Vers une méthodologie dédiée à l'orchestration d'entités communicantes

Zoé Drey

▶ **To cite this version:**

Zoé Drey. Vers une méthodologie dédiée à l'orchestration d'entités communicantes. Langage de programmation [cs.PL]. Université Sciences et Technologies - Bordeaux I, 2010. Français. NNT : . tel-00718634

HAL Id: tel-00718634
https://theses.hal.science/tel-00718634

Submitted on 17 Jul 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Bordeaux 1

Département de formation doctorale en informatique

École doctorale EDMI Bordeaux

N° d'ordre : 4061

Vers une méthodologie dédiée à l'orchestration d'entités communicantes

THÈSE

soutenue le 22/09/2010

pour l'obtention du

Doctorat de l'Université de Bordeaux 1

(spécialité informatique)

par

Zoé DREY

Jury

Président :	Pascal GUITTON,	Professeur à l'Université de Bordeaux 1
Rapporteurs :	Franck Barbier,	Professeur à l'Université de Pau
	Olivier Danvy,	Professeur à l'Université d'Aarhus
Examinateurs :	Charles CONSEL,	Professeur à l'ENSEIRB
	Pascal GUITTON ,	Professeur à l'Université de Bordeaux 1
	Vicente SANCHEZ-LEIGHTON,	Docteur en Mathématique, co-fondateur de la société Hyptique

Laboratoire Bordelais de Recherche en Informatique — UMR 5800

Remerciements

Je remercie les membres du jury, Franck Barbier et Olivier Danvy d'avoir accepté de rapporter cette thèse et pour la richesse de leur commentaires, Pascal Guitton d'avoir bien voulu présider le jury de cette thèse, Vicente Sanchez-Lighton d'avoir bien voulu participer à ce jury et pour nos discussions enrichissantes sur l'assistance à la personne.

Je remercie Charles Consel de m'avoir donné cette chance immense de faire une thèse en ayant simplement cru à mon potentiel, de m'avoir dispensé une excellente formation à la recherche et de m'avoir permis de terminer cette thèse dans de bonnes conditions.

Je remercie Sylvie Embolla, Cathy Roubineau et Christine Parison pour nos diverses interactions.

Je remercie chaleureusement mes collègues de l'équipe Phoenix, featuring Quentin Enard, pour les bons moments passés ensemble et pour leur aide importante dans l'évaluation de Pantagruel auprès des étudiants "programmeurs novices". Merci non galvaudé à Laurent Burgy pour tous ses bons conseils.

Merci à Alexandre Blanquart, Ghislain Desffrasnes et Benjamin Bertran P-B (Hi !) d'avoir collaboré avec moi sans craquer et sans qui Pantagruel serait resté petit et menu (et vilain). Alexandre a posé les fondations du compilateur de Pantagruel, Ghislain a repris le flambeau avec brio, et m'a beaucoup soutenue lors des démonstrations du langage. Merci spécial à Benjamin, arrivé à mon secours avec patience, efficacité, et sens critique. Je regretterai particulièrement son sens de l'humour sans pareil.

Je remercie Julien Bruneau pour tout l'intérêt qu'il a manifesté pour mon travail, en particulier sur la sémantique et pour avoir relu cette partie de ma thèse en détail. Merci de m'avoir aidé, en répondant même à mes mails de détresse du dimanche sur le simulateur derrière lequel le gros Pantagruel se cachait. Et merci pour les happy-hours, avec Nicolas, qui m'ont aidée à tenir bon jusqu'au début de la rédaction (et un peu pendant...).

Je rends un vibrant hommage à Damien Cassou qui a relu et corrigé ma thèse avec une patience et une précision incomparables, tout en diplomatie. C'était également un plaisir d'aller aux conférences avec lui (surtout quand les avions sont en retard). Enfin, je le remercie d'avoir accepté de si souvent donner son temps pour me venir en aide avec grande gentillesse. Si je le pouvais, j'écrirais sa biographie, en témoignage de ma profonde gratitude.

Je suis extrêmement reconnaissante envers Nicolas Loriant qui a été un très agréable et attentionné co-bureau et ami pendant la dernière année de ma thèse. Ses connaissances, son recul et son esprit critique aiguisé m'ont été de nombreuses fois d'une grande utilité (même si ma mauvaise foi m'a parfois interdit de le montrer), y compris pour la rédaction. Son arrivée dans l'équipe a changé le paysage du smart-space (y a des miettes de chocolatine partout) et beaucoup détendu l'atmosphère. Merci Nicolas, pour tout ce que nous avons partagé pendant ton séjour dans l'équipe.

Je remercie Julien Mercadal parce que cette thèse est aussi le fruit d'une collaboration intense avec lui tout au long de ces presque-quatre années. Je garde également de bons souvenirs de nos conversations complices sur La Recherche et le reste. Je suis heureuse d'avoir vécu ces années en sa compagnie : dans notre quotidien au labo, lors des moments passés en dehors, à Braga, et dans les difficiles périodes de doute, où il a été présent et encourageant.

Grand merci à Christophe Calvès, qui, depuis qu'on a fait connaissance au Danemark, a été là lorsque j'étais perdue et a passé des milliers d'heures avec moi : pour écrire de la sémantique et rire des continuations ; pour me donner de nombreux retours sur ma thèse ; pour tester les bières exotiques en conversant à toute heure de la nuit danoise.

Je remercie infiniment Olivier pour tout ce qu'il m'a apporté pendant et après les deux mois passés au Danemark. J'ai eu la chance de bénéficier, au sein d'un environnement (à portée) dynamique, de ses exceptionnelles qualités scientifiques et humaines, de sa rigueur et de sa pédagogie. Au cours de mon séjour à Aarhus je n'ai jamais eu autant envie de dépasser mes limites. Et grâce à lui, j'ai découvert de nouvelles thématiques de recherche passionnantes. J'espère avoir l'occasion de continuer à les explorer.

Merci à Damien Pollet de m'avoir permis d'avoir une jabber-vie sociale pendant la rédaction et de m'avoir aidé à la démarrer, en particulier grâce à la jolie métaphore fractale. Merci à Paul Brauner et Emilie Balland pour l'épique film germanophone sur le mystère de Die Schink. Je remercie aussi Emilie pour son modèle de thèse que j'ai éhontément plagié et surtout : pour s'être intéressée à mon travail dès son arrivée, pour m'avoir énormément soutenue dans le passage difficile de la fin de thèse. Merci à Pierre, Vincent, Benoît, Laure et Emilie pour les importants moments de détente sportive à Roca'.

Merci immense à Anne Veyron, Catherine Meunier, Patrice Cotty et Nicole Penard, qui m'avez accordé de précieuses heures pour m'expliquer les métiers de l'assistance à la personne, et qui avez directement inspiré beaucoup d'éléments de cette thèse. Je suis spécialement reconnaissante envers Nicole pour toute son amitié et son soutien constant tout au long de cette thèse.

