and errors come from 20% of the instances - the exceptions. So it should be up to the business to specify - in a non-technical way - how any type of exception should be handled : a costumer cancels or changes the order, an item is out of stock, or timeout occurs"[Sil07]. La documentation spécifique des exceptions est primordiale. Afin d'aider à identifier les sources d'exceptions, BPMN propose un large éventail de constructions qui permettra de mettre en valeur ces exceptions.

3.1.1 En général

Plusieurs standards apportent une telle notation, avec divers degrés de précision et de complétude. Les diagrammes d'activité d'UML (Unified Modeling Language) sont une notation capable de représenter les processus métiers. "Cependant, son usage est essentiellement restreint à la conception de logiciels orientés-objet, où UML est le standard accepté." [All11]

Un langage de modélisation plus approprié était nécessaire dans ce domaine. La complétude, la multiplication des acteurs, les différentes exceptions, les différents types de tâches sont des concepts qui doivent pouvoir être représentés au sein d'un processus métier. C'est dans cette optique que BPMN va se développer.

3.1.2 Histoire de BPMN

À l'origine, BPMN (Business Process Modelling Notation) a été conçu par le Business Process Management Initiative (BPMI). Il s'agit d'un cortorsium de plusieurs entreprises principalement développant des logiciels. Le but premier était de fournir une notation graphique au langage BPML (Business Process Modeling Language), développé également par BPMI. Par la suite, BPML ne sera plus maintenu et sera abandonné au profit de BPEL.

La première version de BPMN a été produite par l'équipe de Stephen A. White de la société IBM en 2004. Deux ans plus tard, BPMN version 1.0 a été accepté comme standard OMG. "L'OMG (Object Management Group) est une association américaine à but non lucratif créée en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes. L'OMG est notamment à la base des standards UML (Unified Modeling Language), MOF (Meta-Object Facility), CORBA (Common Object Request Broker Architecture) et IDL (Interface Definition Language)." [Wik13b] Par la suite, en 2008, une version 1.1 a été produite et présente plusieurs changements au niveau graphique.

BPMN est un langage présentant plusieurs constructions. Les principales sont :

- Les évènements
- Les tâches
- Les portes
- Les objets représentant les données
- Les intervenants
- Les sous-processus
- Les flux

Grâce à cette base, il est possible de représenter une grande partie des processus métiers sous tous leurs aspects.

En 2009, la version 1.2 sera délivrée. Depuis 2011, BPMN est à sa version 2.0, beaucoup plus complète que les précédentes. Elle présente plusieurs innovations dont :

- Des nouveaux types d'évènements
- Les portes basées sur les évènements
- Les évènement intermédiaires
- Des sous-processus déclenchables par certains évènements
- La collaboration de plusieurs intervenants
- Un symbole pour chaque type de tâche
- Les diagrammes de collaboration
- Les diagrammes de chorégraphie

Le standard BPMN 2.0 décrit le but principal de BPMN comme : "provide a notation that is

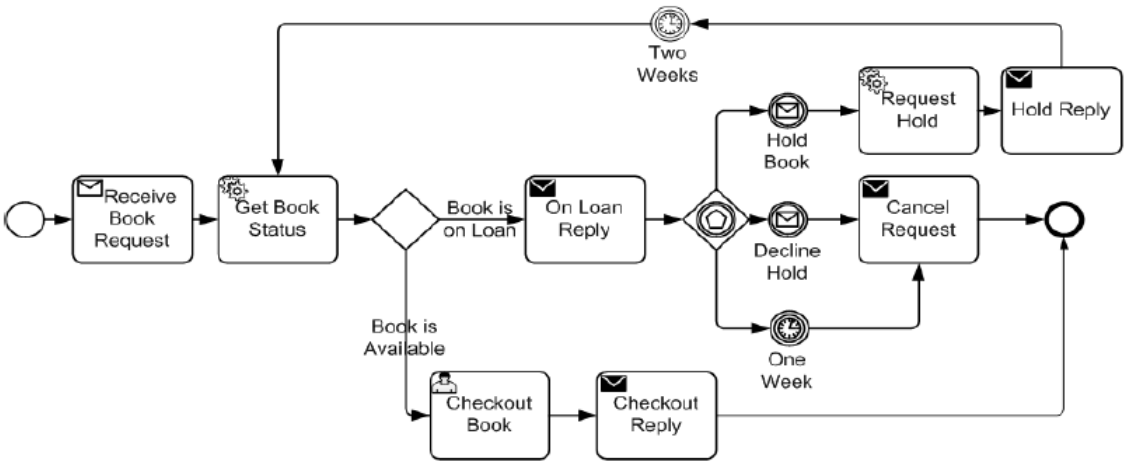

FIGURE 3.1 – BPMN 2.0 : Exemple

readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes"" [OMG11a]. On retrouve bien ici les concepts d'interdisciplinarité. Le standard BPMN 2.0, ne faisant pas moins de 504 pages, définit rigoureusement chaque concept de la notation. Un exemple de diagramme est représenté à la figure 3.1.

La notation BPMN 2.0 est scindée en trois grands types de diagrammes : PROCESS, COLLABORATON et CHOREOGRAPHY. La partie PROCESS est la plus importante et contient tous les éléments de base cités précédemment. La partie COLLABORATION, comme son nom l'indique, permet de réaliser des modèles mettant en situation plusieurs intervenants, s'échangeant des informations au sein d'un processus. La parte CHOREOGRAPHY se concentre sur les interactions entre Participants : "Choreography formalizes the way business Participants coordinate their interactions. The focus is not on orchestrations of the work performed within these Participants, but rather on the exchange of information (Messages) between these Participants" [OMG11a]. Il est important de noter que ces 3 types de diagrammes ne sont pas exclusifs. En effet, il est coutume de voir plusieurs types de diagrammes sur un même processus ; des éléments de PROCESS en relation avec des éléments de CHOREOGRAPHY par exemple.

Cependant, la majeure évolution apportée par la version 2.0 est certainement la définition d'un métamodèle. Les précédentes spécifications n'étaient que sous forme verbale et étaient sujettes à des interprétations différentes. La description de ce métamodèle représente la majeure partie du standard BPMN [OMG11a] et est incontournable pour qui veut acquérir une expérience suffisante pour concevoir un outil permettant de modéliser des diagrammes BPMN. Si bien que depuis la création du métamodèle, le nom BPMN (Business Process Modelling Notation) a changé vers "Business Process Model and Notation". Cela exprime que BPMN est bien plus qu'une simple notation.

De plus, cette nouvelle version propose une sérialisation XML (basée sur XMI et XSD) pour les modèles BPMN, qui se base sur les éléments du métamodèle. Ce qui fait que BPMN se démarque d'autres langages comme WS-BPEL and XPDL. "In fact, the advent of the new

version of BPMN forces many software vendors to change thier tools to become compliant with BPMN 2.0 and this is a time-consuming and expensive activity" [CT12]. Cette mise à niveau, assez volumineuse et différente de la version précédente, implique que les précédents outils compatibles avec BPMN 1.2 doivent parfois revoir toute leur architecture pour mettre en place les nouveautés de la version 2.0. Plusieurs outils, se disant compatibles BPMN 2.0, ne représentent pas l'entièreté du métamodèle. Nous verrons plusieurs implémentations de BPMN et comparerons celles-ci dans la section 3.4.

Plusieurs travaux ont critiqué BPMN par rapport aux éléments, trop nombreux. D'un côté, un tel langage de modélisation se doit d'être complet afin de pouvoir être compatible avec le plus de situations possibles, mais d'autre part, multiplier le nombre de constructions rend le langage difficile à aborder pour un novice, voire familier avec BPMN. Par exemple, juste pour les Tasks, il existe 336 symboles représentables [CT12]. En effet, chaque élément peut être combiné avec un autre, comme les Tasks et les Events. En prenant en compte le nombre de Tasks différentes ainsi que le nombre d'Intermediate Event, on mesure aisément la panoplie de combinaisons possibles. C'est une difficulté supplémentaire à prendre en compte lors de la réalisation d'un outil compatible avec BPMN 2.0.

3.1.3 Utilisation

D'après une étude réalisée par [CT12], BPMN 2.0 est utilisé par 40% des interviewés. BPMN est utilisé dans un but de documentation à 52% et d'exécution à 37%. On peut en retirer, en prenant compte que nous n'avons pas les détails de l'étude, que BPMN est largement utilisé. Cette étude a également porté sur l'usage des différents éléments de la notation. Il ressort aussi de cette étude que 82.35% des interviewés déclarent avoir étendu la notation BPMN avec des éléments personnalisés. Ce chiffre nous intéresse particulièrement dans le cadre de ce travail, car l'implémentation de BPMN dans un outil de métamodélisation permet d'adapter plus facilement le langage selon les besoins de l'utilisateur. Beaucoup plus facilement qu'en utilisant un logiciel entièrement dédié à BPMN, nous y reviendrons.

3.1.4 Analyse de l'efficacité cognitive

Dans [GHA11], les auteurs ont voulu s'intéresser à l'efficacité cognitive de BPMN 2.0 en analysant les éléments graphiques grâce à plusieurs techniques. "Cognitive effectiveness is defined as the speed, ease and accuracy with which a representation can be processed by the human mind" [Moo09]. L'apparence graphique des éléments d'un langage joue un rôle prépondérant dans l'efficacité. En effet, mit en balance avec le grand nombre d'éléments de BPMN2.0, il est primordial d'avoir une approche qui permette de distinguer les éléments entre eux et que leur forme soit équivoque. Il ressort de leur étude plusieurs résultats intéressants. Premièrement, ils constatent que 171 symboles composent les diagrammes PROCESS, ce qui est beaucoup plus que le modèle entité-relation (5 symboles) ou encore YAWL (14 symboles), concurrent de BPMN, nous y reviendrons dans la section 2.2.2. Ce qui montre bien que BPMN est plus fourni que ces concurrents. Deuxièmement et toujours dans la même lignée, ils constatent que 23.6% des métaobjets composant le métamodèle de BPMN 2.0 n'ont pas de correspondance graphique. Aussi, 5.4% des symboles sont utilisés pour représenter plusieurs constructions, surchargeant la notation. [GHA11] Nous constaterons l'importance de ce résultat dans ce travail lors de la réalisation de l'éditeur, se basant sur ce métamodèle. De plus, concernant la transparence sémantique des éléments, signifiant la mesure dans laquelle

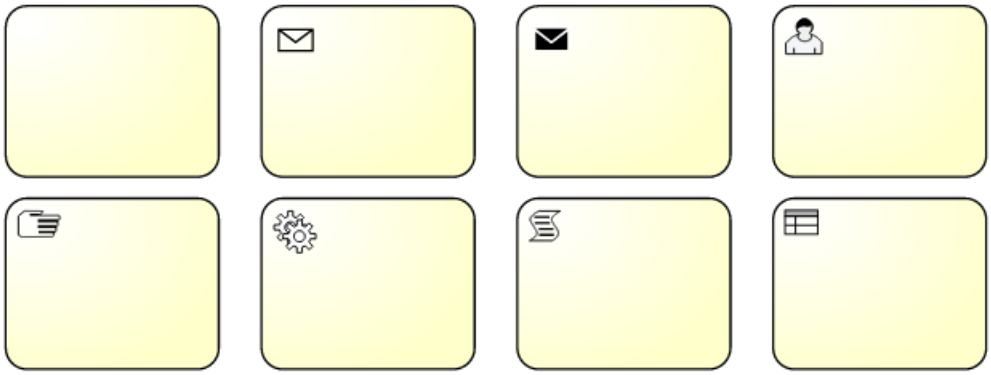

FIGURE 3.2 – BPMN 2.0 : Tâches

la sémantique d'un élément est bien équivoque à travers un symbole, les auteurs proposent quelques améliorations à apporter afin de remédier à des symboles dont la sémantique n'est pas clairement exprimée. Ils proposent de remplacer des symboles par d'autres, plus équivoques. Dans la suite de ce travail, ces changements seront intégrés à l'éditeur. Pour conclure, nous voyons qu'une notation n'est pas efficace cognitivement dans toutes ses constructions. C'est pourquoi il est nécessaire d'avoir un outil permettant d'adapter rapidement, au besoin, le langage BPMN.

3.2 Inventaire des constructions

3.2.1 Liste des éléments

Tâches

La figure 3.2 illustre les différents types de tâches. Une tâche représente un travail, une action effectuée au sein d'un processus. Chaque type de tâche illustre une spécificité supplémentaire par rapport à une tâche abstraite.

Sous-processus

Avec les tâches décrites au point précédent, les sous-processus forment un ensemble que l'on appelle les *Activités*. Les sous-processus sont des tâches complexes qui sont souvent décrites par un processus à part, ou en dépliant le sous-processus (souvent illustré par un clic sur le "+"). Dans l'exemple de référence, le sous-processus "Entrer commande" cache un processus complexe composé des étapes typiques de constitution d'une commande (Sélection des produits, Identification du client, Acceptation des conditions générales, Paiement). La décision de représenter ou non ces tâches complexes en sous-processus revient au concepteur selon le niveau d'abstraction qu'il souhaite illustrer dans le modèle.

Évènements

La figure 3.4 illustre les différents types d'événements BPMN 2.0. Ils sont distingués en trois catégories. "Catching" signifie que l'événement est capturé, reçu. À l'inverse, "Throwing"

Type	Explication
Tâche abstraite	Unité de base de travail
Tâche message (réception)	Représente une tâche en attente d'un message provenant d'un participant externe au processus
Tâche message (envoi)	Représente une tâche envoyant un message à un participant externe au processus
Tâche utilisateur	Représente une tâche réalisée par un humain avec une aide logicielle
Tâche manuelle	Représente une tâche réalisée par un humain sans support applicatif
Tâche service	Représente une tâche utilisant un service Web ou une application automatisée
Tâche script	Représente une tâche réalisée par un moteur de processus métiers
Tâche "Business Rule"	Représente une tâche réalisée par un groupe de travail

FIGURE 3.3 – BPMN 2.0 : Types de sous-processus

Type	Explication
Loop	Détermine un processus répétitif
Multi-Instance	Détermine un processus répétitif dont les instances s'exécutent en parallèle
Compensation	Représente un sous-processus de compensation en réaction à une activité à compenser dans le cadre d'une transaction échouée
Ad-Hoc	Représente un sous-processus dont les tâches internes sont effectuées dans un ordre non déterminé
Compensation et Ad-Hoc	Représente un sous-processus de compensation dont les tâches internes sont effectuées dans un ordre non déterminé

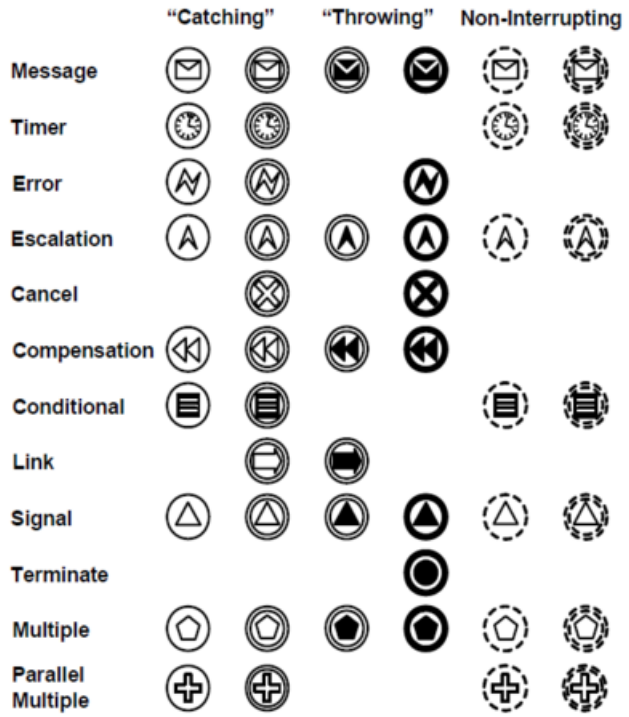

FIGURE 3.4 – BPMN 2.0 : Événements

FIGURE 3.5 – BPMN 2.0 : Données

signifie que l'événement est lancé, émis. "Non-interrupting" signifie que le processus n'est pas interrompu par le déclenchement de cet événement.

De plus, les événements ayant un cercle simple doivent démarrer un processus, ceux ayant un cercle gras doivent terminer un processus. Les événements intermédiaires, entourés par un double cercle plein ou interrompu, doivent se trouver au sein d'un processus entre un événement de départ et un événement de fin.

Données

La figure 3.5 représente les éléments "données". Les données sont des éléments physiques ou des informations qui sont créés, manipulés ou utilisés lors de l'exécution d'un processus.

Portes

La figure 3.6 représente les différents types de portes BPMN. Les portes sont des éléments capables de contrôler le flux de séquence d'un processus comme un choix entre deux alternatives ou l'exécution de plusieurs tâches en parallèle.

Type	Explication
Abstract	Événement sans type particulier, qui indique un point de départ.
Message	Réceptionne / Envoi un message
Timer	Événement cyclique indiquant une durée ou un point dans le temps.
Error	Réceptionne / Envoi une erreur
Escalation	Passage à un niveau de responsabilité supérieur.
Cancel	Réaction par rapport à une transaction annulée
Compensation	Gestion ou déclenchement de compensation
Conditional	Réaction par rapport à une condition, règle
Link	Connecte deux évènements afin de mieux répartir l'agencement graphique
Signal	Signalisation sur plusieurs processus
Terminate	Arrête toute instance du processus en cours
Multiple	Réception / Envoi d'un événement parmi un jeu d'événements
Parallel Multiple	Réception / Envoi de tous les événements parmi un jeu d'événements

Type	Explication
Données	Représente une information passant par le processus
Données (collection)	Représente une collection d'information passant par le processus
Données (Input)	Représente une information externe entrante au processus
Données (Output)	Représente une information externe sortante au processus
Données (Data Store)	Représente un répertoire de données comme une base de données où le processus peut lire et écrire

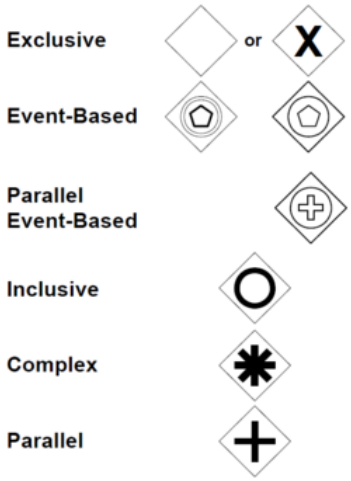

FIGURE 3.6 – BPMN 2.0 : Portes

Type	Explication
Exclusive	Signifie un choix exclusif (1 et 1 seul) entre plusieurs flux possibles
Basée événement	Signifie un choix exclusif basé sur un déclenchement d'événement
Basée événement (parallèle)	Signifie que les flux ultérieurs sont instanciés parallèlement après le déclenchement d'événement
Inclusive	Représente un choix inclusif (1 ou plusieurs) entre plusieurs flux possibles
Complexe	Représente un choix exprimé par une condition inexprimable par les autres portes disponibles
Parallèle	Signifie que les flux ultérieurs sont instanciés parallèlement

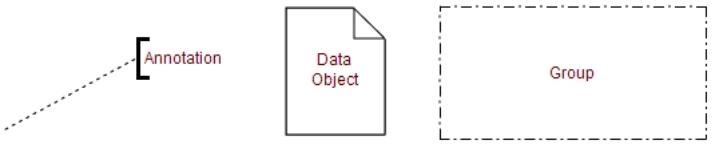

FIGURE 3.7 – BPMN 2.0 : Artéfacts

Type	Explication
Annotation textuelle	Représente une annotation, un commentaire associé à un élément d'un processus
Données	Voir 3.2.1
Groupe	Représente un ensemble d'éléments appartenant à un groupe particulier (aide visuelle)

Artefacts

Les artéfacts, représentés à la figure 3.7, sont des éléments permettant d'ajouter de l'information à un processus, afin d'être mieux compréhensible. Les données, décrites au point précédent, permettent de distinguer les informations circulant dans un processus. Les annotations indiquent un supplément d'information à n'importe quel élément BPMN. Les groupes peuvent regrouper certains éléments d'un processus ayant des points communs. Les annotations et les groupes apportent une meilleure lisibilité à un modèle BPMN s'ils sont utilisés judicieusement sans surcharger le diagramme.

