Synthèse d'applications de réalité virtuelle à partir de modèles

Gwendal Le Moulec

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

INSA RENNES

THESE DE DOCTORAT DE

L'INSA RENNES

COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Gwendal LE MOULEC

Synthèse d'applications de Réalité Virtuelle à partir de modèles

Thèse présentée et soutenue à Rennes, le 26.09.2018
Unité de recherche : IRISA – UMR6074
Thèse N° : 18ISAR 18 / D18 - 18

Rapporteurs avant soutenance :

Tewfik ZIADI	Maître de conférences HDR, Sorbonne Université (Paris 6)
Jean-Pierre JESSEL	Professeur des universités, Université Paul Sabatier (Toulouse)

Composition du Jury :

Daniel MESTRE	Directeur de recherches CNRS, Marseille Président
Tewfik ZIADI	Maître de conférences HDR, Sorbonne Université (Paris 6) Rapporteur
Jean-Pierre JESSEL	Professeur des universités, Université Paul Sabatier (Toulouse) Rapporteur
Valérie Gouranton	Maître de conférences, INSA Rennes Encadrant
Arnaud Blouin	Maître de conférences, INSA Rennes Encadrant
Bruno Arnaldi	Professeur, INSA Rennes Directeur de thèse

Intitulé de la thèse :

Synthèse d'applications de Réalité Virtuelle à partir de modèles

Gwendal Le Moulec

En partenariat avec :

"*Scotty : Voilà, Monsieur, je vous le laisse. Tous les systèmes sont en automatique, parés à répondre. Un chimpanzé et deux stagiaires pourraient le diriger !*
Captain Kirk : Merci, Monsieur Scott. Je vais essayer de ne pas me sentir visé."
Star Trek

"*Au fond de ce couloir sombre dans lequel je m'enfonçais à une vitesse incroyable, il y avait cette lumière qui était plus puissante que mille milliards de soleils et qui pourtant ne m'éblouissait pas. [...] Jamais de ma vie je n'ai rencontré quelque chose d'aussi puissant et d'aussi aimant que cette lumière [...].*"
Extrait du témoignage d'une personne ayant vécu une "expérience de mort imminente"

Remerciements

Merci tout d'abord aux membres du jury de cette thèse : Tewfik Ziadi, Jean-Pierre Jessel et Daniel Mestre. Je vous remercie d'avoir accepté de lire, écouter et commenter mes travaux. Vos retours sont de précieux conseils qui me permettront de m'améliorer dans la suite de ma carrière. Je remercie également mes encadrants : Bruno Arnaldi, Valérie Gouranton et Arnaud Blouin. Leurs retours réguliers et leurs conseils me sont précieux. Il a fallu beaucoup de patience pour m'encadrer pendant ces trois ans. Il y a eu des hauts et des bas. Mais au final, pour moi l'expérience est réussie, alors : merci !

Je remercie particulièrement Yacine "my bro", mon ami, mon compagnon de croisière pendant ces trois ans, que je félicite également pour l'obtention de son diplôme de doctorat. Merci aussi à Haykel, Lokman et Mohammed Yacine pour les échanges riches que nous avons eus. Je remercie également tous les collègues d'Hybrid et de DiverSE : François, Guillaume "GUI", Florian, Hakim, Yoren, Flavien, Adrien, Alexandre, Thierry, Benoît, Guillaume B, Guillaume C, Pierre, Amine, Andéol, et tous les autres. Merci à vous pour votre bonne humeur et votre soutien.

Mention spéciale aux autres enseignants-chercheurs qui m'ont aidé au cours de ma thèse : les membres de mon CSID Mathieu Acher et Richard Kulpa ; ainsi qu'Anatole Lécuyer, Ferran Argelaguet, Maud Marchal, Jean-Marc Jézéquel, Ludovic Hoyet et Benoît Combemale (j'en oublie certainement).

Merci à toute ma famille et à tous mes amis qui m'ont beaucoup soutenu moralement pendant ces trois ans. Merci beaucoup pour toutes ces pensées issues des quatre coins du pays (et au-delà), ces pensées qui m'ont certainement atteint et aidé dans les moments les plus durs. Merci du fond du Cœur, car c'est bien par le Cœur que ces communications passent ! Merci notamment à Hodaifa "Master Yoda", Idriss, Hichem, Mohamed "Moha", Mohammed R, Ismaïl, Adil, Anir, Taha (et un grand merci à son formidable grand-père, un homme très inspirant qui a changé ma vie). Et merci à tous les autres qui se reconnaîtront, mais la liste de noms est bien trop longue pour être retranscrite ici. Elle est en revanche bien présente dans mon Cœur.

Un merci encore plus intense à toute ma famille : Éliaz, Joseph, Mano, Petit Yves, mamie, mes oncles, mes tantes et mes cousins. Et bien-sûr merci à toi Papa, à toi Maman. Merci pour tout votre amour et votre éducation sans lesquels tout cela aurait été impossible.

Je garde le meilleur pour la fin : merci à toi Salma, ma formidable épouse, et à vous Zakaria et Rayan, mes chers enfants, qui ont supporté ces trois années difficiles mais ô combien formatrices.

Table des matières

Introduction

1

La Réalité Virtuelle (RV) est "*un domaine scientifique est technique exploitant l'informatique et des interfaces comportementales en vue de simuler dans un monde virtuel le comportement d'entités 3D, qui sont en interaction en temps réel entre elles et avec un ou des utilisateurs en immersion pseudonaturelle par l'intermédiaire de canaux sensorimoteurs.*" [Arnaldi et al., 2006]. Les usages de la RV peuvent être classés en plusieurs familles : robotique, psychologie, santé, éducation, formation, ingénierie, archéologie, architecture, marketing, art, cinéma, divertissement, *etc.* [Fuchs et al., 2006]. Malgré cette variété, il existe des problématiques techniques communes aux applications de RV, plus ou moins importantes selon leurs objectifs : implémentation des lois physiques, animation d'avatars, techniques d'interaction avec le monde et les objets, suivi de procédures ou d'un scénario, *etc.*

La palette d'outils numériques dont disposent les créateurs et les développeurs d'applications de RV est large. On distingue deux types d'outils : les outils de développement (*e.g.,* Unity3D, CryEngine, Unreal Engine) et les outils de design (*e.g.,* Maya, 3DSMax, Blender). Ces outils sont complémentaires. Les objets 3D que l'on peut produire dans les outils de design peuvent être importés dans les outils de développement. Ces outils de développement généralistes intègrent les composants suivants : éditeur de code pour le comportement des objets, moteur physique, outils pour l'animation d'humanoïdes, *etc.* Ils couvrent donc les problématiques techniques communes aux différentes familles d'applications de RV citées précédemment.

Cependant, certains aspects structurants dans toute application de RV ne bénéficient pas en général de modules de développement spécifiques. Par exemple, la notion de scénario est primordiale en RV. Pourtant, les outils de développement ne proposent pas de modules permettant d'en concevoir. De même, l'implémentation des techniques d'interaction se fait avec des langages de programmation généralistes (*General Purpose Languages*, GPL), *i.e.,* des langages conçus pour développer des logiciels dans tous types de domaines, tels que le Java, le C# ou le C++. Un constat communément admis dans la communauté de RV est que cela engendre d'importants efforts de développement et amène les développeurs à constamment repartir de zéro : des fonctionnalités communes à de nombreuses applications de RV sont réimplémentées à chaque fois, *e.g.,* l'action de prendre un objet. Par ailleurs, les outils sont conçus pour des ingénieurs en informatique. Cependant dans certains cas, les clients qui commandent la production d'une application de RV (*e.g.,* une école de chirurgie souhaitant utiliser des applications de RV pour la formation aux procédures chirurgicales) sont aussi les utilisateurs finaux de cette application de RV. Ils sont alors impliqués dans tout le cycle de vie du logiciel, de l'établissement des spécifications à la maintenance et à la

configuration. Les outils trop techniques ne facilitent pas leur implication, pourtant primordiale.

Il existe une branche de l'informatique dont l'un des objectifs est de résoudre ces problèmes d'abstraction et de réutilisation : l'Ingénierie Dirigée par les Modèles (IDM). L'IDM est une méthodologie de production de logiciels reposant sur la définition de modèles, qui sont une représentation abstraite d'un domaine. L'IDM a notamment pour but de travailler sur certaines parties d'un logiciel à un meilleur niveau d'abstraction que le code. Par exemple, les Lignes de Produits Logiciels (*Software Product Lines*, SPL) offrent aux ingénieurs un niveau d'abstraction adéquat pour travailler sur les différentes variantes d'un même logiciel. Clements et Nothrop définissent une SPL comme un *"ensemble de produits destinés à un marché particulier ou répondants à un certain besoin, [...] développés à partir de composants communs de manière automatisée"* [Clements and Northrop, 2002]. Les SPL permettent la réutilisation de composants pour produire des logiciels appartenant à la même famille [Parnas, 1976, Acher, 2011]. Pour cela, les SPL peuvent utiliser des modèles de variabilité logicielle (*Feature Models*, FM), représentant les composants qui peuvent être utilisés pour produire des logiciels d'une même famille, ainsi que les relations d'interdépendance qui existent entre eux. Le but est de produire plus vite et de manière simplifiée des logiciels de meilleure qualité [Bass et al., 2003, Clements and Northrop, 2002, Pohl et al., 2005] de façon à ce que le coût de production des composants réutilisables soit inférieur au gains liés à la production par réutilisation [Deelstra et al., 2005].

Un autre concept venant de l'IDM est celui de langages dédiés (*Domain-Specific Languages*, DSL). Les DSL permettent de travailler sur un problème à l'aide de langages conçus pour traiter des problématiques spécifiques à un domaine. En cela, ils s'opposent aux GPL. En général, les DSL se fondent en effet sur des mots-clés, des notations, des modèles et des syntaxes familières aux experts du domaine visé [Wile, 2004].

Ces approches reposent sur le fait que l'un des moyens pour produire plus rapidement des logiciels est la réutilisation de concepts et de composants, et que cela nécessite de l'abstraction vis-à-vis du code. En effet, pour que des composants soient réutilisables efficacement, il est nécessaire de disposer de modèles décrivant ces composants et la manière dont ils doivent être réutilisés, ainsi que leur contexte d'utilisation. Par exemple, il faut décrire les interdépendances qui existent entre ces composants. Or les développeurs d'applications de RV et les outils de développement se focalisent sur le code et ne fournissent pas de moyens de plus haut niveau pour faciliter la réutilisation de composants de RV. Toutefois, des efforts sont faits en ce sens : des chercheurs en RV se sont intéressés notamment à la formalisation de la notion de scénario [Cavazza et al., 2002, Claude, 2016] et de modélisation des objets et des interactions [Chevaillier et al., 2012, Bouville et al., 2015]. Même si ces travaux proposent des abstractions pour des éléments fondamentaux en RV, l'établissement de ces abstractions n'est pas guidé par les méthodes issues de l'IDM. Ce manque d'expertise mène à des abstractions incomplètes et donc à un manque à gagner en terme de réutilisation. Par exemple, Grübel *et al.* ont proposé une bibliothèque pour l'éditeur de RV Unity3D permettant de produire des évaluations expérimentales en RV [Grübel et al., 2016]. Cette bibliothèque fondée sur une abstraction des concepts structurant les évaluations expérimentales en RV permet la réutilisation de ces concepts. Par exemple, elle permet l'utilisation de classes existantes pour la spécification des

conditions expérimentales, alors que ces développements sont classiquement faits à la main pour chaque nouvelle évaluation expérimentale. Cependant cette bibliothèque reste très dépendante de Unity3D et de son fonctionnement. Le concepteur d'une évaluation doit gérer à la fois la complexité liée à Unity3D et à l'utilisation des concepts structurant des évaluations expérimentales en RV, ce qui est contraire au principe de séparation des préoccupations (*Separation of Concerns*, SoC) en IDM [Logre, 2017]. Par exemple, les DSL ou les modèles UML sont autant de moyens de représenter les concepts indépendamment d'un éditeur d'applications de RV ou d'un GPL. En résumé, la production d'applications de RV manque d'abstractions, ce qui provoque : (a) un manque de réutilisation et (b) un manque de SoC.

Des manques existent aussi en IDM. Par exemple, les scénarios sur lesquels reposent les applications de RV peuvent être très complexes. D'une étape à l'autre d'un scénario, les actions à réaliser changent, ainsi que les objets qui peuvent être manipulés, de manière conséquente dans certains cas. Ce sont donc des changements de configurations qui peuvent nécessiter une gestion rigoureuse de la variabilité, notamment par l'utilisation de FM. Or les FM seuls ne permettent pas de "scénariser" la configuration, ils ne permettent que des configurations statiques (*e.g.,* on peut déterminer quels éléments caractéristiques d'une famille d'applications de RV seront présents dans une application de RV en particulier, mais on ne peut pas exprimer de changement séquencé de la configuration). Par extension, les SPL ne permettent pas de produire de manière rigoureuse des applications dans lesquelles plusieurs configurations sont utilisées de manière séquentielle. En IDM, Ziadi *et al.* se sont intéressés à la gestion de la variabilité de scénarios [Ziadi et al., 2002]. Cependant cette approche est adaptée à des cas de scénarios dont le nombre de configurations possibles est limité. Or, les scénarios en RV ont potentiellement un nombre très élevé, voire infini, de configurations possibles. Les SPL et les FM ne permettent pas de gérer la variabilité de scénarios au nombre de configurations potentiellement infini.

L'objectif principal que nous nous fixons dans cette thèse est de proposer une approche permettant de faciliter la production des applications de RV, ce qui conduit aux sous-objectifs suivants :

- **O1-ABSTRACTION** : l'approche proposée doit reposer sur les abstractions des éléments structurant les applications de RV (*e.g.,* notions d'objets virtuels, de comportements, de scénarios, *etc.*) ;
- **O2-RÉUTILISATION** : l'approche proposée doit reposer sur la réutilisation de composants et de modèles issus des abstractions définies ;
- **O3-SOC** : l'approche proposée doit reposer sur le principe de SoC, notamment pour séparer les tâches de modélisation et d'implémentation, mais aussi la définition de scénarios d'une part et la définition des objets 3D et de leurs comportements d'autre part ;
- **O4-SPL-SCÉNARIOS** : l'approche proposée doit appliquer le principe de développement de familles de logiciels des SPL, d'une manière adaptée à la production de logiciels reposant sur un scénario, comme les applications de RV.

Afin d'atteindre ces objectifs, nous proposons deux approches qui comblent respectivement les manques en IDM et en RV : l'approche LPLOS (Ligne de Produits Logiciels Orientés Scénarios) et l'approche LPLRV (Ligne de Produits Logiciels pour

la RV). L'approche LPLOS repose sur un modèle de scénarios qui manipule un FM : chaque étape du scénario correspond à une configuration du FM, *i.e.,* un ensemble cohérent de composants du logiciel à produire. L'approche LPLRV repose sur l'approche LPLOS. Le scénario supervise la manipulation des objets virtuels en s'appuyant sur un modèle objets-relations représentant les objets et interactions possibles dans l'application de RV.

Ce manuscrit s'organise de la manière suivante :

— le Chapitre 2 présente l'état de l'art. Celui-ci présente d'une part les approches existantes en IDM pour faciliter la production de logiciels. D'autre part, il présente les techniques existantes en RV pour structurer le développement d'applications de RV. Le chapitre se termine sur une synthèse exposant les limites de l'IDM et de la RV en terme de production optimisée des applications de RV ;

— les Chapitres 3 et 4 présentent respectivement les approches LPLOS et LPLRV. Dans les deux cas, après avoir détaillé les abstractions, concepts et algorithmes qui les caractérisent, nous montrons leur application sur des cas d'utilisation. Pour l'approche LPLOS, nous montrons comment elle peut être appliquée pour la production semi-automatique de documentation pour les DSL. Pour l'approche LPLRV, nous montrons comment elle peut être appliquée pour la production d'applications de RV pour l'évaluation expérimentale en RV, puis pour la production d'applications de RV pour la formation à des procédures chirurgicales ;

— le Chapitre 5 termine la thèse sur une conclusion qui fait un résumé de ce manuscrit et présente des pistes de travaux futurs.

État de l'art

2

Ce chapitre fait l'état de l'art des méthodes existant aujourd'hui pour produire des logiciels à l'aide de procédés d'abstraction, de réutilisation et de SoC. Tout d'abord nous étudierons le cas général en présentant des méthodes issues de l'IDM (Section 1). Ensuite nous étudierons les abstractions qui ont été faites en RV (Section 2). Enfin nous ferons une synthèse en identifiant les limites en IDM et en RV, pour proposer l'approche qui sera la nôtre dans cette thèse (Section 3).

1 Ingénierie Dirigée par les Modèles

L'industrie du logiciel doit faire face à une augmentation constante de la complexité des logiciels. La modélisation permet d'améliorer la maîtrise de cette complexité par le biais de l'IDM [Schmidt, 2006]. En IDM, les modèles se focalisent sur des problèmes spécifiques liés à des domaines particuliers pour faciliter le développement de logiciels. L'IDM aide les ingénieurs en informatique à développer des langages conçus pour traiter des problèmes spécifiques ; ces langages sont les DSL [van Deursen et al., 2000, Fowler, 2010, Mernik et al., 2005]. L'IDM permet également de gérer la production de logiciels partageant une base commune mais présentant des aspects pouvant varier à l'aide des SPL.

Cette section présente tout d'abord une vue d'ensemble du concept de SPL (Section 1.1). Ensuite, elle présente en détails les autres concepts de l'IDM qui peuvent être utilisés en complément des SPL : modélisation de domaines (Section 1.2), et DSL (Section 1.3).

1.1 Lignes de Produits Logiciels

Comme cela a été déjà dit précédemment, une SPL permet de gérer efficacement la synthèse (*i.e.,* la production par réutilisation) de produits logiciels appartenant à une même famille et partageant de ce fait une base commune. Chaque produit logiciel membre d'une même famille a également ses spécificités. On appelle "variabilité" la différenciation qui existe entre ces différents produits logiciels du fait de ces spécificités.

Il existe une approche permettant de formaliser le concept de SPL. Cette approche est représentée en Figure 2.1. L'idée générale sous-tendue par cette approche est principalement de séparer l'ingénierie de domaine de l'ingénierie d'application, *i.e.,* produire des composants attachés à un domaine, utilisables pour synthétiser des logiciels présentant des points communs et des différences (Section 1.1.1). Cette réutilisation de composants se fait par séparation de la spécification et de l'implémentation (Section 1.1.2).

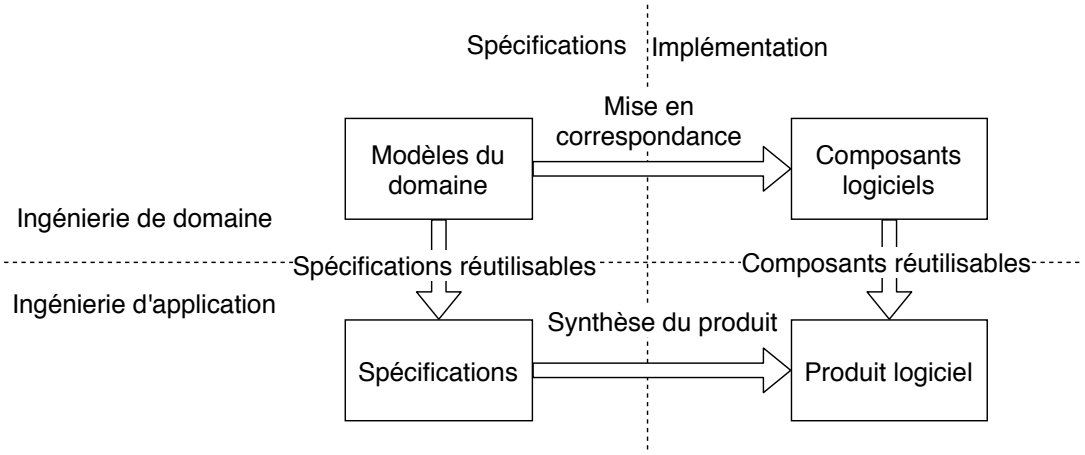


Figure 2.1 – Fonctionnement des SPL (d'après la représentation d'Acher [Acher, 2011]).

1.1.1 Ingénierie de domaine et ingénierie d'application

L'ingénierie de domaine consiste à produire des composants réutilisables, Krueger définissant la réutilisation comme un "*processus de création de logiciels à partir de logiciels existants plutôt qu'en repartant de zéro*" [Krueger, 1992]. Elle se découpe classiquement en quatre phases : analyse du domaine, conception, développement et test. Suite à l'analyse du domaine, on peut par exemple produire un cahier des charges listant les fonctionnalités désirables pour les produits logiciels de la SPL, puis développer une bibliothèque des composants logiciels répondant au cahier des charges. L'approche proposée se concentre sur les étapes de conception et de développement, par la modélisation du domaine et le développement de composants logiciels mis en correspondance avec les éléments du modèle. La modélisation du domaine comporte une modélisation de la variabilité par l'utilisation d'un FM.

L'ingénierie d'application consiste en la synthèse de produits logiciels par réutilisation des composants produits par l'ingénierie de domaine. La synthèse s'appuie sur la sélection des éléments de la modélisation de domaine qui feront partie du logiciel. Cette sélection permet d'exploiter la variabilité définie par le FM. Des mécanismes permettent d'automatiser la réutilisation des composants logiciels. Ces mécanismes sont présentés en Section 1.1.2.

1.1.2 Spécifications et implémentation

Ces deux parties de l'approche de la Figure 2.1 peuvent aussi être appelées "espace du problème" et "espace des solutions". L'idée est de traiter séparément deux types de complexité : la complexité intrinsèque au domaine (et aux logiciels qui peuvent en dériver) et la complexité accidentelle, liée aux technologies utilisées pour l'implémentation [Brooks, 1987]. Une mise en correspondance doit être établie entre la modélisation du domaine et les composants logiciels. Cette mise en correspondance doit pouvoir être exploitée de manière automatique par des mécanismes de synthèse d'un logiciel. Un ou plusieurs DSL peuvent être utilisés à ces fins. En effet, les DSL permettent d'abstraire la complexité liée à l'implémentation en proposant une syntaxe, des mots-clés et des notations cohérentes avec le domaine d'étude [Wile, 2004].

1.1.3 Modèles de variabilité : Feature Models

Les modèles de variabilité permettent de représenter les différentes configurations dans lesquelles un système peut se trouver. Le formalisme des FM (introduit au Chapitre 1) est le standard de modélisation de la variabilité le plus utilisé aujourd'hui. Les FM ont été introduits par Kang *et al.* [Kang et al., 1990]. Les briques de base des FM sont les *features*. Un FM permet d'établir des relations de coexistence entre ses *features*, *e.g.*, la présence de la *feature* f_1 dans le système implique la présence de la *feature* f_2.

Dans cette section, nous donnons tout d'abord la définition formelle d'un FM (Paragraphe A), puis nous précisons la notion de *feature* (Paragraphe B).

A Définition et notations

Formellement, un FM est constitué d'un diagramme de *features* et d'un ensemble de contraintes exprimées en logique propositionnelle. Nous utilisons la définition formelle d'Acher [Acher, 2011] pour définir un FM.

Définition : Feature Model *Un Feature Model FM est un couple* $\langle FD, \psi_{cst} \rangle$ *où* FD *est un diagramme de features et où* ψ_{cst} *est un ensemble de contraintes, chaque contrainte étant une formule de logique propositionnelle sur l'ensemble des features* $\mathcal{F}$ *(mais ni une contrainte d'implication, ni d'exclusion).*

Définition : diagramme de *features* *Un diagramme de features* $FD = \langle G, r, E_{MAND}, \mathcal{F}_{XOR}, \mathcal{F}_{OR}, Impl, Excl \rangle$ *est défini comme suit :*

- $G = (\mathcal{F}, E)$ *est un arbre où* $\mathcal{F}$ *est un ensemble fini de features et* $E \subseteq \mathcal{F} \times \mathcal{F}$ *est un ensemble fini d'arcs (les arcs représentant une décomposition hiérarchique des features,* i.e., *des relations parent-enfants),*
- $r \in \mathcal{F}$ *est la feature racine,*
- $E_{MAND} \subseteq E$ *est l'ensemble des arcs pour lesquels la feature enfant est obligatoire vis-à-vis de sa feature parente,*
- $\mathcal{F}_{XOR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ *et* $\mathcal{F}_{OR} \subseteq \mathcal{P}(\mathcal{F}) \times \mathcal{F}$ *définissent des groupes de features et sont des ensembles de paires de features enfants avec leur feature parente commune. Ces features enfants sont soit exclusives entre-elles vis-à-vis de leur feature parente (groupe XOR), soit inclusives (groupe OR),*
- *les features qui ne sont ni obligatoires ni membres d'un groupe OR ou XOR sont des features optionnelles,*
- *une feature peut être parente de plusieurs groupes OR ou XOR, mais une feature ne peut appartenir qu'à un seul groupe,*
- *un ensemble de contraintes d'implication Impl (resp. de contraintes d'exclusion Excl), chaque contrainte d'implication (resp. d'exclusion) étant une formule propositionnelle de la forme* $A \Rightarrow B$ *(resp.* $A \Rightarrow \neg B$*)) où* $A \in \mathcal{F}$ *et* $B \in \mathcal{F}$*.*

On note que d'après cette définition, les *features* sont des variables booléennes. Un FM est constitué d'un arbre de *features* exprimant des contraintes de coexistence entre les *features*. L'arbre est complété par des contraintes exprimées en logique propositionnelle.

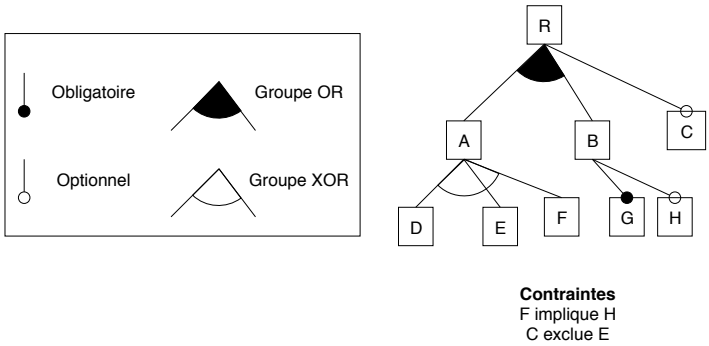


Figure 2.2 – Exemple de FM et légende (d'après la notation de Kang *et al.* [Kang et al., 1990]).

La Figure 2.2 donne un exemple de FM. Les notations utilisées sont celles proposées par Kang *et al.* [Kang et al., 1990]. Dans cet exemple, G et H sont des *features* filles de B. G est obligatoire vis-à-vis de B et H est optionnelle vis-à-vis de B, *i.e.,* si B est sélectionnée, alors G doit également l'être, mais pas nécessairement H. Si B n'est pas sélectionnée, alors G et H non plus. D, E et F appartiennent à un même groupe XOR, donc ces trois *features* sont mutuellement exclusives, mais au moins l'une d'entre-elles est obligatoire si leur *feature* parente, A, est sélectionnée. A et B appartiennent à un même groupe OR fils de R, donc A et B peuvent être sélectionnées mutuellement, ou seulement une des deux, mais il n'est pas possible qu'aucune d'entre elles ne soient sélectionnées car R, la racine, est obligatoire d'office.

Tout FM peut être représenté sous forme d'une formule propositionnelle. Par exemple, le FM de la Figure 2.2 peut être traduit par la formule $\phi = R \wedge \phi_{OR} \wedge \phi_{R,C} \wedge \phi_{XOR} \wedge \phi_{B,G} \wedge \phi_{B,H} \wedge \phi_{impl} \wedge \phi_{excl}$ où :

— $\phi_{OR} = R \Leftrightarrow (A \vee B)$,

— $\phi_{R,C} = C \Rightarrow R$,

— $\phi_{XOR} = (A \Rightarrow (D \vee E \vee F)) \wedge (D \Rightarrow \neg(E \vee F)) \wedge (E \Rightarrow \neg(D \vee F)) \wedge (F \Rightarrow \neg(D \vee E))$,

— $\phi_{B,G} = B \Leftrightarrow G$,

— $\phi_{B,H} = H \Rightarrow B$,

— $\phi_{impl} = F \Rightarrow H$,

— $\phi_{excl} = C \Rightarrow \neg E$.

Définition : configuration. *Une configuration d'un FM est un ensemble de features représentant les features sélectionnées pour une version possible du produit logiciel à synthétiser. La sélection doit respecter les interdépendances et contraintes définies par le FM.*

Par exemple, l'ensemble $\mathcal{C} \subset \mathcal{P}(\mathcal{F})$ défini en Équation (2.1) est l'ensemble des configurations possibles du FM de la Figure 2.2.

$$\mathcal{C} = \{$$
$$\{R, A, D\},$$
$$\{R, A, E\},$$
$$\{R, A, F, B, G, H\},$$
$$\{R, B, G\},$$
$$\{R, B, G, H\},$$
$$\{R, A, B, D, G\},$$
$$\{R, A, B, D, G, H\},$$
$$\{R, A, B, E, G\},$$
$$\{R, A, B, E, G, H\},$$
$$\{R, A, C, D\},$$
$$\{R, A, C, F, B, G, H\},$$
$$\{R, B, C, G\},$$
$$\{R, B, C, G, H\},$$
$$\{R, A, B, C, D, G\},$$
$$\{R, A, B, C, D, G, H\}$$
$$\} \quad (2.1)$$

B Notion de feature

Une *feature* peut s'interpréter de plusieurs manières différentes. Dans cette thèse, nous gardons le mot anglais et ne traduisons pas par le mot "fonctionnalité" car nous considérons que celui-ci ne capture pas les différents sens du mot dans ce contexte. Kang *et al.* ont défini le concept de *feature* comme *"une abstraction fonctionnelle clairement identifiable qui doit être implémentée, testée, livrée et maintenue"* [Kang et al., 1998]. Cette définition indique qu'une *feature* peut en effet être une fonctionnalité d'un logiciel, mais elle peut être aussi un ensemble de données, un composant, un processus, une relation, un ensemble de classes, *etc.* Plusieurs autres définitions ont été proposées [Classen et al., 2008, Apel and Kästner, 2009]. La définition d'Apel et Kästner la décrit comme *"une structure qui étend et modifie la structure d'un programme donné dans le but de satisfaire un besoin, d'implémenter et contenir un choix de design et d'offrir une option de configuration"* [Apel and Kästner, 2009]. Nous choisirons cette définition pour notre thèse. Cette définition a en effet l'avantage d'être précise et adaptée à plusieurs cas d'utilisation. D'après cette définition, une *feature* est le pont entre des besoins, identifiés lors d'une analyse de besoins retranscris par exemple dans un modèle de domaine, et des modèles plus proche de l'implémentation, par exemple un diagramme de classes. Une *feature* peut donc par exemple correspondre à un concept ontologique correspondant à une classe, une activité d'un diagramme d'activité, une ligne de vie d'un diagramme de séquence, *etc.* Plus généralement, un FM est d'après cette définition la projection d'un modèle de domaine [Czarnecki et al., 2006].

1.1.4 Modéliser la variabilité de modèles séquentiels

Ziadi *et al.* se sont intéressés à la gestion de la variabilité de modèles séquentiels par l'utilisation de FM [Ziadi et al., 2002]. Les modèles séquentiels sont par exemple des diagrammes de séquences ou des diagrammes d'activité UML. Chaque *feature* du FM représente une portion du diagramme. Le FM permet alors d'ajouter ou de supprimer des étapes du diagramme, ou encore de substituer des étapes par d'autres.

1.2 Modélisation de domaines

Cette sous-section présente les principales approches pour modéliser un domaine. La production de modèles utilisables en IDM passe par une phase d'analyse du domaine à modéliser. Nous présentons donc tout d'abord les principes de l'analyse de domaine (Section 1.2.1), puis le cadre formel de la modélisation est expliqué (Section 1.2.2). Enfin deux approches de modélisation sont présentées : la modélisation en monde fermé (Section 1.2.3) et la modélisation en monde ouvert (Section 1.2.4).

1.2.1 Analyse du domaine

Tout domaine définit des concepts, des normes, qui sont un moyen d'abstraire une partie de la réalité relative à ce domaine. Ces abstractions sont des modèles. Tout domaine est amené à évoluer et donc sa modélisation n'est pas figée. Il existe des domaines pour lesquels la modélisation est rigoureuse et stable (*e.g.,* conception d'avions, maintenance d'équipement militaire). Pour d'autres domaines, la modélisation est parfois implicite et sujette à de nombreuses variations, guidée par des heuristiques. Pour ces domaines, l'absence de modélisation formelle n'empêche pas la rigueur. Par exemple, la conception d'évaluations expérimentales est fondée sur des principes et des heuristiques auxquelles ont contribué plusieurs acteurs non nécessairement contemporains entre eux. Les travaux de ces acteurs constituent des modèles, même s'ils ne sont pas nécessairement formalisés. La pratique de la modélisation en IDM amène cependant à progressivement généraliser la production de modèles formels. Par exemple, Barišić *et al.* ont proposé un modèle pour la conception et la conduite d'évaluations expérimentales en génie logiciel [Barišic et al., 2014].

1.2.2 Modèles et métamodèles : cadre formel de la modélisation

La Figure 2.3 schématise le cadre formel de la modélisation. Un modèle est une représentation abstraite de concepts issus du monde réel. En IDM, un modèle est formellement décrit par son métamodèle, *i.e.,* le modèle du modèle. Un exemple est celui d'UML : le diagramme de classes est un modèle qui permet de décrire les classes qui structurent un logiciel. Par exemple, un logiciel de gestion des ressources humaines au sein d'une entreprise comporterait entre autres une classe *Employé*. Le métamodèle d'UML décrit quant à lui les concepts que l'on peut utiliser dans un diagramme de classes, entre autres : les concepts de classe, interface, package, relation, *etc.*[1] Un métamodèle est décrit par un méta-métamodèle. Un méta-métamodèle doit être

1. La spécification complète d'UML peut être consultée sur le site de l'*Object Management Group* (OMG) : https://www.omg.org/spec/UML/About-UML/

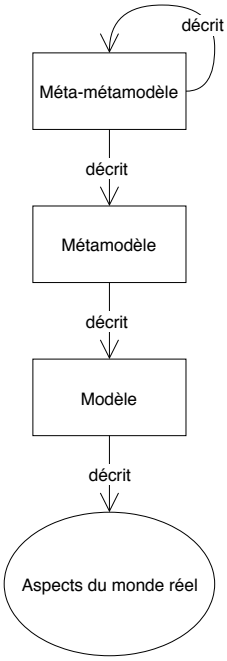


Figure 2.3 – Cadre formel de la modélisation.

réflexif, *i.e.,* se décrire lui-même, pour éviter de devoir définir une infinité de méta-métamodèles. La métamodélisation se fait en général avec des langages proches d'UML, *e.g.,* Ecore [Steinberg et al., 2008].

1.2.3 Modélisation en monde fermé

La modélisation en monde fermé est celle qui est la plus couramment utilisée en IDM. UML et les langages informatiques en général fonctionnent selon ce principe. En modèle "fermé" toute relation existante entre deux concepts doit être explicitement représentée pour être considérée comme vraie. Si un cas d'usage n'est pas conforme au modèle, il faut modifier le modèle pour pouvoir prendre en compte ce cas d'usage. Par exemple en UML, s'il n'existe pas de relation d'héritage entre une classe A et une classe B, alors il est faux de dire que A hérite de B. Autrement dit les mondes fermés fonctionnent selon une logique binaire. Cela est contraignant lorsque le nombre de concepts dans un domaine est important et qu'expliciter toutes les relations qui peuvent exister n'est pas fondamental pour le domaine en question.

Notons que ce paradigme n'empêche pas de calculer ou d'inférer des relations qui ne sont pas explicites en premier lieu. En effet il est possible que certaines relations ne puissent être définies sans connaître le contexte précis d'utilisation d'un modèle. Le problème de l'interopérabilité de plusieurs DSL en est une illustration : lorsque l'on souhaite utiliser plusieurs DSL de manière conjointe, il peut être intéressant d'établir des relations entre des concepts issus de DSL différents. Degueule *et al.* ont proposé une approche permettant de calculer de telles relations [Degueule et al., 2016].

1.2.4 Modélisation en monde ouvert : ontologies

A l'inverse de la modélisation en monde fermé, la modélisation en monde ouvert n'oblige pas à être exhaustif. Le paradigme des mondes ouverts permet ainsi de ne pas préciser toutes les relations existantes entre deux concepts. L'absence de relation d'héritage explicite entre une classe A et une classe B veut alors simplement dire que la filiation entre A et B est inconnue. Ce paradigme permet de prendre en compte des cas d'usages non prévus par la modélisation. Les ontologies permettent ce type de modélisation. Gruber définit une ontologie comme étant "la spécification d'une conceptualisation", une conceptualisation étant "une vue abstraite et simplifiée du monde que l'on veut représenter" [Gruber, 1995].

Les ontologies permettent de définir des concepts et des relations, tout comme les modèles de type UML. Elles permettent en plus de définir des axiomes, qui permettent d'exprimer des relations plus complexes, *e.g.,* "La classe A hérite de toutes les classes ayant un attribut de type B". L'outil le plus connu permettant de décrire des ontologies est Protégé [2], développé par l'université de Standford et fondé sur le format OWL [3].

1.3 Langages dédiés

Cette section présente un concept de génie logiciel de plus en plus utilisé : le concept de DSL [van Deursen et al., 2000, Fowler, 2010, Mernik et al., 2005]. Tout d'abord nous proposons une définition de ce concept ainsi que ses principaux intérêts (Section 1.3.1), notamment les intérêts par rapport aux GPL. Ensuite nous présenterons les différentes étapes du cycle de production d'un DSL : analyse du domaine (Section 1.3.2) ; conception et développement (Section 1.3.3) ; évaluation (Section 1.3.4) [Bezerra and da Silva Barreto, 2014, Giraldo et al., 2016].

1.3.1 Définition et intérêts

A Définition

Mernik *et al.* définissent les DSL ainsi : *"langages conçus sur mesure pour un domaine d'application particulier. Ils offrent des gains significatifs en expressivité et en facilité d'utilisation en comparaison avec des langages de programmations généralistes [(GPL)] dans leur domaine d'application"* [Mernik et al., 2005].

Il existe des DSL pour tout type d'usages : modélisation, programmation, expression de requêtes, formatage, *etc.* Ils peuvent se présenter sous plusieurs formes différentes :

— externe ou interne : un DSL peut être défini comme un langage à-part entière (externe, *e.g.,* SQL) ou sous forme d'un langage utilisable dans un langage hôte (interne), *e.g.,* sous la forme d'une API. Certains langages permettent même de définir des DSL internes en leur conférant leur propre syntaxe, *e.g.,* Scala [4]. Un exemple de DSL interne est RSpec [5], un langage interne à Ruby [6] qui permet de

2. https://protege.stanford.edu/
3. https://www.w3.org/OWL/
4. https://www.scala-lang.org/
5. http://rspec.info/
6. https://www.ruby-lang.org/fr/

Table 2.1 – Comparaison des DSL et des GPL. Ces différences sont générales et la frontière entre DSL et GPL, au regard de ses critères, n'est pas toujours aussi nette.

Critère	DSL	GPL
Utilisateurs cibles	experts d'un domaine spécifique	développeurs
Taille du langage	petite	grande
Communauté d'utilisateurs	petite et localisée	grande et éparpillée
Turing-complet	pas tout le temps	quasiment toujours
Durée de vie	quelques années au plus	plusieurs dizaines d'années

tester des programmes écrits avec ce langage hôte.

— textuel (*e.g.,* SQL) ou graphique (*e.g.,* DSL3S [de Sousa and da Silva, 2018]).

B Intérêts

Les DSL constituent un pont entre l'espace des problèmes et l'espace des solutions (voir Figure 2.1). Les mots-clés, les notations et les syntaxes utilisées pour définir les DSL doivent être facilement utilisables par les experts du domaine visé. Les DSL doivent de même rester petits, simples et conçus pour être adaptés au mode de fonctionnement des organisations utilisatrices [Wile, 2004]. En respectant ces critères, les DSL permettent de tirer parti à la fois de l'expertise des concepteurs (*i.e.,* compétences en IDM) et des utilisateurs (*i.e.,* domaine visé) [Gabbard et al., 1999, Julier et al., 2001, Huang et al., 2012].

C DSL vs GPL

Les DSL sont conçus pour adresser un domaine particulier alors que les GPL sont conçus pour des développeurs à des fins généralistes. Ces différences d'objectif se traduisent par des différences en termes de taille, d'utilisateurs cibles et de durée de vie notamment. Le Tableau 2.1 détaille ces différences.

Des études ont été menées pour montrer le réel intérêt des DSL vis-à-vis des GPL. A titre d'exemple, Johanson *et al.* ont récemment mené une étude empirique pour l'étude des avantages potentiels que l'utilisation d'un DSL pourrait avoir en comparaison avec l'utilisation d'un GPL [Johanson and Hasselbring, 2017]. Le domaine était l'étude d'écosystèmes marins et le but des développements était la simulation de tels écosystèmes. Les experts du domaine, non formés en développement informatique, ont utilisé un DSL et le langage C++ afin de réaliser des développements spécifiques. L'efficacité des développements dans les deux cas ont étés mesurés et comparés. Les résultats montrent des avantages significatifs du DSL sur C++ : les temps de développement sont moindres et la proportion de programmes correctes vis-à-vis des tâches imposées est plus grande.