Enfin, Pierre a été d'un soutien intellectuel et moral incommensurable, il m'a inlassablement encouragée et je lui dois absolument tout : je te dois au moins quarante-deux pages de remerciements pour m'avoir aidé à réfléchir, à relever la tête quand je baissais les bras, à écrire du français et à déparaphraser, à présenter oralement, à me détendre et à prendre confiance en moi, tout en supportant mes rétrogradages récurrents et mon caractère difficile : je suis rigolote, mais ça n'a pas dû être facile tous les jours. Je remercie enfin ma famille, en particulier ma sœur Valérie qui a assisté à ma soutenance.

Et ici est aussi l'ultime occasion de remercier mon ancienne équipe Triskell, et en particulier Benoît Baudry, Didier Vojtisek, Franck Fleurey, Noël Plouzeau, Jim Steel et Jean-Marc Jézéquel, grâce à qui j'ai eu l'envie et la possibilité de faire cette unique expérience. Vous autres amis que je ne cite pas et qui ont été là avec et sans thèse, vous avez toute ma vie pour être remerciés grandiosement.

Table des matières

Table des figures

Abstract

Networked technologies, omnipresent in our surroundings, have increasingly more computing power offering interfaces to easily access their functionalities. These technologies offer a wide testing ground for research, especially in applied computer science. Because people are permanently interacting with these technologies, they form an evident platform to help people in their daily activities. Applications that address people needs are found in various application areas, each related to specific goals : comfort, security, information management, or assisted living. These goals are found in a range of assisted-living scenarios for people with cognitive deficiencies.

Because the residents of these spaces are intimately concerned with these applications, it is crucial to make them easily adaptable to their needs and their preferences. Yet, programming such applications remains very challenging. Indeed, it requires expertise in a number of areas, and in particular in distributed computing. A known way to achieve this challenge is to offer the user the means to program his own applications. This process has also been widely explored by the pervasive computing community. To enable end-user accessibility, these approaches provide a visual interface and a domain-specific vocabulary (*e.g.*, iCAP [DSSK06]) or use a metaphor-based representation (*e.g.*, CAMP [THA04]). However, this accessibility is obtained at the expense of expressiveness, resulting in hardly reusable and adaptable tools for a range of application areas. Therefore, they often address a limited range of user needs.

The goal of this thesis is to propose an approach that bridges the gap between the large variety of user requirements and the applications that satisfy them. To do so, we define Pantagruel, an expressive and accessible language to develop these applications ; it is parameterized by an application area, and it is supported by a visual editor. To further reconcile accessibility with expressiveness, we provide the user with a domain-specific methodology to guide the development of applications. This methodology draws a bridge between the user needs and the applications, and is strongly coupled with the language concepts. In doing so, satisfying the requirements and evolving the applications according to new requirements is facilitated. Because the applications aim at being seamlessly integrated in the user everyday life, the language must also guarantee that they are reliable. To this end, we extend our methodology with a programming approach driven by properties. These properties can be verified using the language semantics, which is formally defined.

A tool-based methodology for prototyping orchestration applications

To make applications easily adaptable, it is necessary to bridge the gap between the user requirements and the application development. The goal of this thesis is to define a methodology that draws such a bridge. It is founded on a thorough study of the assisted-living application area. Specifically, we studied how caregivers build assistive strategy to support an impaired person in his everyday life. It consists of an elementary process where initial requirements are decomposed into increasingly specific goals until the identification of objects that would achieve each of them. These objects are the building blocks for describing the sub-goals as orchestration rules. The next step consists on modelling these objects to enable access to their functionalities and characteristics. Such modeled objects are called *networked entities*. Developing an application amounts to orchestrating these entities to realize the goals. The methodology makes explicit the correspondance between the sub-goals and the applications, thus guiding the developer in adapting the applications to new requirements.

Networked-entity orchestration

Entities deployed in the physical environment where a user lives are the key elements of pervasive computing applications. To make their development accessible to a non-programmer, the functionalities and data provided by entities must be exposed at a high abstraction level. Our approach consists of modelling the entities that compose pervasive computing environments. To do so, we defined Pantagruel, a domain-specific language composed of two language layers. Specifically, we defined a *taxonomy* language to enable a high-level and uniform description of the interaction interface of networked entities. Specifically, this interaction interface give access to the functionalities and the informations necessary to manage a specific application area. The set of entities described via such interfaces form an entity taxonomy. They are then manipulated through an *orchestration* language within if-then rules, where the informations captured by the entities are the rule conditions, and the invoked functionalities are its actions. In this thesis, we expose the key concepts that characterize entity orchestration.

To enable the analysis of the orchestration logic defined in Pantagruel applications, we propose a denotational semantics of the language, highlighting each of the key concepts. To enable application testing, we developed a compiler targeting an execution platform dedicated to pervasive computing applications, based on this semantics.

Towards the development of reliable applications

One of the characteristics of rules is that it is easy to understand the purpose of each rule separately. However, they hardly enable to understand the global behavior of an application. This results in a higher risk to create conflicts while developing rules, unless all the rule logical alternatives are considered by the developer [NMB09]. Furthermore, to convince a non-programmer to adopt a programming tool, it is necessary to ensure that applications do behave as he expects. To do so, applications can be tested on an execution platform (*e.g.* a simulator [BJC09]). We propose to check rule conflicts by using the information provided by the taxonomy. We also propose an approach to guide the user in developing programs that guarantee some behavior

under a particular context. We show how Pantagruel semantics enables the verification of a few properties : non-interference, which is a language property, and action invariants, which are applicative properties (*i.e.* specific to a program). An originality of this approach is that it enables the user to express invariant properties as program patterns. To do so we give a formal definition of these properties, based on the concrete semantics of our language.

Evaluation of the language expressiveness and accessibility

To validate the expressiveness of Pantagruel, we modeled some application areas and developed orchestration applications for these areas. A contribution of this evaluation is the definition of a *design space of entities* of a pervasive computing environment, based on the analysis of the application areas that we targeted. In doing so, we identified the application boundaries of our approach and measured the expressiveness of the taxonomy-language layer. We also evaluate the expressiveness of the orchestration-language layer using an existing classification of pervasive computing applications [SAW94]. In particular, we show that the taxonomy-language layer enhances the expressiveness of the orchestration-language layer. Finally, we evaluate the accessibility of the orchestration language for novice programmers. This evaluation illustrates the accessibility of the language, which is one of the main advantages of domain-specific and visual approaches.