Pool et Lanes

La figure 3.8 représente les "Pool" (Participants pouvant être assimilés à une société) à un processus BPMN 2.0. Les "Lanes" (Couloirs) représentent des sous-entités au sein des participants au processus. Dans l'exemple de référence, le pool "Processus commande" est partagé en deux lanes "Ventes" et "Exécution". Ces éléments sont utilisés pour identifier les exécutants des tâches ainsi que les relations entre-eux.

FIGURE 3.8 – BPMN 2.0 : Pools et Lanes

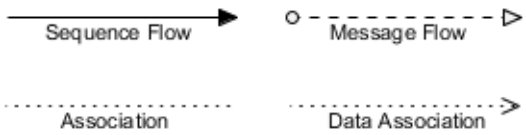

FIGURE 3.9 – BPMN 2.0 : Types de flux

Type	Explication
Sequence Flow	Connecte les éléments au sein d'un processus
Message Flow	Connecte les Pools (Participants) entre eux ainsi que les "Message Event" aux participants
Association	Connecte un élément avec une annotation ("Text Annotation")
Data Association	Connecte les éléments avec les données en particulier

Flux

3.2.2 Constructions spécifiques (Challenges)

Boundary Events En plus des connexions entre éléments via les flux de séquence, il existe une construction particulière, les événements en bordure d'activités. Comme le montre la figure 3.10, ce type d'événement particulier peut être disposé à la bordure de tous les types de tâches ainsi que de sous-processus. Il indique, selon son type (message, erreur, etc..), l'événement pouvant se produire durant l'exécution de l'activité.

Extended SubProcesses Il existe deux représentations graphiques pour les sous-processus. La première, est celle illustrée à la figure 3.11, où le sous-processus est replié. La deuxième, illustrée à la figure 3.12, montre le déroulement du sous-processus déplié.

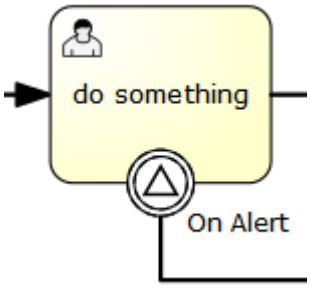

FIGURE 3.10 – BPMN 2.0 : Boundary Event exemple

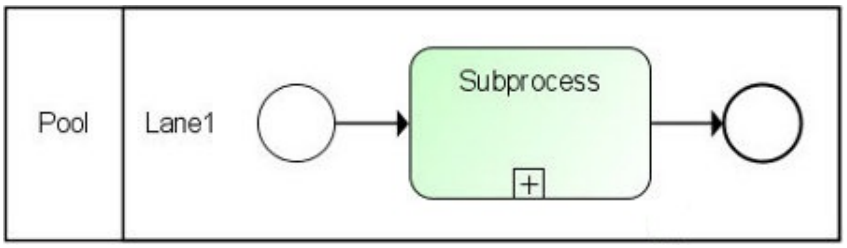

FIGURE 3.11 – BPMN 2.0 : Sous-processus replié

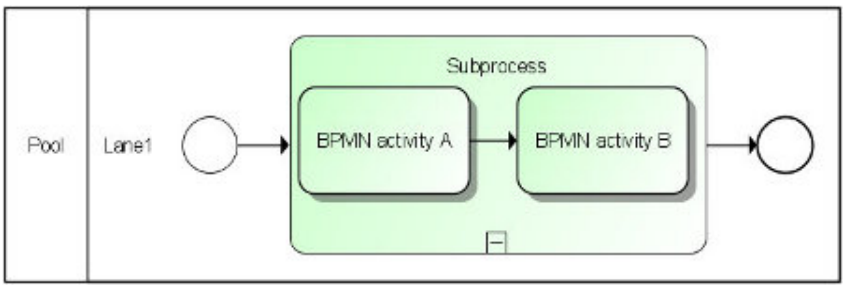

FIGURE 3.12 – BPMN 2.0 : Sous-processus déplié

3.3 Métamodèle

3.3.1 Description

Pour rappel, un métamodèle est une représentation des concepts d'une technique de modélisation. A l'aide de la syntaxe concrète, il permet de créer des modèles du langage qu'il décrit. Il est généralement représenté à l'aide d'un diagramme de classe UML. Ces diagrammes de classe sont composés de trois briques de base : les classes, les attributs et les relations.

Depuis la version 2.0, BPMN est défini par un métamodèle. Il comprend notamment les différents éléments décrits à la section 3.2.1. Cependant, la taille du métamodèle est trop importante pour être décrit entièrement dans cette section. En effet, il contient 147 classes UML. Chacunes de ces classes ont aussi des attributs et des relations avec les autres classes. Le métamodèle complet est illustré à la figure 3.14. Le standard BPMN [OMG11a] décrit rigoureusement chaque élément composant le métamodèle en pas moins de 300 pages.

Afin de pouvoir fournir un aperçu de sa structure, une version très simplifiée a été réalisée et se trouve à la figure 3.13. Dans cette version, beaucoup d'éléments, de propriétés et de relations ont été retirées. Cette structure, décrite ci-après, représente les éléments de base de la partie PROCESS de BPMN.

Cette version simplifiée du métamodèle comporte trois grandes parties. La première comporte l'ensemble des fils de *FlowElement*. Nous observons que l'élément fils, *FlowNode*, est le père des éléments *Event*, *Activity* et *Gateway* qui sont les élements décrit dans la section 3.2.1. L'élément *SequenceFlow* représente un flux de séquence qui est représenté graphiquement par une flèche pleine. Les 2 relations 1..N entre *SequenceFlow* et *FlowNode* expriment la construction d'un processus. Un élément *FlowNode* (un événement, une activité ou une porte) peut être lié à un ou plusieurs flux de séquence en entrée ou en sortie. On peut noter que chaque *FlowElement* a une propriété appelée `name`. Dans ce cas, c'est cette propriété qui sera affichée en texte sur le symbole. La deuxième partie comprend les éléments *Lane* et *LaneSet*. Ils font référence aux Lanes et aux Pool représentant des entités participant à un processus. En suivant les relations, chaque *LaneSet* comprend zéro ou plusieurs *Lane*, qui comprend zéro ou plusieurs *FlowNode*. La troisième partie concerne l'élément *FlowElementsContainer*. C'est un élément abstrait qui illustre un conteneur de *FlowElement*, comme l'indique la relation entre ces deux éléments. BPMN propose deux types de conteneurs, les *Process* et les *SubProcess*. *Process* fait référence à un processus BPMN, un modèle tel que notre exemple de référence 1.3. *SubProcess* référence un sous-processus pouvant contenir des *FlowElement*, visibles lorsqu'il est déplié. On peut remarquer l'élément *Artifact* référencant les artéfacts comme les annotations ou les groupes. Comme l'illustre les relations, un *FlowElementsContainer* est composé de zéro à plusieurs *Artifact*. Aussi, BPMN défini une racine commune à tous les éléments du métamodèle, l'élément *BaseElement*.

3.3.2 Critique

N.Genon [GHA11], a analysé la notation BPMN ainsi que le métamodèle. Il en ressort plusieurs constations. BPMN comporte beaucoup plus de symboles par rapport aux autres langages de même type comme YAWL (14 symboles) ou l'entité-relation (5 symboles). D'après l'étude, N.Genon recense 174 symboles composant la notation BPMN ainsi que 242 constructions possibles. Dans le cadre de l'étude, une partie concernait l'analyse de la clarté sémiotique.

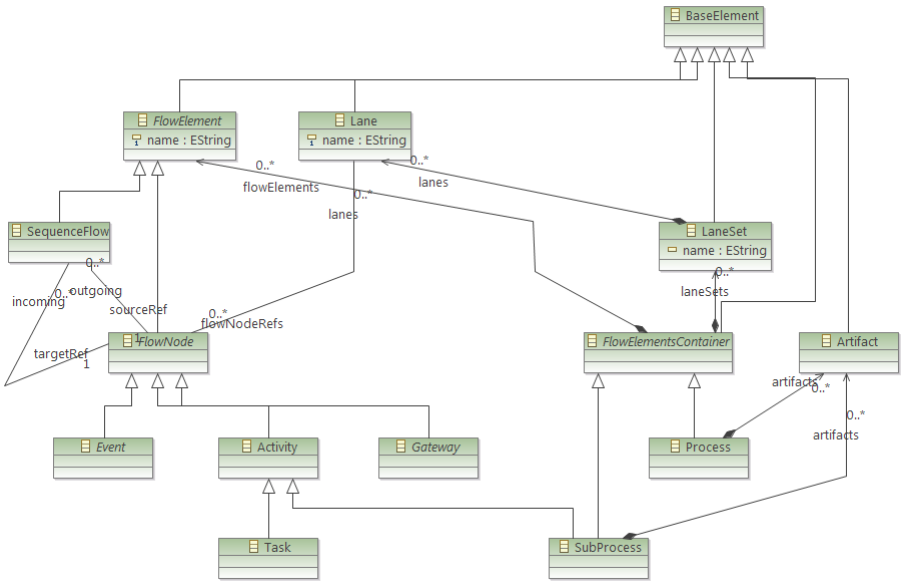

FIGURE 3.13 – BPMN 2.0 : Métamodèle simplifié

La clarté sémiotique signifie qu'il doit y avoir une relation 1 à 1 entre les constructions sémantiques et les symboles graphiques. Autrement dit, chaque classe du métamodèle doit avoir un symbole graphique associé. Il ressort deux résultats significatifs. Premièrement, 23.6% de déficit des symboles. Ce qui signifie que 23.6% des classes du métamodèle n'ont aucun symbole graphique qui leur est associé. Par exemple, nous avons vu l'élément *Gateway* qui n'a pas de correspondance graphique. En effet, ce dernier a un fils par type de porte comme *ExclusiveGateway* ou *ParallelGateway* qui ont bien un symbole graphique associé. Deuxièmement, 5.4% de surcharge de symboles. C'est-à-dire que 5.4% des éléments du métamodèle ont deux symboles graphique associés. Par exemple, l'élément *ExclusiveGateway* peut être représenté par un losange vide et par un losange marqué d'une croix à l'intérieur (voir figure 3.6).

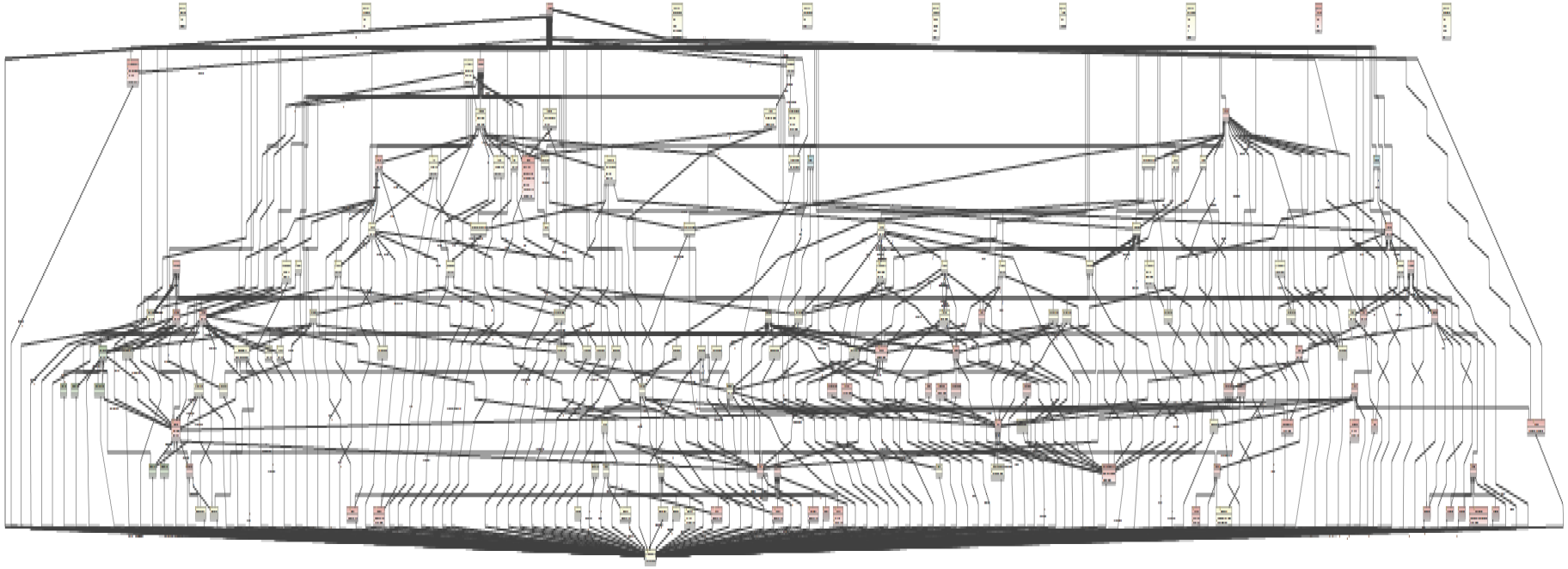

FIGURE 3.14 – BPMN 2.0 : Métamodèle complet

3.4 Comparaison des outils existants

3.4.1 Introduction

Dans le but de créer un nouvel éditeur, il est nécessaire d'avoir une vision panoramique des éditeurs BPMN existants. Cette section comporte une analyse et une comparaison de six éditeurs BPMN populaires :

- Signavio Process Editor
- Microsoft Visio
- BPMN Web Modeler
- Bizagi Process Modeler
- OpenFlexo
- Bonita BPM

Afin de ne pas se perdre dans l'énumération des fonctionnalités différentes et propres à chaque outil, une brève présentation ainsi qu'un aperçu des points forts et faibles de chaque outil sera décrit dans les sous-sections suivantes. La comparaison s'est focalisée sur les critères suivants dans un premier temps :

- La complétude des symboles BPMN 2.0
- L'organisation des modèles
- Les importations et exportations de modèles
- La validation des modèles
- Les liens entre modèles
- La simulation/animation des modèles

Ensuite, afin d'approfondir les fonctionnalités attendues d'un tel outil, nous verrons si les outils en proposent d'autres plus avancées comme :

- La modification des symboles BPMN (forme - couleur)
- Extensibilité - Interaction avec une autre forme de modélisation
- Adaptation à un contexte particulier (nouveaux éléments et constructions)

3.4.2 Constatations communes

Il ressort de l'ensemble des outils plusieurs constatations communes.

L'accès aux éléments BPMN afin de réaliser un modèle se fait toujours par un menu où les éléments se trouvent sous forme d'icônes. Ce menu est la plupart du temps situé sur la gauche de l'écran, voire dans la partie supérieure. Ensuite, le déplacement des éléments se fait par le système "drag&drop", classique pour ce type d'éditeur.

L'ensemble des outils propose une vérification de la syntaxe BPMN 2.0. En d'autres termes, ils sont capables de vérifier l'adéquation d'un modèle créé par l'utilisateur à un ensemble de règles de bonne construction définies dans le standard BPMN [OMG11a]. Par exemple, une règle spécifie qu'un évènement de fin de processus (EndEvent) ne peut avoir aucun flux sortant. Le modèle est soit vérifié en cours de construction, soit après une action de l'utilisateur par un clic sur un bouton illustrant la fonctionnalité. De plus, certains outils identifient et localisent l'erreur sur le diagramme. En interagissant avec ce marqueur, l'outil montre à l'utilisateur un rappel de la règle à appliquer afin de résoudre l'erreur.

3.4.3 Présentation des outils

Signavio Process Editor

Signavio [1] est un logiciel de modélisation de workflow permettant l'édition de modèles BPMN, la modélisation en EPC , Réseaux de Petri, YAWL et d'autres langages. C'est un outil récent, l'édition professionnelle a été lancée fin 2009 et compte aujourd'hui plusieurs milliers d'utilisateurs. À l'initiative de la BPM Academic Initiative, Signavio propose une application en ligne gratuite pour les utilisateurs du monde académique. C'est cette édition (version 8.0.1) qui a été analysée ici.

Signavio implémente l'entièreté des constructions BPMN 2.0 séparées dans les trois types de diagrammes (Process, Collaboration et Choreography). L'utilisateur dispose d'un espace de travail en ligne où sont stockés ses modèles. Les modèles peuvent être interconnectés, c'est-à-dire qu'un sous-processus d'un modèle peut être lié à un autre modèle décrivant son fonctionnement interne. Chaque modèle est versionné et peut être comparé aux autres modèles de l'espace de travail. L'outil permet d'importer et exporter des modèles BPMN à travers le format XML défini dans le standard BPMN ainsi qu'en XPDL 2.1. De plus, il propose une exportation de chaque modèle en PNG, SVG ou PDF.

Par rapport à l'éditeur BPMN, en fonction du type de modèle choisi, l'utilisateur construit classiquement un modèle par Drag&Drop. L'éditeur vérifie la syntaxe en BPMN 2.0, identifie la règle à appliquer en cas d'erreur. Par un tableau en bas de page, il recense les erreurs du diagramme par gravité allant de l'oubli d'un nom de tâche jusqu'à une infraction aux règles de construction BPMN. Cet outil se distingue des autres par une fonctionnalité "Signavio Best Practices" qui applique une série de règles de bonnes pratiques de construction et propose des améliorations aux modèles. De plus, il propose une restructuration de diagramme "BPStruct" qui simplifie les diagrammes. Par exemple, BPStruct peut supprimer les flux inatteignables ainsi que les portes redondantes.