1.3.2 Analyse du domaine

L'étape d'analyse du domaine du DSL suit les principes présentés en Section 1.2.1. Il s'agit de la première étape nécessaire au développement d'un DSL. L'analyse de domaine aboutit en effet à la production de métamodèles (voir

Section 1.2.2) [Van Deursen and Klint, 2002], qui sont à la base de la conception des DSL.

Si la conception des métamodèles à partir de l'analyse peut être faite à la main, des approches ont été proposées pour automatiser cette tâche. Par exemple, des approches ont été proposées pour transformer des modèles de cas d'utilisations en modèles de domaine [Yue et al., 2009]. Les modèles de cas d'utilisation permettent en effet de représenter les spécifications fonctionnelles [Jacobson, 1987]. Une autre approche a été proposée pour analyser les domaines en les représentant sous forme de cartes conceptuelles (*mind-maps* en anglais) [Pescador and de Lara, 2016]. Un outil permet alors de générer automatiquement un métamodèle à partir d'une carte conceptuelle.

1.3.3 Conception et développement

L'étape de conception consiste essentiellement à produire le métamodèle, à définir la grammaire du DSL et à les mettre en correspondance. Il faut ainsi associer à chaque classe du métamodèle l'élément syntaxique (ou graphique) qui le représente. L'intérêt du métamodèle est de représenter la structure du DSL de manière formelle. Il sert de base commune aux différents outils à développer qui permettront d'utiliser le DSL : un éditeur, un compilateur, un module d'auto-complétion de code, *etc.* On peut aussi définir des grammaires et des compilateurs différents à partir d'un même métamodèle tout en étant capable de conserver une équivalence entre les variantes de ce DSL. A titre d'exemple, le DSL *Robot* (présenté en détail en annexe B), est un DSL dont le métamodèle et la grammaire sont respectivement représentés en Figure 2.4 et en extrait de code 2.1. Un exemple de modèle *Robot* est présenté en extrait de code 2.2.

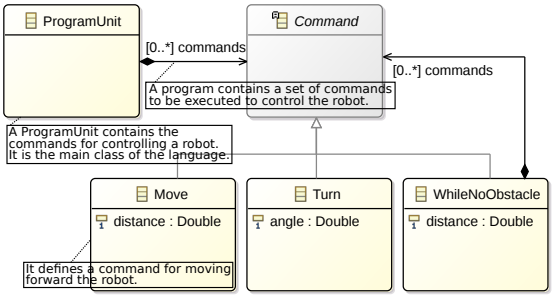


Figure 2.4 – Le métamodèle du DSL *Robot*

Listing 2.1 – La grammaire du DSL *Robot*

```
ProgramUnit: 'begin' (commands+=Command)* 'end';
Command: Move | Turn | WhileNoObstacle;
Move: 'move' '(' distance=Double ')';
Turn: 'turn' '(' angle=Double ')';
WhileNoObstacle: 'whileNoObstacleAt' '(' distance=Double
                 ')' '{' (commands+=Command)* '}';
```

Listing 2.2 – Un programme *Robot*

```
begin
  move(25)
  whileNoObstacleAt(10) {
    move(75)
    turn(90)
    move(50)
  }
  turn(-100)
  move(60)
end
```

Les classes du métamodèle sont associées de manière triviale aux règles de la grammaire, par le nom de la classe.

Il existe des outils de conception de DSL, *e.g.,* Xtext [Bettini, 2013]. Deux approches de conceptions existent : (1) concevoir le métamodèle puis générer la grammaire à partir de celui-ci ou (2) concevoir la grammaire puis générer le métamodèle. L'intérêt de ses approches est de réduire la quantité de travail à fournir pour concevoir le DSL. Aucune d'entre elles n'est nécessairement préférable à l'autre car les deux ont des limites :

— générer la grammaire à partir du métamodèle : cette approche nécessite en général de retoucher la grammaire pour la personnaliser par rapport au métamodèle ;

— générer le métamodèle à partir de la grammaire : cette approche génère un métamodèle très fidèle à la grammaire, mais celui-ci peut y être trop spécifique et donc pas utilisable pour définir d'autres grammaires.

Le développement d'un DSL comprend essentiellement le développement des outils qui permettront d'utiliser le DSL (éditeur, compilateur, *etc.*). Il existe des outils permettant de faciliter le développement de DSL, ainsi que le développement avec ces DSL (*e.g.,* Visual Studio DSL tools [7], Xtext [Bettini, 2013]). Ces outils fournissent des éditeurs permettant de définir le métamodèle et la grammaire du DSL et à partir de cette conception de générer un éditeur pour utiliser le DSL, ainsi qu'un compilateur. Cette génération peut être semi-automatique (*i.e.,* le concepteur intervient dans le développement du compilateur) ou complètement automatique.

1.3.4 Évaluation

Le but de tout DSL étant d'être facilement utilisable par la communauté d'utilisateurs ciblés, il est nécessaire d'évaluer l'utilisabilité de tout DSL nouvellement développé. Des études ont été menés récemment afin de déterminer quelles métriques devraient être utilisées, et comment elles devraient être évaluées.

Albuquerque *et al.* ont par exemple proposé une approche permettant de comparer l'utilisabilité de DSL dans le cadre de leur maintenance [Albuquerque et al., 2015]. Leur approche repose sur le *framework* CDN (*Cognitive Dimensions of Notations*) [Blackwell et al., 2001], qui définit plusieurs critères cognitifs (*e.g.,* cohérence, confusion potentiellement induite) pour évaluer les systèmes de notation, comme les langages. Les auteurs ont identifié deux critère principaux à prendre en compte pour l'évaluation des DSL : l'expressivité et la concision. Les résultats de leur évaluation montrent que ces

7. https://msdn.microsoft.com/en-us/library/ee943825.aspx

critères sont déterminants lors de l'évaluation de DSL, particulièrement au début de leur développement, lorsque des changements de conception peuvent encore être faits.

Barišić *et al.* ont en effet remarqué qu'évaluer un DSL uniquement à la fin de son développement n'encourage pas la prise en compte des résultats de l'évaluation, des changements pouvant être coûteux ou impossibles étant donnés des questions de délais de livraison [Barišic et al., 2014]. Les auteurs ont proposé un modèle formel permettant d'évaluer les DSL à plusieurs étapes de leur développement. Plus récemment, Barišić *et al.* ont proposé le framework USE-ME, permettant de modéliser formellement le contexte, les objectifs et le protocole d'évaluation d'un DSL [Barišić et al., 2018]. D'autres frameworks et métriques d'évaluation ont récemment étés proposées [Challenger et al., 2016, Kahraman and Bilgen, 2015].

Les approches proposées jusqu'à maintenant doivent encore être améliorées. Giraldo *et al.* ont en effet récemment présenté une étude analysant les méthodes d'évaluation de DSL (et plus généralement d'outils de l'IDM) proposées dans les articles académiques et industriels [Giraldo et al., 2016]. Cette étude montre principalement que ces méthodes ne tiennent pas assez compte des contraintes organisationnelles identifiées par Wile [Wile, 2004]. L'évaluation des DSL est donc encore un champ d'étude largement ouvert.

La Section 1 nous a permis de faire un état de l'art des méthodes existant aujourd'hui en IDM pour produire des logiciels à l'aide de procédés d'abstraction, de réutilisation et de SoC. La Section 2 présente un état de l'art de la RV. Cela nous permettra d'identifier les concepts de la RV qui peuvent faire l'objet d'abstractions et de réutilisation, ainsi que la manière dont ils peuvent être traités séparément dans le cadre de la SoC.

2 Structure des applications de RV

La Figure 2.5 présente un schéma de principe résumant les principales caractéristiques des applications de RV. Les applications de RV permettent à des utilisateurs humains d'interagir dans un Environnement Virtuel (EV) avec des objets et des humains virtuels, potentiellement de manière collaborative (voir Section 2.1). Les séquences d'actions réalisées par les acteurs (*i.e.,* humains réels et virtuels) s'appellent des scénarios (voir Section 2.2).

Un exemple d'applications de RV illustrant bien ces concepts sont les applications de RV pour la formation. L'utilisation d'outils virtuels permet en effet de s'entraîner dans des conditions paramétrables avant de se confronter à des situations réelles, qui peuvent être trop risquées ou trop complexes pour un apprenant (*i.e.,* utilisateur de l'application de RV qui a pour rôle de se former) [Arnaldi et al., 2006]. Les opérations de maintenance de certaines pièces mécaniques peuvent par exemple s'avérer dangereuses pour une personne inexpérimentée. Dans certaines situations d'apprentissage, le matériel n'est pas toujours disponible ou est très coûteux, il est donc pratique d'avoir à disposition du matériel virtuel réservé à l'entraînement, comme des simulateurs de conduite de voitures ou d'avions [Arnaldi et al., 2006]. Les applications pour la formation peuvent même prendre la forme de jeux sérieux (*Serious Games* en anglais) pour tirer parti

d'une dimension divertissante à des fins éducatives [Muratet et al., 2009]. Dans les applications de RV pour la formation on peut spécifier les scénarios attendus, *i.e.*, les séquences d'actions qui permettent d'atteindre un but défini répondant à l'objectif de la formation.

La conception d'applications de RV performantes et facilement utilisables nécessite le recours à des évaluations expérimentales. Elles permettent d'étudier les effets d'un système, d'une interface, d'un algorithme, *etc.* sur les utilisateurs du système. En Section 2.3, nous exposons les approches permettant d'évaluer les applications de RV.

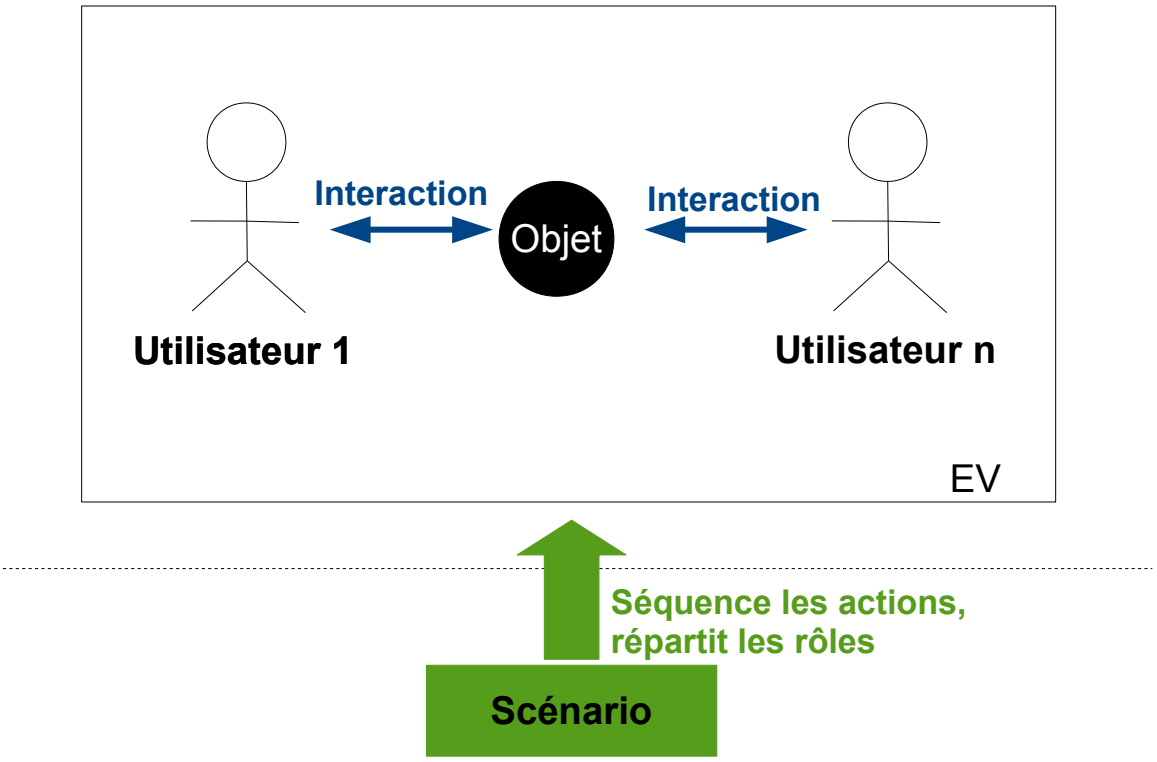


Figure 2.5 – Structure des applications de RV.

2.1 EV interactifs

Cette sous-section a pour but de donner les grands principes qui guident la mise en place d'objets virtuels avec lesquels il est possible d'interagir. La Section 2.1.1 définit la collaboration et présente des mécanismes qui la rendent possible. La Section 2.1.2 présente le concept de modèles objets-relations, qui permet la définition d'interactions. La Section 2.1.3 présente les techniques d'interactions conformes à ces modèles et permettant la manipulation d'objets virtuels.

2.1.1 Définition et gestion de la collaboration en EV

Les applications de RV doivent permettre la collaboration d'utilisateurs physiquement présents au même endroit, ou bien répartis sur des sites distants, et

même avec des humains virtuels. On distingue alors trois niveaux de collaboration [Margery et al., 1999] :

1. les utilisateurs sont conscients les uns des autres (e.g grâce à une représentation sous forme d'avatars) et peuvent communiquer entre eux. C'est un niveau de base qui peut être suffisant pour des applications de téléconférence comme MASSIVE [Greenhalgh and Benford, 1995] ou des réseaux sociaux par exemple ;
2. les utilisateurs peuvent en plus interagir avec des objets de l'environnement de manière individuelle. Il est par exemple possible à un utilisateur d'attraper et de déplacer un objet ou de changer sa couleur. la collaboration se fait à ce niveau encore par la communication entre les utilisateurs, mais elle n'a pas une vocation uniquement sociale. Les utilisateurs peuvent en effet communiquer pour se mettre d'accord sur des transformations à opérer sur l'environnement, en vue d'atteindre un but donné. C'est le cas de Calvin [Leigh and Johnson, 1996], une application de conception architecturale permettant à plusieurs utilisateurs de s'immerger dans un même environnement afin de modifier et de visualiser en direct l'apparence d'une scène d'intérieur ;
3. les interactions avec les objets de l'environnement peuvent être collaboratives. Il est par exemple possible de manipuler un objet à plusieurs. On distingue en général deux sous-niveaux : (3.1) seules des interventions indépendantes sont permises en simultané sur un même objet (*e.g.,* déplacement et changement de la couleur) ou (3.2) le système gère les interventions interdépendantes (*e.g.,* co-manipulation d'un objet, où la position de l'objet dépend des efforts qui lui son appliqués par plusieurs utilisateurs). La Figure 2.6 illustre ce dernier niveau de collaboration.

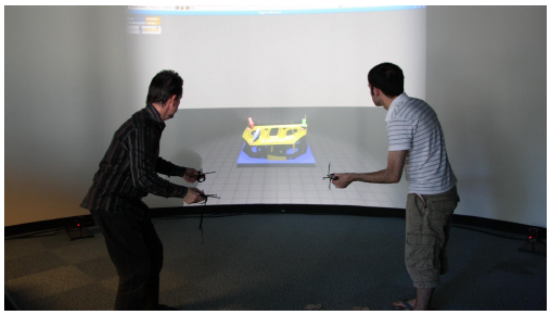


Figure 2.6 – Niveau 3.2 de la collaboration : deux opérateurs déplacent un objet de manière collaborative [Aguerreche et al., 2009]. Le mouvement résultant prend en compte les deux interactions simultanées.

La gestion de la collaboration ne peut pas se passer de mécanismes de synchronisation. En effet, la transmission des événements sur un réseau peut conduire à des ordres différents de réception et donc à des incohérences entre les états de chaque processus. Selon le niveau de collaboration, les mécanismes de synchronisation ne sont pas les mêmes. Au premier niveau, qui ne permet que la communication entre les utilisateurs, il n'y a pas besoin de mécanismes particuliers. Au deuxième niveau, permettant des interactions individuelles avec les objets, il faut munir ces derniers de verrous qui empêchent l'intervention de plus d'une personne. Le troisième niveau,

permettant des interactions simultanées sur un même objet, a été traité par Margery *et al.* [Margery et al., 1999]. Il faut garantir deux propriétés :

1. tous les processus doivent s'accorder sur l'ordre d'exécution des opérations sur un même objet ;
2. tous les processus doivent s'accorder sur la simultanéité des opérations sur un même objet, c'est à dire que si un processus considère que deux opérations ont eu lieu simultanément, alors tous les processus doivent considérer la même chose. Cette propriété est utile pour le niveau 3.2, qui permet les interactions simultanées interdépendantes.

Avant tout, il faut définir ce qu'est une opération sur un objet. Margery *et al.* la définissent comme étant le changement de valeur d'un *handler*[8] [Margery et al., 1999]. Ainsi les opérations peuvent être associées à des événements ordonnés dans le temps.

La première propriété est garantie à l'aide d'une contrainte qui doit être imposée sur les messages échangés sur le réseau[9] et dont se charge l'objet concerné : il faut que les messages soient diffusés en ordre complet, c'est à dire que tous les processus les reçoivent dans le même ordre.

La deuxième propriété s'obtient grâce à un mécanisme d'inscription / désinscription au moment où un utilisateur commence à interagir avec un objet (resp. termine son interaction). A chaque exécution d'opération, l'objet considère que tous les utilisateurs inscrits sont en activité simultanée et opère d'abord la diffusion d'un message contenant le nombre d'inscrits. Ensuite, à chaque opération effectuée, la valeur du *handler* correspondant est à son tour diffusée. Chaque processus attend donc le nombre d'opérations correspondant au nombre reçu au préalable, avant de calculer la transformation résultante. Si un utilisateur reste trop longtemps "détenteur" d'un *handler* sans pour autant réaliser d'interaction, la valeur du *handler* est considérée constante et un signal spécial est diffusé afin de ne pas faire attendre inutilement les processus.

2.1.2 Les modèles objets-relations

Les modèles objets-relations permettent de définir les interactions comme des réalisations de relations définies entre des objets (dont certains sont des acteurs). Par exemple, on peut définir la relation *Taper* qui permet de lier un acteur, un marteau et un clou. La réalisation de cette relation correspond à l'action de l'acteur qui tape le clou avec le marteau afin de l'enfoncer. Une réalisation modifie les caractéristiques d'un ou plusieurs objet. En général, les réalisations ont un but précis, *i.e.,* les changements provoqués sur les objets font changer l'état de ceux-ci, *e.g.,* pour provoquer un événement dans l'application de RV et faire avancer la simulation (*i.e.,* faire avancer le scénario, voir Section 2.2). Il peut exister des préconditions pour une réalisation, *e.g.,* dans le cas de la relation *Taper*, le clou ne doit pas être déjà enfoncé. Des exemples de *frameworks* implémentant cette

8. Entité logicielle qui permet de détecter une requête d'interaction et qui en gère les paramètres. Un *handler* peut être matérialisé dans l'application de RV par une boule de couleur par exemple (voir la Figure 2.6).

9. Un message représente alors une opération.

approche sont STORM [Mollet et al., 2007], VEHA [Chevaillier et al., 2012], Domain-DL [Carpentier, 2015] et #FIVE [Bouville et al., 2015].

Ces modèles possèdent trois avantages majeurs :

— **collaboration :** la modélisation d'interactions collaboratives au niveau 3.2 est rendue possible par la définition de relations impliquant plusieurs acteurs,

— **utilisation de plusieurs objets :** il est facile de définir des interactions liant plusieurs objets (*e.g.,* la relation *Taper* nécessite un marteau et un clou),

— **généricité :** les relations sont en général définies sur des types d'objets plutôt que sur des objets individuels (*e.g.,* tout objet ayant la capacité de taper peut être utilisé dans la relation *Taper*, *e.g.,* un marteau ou une batte).

Les modèles objets-relations sont une évolution d'autres modèles d'interactions qui ont été proposés, notamment celui des objets synoptiques [Badawi and Donikian, 2004, Willans and Harrison, 2001] pour lequel les objets ne sont pas caractérisés par des types, mais par les actions que l'on peut réaliser en les utilisant, en fonction de leur état interne. Par exemple un objet "interrupteur" porterait les actions "allumer la lumière" et "éteindre la lumière", disponibles en fonction de l'état de la lumière ("éteinte" ou "allumée"). Cette approche est adaptée aux interactions impliquant un acteur et un objet. Mais pour les interactions plus complexes, il faut lister explicitement tous les objets qui peuvent entrer en jeu. Par exemple, si on attache une action *Taper* à un objet, il faut lister tous les objets de l'environnement permettant de réaliser cette action, *e.g.,* tous les marteaux et toutes les battes. De plus, Même s'il est possible de définir des interactions collaboratives au niveau 3.2 [Willans and Harrison, 2001], les implémentations existantes de ces approches ne le permettent pas toutes [Badawi and Donikian, 2004].

2.1.3 Techniques d'interaction

Les modèles qui ont étés présentés dans cette section doivent être mis en œuvre, ce qui implique de développer des techniques d'interaction et des métaphores. Foley *et al.* définissent une technique d'interaction comme la manière d'utiliser un dispositif d'interaction pour accomplir une tâche sur un ordinateur [Foley et al., 1996]. Plus récemment, Hachet la définissait comme une méthode permettant d'exécuter une interaction en EV [Hachet, 2003]. Plus précisément, une technique d'interaction implique d'utiliser des interfaces réelles, virtuelles ou hybrides (*i.e.,* mêlant composants réels et virtuels). Une métaphore est une représentation symbolique, souvent visuelle, dont l'objectif est de transmettre une information aux utilisateurs d'un EV (*e.g.,* mettre en surbrillance des objets pour inciter les utilisateurs à les manipuler) [Arnaldi et al., 2006].

Bowman a classé les techniques d'interaction en cinq catégories : la navigation (*i.e.,* le déplacement en EV), la sélection, la manipulation, le contrôle (*i.e.,* modifier la configuration ou l'état d'un EV, *e.g.,* par l'utilisation d'un menu) et l'entrée de symboles (*e.g.,* chiffres et lettres) [Bowman et al., 2004]. Pour chaque catégorie de tâches, différentes techniques d'interaction et métaphores ont été proposées. En fonction de leur catégorie, ces techniques d'interaction et métaphores sont évaluées selon certains critères, comme la facilité d'utilisation, la possibilité de collaborer, l'*awareness* (*i.e.,* le

fait d'avoir conscience de ce qui se passe, de la nature des interactions, de pouvoir suivre leur déroulement), *etc.*

Par exemple, les tâches de sélection et de manipulation peuvent être mises en œuvre par des techniques de lancer de rayon [Bolt, 1980]. Ces techniques se fondent sur l'utilisation d'une manette pour pointer les objets à manipuler et d'un laser virtuel sortant de la manette pour permettre de pointer des objets virtuels. Ces méthodes ont l'avantage de permettre le déplacement d'objets distants et d'être simples, mais rendent difficiles l'interaction avec de petits objets, les mouvement de rotations et les interactions collaboratives [Pinho et al., 2002]. Pour ce qui est de la manipulation, il existe d'autres techniques reposants elles sur des points de contact entre les acteurs et les objets à manipuler [Nguyen et al., 2014, Aguerreche et al., 2009, Louvet and Fleury, 2016]. Ces techniques facilitent la mise en place d'interactions collaboratives.

2.2 Modélisation de scénarios

En EV, les utilisateurs ont en général un but à atteindre qui nécessite l'exécution de plusieurs tâches successives. Cependant, tout ne doit pas nécessairement s'opérer de manière linéaire. L'ensemble des utilisateurs peut se scinder en plusieurs groupes qui réaliseront des tâches en parallèle et des choix peuvent être offerts entre plusieurs tâches possibles... ce qui génère des embranchements complexes. Cette gestion séquentielle des tâches est à l'origine du concept de scénario en RV.

Dans la littérature, il existe une confusion autour de la définition d'un scénario. La question est principalement de savoir s'il s'agit d'une séquence d'actions réalisées pendant une session bien précise ou s'il s'agit de l'ensemble des séquences possibles. Pour clarifier cette notion, nous définissons les concepts de scénario, spécification de scénarios et modèle de scénarios en Section 2.2.1. Nous présentons ensuite les deux grandes classes de modèles de scénarios qui existent dans la littérature : les modèles à scénarios prédéfinis (Section 2.2.2) et les modèles à scénarios émergents (Section 2.2.3). Nous concluons cette sous-section par une synthèse présentant les avantages et inconvénients des deux approches et motivant notre choix pour cette thèse (Section 2.2.4).

2.2.1 Définitions

Nous utiliserons les définitions de Claude pour clarifier la notion de scénario [Claude, 2016]. Les définitions sont illustrées en Figure 2.7 :

— **scénario** : *agencement temporel et causal des actions dans un EV lors d'une session d'utilisation du système de RV*,

— **modèle de scénario** : *formalisme et organisation de données permettant de décrire des scénarios ainsi que les mécanismes qui permettent leurs utilisations par les composants du système de RV*,

— **spécification des scénarios** : *description d'un ensemble de scénarios possibles.*

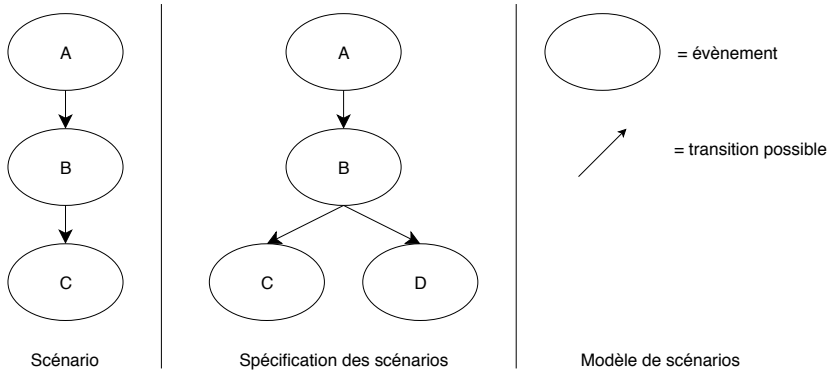


Figure 2.7 – Le modèle de scénario définit comment se structure un scénario et permet de produire des spécifications de scénarios, un scénario étant une exécution possible d'une spécification (crédit : [Claude, 2016]).

2.2.2 Modèles à scénarios prédéfinis

Les modèles à scénarios prédéfinis correspondent à la vision selon laquelle un scénario est une séquence de buts à atteindre (atteindre un but permet de passer au but suivant). Dans cette approche, le cycle de vie d'une tâche est constitué des étapes successives "définition, exécution, conséquence". Pour illustrer, considérons un scénario simple constitué de deux tâches : un utilisateur doit placer une pièce sur un support en la déplaçant à l'aide d'un bras mécanique dont le déplacement se fait à l'aide d'une télécommande munie des boutons *haut*, *bas*, *gauche*, *droite*, *avant* et *arrière*. Le moteur de scénario prépare d'abord les commandes pour la première tâche (définition) : des *handlers* de manipulation (pour chacune des commandes directionnelles) sont mis en place pour commander le bras mécanique. Puis l'opérateur exécute les commandes unes à unes afin de placer la pièce sur le support (exécution). Une fois le but atteint, le système désactive les *handlers* (conséquence) et prépare les commandes pour la tâche suivante (définition) et ainsi de suite.

Les scénarios prédéfinis sont souvent modélisés sous forme de machines à états hiérarchiques [10] ou de modèles équivalents. Dans une machine à états hiérarchique, un état est soit un état atomique, soit une machine à états hiérarchique. Un état représente alors un état possible de l'application de RV et une transition représente une action dans l'application de RV qui permet d'atteindre l'état suivant. Plus précisément, les transitions peuvent être gardées par des conditions d'activation (*e.g.,* deux objets sont entrés en contact) et provoquer des effets dans l'application de RV (*e.g.,* la réalisation d'une relation). Ce formalisme est utilisé par exemple par les sytèmes HCSM [Cremer et al., 1995], HPTS++ [Lamarche and Donikian, 2002] et LORA++ [Gerbaud et al., 2007].

D'autres formalismes ont été proposés, mais ils sont traduisibles sous forme de machine à état hiérarchique, *e.g.,* les diagrammes d'activité UML avec HAVE [Chevaillier et al., 2012]. Toutefois, le formalisme des réseaux de Petri hiérarchiques saufs utilisé par Claude *et al.* pour #SEVEN [Claude et al., 2014], même s'il est mathématiquement équivalent au formalisme des machines à états hiérarchiques,

10. Les machines à états finis (*Finite State Machines*, FSM) d'UML sont des machines à états hiérarchiques.

est plus pratique pour représenter le parallélisme, *i.e.,* des séquences d'actions exécutées simultanément par plusieurs acteurs. Un réseau de Petri est constitué de places (équivalentes aux états) et de transitions. Dans un réseau de Petri hiérarchique, une place est soit une place atomique, soit un réseau de Petri. L'état courant est représenté par un jeton qui se déplace de places en places. Un réseau sauf est un réseau dans lequel chaque place ne peut contenir au maximum qu'un seul jeton. On peut représenter des situations de parallélisme en définissant un jeton par acteur par exemple.

2.2.3 Modèles à scénarios émergents

Cette approche correspond à la vision selon laquelle un scénario doit "émerger" d'une session d'utilisation d'une application de RV. La spécification correspond alors à un ensemble de règles contraignant les interactions. L'utilisateur est laissé libre de ses mouvements mais le but de la simulation n'est atteint que s'il exécute des actions dans un certain ordre. S'il se trompe d'action, une indication peut lui être envoyée pour le prévenir qu'il s'est trompé [Ponder et al., 2003]. Il existe deux classes de scénarios émergents : (1) les scénarios pour lesquels le choix de la prochaine action à réaliser est laissé à l'utilisateur, qui peut être guidé par des objectifs ou des contraintes contextuelles (*e.g.,* la porte est fermée à clé, il faut trouver la clé pour l'ouvrir) et (2) les scénarios proposant une planification automatique reposant sur des techniques issues de l'intelligence artificielle, comme HSP [Bonet and Geffner, 2001]. Dans la première catégorie on trouve notamment les systèmes Theatrix [Paiva et al., 2001] et IDTension [Szilas, 2003]. Dans la seconde catégorie on trouve notamment EmoEmma [Cavazza et al., 2007] et SELDON [Carpentier, 2015].

2.2.4 Synthèse

L'approche des modèles à scénarios émergents peut être intéressante dans le cadre d'applications de RV pour la formation, car elles laissent les utilisateurs libres de leurs actions. C'est à eux de découvrir la bonne combinaison d'actions à réaliser, ainsi que les effets des mauvaises combinaisons. Par exemple, dans le cadre d'applications de RV pour la formation à la maintenance automobile, de mauvaises séquences d'actions peuvent détériorer le véhicule. Cela constitue un mécanisme intéressant d'apprentissage à partir des erreurs commises.

Cependant, les modèles à scénarios émergents ont l'inconvénient d'être plus difficiles à représenter, notamment à cause de la grande complexité que peuvent avoir les systèmes de contraintes. C'est pour cela que nous privilégierons dans cette thèse les modèles à scénarios prédéfinis. De plus, certains modèles de scénarios prédéfinis (*e.g.,* StoryNet [Cremer et al., 1995] et #SEVEN [Claude et al., 2014]) permettent de définir des tâches "libres", *i.e.,* possibles mais n'influant pas sur le déroulement d'un scénario prédéfini, pour ne pas trop contraindre l'utilisateur lors de l'exécution de ces tâches.

2.3 Évaluations expérimentales en RV

La conception d'applications de RV performantes et facilement utilisables nécessite le recours à des évaluations expérimentales [Moreau et al., 2018]. Elles permettent d'étudier les effets d'un système, d'une interface, d'un algorithme, *etc.* sur les utilisateurs du système. La conception d'expériences nécessite de pouvoir facilement reconfigurer une application afin de tester une autre version d'un composant (*e.g.,* pour comparer deux techniques d'interaction). De plus, les résultats d'une expérience n'étant pas absolus, il peut être intéressant de conserver plusieurs configurations possibles et de les proposer en fonction du contexte d'utilisation. Cette sous-section présente les fondements sur lesquels repose l'expérimentation (Section 2.3.1), les méthodes de conception d'évaluations expérimentales en RV (Section 2.3.2), ainsi que les outils existants permettant de faciliter leur conception (Section 2.3.3).

2.3.1 Fondements de l'expérimentation

Concevoir des expériences est fondé sur des principes à suivre rigoureusement [Field and Hole, 2002]. Selon Colman, une expérience est "*une méthode de recherche fondée sur la manipulation d'une ou plusieurs variables indépendantes et le contrôle de variables extérieures qui pourraient avoir une influence sur la variable dépendante*" [Colman, 2015]. Une variable indépendante étant "*une variable que l'expérimentateur fait varier indépendamment des variables extérieures*" et une variable dépendante étant "*une variable potentiellement sujette à être influencée par une ou plusieurs variables indépendantes*". L'idée principale derrière l'expérimentation est de déterminer les relations causales existantes entre différents évènements ou faits [Colman, 2015], *i.e.,* les effets potentiels que les variables indépendantes ont sur les variables dépendantes. Les variables dépendantes correspondent aux données collectées durant les expériences et peuvent être qualitatives ou quantitatives [Creswell, 2013]. Prouver l'existence d'une relation causale entre les variables indépendantes et dépendantes, *i.e.,* assurer la validité interne, est une nécessité [Shadish et al., 2002]. Généraliser des résultats observés sur des petites populations, *i.e.,* assurer la validité externe, nécessite le recours à des méthodes statistiques [Pearl et al., 2014, Field, 2009, Moreau et al., 2018].

Par exemple, dans le domaine de la RV, Latoschick *et al.* [Latoschik et al., 2016] ont conçu un miroir virtuel consistant en un écran affichant l'image d'un avatar imitant les mouvements du sujet (utilisateur du système). L'avatar pouvait prendre une forme parmi quatre possibles, plus ou moins semblables au sujet : un robot, un pantin, un humain et un humain partageant les traits du sujet. Ils ont conduit une évaluation expérimentale permettant d'étudier comment la nature de l'avatar (variable indépendante) influençait le sentiment de se trouver devant un miroir réel (variable dépendante). La relation causale étudiée a été évaluée qualitativement en recueillant l'avis de chaque sujet.

2.3.2 Conception d'évaluations expérimentales en RV

Ces principes généraux sont applicables à tous les domaines pour lesquelles on conçoit des expériences. En informatique, les domaines liés à la conception centrée sur l'humain nécessitent la conduite d'expériences pour pouvoir valider les approches. Ces conceptions impliquent en effet de concevoir des Interfaces Humain-Machine (IHM) : l'influence que les propriétés d'une IHM peuvent avoir sur les utilisateurs doit être étudiée et validée en fonction de certains critères, *e.g.,* le confort de l'utilisateur, l'efficacité des interactions, la facilité de prise en main, *etc.* Gabbard a proposé des lignes directrices et des méthodes pour la conception de logiciels et d'évaluations expérimentales centrées sur l'humain [Gabbard et al., 1999].

Si la méthode classique consiste à conduire des expériences où une population de participants utilise l'interface à évaluer, d'autres approches ont été proposées. Stanney *et al.* utilisent des heuristiques pour guider et évaluer la conception d'interfaces de RV [Stanney et al., 2003]. Tromp *et al.* ont proposé une méthode d'inspection de l'utilisabilité [Tromp et al., 2003], *i.e.,* une simulation de l'utilisation d'une application pour déterminer les besoins de l'utilisateur.

En conception centrée sur l'humain et plus spécialement en RV, certaines caractéristiques des interfaces sont couramment évalués. Il s'agit souvent d'aspects qualitatifs, qui ne sont pas faciles à évaluer à cause de la subjectivité qu'ils induisent [Field and Hole, 2002]. Les aspects les plus couramment évalués sont la charge cognitive induite par les applications de RV, la présence (*i.e.,* le sentiment d'être physiquement présent dans l'application de RV), l'acceptabilité (*i.e.,* si l'application de RV donne envie d'être utilisée ou non) et la qualité de la collaboration [Moreau et al., 2018]. Des recherches ont été menées pour proposer des questionnaires standards. Ainsi, la charge cognitive peut être évaluée par l'utilisation du questionnaire NASA-TLX [Hart and Staveland, 1988]. Le questionnaire SPAM (*Situation Present Assessment Method*) peut être utilisé pour évaluer la présence [Durso et al., 2004], entre autres méthodes d'évaluation [Schubert et al., 2001, Slater et al., 1994, Witmer and Singer, 1998]. Brooke *et al.* ont proposé un questionnaire pour évaluer l'acceptabilité [Brooke et al., 1996]. L'évaluation de la collaboration a aussi été étudiée : Hornbæk a proposé des métriques fondées sur la communication (*e.g.,* nombre de mots prononcés par personne, nombre de questions posées, nombre d'interruptions, *etc.*) [Hornbæk, 2006]. Meier *et al.* ont complété ces métriques en considérant d'autres aspects de la collaboration (*e.g.,* la coordination des acteurs) [Meier et al., 2007].

2.3.3 Faciliter la conception d'évaluations expérimentales

Implémenter des protocoles expérimentaux qui doivent être intégrés dans des logiciels est une tâche chronophage et complexe qui reste encore aujourd'hui majoritairement faite à la main. Or, il existe des outils permettant de la faciliter. Field *et al.* ont proposé IBM SPSS Statistics [Field, 2009], un outil permettant d'automatiser le production d'analyses statistiques de données. Un autre exemple est le projet R [11].

Il existe aussi des outils permettant de modéliser les conditions expérimentales (*e.g.,*

11. https://www.r-project.org/

variables, populations) et d'enregistrer les résultats de l'expérience, *e.g.,* EDA [12] ou Go-Lab [13], mais ils sont surtout utiles pour d'autres domaines que la RV (principalement la biologie, la physique et la chimie). La RV pourrait tirer parti de certains outils référencés par le NHLRC de Californie (*National Heritage Language Resource Center*) [14], plus adaptés à la conception centrée sur l'humain.

3 Synthèse

Dans cette section, nous revenons sur ce qui a été présenté en Sections 1 et 2 et nous confrontons les deux domaines étudiés (l'IDM et la RV). En Section 3.1 nous présentons les limites des pratiques de développement en RV vis-à-vis des méthodes existantes en IDM. En Section 3.2 nous présentons les limites de l'IDM, et notamment les limites vis-à-vis de son application à la RV. En Section 3.3 nous faisons le bilan des limites identifiées et présentons les contributions apportées par la thèse afin de combler ces limites.

3.1 Limites des pratiques de développement et d'évaluation en RV

3.1.1 Développement

Dans cette sous-section, nous présentons tout d'abord les limites générales observées aussi bien dans les pratiques industrielles que dans les approches proposées en recherche (Paragraphe A). Ensuite, nous nous focalisons sur deux cas d'études : le projet PRESTO [Dragoni et al., 2016] (Paragraphe B) et le projet S3PM [Claude, 2016] (Paragraphe C). Ces projets ont la particularité d'utiliser des approches issues de l'IDM, mais de manière limitée et imparfaite.

A Limites générales

Il existe trois niveaux d'analyse et de modélisation des besoins en RV :

1. analyse informelle : l'étude du domaine est réalisée par les développeurs, en collaboration avec les experts du domaine. Le domaine n'est pas modélisé formellement et l'analyse peut prendre la forme de rapports rédigés à l'occasion par les développeurs ;
2. modélisation métier : les développeurs s'appuient sur des modèles ou documents techniques utilisés couramment par les experts du domaine, *e.g.,* de la documentation technique décrivant les procédure de maintenance de chars [Mollet, 2005]. Ces modèles ont l'avantage d'être précis mais difficilement exploitables par des approches liées à l'IDM (documents non structurés, ou structurés sur la base de standards propres au domaine en question) ;
3. modélisation ontologique (voir Section 1.2.4) : des efforts ont été faits pour modéliser formellement certains domaines sous forme d'ontologies [Dragoni et al., 2016, Claude, 2016].

12. https://www.nc3rs.org.uk/experimental-design-assistant-eda
13. http://www.golabz.eu/apps/experiment-design-tool
14. http://international.ucla.edu/nhlrc/data/software

Les langages objets sont majoritairement utilisés en RV (*e.g.,* C# avec le moteur Unity3D). Ces langages permettent l'abstraction et la réutilisation de concepts (*e.g.,* classes, interfaces), mais ne proposent pas de se centrer sur un domaine métier en particulier. En programmation orientée-objet il n'y a pas de notion de familles de logiciels. Ceci rentre en contradiction avec un principe de l'IDM énoncé par Glass : "*la réutilisation-large-échelle fonctionne mieux au niveau de familles de systèmes, et est donc relative à la notion de domaine*" [Glass, 2001].

Cette notion de familles de logiciels est partiellement couverte par la notion de bibliothèque, une bibliothèque proposant des abstractions de concepts qui peuvent être liés à un domaine métier. Cependant, les bibliothèques ne proposent en général pas de gestion rigoureuse de la variabilité. Les informations liées à la variabilité doivent alors être cherchées dans la documentation.