Outline

Next chapter introduces ubiquitous computing as the application area of our contributions. Its definition is centered around the end-user of these applications. We then expose the problematic of developing applications according to variety of end-user requirements in this area. Chapter 3 gives an overview of existing approaches to ease the development of applications, which are applied in the contributions explained in the following chapters. Chapter 4 introduces a domain-specific methodology for guiding the development of orchestration applications, from end-user requirements to the deployment of these applications. Chapter 5 defines the Pantagruel language : its motivations, its key concepts, its semantics, and its compilation towards an execution platform. Chapter 6 illustrates the coupling between the methodology and the language, reusing the case study introduced in chapter 4. Chapter 7 exposes a development process based on verification. Chapter 8 exposes a complete evaluation of the expressiveness and accessibility of our approach. Concluding remarks and future work are provided in Chapter 9.

1 Introduction

Les technologies omniprésentes dans notre environnement intègrent désormais des éléments logiciels facilitant l'accès à leurs fonctionnalités. Ces technologies, déployées dans des réseaux, offrent un vaste laboratoire d'expérimentation pour la recherche et en particulier pour l'informatique appliquée. Les personnes étant en contact quasi-permanent avec elles, ces technologies sont un support évident pour leur rendre des services dans leur vie quotidienne. Ces services concernent divers champs d'applications, chacun servant des objectifs spécifiques : confort, sécurité, accès à l'information ou encore assistance quotidienne. Tous ces objectifs se retrouvent par exemple dans l'assistance aux personnes avec déficiences cognitives. Puisque les applications offrant ces services sont intimement liées aux besoins des utilisateurs, il est indispensable qu'elles s'adaptent facilement à leurs besoins et soient personnalisées selon leurs préférences.

Toutefois, de nombreuses compétences techniques sont requises pour développer de telles applications, ce qui rend difficile la programmation et en particulier l'adaptation des applications aux nombreux besoins de l'utilisateur. Offrir aux utilisateurs les outils leur permettant de programmer eux-mêmes ces applications est une approche largement explorée par la communauté de l'informatique ubiquitaire pour aborder ce problème.

Cette démarche permet de réduire la distance entre les besoins des utilisateurs et leur réalisation par l'application. Malheureusement, ces outils sont souvent peu flexibles, car leur accessibilité est souvent acquise au prix d'une expressivité réduite à un champ d'applications spécifique. Par conséquent, ils ne permettent de répondre qu'à une variété limitée de besoins.

Notre objectif est de proposer une approche établissant une passerelle entre les besoins de l'utilisateur et leur réalisation dans divers champs d'applications. Il s'agit non seulement d'offrir un langage qui concilie expressivité et accessibilité, mais également une méthodologie afin de guider l'utilisateur dans la programmation d'applications en utilisant ce langage. En explicitant la relation entre les besoins et les applications, la méthodologie sert de support pour faciliter à la fois la réalisation des besoins et l'adaptation des applications lorsque de nouveaux besoins apparaissent. L'approche de développement que nous proposons à l'utilisateur doit également lui garantir que les applications sont fiables car elles ont pour finalité d'être intégrées à sa vie quotidienne.

Méthodologie outillée pour la programmation d'applications

Pour faciliter l'adaptation des applications, il faut réduire le fossé séparant les besoins des utilisateurs du développement d'applications. Le but de cette thèse est de définir une méthodologie proposant un premier pas dans cette direction. Cette méthodologie est fondée sur une étude menée auprès d'éducateurs spécialisés dans l'assistance aux personnes avec déficiences cognitives. Elle repose sur un principe élémentaire qui consiste à analyser un besoin en le décomposant en buts jusqu'à atteindre des objets élémentaires. Ces objets sont les briques de base permettant de développer une application. Reste alors à modéliser ces objets afin d'exposer les fonctionnalités et les informations qu'ils peuvent mettre à disposition pour les applications. Nous appelons ces objets modélisés des entités communicantes. Le développement d'applications, guidé par les sous-buts, consiste alors à orchestrer ces entités. Grâce à la méthodologie, la correspondance entre les applications et les sous-buts est explicitée ; le développeur d'applications est ainsi guidé pour l'adaptation de ces applications à de nouveaux besoins.

Orchestration d'entités communicantes

Les entités contenues dans les espaces physiques dans lesquels vivent les utilisateurs sont les éléments de base des applications de l'informatique ubiquitaire. Pour faciliter le développement d'applications par un utilisateur non-programmeur, il est nécessaire de rendre accessibles les fonctionnalités et les informations sur l'environnement que les entités fournissent et recueillent.

Notre approche consiste à modéliser les entités constituant les espaces de l'informatique ubiquitaire. Il s'agit d'un langage de *taxonomie* qui permet de décrire les interfaces d'interaction des entités de manière homogène et haut niveau. Plus précisément, ces interfaces d'interaction mettent à disposition les fonctionnalités et les informations nécessaires à un champ d'application particulier. L'ensemble des entités décrites par ces interfaces constitue une taxonomie d'entités. Celles-ci sont ensuite manipulées par un langage d'*orchestration* utilisant le paradigme à base de règles, où les informations recueillies par les entités sont les conditions de ces règles, et les opérations faisant appel aux fonctionnalités de ces entités en sont les actions.

Nous exposons dans cette thèse les concepts clés caractérisant l'orchestration d'entités. Ces concepts constituent la base de Pantagruel, un langage dédié à l'expression d'applications d'orchestration et composé des sous-couches langage que nous venons d'introduire. Afin de raisonner sur la logique d'orchestration définie par les applications, nous formalisons la sémantique du langage Pantagruel, qui met en évidence chacun des concepts clés qui le caractérisent. Cette sémantique sert ensuite de support pour le développement d'un compilateur de Pantagruel vers une plate-forme dédiée au déploiement d'applications de l'informatique ubiquitaire, permettant le test des applications développées.

Vers une approche dirigée par les propriétés

Lorsqu'on développe des applications à l'aide de règles, il est facile de comprendre l'objectif de ces règles individuellement. Cependant, les approches à base de règles ne permettent pas d'avoir une vision globale de l'application. Cela soulève un problème spécifique : des conflits peuvent facilement apparaître entre les règles si tous les cas de figure des conditions ne sont

pas envisagés. D'autre part, permettre à l'utilisateur non-programmeur de s'assurer que les applications se comportent comme il l'attend est une condition importante pour le convaincre d'adopter un outil de programmation. Nous proposons un moyen de détecter l'existence de conflits en utilisant les informations de la taxonomie. Nous proposons également un moyen de guider l'utilisateur dans l'écriture de programmes, ce qui lui permet de garantir certains comportements selon un contexte particulier.

Pour cela nous montrons que la sémantique du langage Pantagruel facilite la vérification de quelques propriétés : la non-interférence, qui est une propriété du langage, et les invariants d'actions, qui sont des propriétés spécifiques à une application. Du point de vue de l'utilisateur, ces invariants peuvent être vus comme des motifs de programmes. Pour cela nous donnons une définition de ces propriétés, fondée sur les éléments de la sémantique concrète du langage.