Du point de vue de la simulation, Signavio propose un programme de simulation basé sur des scénarios. Pour un modèle particulier, l'utilisateur peut créer plusieurs scénarios qui prennent en compte plusieurs paramètres comme le coût d'exécution des tâches, le temps d'exécution ou encore les ressources à disposition. De manière interactive, la simulation propose une complétion de chaque élément du modèle en l'entourant d'un trait bleu rempli selon si l'élément est atteint par le flux d'exécution. L'utilisateur a le choix entre le mode "Step by Step" où il fait progresser le flux en cliquant sur chaque élément pour le réaliser, ou bien le mode "Case" où l'entièreté du modèle est simulé sans interaction.

Point de vue modifiabilité, Signavio ne permet pas l'introduction de nouveaux éléments au sein des modèles BPMN. Hormis le changement de couleur, il ne permet pas la modification des différents éléments BPMN 2.0. L'outil reste fixé au langage BPMN mais est très complet et facile d'utilisation pour un utilisateur professionnel.

Bizagi Process Modeler

Bizagi est un outil offline entièrement gratuit dédié à l'édition de modèles BPMN 2.0. L'interface d'édition est très proche de celle de Microsoft Visio mais est spécialisée uniquement pour BPMN. C'est un outil très complet qui offre les fonctionnalités de base d'un éditeur.

Lors du test de l'outil, il n'a pas été possible de réaliser les deux types de diagrammes

1. Signavio : http ://www.signavio.com/

BPMN 2.0 : Collaboration et Choreography. Ce type de diagramme n'apparait ni dans les symboles disponibles pour créer les modèles ni dans la documentation de l'outil. Par contre, toutes les constructions "Processus" sont bien implémentées.

Du point de vue de la validation des diagrammes et de l'interopérabilité, Bizagi est au même niveau que ses concurrents. L'outil de simulation permet de simuler par différents scénarios. Chaque scénario est défini par un ensemble de paramètres tels que les ressources disponibles, le temps et le coût d'exécution des différentes tâches.

Concernant l'adéquation avec le métamodèle BPMN 2.0, Bizagi ne propose pas l'édition des propriétés relatives à chaque élément. Par exemple, une tâche a comme propriété un nom, une description et ses exécutants. Nous avons vu dans la partie du métamodèle BPMN (voir section 3.3) qu'il y a d'autres propriétés, non éditables ici. L'outil ne permet pas l'exportation dans le format BPMN XML. Néanmoins, l'exportation et importation en XPDL et Visio est implémentée.

Bizagi permet d'introduire un nouvel élément personnalisé dans l'ensemble des modèles, ce que les autres outils ne proposent pas. Cette fonctionnalité est basique parce qu'un nouvel élément est caractérisé par un nom et par une image. En effet, il n'est pas possible d'associer des propriétés supplémentaires aux nouveaux éléments, ni de définir une représentation en affichant le nom de l'élément sous l'image lors de l'insertion dans le modèle. Par rapport aux éléments BPMN, ils ne sont pas modifiables, sauf uniquement en changeant de couleur.

Microsoft Visio

Microsoft Visio, utilisant le même type d'interface que la suite Office, propose aussi un éditeur BPMN. Cependant, l'éditeur ne propose que les éléments composant le premier type de diagramme BPMN, Process.

L'éditeur utilise le principe d'affinage par un clic droit. Un élément abstrait est sélectionnable et par clic droit, l'utilisateur peut choisir une forme plus spécifique. Par exemple, affiner une tâche abstraite en tâche manuelle.

Point de vue de la vérification, l'outil identifie les erreurs et propose la règle à appliquer pour la corriger. Concernant les liens entre diagrammes, il est possible de lier un sous-processus vers un autre diagramme Visio. Une différence par rapport aux autres outils est que Visio ne peut importer ou exporter que par le format "Visio WorkFlow Interchange". Seul l'export en image PNG ou JPG est possible.

À ce jour, l'éditeur ne propose pas de fonctionnalité de simulation de processus. Comparé aux autres éditeurs, Visio propose beaucoup moins de fonctionnalités. On peut expliquer ce manque par le fait que Visio est un éditeur générique se spécialisant selon le langage de modélisation utilisé.

BPMN Web Modeler

BPMN Web Modeler est un outil d'édition BPMN entièrement en ligne. Étant payant, c'est la version d'essai qui a été testée dans le cadre de ce travail. Il est compatible BPMN 2.0 et rassemble tous les éléments des 3 types de diagrammes au sein d'un même éditeur. De même que Visio, il utilise la technique d'affinage par clic droit.

Point de vue interopérabilité, il gère l'importation et exportation en BPMN 2.0 (XML), XPDL (2.1,2.2,3.0) et "Visio WorkFlow Interchange". De plus, il permet l'exportation en

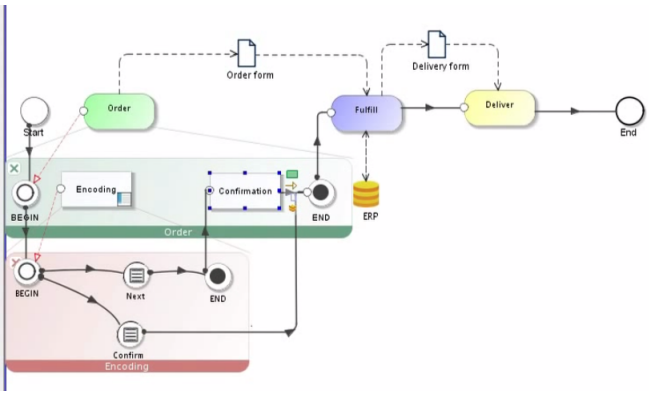

FIGURE 3.15 – OpenFlexo : Processus

format BPEL, EMF et image (JPG, PNG, PDF). Les erreurs des modèles sont identifiées et la règle à appliquer est montrée à l'utilisateur.

Les diagrammes peuvent être liés au sein d'un workspace par utilisateur. L'outil peut mettre à disposition un workspace commun à une équipe de travail dans un package payant. Dans ce package, BPMN Web Modeler propose une édition collaborative de diagrammes (TeamWork) où plusieurs utilisateurs peuvent travailler sur un même modèle en même temps.

La simulation n'est accessible que par paiement et n'a donc pas été traitée.

OpenFlexo

OpenFlexo propose un éditeur BPMN avec une approche différente des autres outils classiques.

Les éléments de "Process" BPMN 2.0 sont disponibles dans l'éditeur. Trois vues sont utilisées. La première sert à modéliser le processus. La seconde permet de créer des acteurs. La troisième rassemble les deux précédentes et montre le processus en relation avec les acteurs sous forme de Pool. Cette approche est différente des autres outils qui utilisent les Pools comme des éléments à placer dans le processus.

OpenFlexo permet aussi de raffiner le déroulement d'une tâche au sein d'un processus. Une tâche peut être divisée en une série d'opérations. De plus, chaque opération peut être liée à une illustration appelée une Story-board, qui permet de la décrire sous forme d'une histoire et d'images. L'ensemble apparaît comme une BD plus facilement compréhensible pour l'exécutant de la tâche. Le processus illustré à la figure 3.15, comporte la tâche "Order" rafinée en opérations. L'opération "Encoding" est décrite sous forme de Story-board, représenté à la figure 3.16. Elle illustre qu'un utilisateur peut choisir ses articles en ligne, les placer dans un caddie en choisissant les articles dans le catalogue en ligne. C'est la première fois qu'un outil permet ce genre de fonctionnalité en tentant d'étendre un modèle BPMN vers un domaine particulier.

L'outil met à disposition un éditeur de documentation pour les processus. En effet, une documentation au format Word peut être générée pour chaque processus. Contrairement aux autres outils, cette documentation est personnalisable au sein de l'outil. L'utilisateur peut définir un canevas dans le lequel son modèle BPMN sera exporté. Une partie est générée automatiquement, l'autre est personnalisable via un éditeur interne au logiciel.

D'un premier abord, il n'existe pas d'interface permettant de modifier l'apparence d'un

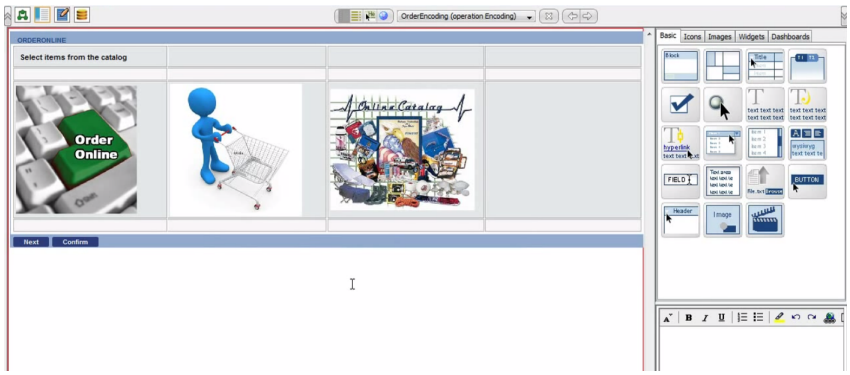

FIGURE 3.16 – OpenFlexo : Story-board

élément BPMN ni d'introduire de nouveaux éléments. Cependant, OpenFlexo propose des symboles non présents de base dans BPMN comme une "porte IF" ou une "porte boucle". L'entreprise OpenFlexo peut adapter l'éditeur à un domaine particulier, mais la fonctionnalité n'est pas accessible depuis l'outil.

Les fonctionnalités de base ne sont pas présentes, mais le but de cet outil est de présenter une approche personnalisée et non d'offrir un éditeur commercial de plus.

Bonita BPM

Bonita BPM, actuellement dans sa version 6.3, est une suite Open Source de Gestion des processus métier très populaire créée en 2001. Depuis 2009, c'est la société dédiée entièrement à ce logiciel, BonitaSoft qui soutient le développement de l'outil.

Bonita est composé de trois éléments. Le Studio Bonita permet de modéliser les processus selon le standard BPMN ainsi que plusieurs fonctionnalités avancées qui seront décrites plus loin. Ensuite, le moteur BPM permet d'exécuter les processus et est distribué sous licence LGPL. Enfin, le portail (Bonita User Experience) permet à chaque utilisateur de recevoir des messages concernant les tâches auxquelles il est associé. Le propriétaire d'un processus peut aussi piloter l'exécution grâce à un reporting graphique. [Wik13a]

Concernant les points de comparaison, Bonita apparaît supérieur aux autres outils. L'outil gère l'entièreté de BPMN 2.0, même les diagrammes de *Collaboration* et de *Choreography*. La validation du diagramme se fait lors de l'édition. Des indications apparaissent en relief sur les éléments et indiquent le problème ainsi que la règle à appliquer. Les différents diagrammes sont liés au sein d'un repository qui peut être commun à plusieurs utilisateurs. Ce repository permet l'édition collaborative des diagrammes. Point de vue de l'importation et exportation des modèles, Bonita gère le format BPMN XML, JBPM (JBoss), XPDL et Visio.

Point de vue de la complétude fonctionnelle, l'outil propose notamment la génération de documentation pour chaque processus. Des connecteurs permettent de connecter Bonita à des systèmes d'information existants comme une base de données SQL, des Web Services ou à un logiciel SAP. Une fonctionnalité permet de lier le serveur LDAP de l'entreprise afin de définir une structure d'organisation avec les acteurs (utilisateurs) et établir le mapping entre les acteurs des modèles et ceux de la structure.

La simulation des processus est très complète. Il est possible d'exécuter différemment les processus dans plusieurs scénarios en variant des paramètres comme le coût, la durée, les

ressources disponibles ou le calendrier. De plus, l'outil propose des indications d'optimisation des processus en générant des rapports de simulation détaillés.

Bonita est donc bien placé en termes de complétude fonctionnelle. Cependant, l'outil semble fixé et non modifiable. Les éléments BPMN ne peuvent pas être modifiés sauf en changeant de couleur comme les autres éditeurs le proposent. Il n'est pas possible d'introduire de nouveaux éléments au sein des processus. De plus, la palette rassemblant les éléments BPMN utilise des icônes non standards au lieu des éléments en taille réduite. Aussi, les éléments ne sont pas classés par catégorie : un événement *Link* ne se trouve pas avec les autres événements mais avec les portes, ce qui peut être perturbant pour l'utilisateur non expert.

3.4.4 Conclusion

Nous voyons que les quatre premiers outils proposent le même genre de fonctionnalités de base, avec une interface différente. Certains tentent de se démarquer en proposant des fonctions plus avancées comme Signavio. Des outils (Signavio) ont choisi de respecter le métamodèle BPMN 2.0 à la lettre en proposant toutes les propriétés éditables de chaque élément, au contraire de Bizagi qui a une approche beaucoup plus simplifiée et ne propose que quelques propriétés de base (nom description) pour chaque élément. En effet, devant la complexité du métamodèle, les outils ont tendance à choisir une approche plus simple et axée sur la modélisation, ce qui permet de ne pas surcharger l'interface avec des propriétés parfois inutiles.

D'autre part, nous avons vu que OpenFlexo se démarque des autres outils en implémentant d'autres fonctionnalités comme la personnalisation de documentation ou l'illustration d'une tâche en une série d'opérations sous forme de Story-board.

D'une manière générale, les outils proposent les mêmes fonctionnalités génériques. Il ressort un manque de possibilités de personnalisation du langage BPMN à un domaine particulier. En utilisant de tels outils, le modélisateur ne peut utiliser que les formes de base de BPMN qui n'expriment pas toujours sa volonté.

3.5 Sémantique de BPMN

Cette section présente un aperçu de la sémantique d'exécution de BPMN 2.0. La sémantique d'exécution d'un langage est la série d'opérations à réaliser par un moteur pour exécuter un processus de son instanciation jusqu'à sa terminaison. Cette partie se base sur le chapitre 13 du standard BPMN [OMG11a] : "BPMN Execution Semantics", décrivant rigoureusement le comportement de chaque élément BPMN au sein d'un processus. Un aperçu de ces éléments sera vu dans cette partie, cependant, certains seront omis, car étant des cas trop particuliers.

Le but de cette section est de parvenir à produire un embryon de simulateur au sein de l'éditeur BPMN à produire. Cependant, le standard définit la sémantique des éléments de manière précise et selon des propriétés propres à chaque élément. Afin de produire un moteur d'exécution, un éditeur doit pouvoir utiliser ces propriétés de manière adéquate lors d'une simulation.

Nous nous intéresserons à quatre volets de ce chapitre qui sont les conditions d'instanciation et de terminaison d'un processus, les activités, les portes et les évènements.

FIGURE 3.17 – BPMN 2.0 : Comportement de multiples flux sortants

3.5.1 Instanciation et terminaison d'un processus

Un processus BPMN est instancié quand l'un de ses événements de départ se déclenche. Chaque occurrence d'un événement de départ crée une nouvelle instance, indépendante des autres déjà créées. D'autre part, un processus peut aussi être démarré par une *porte basée-événement* ou une *tâche recevante* qui n'aurait pas de flux entrant et ayant sa propriété "instanciate" à vrai.

Pour faciliter le comportement des éléments, le standard emploie le concept théorique de jeton ("token").

Chaque *événement de départ* qui se déclenche produit un jeton sur chacun de ses *flux de séquence* sortants. La suite d'un processus se définit selon le comportement spécifique à chaque élément traversé.

Pour être déclaré complète, une instance de processus doit remplir trois conditions :
- Si l'instance a été créée à travers une *porte parallèle*, chaque événement de cette porte doit avoir été déclenché.
- Il n'existe plus de jetons restant au sein de l'instance du processus.
- Il n'y a plus d'*activité* active.

Pour être complétée, tous les jetons d'une instance doivent rejoindre un noeud de fin, sans *flux de séquence* sortant. Dans le cas particulier d'un événement de fin, le comportement associé (comme l'envoi d'un message) se produit avant de compléter l'instance. Si un jeton atteint un *événement de terminaison*, l'entièreté du processus est anormalement terminée.

3.5.2 Les activités

Si une *activité* n'a pas de flux entrant, l'activité est instancée lorsque le *processus* ou *sous-processus* englobant est instancié. Si une activité à plusieurs flux entrants, elle est instanciée à chaque arrivée de jeton sur un de ces flux. En effet, ce cas de figure peut être transformé en un modèle équivalent comportant des portes parallèles comme illustré à la figure 3.17. De la même manière, une *activité* ayant plusieurs flux sortants, chacun d'entre eux recevra un jeton une fois l'*activité* complétée.

Dans le cas particulier où une *activité* n'aurait aucun flux sortant, la condition de terminaison du container (Processus ou Sous-processus) qui l'englobe serait appliquée. Il est important de noter qu'un jeton traverse un flux sans aucune contrainte de temps. Le standard définit précisément le comportement interne d'une *activité* par un diagramme d'état UML dans lequel nous ne rentrerons pas dans le détail. Par simplification, nous considérons qu'une *activité* prend un temps donné ou un signal pour être réalisée. En réalité, chaque type d'activité (Manuel, Service, etc) selon un comportement particulier (Boucle, instanciation multiple) a une exécution particulière bien définie dans le standard.

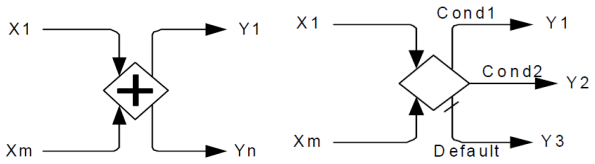

FIGURE 3.18 – BPMN 2.0 : Porte parallèle et Porte exclusive

3.5.3 Les portes

Porte parallèle Afin d'être activée, une *porte parallèle* (figure 3.18)doit avoir au moins un jeton sur chaque flux entrant. La *porte parallèle* consume exactement un jeton sur chaque flux entrant et produit un jeton sur chaque flux sortant.

Porte exclusive Chaque jeton arrivant sur un flux entrant active une *porte exclusive* (figure 3.18). Après activation, un et un seul jeton est produit sur un des flux sortants selon la condition vérifiée.

3.5.4 Les événements

Nous avons vu que les *événements de départ* servent à démarrer une nouvelle instance du processus à chaque fois qu'il se déclenche. Concernant les *événements intermédiaires* et les événements en bordure d'activités, une fois atteint, le processus attend que l'événement survienne avant de continuer. Chaque *événement de fin* consume chaque jeton arrivant sur un flux entrant. À noter que dans le cas particulier des *événements de Terminaison*, le processus est anormalement terminé sans pour autant affecter les autres instances en cours.