En RV, on utilise aujourd'hui des bibliothèques liées à des domaines métiers qui proposent des représentations graphiques d'objets et aussi des mécanismes d'interaction et de comportement des objets. Cependant, les modèles objets-relations, abstractions performantes, sont peu utilisés dans ces bibliothèques. Les modèles de scénarios sont encore moins utilisés. De ce fait, il n'existe presque pas aujourd'hui d'approches permettant l'abstraction et la réutilisation à la fois de concepts liés à un domaine métier, d'aspects de scénarisation et de concepts objets-relations. Les seuls contre-exemples sont à notre connaissance le projet PRESTO [Dragoni et al., 2016] (Paragraphe B) et le projet S3PM [Claude, 2016] (Paragraphe C).

B PRESTO

Dragoni *et al.* ont proposé le projet PRESTO, qui met en correspondance une ontologie dédiée aus *Serious Games* pour la sécurité civile et des bibliothèques graphiques [Dragoni et al., 2016]. Il s'agit à notre connaissance du seul travail en RV qui propose ce type de relation entre une ontologie et des éléments graphiques. PRESTO prévoit également une abstraction des aspects comportementaux des objets. Cependant, l'abstraction proposée ne permet pas de représenter des relations et des interactions. PRESTO propose des éditeurs de code et notamment un DSL qui permettent de s'appuyer sur l'ontologie pour décrire des scénarios [Busetta et al., 2017]. Ce DSL est textuel et vise une communauté d'utilisateurs habitués à développer. Enfin, le projet est confidentiel et cela permet peu de démonstration des résultats.

C S3PM

Le projet S3PM a été présenté par Claude [Claude, 2016]. Ce projet vise à produire des applications de RV pour la formation de personnel de blocs-opératoires, *e.g.,* pour apprendre à préparer une table d'opération. Ce projet repose sur l'ontologie OntoSPM [Gibaud et al., 2014]. Les contributeurs du projet proposent un système de génération de spécifications de scénarios #SEVEN [Claude et al., 2015] à partir de vidéos montrant le déroulement d'une opération. Cette génération repose sur l'annotation de ces vidéos grâce au logiciel SurgeTrack [Claude, 2016]. Ainsi, à chaque fois qu'une action spécifique est réalisée sur la vidéo, les instances de l'ontologie correspondantes sont utilisées en guise d'annotation de l'image affichée. La Figure 2.8 illustre l'utilisation de SurgeTrack.

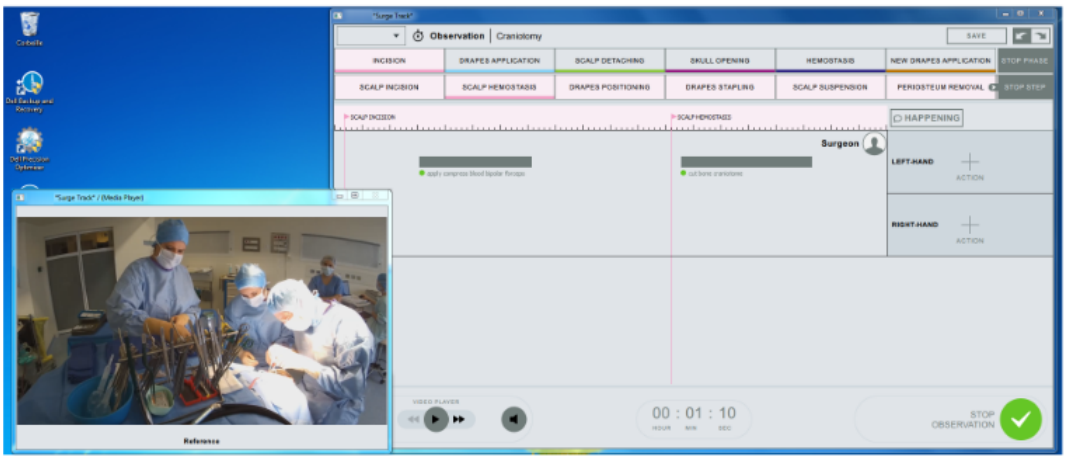


Figure 2.8 – SurgeTrack permet de décrire une procédure en suivant les actions visualisées sur la vidéo, par annotation de celle-ci avec des instances de l'ontologie OntoSPM [Claude, 2016].

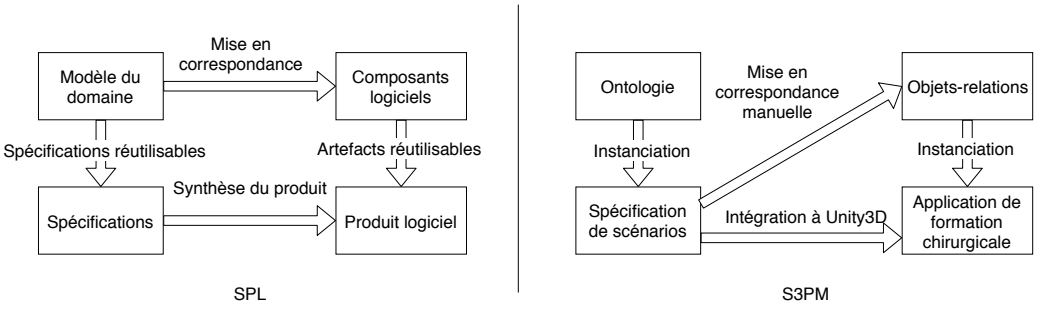


Figure 2.9 – Processus de production d'une application de RV pour la formation dans le cadre de S3PM, en comparaison avec l'approche SPL.

Les instances de l'ontologie mises bout-à-bout sont converties en une spécification de scénarios #SEVEN de façon automatique. Le scénario doit être complété à la main pour mettre les différentes étapes en relation avec l'implémentation. Celle-ci est mise en place grâce à la bibliothèque objets-relations #FIVE. La Figure 2.9 illustre le fonctionnement de S3PM en comparaison avec l'approche SPL. Le passage de l'ontologie vers #SEVEN est correct vis-à-vis de l'approche SPL et correspond à un passage du modèle du domaine (ontologie) à une spécification de scénarios (#SEVEN). La mise en correspondance de #SEVEN et #FIVE n'est en revanche pas satisfaisante d'un point de vue des SPL car dans le cas de S3PM, elle doit être faite pour chaque nouvelle spécification de scénarios générée. La mise en correspondance vers #FIVE devrait être faite vis-à-vis de l'ontologie.

3.1.2 Limites des approches de conception d'évaluations expérimentales en RV

Comme expliqué en Section 2.3, les évaluations expérimentales en RV se définissent par rapport aux conditions expérimentales (variables indépendantes et dépendantes), mais aussi par le protocole expérimental (enchaînement des action d'un sujet confronté à l'application de RV à évaluer). De manière générale, les solutions existantes pour

faciliter la conception d'évaluations expérimentales présentées en Section 2.3.3 se limitent à la modélisation de conditions expérimentales et au traitement statistique des données. Les modèles produits ne sont pas faits pour être traités par des programmes, contrairement aux modèles utilisés en IDM. De plus, la modélisation du protocole expérimental est souvent limitée ou inexistante. Les conséquences sont que les chercheurs finissent pas développer leurs propres solutions *ad hoc* limitées, de manière individuelle.

A notre connaissance, les seuls projets existants à ce jour permettant d'automatiser la conception et le développement d'expériences en RV sont AGENT (que nous avons développé dans le cadre de cette thèse [Le Moulec et al., 2017]) et EVE [Grübel et al., 2016]. Le projet EVE a le mérite d'être très complet et de permettre d'automatiser tous les aspect d'une expérience (définition des variables expérimentales, du protocole, édition des questionnaire et analyses statistiques). Cependant l'implémentation proposée [15] se présente sous la forme d'une bibliothèque pour Unity3D qui n'est pas munie de mécanismes spécifiques d'IDM, comme les SPL ou les DSL par exemple. Elle ne propose pas d'abstraction permettant de modéliser les différents aspects de l'expérience indépendamment de Unity3D. La conception d'une expérience reste "bruitée" par la complexité de Unity3D, ce qui est contraire au principe de SoC. La mise en place d'une expérience est certes facilitée, mais nécessite une prise en main conséquente, imposant de consulter beaucoup de documentation et de bien comprendre l'architecture de la bibliothèque. Il n'y a pas par exemple d'utilisation de DSL qui faciliteraient la prise en main par une syntaxe faite sur mesure et indépendante des éditeurs utilisés en RV.

Les problèmes évoqués dans cette sous-section permettent d'établir une première limite majeure :

L1 : la production d'applications de RV manque d'abstractions, ce qui provoque : (a) un manque de réutilisation et (b) un manque de SoC.

Cette limite peut être comblée en atteignant les objectifs **O1-ABSTRACTION**, **O2-RÉUTILISATION** et **O3-SOC** défini en introduction (Chapitre 1).

3.2 Limites de l'IDM

Dans cette sous-section, nous présentons d'abord les limites propres à l'IDM (Section 3.2.1), puis nous présentons les limites de son application à la RV (Section 3.2.2).

3.2.1 Limites propres à l'IDM

L'utilisation de l'IDM en pratique n'est pas une chose facile [Burden et al., 2014, Whittle et al., 2011, Whittle et al., 2014]. Giraldo *et al.* ont mené une étude pour identifier les principaux problèmes qui empêchent la mise en œuvre des principes de l'IDM dans un contexte industriel [Giraldo et al., 2016]. En plus des insuffisances en

15. https://cog-ethz.github.io/EVE/

matière d'évaluation de la qualité des outils d'IDM développés, les principaux problèmes détectés sont au nombre de quatre :

1. les spécifications de l'IDM sont vieilles et évoluent lentement (*e.g.,* UML).
2. l'adoption de l'IDM soulève des questions pratiques auxquelles peu de réponses sont apportées (*e.g.,* quels sont les principes de l'IDM les plus faciles à appliquer ?)
3. la prolifération des outils d'IDM, notamment due aux divergences de point de vue des chercheurs, complexifie les choix (*e.g.,* quel outil utiliser et pourquoi ?)
4. les organisations et processus des sociétés concernées ne sont pas adaptés (*e.g.,* peu de formation à l'IDM proposée dans les sociétés d'informatique aujourd'hui).

Pour ce qui est des SPL plus particulièrement, mettre en place un développement reposant sur les SPL de manière complètement proactive et en partant de zéro n'est pas sans risques et sans coûts pour une société [Krueger, 2002]. Cependant il existe d'autres stratégies limitant ces risques, par exemple en produisant une SPL par ré-usinage des logiciels existants et en opérant des mises à jours progressives de la SPL lorsque de nouveaux besoins se présentent[Krueger, 2002, Krueger, 2006].

3.2.2 Obstacles à l'application de l'IDM à la RV

Les approches présentées aux Paragraphes B et C appliquent quelques principes des SPL de manière imparfaite. Comme cela est illustré par les deux approches, la notion de scénario est centrale pour les EV. D'une étape à l'autre d'un scénario, les actions à réaliser changent, ainsi que les objets qui peuvent être manipulés. Dans certains cas l'application de RV peut changer de manière conséquente, de nombreux objets, métaphores et interactions peuvent être remplacés par d'autres. Ce sont donc des changements de configurations qui peuvent nécessiter une gestion rigoureuse de la variabilité, notamment par l'utilisation de FM. Or les FM seuls ne permettent pas de "scénariser" la configuration, ils ne permettent que des configurations statiques (*e.g.,* on peut déterminer quels éléments de l'application de RV seront présent dans telle ou telle version de l'application, mais on ne peut pas exprimer de changement séquencé de la configuration). Par extension, les SPL ne permettent pas de produire de manière rigoureuse des applications dans lesquelles plusieurs configurations sont utilisées de manière séquentielle. l'IDM manque donc d'une approche permettant d'utiliser les principes des SPL sur des logiciels (non nécessairement liés à la RV) reposant sur l'exécution d'un scénario.

Ce problème est différent de la problématique traitée par Ziadi *et al.* présentée en Section 1.1.4 : la modélisation de la variabilité de modèles séquentiels (que l'on peut apparenter à des spécifications de scénarios d'après la définition de Claude présentée en Section 2.2.1 [Claude, 2016]) [Ziadi et al., 2002]. Les auteurs proposent des FM pour lesquels les *features* représentent différentes étapes d'une spécification de scénarios. Or "variabiliser" des spécifications de scénarios n'est pas une problématique pertinente en RV. En effet, les modèles et moteurs de scénarios existants et évoqués en Section 2.2 permettent de spécifier et de reconfigurer efficacement des spécifications de scénarios. De plus, les spécifications de scénarios et leur diversité sont trop complexes pour que l'on puisse faire une distinction claire entre des portions de spécifications de scénarios communes à tous les scénarios d'un même domaine et des portions variables. Le nombre

de configurations possibles pour un scénario en RV peut être très grand, voire infini. L'approche de Ziadi *et al.* pourrait toutefois être appliquée à des cas bien particuliers d'applications de RV, pour lesquelles une telle distinction entre parties communes et parties variables serait pertinente. Cependant, nous n'avons pas à ce jour identifié de tels cas. En général, c'est bien la "variabilisation" de l'application de RV et sa reconfiguration scénarisée qui posent problème.

Les problèmes évoqués dans cette sous-section permettent de soulever une deuxième limite majeure :

L2 : les SPL et les FM ne permettent pas de gérer la variabilité de scénarios au nombre de configurations potentiellement infini.

Cette limite peut être comblée en atteignant l'objectif **O4-SPL-SCÉNARIO** défini en introduction (Chapitre 1).

3.3 Contributions de la thèse

Dans cette thèse, nous proposons deux approches qui comblent respectivement les manques en IDM et en RV : LPLOS (Ligne de Produits Logiciels Orientés Scénario) et LPLRV (Ligne de Produits Logiciels pour la RV).

L'approche LPLOS est indépendante de la RV. Son but est de combler la limite **L2**. Elle repose sur un modèle de scénarios qui manipule un FM : chaque étape du scénario correspond à une configuration du FM, *i.e.,* un ensemble cohérent de composants du logiciel à produire.

L'approche LPLRV repose sur LPLOS. Son but est de combler la limite **L1**. Le scénario supervise la manipulation des objets virtuels en s'appuyant sur un modèle objets-relations représentant les objets et interactions possibles dans l'application de RV.

Lignes de Produits Logiciels Orientés Scénario

3

Ce chapitre présente notre première contribution scientifique : l'approche LPLOS. Notre approche s'inspire du concept de SPL. Comme les SPL, notre approche permet la production de logiciels par réutilisation de composants. Ces logiciels ont la particularité d'être structurés par un scénario. Les LPLOS comblent les manques des SPL par rapport à la production de tels logiciels. La Section 1 présente l'approche en tant que telle. La Section 2 présente un cas d'utilisation de l'approche : la génération de documentation pour les DSL. Enfin, la Section 3 fait une synthèse du chapitre.

1 Approche

Dans cette section nous présentons l'approche. L'objectif des LPLOS est présenté en Section 1.1. Une vue d'ensemble de l'approche est présentée en Section 1.2. Un cas d'illustration qui sera utilisé en guise d'exemple pour expliquer l'approche est présenté en Section 1.3. Les différentes parties de l'approche sont détaillées de la Section 1.4 à la Section 1.6.

1.1 Objectif de l'approche LPLOS

L'objectif de l'approche est de donner un cadre pour la synthèse de produits logiciels structurés par un scénario, par réutilisation de composants. Une même LPLOS peut ainsi synthétiser plusieurs produits logiciels appartenant à une même famille.

Un produit logiciel structuré par un scénario peut être représenté par une spécification des scénarios conforme à un modèle de scénarios. Les termes de *scénario*, *spécification de scénarios* et *modèle de scénarios* ont été définis pour la RV au Chapitre 2, en Section 2.2.1. Nous utilisons des définitions similaires dans ce chapitre. Ces définitions sont inspirées de celles que Claude a énoncés pour la RV [Claude, 2016]. Nous les adaptons pour l'IDM. Elle sont illustrées par la Figure 2.7 :

— **scénario** : *agencement temporel et causal d'actions réalisées sur un logiciel* ;

— **modèle de scénario** : *formalisme et organisation de données permettant de décrire des scénarios* ;

— **spécification des scénarios** : *description d'un ensemble de scénarios possibles.*

Les logiciels que nous visons sont produits à partir d'une spécification de scénarios et de composants logiciels réutilisables, lors de la phase de synthèse. Notre approche

ne spécifie pas précisément comment les spécifications de scénarios doivent être traitées lors de la synthèse. Cela dépend en effet des logiciels que l'on veut produire. Nous distinguons au moins deux types de synthèses possibles, bien que l'approche n'y soit pas restreinte :

— **synthèse par transformation** : la spécification de scénarios, qui est une abstraction, est transformée en une forme concrète par utilisation de composants logiciels. Le logiciel produit peut alors être vu comme une spécification de scénarios concrète. Le cas d'utilisation présenté en Section 2 de ce chapitre en est une illustration : nous y présentons une LPLOS permettant de produire de la documentation pour DSL où les concepts d'un DSL sont expliqués les uns à la suite des autres (spécification de scénarios concrète). La documentation est produite à partir d'un diagramme d'activité modélisant la séquence de documentations de concepts (spécification de scénarios abstraite).

— **synthèse par intégration** : la spécification de scénarios et les composants logiciels sont intégrés à un logiciel hôte auquel l'interprétation de la spécification de scénarios est déléguée. Le rôle du logiciel hôte est en général de dérouler un ou plusieurs scénarios définis par la spécification en fonction de critères qui lui sont propres. Les applications de RV que nous souhaitons produire (voir Chapitre 4) entrent dans ce cas de figure. Ces applications de RV sont en effet produites grâce à un éditeur (logiciel hôte) capable d'interpréter des spécifications de scénarios. Chaque session d'utilisation de l'application de RV permet aux utilisateurs de réaliser une séquence d'actions (scénario) conforme à la spécification de scénarios.

1.2 Vue d'ensemble

Le schéma de la Figure 3.1 représente la structure des LPLOS. Il nécessite de fournir en entrée : le modèle du domaine, les composants logiciels implémentant ce modèle et un modèle de scénarios. A chaque utilisation d'une LPLOS, une nouvelle spécification de scénarios peut être définie, ce qui donne lieu à la production d'un nouveau logiciel, *i.e.,* un logiciel produit d'une LPLOS est défini par sa spécification de scénarios. Le modèle du domaine est une abstraction qui permet d'établir un pont entre les spécifications de scénarios et les composants logiciels, par définition des concepts qui caractérisent la famille de logiciels à produire. En effet, les spécifications de scénarios définissent des actions dont l'effet est de modifier la configuration d'un modèle de variabilité (*Feature Model*, FM), lui-même défini à partir du modèle du domaine. Ce FM constitue lui aussi une abstraction de concepts dont l'implémentation se trouve dans les composants logiciels. La phase de synthèse utilise les liens conceptuels entre les composants logiciels et les spécifications de scénario.

La nature et l'utilisation du modèle du domaine et des composants logiciels sont présentés en Section 1.4. Le modèle de scénario, qui permet de définir des spécifications de scénarios, est détaillé en Section 1.5. L'étape de synthèse est expliquée en Section 1.6.

Notons que notre approche respecte le principe suivant : *développer une fois, produire plusieurs*. En d'autres termes, une même LPLOS doit être développée une seule fois et permet de produire plusieurs logiciels. Lorsqu'une LPLOS est développée, un utilisateur doit seulement fournir la spécification de scénarios. Le logiciel correspondant

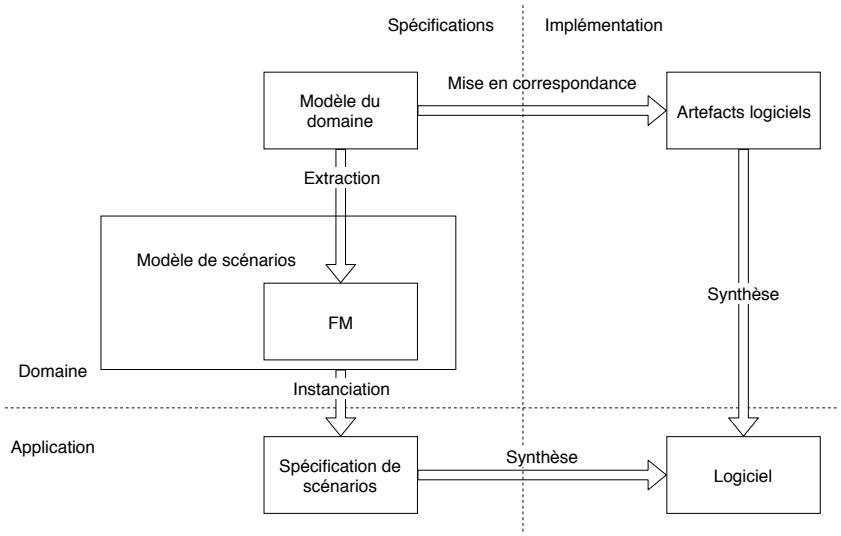


Figure 3.1 – Structure d'une LPLOS.

est ensuite synthétisé automatiquement. Toutefois, il est possible de définir plus de paramètres pour une LPLOS. Par exemple, le cas d'utilisation présenté en Section 2 de ce chapitre, une LPLOS permettant de générer de la documentation pour DSL, peut dans certains cas nécessiter de fournir en entrée de la LPLOS le métamodèle du DSL à documenter ainsi que sa grammaire et un ensemble de modèles du DSL.

1.3 Cas d'illustration : guidage d'un robot

Considérons un robot capable de rouler vers l'avant et de pivoter sur lui-même. Une télécommande permet de contrôler les moteurs de marche avant et de rotation. Le diagramme de classes de la Figure 3.2 représente le modèle interne de la télécommande. La télécommande est constituée de trois boutons : *avancer*, *tourner en sens horaire* et *tourner en sens anti-horaire*. Tant que l'on reste appuyé sur le bouton *avancer*, l'attribut *on* de la classe *Avancer* vaut *vrai* et le robot avance à vitesse constante, sinon il vaut *faux* et le robot n'avance pas. L'appui sur le bouton *tourner en sens horaire* (*resp. tourner en sens anti-horaire*) fait passer l'attribut *sensHoraire* à *vrai* (*resp. faux*). Tant que l'on reste appuyé sur l'un des boutons *tourner en sens horaire* ou *tourner en sens anti-horaire*, l'attribut *on* de la classe *Tourner* vaut *vrai* et le robot se met à tourner sur lui-même à vitesse constante dans le sens indiqué, sinon il vaut *faux* et le robot ne tourne pas. Dans la suite, nous considèrerons que la vitesse de marche avant est de 1m/s et que la vitesse angulaire est de 10°/s. Nous considérerons de plus qu'il n'est pas possible de rester appuyé sur plus d'un bouton à la fois.

Nous nous intéressons au problème suivant : en plus de contrôler directement le robot avec la télécommande, nous souhaitons pouvoir l'utiliser pour générer un programme à charger dans le robot, qui lui fera opérer les mêmes déplacements. Par exemple, considérons la séquence d'actions suivante, réalisée avec la télécommande :

1. appui sur le bouton *avancer* pendant 2,5s ;
2. appui sur le bouton *tourner en sens horaire* pendant 10s ;

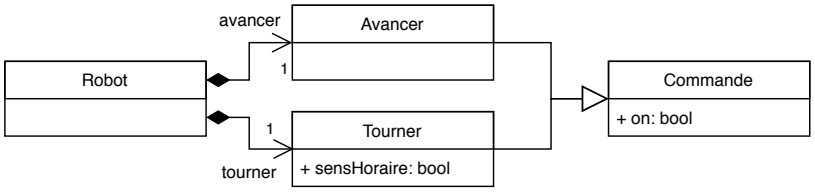


Figure 3.2 – Diagramme de classes du modèle interne de la télécommande de contrôle du robot.

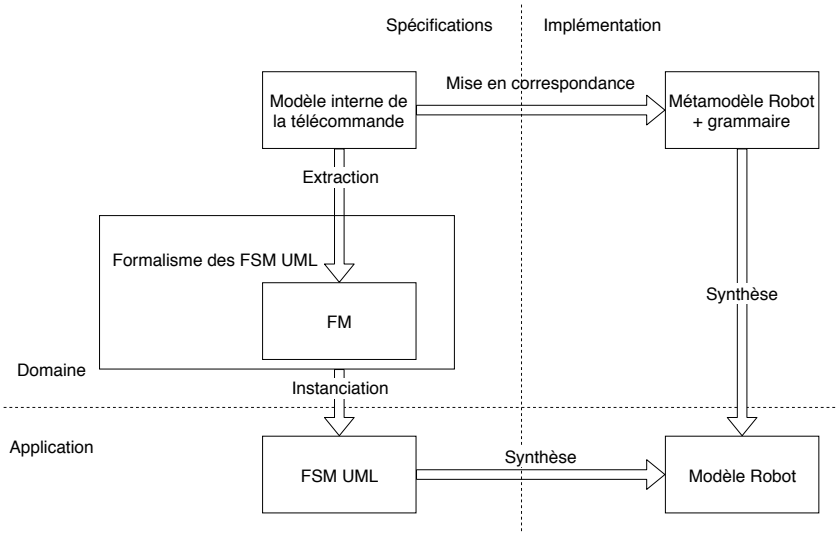


Figure 3.3 – LPLOS pour la génération de modèles *Robot* à partir d'un diagramme d'activités représentant une séquence d'actions réalisées sur la télécommande du robot.

3. appui sur le bouton *avancer* pendant 6s.

Le programme généré par cette séquence d'actions doit lui faire exécuter les mouvements correspondants, à savoir :

1. avancer sur 2,5m ;

2. opérer une rotation de $-100°$;

3. avancer sur 6m.

Le programme sera un modèle du DSL *Robot*, que nous avons introduit en Section 1.3.3 et qui est présenté en détails en annexe B. Par exemple, le modèle *Robot* correspondant aux séquences d'actions précédentes est celui de l'extrait de code 3.1.

Listing 3.1 – Exemple de modèle *Robot* généré à partir d'une séquence d'actions réalisées sur la télécommande du robot.

```
begin
  move(2.5)
  turn(-100)
  move(6)
end
```

Un tel modèle est un logiciel qui a une structure de spécification de scénarios. En effet, il s'agit d'un ensemble d'actions ordonnées chronologiquement. Le problème posé peut donc être résolu en utilisant une LPLOS, représentée en Figure 3.3. La famille de logiciels à produire correspond à l'ensemble des modèles *Robot* qui peuvent être générés à partir de la télécommande. Cette génération correspond à une synthèse par transformation. Cette synthèse utilise le métamodèle et la grammaire du DSL *Robot* pour interpréter une séquences d'actions sur la télécommande ; le métamodèle et la grammaire font donc office de composants logiciels de la LPLOS. Le diagramme des classes de la Figure 3.2, représentant bien le fonctionnement interne du robot, est réutilisé en tant que modèle du domaine. La traduction d'une séquence d'actions réalisées sur la télécommande en modèle *Robot* n'est pas directe : une séquence d'action est d'abord traduite en une FSM UML. Le modèle de scénarios est donc le formalisme des FSM UML.

1.4 Spécification et implémentation du domaine

Cette sous-section explique l'utilisation du modèle du domaine et des composants logiciels dans le cadre de l'approche LPLOS. Le modèle du domaine est expliqué en Section 1.4.1. Sa mise en correspondance avec les composants logiciels est expliquée en Section 1.4.2.

1.4.1 Modèle du domaine

Le concepteur d'une LPLOS doit fournir un modèle représentant le domaine d'appartenance du logiciel à produire. A titre d'exemple, le modèle interne de la télécommande du robot de la Figure 3.2 est un diagramme de classes UML représentatif du guidage d'un robot.

L'approche n'impose pas de formalisme spécifique, mais il est conseillé d'avoir recours à un métamodèle ou à une ontologie, facilitant le traitement automatique du modèle et son instanciation. Nous avons présenté les principes de la modélisation ainsi que les métamodèles et les ontologies dans l'état de l'art (Chapitre 2, Section 1.2). Nous avons alors indiqué les différences fondamentales entre métamodèles et ontologies. Les métamodèles obéissent à une logique de modélisation en monde fermé, *i.e.,* les relations qui existent entre des concepts doivent être explicites, sinon elles sont considérées comme inexistantes. Les ontologies obéissent en revanche à une logique en monde ouvert, *i.e.,* des relations non spécifiées peuvent exister.

Dans notre approche, l'importance de ces différences dépend des choix du concepteur de la LPLOS. En effet le modèle du domaine sert avant tout d'abstraction de départ à partir de laquelle sont établies des correspondances avec un FM et des composants logiciels. Ces liens peuvent être établis manuellement ou automatiquement lors du développement de la LPLOS. Le choix est délégué au concepteur, qui doit essentiellement tenir compte du contexte dans lequel la LPLOS est développée, ainsi que de son objectif. Le choix d'un métamodèle ou d'une ontologie dépend donc en partie de la manière dont le modèle du domaine est traité. Nous préconisons également de réutiliser des modèles déjà existants s'ils sont adaptés plutôt que de développer des modèles sur mesure pour une LPLOS. Cela permet en effet de rester en accord avec

Table 3.1 – Table de correspondance entre le modèle de la télécommande et le DSL *Robot*.

Classe	Valeur des attributs	Commande *Robot*	Distance
Avancer	$on = vrai$	*move*	Δt
Tourner	$on = vrai$, $sensHoraire = vrai$	*turn*	$-10\Delta t$
Tourner	$on = vrai$, $sensHoraire = faux$	*turn*	$10\Delta t$

l'objectif **O3-RÉUTILISATION** défini en introduction (Chapitre 1).

Dans le cas de la LPLOS présentée en Section 1.3, le modèle de la Figure 3.2 est par exemple déjà utilisé dans le contexte d'un guidage télécommandé. Nous le réutilisons pour la LPLOS car tous les concepts dont nous avons besoin sont présents (*i.e.,* avancer, tourner en sens horaire ou anti-horaire). Dans le cas de la LPLOS présentée en Section 2 de ce chapitre, les traitements automatiques imposent que le modèle du domaine soit le métamodèle d'un DSL. Nous présentons également en Section 3 du Chapitre 4 un cas dans lequel une ontologie est utilisée en tant que modèle du domaine. La raison principale en est que cette ontologie est complète et est déjà utilisée dans d'autres contextes.

1.4.2 Composants logiciels et mise en correspondance

Dans notre approche, les composants logiciels sont une représentation concrète du modèle du domaine. En fonction du type de logiciel à produire, il peut s'agir d'une bibliothèque de classes et de composants logiciels de toutes sortes, *e.g.,* des fichiers images ou audio. Dans le cas du robot, il s'agit de la spécification du DSL *Robot*, *i.e.,* du métamodèle et de la grammaire.

La mise en correspondance entre modèle du domaine et composants logiciels est similaire à celle des SPL. Elle permet la synthèse du logiciel à partir des spécifications. Dans notre approche, elle permet d'interpréter la spécification de scénarios lors de l'étape de synthèse du logiciel. Par exemple, la correspondance entre le modèle de domaine de la Figure 3.2 et le DSL *Robot* est exprimée en Tableau 3.1. Ce tableau établit le lien entre l'activation de chaque classe du diagramme de la Figure 3.2 ($on = vrai$) et la commande résultante. La mise en correspondance est paramétrée par le temps d'appui sur un bouton (Δt), information non portée par le modèle du domaine, mais qui sera fournie par la spécification de scénarios.

1.5 Modèle de scénarios

Le modèle de scénario doit permettre de produire des spécifications de scénarios. Ce modèle doit supporter au moins la notion de transition et la notion d'état. Chaque état de la spécification de scénarios correspond à une configuration possible d'un FM. Ce FM est extrait du modèle du domaine. Son rôle est d'exprimer la variabilité intrinsèque du logiciel à produire.

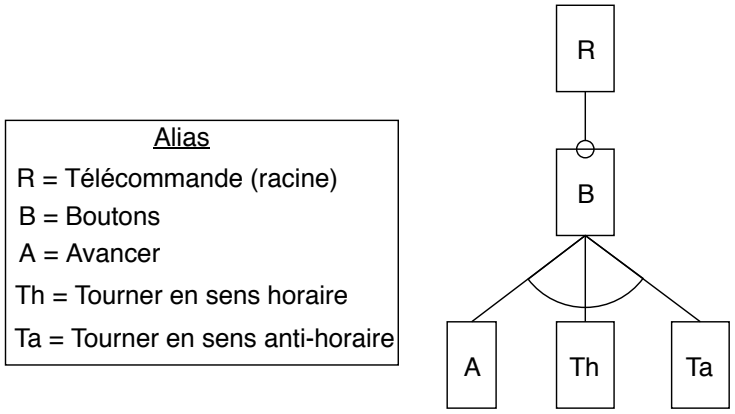


Figure 3.4 – FM construit à partir du modèle représenté en Figure 3.4 et de la connaissance que nous avons du fonctionnement de la télécommande. Les noms des *features* ont été remplacés par des alias pour pouvoir y faire référence plus facilement dans la suite de l'approche.

Table 3.2 – Table de correspondance entre le FM de la Figure 3.4 et le modèle de la Figure 3.2.

Feature	**Classe**	**Valeur des attributs**
A	$Avancer$	$on = vrai$
T_h	$Tourner$	$on = vrai$, $sensHoraire = vrai$
T_a	$Tourner$	$on = vrai$, $sensHoraire = faux$

1.5.1 Production du FM

L'approche LPLOS impose que chaque état du modèle de scénario corresponde à la configuration d'un FM. Ce FM doit être extrait du modèle de domaine. Nous n'imposons rien sur la nature de cette extraction : le FM peut être construit manuellement par le concepteur ou extrait du modèle du domaine de manière automatique. Le FM permet de superviser les configurations successives du logiciel produit au cours de l'exécution du scénario. Par exemple, la Figure 3.4 présente le FM extrait du modèle de la Figure 3.2. Ce FM décrit l'état dans lequel se trouve la télécommande à un instant donné, *i.e.*, quel est le bouton appuyé. Par exemple, la configuration $\{R\}$ signifie qu'aucun bouton n'est appuyé, alors que la configuration $\{R, B, T_a\}$ signifie que le bouton *Tourner en sens anti-horaire* est appuyé.

Notons que toute configuration possible du FM de la Figure 3.4 est soit $\{R\}$, soit de la forme $\{R, B, X\}$ où $X \in \{A, T_h, T_a\}$. En effet, l'obtention d'une configuration peut être vue comme un parcours de la structure arborescente du FM. La racine R est forcément sélectionnée. Son unique *feature* fille, B est optionnelle, donc $\{R\}$ est une configuration valide. Les trois *features* filles de B son assemblées en un même groupe XOR, donc si B est sélectionnée, strictement une d'entre elles doit être sélectionnée. Donc les autres configurations valides sont de la forme $\{R, B, X\}$ où $X \in \{A, T_h, T_a\}$.

Par construction, une correspondance doit émerger entre le FM est le modèle du domaine. Les *features* clés du FM doivent chacune correspondre à un concept du modèle du domaine. Par exemple, la Tableau 3.2 représente la correspondance entre le FM de la Figure 3.4 et le modèle de la Figure 3.2.

L'intérêt du FM par rapport au modèle du domaine est de représenter formellement les relations d'interdépendance entre les concepts du modèle du domaine. Dans l'absolu, il n'est pas impossible de concevoir un modèle de domaine spécifiant ces interdépendances. Cependant, notre solution permet une meilleure SoC. Il est alors possible d'utiliser un modèle de domaine généraliste, à partir duquel il est possible d'extraire plusieurs FM possibles. Dans le cas du modèle de la Figure 3.2 par exemple, nous aurions pu nous intéresser au même problème sans la contrainte qu'un seul bouton à la fois ne puisse être appuyé. Nous aurions alors défini un FM similaire à celui de la Figure 3.4 dans lequel le groupe XOR aurait été remplacé par un groupe OR.

Notons que d'une manière générale, notre approche permet une interchangeabilité des abstractions que nous définissons. Dans l'exemple du robot, nous pourrions par exemple obtenir une nouvelle LPLOS en remplaçant le DSL *Robot* par un autre et en gardant les mêmes modèles de domaine, modèle de scénarios et FM. La SoC permet alors de minimiser et de localiser les changements à faire si l'une des abstractions est changée.

1.5.2 Représentation formelle d'un scénario

Le modèle de scénarios doit être choisi par le concepteur en fonction des caractéristiques de la LPLOS à concevoir. Par exemple, dans le cas du robot, le formalisme des FSM UML est adapté car il permet de représenter clairement une séquence de commandes envoyées et le temps d'appui sur chaque bouton. Dans le cas général, le modèle de scénarios peut aussi être un formalisme équivalent aux machines à états, *e.g.,* les réseaux de Petri saufs. Il peut aussi être un DSL, donc un langage de scénarios taillé sur mesure pour des besoins spécifiques.

Dans notre approche, nous posons un critère formel permettant de définir quels modèles de scénarios sont compatibles avec l'approche.

Critère de validité d'un modèle de scénarios : *Tout modèle de scénarios* m *est compatible avec l'approche LPLOS si et seulement si il existe une transformation de* m *vers le formalisme des FSM UML.*

Nous choisissons les FSM UML car il s'agit d'un standard, qui permet notamment de définir des états hiérarchiques. Ainsi, une FSM classique non hiérarchique est valide car elle peut être représentée sous la forme d'une FSM UML pour laquelle tous les états sont atomiques. Les réseaux de Petri sont utilisables à condition de se limiter aux réseaux saufs.

La Figure 3.5 représente une spécification de scénarios possible pour l'exemple du robot, sous forme de FSM UML. Elle correspond à la séquence d'actions suivante : appui sur le bouton *avancer* pendant 2,5s ; appui sur le bouton *tourner en sens horaire* pendant 10s ; appui sur le bouton *avancer* pendant 6s. Chaque transition est associée à une condition indiquée entre crochets et à une action, précédée du symbole "/". Une condition correspond à un prédicat qui doit être vérifié afin d'activer la condition (*e.g.,* temps d'appui sur une touche dans le cas du robot). Une action consiste dans notre approche à modifier la configuration de l'état de départ afin d'obtenir la configuration d'arrivée. Par exemple sur la machine à états UML de la Figure 3.5, l'action $config :=$

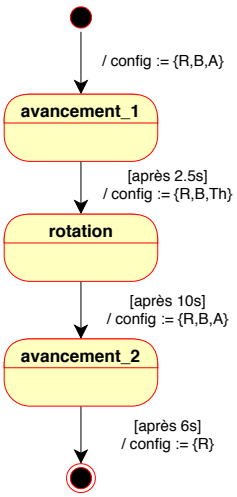


Figure 3.5 – Exemple de spécification de scénarios sous forme de FSM UML, correspondant à la séquence d'actions suivante : appui sur le bouton *avancer* pendant 2,5s ; appui sur le bouton *tourner en sens horaire* pendant 10s ; appui sur le bouton *avancer* pendant 6s. Les conditions d'activation des transitions sont entre crochets et les actions, précédées du symbole "/", correspondent à un changement de configuration ($config := \{...\}$).

$\{R, B, T_h\}$ permet de passer de la configuration $\{R, B, A\}$ à $\{R, B, T_h\}$ et correspond à l'appui sur le bouton *Tourner en sens horaire*.

Notons que dans l'exemple du robot, toute spécification de scénarios est en réalité une liste : tous les états (sauf l'état final) ont un unique successeur. Notons de même que tout état non-initial et non-final correspond à une configuration de la forme $\{R, B, X\}$, où $X \in \{A, T_h, T_a\}$. Tout état non-initial et non-final est donc caractérisé par une unique *feature* parmi A, T_h et T_a.

1.5.3 Vérification formelle de la validité des configurations

Quelque soit le modèle de scénarios choisi, les configurations associées à chaque état doivent être valides vis-à-vis du FM. Les actions de changement de configuration doivent donc permettre de conserver cette validité. Un changement de configuration peut être vu comme la combinaison d'un ajout et d'un retrait de *features*. Par exemple sur la Figure 3.5, l'action $config := \{R, B, T_h\}$ correspond à l'ajout de la *feature* T_h et au retrait de la *feature* A. Nous proposons un algorithme permettant de vérifier, étant donné une action d'ajout, une action de retrait de *features*, un FM et une configuration, si la nouvelle configuration est toujours valide (Algorithme 1).