Evaluation de l'expressivité et de l'accessibilité du langage

Afin de valider l'expressivité de notre langage, nous avons modélisé plusieurs champs d'applications, pour lesquels nous avons défini et déployé des applications d'orchestration. L'une des contributions de cette évaluation est la définition d'un *espace de modélisation* des entités d'un environnement ubiquitaire, fondé sur une analyse des champs d'applications auxquels nous nous sommes intéressés. Nous avons ainsi identifié le périmètre d'applications de notre approche et mesuré l'expressivité du langage de taxonomie. Nous évaluons également l'expressivité du langage d'orchestration à l'aide d'une classification existante sur les applications de l'informatique ubiquitaire [SAW94]. Cette évaluation permet notamment de montrer que, grâce au langage de taxonomie, de nombreuses applications peuvent être envisagées par le langage d'orchestration.

Enfin, nous évaluons la facilité de programmation d'applications à l'aide du langage d'orchestration par des programmeurs novices. Cette évaluation met en évidence l'accessibilité du langage, qui est l'un des apports principaux caractérisant les approches dédiées.

Plan de la thèse

Dans le chapitre suivant, nous présentons une définition de l'informatique ubiquitaire centrée sur l'utilisateur final des applications de ce domaine. Nous y exposons la problématique du développement d'applications pour répondre aux besoins des utilisateurs. Nous présentons ensuite dans le chapitre 3 des moyens de faciliter le développement d'applications, que nous mettons en pratique dans les contributions contenues dans les chapitres suivants :

- Le chapitre 4 donne une vue générale de la méthodologie que nous proposons pour développer des applications répondant aux besoins des utilisateurs. Cette méthodologie s'inspire d'une étude que nous avons menée auprès d'éducateurs spécialisés dans l'assistance à la personne ; en effet, les applications de l'informatique ubiquitaire s'avèrent particulièrement utiles dans ce domaine. Nous illustrons la première étape de cette méthodologie avec un cas d'étude.
- Dans le chapitre 5, nous exposons le langage visuel Pantagruel qui est composé de deux sous-couches : le langage de taxonomie et le langage d'orchestration. Le langage de taxonomie permet de modéliser les informations de contexte communiquées par les entités et leurs

fonctionnalités. Le langage d'orchestration est paramétré par une taxonomie, et permet d'orchestrer les entités modélisées par cette taxonomie. Nous présentons dans ce chapitre les concepts clés qui posent les fondations de Pantagruel. Nous définissons sa sémantique dénotationnelle, qui sert de support pour sa compilation vers une plate-forme d'exécution.
- Le chapitre 6 illustre la deuxième étape de la méthodologie sur le cas d'étude introduit dans le chapitre 4. Nous montrons comment utiliser le langage Pantagruel pour guider la construction d'applications répondant aux besoins d'un utilisateur.
- Dans le chapitre 7, nous exposons un processus de développement d'applications guidé par l'expression de propriétés applicatives, c'est-à-dire les propriétés définies pour un programme particulier. Ce processus vient compléter la méthodologie en permettant à l'utilisateur d'exprimer un programme sous forme de propriétés. Nous montrons en particulier comment la sémantique formelle du langage nous permet d'implémenter facilement ces propriétés et de rendre ainsi les programmes fiables.
- Le chapitre 8 contient une évaluation complète de l'expressivité et de l'accessibilité de notre approche, d'abord au travers d'un état de l'art des approches ayant des caractéristiques communes à la nôtre, ensuite par une évaluation qualitative. Pour cela nous établissons le périmètre d'expressivité de Pantagruel et nous analysons les résultats d'une étude d'accessibilité menée auprès de programmeurs novices.

En guise de conclusion, nous présentons dans le chapitre 9 quelques perspectives de recherche qui peuvent faire suite à nos travaux.

2 L'utilisateur au centre de l'informatique (ubiquitaire)

Dans ce chapitre, nous donnons une définition de l'*informatique ubiquitaire*, qui est une notion centrale dans cette thèse. Nous examinons ensuite la relation entre les utilisateurs et les applications qui sont développées dans ce domaine. Cette relation est établie en identifiant les utilisateurs de l'informatique ubiquitaire, mettant en lumière les besoins d'*adaptation* de ces applications pour ces utilisateurs.

2.1 L'informatique ubiquitaire

L'informatique ubiquitaire désigne l'ensemble des technologies informatiques présentes dans notre quotidien. Une application de l'informatique ubiquitaire, que nous appellons désormais une application ubiquitaire, est un programme informatique utilisant ces technologies. Les nombreuses applications ubiquitaires peuvent être classées par *champs d'applications*, selon les services qu'elles rendent aux personnes. Après avoir donné une définition générale d'un environnement de l'informatique ubiquitaire, nous en présentons trois champs représentatifs.

2.1.1 Une définition

Un *environnement de l'informatique ubiquitaire* (ou environnement ubiquitaire) est un espace physique (une maison, un hôpital, une école, une autoroute, ou une ville) équipé d'une multitude d'*entités* matérielles ou logicielles communiquant grâce à un réseau. Ces entités sont des capteurs (de température, de luminosité), des actionneurs (lampes, caméra), des technologies mobiles (téléphones portables multi-fonctions), ou encore des composants logiciels (agendas, applications web). Ces entités capturent les informations de l'environnement ubiquitaire et sont programmables, donnant ainsi accès à leurs fonctionnalités. L'informatique ubiquitaire est aussi appelée informatique "sensible au contexte", où le contexte représente les informations capturées par les entités.

2.1.1.1 Finalité

L'objectif principal d'un environnement ubiquitaire est de rendre service à ses habitants en améliorant leur vie quotidienne. Cette amélioration concerne le confort, la sécurité, l'accessibilité de l'information, ou encore l'assistance dans les activités quotidiennes des personnes. Des applications originales comme la cuisine intelligente [KRM+06], le guide touristique personnalisé [CDM+00], ou l'entretien automatisé de plantes [LBK+02], ne relèvent désormais plus de la science-fiction.

Prenons le cas d'un système d'entretien des plantes dans un bâtiment professionnel. Un tel système pourrait suppléer le jardinier en cas d'absence. On peut imaginer une application de gestion des plantes qui, au moyen d'appareils d'arrosage adaptés, mesure régulièrement l'humidité de la terre, consulte une base de données donnant les caractéristiques des plantes, et fournit la quantité d'eau nécessaire à chacune. On peut également imaginer un contrôle intelligent de l'éclairage dans une maison. Les lumières peuvent être allumées automatiquement lorsqu'une personne est détectée dans une pièce et que la luminosité ambiante est trop faible. Lorsque cette personne quitte la pièce, les lumières s'éteignent à nouveau. En suivant de pièce en pièce le déplacement de cette personne, on optimise ainsi l'éclairage ambiant. D'autres paramètres peuvent entrer en compte, comme la nature de la pièce ou les activités de la personne dans cette pièce (lecture, télévision, ou dîner aux chandelles). Des exemples plus classiques d'applications concernent les systèmes de détection d'incendie, la gestion d'énergie, ou la gestion d'un hôpital [Shu06].