Chapitre 4

MetaDone

Introduction

MetaDone [Met14a] est un metaCASE développé à l'Université de Namur [Eng00, Eng09]. Implémenté en Java, l'outil évolue continuellement et est utilisé pour des activités de recherche et d'enseignement.

4.1 Base de données

4.1.1 MeTaL1

Les différents modèles créés sont représentés à l'aide d'un langage appelé MeTaL1 [Eng06], dont la structure est illustrée à la figure 4.1. Ce langage est le fruit d'un travail inspiré par un article de Andreas L. Opdahl [OS97]. La représentation utilise des rectangles pour les sommets et les ellipses pour les arrêtes. Les flèches simples représentent la source et la cible d'une arrête, que l'on appelle "domain" et "range". Les flèches en gras signifient une relation d'héritage d'un sous-type vers son super-type. Les flèches pointillées représentent une instanciation, la source étant un objet et la cible le type. Les propriétés et les objets peuvent se confondre.

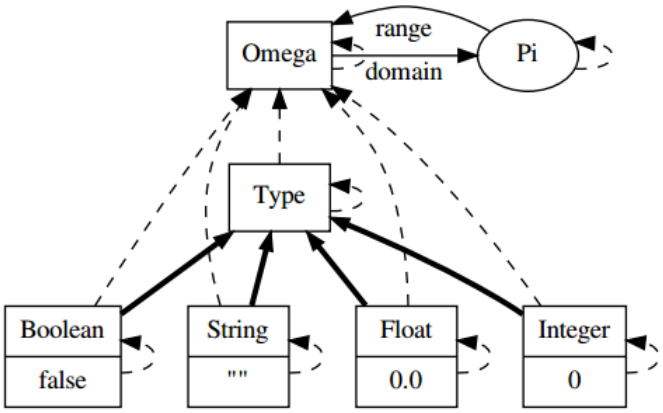

FIGURE 4.1 – MeTaL1 Core

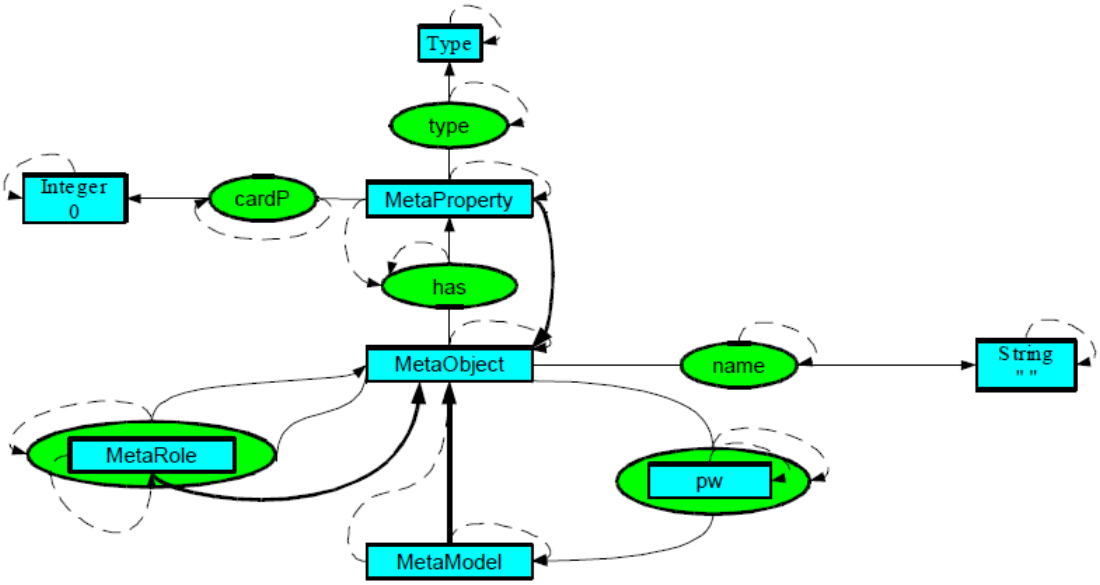

FIGURE 4.2 – MeTaL2 : Structure méta-métamodèle

4.1.2 MeTaL2

MeTaL2 est un langage de métamodélisation conçu pour décrire les langages de modélisation. Il est exprimé lui-même en MetaL1 et peut être perçu comme une couche d'abstraction de premier ordre sur MetaL1.

La représentation utilisée est toujours basée sur les mêmes principes décrits en 4.1.1. Les quatre concepts importants de ce méta-métamodèle sont `MetaObject`, `MetaRole`, `MetaProperty` et `MetaModel`. Chacun de ces concepts hérite de `MetaObject`. Les métaobjets sont identifiés par un nom. Ils font partie, via la relation d'agrégation `pw`, d'un métamodèle. En d'autres termes, un métamodèle est composé d'un ensemble de métaobjets. D'autre part, un métaobjet peut avoir des métapropriétés. Ces métapropriétés ont un type (Integer, Float, String ou Boolean) et une cardinalité exprimée par un entier. Aussi, chaque métaobjet peut être connecté à d'autres métaobjets par l'intermédiaire d'un métarole. Un métarole illustre la relation entre un métaobjet (le "domain") et un autre métaobjet (le "range"). Deux autres concepts ne sont pas illustrés sur la figure 4.2. `CommonMetaObject` représente un supertype par défaut d'un métaobjet. De la même façon, `CommonMetaModel` représente le supertype d'un métamodèle.

Ces quatre concepts peuvent être instanciés en des objets concrets : `Concrete Object`, `Concrete Role`, `Concrete Property` et `Concrete Model`. Un objet concret représente une entité quelconque d'un langage. Une propriété concrète contient une valeur ayant le même type illustré dans la métapropriété qui lui est associée. Par exemple, "Task" est un métaobjet du langage BPMN qui peut avoir plusieurs propriétés (temps d'exécution, propriétaire,...) et être lié à d'autres objets concrets (événements, activités, ...).

Tous ces concepts qui définissent MeTaL2 sont implémentés au sein de MetaDone. Un utilisateur peut y accéder grâce à l'API Java au travers de plug-ins (voir section 4.3).

Enfin, notons que puisque MetaL2 est une construction de premier ordre, il peut être modifié ou étendu par l'utilisateur selon ses besoins. Cette caractéristique est rarement ren-

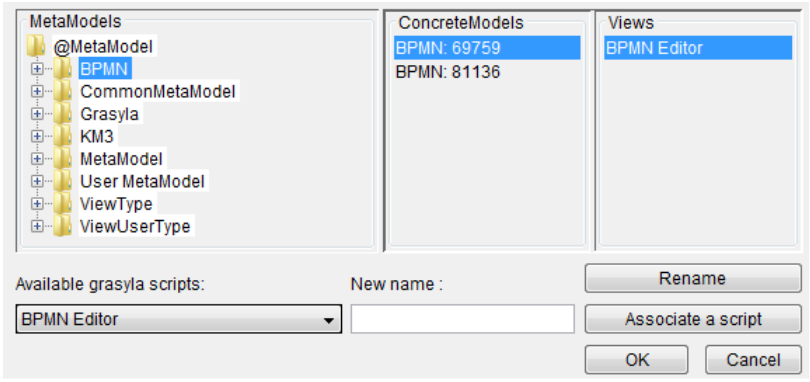

FIGURE 4.3 – MetaDone : Organisation des métamodèles-modèles-vues

contrée dans les produits concurrents. Ainsi, les concepts de métarole ordonné ou de propriétés énumérées sont définies par réification. Ce principe sera utilisé ultérieurement pour définir Grasyla (section 4.2) et d'autres extensions.

4.2 Grasyla 2

GraSyla (Graphical Symbolic Language) [EM13] est un DSML permettant de définir une syntaxe concrète pour les modèles définis en MeTaL2 (section 4.1.2). Il est défini par un métamodèle implémenté en MeTaL2 chargé au démarrage de MetaDone. La syntaxe concrète est définie par un script Grasyla qui est aussi un modèle Grasyla. Un script Grasyla est composé d'un ensemble d'équations qui associent un métaobjet à une expression qui exprime la manière dont une instance doit être représentée. Grasyla permet de décrire plusieurs types de notations différentes. De plus, les expressions ont toutes le même pattern syntaxique, car Grasyla est un langage régulier ayant une grammaire définie dans [EM13]. La syntaxe de Grasyla est facilement extensible de même que l'interpréteur qui est implémenté comme un framework générique.

Au sein de MetaDone, illustrés à la figure 4.3, plusieurs métamodèles sont disponibles. L'utilisateur peut créer plusieurs modèles concrets selon la structure définie dans un métamodèle. Pour chacun de ces modèles concrets, plusieurs vues peuvent être associées. Ces vues (le rendu graphique) sont produites par l'interpréteur Grasyla qui prend un métamodèle, un modèle concret et un script Grasyla en entrée. Il est donc possible d'afficher plusieurs représentations d'un même modèle concret en sélectionnant un script Grasyla différent pour chaque vue. Plusieurs vues peuvent aussi être créées à partir d'un seul script Grasyla. Une vue apparaît dans un éditeur où l'utilisateur peut interagir avec les objets en utilisant les fonctions de Grasyla.

Afin de permettre une manipulation avancée des métaobjets, Grasyla permet de définir des scripts en Groovy. À travers ces scripts, l'API Java de MetaDone est entièrement accessible ce qui permet de manipuler le modèle durant la manipulation de l'éditeur de manière transparente.

4.2.1 Script Grasyla

Un script Grasyla est composé de deux grandes parties : le *header* et le *body*.

Le *header* déclare une `notation` associée à un métamodèle. Une `notation` est identifiée par son nom et doit être unique pour un métamodèle. Une `description` permet d'exprimer un commentaire destiné à l'utilisateur. La clause `import` permet d'importer d'autres scripts Grasyla. Il est possible de définir des scripts qui sont des fragments de programme dans un langage de programmation. Des variables globales peuvent être déclarées par des clauses `define`. La clause `model` permet de spécifier quels métamodèles peuvent être exprimés par le script Grasyla. Les clauses `functor` et `requires` permettent de paramétrer l'interpréteur en indiquant le foncteur de départ (la première équation à interpréter) et d'autres options avancées.

Le *body* est composé d'un ensemble d'équations. Chaque équation est composée de deux parties séparées par un "=" : la partie gauche indique le concept à afficher et la partie droite indique comment afficher ce concept. Ainsi, il est possible de définir une représentation pour chaque métaobjet. Chaque équation commence par le caractère '$'. La partie gauche est composée de quatre éléments : le `foncteur` (facultatif) permet de restreindre l'utilisation de l'équation à un contexte précis, la `multiplicité` permet d'indiquer combien (no, one ou many) de concepts l'équation veut représenter, la `métaclasse` permet de restreindre l'utilisation de l'équation selon le type de de l'objet (metaobject, metaproperty ou metarole) et enfin le nom identifiant le métaobjet. La partie droite est une expression permettant d'exprimer la représentation graphique d'une instance d'un métaobjet. Ces expressions sont prédéfinies dans Grasyla avec beaucoup d'options disponibles afin de paramétrer l'affichage (voir documentation [EM13]).

Les expressions de droite d'une équation définissent la représentation d'un objet. Grasyla permet de gérer un grand nombre de représentations complexes dont la plupart sont citées ci-après.

- Des formes géométriques : triangles, flèches, rectangles, cercles, segments, carrés, ...
- Des images en format Bitmap : JPEG, PNG, ...
- Des flèches
- Des images en format SVG. Ces fichiers SVG peuvent être composés de plusieurs zones éditables (voir section 5.3.2).

Chacune de ces représentations peuvent être paramétrables à l'aide de plusieurs options comme le type de bordure (arrondies, cylindriques, ...), le type de trait (gras, pointillé, ...), la couleur, la rotation selon un nombre de degrés, etc. Ces options sont nombreuses et différentes selon la représentation utilisée. La documentation ([EM13]) recense ces représentation dans le catalogue des expressions.

De plus, Grasyla permet à l'utilisateur d'utiliser les *JComponent* de la libraire Java *Swing*. Les *JComponent* gérés par Grasyla sont cités ci-après.

- Des menus : JMenu, JMenuItem.
- Des listes : JList
- Des boutons : JButton, JCheckBox, JComboBox, JRadioButton (ainsi que les groupes de boutons)
- Des barres de progression : JProgressBar
- Des barres de défilement : JScrollBar
- Des onglets : JTabbedPane
- Des tables : JTable
- ...

Grayla gère aussi les actions telles que le clic, le double clic, le clic droit, la sélection ainsi que la validation. Grasyla offre un environnement de modélisation par l'intermédiaire

```
mo_bpmn_Artifact = mm_bpmn.produceMetaObject("Artifact", mo_bpmn_BaseElement);
mo_bpmn_TextAnnotation = mm_bpmn.produceMetaObject("TextAnnotation", mo_bpmn_Artifact);
mp_bpmn_TextAnnotation_Text = mo_bpmn_TextAnnotation.produceMetaProperty(String.class, "TextAnnotation.text", 1, mm_bpmn);
```

FIGURE 4.4 – MetaDone : Création métaobjets

```
$ node one metaobject TextAnnotation = boxH {
    "["
    value($"TextAnnotation.text")
}
```

FIGURE 4.5 – Grasyla : Exemple d'équation

d'un graphe où les formes d'un modèle peuvent être alignées géométriquement. Il faut noter que MetaDone implémente le mécanisme `Undo/Redo` qui permet à l'utilisateur de revenir en arrière dans la manipulation d'un modèle. Cette fonctionnalité est importante dans un tel environnement car l'utilisateur est souvent sujet à supprimer/modifier des éléments par erreur.

4.2.2 Exemple

Cette section illustre par un exemple la création d'une partie du métamodèle de BPMN ; une *TextAnnotation* et sa représentation par une équation Grasyla.

La figure 4.4 représente la création de plusieurs métaobjets (dont une métapropriété). La variable `mo_bpmn_Artifact` désigne le métaobjet correspondant à la superclasse *Artifact*, ayant pour superclasse *BaseElement*. De la même manière, `mo_bpmn_TextAnnotation` est le métaobjet correspondant à une *TextAnnotation* ayant pour superclasse *Artifact*. La métapropriété `mp_bpmn_TextAnnotation_text` représente une propriété "text" du métaobjet `mo_bpmn_TextAnnotation`. Elle est de type String, identifiée par le nom "TextAnnotation.text" et a une cardinalité de 1.

Ces métaobjets ont une équation correspondante au sein du script Grasyla décrivant le métamodèle BPMN. La figure 4.5 illustre l'équation décrivant le métaobjet *TextAnnotation*. La partie gauche de l'équation désigne littéralement "un métaobjet TextAnnotation". Le foncteur `node` sert à placer l'équation dans l'organisation globale du script Grasyla. La partie droite décrit une expression. Dans ce cas, c'est une boîte horizontale composée de deux éléments. Le premier est le caractère '[' qui est spécifique à une annotation BPMN. Le second est la valeur de la propriété "TextAnnotation.text", soit le texte du commentaire à afficher après '['.

4.3 Architecture

Tous les besoins n'étant pas anticipés dans un metaCASE générique comme MetaDone, une architecture extensible sous forme de plugins est mise en place. MetaDone est implémenté selon le framework OSGi sous la forme de plusieurs *bundles*. Sans rentrer dans les détails, OSGi permet de décomposer une architecture en une série de *bundles* qui peuvent être chargés "à chaud" lors du fonctionnement d'un logiciel. Le framework [ME11] fournit deux mécanismes afin d'implémenter un plugin. La première est le mécanisme de chargement de classes Java. Ce mécanisme n'est disponible que sur la version *standalone* de MetaDone. La seconde, plus

appropriée, est d'utiliser l'architecture OSGi. La procédure pour implémenter un tel plugin est décrite dans [ME11]. Par exemple, l'implémentation de BPMN au sein de MetaDone se fera par la création d'un plugin.

Chapitre 5

Implémentation

Ce chapitre explique l'implémentation de BPMN au sein de MetaDone. L'intégration du plug-in BPMN fera l'objet de la première section en tant que support de base du métamodèle et du fichier Grasyla, qui seront expliqués dans les deux sections suivantes. Ensuite, une section donnera un aperçu du code par une série de métriques. Enfin, deux modèles donnant un aperçu de l'éditeur termineront le chapitre.

5.1 Architecture du plug-in

Comme expliqué dans la section 4.3, MetaDone est construit sur une architecture basée sur le framework OSGi qui permet d'ajouter des plugins afin d'étendre ses fonctionnalités. Un manuel d'explication ainsi qu'une procédure de création d'un plugin de base pour MetaDone est présentée dans le document [ME11].

Concrètement, un plugin est un projet Java. Les différents plugins ainsi que le cœur de MetaDone sont illustrés à la figure 5.1. Les projets préfixés par `metadone_` forment le cœur de MetaDone. Les plugins sont préfixés par `plugin_`. Dans notre cas, le plugin se nomme `plugin_bpmn`.

Tout plugin doit contenir les éléments suivants : un *Activator*, un *Manifest* et un fichier *build.properties*. Ces trois éléments sont utilisés par le framework OSGi. Après la modification du fichier de configuration de MetaDone, le plug-in BPMN sera chargé au démarrage.

L'implémentation du métamodèle de BPMN, décrit dans la section 5.2, constitue le fichier "BPMN.java". Cette classe est invoquée dès que le plugin est chargé par MetaDone, créant les différents metaobjets constituant le metamodèle. Les fichiers Grasyla, constituant la syntaxe concrète de BPMN, sont situés dans le dossier "grasyla". Sa structure est décrite à la section 5.3. Enfin, les ressources telles que les fichiers SVG et la documentation sont rassemblées dans le dossier "resources".

5.2 Métamodèle

L'implémentation d'un métamodèle se réalise de manière programmatique à l'aide de l'API de MetaDone, basée sur MeTaL2. Cette API, sous forme d'interfaces, permet de créer les métaobjets (métaobjets, métaroles, métamodèles, métapropriétés) correspondants à un métamodèle comme celui de BPMN. Les quatre fonctions qui nous intéressent sont reprises ci-dessous.