L'Algorithme 1 prend en entrée un FM, une configuration de celui-ci, un ensemble de *features* du FM à ajouter à la configuration et un ensemble de *features* à retirer à la configuration. Le FM est d'abord converti en prédicat (Ligne 2), suivant le même modèle que celui qui est présenté au Chapitre 2, Section 1.1.3. A titre d'exemple, la représentation du FM de la Figure 3.4 est donnée en Équation (3.1). Ensuite, une application mathématique $variablesFixées$ associant à chaque *feature* à ajouter (*resp.* retirer) la valeur $vrai$ (*resp.* $faux$) est créée (Lignes 3 à 9). Puis le prédicat représentant le FM est réduit (Ligne 10) : chaque variable du prédicat ayant une valeur associée

Algorithm 1 Algorithme de vérification de la validité d'une configuration en cas d'ajout et de retrait de *features*.

```
1: procedure CONFIGURATIONVALIDE(fm, configuration, ajout, retrait)
2:     prédicat := transformerFmEnPrédicat(fm)
3:     variablesVrai := ∅
4:     variablesFaux := ∅
5:     for each f ∈ ajout do
6:         variablesVrai := variablesVrai ∪ {f ↦ vrai}
7:     for each f ∈ retrait do
8:         variablesFaux := variablesFaux ∪ {f ↦ faux}
9:     variablesFixées := variablesVrai ∪ variablesFaux
10:    prédicat := réduire(prédicat, variablesFixées)
11:    solution := résoudre(prédicat)
12:    if solution ≠ indéterminé then
13:        return solution
14:    else
15:        variables := obtenirVariables(prédicat)
16:        variablesVrai := ∅
17:        variablesFaux := ∅
18:        for each f ∈ variables ∩ configuration do
19:            variablesVrai := variablesVrai ∪ {f ↦ vrai}
20:        for each f ∈ variables − configuration do
21:            variablesFaux := variablesFaux ∪ {f ↦ faux}
22:        variablesFixées := variablesVrai ∪ variablesFaux
23:        prédicat := réduire(prédicat, variablesFixées)
24:        return résoudre(prédicat)
```

par *variablesFixées* est remplacée par cette valeur, puis simplifiée. Un exemple de réduction pour la proposition de l'Équation (3.1) est proposé en Équation (3.2). Puis, le prédicat simplifié est résolu (Ligne 11) : la méthode *résoudre*($formule$) retourne $formule$ si c'est un booléen, sinon la valeur *indéterminé* est retournée. Si c'est un booléen, alors il est retourné et l'algorithme se termine (Ligne 13). Une valeur *vrai* indique alors que la configuration peut être modifiée comme souhaité, une valeur $faux$ indique que le changement souhaité est impossible. Une valeur *indéterminé* veut dire que la validité de la nouvelle configuration dépend de la configuration actuelle et qu'il faut donc vérifier les valeurs des *features* de celle-ci. Dans ce cas, le prédicat est de nouveau réduit puis résolu en associant à toutes les variables restantes leur valeur (Lignes 15 à 24). La valeur de la résolution est retournée et l'algorithme prend fin.

$$\begin{aligned} prédicat = R \wedge (B \Rightarrow R) \wedge (B \Leftrightarrow (A \vee T_h \vee T_a)) \\ \wedge (A \Rightarrow (\neg T_h \wedge \neg T_a)) \wedge (T_h \Rightarrow (\neg A \wedge \neg T_a)) \wedge (T_a \Rightarrow (\neg A \wedge \neg T_h)) \end{aligned} \quad (3.1)$$

$$réduire(prédicat, \{R \mapsto faux\}) = faux \wedge ... \wedge (T_a \Rightarrow (\neg A \wedge \neg T_h)) = faux \quad (3.2)$$

1.6 Synthèse d'un logiciel

La dernière étape est celle de la synthèse du logiciel. Elle est réalisée par réutilisation des composants logiciels correspondants à une spécification constituée d'un modèle instanciant le modèle du domaine. Dans notre approche, ces éléments sont également mis en relation avec la spécification de scénarios produite par le concepteur.

Cette étape est la plus dépendante du logiciel à produire. Tout d'abord, il existe différents types de synthèse, *e.g.,* la synthèse par transformation et la synthèse par intégration (voir Section 1.1). Ensuite, la manière d'utiliser les composants logiciels dépend du but des logiciels à produire. Par exemple, le cas d'illustration et le cas d'utilisation présenté en Section 2 de ce chapitre (production de documentation pour les DSL) sont deux cas de synthèse par transformation pour lesquels la grammaire d'un DSL est utilisée comme composant logiciel. Cependant, dans le premier cas nous cherchons à produire du code et dans le deuxième cas de la documentation. Les deux processus de synthèse sont donc différents, même s'il existe quelques points communs techniques, impossibles à étendre au cas général.

Dans notre cas d'illustration, le logiciel à produire est un modèle *Robot*, représenté sous forme textuelle selon sa grammaire. Par exemple, le logiciel produit par interprétation de la spécification de scénarios de la Figure 3.5 est le modèle *Robot* correspondant à l'extrait de code 3.1. Dans ce cas, la spécification de scénarios peut par exemple être interprétée par l'Algorithme 2.

L'Algorithme 2 prend en entrée la FSM UML de la Figure 3.5 (spécification de scénarios) et les tables de mise en correspondance (Tableaux 3.1 et 3.2). Il retourne le modèle *Robot* correspondant sous forme de liste, *e.g.,* [“$begin$”, “$move(2.5)$”, “$turn(-100)$”, “$move(6)$”, “end”]. Tout d'abord, les tables sont fusionnées selon leurs colonnes communes **Classe** et **Valeur des attributs** (Ligne 2), ce qui donne le

Algorithm 2 Algorithme d'interprétation d'une séquence d'actions effectuées sur la télécommande du robot (*e.g.,* FSM UML de la Figure 3.5) pour la production du modèle *Robot* équivalent.

```
1: procedure INTERPRÉTEUR(fsmUml, tableCorrespFM, tableCorrespMdl)
2:     tableCorresp := fusionner(tableCorrespFM, tableCorrespMdl)
3:     modèleRobot := ["begin"]
4:     étatInitial := obtenirÉtatInitial(fsmUml)
5:     transition := obtenirTransitionAval(étatInitial)
6:     état := obtenirÉtatAval(transition)
7:     modèleRobot := Interpréteur_récursion(état, tableCorresp, modèleRobot)
8:     modèleRobot := ajouter(modèleRobot, "end")
9:     return modèleRobot
10: procedure INTERPRÉTEUR_RÉCURSION(état, tableCorresp, modèleRobot)
11:     if estFinal(état) then
12:         return modèleRobot
13:     transition := obtenirTransitionAval(état)
14:     config := obtenirConfig(état)
15:     featureAction := obtenirÉlément(config − {R, B})
16:     condition := obetnirCondition(transition)
17:     temps := obtenirTemps(condition)
18:     cmdDSL := obtenirCommandeDSL(tableCorresp, featureAction)
19:     distance := calculerDistance(tableCorresp, featureAction, temps)
20:     modèleRobot := ajouter(modèleRobot, cmdDSL + "(" + distance + ")")
21:     étatAval := obtenirÉtatAval(transition)
22:     corpsInterpréteur(étatAval, tableCorresp, modèleRobot)
```

Table 3.3 – Fusion des Tableaux 3.1 et 3.2.

Feature	**Classe**	**Valeur des attributs**	**Commande *Robot***	**Distance**
A	*Avancer*	$on = vrai$	*move*	Δt
T_h	*Tourner*	$on = vrai$, $sensHoraire = vrai$	*turn*	$-10\Delta t$
T_a	*Tourner*	$on = vrai$, $sensHoraire = faux$	*turn*	$10\Delta t$

Tableau 3.3. Ensuite, le modèle *Robot* est initialisé avec une commande *begin*, obligatoire (Ligne 3).

L'étape suivante consiste à calculer le reste du modèle *Robot*. Pour cela, la procédure récursive *Interpréteur_récursion* est utilisée (Ligne 7). Cette procédure nécessite de passer en entrée un état d'une FSM UML. Cet état doit être le premier état non-initial d'une spécification de scénarios de la LPLOS (Lignes 4 à 6). Le principe de la procédure est le suivant : pour un état donné (donc une configuration du FM) et la condition de la transition suivante (donc le temps d'appui sur un bouton), la commande *Robot* correspondante est calculée. La condition d'arrêt de la procédure est donc que l'état en entrée est l'état final (Ligne 11). Pour tout autre état, la transition en aval est récupérée (Ligne 13), puis la *feature* caractéristique de la configuration (A, T_h ou T_a) est récupérée, ainsi que le temps d'appui sur le bouton (Lignes 14 à 17). Ensuite, la commande *Robot* est déduite de la table de correspondance (Lignes 18 à 20). La procédure est ensuite appelée récursivement sur l'état suivant (Ligne 22).

Lorsque toute la spécification de scénarios a été transformée, une dernière commande *end*, obligatoire d'après la grammaire de *Robot* (voir Annexe B), est ajoutée à la fin du modèle (Ligne 8), avant que celui-ci ne soit retourné (Ligne 9).

Notons que la procédure de synthèse du modèle *Robot* prend en compte la grammaire de manière implicite. Le respect de cette dernière est en effet assuré par construction dans le Tableau 3.1 (colonne **Commande *Robot***) et l'Algorithme 2 (Lignes 3, 8 et 20).

2 Cas d'utilisation : production de documentation pour DSL

Dans cette section, nous présentons une application de notre approche adaptée de l'un des articles publiés durant cette thèse [Le Moulec et al., 2018]. Cette application est une LPLOS permettant de générer de la documentation pour les DSL de manière semi-automatique. La Section 2.1 présente les motivations et une formalisation du problème à résoudre. La Section 2.2 présente la conception de la LPLOS. La Section 2.3 présente l'outil dans lequel cette LPLOS a été implémentée, *Docywood*.

2.1 Introduction au cas d'utilisation

Dans cette section, nous présentons tout d'abord les raisons qui ont motivé la conception d'une LPLOS pour la production de documentation de DSL (Section 2.1.1). Ensuite, nous formalisons le problème à résoudre sous forme de quatre propriétés à vérifier (Section 2.1.2).

2.1.1 Motivations

Les DSL sont habituellement de petits langages, mais leur développement nécessite un effort conséquent [Sprinkle et al., 2009, Voelter et al., 2013]. Ainsi, syntaxes concrètes, éditeurs et compilateurs sont des exemples de principaux composants faisant partie de l'écosystème entourant les DSL. Des travaux de recherches se concentrent sur le défi de faciliter le développement de ces composants afin de réduire les coûts de développement et de maintenance des DSL [Bettini, 2013, Degueule et al., 2015, Cánovas and Cabot, 2013, Kosar et al., 2016].

Le cas d'utilisation que nous proposons dans cette section est une LPLOS dont le but est de faciliter la production et la maintenance de documentation utilisateur pour les DSL. Documenter un DSL est en effet une tâche importante mais chronophage [Mernik et al., 2005, Sprinkle et al., 2009]. Cette tâche, cependant, est nécessaire afin de diffuser un DSL, permettre son apprentissage par ses utilisateurs potentiels [Sprinkle et al., 2009] et limiter le "*problème de cacophonie des langages*" : les langages sont difficiles à apprendre et l'utilisation d'une multitude de langages est plus bien plus compliqué que d'en utiliser un seul [Fowler, 2005].

2.1.2 Formalisation du problème

Les langages informatiques sont des logiciels à part entière [Favre et al., 2010] et les relations entre les API et les DSL ont été étudiées [Mernik et al., 2005, Bravenboer and Visser, 2004]. De ses travaux, il ressort que la documentation des DSL et celle des API devraient répondre aux mêmes critères : les pièges liés à la documentation d'API ont été identifiés à partir des retours de leurs utilisateurs [Uddin and Robillard, 2015] et à partir de ces pièges, nous avons dérivé quatre propriétés que les utilisateurs et les concepteurs de DSL peuvent attendre d'une documentation.

Propriété #1 – Documentation exhaustive. l'incomplétude de la documentation a été identifiée comme le problème le plus important qui affecte la documentation d'API. Cette propriété garantit que le critère de couverture a été respecté. Nous définissons ce critère ainsi : tous les concepts du DSL doivent être couverts par la documentation. Quand le modèle du domaine du DSL est un métamodèle, ce critère peut être décomposé en trois sous-critères : tous les attributs, références et classes du métamodèle sont couverts par la documentation.

Propriété #2 – Contextualisation de la documentation. La documentation doit être contextualisée de façon à répondre aux besoins des utilisateurs du DSL, *i.e.,* sa forme et son contenu doivent être adaptés à l'utilisation qui va en être faite. Cela limite les problèmes liés à des documentations constituées de nombreux et longs paragraphes pas toujours pertinents du point de vue des utilisateurs.

Propriété #3 – Documentation sous plusieurs formes. Maintenir de la documentation manuellement sous plusieurs formes (pages web, documents, intégrée à un environnement de développement, *etc.*) est un tâche complexe qui peut déboucher sur une documentation obsolète.

Propriété #4 – Documentation reposant sur des exemples. Les utilisateurs apprécient le recours à des exemples dans la documentation, qui doivent être

suffisamment bien expliqués.

Au tout début du développement d'un DSL, l'analyse du domaine peut aider à indentifier les concepts du DSL et à produire de la documentation, même peu précise [Van Deursen and Klint, 2002]. Les concepteurs du DSL peuvent écrire manuellement de la documentation en utilisant des outils tels que EcoreTools [Steinberg et al., 2008]. Ces différentes approches ne permettent pas de satisfaire ces quatre propriétés. En effet, si les concepteurs peuvent documenter un métamodèle, cette documentation est surtout destinée à d'autres concepteurs, afin de développer les outils complémentaires au DSL (*e.g.,* éditeurs, compilateurs). Or, Les quatre propriétés concernent les utilisateurs d'un DSL, qui l'utiliserons grâce aux outils complémentaires. De plus, la documentation des métamodèles ne s'appuie en général pas sur des exemples. La LPLOS proposée satisfait quant à elle ces quatre propriétés.

La Section 2.2 détaille la conception de la LPLOS. La Section 2.3 présente l'implémentation de la LPLOS en un outil, *Docywood*[1], et en compare les caractéristiques avec les quatre propriétés précédemment définies.

Une évaluation de *Docywood* au travers d'une expérience est présentée dans l'article [Le Moulec et al., 2018]. Nous ne détaillons pas cette évaluation dans ce manuscrit car nous considérons que bien que nécessaire à la validation théorique de l'utilisabilité de *Docywood*, cette évaluation reste propre à ce seul cas d'utilisation et ne peut pas être généralisée à l'approche des LPLOS en elle-même. Nous nous contenterons dans ce manuscrit de donner les conclusions de cette évaluation. En l'occurrence, *Docywood* a été testé sur les deux DSL *ThingML* et *Target Platform Definition*. Les retours d'utilisateurs et des concepteurs de ces DSL ont montré des gains qualitatifs par rapport au défi qu'est la documentation des DSL. L'évaluation expérimentale a montré en particulier que la documentation générée par *Docywood* pour *ThingML* permettait aux utilisateurs de produire des modèles plus précis de ce DSL.

2.2 Conception de la LPLOS

Cette sous-section présente la conception de la LPLOS en détails. Tout d'abord, nous présentons les spécifications précises de la LPLOS que nous souhaitons implémenter (Section 2.2.1). Puis, la Section 2.2.2 détaille le contenu de la documentation à produire. La Section 2.2.3 donne une vue d'ensemble du fonctionnement de la LPLOS. Les différentes étapes sont ensuite détaillées en Sections 2.2.4 et 2.2.5. Nous illustrerons les explications au fur et à mesure en montrant l'utilisation de la LPLOS pour documenter le DSL *Robot* (présenté en Annexe B). Nous rappelons ici le métamodèle de *Robot* (Figure 3.6), ainsi que sa grammaire (extrait de code 3.2) et un exemple de modèle *Robot* (extrait de code 3.3).

Listing 3.2 – La grammaire Xtext du DSL *Robot*

```
ProgramUnit: 'begin' (commands+=Command)* 'end';
Command: Move | Turn | WhileNoObstacle;
```

1. https://github.com/arnobl/comlanDocywood.

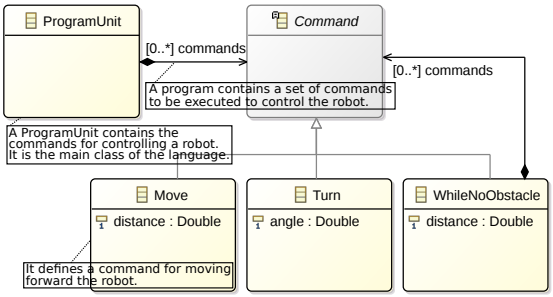


Figure 3.6 – Métamodèle du langage *Robot*.

```
Move: 'move' '(' distance=Double ')';
Turn: 'turn' '(' angle=Double ')';
WhileNoObstacle: 'whileNoObstacleAt' '(' distance=Double ')' '{'
(commands+=Command)* '}';
```

Listing 3.3 – Un modèle *Robot*

```
begin
  move(25)
  whileNoObstacleAt(10) {
    move(75)
    turn(90)
    move(50)
  }
  turn(-100)
  move(60)
end
```

2.2.1 Spécification de la LPLOS

La LPLOS proposée se concentre sur des DSL externes définis à l'aide d'une grammaire. Pour rappel, un DSL externe est un langage à-part entière, au contraire d'un DSL interne qui est utilisable dans un langage hôte (voir Chapitre 2, Section 1.3.1). Les DSL externes représentent 50% des DSL [Kosar et al., 2016]. Si les DSL graphiques peuvent être traités avec cette LPLOS, les DSL ne reposant pas sur une grammaire devraient induire des changements conséquents de la LPLOS. Par ailleurs, la LPLOS est généralisable à tous les méta-métamodèles qui supportent les concepts de classe, attribut, relation, héritage et cardinalité. C'est le cas de plusieurs langages de méta-métamodélisation largement utilisés, tels que Ecore (utilisé dans notre implémentation), UML[2] ou MetaEdit+ [Tolvanen and Kelly, 2009]. Enfin, la LPLOS est plutôt pensée pour documenter les DSL à destination d'utilisateurs habitués à développer. Les autres DSL peuvent être documentés avec cette LPLOS, mais il faudrait alors valider la facilité d'utilisation de celle-ci par des utilisateurs non habitués à programmer.

2. http://www.omg.org/spec/UML

La LPLOS produit de la documentation utilisateur à partir d'éléments issus du développement même des DSL : leur métamodèle, leur grammaire et des modèles couvrants les concepts de ces DSL. Pour chaque concept d'un DSL, le métamodèle, la grammaire et le modèle sont tronqués par une méthode dite de *slicing* [Blouin et al., 2015, Blouin et al., 2011], afin de garder uniquement les éléments qui sont en relation avec ce concept. Une unité de documentation dédiée à ce concept peut alors être produite. Elle est composée d'un exemple d'illustration et d'explications à propos du concept et de ses paramètres, en langage naturel. Ces explications ne sont pas entièrement générées par notre LPLOS : la documentation du métamodèle est extraite afin d'être utilisée dans la documentation générée. L'un des avantages de notre LPLOS est sa capacité à réutiliser des composants déjà existants pour produire de la documentation sous plusieurs formes différentes. Ainsi, notre documentation est à ce jour générée sous deux formes :

1. en format *Markdown* pour être facilement intégrée à des wikis ;
2. en code Java pour être intégrée à l'éditeur de DSL Xtext [Bettini, 2013] et ainsi proposer aux utilisateurs d'un DSL une documentation contextuelle lors de l'auto-complétion.

Notre LPLOS respecte un critère de couverture : la documentation produite pour un DSL explique tous les concepts du métamodèle (*i.e.,* toutes les classes, attributs et références).

2.2.2 Structure et contenu de la documentation

La documentation produite par la LPLOS est un ensemble d'unités de documentation qui permettent chacune de documenter un concept du DSL étudié. Nous nommons ces unités des *documentations de concept*.

Une documentation de concept est composée : d'un modèle du DSL conforme à la grammaire pour illustrer le concept en question ; d'une documentation textuelle de ce concept ; d'instructions pour créer l'exemple d'illustration ; des références vers les autres documentations de concepts liés.

Une documentation de concept est extraite d'un modèle fourni en entrée de la LPLOS. La documentation textuelle est extraite de la documentation du métamodèle. Si le DSL étudié n'a pas de métamodèle documenté, la LPLOS fonctionne mais l'explication des concepts ne peut pas être produite.

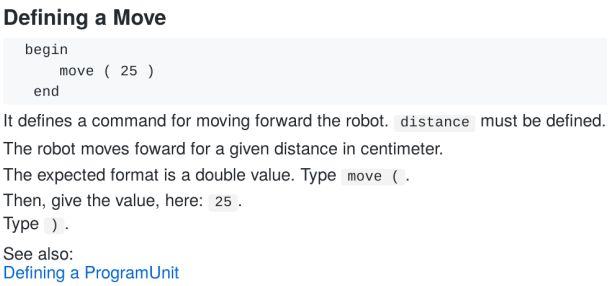


Figure 3.7 – La documentation de concept expliquant le concept *Move* du DSL *Robot*.

La Figure 3.7 illustre une documentation de concept du DSL *Robot*. La documentation de concept commence par un modèle extrait du programme 3.3 qui

couvre le concept ciblé, *i.e.,* la commande *Move*, ainsi que les éléments qui y sont liés (ici, *ProgramUnit* et l'attribut *distance*). Ensuite, le texte de la documentation de concept explique les différents éléments pas encore expliqués dans d'autres documentations de concept : *ProgramUnit* dispose de sa propre documentation de concept, ce concept n'est donc pas expliqué dans la documentation de *Move*. Le texte généré utilise : la documentation extraite du métamodèle de *Robot* ; la syntaxe concrète définie par sa grammaire ; le modèle d'illustration (extrait de code 3.3). Les liens vers les documentations des concepts liés sont fournis. Par exemple la documentation du concept *Move* se termine par un lien vers la définition de *ProgramUnit*.

2.2.3 Vue d'ensemble

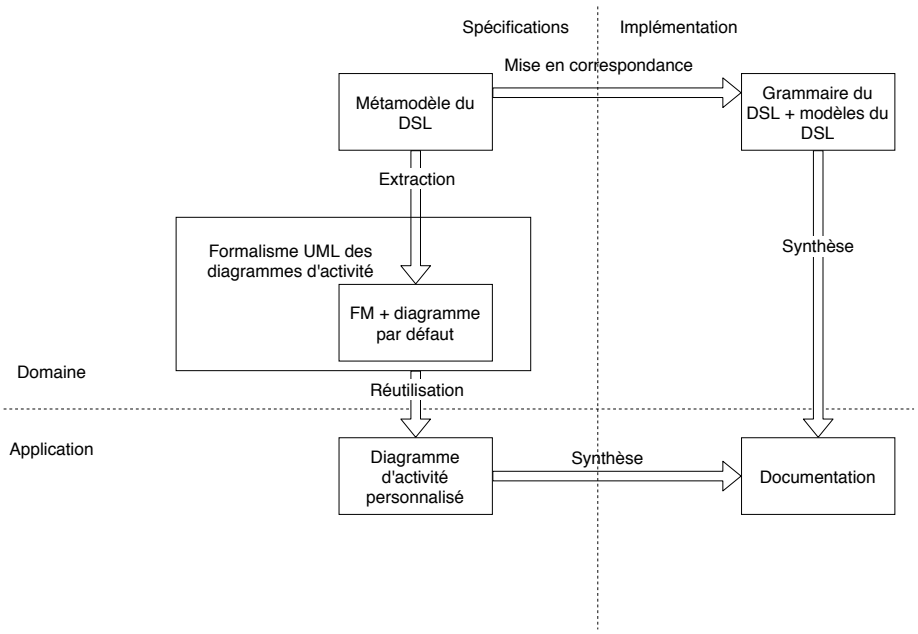


Figure 3.8 – Vue d'ensemble de la LPLOS pour la génération de documentation de DSL.

La Figure 3.8 donne une vue d'ensemble de la LPLOS. Le modèle du domaine est le métamodèle du DSL à documenter. Il est mis en correspondance avec la grammaire du DSL et un ensemble de modèles du DSL, *i.e.,* des instances de ce même métamodèle. Une grammaire en tant que telle est une spécification d'un DSL. Ici, elle est du côté de l'implémentation car elle est tout de même une implémentation du métamodèle, dans le sens où elle lui fournit une syntaxe concrète.

Le modèle de scénarios utilisé dans la LPLOS est le formalisme des diagrammes d'activité UML. Le FM est extrait du métamodèle par un algorithme présenté en Section 2.2.4. Dans notre LPLOS, l'extraction du FM s'accompagne de la génération d'une première spécification de scénarios, conforme au modèle de diagramme d'activité que nous proposons. Ce premier diagramme constitue une spécification par défaut qui peut être modifiée manuellement par le concepteur du DSL à documenter, pour obtenir d'autres spécifications. La documentation est ensuite synthétisée à partir d'une spécification de scénarios, de la grammaire et d'un modèle du DSL.

2.2.4 Modèle et spécifications de scénarios

Le modèle de scénarios utilisé dans cette LPLOS est le formalisme UML des diagrammes d'activité (adapté aux utilisateurs cibles de la documentation générée, qui sont des développeurs). La première fois que le concepteur du DSL utilise la LPLOS, un tel diagramme est généré automatiquement, ainsi que le FM associé. Ce diagramme décrit les différentes séquences possibles de documentations de concept, une séquence correspondant donc à un scénario. Le diagramme généré permet de proposer une spécification de scénarios standard. Il peut ensuite être modifié par le concepteur du DSL pour changer la spécification de scénarios de documentation.

A Génération du diagramme d'activités et du FM

L'Algorithme 3 détaille la génération du diagramme d'activité ainsi que du FM. La Figure 3.10 illustre cet algorithme pour le DSL *Robot*. Le Tableau 3.4 complète la Figure 3.10 en précisant la configuration du FM pour chaque activité. Le premier objectif du diagramme est d'ordonnancer la génération des documentations de concept. Cet ordre est obligatoire puisque la documentation d'un concept va contenir des détails à propos de ses concepts parents. Par exemple, la classe *Move* du DSL *Robot* doit être expliquée après la classe *ProgramUnit* à cause de la composition qui existe entre ces deux classes dans le métamodèle de *Robot* (voir Figure 3.6).

L'algorithme suit la structure du métamodèle. La première étape consiste à détecter la classe racine du métamodèle (Ligne 2), *i.e.,* la classe représentant le concept duquel dépendent tous les autres. Pour le DSL *Robot*, la classe racine est *ProgramUnit* (voir Figure 3.10a). Tout modèle *Robot* doit en effet comporter au minimum une instance de *ProgramUnit*. L'activité correspondant à cette classe est alors produite (Ligne 8). Le FM est initialisé et la *feature* racine est créée, correspondant de même à la classe racine (*feature ProgramUnit*, voir Figure 3.9).

L'algorithme repose en grande partie sur une méthode dite de *slicing* [Blouin et al., 2015] pour obtenir une portion minimale de métamodèle correspondant à un ensemble de critères, un critère étant un élément du métamodèle. Le *slicing* nous permet de récupérer petit-à-petit les différents concepts à documenter. Pour un critère donné (donc un concept en entrée), cette opération retourne le concept plus les concepts qui y sont fortement liés (*i.e.,* attributs et relations de cardinalité minimale supérieure ou égale à 1 si le critère est une classe, classe ciblée si le critère est une relation). Tous les concepts appartenant à la portion obtenue seront expliqués dans la même documentation de concept. Par exemple, le *slicing* avec pour critère *ProgramUnit* produit une portion de métamodèle constituée uniquement de la classe *ProgramUnit*, car celle-ci n'a pas d'élémets obligatoires (voir Figure 3.10a).

L'Algorithme 3 récupère ainsi un élément *elt* du métamodèle, créé l'activité et la *feature* correspondantes (Lignes 12 et 13), calcule la portion du métamodèle à documenter par *slicing* et l'ajoute à l'ensemble des éléments expliqués (Ligne 16). Ensuite, tous les éléments contenus dans l'élément courant *elt* (*e.g.,* attributs et références d'une classe) qui ne sont pas encore expliqués sont rassemblés (Ligne 19). Le diagramme d'activité ainsi que le FM sont ensuite construits récursivement sur ces éléments : pour chacun d'eux, une activité va être produite à la suite de l'activité

courante et une *feature* fille va être ajoutée à la *feature* courante (Lignes 20 et 21). L'ajout de *features* filles (Ligne 18) se fait de la manière suivante (voir Figure 3.9) :

— si la *feature* parente correspond à une classe dans le métamodèle, alors chacune de ses *features* filles correspond à un de ses attributs ou références. La *feature* fille est obligatoire si l'attribut ou la référence a une cardinalité minimale supérieure ou égale à 1, optionnelle sinon ;

— si la *feature* parente correspond à une relation, les *features* filles correspondent à la classe cible de la relation et aux classes qui en héritent (en ignorant les classes abstraites). Elle sont alors regroupées en un bloc OR.

On note que la validité des configurations correspondant à chaque activité est assurée par construction.

Algorithm 3 Algorithme de génération du diagramme d'activités.

```
1: procedure GENERATEACTIVITYDIAGRAM(metamodel)
2:     rootClass := extractRootClass(metamodel)
3:     initNode := new InitNode
4:     emptyFeature := new EmptyFeature
5:     activityDiag := {initNode}
6:     featureModel := {emptyFeature}
7:     configurations := ∅
8:     genActivity(rootClass, metamodel, initNode, emptyFeature, ∅, activityDiag, featureModel, ∅)
9:     return activityDiag, featureModel
10:
11: procedure GENACTIVITY(elt, metamodel, prevActivity, parentFeature, explained, diag,
    fm, prevSelectedFeature)
12:     activity := createActivity(elt)
13:     feature := createFeature(elt)
14:     prevSelectedFeature := prevSelectedFeature ∪ {feature}
15:     activity.setConfiguration(prevSelectedFeature)
16:     explained := explained ∪ sliceMetamodel(metamodel, elt)
17:     diag := diag ∪ {prevActivity → activity}
18:     fm := fm ∪ addFeature(parentFeature, feature)
19:     elements := getElements(elt) \ explained
20:     for each e ∈ elements do
21:         genActivity(e, metamodel, activity, feature, clone(explained), diag, fm,
22:         clone(prevSelectedFeature))
```

La Figure 3.10 montre le déroulement de l'algorithme pour l'exemple du DSL *Robot*. La classe *Command* est abstraite et ne donne pas lieu à la création d'une activité. Ce sont les classes qui en héritent qui sont considérées. La construction prend fin lorsque tous les concepts du métamodèle ont été visités. L'ensemble de critères de l'opération de *slicing* est alors le métamodèle lui-même. Le critère de couverture est donc vérifié.

B Production d'une spécification de scénarios par personnalisation du diagramme d'activité

Une fois que le diagramme d'activité a été généré, un concepteur du DSL peut fusionner, modifier des activités et changer leur ordre. Le but est de permettre

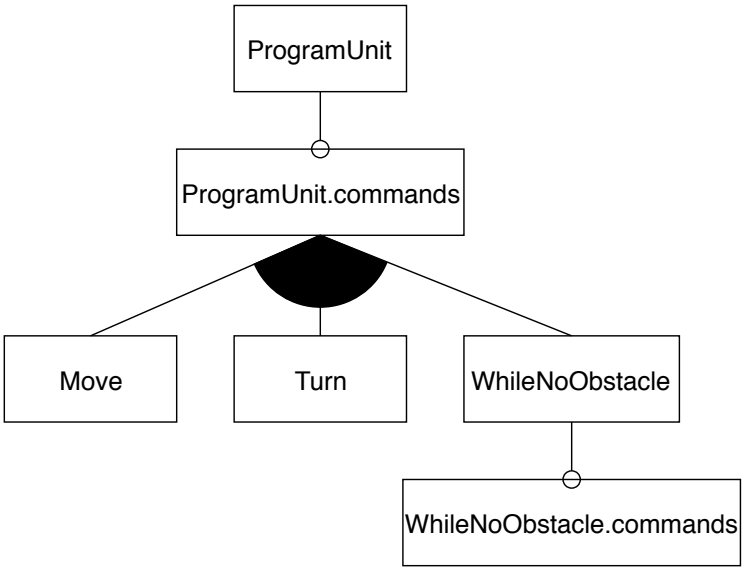
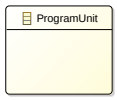


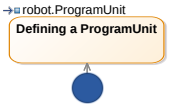
Figure 3.9 – FM généré pour le langage *Robot*.

Table 3.4 – Configuration dérivée du FM de la Figure 3.9 pour chaque activité générée.

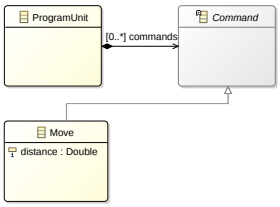
Activité	**Configuration**
"Defining a ProgramUnit"	*ProgramUnit*
"Defining a Move"	*ProgramUnit*, *ProgramUnit.commands*, *Move*
"Defining a Turn"	*ProgramUnit*, *ProgramUnit.commands*, *Turn*
"Defining a WhileNoObstacle"	*ProgramUnit*, *ProgramUnit.commands*, *WhileNoObstacle*
"Defining a WhileNoObstacle with a command"	*ProgramUnit*, *ProgramUnit.commands*, *WhileNoObstacle*, *WhileNoObstacle.commands*



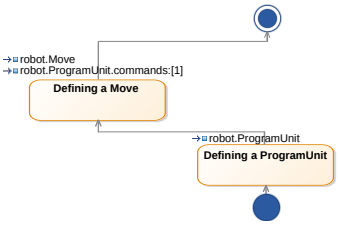
(a) Critère de *slicing* : *ProgramUnit*.



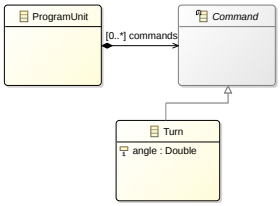
(b) Activité produite avec *ProgramUnit*.



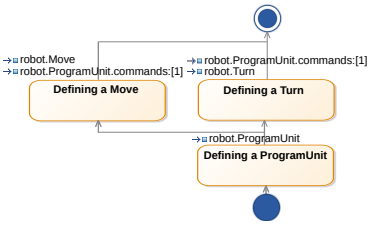
(c) Critères de *slicing* : *ProgramUnit*, *commands* et *Move*.



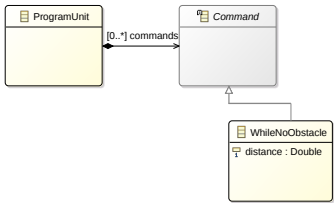
(d) Production de l'activité *Move*



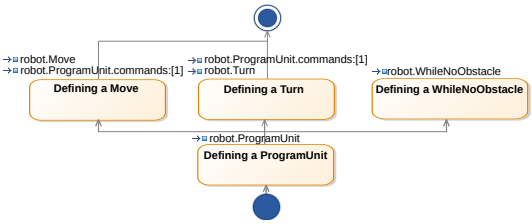
(e) Critères de *slicing* : *ProgramUnit*, *commands* et *Turn*



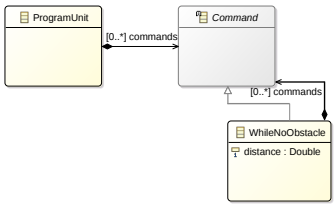
(f) Production de l'activité *Turn*



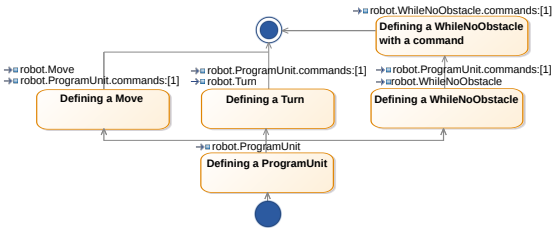
(g) Critères de *slicing* : *ProgramUnit*, *commands* et *WhileNoObstacle*



(h) Production de l'activité *WhileNoObstacle*.



(i) Critères de *slicing* : *ProgramUnit*, *commands*, *WhileNoObstacle* et sa composition *commands*



(j) Productionde l'activité correspondant à la composition *commands* de *WhileNoObstacle*

Figure 3.10 – Illustration de l'Algorithme 3 avec le DSL *Robot*. A gauche, les portions de métamodèle obtenues par *slicing*. A droite, le diagramme d'activité correspondant.

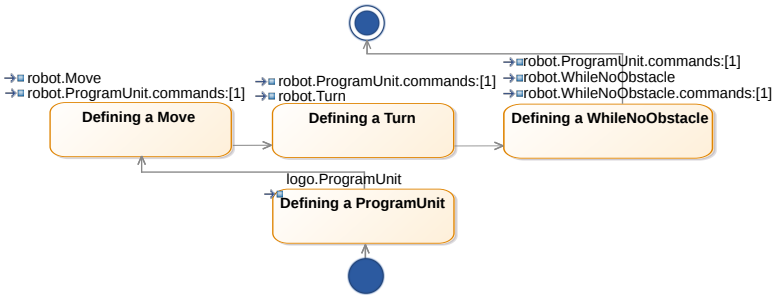


Figure 3.11 – Le diagramme d'activité de la Figure 3.10j modifié par un concepteur de DSL.

Defining a Turn

```
begin
    turn ( -100 )
end
```

It defines a command for rotating the robot. `angle` must be defined.
The robot rotates following a given rotation angle in degree.
The expected format is a double value. Type `turn (` .
Then, give the value, here: `-100` .
Type `)` .

See also:
Defining a ProgramUnit

Figure 3.12 – Documentation de la classe *Turn* générée à partir du diagramme de la Figure 3.10j.

au concepteur de personnaliser la documentation générée. Par exemple avec *Robot*, un concepteur pourrait vouloir expliquer la classe *Turn* après la classe *Move*, en utilisant celle-ci. La Figure 3.11 représente le diagramme de la Figure 3.10j ainsi modifié par un concepteur. La classe *Turn* vient après la classe *Move*. La configuration de l'activité "Defining a *Turn* " est maintenant l'ensemble de *features* $\{ProgramUnit, ProgramUnit.commands, Move, Turn\}$. La documentation de *Turn* (voir Figure 3.13) contient maintenant une commande *Move* dans l'exemple de code, ainsi qu'une référence vers la documentation de *Move*, contrairement à la documentation de *Turn* générée à partir du diagramme d'activité original (voir Figure 3.12). Cependant, le concept *Move* est expliqué dans sa propre documentation, pas dans celle de *Turn*.

Une autre modification par rapport au diagramme original est la fusion des activités "*Defining a* WhileNoObstacle" et "*Defining a* WhileNoObstacle *with a command*". Le but en est d'expliquer *WhileNoObstacle* et sa référence optionnelle *commands* dans la même documentation.

2.2.5 Génération de la documentation

La génération de la documentation correspond à la dernière étape de l'approche LPLOS, *i.e.,* l'étape de la synthèse. Le processus de génération prend donc en entrée le diagramme d'activité ainsi que le modèle concret et la grammaire du DSL. L'Algorithme 4 détaille ce processus. Chaque activité du diagramme conduit à la production d'une documentation de concept. La génération de la documentation commence avec la première activité du diagramme (Lignes 1 et 2), *e.g.,* l'activité

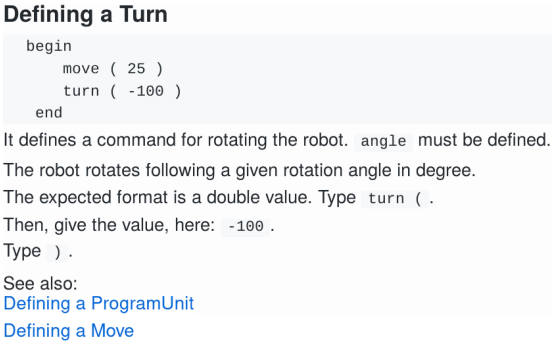


Figure 3.13 – Documentation de la classe *Turn* générée à partir du diagramme modifié manuellement de la Figure 3.11.

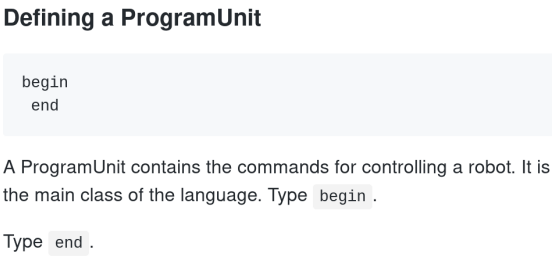


Figure 3.14 – Documentation du concept *ProgramUnit* de *Robot*

"*Defining a ProgramUnit*" pour le diagramme d'activité généré de *Robot*. Les éléments du métamodèle associés à cette activité (*i.e.,* la classe *ProgramUnit*) sont obtenus par consultation de la configuration du FM associée à l'activité (Ligne 5). En utilisant ces éléments du métamodèle, un *slicer* est utilisé pour obtenir la portion minimale du métamodèle utilisant les éléments de l'activité (Ligne 7). Un second *slicer* est ensuite utilisé pour obtenir un modèle contenant uniquement les éléments de la portion de métamodèle (Ligne 8). Le modèle concret sera utilisé pour expliquer le concept courant. Par exemple avec *ProgramUnit*, le modèle utilisé en entrée est celui donné à l'extrait de code 3.3. Le modèle tronqué qui n'utilise que *ProgramUnit* est montré en Figure 3.14. Seules les instructions *begin* et *end*, correspondant à *ProgramUnit*, sont gardées. Pour ce faire, la grammaire de *Robot* est aussi tronquée afin de garder seulement les règles nécessaires (Ligne 14) :

```
ProgramUnit : 'begin' 'end';
```

Le texte de la documentation de concept est alors produit en utilisant le métamodèle et la grammaire tronqués (Ligne 9). Les règles de la grammaire sont analysées afin de fournir progressivement à la fois les instructions de programmation et le texte extrait de la documentation du métamodèle. Par exemple sur la documentation représentée en Figure 3.14, la première ligne "*A* ProgramUnit *[...] the language*" vient de la documentation du métamodèle. Les instructions de programmation "*Type* begin. *Type* end." sont générées à partir de la grammaire tronquée.