2.1.1.2 Réalisation

Les nombreux progrès de l'informatique distribuée et de ses domaines connexes, ainsi que l'apparition de nouvelles générations d'entités font de l'informatique ubiquitaire une réalité. L'expansion impressionnante des réseaux de tous types permet désormais à ces entités de facilement communiquer entre elles (on les appelle aussi des *entités communicantes*). De plus, ces entités fournissent des interfaces de haut niveau, facilitant l'accès à leurs fonctionnalités et aux données qu'elles capturent dans l'environnement.

Ces données représentent un *contexte* particulier de l'environnement ; elles sont de natures très différentes : par exemple, un capteur de température recueille des données numériques ; un logiciel de gestion d'adresses contient une base de données d'informations sur les contacts d'une personne ; un détecteur de mouvement récolte des informations de localisation. Les entités réagissent à ces informations en faisant appel à leurs fonctionnalités, qui à leur tour peuvent modifier le contexte de l'environnement. Par exemple, un climatiseur modifie la température ambiante, un robinet d'arrosage automatique modifie l'humidité recueillie par un capteur d'humidité. Programmer des applications pour l'informatique ubiquitaire consiste à *orchestrer* ces entités, c'est-à-dire à définir la manière dont ces entités doivent interagir en fonction du contexte.

Défi

L'informatique ubiquitaire s'est développée dans de nombreux champs d'applications. Chaque champ d'applications implique des *utilisateurs*, auxquels ces applications doivent pouvoir être facilement adaptées pour satisfaire leurs besoins et préférences en confort, sécurité, ou encore assistance quotidienne. Ce défi prend toute sa dimension pour des utilisateurs mentalement ou physiquement déficients à qui l'on souhaite permettre de vivre à domicile de manière autonome [Sta02]. En effet, pour ces personnes, les applications doivent constamment être adaptées, au fur et à mesure qu'elles gagnent ou perdent en autonomie. Adapter ces applications nécessite de bien identifier le champ d'applications que l'on veut cibler et le type d'utilisateurs concernés par les applications. L'étude des utilisateurs permet de déterminer la manière d'adapter les

applications. L'identification d'un champ d'applications permet de collecter les moyens nécessaires (les entités et les connaissances des experts du domaine) pour développer des applications pour les utilisateurs. Nous examinons à présent trois champs offrant un aperçu des besoins des utilisateurs pour les applications ubiquitaires.

2.1.2 Des champs d'applications

Les applications peuvent être classées selon les services qu'elles rendent aux utilisateurs. Les champs d'applications que nous couvrons dans cette thèse sont des combinaisons de tout ou partie de trois champs particuliers. Chacun de ceux-ci exhibe une variété spécifique de services : l'automatisation d'appareils dans une maison, la personnalisation de l'information numérique dans les environnements publics, et enfin l'apport de support technologique aux activités des personnes.

Domotique La domotique désigne l'ensemble des technologies et techniques mises en œuvre pour apporter des fonctions de gestion de la maison, telles que l'automatisation du fonctionnement d'appareils pour délester ses habitants de certaines tâches. Des applications ordinaires de la domotique sont les systèmes de gestion d'énergie, servant à automatiser les appareils de climatisation et d'éclairage pour le confort de l'habitant. Ces applications doivent pouvoir être paramétrées selon les préférences de l'habitant et l'état courant de l'environnement (par exemple, la température, la luminosité ambiante, ou l'heure de la journée). La domotique est généralisable à des espaces physiques plus larges, que sont les écoles, les musées, ou les bâtiments d'entreprise. Elle porte alors parfois la dénomination d'*immotique*.

Gestion de l'information L'information numérique est désormais disponible partout : dans les espaces publics, dans les lieux professionnels, dans les entités mobiles. Des exemples d'information numérique sont la publicité, la météorologie, les actualités politiques. Ces informations sont constamment mises à jour et doivent être personnalisées selon les personnes à qui elles sont destinées. Un objectif des applications de gestion d'information est de recueillir et traiter des sources d'informations aussi variées que versatiles, telles que des agendas personnels ou d'entreprise, des bulletins d'information disponibles sur Internet (flux RSS et Twitter). Des applications typiques de ce champ sont les gestionnaires de conférence ou de bulletins d'information dans les entreprises et l'affichage d'emplois du temps des étudiants dans les écoles.

Assistance à la personne L'assistance à la personne est l'ensemble des aides apportées à une personne pour lui permettre de vivre de manière digne et autonome à domicile. Les applications d'aide à une personne ont pour objectif de lui fournir une assistance technologique compensant ses déficiences. En particulier, par un suivi adapté des actions d'une personne déficiente, ces applications pourront l'aider à coordonner ses activités quotidiennes, par exemple pour compenser ses problèmes de mémorisation. Pour cela ces applications doivent mettre à sa disposition des technologies d'assistance agissant de manière appropriée pour la guider dans l'accomplissement de ses activités. L'avantage de telles applications est de minimiser l'intervention de tiers pour assister les personnes déficientes et d'augmenter leur indépendance.

2.2 L'utilisateur et les applications

L'utilisateur est au centre des applications que nous venons de décrire. Nous définissons ici la notion d'utilisateur dans le contexte de ces applications. Cette définition constitue le point de départ pour établir la relation entre les utilisateurs d'un environnement ubiquitaire et les différentes manières d'adapter les applications pour répondre à leurs besoins. Cette relation est ensuite précisée, en attribuant aux utilisateurs la fonction d'acteurs potentiels dans le développement des applications.

2.2.1 Une taxonomie d'utilisateurs

Notre analyse des différents champs d'applications nous a permis d'identifier quatre catégories d'utilisateurs en fonction de leurs attentes et des interactions qu'ils entretiennent avec une application ubiquitaire. Ces catégories, que nous présentons ici, constituent une taxonomie. Elle sert de support à la section suivante qui définit la notion d'adaptation d'applications.

L'utilisateur générique L'utilisateur générique est une personne interagissant avec des applications qui ne sont pas destinées à être personnalisées pour l'individu. Des applications de ce type sont celles destinées à des groupes d'utilisateurs ou à des utilisateurs interagissant avec elles de manière temporaire. On peut inclure dans cette catégorie les étudiants dans un établissement scolaire, les patients dans un hôpital, ou les visiteurs d'un musée. Ces utilisateurs peuvent tout au plus décider d'interagir ou pas avec les applications qui leur sont proposées.