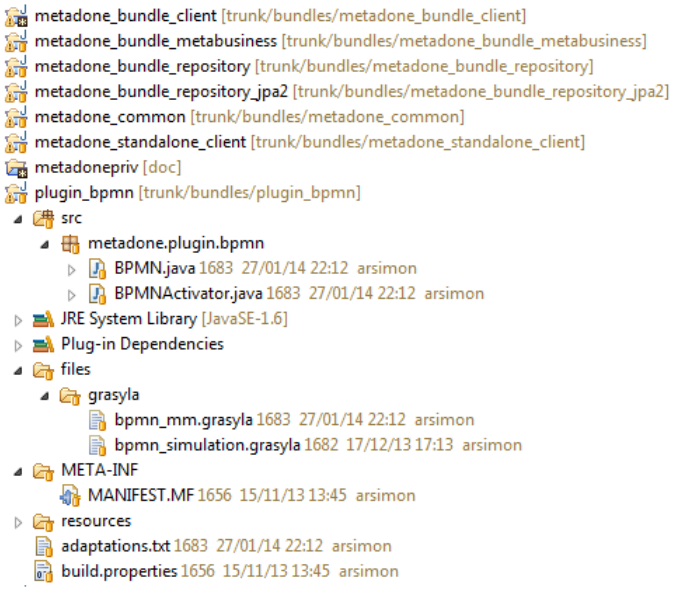

FIGURE 5.1 – MetaDone : Décomposition en projets Java

La procédure suivie est d'encoder chaque métaobjet décrit dans le standard BPMN [OMG11a] ainsi que ses métaroles et métapropriétés. L'implémentation du métamodèle simplifié reprise à la figure 3.13 est illustrée dans le code suivant. Cette portion de code ne reprend que 32 métaobjets, tandis que l'entièreté du métamodèle implémenté en comprend environ 250. Comparativement à la grandeur du métamodèle de BPMN, la quantité main d'oeuvre nécessaire pour écrire les 300 lignes de code (hormis déclarations et autres fonctions) n'est pas importante étant donné le caractère répétitif des fonctions à invoquer.

Il est important de noter que l'entièreté du métamodèle a été implémentée manuellement, ce qui est assez répétitif étant donné le nombre important de métaobjets composant le métamodèle de BPMN. Cependant, avec l'expérience de l'implémentation de BPMN, on doit pouvoir réaliser une application pouvant générer automatiquement le code correspondant à un métamodèle. Une telle application permettrait un gain de temps considérable. Par exemple, le métamodèle BPMN est disponible dans une version ECore [1] (basé sur XML) comme beaucoup d'autres métamodèles. Un générateur automatique de métamodèle à partir d'un fichier ECore serait un plus pour MetaDone.

```
1  /*
2   * Interfaces
3  */
4
5  public interface MetaModel extends MetaObject {
6      ...
7      public MetaObject produceMetaObject(String name, MetaObject... supertypes)
           throws BadPreCondition;
8      ...
9      public MetaRole produceMetaRole(String name, MetaRole.Cardinality
           cardinality, MetaObject domain, MetaObject range) throws BadPreCondition;
10     ...
```

1. ECore : métamétamodèle de Eclipse Modelling Framework (EMF)

```
11      public MetaModel produceMetaModel(String name, MetaObject... supertypes)
            throws BadPreCondition;
12      ...
13  }
14
15  public interface MetaObject extends WorkspaceObject, Comparable<MetaObject> {
16      ...
17      public <T> MetaPropertyExt<T> produceMetaProperty(Class<T> type, String
            name, int cardinality, MetaModel metamodel) throws BadPreCondition;
18      ...
19  }
```

```
1  /*
2   * Simple Metamodel
3   */
4
5  final MetaModel main = ws.getMainMetaModel();
6
7  mm_bpmn = main.produceMetaModel("BPMN");
8
9  mo_bpmn_BaseElement = mm_bpmn.produceMetaObject("BaseElement");
10
11 mo_bpmn_FlowElementsContainer =
       mm_bpmn.produceMetaObject("FlowElementsContainer", mo_bpmn_BaseElement);
12
13 mo_bpmn_FlowElement = mm_bpmn.produceMetaObject("FlowElement",
       mo_bpmn_BaseElement);
14 mp_bpmn_FlowElement_Name =
       mo_bpmn_FlowElement.produceMetaProperty(String.class, "FlowElement.name", 1,
       mm_bpmn);
15
16 mo_bpmn_FlowNode = mm_bpmn.produceMetaObject("FlowNode", mo_bpmn_FlowElement);
17
18 mo_bpmn_Activity = mm_bpmn.produceMetaObject("Activity", mo_bpmn_FlowNode);
19
20 mo_bpmn_Task = mm_bpmn.produceMetaObject("Task", mo_bpmn_Activity,
       mo_bpmn_InteractionNode);
21 mo_bpmn_SubProcess = mm_bpmn.produceMetaObject("SubProcess", mo_bpmn_Activity,
       mo_bpmn_FlowElementsContainer);
22
23 mo_bpmn_DataStore = mm_bpmn.produceMetaObject("DataStore",
       mo_bpmn_ItemAwareElement, mo_bpmn_RootElement);
24 mp_bpmn_DataStore_name = mo_bpmn_DataStore.produceMetaProperty(String.class,
       "DataStore.name", 1, mm_bpmn);
25
26 mo_bpmn_Event = mm_bpmn.produceMetaObject("Event", mo_bpmn_FlowNode,
       mo_bpmn_InteractionNode);
27 mo_bpmn_CatchEvent = mm_bpmn.produceMetaObject("CatchEvent", mo_bpmn_Event);
28 mo_bpmn_BoundaryEvent = mm_bpmn.produceMetaObject("BoundaryEvent",
       mo_bpmn_CatchEvent);
29 mo_bpmn_IntermediateCatchingEvent =
       mm_bpmn.produceMetaObject("IntermediateCatchingEvent", mo_bpmn_CatchEvent);
30 mo_bpmn_StartEvent = mm_bpmn.produceMetaObject("StartEvent",
       mo_bpmn_CatchEvent);
31 mo_bpmn_ThrowEvent = mm_bpmn.produceMetaObject("ThrowEvent", mo_bpmn_Event);
32 mo_bpmn_IntermediateThrowEvent =
       mm_bpmn.produceMetaObject("IntermediateThrowEvent", mo_bpmn_ThrowEvent);
```

```
33 mo_bpmn_EndEvent = mm_bpmn.produceMetaObject("EndEvent", mo_bpmn_ThrowEvent);
34
35 mr_bpmn_SequenceFlow = mm_bpmn.produceMetaRole("SequenceFlow",
       MetaRole.Cardinality.MANY_TO_MANY, mo_bpmn_FlowNode, mo_bpmn_FlowNode);
36
37 mo_bpmn_Gateway = mm_bpmn.produceMetaObject("Gateway", mo_bpmn_FlowNode);
38 mo_bpmn_ParallelGateway = mm_bpmn.produceMetaObject("ParallelGateway",
       mo_bpmn_Gateway);
39 mo_bpmn_ExclusiveGateway = mm_bpmn.produceMetaObject("ExclusiveGateway",
       mo_bpmn_Gateway);
40 mo_bpmn_InclusiveGateway = mm_bpmn.produceMetaObject("InclusiveGateway",
       mo_bpmn_Gateway);
41 mo_bpmn_ComplexGateway = mm_bpmn.produceMetaObject("ComplexGateway",
       mo_bpmn_Gateway);
42
43 mo_bpmn_Artifact = mm_bpmn.produceMetaObject("Artifact", mo_bpmn_BaseElement);
44 mo_bpmn_Group = mm_bpmn.produceMetaObject("Group", mo_bpmn_Artifact);
45 mo_bpmn_TextAnnotation = mm_bpmn.produceMetaObject("TextAnnotation",
       mo_bpmn_Artifact);
46
47 mo_bpmn_Lane = mm_bpmn.produceMetaObject("Lane", mo_bpmn_BaseElement);
48 mp_bpmn_Lane_Name = mo_bpmn_Lane.produceMetaProperty(String.class, "Lane.name",
       1, mm_bpmn);
49 mo_bpmn_LaneSet = mm_bpmn.produceMetaObject("LaneSet", mo_bpmn_BaseElement,
       mo_bpmn_InteractionNode);
50 mp_bpmn_LaneSet_Name = mo_bpmn_LaneSet.produceMetaProperty(String.class,
       "LaneSet.name", 1, mm_bpmn);
```

5.3 Grasyla

5.3.1 Construction du fichier

Le processus de construction du fichier Grasyla est l'aboutissement de plusieurs adaptations, corrections qui ont permis l'affichage correct des éléments BPMN sur un même graphe. En effet, BPMN présente plusieurs difficultés dans sa représentation. Par exemple, une difficulté majeure a été la représentation des sous-processus. Ces sous-processus peuvent être dépliés, ce qui implique que les éléments internes au sous-processus doivent être présentés d'une manière différente des éléments formant le processus de base. Cette partie sera approfondie dans la section 5.3.2.

Concrètement, le fichier Grasyla est composé d'équations qui s'appellent mutuellement selon le modèle construit par l'utilisateur. L'ensemble des foncteurs accompagnant chaque équation forme une arborescence, représentée à la figure 5.2, où la racine définit la présentation de l'éditeur (graphe, menus, etc) et les extrémités représentent les éléments de base (node) tels que les *Task*, les *Gateway* ou encore les *Event*.

Cet ensemble d'équations permet d'afficher l'ensemble des éléments du modèle créé avec l'éditeur BPMN. L'ensemble des éléments BPMN a été séparé en 4 sous-ensembles selon la manière dont ils doivent s'afficher sur le graphe. Par exemple, les *FlowElements* sont toujours liés à un conteneur (Processus, Sous-processus, Lane) tandis que les *Artifacts* sont indépendants dans leur placement dans le modèle.

FIGURE 5.2 – Fichier Grasyla : arborescence

5.3.2 Points importants

Quelques points importants concernant l'implémentation de la syntaxe concrète de BPMN à l'aide de Grasyla sont repris aux sous-sections suivantes.

Création objets concrets

Afin de permettre l'affichage correct des éléments, il est nécessaire de créer chaque élément concret et de le lier au modèle existant. C'est pourquoi, à chaque création d'objet via le menu, MetaDone va intégrer ce nouvel objet au modèle par l'intermédiaire d'une fonction en Groovy. Grasyla permet d'insérer de tels scripts et offre une interface par laquelle un script Groovy peut interagir avec tous les éléments du contexte comme le modèle, les variables, le métamodèle ou encore le workspace de MetaDone.

La portion de code suivant illustre l'appel à une fonction permettant de créer un nouvel objet et de le lier au modèle. Le processus, correspondant au modèle courant, est stocké dans une variable *processus*. Le second argument précise l'objet à créer, en l'occurrence une *Task*. L'appel à la fonction de l'interface de MetaDone `createRole(metarole, domain, range)` permet de créer le rôle afin de lier l'objet créé au processus.

```
1  Util.createForProcess(context.getVariable("processus"),"Task") //appel
2
3  class Util {
4      static def createForProcess(domain, range) {
5          Context.concretemodel.createRole(
6              Context.metamodel.getMOByName("FlowElementContainer_FlowElement")
7                  .narrow2MetaRole(),//role
8              domain,//domaine (processus)
9              Context.concretemodel.createObject(
10                 Context.metamodel.getMOByName(range))//range (Task)
11         )
```

```
12      }
13  }
```

Les sous-processus

Un sous-processus a deux représentations possibles dans un modèle BPMN. Soit il peut être plié (voir figure 5.3) ou il peut être déplié (voir figure 5.4). La première cache les éléments internes au sous-processus et affiche uniquement son nom. La seconde affiche les éléments internes sous forme d'un flux encadré. Dans la conception de l'éditeur, cet aspect a compliqué l'affichage des éléments selon leur emplacement.

Le code simplifié suivant exprime la représentation des sous-processus. Une variable *isExpanded* associée à un *FlowElementContainer* (supertype d'un sous-processus) exprime si le conteneur est déplié ou plié. Selon la valeur de cette variable, l'équation adéquate est choisie. La première équation exprime le point de choix par une condition sur la variable *isExpanded* et redirige soit vers l'équation 2 qui exprime un sous-processus déplié, soit vers l'équation 3 qui représente un sous-processus plié. Les éléments au sein du sous-processus déplié sont insérés par la commande *free* qui permet de déplacer les éléments à l'aide de la souris sans les sortir du sous-processus. Le rectangle extérieur s'agrandit si l'utilisateur déplace un élément hors du sous-processus. De plus, l'éditeur permet de changer la représentation du sous-processus par le menu contextuel (clic droit).

```
1  $ node one metaobject SubProcess = if {
2      //switch on the appropriate equation
3      guard { condition : [ bool($"FlowElementsContainer.isExpanded")]
4              expanded($*self)  }
5      guard { condition : [ not {
           bool($"FlowElementsContainer.isExpanded") } ]
6              collapsed($*self) }
7  }
8
9  $ expanded one metaobject SubProcess = boxV {
10
11     value($"FlowElement.name")
12
13     border : "roundedrectangle" {
14         harc : 20
15         varc : 20
16         stroke : "solid"
17     }
18
19     free {
20         //the inside elements
21         node($FlowElementContainer_FlowElement.range)
22     }
23 }
24
25 $ collapsed one metaobject SubProcess = box {
26
27     border : "roundedrectangle" {
28         harc : 20
29         varc : 20
```

FIGURE 5.3 – BPMN : Sous-processus plié

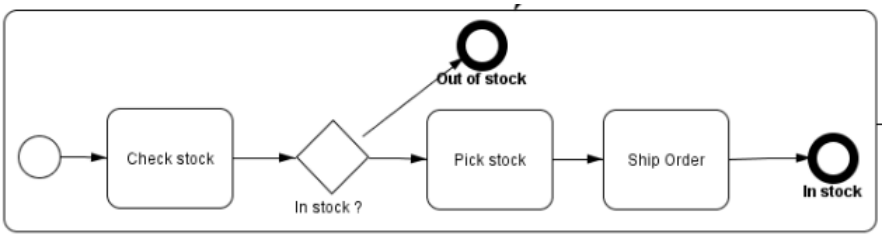

FIGURE 5.4 – BPMN : Sous-processus déplié

```
30          stroke : "solid"
31      }
32
33      boxV{&SubProcess}// + framed
34  }
```

Les lanes et pools

Les lanes et pool sont les représentations des participants à un processus BPMN. Dans la figure 5.5, le pool *Budget Process* contient deux lanes *Employee* et *Manager*. Afin d'optimiser les diagrammes, l'éditeur devait permettre le redimensionnement des pool et lanes. Cependant, l'outil d'édition de graphes de MetaDone ne permet pas le redimensionnement par déplacement d'un clic sur une ligne d'un pool. Une alternative a été choisie afin de contourner ce problème : les boutons. Disposés sur les côtés, les boutons "+" et "-" permettent d'augmenter ou diminuer la largeur ou la longueur d'un pool. À cette fin, deux variables ont été rajoutées à un *Pool* : *height* et *width*. Ces variables contiennent les grandeurs relatives à chaque côté d'un *Pool* et sont modifiées par un clic sur un des boutons. Ainsi, dans le cas d'un pool contenant un autre pool, la fonctionnalité est toujours adaptée.

De plus, l'ajout d'une *Lane* à un *Pool* est possible par le menu contextuel (clic droit) sur le *Pool* concerné.

Images SVG

Les éléments BPMN qui ne sont pas représentables avec les formes gérées par Grasyla sont affichés grâce à une image SVG. SVG [2] est un format de données permettant de décrire une image. L'utilisation de ce format vectoriel permet beaucoup plus de flexibilité par rapport aux formats tels que JPEG, PNG ou GIF qui utilisent une représentation bitmap. En effet, des zones spécifiques peuvent être ajoutées à une image SVG. D'autres images SVG peuvent ainsi être insérées dans ces zones afin de construire une image composée.

L'exemple présenté à la figure 5.6 illustre l'insertion des trois barres verticales (2), symbolisant une collection, dans la zone indiquée (1). Il résulte une nouvelle image (3) composée

2. SVG : Scalable Vector Graphics

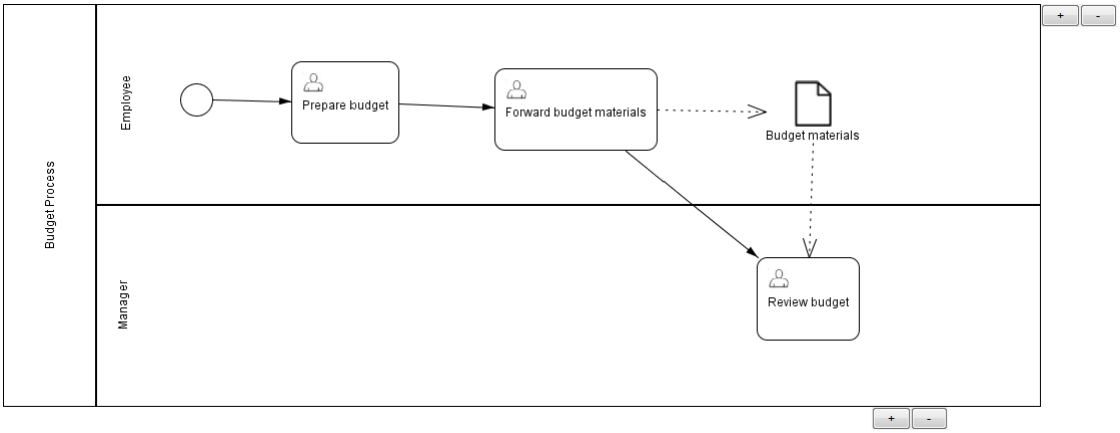

FIGURE 5.5 – BPMN : Pool et Lanes

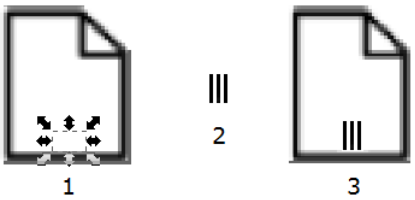

FIGURE 5.6 – SVG : Insertion par zones

des deux premières. Ce mécanisme est utilisé dans de nombreux cas comme les éléments intérieurs aux activités (Boucle, Sous-processus, Collection, Ad-Hoc, ...) ou encore dans l'insertion des symboles relatifs aux différents types d'événements (Cancel, Error, Message, ...) dans les cercles.