Les activités sont traitées récursivement (Lignes 10 et 11). Par exemple l'activité "Defining a *Turn*" suit l'activité "Defining a *ProgramUnit*". La documentation générée de *Turn* est présentée en Figure 3.12. Le modèle en entrée est maintenant tronqué à partir des classes *ProgramUnit* et *Turn*. Le *slicer* produit un modèle contenant une instance de chaque concept. Une commande *turn(-100)* est maintenant présente à

Algorithm 4 Génération de la documentation

```
Require: metamodel, grammar, models, activityDiagram
 1: rootActivity := getRootActivity(activityDiagram)
 2: produceDoc(rootActivity, prevElts, metamodel, models, grammar)
 3:
 4: procedure PRODUCEDOC(activity, mm, models, grammar)
 5:     elements := findElements(activity.getConfiguration(), mm)
 6:     inputSlicer := elements
 7:     slicedMM := sliceMetamodel(mm, inputSlicer)
 8:     slicedModel := findAndSliceModel(slicedMM, models)
 9:     produceMarkdownText(slicedMM, slicedModel, prevElts, grammar)
10:     for each nextActivity in activity.getOutputs() do
11:         produceDoc(nextActivity, mm, models, grammar)
12:
13: procedure PRODUCEMARKDOWNTEXT(slicedMM, model, grammar)
14:     codeExample := fromXMItoConcreteSyntax(model)
15:     actions := extractActions(slicedMM, grammar)
16:     text := toText(actions, codeExample)
```

l'intérieur du bloc *begin/end*. La documentation contient une référence vers les activités précédentes, *i.e.,* "Defining a *ProgramUnit*" dans le cas présent. La documentation est générée au format *Markdown* et aussi sous forme de fragments Xtext utilisables pour une documentation contextuelle en éditeur.

2.3 Implémentation

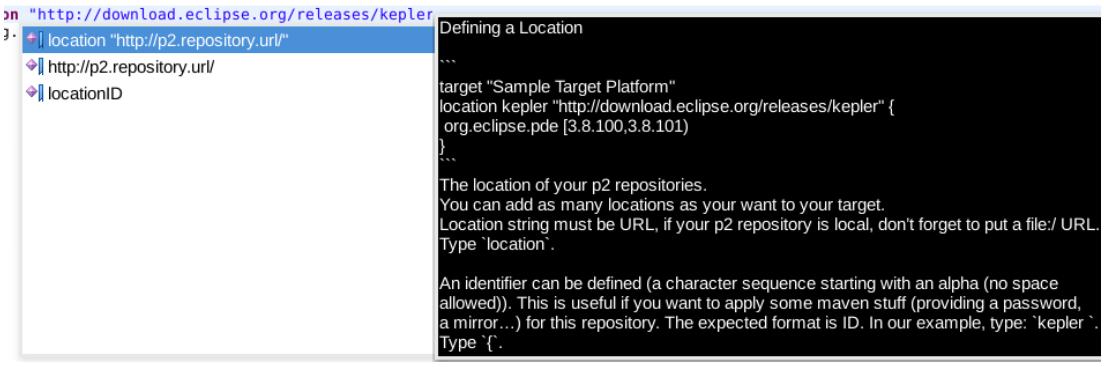


Figure 3.15 – Exemple de documentation contextuelle avec un éditeur Xtext.

Nous avons implémenté la LPLOS en un outil nommé *Docywood*, développé à partir du *framework* EMF (*Eclipse Modeling Framework*) [Steinberg et al., 2008] et de Xtext [Bettini, 2013] qui permet de définir des syntaxes concrètes pour des DSL. La méthode de *slicing* est réalisée par l'utilisation du DSL Kompren [Blouin et al., 2015, Blouin et al., 2011]. Notons que le FM n'est pas implémenté explicitement dans *Docywood*, *i.e.,* il n'y a pas de structure de données correspondant au FM dans le code de *Docywood*. Malgré cela, *Docywood* reste conforme à la LPLOS, notamment grâce au fait que le diagramme d'activité généré et les informations qu'il porte sont

suffisantes.

Nous revenons maintenant sur les quatre propriétés définies en Section 2.1.2 et nous évaluons analytiquement la manière dont notre LPLOS répond à ces propriétés.

Propriété #1 – documentation exhaustive – la LPLOS proposée repose sur un critère de couverture afin de respecter la propriété #1. Cependant dans le cas de notre implémentation, il n'y a pas de documentation générée pour toutes les références du métamodèle. Pare exemple dans le cas de *Robot*, la documentation n'est pas générée pour la référence *WhileNoObstacle.commands* car *Commands* a déjà une documentation générée par l'outil. C'est un choix de design pour *Docywood* qui peut être changé dans une future version.

Propriété #2 – contextualisation de la documentation – *Docywood* permet la génération de documentation contextuelle Xtext utilisable en direct lors d'une tâche de développement sur un éditeur supportant le DSL en question. De plus, la documentation générée en *Markdown* est constituée de plusieurs petits blocs expliquant chacun un concept différent.

Propriété #3 – documentation sous plusieurs formes– maintenir une documentation se présentant sous plusieurs formes à des endroits différents est un problème courant que rencontrent les concepteurs de DSL. Notre outil permet de générer deux formes de documentation (*Markdown* et Xtext) à partir d'un même métamodèle. Une mise à jour de la documentation permet alors de générer de nouveau les deux formes de documentation de manière synchronisée. Cependant, *Docywood* écrase les modifications apportées par le concepteur du DSL au diagramme d'activité si celui-ci est généré de nouveau. Dans une future version, ce problème pourrait être évité en proposant au concepteur du DSL de modifier le diagramme de manière minimaliste : les activités et transitions non concernées par des changements ne seraient pas modifiées.

Propriété #4 – documentation reposant sur des exemples – c'est le cas de la documentation produite par *Docywood*. Une limite de notre outil est que le modèle concret fourni pour servir d'exemple doit jouer correctement son rôle, il doit donc être choisi avec soin par le concepteur du DSL. Ce modèle doit être simple et suffisamment explicite pour pouvoir être compris par les utilisateurs de la documentation.

3 Synthèse du chapitre

Dans ce chapitre, nous avons présenté le concept de LPLOS. Cette approche permet la conception d'outils de production de logiciels reposant sur un scénario. Elle prend en entrée un modèle de domaine et les composants logiciels associés. Ensuite, le concepteur doit produire une spécification de scénarios. Celle-ci doit être conforme à un modèle de scénarios, qui doit lui-même supporter une relation d'équivalence avec le modèle de scénarios de référence de l'approche : le formalisme des FSM UML. Chaque état correspond à une configuration d'un FM extrait du modèle de domaine fourni. Cette extraction peut être automatique ou manuelle. De plus, une première spécification de scénarios peut être obtenue automatiquement si le cas d'utilisation s'y prête. La synthèse d'un logiciel est ensuite faite par mise en relation de la spécification de scénarios avec un modèle instanciant le modèle du domaine et les composants logiciels. Chaque spécification de scénarios qui peut être produite constitue ainsi la spécification d'une

nouvelle version du logiciel à produire.

Notre approche visait à répondre à la limite **L2** identifiée dans l'état de l'art (Chapitre 2, Section 3.2) : les SPL et les FM ne permettent pas de gérer la variabilité de scénarios au nombre de configurations potentiellement infini. Pour cela, elle devait remplir l'objectif **O4-SPL-SCÉNARIO** défini en introduction (Chapitre 1) : l'approche proposée doit appliquer le principe de développement de familles de logiciels des SPL, d'une manière adaptée à la production de logiciels reposant sur un scénario, comme les applications de RV. Le modèle de scénario définissant des états correspondant chacun à une configuration d'un FM représentant le domaine permet cela.

Nous avons présenté un cas d'utilisation décrivant une application de l'approche. Ce cas d'utilisation consiste à générer de la documentation pour des DSL. Cette documentation explique les concepts uns-à-uns. Elle respecte quatre propriétés requises pour une documentation utilisable :

1. exhaustivité ;
2. adaptée au contexte d'utilisation ;
3. facile à maintenir (notamment sur plusieurs plateformes) ;
4. s'appuyant sur des exemples.

Dans ce cas d'utilisation, la spécification de scénarios est un diagramme d'activité décrivant l'ordre dans lequel les concepts du DSL doivent être expliqués. Une version par défaut de ce diagramme peut-être générée puis personnalisée.

Dans le chapitre suivant, nous présentons le concept de LPLRV. Cette approche repose sur le concept de LPLOS. Les applications de RV constituent en effet une grande famille de logiciels reposant sur le concept de scénario. L'approche LPLRV précisera donc comment ce domaine met en œuvre les différents concepts présents dans l'approche LPLOS.

Lignes de Produits Logiciels pour la RV

4

Ce chapitre présente le concept de Lignes de produits Logiciels pour la RV (LPLRV). Cette approche est une adaptation du concept de Ligne de Produits Logiciels Orientés Scénario (LPLOS) au domaine de la RV.

L'approche LPLOS a été présentée au Chapitre 3. Elle permet la production semi-automatique de logiciels définis à partir d'une spécification de scénarios. Cette production se fait par mise en relation de composants logiciels réutilisables avec une spécification de scénarios. Un modèle du domaine permet de faire le lien entre les composants logiciels et les concepts manipulés par la spécification de scénarios.

La Section 1 présente une vue d'ensemble de l'approche LPLRV. La Section 2 présente un premier cas d'utilisation : l'outil AGENT permettant la génération automatique de protocoles expérimentaux en RV. La Section 3 présente un second cas d'utilisation : l'outil Tremplin permettant la génération semi-automatique d'applications de RV pour la formation aux procédures chirurgicales.

1 Approche

Cette section présente en détail l'approche LPLRV. La Section 1.1 présente un cas d'illustration que sera utilisé pour expliquer l'approche tout au long de la section. La Section 1.2 présente une vue d'ensemble de l'approche LPLRV. L'approche est présentée en détail en Sections 1.3 à 1.5.

1.1 Cas d'illustration

L'exemple utilisé pour expliquer l'approche LPLRV repose sur l'exemple du robot télécommandé que nous avons introduit au Chapitre 3, Section 1.3. Dans cet exemple, notre but est de produire une application de RV dans laquelle nous pourrons contrôler un robot virtuel. Une spécification de scénarios permettra de simuler une séquence d'appuis sur les boutons de la télécommande.

1.2 Vue d'ensemble

Le but de l'approche LPLRV est de produire une application de RV à partir du modèle du domaine. Pour rappel, le modèle du domaine est une abstraction définissant les concepts qui caractérisent la famille de logiciels à produire. Une vue d'ensemble

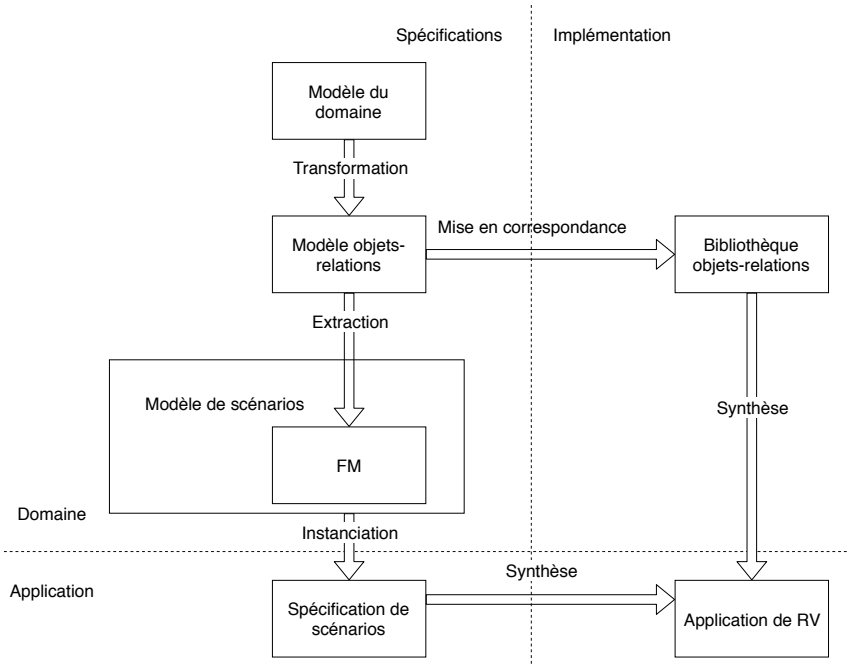


Figure 4.1 – Structure et fonctionnement d'une LPLRV.

de l'approche est représentée en Figure 4.1. Elle reprend la structure de LPLOS avec quelques ajouts et précisions. Ainsi, le modèle du domaine doit toujours être fourni en entrée, mais il passe par une étape de transformation vers un modèle objets-relations. Les modèles objets-relations ont été présentés dans l'état de l'art (Chapitre 2, Section 2.1.2). Un modèle objets-relations permet de décrire les objets virtuels qui composent une application de RV en se fondant sur leurs caractéristiques et les relations qui les lient entre-eux, *e.g.,* les interactions possibles. Ce modèle sert de point de départ pour générer les composants logiciels, qui prennent la forme d'une bibliothèque objets-relations. Ces éléments sont détaillés en Section 1.3.

Comme pour l'approche LPLOS, le modèle de scénarios permet de définir des spécifications de scénarios qui changent la configuration du FM. Ce dernier est obtenu à partir du modèle objets-relations. Nous détaillons le modèle de scénarios et l'obtention du FM à partir du modèle objets-relations en Section 1.4.

La dernière étape correspond à la phase de synthèse [1] de l'application de RV. Dans le cadre de l'approche LPLRV, elle consiste à intégrer la spécification de scénarios et la bibliothèque objets-relations à un projet d'application de RV (*e.g.,* un projet Unity3D). Il s'agit donc d'une synthèse par intégration (voir Chapitre 3, Section 1.1). Cette dernière étape est présentée en Section 1.5.

1.3 Définition du domaine

Tout comme dans l'approche LPLOS, le modèle du domaine doit être fourni. Dans le cas d'illustration, le modèle est celui de la Figure 3.2. Ce modèle doit être transformé

1. La notion de synthèse est définie au Chapitre 3 en Section 1.1 comme la phase durant laquelle les composants logiciels et la spécification de scénarios sont mis en relation pour produire le logiciel.

pour prendre une forme plus adaptée pour décrire les objets et comportements qui peuplent un Environnement Virtuel (EV) : le formalisme objets-relations (voir Chapitre 2, Section 2.1.2). La traduction du modèle de domaine en modèle objets-relations est présentée en Section 1.3.2. Le modèle objets-relations doit ensuite être mis en correspondance avec son implémentation, *i.e.,* la bibliothèque objets-relations. Cette mise en correspondance est présentée en Section 1.3.3. Elle consiste à établir un lien entre les composants logiciels de la bibliothèque objets-relations (*e.g.,* les différents objets virtuels) et leur abstraction dans le modèle objets-relations. Elle peut par exemple relier un objet virtuel avec l'abstraction d'objet du même nom dans le modèle objets-relations.

1.3.1 Modèle du domaine

La définition du modèle du domaine suit les mêmes principes que pour l'approche LPLOS (voir Chapitre 3, Section 1.4.1). Le modèle doit donc représenter fidèlement les concepts qui caractérisent la famille d'applications de RV à produire. Il est préférable de choisir un modèle existant pour favoriser la réutilisation, plutôt que de définir un modèle spécifique à une LPLRV. Les choix de modélisation dépendent aussi du contexte d'utilisation de la LPLRV. Par exemple, la LPLRV présentée en Section 2 de ce chapitre repose sur l'utilisation d'un DSL ; la syntaxe graphique de ce DSL est utilisée pour définir ce modèle. Nous présentons également en Section 3 un cas dans lequel une ontologie est utilisée en tant que modèle du domaine. La raison principale en est que cette ontologie est complète et est déjà utilisée dans d'autres contextes.

L'utilisation du modèle du domaine dans l'approche LPLRV est dans le cas général moins contraignante que pour l'approche LPLOS. En effet, le modèle du domaine n'est pas utilisé directement, mais il est d'abord transformé en modèle objets-relations. C'est le modèle objets-relations qui est utilisé pour mettre en relation les composants RV (*i.e.,* la bibliothèque objets-relations) et une spécification de scénarios.

1.3.2 De la modélisation du domaine au modèle objets-relations

Un modèle objets-relations est composé d'une structure hiérarchique liant objets, types et états, ainsi que d'un ensemble de relations entre ces objets. Ainsi, chaque objet est composé d'un ou plusieurs types. Chaque type est défini comme un ensemble d'états que peuvent prendre les objets associés à ce type. Une relation peut se définir comme une transformation d'objets en modifiant leur état, en créant de nouveaux objets ou en en détruisant. Une relation est définie sur les types des objets qu'elle peut impacter et son application à un ensemble effectif d'objets se nomme une réalisation. Par exemple le diagramme UML de la Figure 4.2 définit une relation *Taper* impliquant un objet de type *Tapeur*, un objet de type *Tapé* et un *Acteur* réalisant l'action. Tout objet ayant la capacité de taper (*Tapeur*) peut ainsi entrer en jeu dans la relation, *e.g.,* un marteau ou un maillet. De même, tout objet pouvant être tapé (*Tapé*) peut entrer en jeu dans la relation, *e.g.,* un clou ou un piquet. Cet exemple illustre le caractère générique des relations : elles permettent de mettre en relations des objets différents, pourvus qu'ils aient une capacité spécifique.

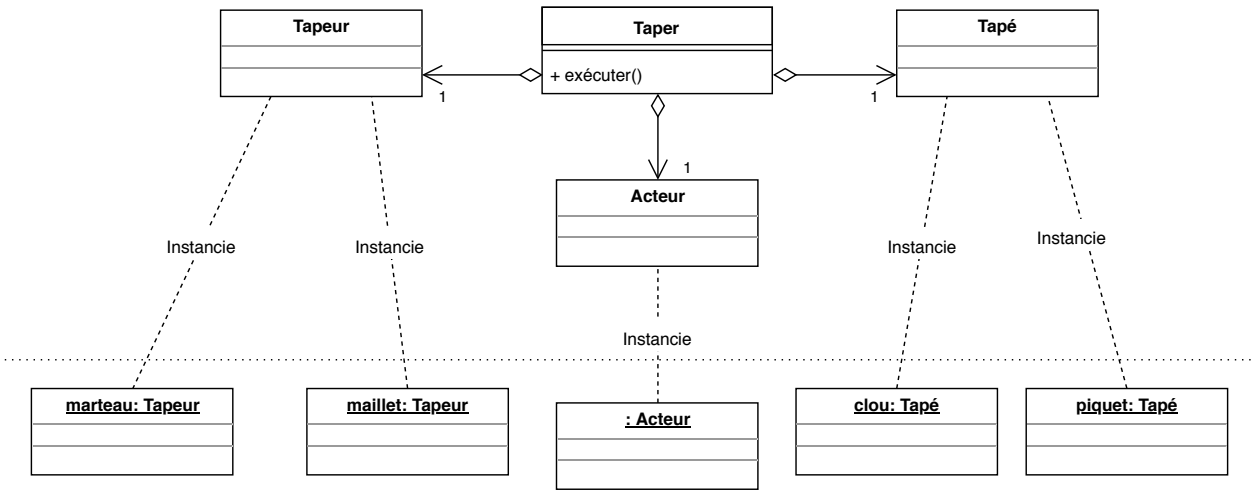


Figure 4.2 – Exemple de relation définie en UML. Les aggrégations expriment une dépendance relationnelle entre les classes *Tapeur*, *Tapé*, *Acteur* et *Taper*.

Définition d'un modèle objets-relations *Un modèle objets-relations* $MOR = \langle O, T, Objet, E, E_{\perp\top}, C, A, R, r_{activer}, r_{désactiver} \rangle$ *est une structure définie comme suit :*

- O *est l'ensemble des objets ;*
- T *est l'ensemble des types, chaque type pouvant définir un ensemble de champs (comme les classes UML) ;*
- $Objet \notin T$ *est un type spécial porté par tous les objets de* O *qui permet de spécifier les caractéristiques communes à tous les objets. Dans notre approche, un objet est simplement une entité 3D qui peut être présente ou non dans un EV à un instant donné ;*
- E *est l'ensemble des états ;*
- $E_{\perp\top} = \{\perp, \top\} \notin E$ *comporte deux états spéciaux attachés au type Objet et permettant respectivement d'encoder par défaut que l'objet en question est désactivé ou activé,* i.e., *absent ou présent dans l'application de RV ;*
- $C : O \mapsto \mathcal{P}(T)$ *est une application définissant une composition de types pour chaque objet ;*
- $A : T \mapsto \mathcal{P}(E)$ *est une application associant un ensemble d'états à chaque type, telle que* $\forall t \in T, |A(t)| \geq 1$. *Tout type* t *tel que* $|A(t)| = 1$ *possède un unique état symbolique* ε *qui exprime le fait qu'un objet ne peut pas changer d'état par rapport à ce type ;*
- R *est l'ensemble des relations, une relation* $r \in R$ *étant une structure* $r = \langle V_t, V_e, P \rangle$ *telle que :*
  - $V_t = (t_0, ..., t_n)$ *et* $V_e = (e_0, ..., e_n)$ *sont des vecteurs tels que* $\forall i \in [0, n], t_i \in T \cup \{Objet\}$ *et* $(t_i \in T \Rightarrow e_i \in A(t_i)) \wedge (t_i = Objet \Rightarrow e_i \in E_{\perp\top})$, *qui représentent respectivement les types sur lesquels est définie la relation et les états que doivent prendre les objets correspondants lors de la réalisation de la relation ;*

  - $P \subseteq \mathcal{P}(\{0, ..., n\})$ *est un ensemble tel que* $\forall p \in P, \forall i, j \in p, \exists o \in O$ *tel que* $t_i, t_j \in C(o)$ *; chaque élément de* P *est un ensemble d'indices qui indique que les types de* V_t *correspondants à ces indices doivent être liés au même objet lors de la réalisation de la relation ;*
- *pour une telle relation* r*, on peut définir sa réalisation* $\rho = \langle (o_0, ..., o_n), (t_0, ..., t_n), (e_0, ..., e_n), P \rangle$*, où* $\forall i \in [0, n], o_i \in O$ *et* $t_i \in C(o_i) \cup Objet$ *; le respect des contraintes définies par* P *implique alors que* $\forall p \in P, \forall i, j \in p$ *alors* $o_i = o_j$ *;*
- $r_{activer}, r_{désactiver} \in R$ *sont deux relations spéciales permettant respectivement d'activer et de désactiver les objets, on a* $r_{activer} = \langle (Objet), (\top), \emptyset \rangle$ *et* $r_{désactiver} = \langle (Objet), (\bot), \emptyset \rangle$*. Leur intérêt est de pouvoir représenter la création et la destruction d'objets.*

La Figure 4.3 représente le modèle objets-relations pour l'exemple du robot, dérivant du modèle de la Figure 3.2. Le concept *Robot* est transformé en objet, qui sera le seul de notre EV. Afin de représenter les commandes du robot, nous lui avons associé deux types : *Avançant* et *Tournant*. Un objet de type *Avançant* est un objet qui peut se trouver dans un état de marche ou dans un état d'arrêt (*resp. on* et *off*). En état de marche, un tel objet avance à une vitesse constante. A l'arrêt, il n'avance pas (sa vitesse est nulle). Ce type caractérise les objets qui le portent par une position dans le plan, décrite par les deux dimensions x, y. Un objet de type *Tournant* peut être dans l'un des trois états *horaire*, *anti-horaire* ou *off*. Si l'objet de type *Tournant* se trouve dans l'un des états *horaire* ou *anti-horaire*, cela signifie qu'il tourne à vitesse constante dans le sens spécifié. Dans l'état *off*, il ne tourne pas (sa vitesse angulaire est nulle). Ce type caractérise les objets qui le portent par une position angulaire dans le plan de rotation, définie par la valeur d'un *angle*. Nous définissons les relations :

- $avancer = \langle (Avançant), (on), \emptyset \rangle$, qui permet de mettre un objet de type *Avançant* en état de marche ;
- $stopAvancer = \langle (Avançant), (off), \emptyset \rangle$, qui permet de mettre un objet de type *Avançant* en état d'arrêt ;
- la relation paramétrique $tourner(sens) = \langle (Tournant), (sens), \emptyset \rangle$, où $sens \in \{horaire, antiHoraire\}$, qui permet de mettre un objet de type *Tournant* en rotation dans le sens indiqué ;
- $stopTourner = \langle (Tournant), (off), \emptyset \rangle$, qui permet de mettre un objet de type *Tournant* en état d'arrêt.

Il n'existe pas de méthode systématique permettant de transformer un modèle de domaine en un modèle objets-relations. En effet, un modèle objets-relations suit une approche de modélisation entités-composants, alors que les modèles de domaines (UML ou ontologies) suivent une modélisation par hiérarchie de classes et compositions. A notre sens, ces deux approches sont deux paradigmes différents entre lesquels il n'existe pas d'équivalence dans le cas général, même s'il existe des heuristiques qui peuvent faciliter le passage de l'un à l'autre. Par exemple, les objets correspondent en général aux instances des classes du métamodèle du domaine. Les types de ces objets correspondent en général aux classes ainsi instanciées, ainsi qu'aux classes parentes de ces classes. Cependant la composition peut être interprétée de plusieurs façons. Par exemple, la Figure 4.4 présente un modèle de domaine associant la classe *Roue* par composition

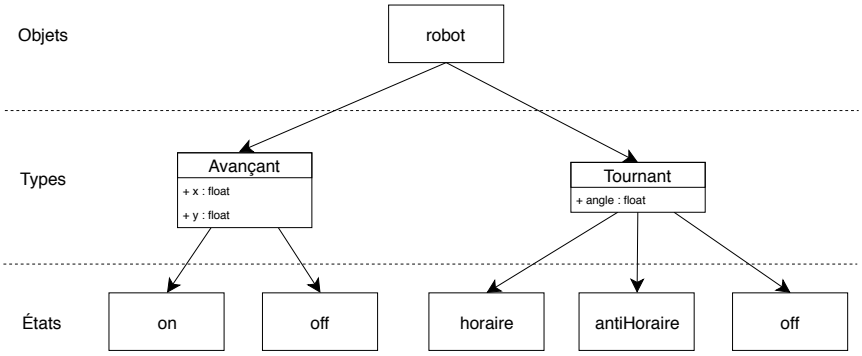


Figure 4.3 – Hiérarchie objets-types-états du modèle objets-relations pour le robot. Le type spécial *Objet* n'est pas représenté.

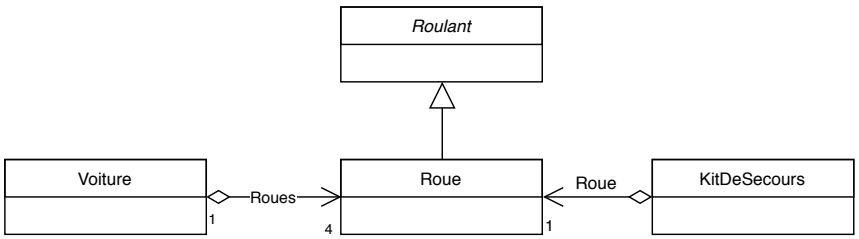


Figure 4.4 – Exemple de diagramme de classes UML. La composition *Roues* permet d'associer aux instances de la classe *Voiture* le type *Roulant*. En revanche, le même raisonnement est faux pour la composition *Roue* : un kit de secours qui contient une roue (de secours) ne peut pas rouler, bien qu'une roue le puisse.

aux classes *Voiture* et *KitDeSecours*. La classe *Roue* hérite d'une classe *Roulant*. Il serait logique d'associer aux instances de *Voiture* le type *Roulant*, par composition. En effet, une voiture peut rouler car elle est composée de roues, une roue pouvant rouler "par elle-même". En revanche, le même raisonnement ne marche pas pour la composition de *KitDeSecours* vers *Roue*. Ce raisonnement vient directement de la connaissance que nous avons du domaine et n'est pas retranscrite sur le modèle de domaine. C'est un exemple qui montre que malgré l'existence d'heuristiques, il n'est pas possible d'établir d'équivalence rigoureuse entre modèles de domaine et modèles objets-relations dans le cas général. Cette transformation doit donc se faire par collaboration de l'expert du domaine et du concepteur de la LPLRV.

Notons que le modèle objets-relations ne définit pas de liaisons (relation, composition ou agrégation) entre les types. En effet, comme vu précédemment, ces concepts peuvent prêter à de nombreuses interprétation différentes. Nous distinguons au moins deux motivations amenant à définir une liaison en modélisation orientée-objet : (1) la liaison structurelle et (2) la liaison relationnelle. La liaison structurelle indique qu'un objet fait partie d'un autre. C'est le cas des liaisons présentées en Figure 4.4 qui indiquent qu'une voiture est composée de quatre roues et qu'un kit de secours est composé d'une roue. La liaison relationnelle indique qu'il existe une interdépendance entre deux concepts. La Figure 4.2 donne un exemple d'une telle liaison.

En RV, la différence entre ces deux sortes de liaisons est fondamentale. En effet, dans les éditeurs (*e.g.,* Unity3D), la liaison structurelle se traduit par des relations hiérarchiques entre des objets d'une scène 3D, comme l'illustre la Figure 4.5. La

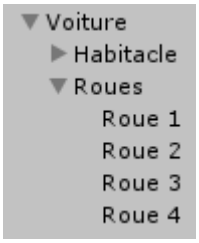


Figure 4.5 – Exemple de hiérarchie Unity3D décrivant la structure d'une voiture.

liaison relationnelle peut se traduire sous différentes formes, *e.g.,* par l'utilisation d'attributs dans les classes. Dans notre approche du paradigme objets-relations, la liaison structurelle n'est pas représentée. Cette tâche est déléguée au concepteur de l'application de RV qui la réalisera par l'utilisation d'un éditeur. La liaison relationnelle se fait par la définition de relations.

1.3.3 Mise en relation avec la bibliothèque de composants RV

Le modèle objets-relations doit être implémenté sous forme de bibliothèque objets-relations. Nous n'imposons pas de technologies en particulier mais nous prendrons l'exemple d'une traduction du modèle en modèle #FIVE [Bouville et al., 2015] pour une intégration sur Unity3D [2]. Nous allons reprendre les éléments qui composent un modèle objets-relation et expliquer comment ils peuvent être traduits en concepts #FIVE / Unity3D :

— les objets sont traduits en instances de *Prefabs* Unity3D, *i.e.,* des prototypes d'objets 3D qui peuvent être instanciés ;

— les types sont traduits en types #FIVE. Un type #FIVE est fondamentalement un script Unity que l'on attache au *Prefab* de chaque objet qui porte ce type ;

— La notion d'état n'existe pas en #FIVE. Les états peuvent être traduits en attributs des types #FIVE ;

— Les relations sont traduites en relations #FIVE. Une relation #FIVE est un script comportant un attribut pour chaque type affecté par la relation, une méthode *IsRunnable* vérifiant les conditions préalables à la réalisation et une méthode *Run* contenant le code s'exécutant lors de la réalisation.

La Figure 4.6 représente l'implémentation de la relation *avancer*.

2. Un projet Unity3D est composé de plusieurs scènes. Une scène est un environnement 3D que l'on peut peupler d'objets 3D. Elle est décrite par un graphe de scène, *i.e.,* une hiérarchie parent-enfants d'objets 3D. Unity3D fonctionne selon le paradigme entités-composants : le développeur peut attacher des composants à chaque objet de la hiérarchie. La nature de chaque objet est déterminée par ses composants. Ainsi, chaque objet possède au moins un composant *Transform* qui décrit sa position dans l'espace par rapport à son parent. Certains composants permettent de décrire l'apparence de l'objet (*e.g.,* sa forme, sa couleur), d'autres ses propriétés physiques (*e.g.,* sensible à la gravité, boîte de collision), *etc.* Il est également possible d'attacher des scripts aux objets pour programmer des comportement spécifiques, *e.g.,* pour programmer les événements liés à l'objet lorsqu'il rentre en collision avec un autre, *etc.* Tout composant ou objet en Unity3D hérite de la classe *GameObject*.

```
// Relation Avancer
0 references
public class Avancer : UFRelation
{
    // Référence au type Avançant
    [UFObjectPattern("Robot", "Avançant")]
    public Avançant av;

    // Condition préalable à la réalisation
    10 references
    public override bool IsRunnable()
    {
        return true;
    }

    // Code exécuté lors de la réalisation
    10 references
    public override void Run(Action resultCallback)
    {
        // Passage à l'état "on"
        av.On();
    }
}
```

Figure 4.6 – Implémentation de la relation *Avancer* en #FIVE.

1.4 FM et modèle de scénarios

Le rôle du modèle de variabilité (*Feature Model*, FM) est d'exprimer la variabilité intrinsèque au logiciel à produire. En l'occurrence, le FM doit permettre de représenter les états possibles des objets définis par le modèle objets-relations (voir Section 1.4.1). Le FM est généré à partir du modèle objets-relations (voir Section 1.4.2). Le modèle de scénario doit permettre de produire des spécifications de scénarios. Ce modèle doit, comme pour l'approche LPLOS, supporter au moins la notion de transition et la notion d'état. Chaque état de la spécification de scénarios correspond à une configuration possible du FM (voir Section 1.4.3).

1.4.1 Intérêt de la représentation du modèle objets-relations sous forme de FM

L'intérêt du FM est de permettre de définir les configurations possibles du modèle objets-relations. En effet, le modèle objets-relations est caractérisé par des objets qui peuvent apparaître dans l'EV, en disparaître, ou changer d'état. Ces changements permettent de définir un ensemble fini d'états possibles du modèle objets-relations, par combinaison des états possibles des objets qui le composent. Chaque état possible du modèle objets-relations correspond à une configuration du FM qui le représente.

Considérons par exemple le modèle objets-relations dont la hiérarchie objets-types-états est représentée en Figure 4.3. Nous y associons le FM de la Figure 4.7 (la procédure semi-automatique permettant d'obtenir ce FM à partir du modèle objets-relations est détaillée en Section 1.4.2). Les deux états *off* ont été différenciés pour éviter toute ambigüité : *aOff* pour le type *Avançant* et *tOff* pour le type *Tournant*. Pour le reste, la correspondance entre ces modèles est triviale. On peut par exemple représenter l'état de repos du robot par la configuration $c_1 = \{Racine, robot, Avançant, Tournant, aOff, tOff\}$. L'état du modèle objets-relations dans lequel le robot est en marche avant est représenté par la configuration $c_2 =$

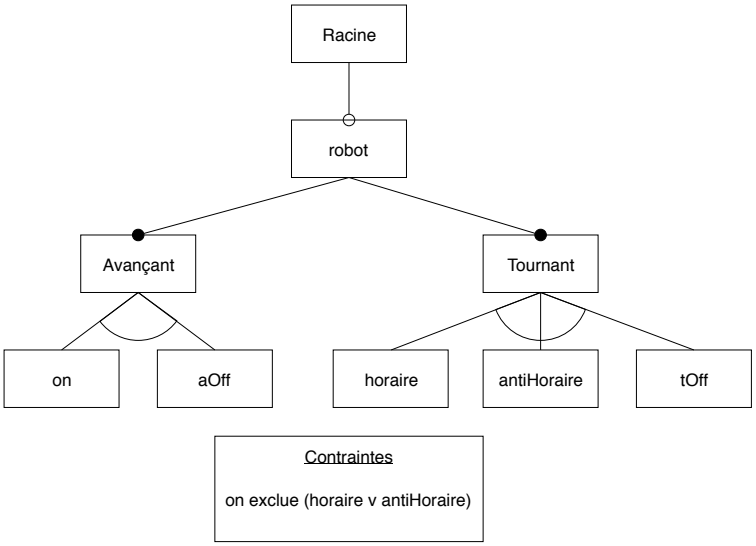


Figure 4.7 – FM représentant le modèle objets-relations du robot. La contrainte d'exclusion empêche celui-ci d'être à la fois en rotation et en translation vers l'avant.

$\{Racine, robot, Avançant, Tournant, on, tOff\}$.

Il apparaît alors qu'il existe une correspondance entre les relations et les changements de configuration. En effet, le passage de la configuration c_1 à la configuration c_2 correspond à la réalisation de la relation *avancer*, et on notera $avancer(c_1) = c_2$. Nous pouvons donc définir les réalisations comme des changements de configuration. Toute relation correspond soit à un changement d'état d'un objet, soit à l'activation ($r_{activer}$) ou à la désactivation ($r_{désactiver}$) d'un objet. Pour toutes les relations qui correspondent à un changement d'état, le changement de configuration correspondant consiste à sélectionner une *feature* dans un groupe XOR représentant les états possibles d'un objet relativement à un type. C'est le cas de la relation *avancer*. Pour les relations $r_{activer}$ et $r_{désactiver}$, le changement de configuration correspond à l'activation ou à la désactivation d'une *feature* représentant un objet. Par exemple, $r_{activer}(\{Racine\}) = \{Racine, robot, Avançant, Tournant\}$. On remarque que les configurations obtenues par réalisation ne sont pas forcément valides. Il peut être nécessaire de combiner les réalisations pour assurer la validité, *e.g.,* $r_{activer}(\{Racine\}) = \{Racine, robot, Avançant, Tournant\}$ n'est pas valide, mais $stopTourner(avancer(r_{activer}(\{Racine\}))) = \{Racine, robot, Avançant, Tournant, on, tOff\}$ l'est.

1.4.2 Du modèle objets-relations au FM

Le FM que nous souhaitons produire doit représenter les différents concepts présents dans le modèle objets-relations. Comme dans l'approche LPLOS, les états des spécifications de scénarios feront référence à une configuration du FM. Les transitions correspondront donc à un changement de configuration du FM, donc à une ou plusieurs réalisations de relations du modèle objets-relations. Les réalisations sont définies par rapport aux objets qu'elles impactent, aux états que doivent atteindre les objets et par rapport aux types liant ces objets et ces états. Les types n'étant liés qu'à un seul état

($|A(t)| = 1$) ne peuvent pas être concernés par une réalisation [3]. Cela implique deux choses pour le FM : (1) il n'est pas nécessaire de représenter les types n'étant liés qu'à un seul état et (2) mis à part ces types, les structures objets-types-états doivent être représentées explicitement.

La traduction du modèle objets-relations en FM se fait par l'Algorithme 5. Nous optons pour une approche simple consistant à reproduire la hiérarchie objets-types-états du modèle objets-relations. Ainsi, chaque objet est transformé en *feature* fille optionnelle de la racine (Ligne 7). La caractéristique optionnelle permet de modéliser les états $\bot$ et $\top$ associés au type *Objet* de l'objet. Chaque type d'un objet (autre qu'*Objet*) étant lié à strictement plus qu'un état (Ligne 10) est transformé en une *feature* fille obligatoire de cet objet (Ligne 11). Une *feature* correspondant à un type est obligatoire car dans notre approche, un type ne peut pas être détaché d'un objet. Enfin, chaque état est transformé en *feature* fille du type auquel il correspond, les états liés aux même types étant regroupés en un groupe XOR. En effet, les états sont exclusifs entre eux. Le résultat de la transformation du modèle objets-relations présenté en Figure 4.3 est le FM de la Figure 4.8.

Algorithm 5 Algorithme de transformation d'un modèle objets-relations en FM.

```
1:  procedure TRANSFORMATIONOBJETSTYPESVERSFM(modèleObjetsRelations)
2:      r := créerFeatureRacine()
3:      fm := créerFm(r)
4:      objets := obtenirObjets(modèleObjetsRelations)
5:      c := obtenirCompositionObjetsTypes(modèleObjetsRelations)
6:      a := obtenirAssociationTypesÉtats(modèleObjetsRelations)
7:      ajouterFeaturesFillesOptionnelles(r, objets)
8:      for each o ∈ objets do
9:          for each t ∈ c(o) do
10:             if |a(t)| > 1 then
11:                 ajouterFeatureFilleObligatoire(o, t)
12:                 ajouterGroupeXOR(t, a(t))
        return fm
```

Notons qu'une fois le FM produit, il est tout à fait possible de le retoucher à la main. Il est en particulier possible d'ajouter des contraintes logiques entre les *features* comme le permet le formalisme des FM, par exemple pour exprimer des contraintes qui existeraient entre les états de types différents. Il est aussi possible de proposer un autre FM, en particulier si celui-ci a déjà été réalisé et est compatible avec l'approche objets-relations.

Dans le cas de l'exemple du robot, nous souhaitons que le robot ne puisse pas être à la fois en rotation et en translation vers l'avant. Cela revient à ajouter la contrainte logique suivante, qui est une contrainte d'exclusion : $on \Rightarrow \neg(horaire \vee antiHoraire)$ (par contraposée, on obtient la contrainte d'exclusion équivalente $(horaire \vee antiHoraire) \Rightarrow aOff$). On obtient donc, après cette modification, le FM de la Figure 4.7.

3. Les types liés à un seul état peuvent être utiles du point de vue du modèle objets-relations, par exemple pour stocker des données relatives à l'objet ou définir des comportements qui ne sont pas gérés par les relations.