Le prescripteur Le prescripteur est un utilisateur qui a des attentes précises sur les applications : c'est un client qui fait des commandes d'applications, mais qui n'a pas forcément les connaissances ou les moyens nécessaires pour définir précisément la manière dont il souhaite voir ses besoins réalisés. Un exemple de prescripteur est le directeur de musée qui souhaite mettre en place des applications de guide intelligent de visite de musée pour optimiser le flux de visiteurs. Un autre exemple est représenté par les personnes qui nécessitent une assistance dans leurs activités quotidiennes : les applications d'assistance doivent être parfaitement adaptées à leurs attentes pour leur permettre d'acquérir de l'autonomie.

L'utilisateur de confort L'utilisateur de confort a des besoins sommaires d'adaptation d'applications. Ces besoins ne sont pas vitaux et concernent par exemple son confort quotidien. On peut inclure dans cette catégorie l'habitant d'une maison : celui-ci peut souhaiter que les appareils installés dans les différentes pièces de sa maison soient configurés selon ses préférences. Un exemple de besoin en configuration est le réglage de la température moyenne pour une application de climatisation, ou la configuration du son et de l'heure d'enregistrement d'émissions pour sa télévision selon l'heure de la journée et la pièce dans laquelle il se trouve.

L'expert d'un métier L'expert d'un métier ou *expert-métier* est un utilisateur "par procuration", c'est-à-dire qu'il n'utilise pas directement les applications, mais il possède les connaissances nécessaires pour déterminer les moyens requis pour réaliser les besoins des utilisateurs

précédents. En cela il est capable d'appréhender la manière dont les applications peuvent servir les utilisateurs. Par exemple, l'administrateur de sécurité d'un bâtiment connaît les paramètres et les éléments matériels nécessaires à la mise en place d'un système de détection d'intrusion pour la sécurité de ce bâtiment. Un autre exemple, que nous utilisons par la suite pour poser les fondations de cette thèse, est l'éducateur spécialisé. Son métier est de définir et de mettre en place des solutions d'assistance pour faciliter le quotidien des personnes déficientes, et les applications constituent potentiellement un outil fondamental pour l'aider dans cette tâche.

2.2.2 Adapter les applications aux besoins

Les quatre catégories d'utilisateurs que nous venons de décrire nécessitent que les applications ubiquitaires soient adaptables à différents niveaux. Nous examinons à présent ces niveaux d'adaptation. Chaque niveau raffine le précédent, au sens où l'adaption doit être de plus en plus précise.

Paramétrer des applications prédéfinies Les applications prédéfinies sont des boîtes noires prêtes à être installées, qui exposent uniquement certains paramètres. Ces applications donnent un degré limité de liberté pour l'adaptation aux besoins, qui se limite à donner des valeurs que l'application utilise pour convenir à une configuration souhaitée par l'utilisateur. Par exemple, les applications de contrôle de la climatisation peuvent être préinstallées et offrir une interface de paramétrage, afin de régler la température moyenne préférée de l'habitant de la maison. Un autre type de paramétrage peut être de simplement contrôler la mise en route ou l'interruption de tout ou partie de l'application. Les utilisateurs ciblés par ce niveau d'adaptation sont essentiellement les utilisateurs génériques ou les utilisateurs de confort.

Composer des applications On peut augmenter le degré de liberté permis par les applications précédentes avec une approche basée sur les *composants sur étagère* (COTS ou *commercial off-the-shelf*). Chacun de ces composants, par définition prêts-à-l'emploi et disponibles sur le marché, offrent des fonctionnalités destinées à répondre à un besoin spécifique. Il est possible de les composer pour définir une application complète. Le logiciel de composition de services OSCAR [NES08] met en œuvre ce type d'adaptation en proposant de connecter des services prédéfinis pour la gestion des médias dans une maison. Chaque service est une application servant à contrôler un médium particulier (une télévision, un écran, une caméra) via un ensemble fixé de fonctionnalités paramétrables. Une application complète est un ensemble de connexions entre ces services et permet de faire interagir ces différents médias de manière cohérente. Les utilisateurs ciblés par ce niveau d'adaptation sont les utilisateurs de confort, et dans une certaine mesure les experts-métier.

Programmer des applications ad hoc On peut décider de développer des applications sur mesure, en partant de zéro, afin d'obtenir une solution parfaitement adaptée aux besoins. Par exemple, les personnes déficientes ont des besoins très précis en matière d'assistance. Pour cela, paramétrer des applications prédéfinies ou composer des applications ne suffit plus, car leur

degré de configuration est rapidement limité. En particulier, dans une approche COTS, l'extension des composants pour intégrer des fonctionnalités non prévues peut être fastidieux [McK99]. *Programmer* devient alors nécessaire pour rendre possible l'ajustement des applications aux besoins de chaque individu. Ce besoin apparaît dans d'autres champs d'applications tels que les systèmes de détection d'intrusion dans les lieux professionnels, où chaque espace professionnel peut avoir des spécificités et des contraintes propres selon les règles de sécurité imposées par le type d'espace (espaces militaires, laboratoires contenant du matériel coûteux et/ou dangereux). Les utilisateurs nécessitant ce niveau de précision sont les experts-métier, et éventuellement les prescripteurs.

2.2.3 Des utilisateurs aux applications

Puisque l'utilisateur est au centre des applications, et qu'il est celui qui a la meilleure connaissance de ses besoins, la situation idéale serait que ce soit lui-même qui adapte ces applications selon ses besoins et ses préférences. Fischer argumente cette proposition dans [FNY09] par le constat de la disparition progressive entre le rôle d'utilisateur et celui de développeur, dans de nombreux domaines du développement logiciel incluant la programmation Web. A la différence de Fischer, qui définit une échelle entre utilisateurs et experts en développement logiciel en fonction des outils de développement, nous proposons de classer les différents utilisateurs définis plus haut en fonction des degrés d'adaptation des applications que nous avons identifiés. Ce classement est illustré par la figure 2.1.

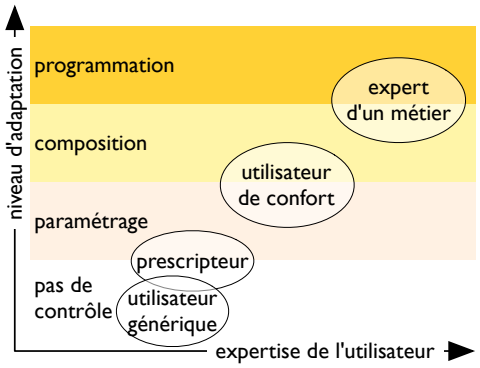

FIG. 2.1: Relation entre l'utilisateur et l'adaptation des applications

L'utilisateur générique n'a aucun contrôle sur l'application, ou au mieux, une possibilité de paramétrage réduite à l'interruption ou non d'une application. Le prescripteur a des besoins précis et fait des commandes d'applications, mais n'est pas en mesure de développer les applications lui-même, ce qui le place, en tant qu'acteur dans l'adaptation des applications, au même niveau que l'utilisateur générique. L'utilisateur de confort peut paramétrer lui-même des applications prédéfinies pour son environnement, c'est-à-dire fournir des valeurs que l'application utilise ensuite pour s'adapter à ses préférences. Il peut au mieux composer des applications prédéfinies disponibles sur le marché. Enfin, l'expert-métier doit pouvoir disposer d'outils qui lui permettent de programmer des applications ad hoc en utilisant ses connaissances du métier.