Pratiquement, Grasyla permet cette insertion d'une manière simple. Chaque zone du fichier SVG étant identifiée par un numéro, les fichiers renseignés entre "[ ]" seront insérés dans les zones suivant l'ordre dans lequel ils sont indiqués. Le code suivant illustre le métaobjet *DataObject* comme étant une image SVG *Data_Object.svg* où la zone 1 à été remplacée par l'image *Data_Collection.svg*. Dans le fichier Grasyla reprenant l'entièreté de la syntaxe, les chemins ont été indiqués sous forme de variables (exemple : *&Data_Object*) afin de permettre une modification à un seul endroit.

```
1 $ node one metaobject DataObject = svg {
2     file : "Data_Object.svg" [ "Data_Collection.svg" ]
3 }
```

Les propriétés

Après l'étude des différents éditeurs BPMN connus (section 3.4), il ressort que beaucoup d'éditeurs ne proposent pas une capacité d'édition des propriétés relatives à chaque élément. Le métamodèle de BPMN défini de nombreuses propriétés pour chaque élément. L'éditeur Signavio respecte rigoureusement les propriétés du métamodèle et permettait d'assigner des valeurs à chacune. Le métamodèle de BPMN 2.0 étant bien intégré dans le plug-in, ainsi donc que les propriétés, il était possible de permettre à l'utilisateur de leur assigner des valeurs au

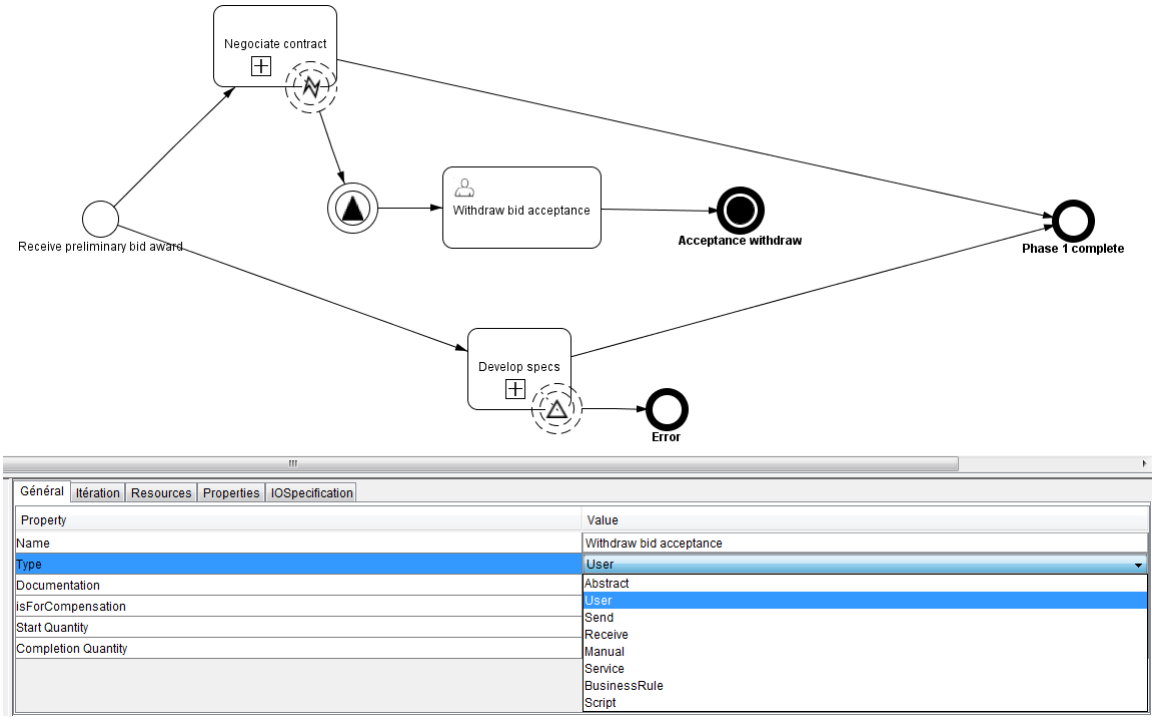

FIGURE 5.7 – Editeur BPMN : Propriétés

travers de l'éditeur.

Une partie de l'écran, en bas à droite de l'éditeur, permet d'afficher les propriétés relatives à l'élément sélectionné. Suite à un clic de l'utilisateur, les propriétés de l'élément s'affichent dans plusieurs onglets selon leur catégorie. Chaque propriété est accompagnée d'un champ d'édition, une *checkbox*, une *selectbox* ou encore d'une table permettant d'insérer plusieurs valeurs en cas de propriété multiple. Les tables sont des tableaux où chaque ligne représente une entrée de la propriété. De plus, chacune de ces lignes est associée à un bouton *Delete* permettant de la supprimer.

La figure 5.7 illustre cet aspect de l'éditeur. Les propriétés de la tâche *Withdraw bid acceptance* sont affichées selon plusieurs onglets. L'onglet *Général* affiche certaines propriétés comme le nom ou le type d'activité. Par sélection, l'utilisateur peut facilement changer le type de l'activité selon la situation à modéliser. Il est important de noter que la partie basse, comprenant les différents onglets ainsi que les différents champs d'éditions relatifs aux propriétés, est définie dans le fichier Grasyla par l'utilisateur. Selon les besoins, cette partie peut être facilement modifiée. Par exemple, selon le degré d'expertise et les besoins du modélisateur, plusieurs propriétés peuvent être cachées ou regroupées.

Dans un travail futur, un outil de simulation pourrait être associé à l'éditeur. La possibilité d'édition des propriétés est un prérequis indispensable à la simulation d'un modèle BPMN.

5.4 Métriques

Afin de donner une appréciation de la taille du plug-in BPMN, cette section reprend quelques métriques concernant le code Java et le fichier Grasyla.

Concernant l'intégration de la représentation des éléments BPMN, l'éditeur est capable de représenter 127 éléments sur les 136 possibles. La partie Process de BPMN est couverte à l'exception de 4 éléments : les *CallActivity* dépliées, les *Transaction* dépliées, les *Pool* verticaux et les *Message Flow Decorator*. Les deux premiers étant basés sur le même principe que le sous-processus, leur intégration ne nécessiterait pas un effort excessif. Les autres éléments composent les parties *Collaboration* et *Choreography* : les *Conversation Link*, les types de tâches de choreography, la *Conversation*, la *Subconversation* et la *Call Conversation*.

Le métamodèle, décrit grâce à l'assignation des variables en Java, recense 329 métaobjets au sens large. Parmi ces métaobjets, le métamodèle compte 85 métaobjets stricts, 96 métarôles (relations d'héritage non comprises) ainsi que 148 métapropriétés. Ces métaproriétés sont principalement de type String (96), Boolean (31) et Long (9). Pour rappel, certaines propriétés ont été ajoutées comme la longueur et hauteur des *Pool*.

Le fichier Grasyla est composé de 3723 lignes de code. Cependant, la représentation des propriétés des différents éléments reprend environ 2200 lignes. Donc, environ 1500 lignes sont utilisées pour représenter les éléments BPMN. Le métamodèle compte autant de lignes que de métaobjets à instancier, c'est-à-dire 329. Les équations Grasyla sont au nombre de 106. Si on supprime les équations décrivant les propriétés, il reste 50 équations soit environ une équation pour deux métaobjets. On peut remarquer que l'équation du métaobjet *Task* permet d'afficher les 8 types de tâches différents. Les événements sont aussi rassemblés dans 6 équations capables de représenter les 58 différents types d'événements en changeant le symbole intérieur. D'autre part, certains métaobjets nécessitent plusieurs équations, comme le *Sous-Processus* (3 équations). D'autres équations (11) n'ont pas de représentation graphique mais servent à définir l'arborescence du fichier.L'équation la plus profonde est celle représentant un *Pool*, elle compte 142 lignes.

Concernant les variables, le fichier Grasyla compte 46 `define` servant à définir les chemins absolus vers les fichiers SVG. De plus, 9 variables sont utilisées notamment pour contenir la référence de l'objet sélectionné par l'utilisateur ou encore le nombre de *Lane* contenues dans un *Pool* afin d'adapter la taille correctement.

Les scripts en Groovy tiennent une place importante dans le fichier Grasyla. En effet, 122 scripts composent le fichier Grasyla. En omettant la partie relative aux propriétés, le fichier Grasyla contiendrait 55 scripts. Ils sont principalement utilisés lors de la création des éléments afin de les lier au métamodèle existant. Chaque script est en moyenne composé de 4 lignes.

5.5 Modélisation de l'exemple de référence

Après l'implémentation de l'éditeur, l'exemple de référence introduit à la section 1.3 peut à présent être entièrement modélisé avec MetaDone. Ce modèle est illustré à la figure 5.8.

À titre de comparaison avec la modélisation sous l'environnement Signavio (voir figure 1.1), l'ensemble des éléments a été correctement modélisé sous MetaDone. Signavio étant un logiciel très avancé, nous pouvons remarquer que la modélisation sous MetaDone est presque semblable. Ce qui, en prenant en compte le temps investi dans l'implémentation de l'éditeur sous MetaDone, est un résultat très positif.

5.6 Illustrations

Afin d'illustrer les capacités de l'éditeur, deux modèles ont été choisis.

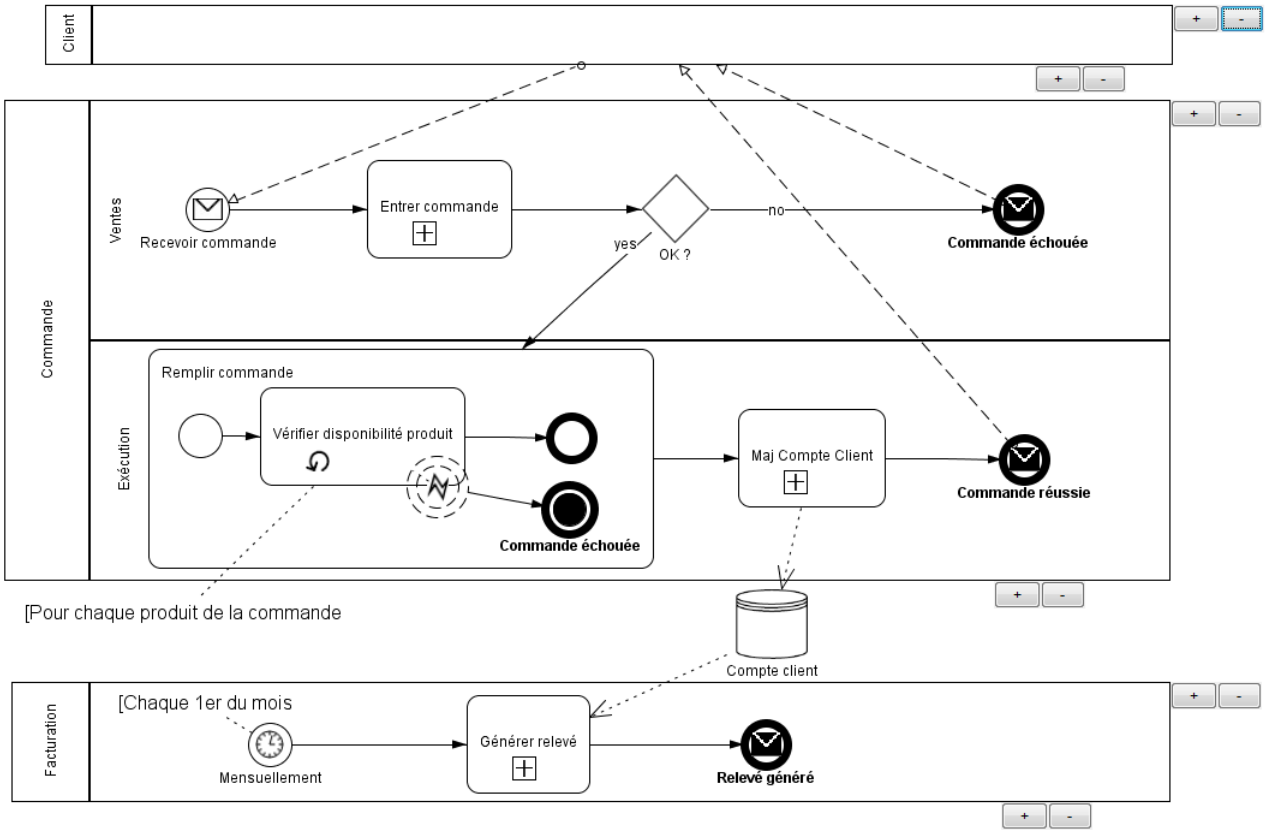

FIGURE 5.8 – Exemple contextuel modélisé par MetaDone

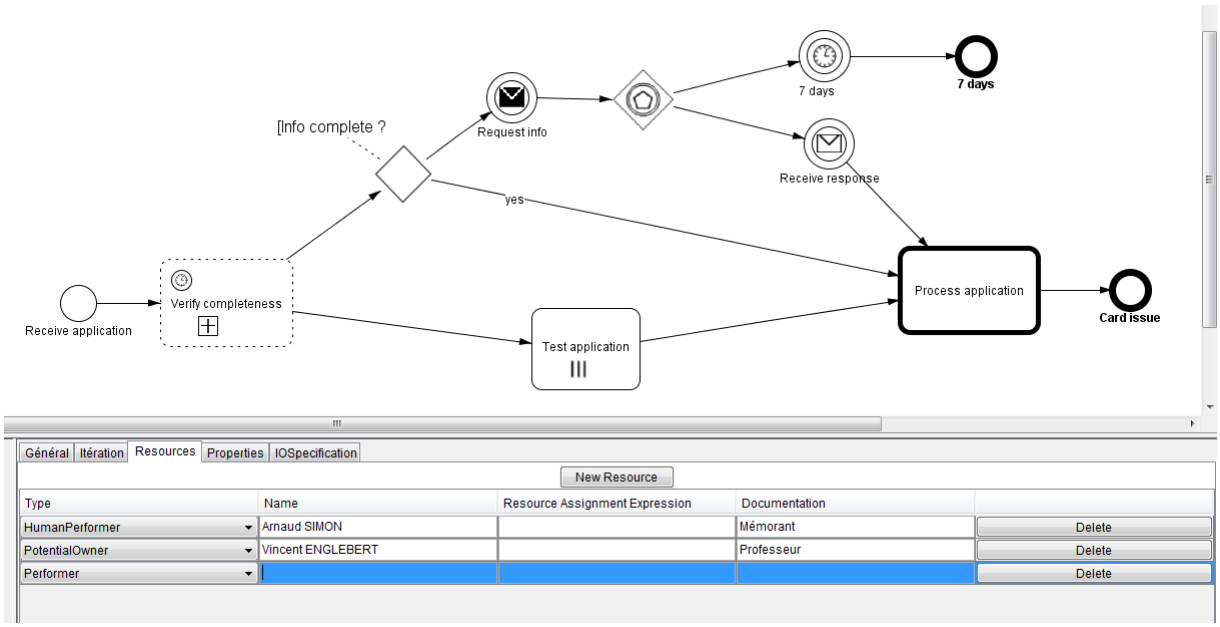

FIGURE 5.9 – Editeur BPMN : Exemple 1

Le premier modèle, représenté à la figure 5.9, illustre le processus d'acceptation d'une application logicielle. Ce modèle montre plusieurs éléments BPMN gérés par l'éditeur comme les différents types d'événements ou les types d'activités. Une table est utilisée afin d'ajouter des ressources à la tâche *Test application*. Un bouton permet de créer une nouvelle entrée sous forme de champs éditables par l'utilisateur. De plus, chaque entrée peut être supprimée par un bouton Delete.

Le second modèle, représenté à la figure 5.10, illustre le processus fictif d'enregistrement d'un nouvel étudiant à l'Université de Namur. Il présente 2 *Pool* : Etudiant et UNamur. Le service des inscriptions attend une inscription de la part d'un étudiant. Le sous-processus1 exprime l'enregistrement interne de l'étudiant. Ensuite la création des logins (EID) et des logins propres aux facultés s'exécute parallèlement. Le processus se termine en envoyant un message de confirmation à l'étudiant.

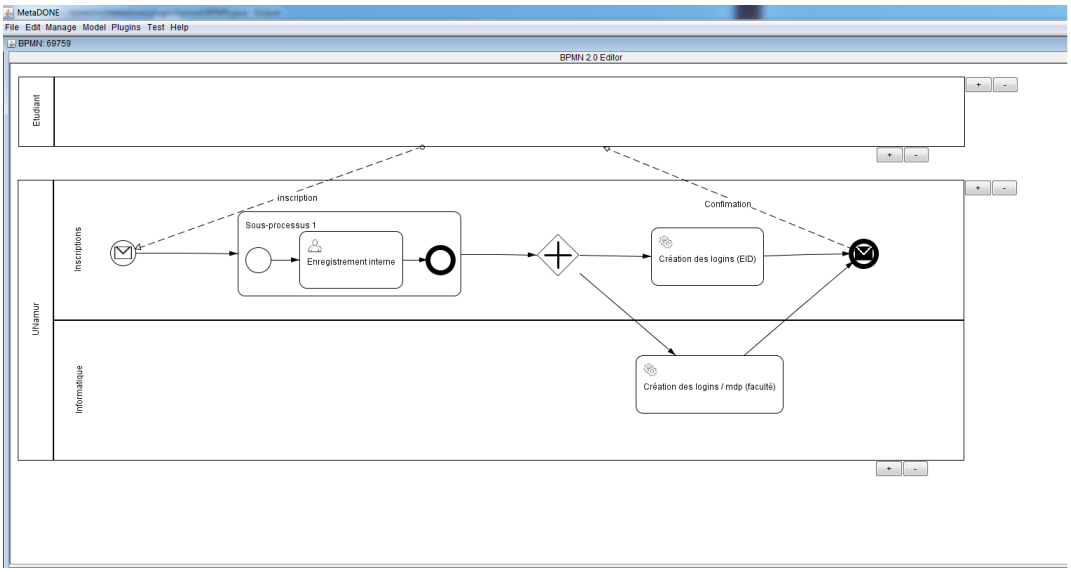

FIGURE 5.10 – Editeur BPMN : Exemple 2

Chapitre 6

Approche prospective

6.1 Discussions sur l'éditeur

Nous avons vu que l'implémentation de l'éditeur BPMN au sein d'un outil de métamodélisation (MetaDone), en comparaison avec la richesse des éléments, présente plusieurs avantages et inconvénients par rapport à l'implémentation dans un outil classique comme ceux décrits à la section 3.4.

Du point de vue de la complétude fonctionnelle, l'éditeur BPMN comprend toutes les fonctions de base d'un éditeur. D'abord, l'outil de modélisation reprend presque tous les éléments de BPMN (Partie *PROCESS*) ainsi que leurs nombreuses propriétés. Les modèles sont sauvegardés dans une base de données H2. Afin de permettre l'interopérabilité avec un autre éditeur, le plugin BPMN implémente un embryon d'exportateur de modèles. En effet, la plupart des outils classiques (voir section 3.4) permettent l'exportation des modèles en plusieurs formats comme BPMN XML, XPDL ou Visio WorkFlow Interchange. L'embryon d'exportateur prend un modèle BPMN en entrée et produit un fichier BPMN XML en parcourant l'arborescence du modèle. Pour l'instant, seulement la structure de base et quelques éléments sont implémentés. Afin de poursuivre, chaque élément BPMN doit être ajouté conformément au standard BPMN qui définit le schéma XML correspondant à chaque élément BPMN.