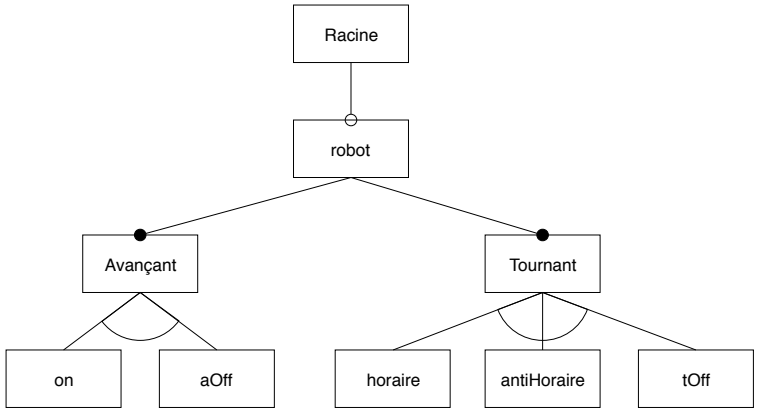


Figure 4.8 – FM dérivé du modèle objets-relations du robot (voir Figure 4.3).

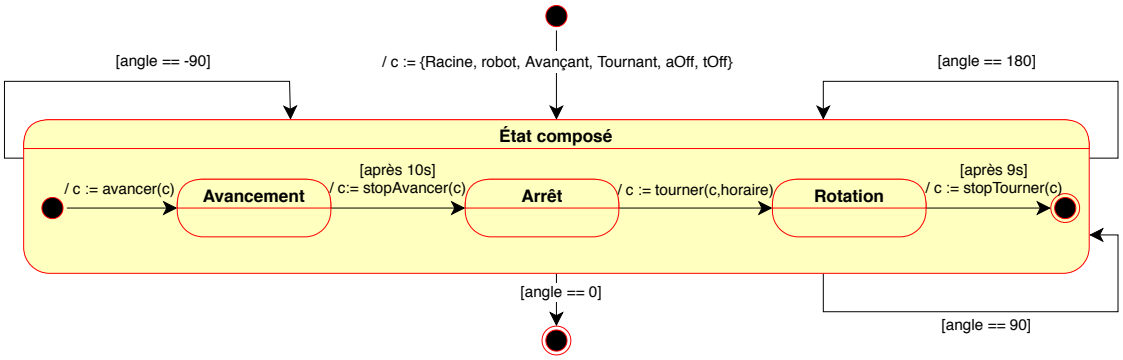


Figure 4.9 – Exemple de spécification de scénario pour le guidage d'un robot. Les actions associées aux transitions sont des réalisations. La spécification de scénario faire décrire au robot un carré de 10m *times* 10m. La vitesse de translation vers l'avant est de 1m/s et la vitesse angulaire est de 10°/s.

1.4.3 Modèle et spécifications de scénarios

Cette partie de l'approche LPLRV est identique à celle de l'approche LPLOS. Un seul détail change : comme nous l'avons vu, nous pouvons exprimer les changements de configuration sous forme de réalisations. Nous proposons donc d'adapter le formalisme de référence (les FSM UML) de la manière suivante : seul l'état initial sera annoté par une configuration (ce qui permettra de donner l'état initial de l'application de RV) et les transitions seront annotées par la réalisation correspondant au changement de configuration. La Figure 4.9 représente un exemple de spécifications de scénarios pour le guidage du robot.

1.5 Synthèse de l'application de RV

La synthèse de l'application de RV se fait par intégration du scénario et de la bibliothèque objets-relations à un projet de développement d'application de RV (*e.g.*, un projet Unity3D). C'est au développeur de fournir l'interpréteur de la spécification de scénarios établie. L'interpréteur devra notamment faire le lien entre les exécutions de réalisation telles qu'elles sont exprimées dans le scénario et les relations de la

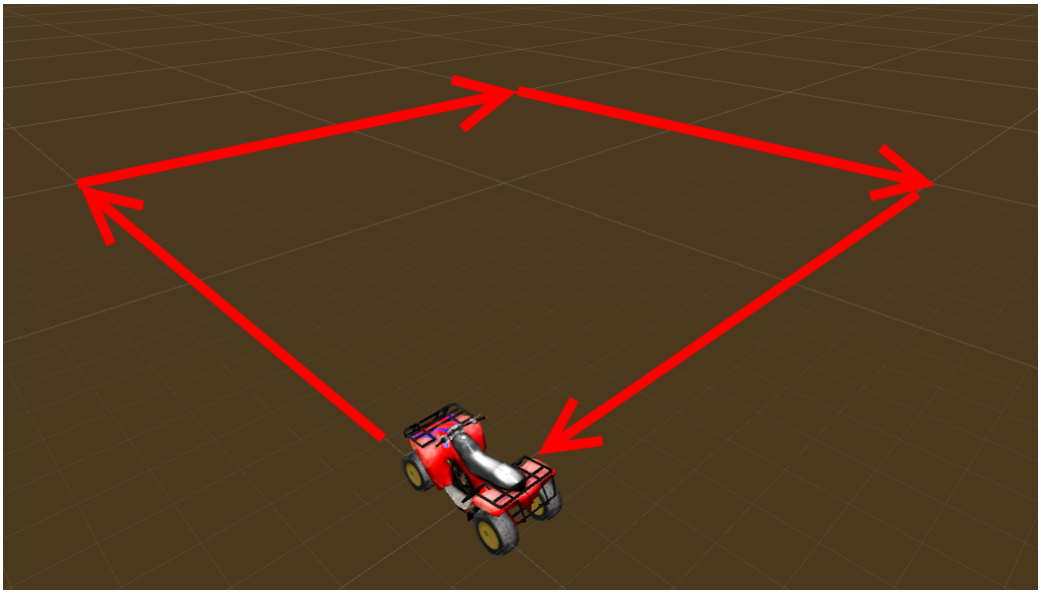


Figure 4.10 – Trajectoire du robot dans l'application de RV synthétisé, d'après le scénario de la Figure 4.9.

bibliothèque objets-relations. La Figure 4.10 représente un tel EV synthétisé, avec un robot et la trajectoire que celui-ci suit en exécutant le scénario de la Figure 4.9.

2 Cas d'utilisation : génération de protocoles expérimentaux pour la RV

Dans cette section, nous présentons un cas d'utilisation issu d'un de nos articles publiés au cours de la thèse [Le Moulec et al., 2017]. Nous présentons une LPLRV construite autour du DSL graphique AGENT, dédié à la génération automatique de protocoles expérimentaux en RV. AGENT a été développé grâce à l'outil *DSL Tools*[4], une extension de Visual Studio[5]. La Section 2.1 présente les motivations qui nous ont conduit à développer AGENT. La Section 2.2 résume la phase d'analyse qui nous a permis de concevoir AGENT. La Section 2.3 présente une vue d'ensemble de l'approche. La Section 2.4 présente un cas d'illustration qui sera utilisé tout au long de la section pour expliquer l'approche. Les Sections 2.5 à 2.8 détaillent le fonctionnement d'AGENT. La Section 2.9 résume les résultats que nous avons obtenu.

2.1 Motivations

L'un des intérêts des chercheurs en RV est l'analyse du comportement [Slater, 2009], de la perception [Interrante et al., 2006] et de l'interaction [Bowman et al., 1999] humaines en EV. Ces domaines de recherche nécessitent la conduite d'évaluations expérimentales afin de valider les hypothèses des chercheurs. Les évaluations

4. https://msdn.microsoft.com/en-us/library/bb126259.aspx
5. https://www.visualstudio.com

expérimentales permettent par exemple d'étudier les effets d'un système, d'un algorithme, d'une interface, *etc.* sur des utilisateurs. En effet, d'après Andrew Colman [Colman, 2015], "*les méthodes expérimentales [...] [permettent] une étude rigoureuse des relations de cause à effet*".

Les évaluations expérimentales doivent respecter un certain nombre de règles, *e.g.,* la définition des variables dépendantes et indépendantes et du protocole de l'expérience [Field and Hole, 2002]. Dans l'état de l'art, nous avons présenté différentes approches et outils ayant été développés afin de faciliter la production d'évaluations expérimentales en RV, qui constitue une tâche complexe et chronophage (Chapitre 2, Section 2.3). Dans l'ensemble, le constat est qu'il n'existe à ce jour qu'un seul outil adapté à la RV qui permette de faciliter ces développements : EVE [Grübel et al., 2016]. Cet outil, bien que complet, reste très dépendant de l'éditeur Unity3D et manque d'abstractions, de réutilisation et de séparation des préoccupations (*Separation of Concerns*, SoC). Autrement dit, cet outil n'exploite pas les possibilités qu'offre l'Ingénierie Dirigée par les Modèles (IDM) à ce sujet. C'est pourquoi nous avons proposé le langage dédié (*Domain-Specific Language*, DSL) graphique AGENT [Le Moulec et al., 2017]. Nous avons choisi de développer un DSL afin de proposer des notations et des concepts spécifiques au développement d'évaluations expérimentales en RV. AGENT permet de modéliser les conditions et le protocole d'une évaluation expérimentale, puis de générer celui-ci sous forme de code XML et C#. Nous présentons en quoi AGENT est une LPLRV et nous expliquons son fonctionnement.

2.2 Analyse du domaine

A notre connaissance, aucun modèle standard et formel n'a été proposé à ce jour pour représenter les évaluations expérimentales en RV. Dans cette section, nous proposons une telle formalisation sous forme de propriétés que doivent remplir les évaluations expérimentales en RV. Ce modèle est issu de l'analyse d'articles et posters publiés à l'occasion de la conférence internationale VRST 2016, où des évaluations expérimentales portant sur plusieurs sous-domaines de la RV sont présentées : formation, cinéma, perception des distances, interfaces utilisateurs 3D, appareils, gestion de la latence, haptique, *etc.* Nous nous sommes concentrés sur la conception d'évaluations expérimentales : les méthodes d'analyse statistique n'ont pas été étudiées. Nous considérons que les articles récents en RV publiés dans une conférence telle que VRST sont assez représentatifs de ce que l'on peut faire en terme d'évaluations expérimentales.

Notre principale observation a été que concevoir une évaluation expérimentale en RV peut être divisé en deux tâches majeures : (1) définition des conditions expérimentales (variables dépendantes et indépendantes) et (2) conception du protocole. Nous avons donc pu regrouper les propriétés identifiées en deux groupes, que nous présentons respectivement en Sections 2.2.1 et 2.2.2.

2.2.1 Variables

Il existe deux types de variables : les variables dépendantes et indépendantes (voir Chapitre 2, Section 2.3.1). Les variables dépendantes peuvent être quantitatives : elles peuvent par exemple correspondre à des mesures physiologiques issues de capteurs, *e.g.,* la mesure de la pression artérielle d'un sujet, mais aussi à la mesure du temps nécessaire pour réaliser une tâche, ou à la mesure de la précision d'un algorithme. Elles peuvent aussi être qualitatives, *e.g.,* un score attribué par un sujet à une technique d'interaction pour évaluer son utilisabilité ou le sentiment de présence qu'elle induit. Ces mesures qualitatives peuvent être recueillies à l'aide de questionnaires. Deux premières propriétés peuvent alors être émises :

P1 : les variables dépendantes correspondent à des données que l'on peut représenter sous diverses formes (entiers, nombres décimaux, booléens, notes, des types personnalisés, *etc.*)

P2 : les variables dépendantes correspondent à des données qui peuvent venir de trois sortes de sources différentes : des capteurs physiques, des mesures venant de l'EV et de questionnaires subjectifs.

Les variables indépendantes correspondent à ce que l'on doit évaluer, *e.g.,* métaphores, techniques d'interaction, algorithmes, matériel, *etc.* Ce sont donc des fonctionnalités physiques ou logicielles qui peuvent varier. Il existe deux façons de définir les variables indépendantes :

— par comparaison de plusieurs valeurs possibles pour une même variable, *e.g.,* on peut comparer plusieurs versions possibles d'une technique d'interaction pour déterminer laquelle est la plus efficace ;

— par l'étude de l'effet de la présence ou de l'absence d'une variable qui n'a qu'une valeur possible, *e.g.,* on peut comparer deux EV dans lesquels une métaphore est respectivement activée ou désactivée et en étudier l'influence sur le sentiment de présence des sujets.

Les variables indépendantes peuvent donc être assimilées à deux types : des variables booléennes (pour comparer la présence et l'absence) et des variables énumératives (pour comparer plusieurs valeurs possibles). Deux propriétés supplémentaires peuvent être ajoutées :

P3 : les variables indépendantes sont de deux types : des booléens et des énumérations.

P4 : les variables indépendantes et les différentes valeurs qu'elles peuvent prendre correspondent à des fonctionnalités physiques ou logicielles qui peuvent varier.

Toutes les valeurs possibles d'une même variable ne sont pas nécessairement présentées à tous les sujets d'une évaluation expérimentale. Dans certains cas, plusieurs groupes de participants exposés à des conditions différentes sont définis. Par exemple,

pour évaluer l'effet d'une nouvelle technique de navigation sur le sentiment de présence, deux groupes peuvent être mis en place : un groupe de témoin utilisant une technique de navigation dont l'effet est connu et un groupe utilisant la nouvelle technique de navigation. Ainsi, les conditions auxquelles sont exposés tous les sujets sont appelées "facteurs intra-sujets" et celles qui varient selon le groupe sont appelées "facteurs inter-sujets". Une propriété peut être émise :

P5 : les variables indépendantes inter-sujets imposent de concevoir un protocole expérimental différent pour chaque groupe de sujets.

2.2.2 Protocole

Le protocole expérimental est la séquence d'actions que les sujets réalisent durant une évaluation expérimentale. Nous considérons dans la suite que le protocole débute lorsque le sujet commence à utiliser l'EV de l'évaluation. S'il y a des facteurs inter-sujets et donc différents groupes, à chaque groupe est attribué un protocole. Ces protocoles diffèrent seulement au niveau des facteurs inter-sujets. A part cela, il sont identiques en tous points. C'est ce que précise la propriété **P6** :

P6 : les protocoles de chaque groupe diffèrent uniquement au niveau des facteurs inter-sujets.

En général, les protocoles sont organisées en deux phases : (1) une phase d'étalonnage et/ou d'entraînement et (2) d'une phase de mesure. Le rôle d'une phase d'entraînement est de familiariser les sujets à l'utilisation d'un EV, pour éviter d'introduire des biais dans les mesures. Dans ce type de phases, les sujets commencent à utiliser l'EV en s'exposant aux différentes conditions de l'évaluation expérimentale, sans que des mesures ne soient réalisées. Une phase d'étalonnage permet de configurer correctement le matériel utilisé.

La phase de mesures permet d'enregistrer les valeurs des variables dépendantes. Les sujets sont alors successivement exposés à différentes conditions qui sont répétées un nombre déterminé de fois et dans un ordre aléatoire. Une dernière propriété peut être émise :

P7 : une phase de mesure est en général précédée d'une phase d'étalonnage et/ou d'entraînement permettant au sujet d'apprendre à se servir du matériel mis à sa disposition, sans que les variables dépendantes ne soient prise en compte.

2.3 Vue d'ensemble

La Figure 4.11 présente une vue d'ensemble du fonctionnement du DSL AGENT. Comme le montre la figure, AGENT est une LPLRV par son fonctionnement. Le métamodèle d'AGENT est divisé en deux parties : la première partie décrit les conditions expérimentales et la seconde le protocole expérimental. La première partie du

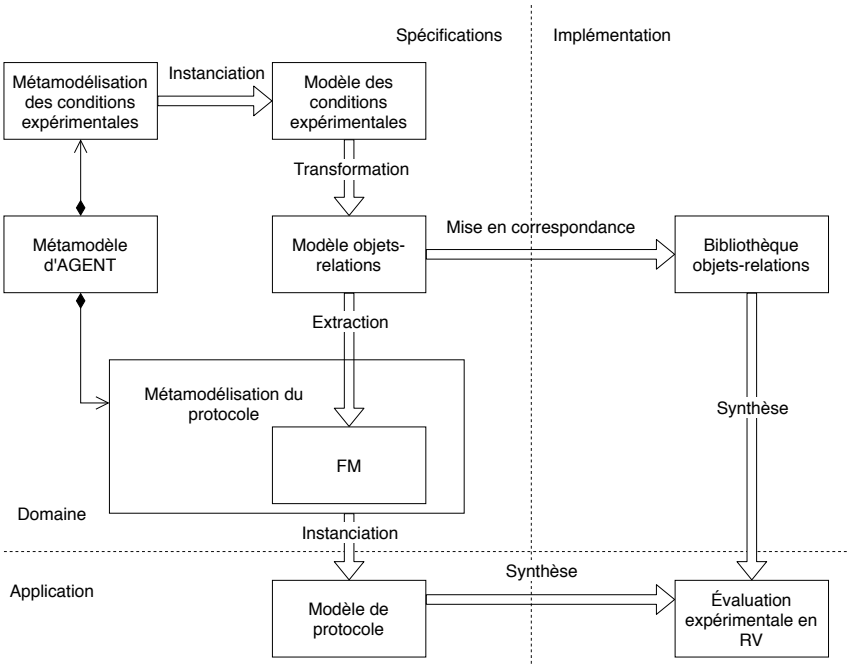


Figure 4.11 – Vue d'ensemble du fonctionnement du DSL AGENT, qui est aussi une LPLRV.

métamodèle permet de produire le modèle des conditions expérimentales, qui fait office de modèle du domaine (voir Section 2.5). Le modèle des conditions expérimentales est traduit sous la forme d'un modèle objets-relations, lui-même mis en correspondance avec une bibliothèque objets-relations dont le rôle est principalement d'associer les variables expérimentales aux composants hardware ou logiciels qui leur correspondent (*e.g.,* associer chaque valeur d'une variable indépendante à l'algorithme qui lui correspond, voir Section 2.6). La seconde partie du métamodèle constitue le modèle de scénarios. Les spécifications de scénarios produites correspondent donc à des protocoles d'évaluations expérimentales (voir Section 2.7). La compilation d'un modèle AGENT produit du code XML et C# qui peuvent être intégrés à un projet Unity3D, et mis en relation avec les composants de RV associés au modèle des conditions expérimentales (voir Section 2.8).

2.4 Cas d'illustration

Nous allons utiliser l'exemple présenté dans cette sous-section afin d'illustrer le fonctionnement d'AGENT. Considérons une évaluation expérimentale de RV au cours de laquelle chaque sujet doit réaliser une tâche t. Les variables indépendantes sont :

— I_b, une variable booléenne inter-sujets,

— I_e, une variable intra-sujet avec deux valeurs possibles : L_1 et L_2.

On a donc deux groupes de sujets G_b et $G_{\neg b}$ qui seront chacun exposés à un ensemble de conditions, respectivement $C_b = \{(I_b, L_1), (I_b, L_2)\}$ et $C_{\neg b} = \{(\neg I_b, L_1), (\neg I_b, L_2)\}$.

Les variables dépendantes sont :

— D_q, un ensemble de réponses à un questionnaire, sous forme de notes données sur

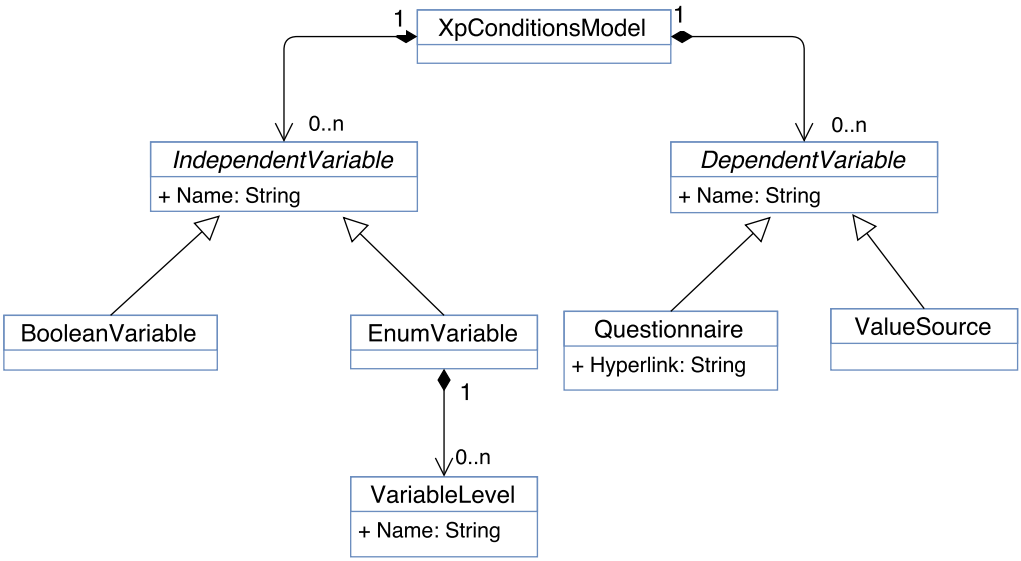


Figure 4.12 – Partie du métamodèle d'AGENT représentant les conditions expérimentales.

une échelle de Likert, pour évaluer t en fonction des valeurs possibles des variables indépendantes,

— D_s, la vitesse d'exécution de t.

Le protocole se déroule de la manière suivante :

1. phase d'entraînement : le sujet exécute 2×4 fois t, *i.e.,* 4 fois chaque condition de C_b ou $C_{\neg b}$, sans qu'aucune donnée ne soit enregistrée ;
2. phase d'acquisition des données : le sujet exécute 2×32 fois t, *i.e.,* 32 fois chaque condition de C_b ou $C_{\neg b}$, D_s étant enregistrée ;
3. phase d'évaluation subjective : le sujet répond au questionnaire.

2.5 Modèle du domaine, modèle objets-relations et FM

Dans cette section nous présentons la partie du métamodèle d'AGENT responsable de la modélisation des conditions expérimentales. Nous présentons également les modèles qui instancient cette partie du métamodèle.

2.5.1 Modélisation des conditions expérimentales

La partie du métamodèle présentée en Figure 4.12 permet de décrire les variables indépendantes et dépendantes. Un modèle est composé de variables indépendantes et dépendantes, *resp.* représentées par les classes *IndependentVariable* et *DependentVariable*. Les variables indépendantes peuvent être de type *BooleanVariable* ou *EnumVariable* (voir **P3**), les valeurs possible d'une *EnumVariable* étant représentées par la classe *VariableLevel*. Les variables dépendantes peuvent être de type *Questionnaire* ou *ValueSource* (voir **P2**), *i.e., resp.* subjectives ou objectives.

La Figure 4.13 présente le modèle des conditions expérimentales pour l'exemple de la Section 2.4. Les variables indépendantes booléennes sont représentées en vert. Les

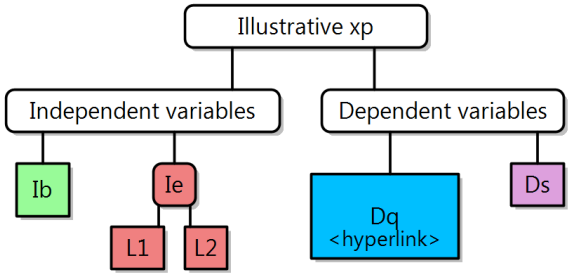


Figure 4.13 – Modèle des conditions expérimentales de l'exemple présenté en Section 2.4.

variables indépendantes de type "énumération", ainsi que leurs valeurs possibles sont représentées en rouge. Les variables dépendantes subjectives sont représentées en bleu et on peut spécifier un lien hypertexte pointant vers le questionnaire. Les variables dépendantes objectives sont représentées en mauve.

2.5.2 Modèle objets-relations et FM

Le modèle des conditions expérimentales est converti en un modèle objets-relations. La conversion de l'un à l'autre se fait de la manière suivante :

- les instances des classes *EnumVariable*, *BooleanVariable*, *Questionnaire* et *ValueSource* du métamodèle sont converties en objets,
- un type *IndependentVariable* est attaché aux objets dérivant des instances de *EnumVariable* et *BooleanVariable*. Il comporte un champ *assetName* permettant de lier la variable à un élément de la bibliothèque objets-relations représentant son implémentation, par son nom. Par exemple, si L_1 et L_2 représentent chacun une version différente d'un algorithme, le champ *assetName* portera le nom d'une ressource comportant les deux versions de l'algorithme. Par exemple sous Unity3D, cet élément peut être un *GameObject* auquel sont attachés les scripts permettant d'exécuter L_1 et L_2 ;
- de même, un type *ValueSource* et un type *Questionnaire* sont ajoutés pour les objets dérivant de variable dépendantes. Le champ *dataSource* permet de lier chaque variable dépendante à la source produisant les données à mesurer. Le champ *hyperlink* permet d'associer un lien hypertexte à un questionnaire ;
- à chaque instance de la classe *EnumVariable* est associé un type qui lui est propre (*e.g.,* le type *IeEnumerable* pour l'instance I_e) ;
- Les instances de la classe *VariableLevel* sont converties en états, qui sont rattachés au type correspondant (*e.g.,* les états L_1 et L_2 relatifs à I_e sont rattachés à *IeEnumerable*).

La Figure 4.14 présente la hiérarchie objets-types-états du modèle objets-relations correspondant au modèle des conditions expérimentales de la Figure 4.13. Les relations spéciales $r_{activer}$ et $r_{désactiver}$ issues de la définition d'un modèle objets-relations (voir Section 1.3.2) seront utilisées pour représenter l'activation ou la désactivation de la variable indépendante I_b, de la variable dépendante D_s (activée : la vitesse d'exécution de la tâche est enregistrée, désactivée : la vitesse d'exécution de la

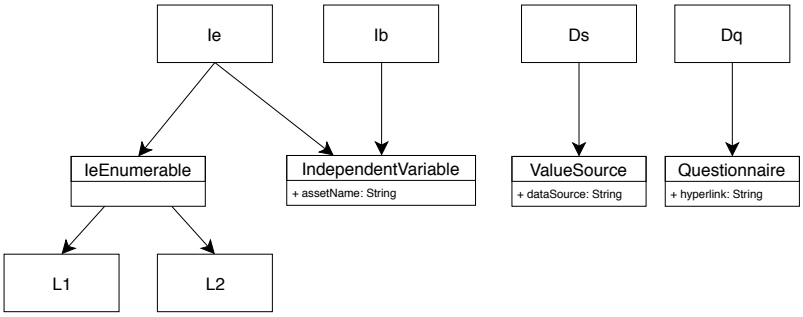


Figure 4.14 – Hiérarchie objets-types-états du modèle objets-relations correspondant au modèle des variables expérimentales de la Figure 4.13. Le type spécial *Objet*, commun à tous les objets, n'est pas représenté. Les états uniques (*i.e.,* attachés à un type mono-état), à vocation symbolique, ne sont pas représentés non plus.

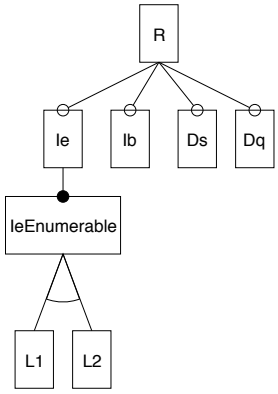


Figure 4.15 – FM dérivé du modèle objets-relations de la Figure 4.14.

tâche n'est pas enregistrée) et pour l'ouverture et la fermeture du questionnaire D_q. Nous noterons les réalisations respectives $activate(o)$ et $deactivate(o)$, où o est l'objet à activer (*resp.* désactiver), $o \in \{I_b, D_s, D_q\}$. Pour chaque énumération défini dans le modèle objets-relations, on ajoute une relation paramétrique, qui permettra de faire passer la variable dépendante à l'état souhaité. Par exemple, pour le modèle représenté en Figure 4.14, on peut définir la relation $activateState(e) = \langle(IeEnumerable, Independentvariable), (e, \varepsilon), \{\{0, 1\}\}\rangle$, où $e \in \{L_1, L_2\}$ et où ε est l'état unique associé à *IndependentVariable*. On remarque que les deux types impliqués dans la relation doivent correspondre au même objet, car $P = \{\{0, 1\}\}$.

Le FM résultant est représenté en Figure 4.15.

2.6 Mise en correspondance avec la bibliothèque de composants RV

AGENT est fourni avec une bibliothèque de composants déjà prête à l'emploi. Seule la mise en correspondance avec le modèle doit être faite par le concepteur de l'évaluation expérimentale.

La bibliothèque permet d'implémenter les types et relations définis dans le modèle objets-relations. Le type *IndependentVariable* permet de lier les variables indépendantes avec leur implémentation concrète. Il est traduit en un script Unity3D portant un champ de type *string* et permettant de définir le champ *assetName* comme un identifiant faisant

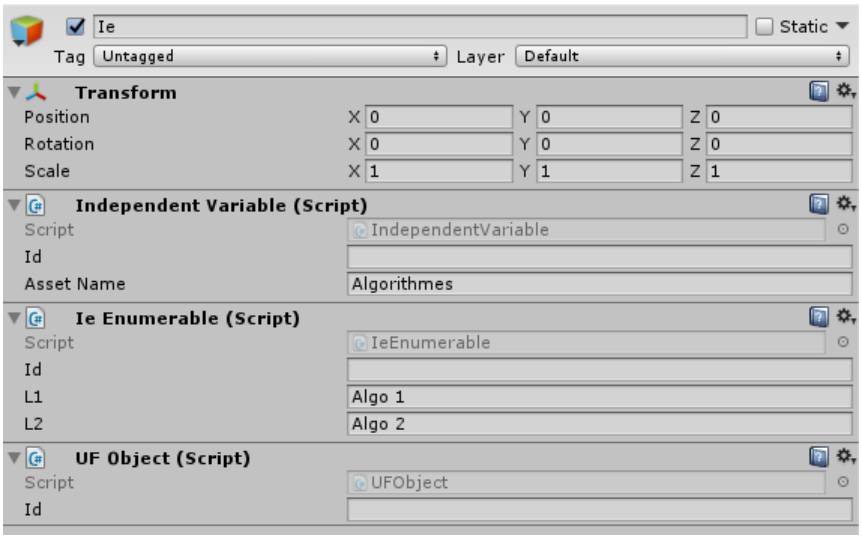


Figure 4.16 – Structure de l'objet I_e avec l'implémentation #FIVE / Unity3D. Les types *IeEnumerable* et *IndependentVariable* correspondent aux scripts C# du même nom. La classe *UFObject* est issue de #FIVE ; elle permet d'indiquer que I_e est un objet d'un point de vue objets-relations. I_e correspond à un ensemble d'algorithmes identifiés par le nom *Algorithmes* et les algorithmes possibles ont pour nom *Algo 1* et *Algo 2*.

référence à la ressource correspondant à la variable indépendante. Par exemple, si $L1$ et $L2$ représentent chacun une version différente d'un algorithme, le champ *assetName* peut correspondre au nom de l'objet de la scène qui comporte les deux versions de cet algorithme. Le fait de définir un identifiant permet de pointer vers n'importe quel type d'objet Unity3D, *e.g.,* un *GameObject* ou un script. Cette généricité permet de vérifier **P4**. Le type *ValueSource* comporte de la même manière un champ permettant d'accéder à la source de données correspondant à la variable dépendante. Le type *Questionnaire* contient un champ correspondant au lien hypertexte pointant vers le questionnaire. **P1** et **P2** sont donc également vérifiées. La Figure 4.16 représente l'implémentation #FIVE / Unity3D de l'objet I_e. La Figure 4.17 représente l'implémentation de la relation *activerÉtat*.

2.7 Modèle de scénarios

La Figure 4.18 représente la partie du métamodèle responsable du protocole. Un protocole est une liste ordonnée d'états. La Figure 4.19 représente le modèle de protocole de l'évaluation expérimentale du cas illustratif pour le groupe G_b. Le protocole du groupe $G_{\neg b}$ est le même, sauf que la référence à I_b n'est pas présente.

Un protocole est constitué de cinq types d'états : *StartState* (représenté par un disque vert), *EndState* (représenté par un disque rouge), *SimpleState* (en jaune), *RandomLoopState* (en mauve) et *CustomizedLoopState* (en orange, voir modèle de protocole de la Figure 4.24). Il est possible d'attacher des conditions (rectangles bleus) aux instances de *RandomLoopState* et de *CustomizedLoopState*. Ces états représentent en effet des répétitions de passages sur différentes conditions. Le nombre de répétitions de chaque condition peut être spécifié en haut à droite de ces états (voir Figure 4.19).

```
// Relation activateState
0 references
public class ActivateState : UFRelation
{
    //--------------------------------------------------------------------------------------------------------

    // Références aux types IeEnumerable et IndependentVariable
    // Les annotations UFObjectPattern permettent d'exprimer à quel objet doivent être attachés les types
    // (ici, les deux types doivent être attachés au même objet Ie)

    // Référence au type IeEnumerable
    [UFObjectPattern("Ie", "IeEnumerable")]
    public IeEnumerable ieEnum;

    // Référence au type Indepedentvariable
    [UFObjectPattern("Ie", "IndependentVariable")]
    public IndependentVariable ieVar;
    //--------------------------------------------------------------------------------------------------------

    // Paramètre de la relation correspondant à l'état à activer (L1 ou L2)
    public string stateToActivate;

    // Méthode issue de #FIVE permettant d'exprimer des conditions de réalisation de la relation
    12 references
    public override bool IsRunnable()
    {
        return stateToActivate == ieEnum.l1 || stateToActivate == ieEnum.l2;
    }

    // Code exécuté lors de la réalisation
    12 references
    public override void Run(Action resultCallback)
    {
        // Récupération du GameObject correspondant à IndependentVariable par son nom (e.g "Algorithmes")
        Transform varTransform = GameObject.Find(ieVar.assetName).transform;

        // Parcours des enfants du GameObject
        for(int i = 0; i < varTransform.childCount; i++)
        {
            // Si l'enfant a pour nom celui de l'état à activer, on l'active. Sinon, on le désactive.
            GameObject child = varTransform.GetChild(i).gameObject;
            if (child.name == stateToActivate)
                child.SetActive(true);
            else
                child.SetActive(false);
        }
    }
}
```

Figure 4.17 – Code de la relation *activateState* en #FIVE / Unity3D.

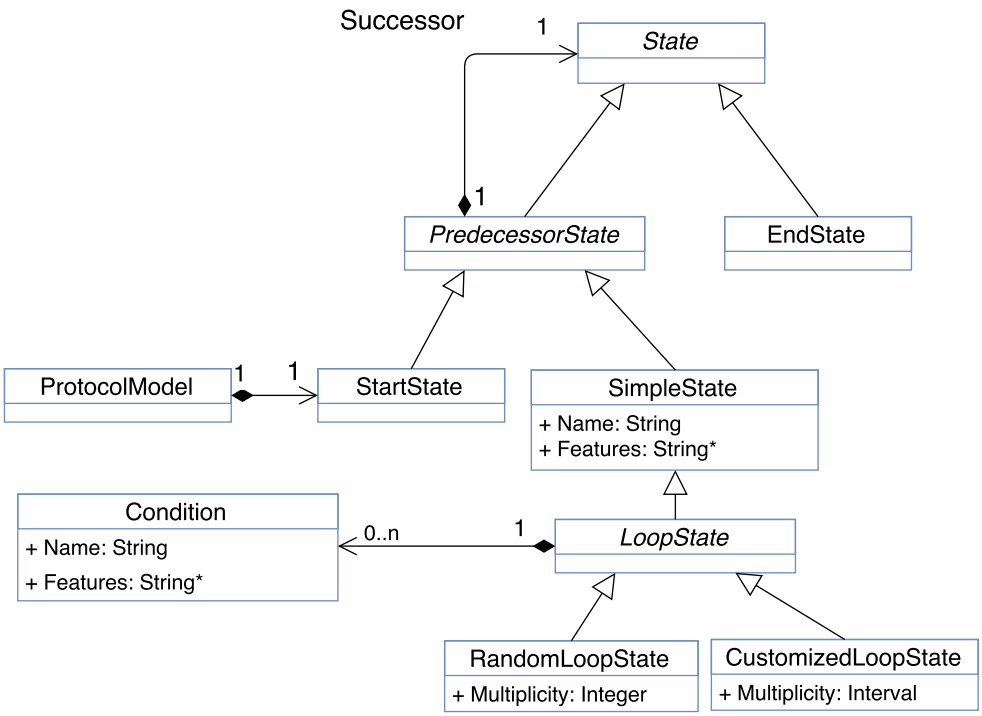


Figure 4.18 – Partie du métamodèle d'AGENT responsable des protocoles.

Les instances de *RandomLoopState* indiquent une répétition en ordre aléatoire de chaque condition. Par exemple, l'état *Acclim* permet 4 répétitions pour chacune des deux conditions *Condition1* et *Condition2* en ordre aléatoire. Les instances de *CustomizedLoopState* permettent d'autres sortes de répétition, *e.g.,* une répétition en ordre déterministe ou basée sur le choix du sujet. Ce genre de répétition est en effet parfois utilisé pour les phases d'entraînement. On peut alors spécifier le nombre de répétitions sous forme d'intervalle (*e.g.,* $0..*$ pour un nombre libre de répétitions).

Chaque état est caractérisé par l'ensemble des *features* définissant la configuration correspondante, par approche implicite. Par exemple dans l'état *Questionnaire* du protocole de la Figure 4.19, seule la *feature* D_q est listée, donc la configuration correspondante est $\{R, D_q\}$. Dans le cas des instances de *RandomLoopState* et *CustomizedLoopState*, on doit prendre en compte aussi la configuration correspondant aux conditions qui y sont attachées. Ainsi, les *features* portées par l'état lui-même correspondent aux *features* qui resteront dans la configuration quelque soit la condition. Les *features* portées par une condition ne seront présentes que lors du passage par cette condition. Par exemple pour l'état *DataAcq*, les deux configurations qui vont s'alterner, chacune 32 fois, sont $\{R, I_b, D_s, I_e, IeEnumerable, L_1\}$ (*Condition1*) et $\{R, I_b, D_s, I_e, IeEnumerable, L_2\}$ (*Condition2*).

2.8 Compilation et synthèse

Cette section indique comment le scénario est compilé puis intégré à un projet de développement d'une évaluation expérimentale de RV, dans le cadre de la phase de synthèse.

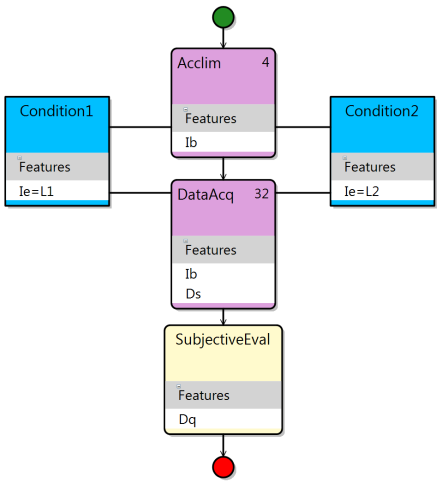


Figure 4.19 – Exemple de modèle de protocole.

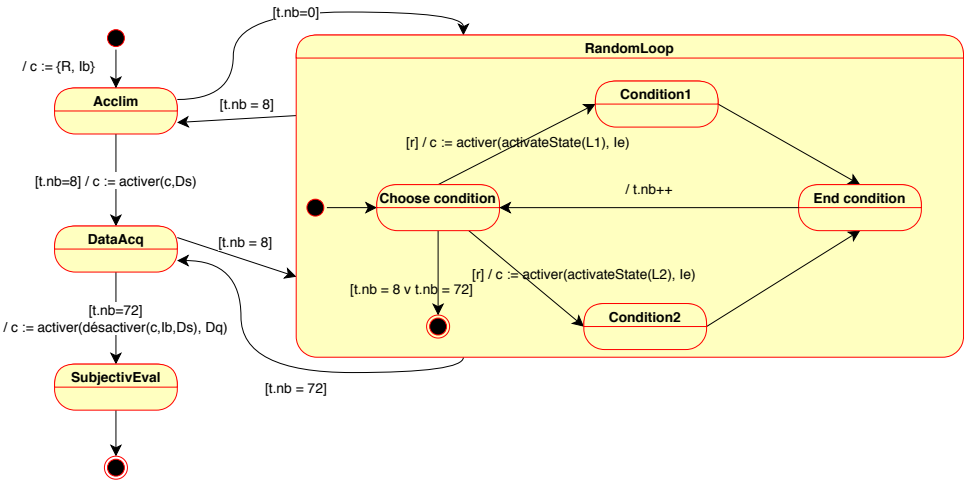


Figure 4.20 – FSM UML issue de la compilation du protocole de la Figure 4.19.

2.8.1 Compilation du modèle de scénarios

Une fois qu'un modèle de protocole a été produit, celui-ci peut être compilé en code XML et C# pour une intégration à Unity3D. Ce code représente le protocole sous forme d'une machine à états hiérarchique, conforme au formalisme des FSM UML. La Figure 4.20 représente la FSM UML correspondant au modèle de protocole de la Figure 4.19. La variable $t.nb$ correspond au nombre total de fois que la tâche a été réalisée. La condition $[r]$ indique que la transition est aléatoirement activée.

La compilation se fait selon les règles suivantes :

— **transformation des états *Start*, *End* et *SimpleState* :** ils sont respectivement transformés en états initiaux, finaux et atomiques ;

— **transformation des transitions :** toutes les transitions doivent avoir comme effet de sélectionner ou déselectionner les *features* spécifiées par le modèle de protocole. Pour ce qui est des gardes (conditions d'activation de la transition), deux possibilités existent :

1. l'origine de la transition est un état *Start*, *End* ou *SimpleState*. Dans ce cas la transition n'est pas gardée ;
2. l'origine de la transition est un état *LoopState*. Dans ce cas, la transition générée est gardée par la condition que le nombre de boucles spécifié par l'état *LoopState* a été atteint ;

— **transformation des états *LoopState* :** ces états sont transformés en sous-machines à états (état non-atomique). Chaque instance de *Condition* attachée à l'état est transformé en état atomique. La sous-machine à états boucle ensuite sur ces états de façon à respecter le nombre de boucles spécifié.