2.2.4 Illustration pour l'assistance à la personne

Pour illustrer la pertinence de cette taxonomie d'utilisateurs, nous étudions l'exemple de l'assistance à la personne.

Définition et enjeux L'assistance à la personne a pour objectif d'apporter un soutien aux personnes déficientes physiquement et/ou mentalement dans leur vie quotidienne. Selon l'article L114 du code de l'action sociale et des familles, le handicap constitue "*toute limitation d'activité ou restriction de participation à la vie en société subie dans son environnement par une personne en raison d'une altération substantielle, durable ou définitive d'une ou plusieurs fonctions physiques, sensorielles, mentales, cognitives ou psychiques, d'un polyhandicap ou d'un trouble de santé invalidant*". A cause de leur déficience, les personnes handicapées ont des difficultés à mener des activités basiques de leur quotidien, comme l'habillement, la toilette, ou la cuisine : ces personnes ont souvent en commun la difficulté à structurer leurs activités, requérant la mise en place de *stratégies éducatives* afin de les assister dans l'organisation de leurs activités quotidiennes. Une stratégie éducative consiste à définir un objectif spécifique et d'amener progressivement la personne à le réaliser de manière autonome. Ces objectifs peuvent être l'utilisation de transports en commun, la cuisine, ou la pratique de loisirs. Du fait de la difficulté de ces personnes à séquencer les tâches permettant d'atteindre ces objectifs, elles ont besoin d'être accompagnées par des éducateurs spécialisés. Les éducateurs spécialisés jouent un rôle de premier plan dans l'élaboration des solutions d'assistance à ces personnes.

En particulier, ils se chargent de mettre en place tous les éléments ou supports nécessaires pour guider au quotidien les personnes déficientes dans leur environnement. Ces éléments sont humains (présence d'aides médico-psychologiques auprès de la personne) ou matériels (utilisation d'images servant d'aide-mémoire, d'interfaces adaptées aux objets du quotidien – horloges, téléphones, télécommandes d'appareils électro-ménagers). Dans ce domaine, il est nécessaire d'adapter en permanence les applications lorsque les personnes déficientes évoluent. Par exemple, lorsqu'une personne progresse et acquiert des mécanismes cognitifs lui permettant de réaliser des tâches seule, les applications peuvent être assouplies, c'est-à-dire réduire leur intervention dans la vie de la personne. Dans d'autres cas, les applications doivent faire face au déclin intellectuel et physique de personnes, comme c'est le cas de personnes atteintes de maladies dégénératives. L'assistance doit alors être renforcée : les applications doivent être étendues pour répondre aux nouveaux besoins qui apparaissent.

Dans les paragraphes qui suivent, nous proposons de placer les personnes impliquées dans la réalisation d'applications d'assistance dans chacune des catégories de notre taxonomie d'utilisateurs.

Un prescripteur : la personne déficiente La personne déficiente est la personne à qui les applications d'assistance sont destinées in fine. Ces applications doivent parfaitement s'accorder à chaque individu. La personne déficiente a un pouvoir de décision fort sur les applications et sur leur niveau d'intervention dans sa vie quotidienne, mais n'a pas les moyens de mettre en œuvre ces décisions.

Un utilisateur de confort : l'aide médico-psychologique L'aide médico-psychologique assiste les éducateurs spécialisés dans le suivi quotidien d'une personne déficiente. Elle a des connaissances éducatives, sociales et psychologiques lui permettant de mettre en œuvre les stratégies éducatives définies par l'éducateur spécialisé ou de l'assister dans cette tâche. L'aide médico-psychologique peut intervenir ponctuellement pour paramétrer les applications, selon ses observations quotidiennes des réactions de la personne déficiente vis-à-vis du comportement de l'application.

Un expert du métier : l'éducateur spécialisé L'éducateur spécialisé a pour métier de construire des projets d'aide individualisée. Ces projets regroupent plusieurs types de stratégies éducatives (liées par exemple aux aspects cognitifs, aux relations sociales, ou à l'autonomie dans la vie quotidienne) destinées aux personnes déficientes. L'éducateur spécialisé a en outre pour rôle d'évaluer régulièrement la pertinence du projet, et d'adapter les stratégies éducatives en fonction de la manière dont la personne déficiente y réagit. L'éducateur spécialisé, en tant qu'expert du métier, a un intérêt particulier à programmer des applications d'assistance à la personne. Ces applications peuvent en effet remplacer ou augmenter les supports matériels qu'il met en place actuellement afin d'apporter une aide plus efficace à la personne déficiente. De plus, elles permettraient de réduire son intervention dans le quotidien des personnes.

Positionnement

Dans cette thèse, nous nous intéressons à l'expert-métier et souhaitons l'amener vers la programmation d'applications pour son domaine. Or, cet utilisateur n'est pas programmeur de formation : il faut donc lui fournir les outils appropriés pour programmer des applications fiables réalisant ses besoins.

Faciliter la programmation est un véritable problème dans le domaine de l'informatique ubiquitaire, notamment lorsqu'on cible des non-programmeurs. Nous en examinons les raisons ci-après.

2.3 Problématique : faciliter la programmation

Si les technologies de l'informatique ubiquitaire permettent d'envisager un large périmètre d'applications, l'expertise qu'elles requièrent rend difficile l'activité de programmation d'applications par des utilisateurs non-programmeurs. La question principale qui se pose est donc la suivante : comment tirer parti de la richesse des technologies pour, d'une part répondre aux besoins des utilisateurs en matière d'applications, et d'autre part, rendre accessible la tâche de programmation d'applications à des non-programmeurs ? Pour répondre à cette double question, on a besoin d'une approche de programmation qui soit à la fois expressive et accessible. Nous appelons *expressivité* la capacité d'une approche à décrire, à l'aide de notations appropriées, des applications pour la grande variété des besoins exprimables dans divers champs d'applications. L'*accessibilité* concerne la facilité d'utilisation des outils de programmation par des non-programmeurs. Nous détaillons ici les obstacles à ces deux exigences faisant de la facilité

de programmation un problème délicat : ce sont d'une part l'expertise requise pour manipuler les technologies, et d'autre part la richesse des champs d'applications.