Du point de vue de l'adaptation, MetaDone et Grasyla permettent de modifier rapidement le métamodèle ainsi que la syntaxe concrète afin d'incorporer des changements. N. Genon, dans son article [GHA11], propose une modification de certaines icônes comme l'illustre la figure 6.1. Les icônes BPMN en (A) sont, selon la Physique des Notations, sémantiquement opaque. C'est-à-dire que leur représentation graphique n'évoque pas clairement leur sémantique. N.Genon propose de remplacer chaque icône en (A) par une icône (B). "En conséquence, et même si elles peuvent apparaître sémantiquement perverses au premier abord pour plusieurs personnes, ces icônes deviendraient plus évocatrices pour les novices"[GHA11]. Après leur intégration au sein de MetaDone, ces changements impliquent simplement le changement de cinq fichiers SVG dans le fichier Grasyla. Ce qui correspond à un temps de travail de quelques minutes. De même, MetaDone permet de modifier rapidement le langage BPMN en ajoutant par exemple une propriété à un élément, remplacer un élément par un autre ou encore ajouter un nouvel élément personnalisé.

De plus, l'implémentation d'un éditeur BPMN implique que MetaDone et Grasyla puissent modéliser un langage complexe. De la même façon que l'éditeur URN a inspiré la création de l'éditeur BPMN, l'éditeur BPMN pourrait inspirer l'intégration d'autres langages au sein

FIGURE 6.1 – (A) Icônes sémantiquement opaques et (B) Icônes plus sémantiquement transparentes (Source : [GHA11])

de MetaDone. D'autres langages, comme plusieurs types de diagrammes UML, présentent les mêmes types de constructions que BPMN : des formes géométriques reliées par des flèches, l'inclusion de formes dans d'autres formes (sous-processus), ... ce qui indique que leur intégration serait possible sans nécessiter une main d'œuvre trop importante.

6.2 Embryon de simulateur (animation de BPMN)

6.2.1 Contexte

La plupart des éditeurs BPMN présentés dans la section 3.4 proposent un outil de simulation de modèle BPMN. La simulation d'un modèle se fait selon un scénario qui est composé d'un ensemble de paramètres comme le coût, la durée ou encore les ressources disponibles. Visuellement, la simulation (Signavio) s'exécute par une animation du modèle soit entièrement soit pas à pas suivant les interactions de l'utilisateur. Le modèle s'anime grâce à la mise en valeur de l'état d'avancement du modèle. Signavio exprime l'avancement du processus par les éléments du modèle qui se colorent à la manière d'une barre de progression interactive. En suivant cette technique, un embryon de simulateur, utilisant les couleurs pour animer un modèle BPMN, a été réalisé au sein de l'éditeur BPMN de MetaDone. Son fonctionnement est décrit à la section suivante.

6.2.2 Principe

L'animation proposée est basée sur la sémantique de BPMN décrite à la section 3.5. Cette animation ne se veut pas être un simulateur de modèle, mais un exemple permettant à la fois de montrer l'animation d'un petit modèle BPMN et de montrer d'autres facettes du langage Grasyla.

Un exemple d'animation est illustré aux figures 6.2 et 6.3. L'animation gère les éléments de base de BPMN comme les *StartEvent*, *Task*, *Parallel Gateway*, *Exclusive Gateway* ainsi que les *EndEvent*. Le modèle d'exemple reprend ces différents éléments disposés de manière à illustrer l'exécution d'une instance du processus. Comme le suggère la sémantique de BPMN, la progression est identifiée par un jeton traversant les différents éléments en suivant le sens du flux. Chaque *StartEvent* peut recevoir des jetons par une option du menu contextuel. Ensuite, chaque tâche potentiellement exécutable s'affiche de couleur verte, celles non exécutables (sans jeton) sont affichées en rouge. L'utilisateur peut exécuter les tâches vertes par une option du menu contextuel. Aux différents points de choix (*Exclusive Gateway*), l'utilisateur peut choisir le flux par lequel le jeton doit transiter en utilisant le menu contextuel de la porte. Finalement, le jeton aboutit à un *EndEvent*, ce qui termine l'exécution de l'instance.

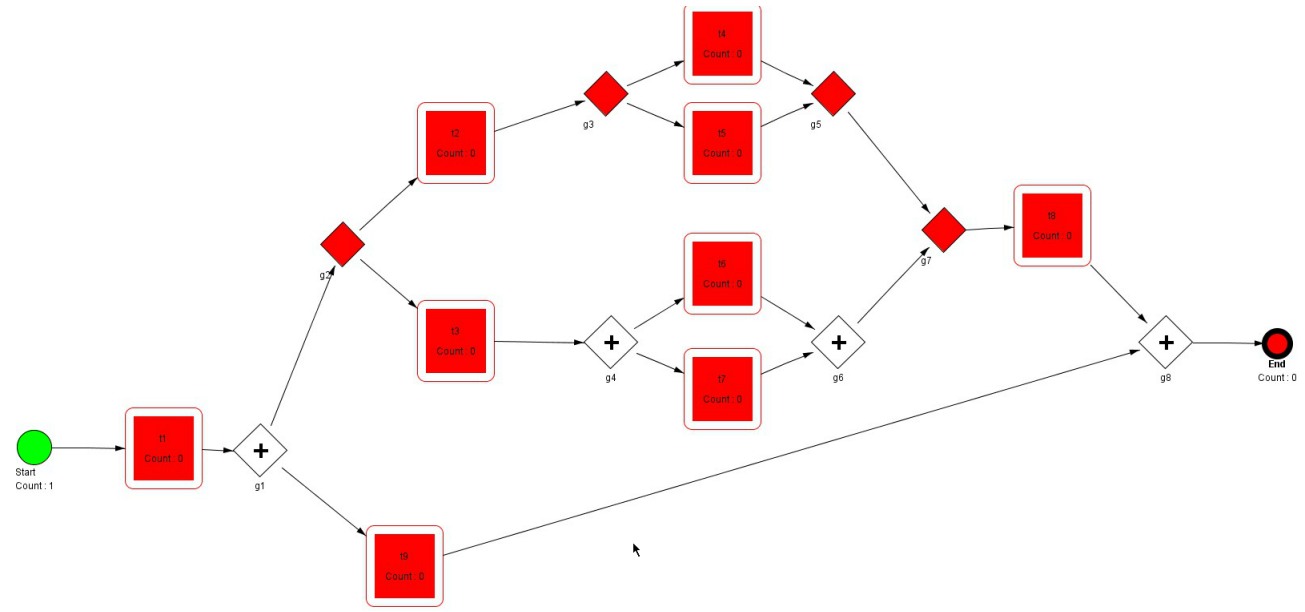

FIGURE 6.2 – Animation de BPMN : Exemple 1

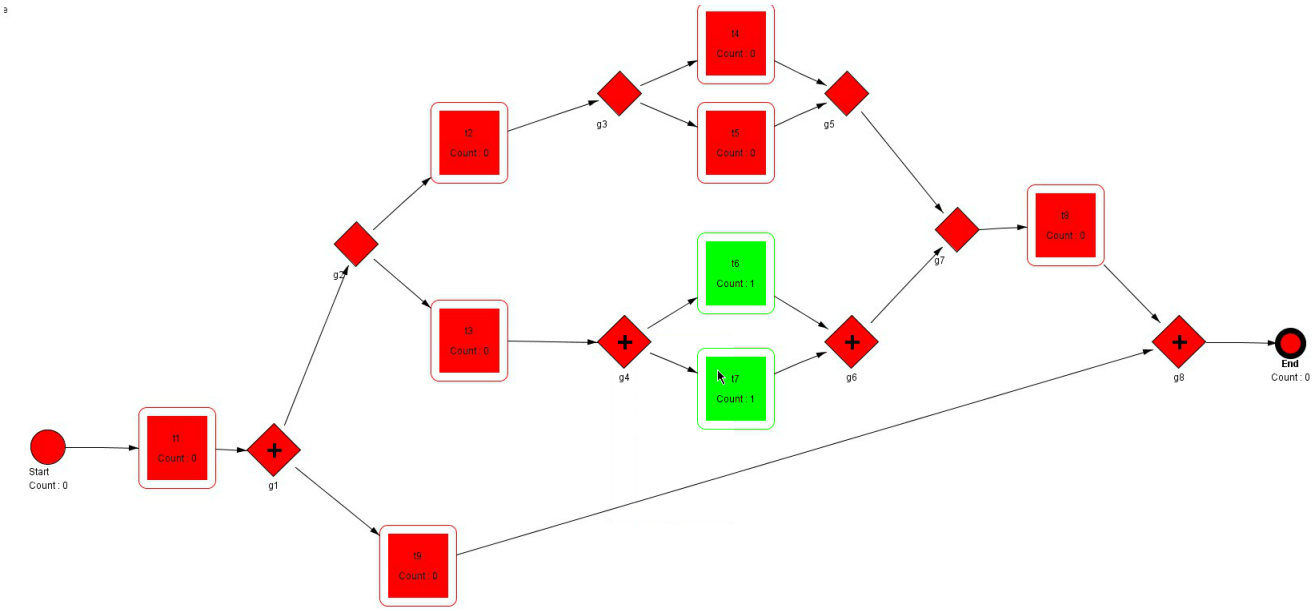

FIGURE 6.3 – Animation de BPMN : Exemple 2

6.2.3 Grasyla

L'animation de BPMN est gérée par un modèle Grasyla. Une propriété `count` a été ajoutée à l'élément *FlowNode* (supertype des éléments de base) afin de représenter un nombre de jetons détenu par un élément. La partie modélisation est identique à l'éditeur duquel les propriétés ainsi que les éléments non gérés ont été retirés. Le passage des jetons est réalisé à l'aide de fonctions Groovy appelées par le menu contextuel de chaque élément. La portion de code suivante illustre l'exécution d'un évènement. Prenant l'événement en entrée, la fonction `performEvent(event)` vérifie que la propriété symbolisant le jeton est positive et non nulle. Ensuite, toutes les propriétés des éléments suivant l'évènement selon le flux sont incrémentées de 1. Finalement, le compteur de l'événement est décrémenté. Le principe de cette fonction est applicable aux autres éléments comme les *Task* ou les *Parallel Gateway* avec de légères adaptations. La fonction `resetGatewayColor()` sert à redéfinir les couleurs adéquates pour chaque élément du modèle selon son compteur de jetons.

```
1  static def performEvent(event) {
2          def v = event.getOnePropertyValue(count)
3          if(v > 0 ) {
4              def next = event.getAllConcreteRolesDomainCO(sequenceflow)
5              for ( e in next) {
6                  increment(e, 1)
7              }
8              increment(event, -1)
9          }
10         resetGatewayColor()
11     }
```

6.2.4 Conclusion

Cette démarche visait à montrer une autre utilisation de Grasyla à travers une animation interactive d'un modèle par des couleurs. L'animation d'un processus composé de quelques éléments dont les portes exclusives et parallèles servent aussi à illustrer l'exécution d'une instance selon les flux (*SequenceFlow*) empruntés. Il faut noter que cet animateur ne respecte pas la sémantique de BPMN 2.0. Il ressort que Grasyla permet, grâce des mécanismes comme les scripts Groovy décrits précédemment, d'animer un modèle BPMN. Le script Grasyla de modélisation, restreint aux éléments gérés par l'animateur, a été étendu d'environ 200 lignes. Donc, la structure de modélisation a pu être réutilisée dans une autre finalité que la modélisation sans nécessiter un investissement important.

6.3 Critique de BPMN

La réalisation de ce travail a permis de mettre en valeur les aspects positifs et négatifs de BPMN 2.0.

Par rapport à ses principaux concurrents comme YAWL ou UML, BPMN 2.0 se démarque en proposant une notation riche en constructions. La nouvelle version (2.0) a introduit de nouveaux éléments permettant d'augmenter l'expressivité afin de couvrir les besoins de modélisation de processus de plus en plus complexes. Comme décrit en 3.1.4, BPMN 2.0 est composé de 171 symboles tandis que YAWL n'en compte que 14. Cependant, comme le souligne N.Genon [GHA11], on peut demander aux experts s'il est nécessaire d'avoir 171

symboles même si le but est de produire des modèles détaillés pour d'autres experts ou pour l'exécution de workflow.

En plus de la notation (trop ?) riche, BPMN 2.0 présente un métamodèle d'une taille comparable. Comme nous l'avons vu dans la section 3.3, il possède 147 classes UML qui sont décrites en 300 pages dans le standard [OMG11a]. Après la réalisation de l'éditeur, il ressort que le métamodèle a été conçu en fonction de la notation et non l'inverse. En effet, la version 1.2 de BPMN, qui reprend plusieurs constructions raffinées dans la version 2.0, n'était pas décrite par un métamodèle. Les concepteurs de BPMN 2.0 ont conçu le métamodèle afin de pouvoir représenter l'ensemble des constructions graphiques et leurs différents attributs. La méthodologie à suivre pour l'implémentation d'un éditeur au sein de MetaDone impose de décrire le métamodèle pour ensuite exprimer les métaobjets dans une forme graphique, ce qui se prêtait bien à la conception de BPMN comme le métamodèle est conçu pour représenter toutes les constructions. D'autre part, cette conception a induit quelques problèmes qui ont été une source d'obstacle dans l'implémentation de l'éditeur. En effet, un point important était de garder l'adéquation entre le modèle construit visuellement avec symboles de BPMN et le modèle sous-jacent formé par les objets concrets.

Le métamodèle comporte plusieurs curiosités parmi les suivantes.

Premièrement, le métaobjet *RessourceRole*, relié à une *Activity* (0..1 - *) permet de retrouver les *Performer* d'une tâche spécifique. Cependant, la relation 0..1 ne permet pas de retrouver les tâches auxquelles un performer est assigné. De cette façon, on sacrifie l'information liée au métamodèle (la relation * - * permettrait d'exprimer davantage) au profit de la notation graphique.

Deuxièmement, les types de symboles représentant les événements sont définis par un sous-ensemble de modèle comportant plusieurs objets. Concrètement, un *StartEvent* peut être lié à plusieurs *MessageEventDefinition* qui est la superclasse des types d'événements précis comme Signal ou Message. L'élément graphique est donc la représentation de 2 objets (*StartEvent* et *MessageEventDefinition*) liés par un rôle. Cette complexité implique des difficultés dans la conception d'un éditeur.

Enfin, l'artéfact *Group* comporte un titre en haut à gauche dans la notation graphique. Cependant, le métamodèle ne mentionne pas de propriété représentant ce titre. Dans le métamodèle de l'éditeur, une propriété a été ajoutée au métaobjet *Group* afin de contenir cette information.

Plusieurs auteurs proposent une adaptation de BPMN. Moody [Moo09] propose deux dialectes. L'un composé d'un sous-ensemble des éléments BPMN composant une version simplifiée (pour la modélisation à la main) de la seconde qui comprendrait l'entièreté des éléments (pour la modélisation assistée par ordinateur). Ces différentes adaptations ont comme point commun de simplifier la notation, tandis que la version 2.0 de BPMN apporte justement plus de complexité. Cela projette peut-être les prémices d'une future version de BPMN.

6.4 Critique de MetaDone

Lors de l'implémentation et des tests de l'éditeur BPMN, MetaDone s'est révélé être un logiciel très stable. Malgré les erreurs provoquées par le chargement d'un script Grasyla mal construit, la partie kernel de MetaDone n'a jamais éprouvé le moindre bug. En effet, chaque fonction est documentée et chaque test de précondition est rigoureusement implémenté.

Entre autres, l'accès aux différentes fonctions par l'intermédiaire des interfaces permet à

un utilisateur d'ajouter et modifier un plugin sans connaître le fonctionnement interne de MetaDone.

Du point de vue du repository, la base de données H2 remplit parfaitement son rôle en stockant les différents modèles. On remarque cependant que les modèles concrets sont identifiés par un numéro composé de chiffres et ne peuvent être renommés par l'interface de MetaDone, au contraire des vues qui leur sont associées. Il serait intéressant de permettre de modifier ce numéro en une chaîne de caractères ayant un sens pour l'utilisateur. Aussi, une miniature du modèle sous forme d'image associée à chaque vue pourrait apporter une meilleure distinction entre les modèles.

Du point de vue de la documentation, le code de MetaDone est entièrement documenté sous forme de JavaDoc. De plus, plusieurs rapports techniques concernant MetaL 1 et 2 ([Eng06, Eng07] ainsi que la procédure d'implémentation d'un plugin de base ([ME11]) permettent à un utilisateur novice d'avoir une première approche de MetaDone.

6.4.1 Grasyla 2

Du point de vue de la documentation, le langage Grasyla est rigoureusement décrit dans un manuel technique d'environ 80 pages ([EM13]). D'abord, ce document permet aux néophytes d'acquérir les premières bases de Grasyla à travers une introduction à MeTaL ainsi que plusieurs exemples représentatifs comme la modélisation d'un réseau de Petri en Grasyla. De plus, Grasyla est défini par une grammaire BNF (Backus Naur Form). Par la suite, ce document devient un manuel de référence contenant toutes les différentes constructions possibles en Grasyla. Chaque expression (if, svg, box, ...) est recensée dans ce catalogue. Une expression peut avoir différentes options, actions (clics, double clic, ...) qui sont documentées. De plus, lorsqu'une expression est d'un niveau plus avancé, un exemple accompagne la description. Le document présente aussi les interfaces accessibles par les scripts Groovy qui offrent un accès au workspace de MetaDone. Enfin, le modèle Grasyla des réseaux de Petri est expliqué en plusieurs pages détaillées, ce qui permet d'avoir un exemple complet qui reprend les points importants de Grasyla.

Du point de vue de l'apprentissage du langage, Grasyla est un langage à part des langages de programmation comme Java ou C. Il utilise une syntaxe différente des langages orientés objet, ce qui peut être un obstacle dans la première approche du langage. Plusieurs constructions, comme les if-then-else, sont relativement complexes et nécessitent une attention particulière lors de l'écriture d'un script. Les foncteurs, qui permettent de grouper et différencier plusieurs équations, sont difficiles à appréhender lors de la première approche. D'autres facettes de Grasyla peuvent constituer des obstacles à l'apprentissage du langage. Cependant, plusieurs aides sont à la disposition des utilisateurs comme la documentation très complète et détaillée de Grasyla. Plusieurs langages, du plus simple au plus compliqué, sont implémentés à l'aide de Grasyla et peuvent être une autre manière d'approcher le langage. Finalement, après avoir expérimenté Grasyla pendant plusieurs jours, les mécanismes se révèlent être répétitifs ce qui permet de se détacher petit à petit de la documentation et de réaliser des constructions plus complexes.