2.8.2 Intégration de la spécification de scénarios à l'EV

Deux bibliothèques de classes sont fournies pour intégrer une spécification de scénarios à l'EV.

La bibliothèque d'ordonnancement des conditions fourni des algorithmes pour ordonner l'enchaînement des conditions dans le cadre des états de type *RandomLoopState* et *CustomizedLoopState* : aléatoire à graine constante, aléatoire à graine variable, déterministe, contrôlé par l'utilisateur, *etc.* La bibliothèque de gestion des tâches permet de détecter l'exécution d'une tâche et de lancer les événements liés à cette exécution. Par exemple, si la tâche consiste à se déplacer d'un point de départ D à un point d'arrivée A, la bibliothèque doit permettre de détecter la fin de la tâche lors de l'arrivée au point A et de déplacer automatiquement l'avatar du sujet au point D pour préparer une nouvelle exécution de la tâche.

2.9 Évaluation d'AGENT

Notre approche vérifie les propriétés identifiées en Section 2.2.

— **P1** et **P2** : le type des données enregistrées est déterminé par le concepteur de l'évaluation expérimentale lors de l'étape d'implémentation du modèle objets-relations. De plus, il est possible de définir des questionnaires en spécifiant leur lien hypertexte ;

— **P3** : les variables indépendantes sont transformées soit (1) en *features* optionnelles feuilles du FM, qui sont assimilables à des booléens, soit (2) en groupes XOR, assimilables à une énumération ;

— **P4** : l'étape de mise en correspondance entre le modèle et la bibliothèque objets-relations permet d'associer les variables indépendantes à tout type de composant ;

— **P5** et **P6** : il est possible de modéliser un protocole par groupe de sujets ;

— **P7** : il est possible de modéliser les phases d'entraînement comme des états de type *RandomLoopState* ou *CustomizedLoopState* pour lesquels les *features* correspondant aux variables dépendantes ne sont pas sélectionnées.

AGENT a été testé sur deux cas d'utilisations, qui sont les expériences présentées par nous-mêmes dans un précédent article [Le Moulec et al., 2016] et par Mossel et Koessler [Mossel and Koessler, 2016]. La première de ces évaluations expérimentales suit exactement la même conception que l'exemple illustratif que nous avons utilisé

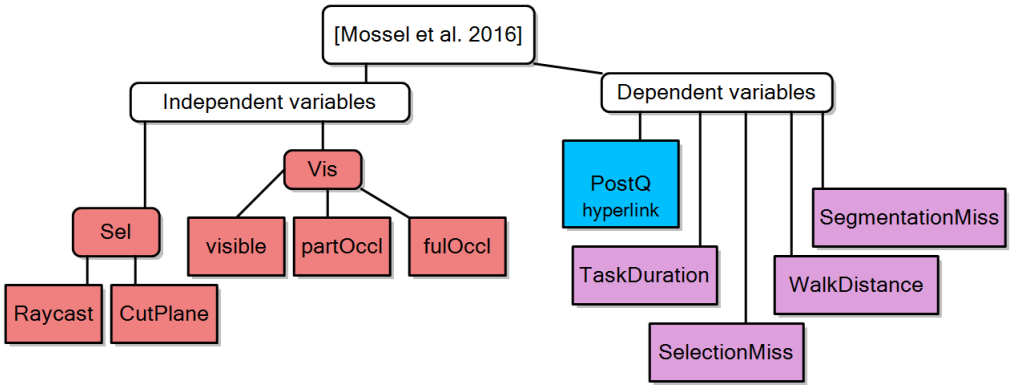


Figure 4.21 – Modèle des conditions expérimentales de l'évaluation présentée par Mossel et Koessler [Mossel and Koessler, 2016].

tout au long de la Section 2 et donc nous n'allons pas détailler de nouveau l'utilisation d'AGENT pour cette évaluation expérimentale.

Nous allons en revanche détailler l'utilisation d'AGENT pour l'évaluation expérimentale de Mossel et Koessler. Dans cette évaluation expérimentale, les auteurs comparent deux techniques de sélection. Ils évaluent leur effet sur l'efficacité de l'utilisateur pour la réalisation d'une tâche de sélection en fonction de sa difficulté. AGENT a été utilisé pour modéliser et implémenter les conditions expérimentales (Section 2.9.1) et le protocole (Section 2.9.2).

2.9.1 Modèle des conditions expérimentales

La Figure 4.21 montre le modèle des conditions expérimentales d'AGENT pour l'évaluation présentée par Mossel et Koessler [Mossel and Koessler, 2016]. Les variables indépendantes sont données explicitement dans l'article : deux techniques de sélection *Raycast* et *CutPlane* doivent être utilisées dans trois cas de figure, chacun correspondant à un niveau difficulté : sélectionner un objet complètement visible (*visible*), partiellement occulté (*partOccl*) ou complètement occulté (*fulOccl*). Il y a donc deux variables énumératives, l'une correspondant à la technique de sélection (*Sel*) et l'autre correspondant au niveau de difficulté de la visualisation de l'objet (*Vis*). Les variables dépendantes sont également données explicitement dans l'article : quatre grandeurs sont mesurées pour évaluer la performance du sujet lors de la réalisation d'une tâche : le temps nécessaire à sa réalisation (*TaskDuration*), la distance totale parcourue par le sujet dans l'EV (*WalkDistance*) et deux mesures du nombre d'erreurs commises lors de la tâche (*SegmentationMiss* et *SelectionMiss*). De plus, des mesures de performances subjectives sont récoltées grâce à un questionnaire (*postQ*). Le modèle objets-relations et le FM sont respectivement représentés en Figures 4.22 et 4.23. Les relations du modèle objets-relations (en plus des relations spéciales $r_{activer}$ et $r_{désactiver}$) sont les relations paramétriques suivantes (L'unique état de *IndependentVariable*, non représenté sur la Figure 4.22, est ε) :

— $activerÉtatSel(e) = \langle(SelEnumerable, IndependentVariable), (e, \varepsilon), \{\{0, 1\}\}\rangle$, où $e \in \{Raycast, CutPlane\}$,

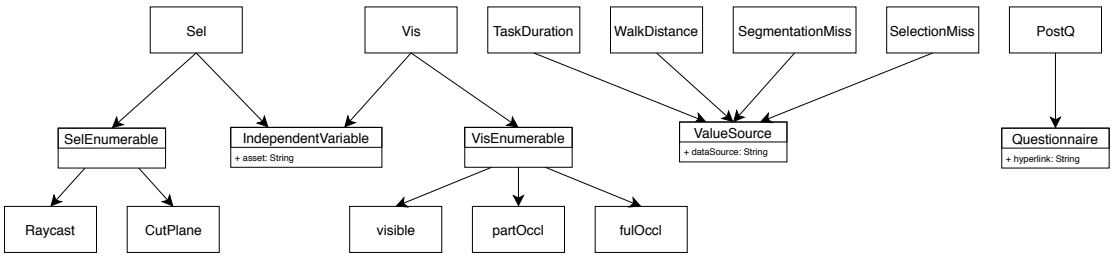


Figure 4.22 – Hiérarchie objets-types-états du modèle objets-relations correspondant à l'évaluation expérimentale de Mossel et Koessler [Mossel and Koessler, 2016].

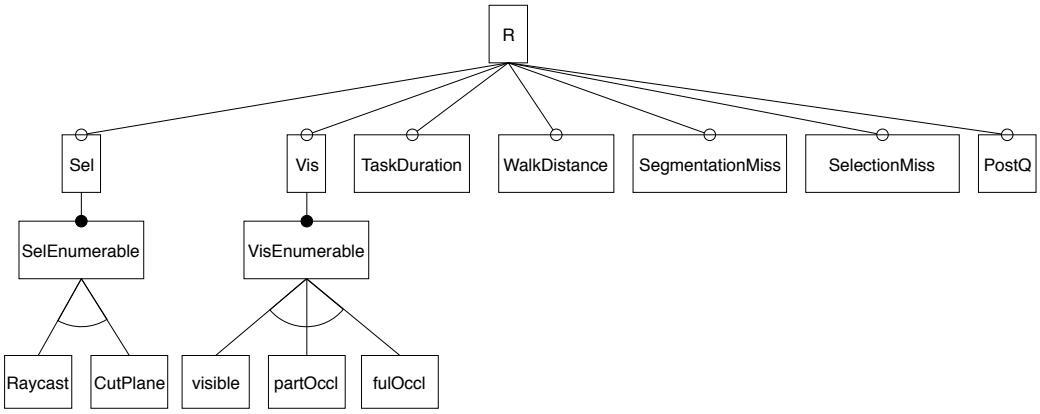


Figure 4.23 – FM correspondant à l'évaluation expérimentale de Mossel et Koessler [Mossel and Koessler, 2016].

— $activerÉtatVis(e) = \langle(VisEnumerable, IndependentVariable), (e, \varepsilon), \{\{0,1\}\}\rangle$, où $e \in \{visible, partOccl, fulOccl\}$.

2.9.2 Modèle du protocole

La Figure 4.24 montre le modèle de protocole pour l'évaluation présentée par Mossel et Koessler [Mossel and Koessler, 2016]. Le protocole est divisé en une phase d'entraînement (*Training*), une phase de mesures (*Experiment*) et une phase de réponse au questionnaire (*PostQuestionnaire*).

La phase d'entraînement est une instance de *CustomizedLoopState* de multiplicité $0..*$ car les sujets sont laissés libres d'interagir comme ils le veulent dans l'EV jusqu'à ce qu'ils se sentent prêts pour la "vraie" évaluation. La bibliothèque d'ordonnancement des conditions permet d'implémenter cette phase (voir Section 2.8.2). Dans cette phase telle qu'elle est décrite dans l'article, les sujets peuvent interagir alternativement avec les deux techniques de sélection sur des objets dont les conditions de visibilité peuvent varier entre les trois valeurs possibles *visible*, *partOccl* et *fulOccl*. Ces objets sont tous présents en même temps dans l'EV, donc les trois valeurs possibles de visibilité le sont tout autant (certains objets sont visibles, d'autres partiellement occultés et d'autres entièrement occultés). AGENT ne permet pas de représenter plusieurs valeurs d'une même variable présentes toutes en même temps dans un EV, du fait du modèle objets-relations et du FM générés à partir du modèle des conditions expérimentales, qui ne définissent qu'une occurrence de la variable et donc un seul groupe XOR pour ces valeurs

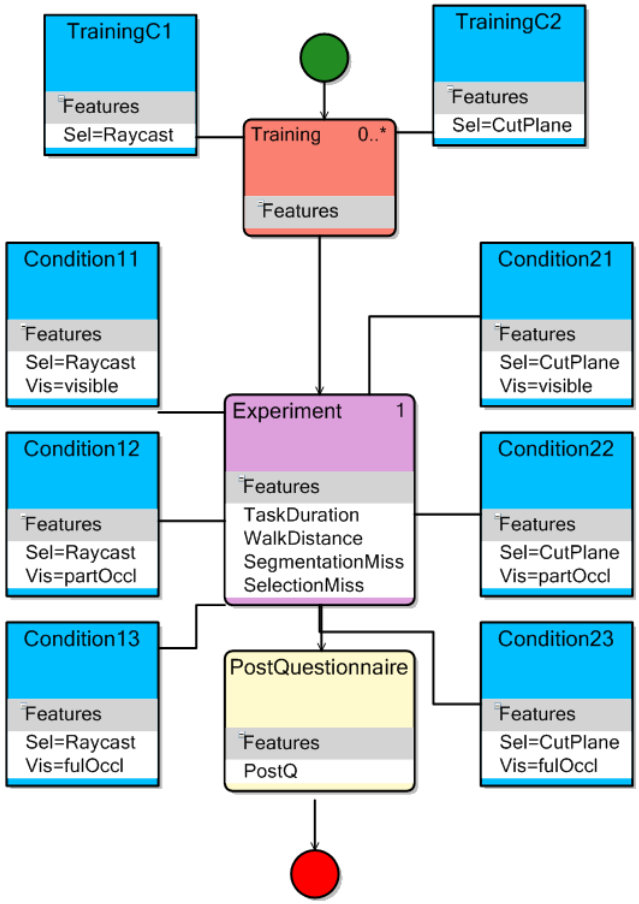


Figure 4.24 – Modèle du protocole pour l'évaluation présentée par Mossel et Koessler [Mossel and Koessler, 2016].

possibles. Par exemple, *Vis* et le groupe XOR composé de ses valeurs possibles *visible*, *partOccl* et *fulOccl* n'apparaît qu'une seule fois dans le FM de la Figure 4.23. Pour permettre de respecter la conception originale de la phase d'entraînement, il faudrait ajouter au modèle objets-relations de la Figure 4.22 des objets similaires à *Vis*, *i.e.,* des objets portant les mêmes types et dont on peut donc faire varier la visibilité. La solution consiste à créer de tels objets directement dans l'EV lors de l'étape de synthèse grâce à la bibliothèque objets-relations. Ces objets sont utilisables lors de la phase d'entraînement, mais leur état n'est ni considéré comme une condition expérimentale, ni une *feature* pouvant varier, du point de vue du modèle de protocole.

La phase de mesure met en scène un unique objet qui peut être visible, partiellement occulté, ou entièrement occulté selon les conditions. La bibliothèque de gestion des tâches peut être utilisée pour développer un script détectant la réalisation de la tâche de sélection et transportant alors l'avatar du sujet à son point de départ dans l'EV.

AGENT constitue un cas d'utilisation montrant une implémentation de l'approche LPLRV pour la production d'évaluations expérimentales en RV. AGENT est un DSL qui permet de représenter le modèle de domaine et les spécifications de scénarios avec une syntaxe graphique qui lui est propre.

Le cas d'utilisation que nous présentons en Section 3 montre une autre manière d'implémenter l'approche pour un autre domaine et dans un autre contexte. Cette autre implémentation est Tremplin. Il s'agit d'une LPLRV dont le but est de produire des applications de formation aux procédures chirurgicales. C'est une amélioration proposée dans le cadre du projet Sunset, le successeur du projet S3PM présenté au Chapitre 2, Section 3.1.1. Les limites qui ont été identifiées pour S3PM sont valides pour Sunset. Notre but est de les corriger.

3 Cas d'utilisation : production d'une application de formation aux procédures chirurgicales

Dans cette section, nous présentons un cas d'utilisation de l'approche LPLRV reposant sur la production d'applications de formation aux procédures chirurgicales, dans le cadre du projet Sunset. La Section 3.1 fait un rappel du contexte dans lequel ce travail s'inscrit et présente les objectifs de notre contribution. La Section 3.2 présente une vue d'ensemble de la LPLRV que nous avons mis en place pour ce cas d'utilisation. Les Sections 3.3 à 3.7 présentent les différentes étapes du processus ainsi mis en place.

3.1 Motivations

Le projet Sunset vise à produire des applications de formation en EV pour le personnel de blocs-opératoires, *e.g.,* pour apprendre à préparer une table d'opération. Ce projet repose sur l'ontologie OntoSPM [Gibaud et al., 2014]. Les contributeurs du projet proposent un système de génération de spécifications de scénarios #SEVEN [Claude et al., 2015] à partir de vidéos montrant le déroulement d'une opération. Cette génération repose sur l'annotation de ces vidéos grâce au logiciel SurgeTrack [Claude, 2016]. Le projet Sunset intègre déjà des mécanismes

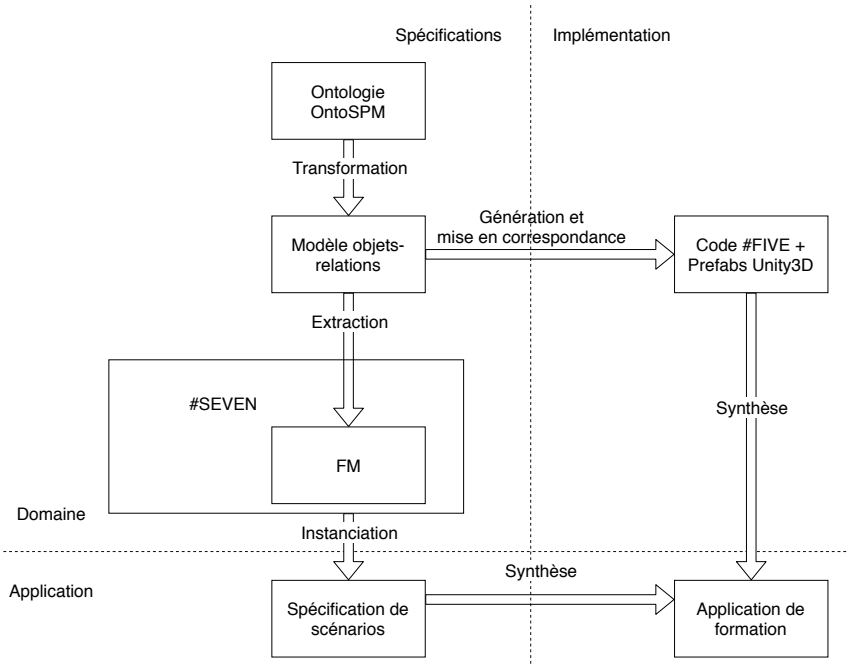


Figure 4.25 – Vue d'ensemble de Tremplin.

d'automatisation de production de scénarios via une ontologie, qui permet de représenter la connaissance métier. Nous nous focalisons sur l'automatisation de la production des composants RV (modèle objets-relations) et nous proposons une amélioration de l'automatisation de la production de scénarios. Le projet permet en effet déjà une production automatique de spécification de scénarios à partir de l'ontologie OntoSPM. Nous ne souhaitons pas remplacer ce module fonctionnel, mais proposons de mettre en relation la spécification de scénarios et la bibliothèque objets-relations. Cette mise en relation est en effet réalisée manuellement dans la version actuelle du projet.

3.2 Vue d'ensemble

La Figure 4.25 présente une vue d'ensemble de la LPLRV mise en place. Le modèle du domaine est l'ontologie OntoSPM, présentée en Section 3.3. Le mécanisme de transformation de cette ontologie en modèle objets-relations est présentée en Section 3.4. La génération du code #FIVE [Bouville et al., 2015] correspondant est présentée en Section 3.5. Le modèle de scénarios utilisé est #SEVEN [Claude et al., 2015]. Dans Sunset, une spécification de scénarios est générée par annotation de vidéos par le biais du logiciel SurgeTrack (voir état de l'art). Tremplin s'appuie sur ce même mécanisme, que nous adaptons afin d'y intégrer notre approche. Nous présentons cette adaptation en Section 3.6. Enfin, la synthèse du projet de développement d'application de RV correspondant, réalisé sur Unity3D, est présentée en Section 3.7.

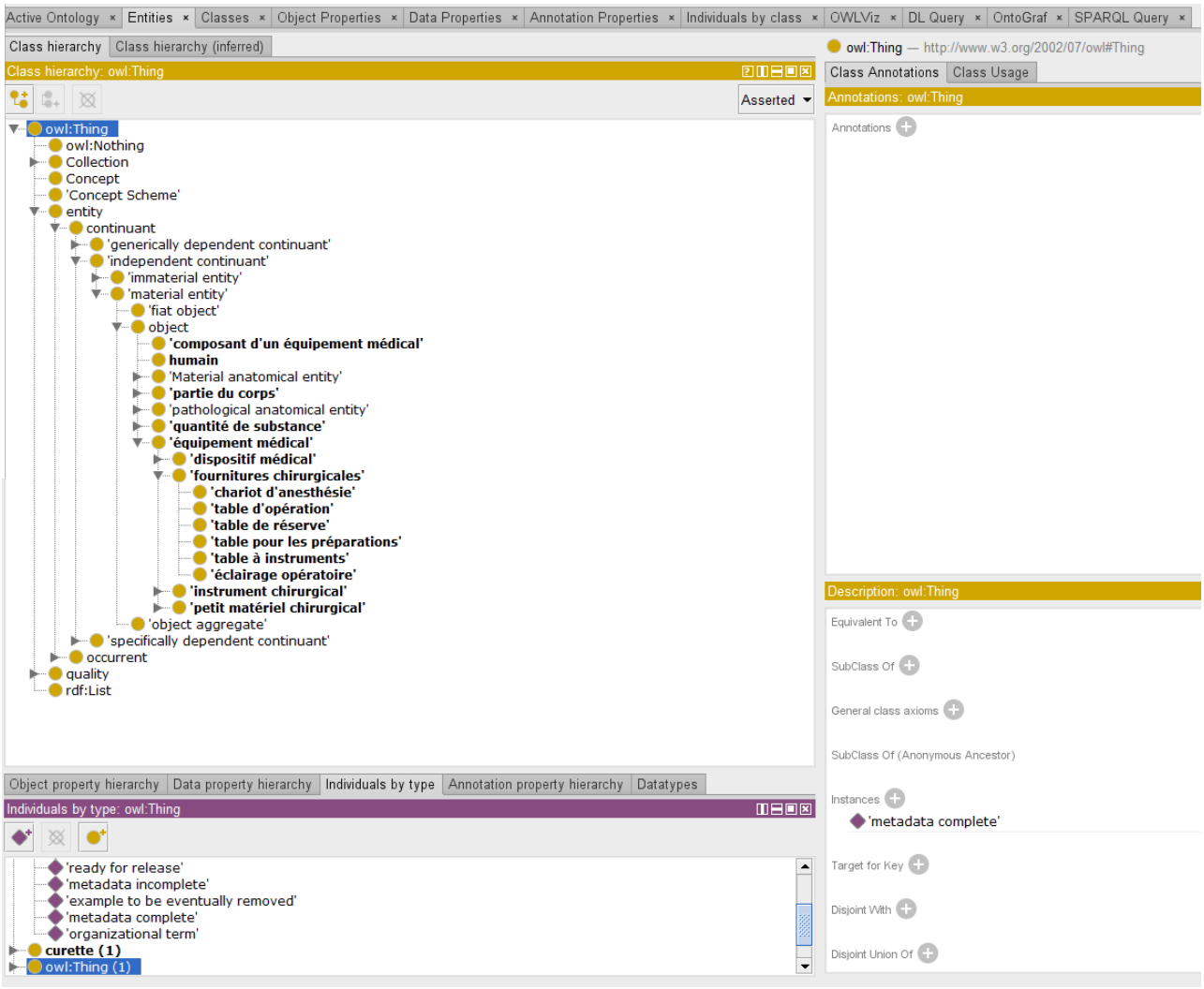


Figure 4.26 – Extrait de l'ontologie OntoSPM, visualisée sur l'outil Protégé.

3.3 Modélisation du domaine par une ontologie

La première étape consiste à produire une ontologie représentant le domaine. La responsabilité en revient aux experts du domaine, qui peuvent être assistés par des informaticiens. Cette ontologie est dite spécifiée car elle est produite à la main et représente le domaine de manière fidèle, sans se soucier de l'implémentation. Elle est composée de classes représentant les concepts du domaine et d'assertions qui précisent les liens entre ces concepts. Théoriquement pour les ontologies il n'existe pas de syntaxe, de format ou de standard afin de représenter les classes et les assertions, qui peuvent être exprimées en langage naturel. Dans la pratique, on utilise des langages formels pour exprimer les assertions, comme le langage défini par le format d'ontologies OWL, très proche du langage naturel. La Figure 4.26 représente un extrait de l'ontologie

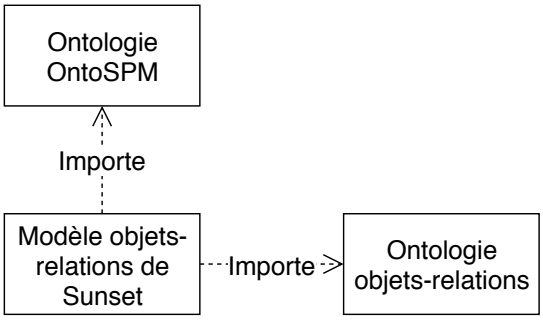


Figure 4.27 – Structure des ontologies utilisées par Tremplin. L'ontologie originale OntoSPM et l'ontologie décrivant le formalisme objets-relations sont importées par une troisième ontologie décrivant le modèle objets-relations associé à Sunset.

OntoSPM [Gibaud et al., 2014], visualisée sur l'outil Protégé [6].

3.4 Production du modèle objets-relations

L'ontologie n'est pas construite sur un modèle objets-relations, du moins pas explicitement. L'ontologie est large et il n'est donc pas concevable de reconstruire un modèle objets-relations depuis zéro. La réutilisation de l'ontologie est primordiale dans le cas présent. Nous proposons donc d'étendre l'ontologie avec les concepts de la modélisation objets-relations. Il y a intérêt à ne pas modifier directement l'ontologie mais bien à en développer une extension car l'ontologie de base pourra toujours être utilisée dans d'autres contextes, alors que l'extension est propre à la LPLRV en développement.

Nous commençons par définir une deuxième ontologie chargée de représenter le formalisme des modèles objets-relations, tel qu'il est décrit en Section 1.3.2. Nous créons le modèle objets-relations de Sunset par synthèse de l'ontologie OntoSPM et de cette deuxième ontologie (voir Figure 4.27). Nous obtenons donc une troisième ontologie par ajout d'héritages aux classes de Sunset : les classes correspondants à des relations héritent du concept *Relation* de l'ontologie objets-relations, les concepts correspondant à des types de la classe *Type*, *etc.*. Pour Sunset, nous faisons une différenciation entre deux sortes de types : les types abstraits qui décrivent en général certaines caractéristiques des objets qui possèdent ces types, *e.g.,* "coupant", et les types concrets représentant directement des objets, *e.g.,* "scalpel". Ainsi un scalpel est défini dans le modèle objets-relations par l'agrégation des types abstraits "coupant" et "prenable" et du type concret "scalpel". Les types abstraits sont ceux qui sont utilisés dans les relations, ce qui permet la généricité de celles-ci. Par exemple, la relation "couper" utilise le type "coupant", ce qui lui permet d'être utilisée avec n'importe quel objet coupant, *e.g.,* un scalpel ou des ciseaux. Les types concrets sont utiles pour la génération de *Prefabs* d'objets (voir Section 3.5).

6. https://protege.stanford.edu/

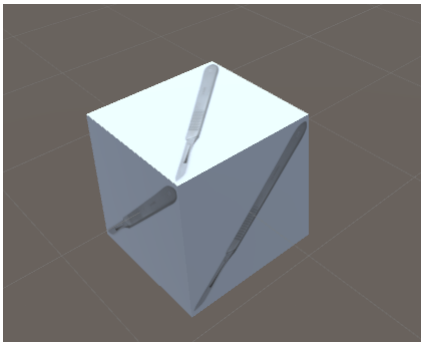


Figure 4.28 – *Prefab* Unity3D généré pour le type concret *Scalpel*.

3.5 Génération de la bibliothèque objets-relations

Cette section présente le code et les éléments Unity3D générés pour l'implémentation en #FIVE de la bibliothèque objets-relations, ainsi que le processus de génération.

3.5.1 Code et composants générés

La bibliothèque objets-relations utilisée est #FIVE [Bouville et al., 2015]. La génération se fait pour l'éditeur Unity3D, la version de #FIVE que nous utilisons étant développée pour cette plate-forme. Tremplin génère donc le code #FIVE correspondant au modèle objets-relations. Le code généré est un squelette devant être complété par les développeurs. Cependant, Tremplin ne gère pas la génération des états dans sa version actuelle. Nous générons de plus un *Prefab* pour chaque type concret, *e.g.,* "scalpel". Les *Prefabs* générés sont, comme le code, destinés à être repris à la main par des développeurs et des designers. Ceux que nous générons servent simplement de base pour commencer à prototyper l'application. Ils se présentent sous la forme d'un cube sur lequel une image représentant le type concret est collée à partir d'une recherche d'image sur internet (réalisée grâce aux API fournies par les moteurs de recherche). Par exemple, le *Prefab* correspondant au type "scalpel" est représenté en Figure 4.28.

3.5.2 Processus de génération

Le processus de génération est présenté en Figure 4.29. le modèle objets-relations ontologique peut être questionné via des requêtes SPARQL [7], un langage de requêtes adapté aux ontologies. Le module de requêtes SPARQL nous permet d'extraire un modèle intermédiaire représentant la structure d'un modèle #FIVE. Ce modèle est ensuite transformé en code #FIVE utilisable via l'API de l'éditeur #FIVE, qui permet de créer des modèles #FIVE à partir d'une interface graphique. Cela nous permet de produire un code #FIVE homogène avec le code généré par l'éditeur, et qui peut donc être retravaillé via celui-ci.

7. https://www.w3.org/TR/rdf-sparql-query/

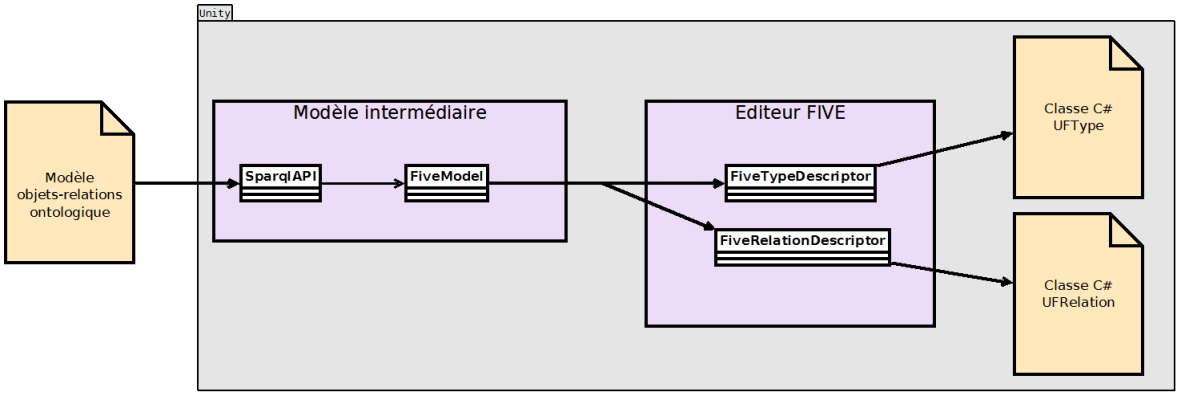


Figure 4.29 – Processus de génération du code #FIVE à partir du modèle objets-relations ontologique.

3.6 Scénarios

Comme nous l'avons expliqué, Tremplin intègre le processus de production de spécifications de scénarios #SEVEN de Sunset. Des vidéos représentant des procédures chirurgicales sont annotées par des instances de l'ontologie OntoSPM via le logiciel SurgeTrack. Un scénario #SEVEN est généré à partir de ces annotations. Bien que notre approche n'interfère pas avec ce processus, elle permet l'utilisation du modèle objets-relations (qui n'est qu'une extension de l'ontologie OntoSPM) afin de définir les annotations utilisées dans SurgeTrack. SurgeTrack permet alors de créer des objets (au sens du modèle objets-relations) en annotant ceux qui sont visibles sur la vidéo, et de leur associer des actions, *i.e.,* des réalisations de relations. Considérons par exemple un modèle objets-relations permettant d'utiliser des objets coupants (*e.g.,* ciseaux) pour couper certains tissus (*e.g.,* du coton). Si un extrait vidéo montre une infirmière de bloc en train de couper un morceau de coton avec des ciseaux, le concepteur devra créer les objets *ciseaux_1*, *infirmière_1* et *coton_1*, puis les réalisations des relations *Prendre* (pour prendre les ciseaux puis le morceau de coton) et *Couper*. La portion de spécifications de scénario #SEVEN correspondante sera générée (représentée sous une forme simplifiée en Figure 4.30). Pour rappel, #SEVEN permet d'adjoindre un *Capteur* (losange rouge) et un *Effecteur* (carré vert) à chaque transition, *i.e.,* une condition d'activation et une action résultante respectivement. Le comportement associé à chacun d'entre eux est codé en un script C#. Notre approche permet alors de générer un script pour chaque *Effecteur*, ce script correspondant simplement à la réalisation d'une relation. Les scripts générés peuvent être complétés à la main. Notons que le FM représentant le modèle objets-relations n'est pas utilisé en tant que tel dans ce processus, mais il est un équivalent du modèle objets-relations utilisé.

3.7 Synthèse du projet Unity

Une fois générée, la spécification de scénarios #SEVEN, qui est un fichier XML, peut être intégré au projet Unity3D via le moteur #SEVEN développé pour Unity3D. Le scénario est alors opérationnel pour fonctionner avec le code #FIVE et les *Prefabs*

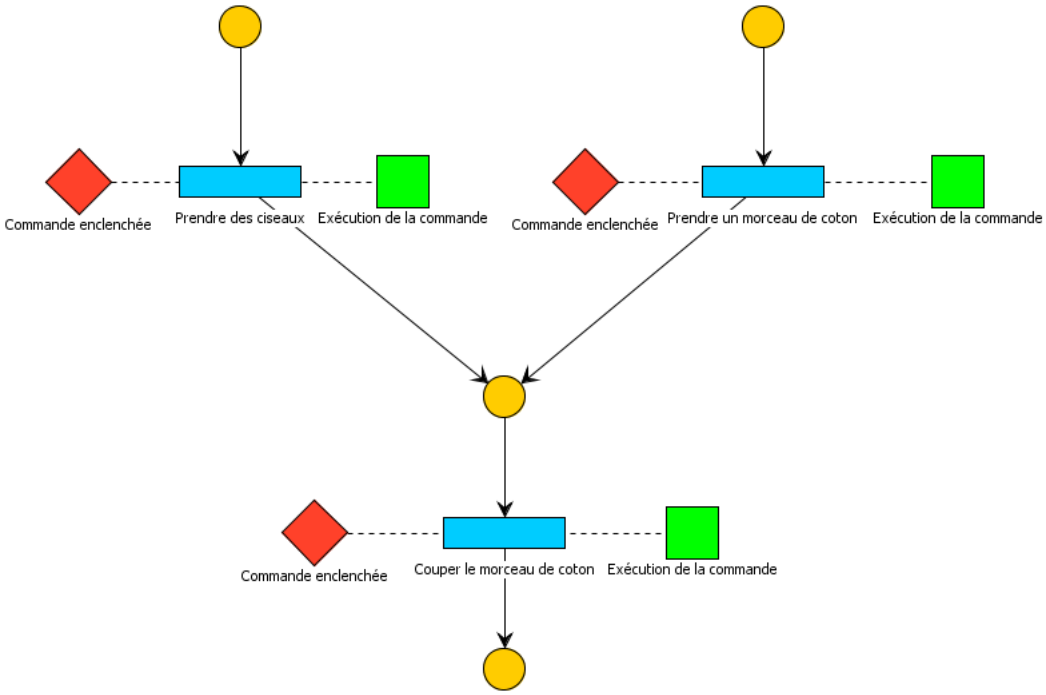


Figure 4.30 – Portion de réseau #SEVEN pour la réalisation de la relation *Couper* par une infirmière, en utilisant des ciseaux et un morceau de coton.

générés, il peut toutefois être retouché par des développeurs. Beaucoup d'aspects des applications de RV non gérés par notre approche doivent cependant être développés, *e.g.,* animation des acteurs, développement des interactions et des métaphores, mise en évidence des objets avec lesquels ils est possible d'interagir, *etc.* La Figure 4.31 montre une portion de la scène finalisée d'une application produite par Tremplin et une portion du scénario généré correspondant à l'action de l'acteur. La portion de spécification de scénarios #SEVEN gère la prise puis le relâchement d'un scalpel. L'acteur, représenté par un avatar d'assistant de bloc, exécute le scénario de prise puis de relâchement d'un scalpel. Ce scénario découle de la portion de spécification de scénarios.

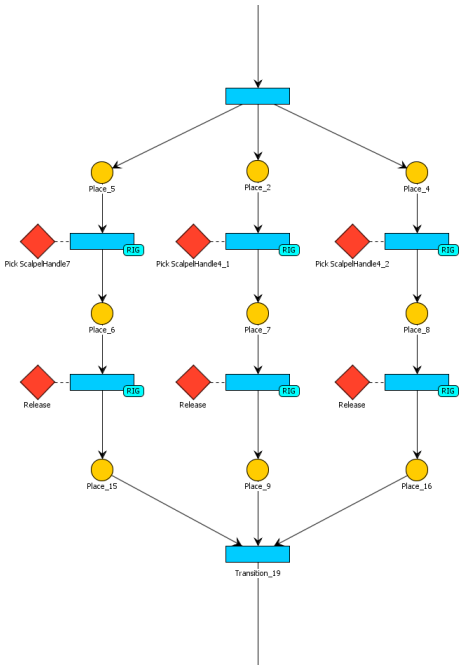
3.8 Discussion

Tremplin permet d'automatiser des tâches qui étaient jusqu'ici faites à la main dans le cadre du projet Sunset. le simple fait d'étendre l'ontologie de base pour en faire un modèle objets-relations permet :

— de générer le squelette du code #FIVE correspondant et des prototypes d'objets ;

— de générer les scripts de #SEVEN responsables de la réalisation des relations ;

— de faire le lien entre le diagramme #SEVEN, les prototypes d'objets et le code générés, de façon à ce qu'une première version "prototype" de l'application soit utilisable.



(a) Portion de spécification de scénarios #SEVEN gérant la prise puis le relâchement d'un scalpel (projet Sunset).

(b) Un acteur représenté par un avatar d'assistant de bloc en train d'exécuter le scénario de prise puis relâchement d'un scalpel découlant de la portion de spécification de scénarios de la Figure 4.31a (projet Sunset).

Figure 4.31 – Portion de la scène finalisée d'une application produite par Tremplin et portion du scénario généré.

Sans cette extension de l'ontologie, seule la structure du diagramme #SEVEN pouvait être générée et tout le reste devait être fait à la main. Du point de vue des développeurs, l'ontologie (le modèle du domaine), faisait office de documentation métier, pour faciliter la compréhension du domaine et donc le développement du modèle objets-relations. De plus, en tant que documentation, elle ne répondait pas à tous les critères définis au Chapitre 3, Section 2.1.2 (son but original n'étant pas de servir de documentation pour des développeurs) :

— **propriété #1 – documentation exhaustive** – l'ontologie représente fidèlement le domaine de la chirurgie ;

— **propriété #2 – contextualisation de la documentation** – l'ontologie peut être difficile à exploiter pour les développeurs non habitués à ce formalisme, encore moins au format OWL utilisé et à tous les mots-clés et concepts à apprendre pour savoir la lire ;

— **propriété #3 – documentation sous plusieurs formes** – la documentation se trouve sous une seule forme, *i.e.,* un fichier OWL éditable que les développeurs doivent se partager. Cela peut poser des problèmes de version , *i.e.,* un développeur doit s'assurer de disposer de la dernière version disponible et non d'une version antérieure ou modifiée par un autre développeur ;

— **propriété #4 – documentation reposant sur des exemples** – l'ontologie est une base de connaissances et non un langage ou une API. Elle peut cependant être interrogée avec des requêtes SPARQL. Elle n'intègre elle-même pas de documentation ou d'exemples à ce sujet, que l'on doit aller chercher ailleurs.

Tremplin permet d'utiliser l'ontologie comme ce qu'elle est vraiment, *i.e.,* une base de connaissances métiers, avec un traitement automatique permettant la conversion vers un modèle objets-relations. De plus, cette extension de l'ontologie permet d'enrichir le processus déjà existant de production automatique de la spécifications de scénarios.

4 Synthèse du chapitre

Dans ce chapitre nous avons présenté l'approche LPLRV. Son but était de répondre à la limite **L1** identifiée dans l'état de l'art (Chapitre 2, Section 3.1) : la production d'applications de RV manque d'abstractions, ce qui provoque : (a) un manque de réutilisation et (b) un manque de séparation des préoccupations (*Separation of Concerns*, SoC). Cette approche permet l'automatisation de la production d'applications de RV. Elle répond à notre objectif d'abstraction (**O1-ABSTRACTION**) par sa construction centrée sur les notions de modèle objets-relations, des scénarios et de modèle de domaine métier. Ces trois concepts structurent en effet toutes les applications de RV. Leur séparation nette dans notre approche est cohérente avec le principe de SoC recherché (**O3-SOC**). Par ces abstractions, notre approche permet la réutilisation (**O2-RÉUTILISATION**). En effet, un même modèle de domaine peut donner lieu à la production de plusieurs applications de RV différentes, avec des spécifications de scénarios différentes. De même, un même modèle objets-relations peut être utilisé de plusieurs manières différentes et implémenté avec des technologies variées. Le modèle objets-relations est traduit en FM, ce qui permet

d'en définir les configurations possibles. Ainsi, le modèle de scénario permet, par la définition de spécifications de scénarios modifiant ces configurations, d'exprimer une grande variété d'applications de RV possibles, tous reposant sur le même modèle de domaine. De plus, le modèle de scénarios et le modèle objets-relations constituent un intermédiaire entre le modèle de domaine et le code. Ceci permet un développement plus centré sur le domaine et abstrayant en partie la complexité d'une programmation plus "classique", *i.e.,* qui s'appuierait uniquement sur des GPL.

Nous avons présenté deux cas d'utilisations implémentant notre approche. Le premier cas est celui d'AGENT, qui permet d'automatiser le production d'applications de RV pour l'évaluation expérimentale. AGENT est un DSL, ce qui contribue à réduire le fossé existant entre le domaine (ici, celui des évaluations expérimentales en RV) et les développements. Le métamodèle d'AGENT permet de plus de fournir une abstraction haut niveau du domaine, qui divise le problème en deux parties : la définition des conditions expérimentales et la définition du protocole. Le modèle des conditions expérimentales permet de s'abstraire du modèle objets-relations. De plus, un tel modèle peut être utilisé pour définir des protocoles différents.

Le deuxième cas d'utilisation est une amélioration proposée pour un projet déjà existant et proposant déjà des mécanismes d'automatisation : le projet Sunset. La principale contribution que nous avons apportée est un processus de génération d'un prototype de l'application de RV, implémenté en l'outil Tremplin. Le prototype généré comprend des prototypes d'objets virtuels et le squelette du code #FIVE [Bouville et al., 2015], une bibliothèque objets-relations. Ce processus convertit automatiquement le modèle du domaine, une ontologie, en code. Cette contribution nous a également permis d'améliorer le processus de production automatique d'une spécification de scénarios à partir de l'ontologie, en le mettant en correspondance avec le modèle objets-relations. Ceci est une manière de réduire le fossé entre le domaine métier et les développements.

Conclusion et travaux futurs 5

1 Conclusion

Au cours de cette thèse nous avons cherché à automatiser la production d'applications de RV. En effet, le développement de telles applications de RV reste aujourd'hui très "artisanal". Les développeurs ont tendance à refaire les mêmes choses d'un projet à un autre et parfois ils ne s'en rendent pas compte. Il y a un manque de réutilisation et de séparation des préoccupations (*Separation of Concerns*, SoC) qui trouve son origine dans un manque d'abstraction. En effet, les concepteurs d'applications de RV n'identifient pas toujours les concepts structurants des applications de RV, et mêmes lorsqu'ils le font, ils ont tendance à développer leurs propres bibliothèques adaptées à leur contexte. Ces manques de réutilisation et de SoC limitent aussi la capacité de maintenance des applications de RV développés.

Nous avons présenté le domaine de l'Ingénierie Dirigée par les Modèles (IDM), qui cherche à répondre à ces problèmes d'abstraction, de réutilisation et de SoC. Nous nous sommes en particulier intéressés au concept des lignes de produits logiciels (*Software Product Lines*, SPL). Les SPL s'appuient sur la notion de familles de logiciels. Une famille de logiciels est un ensemble de logiciels reposants sur un socle commun et partageant des composants, mais ayant des configurations différentes. Cette différence s'appelle la variabilité logicielle. Les SPL permettent d'exprimer cette variabilité en s'appuyant sur des modèles de variabilité (*Feature Models*, FM). Elles proposent ainsi un processus permettant d'automatiser la production de logiciels appartenant à une même famille.

Nous avons ensuite identifié les principaux concepts structurant les applications de RV comme étant les notions de scénarios, de modèles objets-relations et de domaines métiers. Nous avons constaté que les SPL sont peu adaptées aux logiciels structurés autour d'un scénario comme le sont les applications de RV. De manière générale, il n'existe pas d'approches en IDM permettant d'automatiser le production de tels logiciels.

Nous nous sommes proposés les objectifs suivants, qui découlent des limites identifiées en IDM et en RV :

— **O1-ABSTRACTION** : l'approche proposée doit reposer sur les abstractions des éléments structurant les applications de RV (*e.g.,* notions d'objets virtuels, de comportements, de scénarios, *etc.*) ;

— **O2-RÉUTILISATION** : l'approche proposée doit reposer sur la réutilisation de composants et de modèles issus des abstractions définies ;

— **O3-SOC** : l'approche proposée doit reposer sur le principe de SoC, notamment pour séparer les tâches de modélisation et d'implémentation, mais aussi la définition de scénarios d'une part et la définition des objets 3D et de leurs comportements d'autre part ;

— **O4-SPL-SCÉNARIOS** : l'approche proposée doit appliquer le principe de développement de familles de logiciels des SPL, d'une manière adaptée à la production de logiciels reposant sur un scénario, comme les applications de RV.

Nous avons proposé l'approche LPLOS dans l'objectif de répondre à l'objectif **O4-SPL-SCÉNARIOS**. Cette approche permet de dériver un FM à partir d'un modèle de domaine. Un modèle de scénarios permet de définir des spécifications de scénarios dont chaque état correspond à une configuration de ce FM. Ainsi, le modèle de domaine permet de définir une famille de logiciels et chaque spécification de scénarios permet de produire automatiquement un membre de cette famille. Nous avons montré une application de cette approche pour la production automatique de documentation pour les DSL.

Nous avons ensuite utilisé cette approche afin de définir l'approche LPLRV, permettant de répondre aux objectifs **O1-ABSTRACTION**, **O2-RÉUTILISATION** et **O3-SOC**. Cette approche décrit l'application de RV à produire à travers un modèle objets-relations dérivé du modèle de domaine. Le FM permet alors de décrire les configurations possibles de ce modèle et le scénario permet ainsi de définir les séquences d'actions qui vont transformer l'application de RV afin d'atteindre un but donné. Nous avons montré deux applications possibles de cette approche : l'une pour la production automatique d'évaluations expérimentales en RV et l'autre pour la production automatique d'applications de RV pour la formation aux procédures chirurgicales. Ce dernier cas peut être généralisé à tous types d'applications de RV pour la formation.

2 Travaux futurs

Cette thèse nous a permis de proposer des solutions pour automatiser le développement d'applications de RV par l'abstraction, la réutilisation de composants communs et l'application du principe de SoC. Les abstractions proposées nous ont permis de réduire le fossé existant entre l'expertise métier et l'expertise technique de développement. Nous avons même proposé un DSL pour la production d'évaluations expérimentales en RV. Des des travaux futurs, nous aimerions réduire davantage ce fossé en permettant à l'expert métier de participer de manière significative à la production de telles applications de RV. Nous aimerions lui permettre de configurer et de maintenir lui-même l'application de RV, afin par exemple de l'adapter à de nouveaux besoins. Nous envisageons de proposer des paradigmes reposants sur le concept du "construire en faisant" [Mollet, 2005] : la production de spécifications de scénarios se fait par interaction de l'utilisateur avec un prototype d'application de RV. L'utilisateur peut alors produire le code en interagissant de manière naturelle avec l'application de RV. Nous pensons que ces paradigmes peuvent constituer une extension de notre approche.

Le modèle de scénarios serait exploité d'une manière différente de celle que nous proposons, *i.e.,* une édition de spécifications de scénarios faite à la main par utilisation

d'un langage graphique (*e.g.,* diagrammes d'activités, machines à états hiérarchiques, #SEVEN). De manière générale, nous pensons qu'il peut être intéressant de proposer d'autres approches afin de couvrir des cas d'utilisation plus variés. *Docywood* propose déjà de générer un diagramme d'activité par algorithme pour la documentation de DSL. Il pourrait être intéressant d'exploiter ce concept. Nous pensons que cela pourrait servir à d'autres cas d'utilisations liés à la formation ou l'éducation. L'idée est de pouvoir générer une spécification de scénarios couvrant certains concepts du domaine d'étude (*e.g.,* les mathématiques) et définissant une séquence d'exercices à résoudre. Des chercheurs en Génie Logiciel s'y sont d'ailleurs déjà intéressés [Sadigh et al., 2012, Polozov et al., 2015, Gómez-Abajo et al., 2016].

Nous envisageons également d'étudier la production de spécifications de scénarios par et/ou pour une IA. Un exemple de cas d'utilisation pourrait être la cybersécurité. Ici, le modèle de domaine consisterait en une abstraction des différentes ressources informatiques qui entrent en jeu dans le cadre d'attaques logicielles et des mécanismes de défense associés. Des scénarios d'attaques pourraient être produits et le but pour l'IA pourrait être de proposer un scénario de défense.

Publications de l'auteur

A

Conférences internationales

Gwendal Le Moulec, Ferran Argelaguet, Valérie Gouranton, Arnaud Blouin, and Bruno Arnaldi. 2017. *Agent : automatic generation of experimental protocol runtime.* In Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology (VRST '17). ACM, New York, NY, USA, Article 10, 10 pages. DOI : https ://doi.org/10.1145/3139131.3139152

Gwendal Le Moulec, Ferran Argelaguet, Anatole Lécuyer, and Valérie Gouranton. 2016. *Take-over control paradigms in collaborative virtual environments for training.* In Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology (VRST '16). ACM, New York, NY, USA, 65-68. DOI : https ://doi.org/10.1145/2993369.2993410

Journal

Gwendal Le Moulec, Arnaud Blouin, Valérie Gouranton, Bruno Arnaldi, Automatic Production of End User Documentation for DSLs, Computer Languages, Systems & Structures, 2018, ISSN 1477-8424, https ://doi.org/10.1016/j.cl.2018.07.006. (http ://www.sciencedirect.com/science/article/pii/S1477842417301811) Keywords : Software documentation ; Domain-specific language ; Model slicing

DSL Robot

B

Cette annexe présente le DSL *Robot*, développé lors de la thèse et utilisé comme exemple d'illustration à plusieurs reprises dans le manuscrit. La Figure B.1 décrit le métamodèle du langage. Un utilisateur peut définir un programme (*ProgramUnit*) pour conduire un robot grâce à des commandes (*Command*). Les commandes sont : avancer (*Move*) ; tourner sur soi-même selon un angle donné (*Turn*) ; une boucle *while* un peu particulière qui permet d'exécuter des commandes tant que le robot ne rencontre pas d'obstacles (*WhileNoObstacle*). Tous les éléments du métamodèle sont documentés (La Figure B.1 montre la documentation de trois éléments, respectivement *ProgramUnit*, *ProgramUnit.commands* et *Move*). Cette documentation, écrite par les concepteurs du DSL, est intégrée au métamodèle. Par exemple avec l'outil *EcoreTools*, une telle documentation peut être rédigée sous forme d'annotations des éléments du métamodèle.

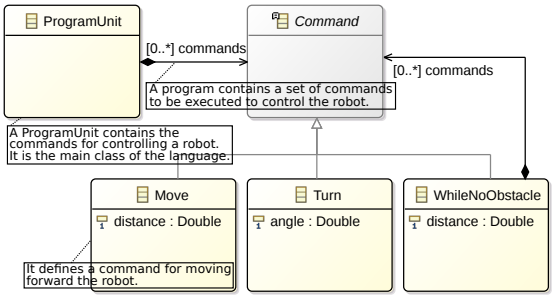


Figure B.1 – Métamodèle du langage *Robot*.

L'extrait de code B.1 est un exemple de modèle *Robot* respectant la grammaire du langage, dont la spécification sous forme de code Xtext [Bettini, 2013] – un outil permettant de concevoir des DSL – est donnée par l'extrait B.2. Un programme commence par la commande *begin* et se termine par la commande *end*. Les commandes *Move*, *Turn* et *WhileNoObstacle* correspondent respectivement aux mots-clés *move*, *turn* et *whileNoObstacleAt*, suivis par leurs paramètres déclarés entre parenthèses. Une commande *WhileNoObstacle* définit ses sous-commandes entre accolades.

Listing B.1 – Un modèle *Robot*

```
begin
  move(25)
  whileNoObstacleAt(10) {
    move(75)
    turn(90)
    move(50)
```

```
  }
  turn(-100)
  move(60)
end
```

Listing B.2 – La grammaire Xtext du DSL *Robot*

```
ProgramUnit: 'begin' (commands+=Command)* 'end';
Command: Move | Turn | WhileNoObstacle;
Move: 'move' '(' distance=Double ')';
Turn: 'turn' '(' angle=Double ')';
WhileNoObstacle: 'whileNoObstacleAt' '(' distance=Double ')' '{'
(commands+=Command)*
'}';
```

Table des figures

Bibliographie

[Acher, 2011] Acher, M. (2011). *Managing, multiple feature models : foundations, languages and applications*. PhD thesis. Thèse de doctorat dirigée par Lahire, Philippe et Collet, Philippe Informatique Nice 2011. 2, 6, 7, 106

[Aguerreche et al., 2009] Aguerreche, L., Duval, T., Lécuyer, A., et al. (2009). Short paper : 3-hand manipulation of virtual objects. *JVRC 2009*. 18, 21, 106

[Albuquerque et al., 2015] Albuquerque, D., Cafeo, B., Garcia, A., Barbosa, S., Abrahão, S., and Ribeiro, A. (2015). Quantifying usability of domain-specific languages : An empirical study on software maintenance. *Journal of Systems and Software*, 101 :245 – 259. 15

[Apel and Kästner, 2009] Apel, S. and Kästner, C. (2009). An overview of feature-oriented software development. *Journal of Object Technology*, 8(5) :49–84. 9

[Arnaldi et al., 2006] Arnaldi, B., Fuchs, P., and Guitton, P. (2006). Introduction à la réalité virtuelle. *Le traité de la réalité virtuelle-3ème édition*, pages Volume–4. 1, 16, 20

[Badawi and Donikian, 2004] Badawi, M. and Donikian, S. (2004). Autonomous agents interacting with their virtual environment through synoptic objects. *CASA 2004*, pages 179–187. 20

[Barišić et al., 2018] Barišić, A., Amaral, V., and Goulão, M. (2018). Usability driven dsl development with use-me. *Computer Languages, Systems & Structures*, 51 :118 – 157. 16

[Barišic et al., 2014] Barišic, A., Amaral, V., Goulão, M., and Barroca, B. (2014). Evaluating the usability of domain-specific languages. In *Software Design and Development : Concepts, Methodologies, Tools, and Applications*, pages 2120–2141. IGI Global. 10, 16

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional. 2

[Bettini, 2013] Bettini, L. (2013). *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing Ltd. 15, 46, 49, 57, 105

[Bezerra and da Silva Barreto, 2014] Bezerra, D. R. and da Silva Barreto, R. (2014). Domain engineering : A practical application in analysis and design of a generative query language. In *Software Components, Architectures and Reuse (SBCARS), 2014 Eighth Brazilian Symposium on*, pages 53–63. IEEE. 12

[Blackwell et al., 2001] Blackwell, A. F., Britton, C., Cox, A., Green, T. R. G., Gurr, C., Kadoda, G., Kutar, M. S., Loomes, M., Nehaniv, C. L., Petre, M., Roast, C., Roes, C., Wong, A., and Young, R. M. (2001). Cognitive dimensions of notations : Design tools for cognitive technology. *Cognitive Technology*, pages 325 – 341. 15

[Blouin et al., 2011] Blouin, A., Combemale, B., Baudry, B., and Beaudoux, O. (2011). Modeling Model Slicers. In *ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, pages 62–76. 49, 57

[Blouin et al., 2015] Blouin, A., Combemale, B., Baudry, B., and Beaudoux, O. (2015). Kompren : modeling and generating model slicers. *Software & Systems Modeling*, 14(1) :321–337. 49, 51, 57

[Bolt, 1980] Bolt, R. A. (1980). *"Put-that-there" : Voice and gesture at the graphics interface*, volume 14. ACM. 21

[Bonet and Geffner, 2001] Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1) :5 – 33. 23

[Bouville et al., 2015] Bouville, R., Gouranton, V., Boggini, T., Nouviale, F., and Arnaldi, B. (2015). #FIVE : High-Level Components for Developing Collaborative and Interactive Virtual Environments. In *Proceedings of Eighth Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS 2015), conjunction with IEEE Virtual Reality (VR)*, Arles, France. 2, 20, 67, 89, 92, 97

[Bowman et al., 2004] Bowman, D., Kruijff, E., LaViola Jr, J. J., and Poupyrev, I. P. (2004). *3D User interfaces : theory and practice, CourseSmart eTextbook*. Addison-Wesley. 20

[Bowman et al., 1999] Bowman, D. A., Johnson, D. B., and Hodges, L. F. (1999). Testbed Evaluation of Virtual Environment Interaction Techniques. In *VRST '99 : Proceedings of the ACM symposium on Virtual reality software and technology*, pages 26–33. ACM. 72

[Bravenboer and Visser, 2004] Bravenboer, M. and Visser, E. (2004). Concrete syntax for objects : domain-specific language embedding and assimilation without restrictions. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 365–383. ACM. 46

[Brooke et al., 1996] Brooke, J. et al. (1996). Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194) :4–7. 25

[Brooks, 1987] Brooks, F. P. J. (1987). No silver bullet essence and accidents of software engineering. *Computer*, 20(4) :10–19. 6

[Burden et al., 2014] Burden, H., Heldal, R., and Whittle, J. (2014). Comparing and contrasting model-driven engineering at three large companies. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 14 :1–14 :10, New York, NY, USA. ACM. 29

[Busetta et al., 2017] Busetta, P., Robol, M., Calanca, P., and Giorgini, P. (2017). Presto script : scripting for serious games. 27

[Cánovas and Cabot, 2013] Cánovas, J. and Cabot, J. (2013). Enabling the Collaborative Definition of DSMLs. In *International Conference on Advanced Information Systems Engineering*, pages 272–287. 46

[Carpentier, 2015] Carpentier, K. (2015). *Dynamic personalized orchestration in virtual environment for training*. Theses, Université de Technologie de Compiègne. 20, 23

[Cavazza et al., 2002] Cavazza, M., Charles, F., and Mead, S. J. (2002). Character-based interactive storytelling. *IEEE Intelligent Systems*, 17(4) :17–24. 2

[Cavazza et al., 2007] Cavazza, M., Lugrin, J.-L., Pizzi, D., and Charles, F. (2007). Madame bovary on the holodeck : Immersive interactive storytelling. In *Proceedings of the 15th ACM International Conference on Multimedia*, MM '07, pages 651–660, New York, NY, USA. ACM. 23

[Challenger et al., 2016] Challenger, M., Kardas, G., and Tekinerdogan, B. (2016). A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. *Software Quality Journal*, 24(3) :755–795. 16

[Chevaillier et al., 2012] Chevaillier, P., Trinh, T. H., Barange, M., Loor, P. D., Devillers, F., Soler, J., and Querrec, R. (2012). Semantic modeling of virtual environments using mascaret. In *2012 5th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS)*, pages 1–8. 2, 20, 22

[Classen et al., 2008] Classen, A., Heymans, P., and Schobbens, P.-Y. (2008). What's in a feature : A requirements engineering perspective. In Fiadeiro, J. L. and Inverardi, P., editors, *Fundamental Approaches to Software Engineering*, pages 16–30, Berlin, Heidelberg. Springer Berlin Heidelberg. 9

[Claude, 2016] Claude, G. (2016). *Actions sequencing incollaborative virtual environment*. Theses, INSA de Rennes. 2, 21, 22, 26, 27, 28, 30, 33, 88, 106

[Claude et al., 2015] Claude, G., Gouranton, V., and Arnaldi, B. (2015). Versatile scenario guidance for collaborative virtual environments. In *10th International Conference on Computer Graphics Theory and Applications (GRAPP'15)*. 27, 88, 89

[Claude et al., 2014] Claude, G., Gouranton, V., Berthelot, R. B., and Arnaldi, B. (2014). Short paper :# seven, a sensor effector based scenarios model for driving collaborative virtual environment. In *ICAT-EGVE, International Conference on Artificial Reality and Telexistence, Eurographics Symposium on Virtual Environments*, pages 1–4. 22, 23

[Clements and Northrop, 2002] Clements, P. and Northrop, L. (2002). *Software product lines : practices and patterns*, volume 3. Addison-Wesley Reading. 2

[Colman, 2015] Colman, A. M. (2015). *A dictionary of psychology*. Oxford University Press, USA. 24, 73

[Cremer et al., 1995] Cremer, J., Kearney, J., and Papelis, Y. (1995). Hcsm : a framework for behavior and scenario control in virtual environments. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 5(3) :242–267. 22, 23

[Creswell, 2013] Creswell, J. W. (2013). *Research design : Qualitative, quantitative, and mixed methods approaches*. Sage publications. 24

[Czarnecki et al., 2006] Czarnecki, K., Hwan, C., Kim, P., and Kalleberg, K. T. (2006). Feature models are views on ontologies. In *10th International Software Product Line Conference (SPLC'06)*, pages 41–51. 9

[de Sousa and da Silva, 2018] de Sousa, L. M. and da Silva, A. R. (2018). Usability evaluation of the domain specific language for spatial simulation scenarios. *Cogent Engineering*, 5(1). 13

[Deelstra et al., 2005] Deelstra, S., Sinnema, M., and Bosch, J. (2005). Product derivation in software product families : a case study. *Journal of Systems and Software*, 74(2) :173–194. 2

[Degueule et al., 2015] Degueule, T., Combemale, B., Blouin, A., Barais, O., and Jézéquel, J.-M. (2015). Melange : A Meta-language for Modular and Reusable Development of DSLs. In *In Proc. of SLE'15*. 46

[Degueule et al., 2016] Degueule, T., Combemale, B., Blouin, A., Barais, O., and Jézéquel, J.-M. (2016). Safe Model Polymorphism for Flexible Modeling. *Computer Languages, Systems and Structures*, 49 :30. 11

[Dragoni et al., 2016] Dragoni, M., Ghidini, C., Busetta, P., Fruet, M., and Pedrotti, M. (2016). An ontology for supporting the evolution of virtual reality scenarios. In Tamma, V., Dragoni, M., Gonçalves, R., and Ławrynowicz, A., editors, *Ontology Engineering*, pages 33–44, Cham. Springer International Publishing. 26, 27

[Durso et al., 2004] Durso, F. T., Dattel, A. R., Banbury, S., and Tremblay, S. (2004). Spam : The real-time assessment of sa. *A cognitive approach to situation awareness : Theory and application*, 1 :137–154. 25

[Favre et al., 2010] Favre, J.-M., Gasevic, D., Lämmel, R., and Pek, E. (2010). Empirical language analysis in software linguistics. In *International Conference on Software Language Engineering*, pages 316–326. Springer. 46

[Field, 2009] Field, A. (2009). *Discovering statistics using SPSS*. Sage publications. 24, 25

[Field and Hole, 2002] Field, A. and Hole, G. (2002). *How to design and report experiments*. Sage. 24, 25, 73

[Foley et al., 1996] Foley, J. D., van Dam, A., Feiner, S. K., and Hughes, J. F. (1996). Computer graphics principles and practice, assison-wesley. *Reading, Massachusetts*. 20

[Fowler, 2005] Fowler, M. (2005). Language workbenches : The killer-app for domain specific languages. 46

[Fowler, 2010] Fowler, M. (2010). *Domain-specific languages*. Pearson Education. 5, 12

[Fuchs et al., 2006] Fuchs, P., Moreau, G., Coquillart, S., and Burkhardt, J.-M. (2006). *Le traité de la réalité virtuelle*, volume 2. Presses des MINES. 1

[Gabbard et al., 1999] Gabbard, J. L., Hix, D., and Swan, J. E. (1999). User-centered design and evaluation of virtual environments. *IEEE computer Graphics and Applications*, 19(6) :51–59. 13, 25

[Gerbaud et al., 2007] Gerbaud, S., Mollet, N., and Arnaldi, B. (2007). Virtual Environments for Training : From Individual Learning to Collaboration with Humanoids. In *Edutainment*, Hong-Kong, Hong Kong SAR China. 22

[Gibaud et al., 2014] Gibaud, B., Penet, C., and Pierre, J. (2014). Ontospm : a core ontology of surgical procedure models. *SURGETICA Google Scholar*. 27, 88, 91

[Giraldo et al., 2016] Giraldo, F. D., España, S., Pastor, Ó., and Giraldo, W. J. (2016). Considerations about quality in model-driven engineering. *Software Quality Journal*. 12, 16, 29

[Glass, 2001] Glass, R. L. (2001). Frequently forgotten fundamental facts about software engineering. *IEEE software*, 18(3) :112–111. 27

[Gómez-Abajo et al., 2016] Gómez-Abajo, P., Guerra, E., and de Lara, J. (2016). A domain-specific language for model mutation and its application to the automated generation of exercises. *Computer Languages, Systems & Structures*. 101

[Grübel et al., 2016] Grübel, J., Weibel, R., Hölscher, C., and Schinazi, V. R. (2016). EVE : A Framework for Experiments in Virtual Environments. In *Proceedings of Spatial Coginition 2016*. 2, 29, 73

[Greenhalgh and Benford, 1995] Greenhalgh, C. and Benford, S. (1995). Massive : a collaborative virtual environment for teleconferencing. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2(3) :239–261. 18

[Gruber, 1995] Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing ? *International Journal of Human-Computer Studies*, 43(5) :907 – 928. 12

[Hachet, 2003] Hachet, M. (2003). *Interaction avec des environnements virtuels affichés au moyen d'interfaces de visualisation collective*. PhD thesis. Thèse de doctorat dirigée par Guitton, Pascal Informatique et mathématiques Bordeaux 1 2003. 20

[Hart and Staveland, 1988] Hart, S. G. and Staveland, L. E. (1988). Development of nasa-tlx (task load index) : Results of empirical and theoretical research. *Advances in Psychology*, 52 :139 – 183. Human Mental Workload. 25

[Hornbæk, 2006] Hornbæk, K. (2006). Current practice in measuring usability : Challenges to usability studies and research. *International Journal of Human-Computer Studies*, 64(2) :79 – 102. 25

[Huang et al., 2012] Huang, W., Alem, L., and Livingston, M. A. (2012). *Human factors in augmented reality environments*. Springer Science & Business Media. 13

[Interrante et al., 2006] Interrante, V., Ries, B., and Anderson, L. (2006). Distance Perception in Immersive Virtual Environments, Revisited. In *IEEE Virtual Reality*, pages 3–10. IEEE. 72

[Jacobson, 1987] Jacobson, I. (1987). Object-oriented development in an industrial environment. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 183–191. ACM. 14

[Johanson and Hasselbring, 2017] Johanson, A. N. and Hasselbring, W. (2017). Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation : a controlled experiment. *Empirical Software Engineering*, 22(4) :2206–2236. 13

[Julier et al., 2001] Julier, S., Feiner, S., and Rosenblum, L. (2001). *Mobile augmented reality : a complex human-centered system*. Springer Science & Business Media. 13

[Kahraman and Bilgen, 2015] Kahraman, G. and Bilgen, S. (2015). A framework for qualitative assessment of domain-specific languages. *Software & Systems Modeling*, 14(4) :1505–1526. 16

[Kang et al., 1990] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst. 7, 8, 106

[Kang et al., 1998] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). Form : A feature-;oriented reuse method with domain-;specific reference architectures. *Annals of Software Engineering*, 5(1) :143. 9

[Kosar et al., 2016] Kosar, T., Bohra, S., and Mernik, M. (2016). Domain-specific languages : A systematic mapping study. *Information and Software Technology*, 71 :77–91. 46, 48

[Krueger, 2002] Krueger, C. (2002). Easing the transition to software mass customization. In van der Linden, F., editor, *Software Product-Family Engineering*, pages 282–293, Berlin, Heidelberg. Springer Berlin Heidelberg. 30

[Krueger, 1992] Krueger, C. W. (1992). Software reuse. *ACM Comput. Surv.*, 24(2) :131–183. 6

[Krueger, 2006] Krueger, C. W. (2006). New methods in software product line development. In *10th International Software Product Line Conference (SPLC'06)*, pages 95–99. 30

[Lamarche and Donikian, 2002] Lamarche, F. and Donikian, S. (2002). Automatic orchestration of behaviours through the management of resources and priority levels. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems : Part 3*, AAMAS '02, pages 1309–1316, New York, NY, USA. ACM. 22

[Latoschik et al., 2016] Latoschik, M. E., Lugrin, J.-L., and Roth, D. (2016). Fakemi : A fake mirror system for avatar embodiment studies. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 73–76, New York, NY, USA. ACM. 24

[Le Moulec et al., 2017] Le Moulec, G., Argelaguet, F., Gouranton, V., Blouin, A., and Arnaldi, B. (2017). AGENT : Automatic Generation of Experimental Protocol Runtime. In *ACM Symposium on Virtual Reality Software and Technology (VRST)*, Virtual Reality Software and Technology, Gothenburg, Sweden. 29, 72, 73

[Le Moulec et al., 2016] Le Moulec, G., Argelaguet, F., Lécuyer, A., and Gouranton, V. (2016). Take-over control paradigms in collaborative virtual environments for training. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 65–68, New York, NY, USA. ACM. 84

[Le Moulec et al., 2018] Le Moulec, G., Blouin, A., Gouranton, V., and Arnaldi, B. (2018). Automatic production of end user documentation for dsls. *Computer Languages, Systems and Structures*. 45, 47

[Leigh and Johnson, 1996] Leigh, J. and Johnson, A. E. (1996). Calvin : an immersimedia design environment utilizing heterogeneous perspectives. In *Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on*, pages 20–23. IEEE. 18

[Logre, 2017] Logre, I. (2017). *Preserving separation of concerns while integrating heterogeneous domains in software systems*. Theses, Université Côte d'Azur. 3

[Louvet and Fleury, 2016] Louvet, J.-B. and Fleury, C. (2016). Combining bimanual interaction and teleportation for 3d manipulation on multi-touch wall-sized displays. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 283–292, New York, NY, USA. ACM. 21

[Margery et al., 1999] Margery, D., Arnaldi, B., and Plouzeau, N. (1999). *A general framework for cooperative manipulation in virtual environments*. Springer. 18, 19

[Meier et al., 2007] Meier, A., Spada, H., and Rummel, N. (2007). A rating scheme for assessing the quality of computer-supported collaboration processes. *International Journal of Computer-Supported Collaborative Learning*, 2(1) :63–86. 25

[Mernik et al., 2005] Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4) :316–344. 5, 12, 46

[Mollet, 2005] Mollet, N. (2005). *De l'objet-Relation au Construire en Faisant : application à la spécification de scénarios de formation à la maintenance en réalité virtuelle*. PhD thesis. Thèse de doctorat dirigée par Arnaldi, Bruno Informatique Rennes 1 2005. 26, 100

[Mollet et al., 2007] Mollet, N., Gerbaud, S., and Arnaldi, B. (2007). STORM : a Generic Interaction and Behavioral Model for 3D Objects and Humanoids in a Virtual Environment. In *IPT-EGVE the 13th Eurographics Symposium on Virtual Environments*, volume Short Papers and Posters, pages 95–100, Weimar, Germany. 20

[Moreau et al., 2018] Moreau, G., Arnaldi, B., and Guitton, P. (2018). *Virtual Reality, Augmented Reality : myths and realities*. Computer engineering series. ISTE. 24, 25

[Mossel and Koessler, 2016] Mossel, A. and Koessler, C. (2016). Large scale cut plane : An occlusion management technique for immersive dense 3d reconstructions. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 201–210, New York, NY, USA. ACM. 84, 85, 86, 87, 108

[Muratet et al., 2009] Muratet, M., Torguet, P., Jessel, J.-P., and Viallet, F. (2009). Towards a serious game to help students learn computer programming. *Int. J. Comput. Games Technol.*, 2009 :3 :1–3 :12. 17

[Nguyen et al., 2014] Nguyen, T. T. H., Duval, T., and Pontonnier, C. (2014). A new direct manipulation technique for immersive 3d virtual environments. In *ICAT-EGVE 2014 : the 24th International Conference on Artificial Reality and Telexistence and the 19th Eurographics Symposium on Virtual Environments*, page 8. 21

[Paiva et al., 2001] Paiva, A., Machado, I., and Prada, R. (2001). Heroes, villians, magicians, &hellip ; : Dramatis personae in a virtual story creation environment. In *Proceedings of the 6th International Conference on Intelligent User Interfaces*, IUI '01, pages 129–136, New York, NY, USA. ACM. 23

[Parnas, 1976] Parnas, D. (1976). On the design and development of program families. SE-2 :1– 9. 2

[Pearl et al., 2014] Pearl, J., Bareinboim, E., et al. (2014). External validity : From do-calculus to transportability across populations. *Statistical Science*, 29(4) :579–595. 24

[Pescador and de Lara, 2016] Pescador, A. and de Lara, J. (2016). Dsl-maps : From requirements to design of domain-specific languages. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 438–443, New York, NY, USA. ACM. 14

[Pinho et al., 2002] Pinho, M. S., Bowman, D. A., and Freitas, C. M. (2002). Cooperative object manipulation in immersive virtual environments : framework and techniques. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 171–178. ACM. 21

[Pohl et al., 2005] Pohl, K., Böckle, G., and van Der Linden, F. J. (2005). *Software product line engineering : foundations, principles and techniques.* Springer Science & Business Media. 2

[Polozov et al., 2015] Polozov, O., O'Rourke, E., Smith, A. M., Zettlemoyer, L., Gulwani, S., and Popovic, Z. (2015). Personalized mathematical word problem generation. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015*, pages 381–388. 101

[Ponder et al., 2003] Ponder, M., Herbelin, B., Molet, T., Schertenlieb, S., Ulicny, B., Papagiannakis, G., Magnenat-Thalmann, N., and Thalmann, D. (2003). Immersive vr decision training : telling interactive stories featuring advanced virtual human simulation technologies. In *Proceedings of the workshop on Virtual environments 2003*, pages 97–106. ACM. 23

[Sadigh et al., 2012] Sadigh, D., Seshia, S. A., and Gupta, M. (2012). Automating exercise generation : A step towards meeting the MOOC challenge for embedded systems. In *Proceedings of the Workshop on Embedded and Cyber-Physical Systems Education*, page 2. ACM. 101

[Schmidt, 2006] Schmidt, D. C. (2006). Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY*, 39(2) :25. 5

[Schubert et al., 2001] Schubert, T., Friedmann, F., and Regenbrecht, H. (2001). The experience of presence : Factor analytic insights. *Presence : Teleoperators and Virtual Environments*, 10(3) :266–281. 25

[Shadish et al., 2002] Shadish, W. R., Cook, T. D., and Campbell, D. T. (2002). *Experimental and quasi-experimental designs for generalized causal inference.* Wadsworth Cengage learning. 24

[Slater, 2009] Slater, M. (2009). Place illusion and plausibility can lead to realistic behaviour in immersive virtual environments. *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, 364(1535) :3549–3557. 72

[Slater et al., 1994] Slater, M., Usoh, M., and Steed, A. (1994). Depth of presence in virtual environments. *Presence : Teleoperators and Virtual Environments*, 3(2) :130–144. 25

[Sprinkle et al., 2009] Sprinkle, J., Mernik, M., Tolvanen, J.-P., and Spinellis, D. (2009). What kinds of nails need a domain-specific hammer ? *IEEE software*, 26(4). 46

[Stanney et al., 2003] Stanney, K. M., Mollaghasemi, M., Reeves, L., Breaux, R., and Graeber, D. A. (2003). Usability engineering of virtual environments (ves) : identifying multiple criteria that drive effective ve system design. *International Journal of Human-Computer Studies*, 58(4) :447 – 481. 25

[Steinberg et al., 2008] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF : eclipse modeling framework.* Pearson Education. 11, 47, 57

[Szilas, 2003] Szilas, N. (2003). Idtension : a narrative engine for interactive drama. In *Proceedings of the technologies for interactive digital storytelling and entertainment (TIDSE) conference*, volume 3, pages 1–11. 23

[Tolvanen and Kelly, 2009] Tolvanen, J.-P. and Kelly, S. (2009). Metaedit+ : Defining and using integrated domain-specific modeling languages. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 819–820. ACM. 48

[Tromp et al., 2003] Tromp, J. G., Steed, A., and Wilson, J. R. (2003). Systematic usability evaluation and design issues for collaborative virtual environments. *Presence : Teleoperators and Virtual Environments*, 12(3) :241–267. 25

[Uddin and Robillard, 2015] Uddin, G. and Robillard, M. P. (2015). How API documentation fails. *IEEE Software*, 32(4) :68–75. 46

[Van Deursen and Klint, 2002] Van Deursen, A. and Klint, P. (2002). Domain-specific language design requires feature descriptions. *CIT. Journal of computing and information technology*, 10(1) :1–17. 14, 47

[van Deursen et al., 2000] van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages : An annotated bibliography. *SIGPLAN Not.*, 35(6) :26–36. 5, 12

[Voelter et al., 2013] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C., Visser, E., and Wachsmuth, G. (2013). *DSL engineering : Designing, implementing and using domain-specific languages*. CreateSpace. 46

[Whittle et al., 2011] Whittle, J., Clark, T., and Kühne, T. (2011). Model driven engineering languages and systems. In *14th International Conference, MODELS*, pages 16–21. Springer. 29

[Whittle et al., 2014] Whittle, J., Hutchinson, J., and Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE Software*, 31(3) :79–85. 29

[Wile, 2004] Wile, D. (2004). Lessons learned from real dsl experiments. *Science of Computer Programming*, 51(3) :265 – 290. 2, 6, 13, 16

[Willans and Harrison, 2001] Willans, J. S. and Harrison, M. D. (2001). Prototyping pre-implementation designs of virtual environment behaviour. In Little, M. R. and Nigay, L., editors, *Engineering for Human-Computer Interaction*, pages 91–108, Berlin, Heidelberg. Springer Berlin Heidelberg. 20

[Witmer and Singer, 1998] Witmer, B. G. and Singer, M. J. (1998). Measuring presence in virtual environments : A presence questionnaire. *Presence : Teleoperators and Virtual Environments*, 7(3) :225–240. 25

[Yue et al., 2009] Yue, T., Briand, L. C., and Labiche, Y. (2009). A use case modeling approach to facilitate the transition towards analysis models : Concepts and empirical evaluation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 484–498. Springer. 14

[Ziadi et al., 2002] Ziadi, T., Hélouët, L., and Jézéquel, J.-M. (2002). Modeling behaviors in product lines. In *Proceedings of REPL'02 (workshop on Requirements Engineering for Product Lines)*, Essen, Germany. 3, 10, 30

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Synthèse d'applications de Réalité Virtuelle à partir de modèles

Nom Prénom de l'auteur : LE MOULEC GWENDAL

Membres du jury :
- Monsieur ARNALDI BRUNO
- Monsieur BLOUIN Arnaud
- Madame GOURANTON Valérie
- Monsieur JESSEL Jean-Pierre
- Monsieur MESTRE Daniel
- Monsieur ZIADI Tewfik

Président du jury : Daniel MESTRE

Date de la soutenance : 26 Septembre 2018

Reproduction de la these soutenue

☒ Thèse pouvant être reproduite en l'état
☐ Thèse pouvant être reproduite après corrections suggérées

Fait à Rennes, le 26 Septembre 2018

Signature du président de jury

Le Directeur,

M'hamed DRISSI

Titre : Synthèse d'applications de Réalité Virtuelle à partir de modèles

Mots clés : SPL, DSL, scénario, ontologie, modèles objets-relations

Résumé : Les pratiques de développement des logiciels de Réalité Virtuelle (RV) ne sont pas optimisées. Ainsi, chaque société utilise ses propres méthodes. L'objectif de la thèse est d'automatiser la production et l'évaluation des logiciels de RV en utilisant des techniques issues de l'Ingénierie Dirigée par les Modèles (IDM).
Les approches existantes en RV ne permettent pas de tirer parti des points communs que partagent les applications de RV.
Ces manques de réutilisation, d'abstraction et de séparation des préoccupations sont des problèmes connus en IDM, qui propose le concept de Ligne de Produits Logiciels (*Software Product Line*, SPL) pour automatiser la production de logiciels de la même famille par réutilisation de composants communs. Cependant cette approche n'est pas adaptée au développement de logiciels reposant sur un scénario, comme en RV.
Nous proposons deux approches qui comblent respectivement les manques en IDM et en RV : LPLOS (SPL Orientée Scénario) et LPLRV (SPL pour la RV). LPLOS repose sur un modèle de scénarios qui manipule un modèle de variabilité logicielle (*Feature Model*, FM). Chaque étape du scénario correspond à une configuration du FM. LPLRV repose sur LPLOS. Le scénario supervise la manipulation des objets virtuels, générés automatiquement à partir d'un modèle.
Nous avons implémenté ces approches au sein d'outils qui ont été essayés sur des exemples et évalués par des utilisateurs cibles. Les résultats soutiennent l'utilisation de ces approches pour la production de logiciels reposant sur un scénario.

Title : Model-driven Virtual Reality applications synthesis

Keywords: SPL, DSL, scenario, ontology, object-relation model

Abstract: Development practices in Virtual Reality (VR) are not optimized. For example, each company uses its own methods. The goal of this PhD thesis is to automatize development and evaluation of VR software with the use of Model-Driven Engineering (MDE) technics.
The existing approaches in VR do not take advantage of software commonalities. Those lacks of reuse, separation of concerns and abstraction are known problems in MDE, which proposes the Software Product Line (SPL) concept to automatize the production of software belonging to the same family, by reusing common components. However, this approach is not adapted to software based on a scenario, like in VR.
We propose two approaches that respectively address the lacks in MDE and VR: LPLOS (Scenario-Oriented Software Product Line) and LPLRV (VR SPL). LPLOS is based on a scenario model that handles a software variability model (Feature Model, FM). Each scenario step matches a configuration of the FM. LPLRV is based on LPLOS. The scenario manages virtual objects manipulation, the objects being generated automatically from a model.
We implemented these approaches inside tools that have been tried on examples and evaluated by their target users. The results promote the use of these frameworks for producing scenario-based software.