Du point de vue de la maintenance et de la programmation, un script Grasyla est facile à maintenir et à modifier. Nous l'avons vu dans le cas de BPMN, la personnalisation d'un élément peut se résumer à la modification d'une dizaine de lignes localisées. Par exemple, Grasyla permet de définir une représentation d'un cercle pointillé contenant un rectangle contenant lui même une chaîne de caractères en quelques lignes de code. L'ajout d'une équation

afin d'introduire un nouveau concept est possible sans pour autant nuire aux autres équations déjà implémentées. Par rapport à des libraires graphiques comme Swing, Grasyla permet de gérer les constructions en un nombre réduit de lignes. Cependant, Grasyla ne permet pas de gérer autant de paramètres que Swing bien que l'ajout d'options dans les différentes expressions est possible. Lors de la programmation, on remarque rapidement l'absence d'un environnement de développement dédié à Grasyla. Il est plus confortable pour un programmeur d'utiliser un outil comme Notepad++, associé avec le fichier décrivant la syntaxe de Grasyla, afin de bénéficier de la coloration syntaxique. Cependant, les éventuelles erreurs syntaxiques n'apparaissent pas et il est nécessaire de charger le script dans MetaDone afin de déceler les éventuelles erreurs de programmation. Il serait intéressant de développer un plugin Eclipse qui prendrait en charge la coloration syntaxique, la détection d'erreurs ainsi que l'autocomplétion. Un tel environnement, si le nombre d'utilisateurs de Grasyla augmente, serait un atout.

Du point de vue du chargement des scripts, les modèles Grasyla peuvent être chargés autant de fois que nécessaire sans redémarrage de MetaDone. En phase de développement, il faut d'abord définir le métamodèle avant de charger le script Grasyla décrivant sa syntaxe concrète. Une fois le script chargé au sein de MetaDone et après sauvegarde du workspace, il n'est plus nécessaire de charger de nouveau le script Grasyla lors d'un prochain démarrage de MetaDone. Ceci permet un gain de temps, le temps de chargement ne dépendant plus que de la complexité du métamodèle.

Du point de vue de la complétude fonctionnelle, Grasyla permet de décrire des langages à la notation complexe comme BPMN. Les formes de base comme les formes géométriques ainsi que les SVG ont permis de représenter la quasi-totalité de la notation BPMN. Ainsi, il serait possible d'implémenter plusieurs langages connus comme les diagrammes de classe, les diagrammes d'activité UML ou d'autres notations personnalisées sans ajouter de fonctionnalités à Grasyla. L'utilisation des scripts Groovy permet une manipulation avancée des objets et du workspace à l'aide de fonctions proches de Java. Cependant, Grasyla pourrait encore être amélioré dans une future version. Voici quelques suggestions :

– Grasyla ne permet pas directement de créer plusieurs objets en même temps et de les lier par des rôles. Cela ne peut se faire graphiquement en créant chaque objet séparément. Dans ce travail, l'éditeur utilise des scripts Groovy afin de permettre ces manipulations avancées, ce qui surcharge la manipulation lors de l'édition d'un modèle. Il faudrait pouvoir extraire les scripts Groovy de Grasyla sous forme de fichiers qui seraient gérés par l'interpréteur côté Java plutôt que du côté Grasyla. Le script Grasyla ne comporterait plus que l'appel à ces fonctions, ce qui allégerait la structure du fichier.
  Par exemple, la portion de code suivante illustre l'appel à une classe Groovy interne au fichier Grasyla. La classe *Util* est définie dans la première partie du fichier et contient une fonction `doSomething()`. Dans la suite du même fichier Grasyla, on distingue plusieurs appels à cette fonction.

```
1 //Fichier Grasyla - BPMN 2.0
2
3 //Classe et implementation
4 script grv s{{{
5     import metadone.metabusiness.script.groovy.Context;
6
7     class Util {
8         static def doSomething() {
```

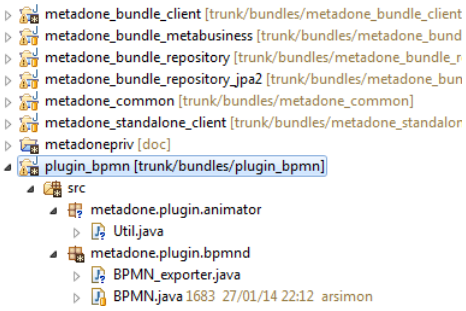

FIGURE 6.4 – Classe Util extraite

```
9                 ...
10            }
11        }
12 }}}
13
14 ...
15
16 //Appel
17 grv{s{{{Util.doSomething()}}}}
```

La proposition serait d'extraire cette classe du fichier Grasyla en gardant l'appel à la fonction. La classe extraite serait une classe Java faisant partie du plugin (voir figure 6.4 illustrée par la portion de code suivante.

```
1 //Classe Java externe au fichier Grasyla
2
3 import metadone.metabusiness.script.groovy.Context;
4
5 public class Util {
6     public void doSomething() {
7         ...
8     }
9 }
```

– Lors de l'implémentation des propriétés dans le script Grasyla, il est coutume d'associer une propriété à un champ d'édition (TextField) en Swing. L'utilisateur peut introduire la valeur de la propriété dans le champ qui lui est associé. Cependant, la création d'un TextField pour une propriété est très répétitive et nécessite 4 à 5 lignes. Dans le cas de BPMN, le nombre important de propriétés augmente considérablement le nombre de lignes à produire pour les implémenter. Une solution serait d'associer des raccourcis à l'expression d'un TextField d'une propriété. Ainsi, il serait possible de définir un champ d'édition pour une propriété en une seule ligne. Ces raccourcis seraient aussi associés à d'autres types de champs d'édition comme des *CheckBox* ou des *ComboBox*.
La portion de code suivante illustre un tableau composé d'une ligne comportant le nom d'une propriété "Name" ainsi qu'un champ d'édition affichant la valeur de la propriété et permettant de la modifier. La proposition serait de remplacer l'expression *textField* par une nouvelle expression *ShortTextField* au comportement équivalent. Ainsi, le nombre de lignes serait réduit.

```
1 table {
```

```
2      tr { "Name" textfield {
3                   value($"FlowElement.name")
4                   action: "validate" {
5                         update{$"FlowElement.name"}
6                   }
7            }
8      }
9      ...
10 }
11
12 table {
13     tr { "Name" ShortTextfield { $"FlowElement.name" }
14     }
15     ...
16 }
```

– Dans le cas de BPMN, les sous-processus pouvaient faire référence à un autre modèle d'un même workspace. Il serait intéressant de permettre d'implémenter des liens "hypertext" qui permettraient d'afficher un autre modèle lors d'une action sur un élément. Concrètement, un double clic sur un sous-processus BPMN ouvrirait une autre fenêtre contenant le processus ciblé.

Du point de vue des langages déjà implémentés, Grasyla permet d'exprimer une notation en un nombre réduit de lignes. Plusieurs langages sont déjà implémentés comme les réseaux de Petri, URN, GRL ou encore BPMN. Ces différents langages prouvent que Grasyla est bien adapté à la description des syntaxes concrètes.

6.5 Future work

L'éditeur BPMN permet de modéliser des modèles de la partie *Process*, ce qui couvre la majeure partie des besoins en modélisation BPMN. Les deux autres parties (*Collaboration* et *Choreography*) pourront être ajoutées à l'éditeur. L'utilisation de fichiers SVG ou l'ajout d'une nouvelle forme de base à Grasyla permettra de gérer les éléments propres à ces parties. Concrètement, il faudrait ajouter trois nouvelles expressions à Grasyla illustrées à la figure 6.5. Le premier élément est une *Choreography Task*. Le second est un sous-processus de Choreography pouvant contenir d'autres éléments lorsqu'il est étendu. Le dernier élément est une relation de Conversation. Concernant les autres éléments, leur intégration peut se faire à l'aide d'images SVG. Donc, l'intégration des deux parties (*Collaboration* et *Choreography*) est réalisable si ces derniers éléments sont ajoutés à Grasyla.

Concernant l'exportation des modèles, nous avons vu qu'un embryon d'export en BPMN XML a été implémenté. A l'aide du standard BPMN, la suite de l'implémentation de l'exportation devrait nécessiter quelques jours de travail.

Concernant la simulation, l'animation proposée dans ce travail ne propose qu'une base réduite pour un futur simulateur. En effet, en raison des nombreux éléments et propriétés à prendre en compte, le développement d'un simulateur serait difficile. Il serait plus productif d'exporter le modèle en forme XML et d'utiliser un simulateur déjà implémenté en important le modèle.

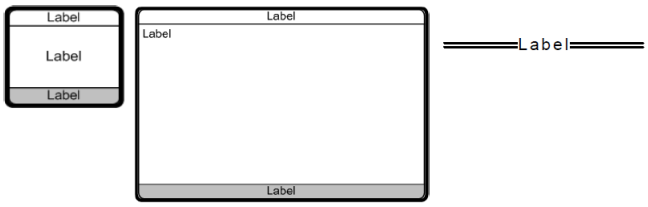

FIGURE 6.5 – Nouveaux éléments

Chapitre 7

Conclusion

Dans ce mémoire, nous avons présenté une approche permettant l'implémentation d'un langage de modélisation, BPMN, au sein d'un outil de métamodélisation, MetaDone. Le but était de montrer la capacité de MetaDone, à l'aide du langage Grasyla, à gérer un langage complexe autant par l'importance de son métamodèle que par la richesse de ses constructions.

BPMN a été présenté et il ressort qu'il s'agit d'un langage composé de nombreuses constructions et doté d'un métamodèle complexe. D'autre part, MetaDone a été le support d'implémentation de plusieurs langages comme les réseaux de Petri et URN, ce qui nous indiquait des perspectives optimistes concernant l'implémentation d'un éditeur BPMN.

En implémentant cet éditeur, il a été montré que MetaDone fournit un environnement pouvant supporter l'intégration de langages de modélisation aux constructions variées. De plus, l'accomplissement de ce travail montre que Grasyla 2 est un langage permettant la spécification d'une interface graphique adaptée à un langage de modélisation.

Bibliographie

[Ald91] Albert Alderson. MetaCASE technology. In *Software Development Environments and CASE Technology*, pages 81–91. Springer, 1991.

[All11] T. Allweyer. *BPMN 2.0 : Introduction to the Standard for Business Process Modeling*. Books on Demand, 2011.

[ATo] AToM3 a tool for multi-formalism meta-modelling. http://atom3.cs.mcgill.ca/.

[CN89] Minder Chen and Jay F Nunamaker. MetaPlex : an integrated environment for organization and information system development. In *Proceedings of the tenth international conference on Information Systems*, pages 141–151. ACM, 1989.

[CR88] Elliot J. Chikofsky and Burt L. Rubenstein. CASE : reliability engineering for information systems. *Software, IEEE*, 5(2) :11–16, 1988.

[CT12] Michele Chinosi and Alberto Trombetta. BPMN : An introduction to the standard. *Computer Standards & Interfaces*, 34(1) :124 – 134, 2012.

[dLV02] Juan de Lara and Hans Vangheluwe. Using AToM as a meta-case tool. In *ICEIS'02*, pages 642–649, 2002.

[DtH01] Marlon Dumas and Arthur H.M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In Martin Gogolla and Cris Kobryn, editors, *"UML" 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin Heidelberg, 2001. http://dx.doi.org/10.1007/3-540-45441-1_7.

[Ecl14] Eclipse CDT, 2014. http://www.eclipse.org/.

[EK00] Eric Engstrom and Jonathan Krueger. Building and rapidly evolving domain-specific tools with DOME. In *Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on*, pages 83–88. IEEE, 2000.

[EM13] Vincent Englebert and Krzysztof Magusiak. The grasyla 2 language. Technical report, University of Namur - The PReCISE Research Center, 2013.

[Eng00] Vincent Englebert. *A Smart MetaCASE : Towards an Integrated Solution*. PhD thesis, University of Namur, Computer Science Dept. Rue grandgagnage 21. 5000 Namur. Belgique, May 2000.

[Eng06] Vincent Englebert. Metal1 : a formal specification. Technical report, University of Namur, 2006.

[Eng07] Vincent Englebert. Metal2 : a formal specification. Technical report, University of Namur, 2007.

[Eng09] Vincent Englebert. The MetaDone architecture : an overview. Technical report, University of Namur, 2009.

[ESU97] Jürgen Ebert, Roger Süttenbach, and Ingar Uhe. MetaCASE in practice : a Case for KOGGE. In *IN*, pages 203–216. Springer, 1997.

[GHA11] Nicolas Genon, Patrick Heymans, and Daniel Amyot. Analysing the cognitive effectiveness of the BPMN 2.0 visual notation. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 377–396, Berlin, Heidelberg, 2011. Springer-Verlag. http://dl.acm.org/citation.cfm?id=1964571.1964605.

[GME08] GME : Generic modeling environment, 2008. http://www.isis.vanderbilt.edu/Projects/gme/.

[IDE14] IDEF0 Function Modeling Method, 2014. http://www.idef.com/.

[IL97] Hosein Isazadeh and David Alex Lamb. CASE Environments and MetaCASE Tools, 1997.

[Jen87] Kurt Jensen. *Coloured petri nets*. Springer, 1987.

[KK84] Jeffrey E Kottemann and Benn R Konsynski. Dynamic metasystems for information systems development. In *Proc. of the 5th Intl. Conf. on Information Systems*, pages 187–204, 1984.

[KS96] Steven Kelly and Kari Smolander. Evolution and issues in metaCASE. *Information and Software Technology*, 38(4) :261 – 266, 1996.

[KT08] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling : Enabling Full Code Generation*. Wiley, 2008. http://www.bibsonomy.org/bibtex/2c901dbe97bad0c4ad96b73f04d39a02a/voj.

[Leo99] John Leonard. Systems engineering fundamentals. Technical report, DTIC Document, 1999. http://ocw.mit.edu/courses/aeronautics-and-astronautics/16-885j-aircraft-systems-engineering-fall-2005/readings/sefguide_01_01.pdf.

[LKT04] Janne Luoma, Steven Kelly, and Juha-Pekka Tolvanen. Defining domain-specific modeling languages : Collected experiences. In *4 th Workshop on Domain-Specific Modeling*, 2004.

[LMB+01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, volume 17, 2001.

[McC88] Carma McClure. *CASE is software automation*. Prentice-Hall, Inc., 1988.

[ME11] Krzysztof Magusiak and Vincent Englebert. MetaDone : plugin architecture. Technical report, University of Namur, 2011.

[Met04] MetaCase. ABC TO METACASE TECHNOLOGY, 2004. http://www.metacase.com/papers/ABC_to_metaCASE.pdf.

[Met14a] MetaDone - DSML Environement, 2014. http://www.metadone.be.

[Met14b] MetaEdit+ homepage, 2014. http://www.metacase.com/mep/.

[Moo09] Daniel Moody. The physics of notations : Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Softw. Eng.*, 35(6) :756–779, November 2009. http://dx.doi.org/10.1109/TSE.2009.67.

[MRTL93] Pentti Marttiin, Matti Rossi, Veli-Pekka Tahvanainen, and Kalle Lyytinen. A comparative review of CASE shells : A preliminary framework and research outcomes. *Information & management*, 25(1) :11–31, 1993.

[Mur89] Tadao Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[OMG11a] OMG. Business process model and notation (BPMN) version 2.0. Technical report, Object Management Group (OMG), jan 2011. http://taval.de/publications/BPMN20.

[OMG11b] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. Technical report, Object Management Group, August 2011. http://www.omg.org/spec/UML/2.4.1.

[Ope14] Openflexo (https ://openflexo.org/), 2014. https ://openflexo.org/.

[OS97] Andreas L. Opdahl and Guttorm Sindre. Facet modelling : An approach to flexible and integrated conceptual modelling. *Information Systems*, 22(5) :291 – 323, 1997.

[PK91] Dewayne E Perry and Gail E Kaiser. Models of software development environments. *Software Engineering, IEEE Transactions on*, 17(3) :283–295, 1991.

[RK99] Matti Rossi and Steven Kelly. Construction of a CASE tool : The case for metaedit+. CoSET'99, 1999.

[S+92] August-Wilhelm Scheer et al. *Architecture of integrated information systems*. Springer, 1992.

[Sil07] B. Silver. *BPMN and the Business Process Expert*. 2007. https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/10852310-ac6a-2a10-02be-d83f4d2dd647.

[SLTM91] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. Metaedit—a flexible graphical environment for methodology modelling. In *Advanced Information Systems Engineering*, pages 168–193. Springer, 1991.

[THV96] Arthur HM Ter Hofstede and TF Verhoef. MetaCASE : Is the game worth the candle ? *Information Systems Journal*, 6(1) :41–68, 1996.

[TR03] Juha-Pekka Tolvanen and Matti Rossi. MetaEdit+ : Defining and using domain-specific modeling languages and code generators. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 92–93, New York, NY, USA, 2003. ACM. http ://doi.acm.org/10.1145/949344.949365.

[TWTR06] A. Tsai, Jiacun Wang, W. Tepfenhart, and D. Rosca. EPC Workflow Model to WIFA Model Conversion. In *Systems, Man and Cybernetics, 2006. SMC '06. IEEE International Conference on*, volume 4, pages 2758–2763, Oct 2006.

[VDA98] Wil M. P. VAN DER AALST. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08(01) :21–66, 1998. http ://www.worldscientific.com/doi/abs/10.1142/S0218126698000043.

[vdAADtH04] Wil M. P. van der Aalst, Lachlan Aldred, Marlon Dumas, and Arthur H. M. ter Hofstede. Design and implementation of the YAWL system. In *Proceedings of Advanced Information Systems Engineering : 16th International Conference, CAiSE 2004, Riga, Latvia, June 7-11, 2004 — Volume 3084 of Lecture Notes in Computer Science / Anne Persson, Janis Stirna (Eds.)*, pages 142–159. Springer-Verlag, August 2004.

[vdAtH05] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL : Yet Another Workflow Language. *Inf. Syst.*, 30(4) :245–275, June 2005. http://dx.doi.org/10.1016/j.is.2004.02.002.

[Whi04] Stephen A White. Process modeling notations and workflow patterns. *Workflow handbook*, 2004 :265–294, 2004.

[Whi05] Stephen White. Using BPMN to model a BPEL process. *BPTrends*, 3(3) :1–18, 2005.

[Wik13a] Wikipédia. Bonita — wikipédia, l'encyclopédie libre, 2013. http://fr.wikipedia.org/w/index.php?title=Bonita&oldid=99628753.

[Wik13b] Wikipédia. Object management group — wikipédia, l'encyclopédie libre, 2013. http://fr.wikipedia.org/w/index.php?title=Object_Management_Group&oldid=89832337.

[Wik14a] Wikipédia. Smalltalk — wikipédia, l'encyclopédie libre, 2014. http://fr.wikipedia.org/w/index.php?title=Smalltalk&oldid=102227495.

[Wik14b] Wikipédia. Workflow — wikipédia, l'encyclopédie libre, 2014. http://fr.wikipedia.org/w/index.php?title=Workflow&oldid=105185623.

[Yaw14] YAWL, Yet Another Workflow Language, 2014. http://www.yawlfoundation.org/.