2.3.1 Obstacle à l'accessibilité : expertise

Programmer des applications pour l'informatique ubiquitaire suppose une expertise dans nombre de domaines connexes, comme : l'informatique distribuée (la programmation concurrente et la gestion d'applications distantes), les réseaux (les protocoles de communication et l'interopérabilité entre ces protocoles), la sécurité (la restriction d'accès à certaines données et la gestion de la vie privée). Ces prérequis rendent la programmation d'applications difficile pour un non-programmeur, et augmentent la distance entre les besoins exprimés et leur réalisation par les applications. L'approche naturelle pour réduire cette distance est de masquer les complexités des technologies exigeant une expertise dans les domaines pré-cités, c'est-à-dire d'abstraire ces préoccupations des objectifs strictement liés à la programmation des applications.

2.3.2 Problème d'expressivité : richesse des champs d'applications

Nous avons vu que les besoins en applications existent dans de nombreux champs, et présupposent au moins autant de métiers. Pour rendre possible l'adaptation des applications par un expert-métier, il faut pouvoir capturer le vocabulaire pertinent pour le champ d'applications qui intéresse l'expert. Les éléments clés permettant de capturer ce vocabulaire sont les entités communicantes. Or, bien que leurs fonctionnalités soient de haut niveau, il n'est pas garanti que les moyens d'y accéder exhibent les concepts appropriés pour exprimer des applications dans un champ donné et pour un métier spécifique. Il y a par ailleurs un lien de cause à effet entre la capacité à aborder un grand nombre de champs d'applications et celle à satisfaire un large panel de besoins : atteindre le premier objectif ouvre la possibilité de prendre en compte l'évolution et l'apparition de nouveaux besoins.

2.3.3 Limites de l'existant

Deux types d'approches existent pour s'affranchir des obstacles que nous venons de citer. Les premières sont celles basées sur des intergiciels et les deuxièmes sont les outils de programmation pour l'utilisateur. Cependant, ces approches parviennent difficilement à associer expressivité et accessibilité, comme nous le voyons ci-après.

2.3.3.1 Approches reposant sur des intergiciels

Les intergiciels sont des couches logicielles masquant les complexités de l'informatique distribuée et de ses domaines connexes. Elles offrent des bibliothèques et des interfaces de programmation pour programmer et connecter facilement des applications entre elles. Gaia [RHC+02], One.World [Gri04] et Aura [GSS02] sont des intergiciels dédiés à la programmation d'applications de l'informatique ubiquitaire pour un grand nombre de champs d'applications. Cependant, ces approches reposent sur un langage de programmation généraliste (C++, Java, ...). En conséquence, les solutions exprimées sont difficilement lisibles pour des non-programmeurs ; elles ne

réduisent pas la distance séparant les besoins de leur réalisation. De ce fait, elles ne sont pas accessibles à des utilisateurs non-programmeurs. De plus, ces approches restent généralistes, faisant de la programmation d'applications une activité coûteuse et génératrice d'erreurs. Il est donc d'autant plus difficile de faire évoluer ces applications pour satisfaire aux besoins et préférences des utilisateurs.

2.3.3.2 Outils de programmation pour l'utilisateur

Pour offrir aux utilisateurs les moyens de programmer eux-mêmes les applications dont ils ont besoin, il existe des outils destinés à des non-programmeurs. Grâce à ces outils, les utilisateurs développent leurs propres applications, consistant à orchestrer les entités installées dans leur maison. Parmi ces approches, citons CAMP [THA04], iCAP [DSSK06] et Topiary [LHL04]. Malheureusement, cette accessibilité est souvent faite au détriment de l'expressivité : ces approches se restreignent volontairement à un champ d'applications particulier pour augmenter leur accessibilité et proposent à l'utilisateur un vocabulaire limité. En limitant ainsi leur expressivité, ces approches ne peuvent s'adapter que dans une certaine mesure à l'évolution des besoins de l'utilisateur. De plus, leur restriction à un champ d'applications constitue un frein à l'adaptation de ces applications à l'évolution continuelle des technologies pour l'informatique ubiquitaire.

2.3.4 Proposition : méthodologie outillée

Nous venons de voir que l'utilisateur, et plus spécialement l'expert-métier, a une place centrale dans le développement des applications. Pour qu'il puisse programmer des applications, nous avons vu qu'il faut lui proposer des outils accessibles et expressifs. Mais cela n'est pas suffisant : il faut en plus garantir qu'une application définie par l'utilisateur répond précisément aux besoins initiaux qu'il a exprimés. Pour le guider dans cette activité, nous proposons une *méthodologie outillée*. Cette méthodologie crée une passerelle entre l'expression initiale des besoins et les applications qui les réaliseront. Son étape d'analyse des besoins permet d'identifier les briques de base nécessaires à la construction d'une application. Nous proposons d'associer à cette méthodologie un langage visuel pour modéliser, puis manipuler ces briques de base afin de programmer une application pouvant être testée.

2.4 Apports de cette thèse

Cette thèse propose une méthodologie couplée à un langage accessible et permettant de guider des utilisateurs non-programmeurs dans la construction d'applications ubiquitaires. Nous présentons ici les apports de cette thèse, qui ont permis de faire un premier pas vers cet objectif.

2.4.1 Vers une méthodologie pour guider la programmation

Nous souhaitons amener un expert-métier à exprimer des applications répondant précisément à ses besoins. Pour cela, nous avons défini une méthodologie, fondée sur une étude menée auprès d'éducateurs spécialisés, pour guider la définition d'applications en partant d'une analyse des

besoins adaptée au domaine de l'orchestration d'entités. Cette méthodologie est outillée et combine une démarche d'analyse (*top-down*) consistant à décomposer les besoins pour exhiber des entités, avec une approche de synthèse (*bottom-up*) consistant à orchestrer les entités pour mettre en oeuvre les applications réalisant ces besoins. En procédant ainsi, cette méthodologie a pour ambition d'offrir un support facilitant la création et l'évolution des applications selon les besoins, en établissant une relation entre l'expression informelle des besoins et leur réalisation. Nous avons évalué la pertinence de cette méthodologie auprès d'experts-métier de l'assistance à la personne.

2.4.2 Un langage paramétré par un domaine

Nous souhaitons outiller chaque étape de la méthodologie afin d'accompagner la définition et la mise en œuvre d'une application, depuis les besoins jusqu'à la phase de test et de déploiement. Pour cela, nous proposons un langage de programmation visuel, *Pantagruel*. Ce langage permet de modéliser un champ d'applications par une taxonomie d'entités mettant à disposition leurs fonctionnalités et les données qu'elles capturent sur l'environnement. Il permet ensuite d'orchestrer ces entités, en utilisant des paradigmes adaptés à un utilisateur non-programmeur. Nous avons réalisé une évaluation qualitative de son expressivité sur les champs d'applications présentés dans ce chapitre. Nous avons également évalué son accessibilité auprès de vingt utilisateurs non-programmeurs. Enfin, nous avons formalisé ce langage en définissant sa sémantique dénotationnelle.

2.4.3 Une approche guidée par la vérification

Faciliter la programmation des applications est une chose. Pour qu'un utilisateur, et a fortiori l'expert-métier, accepte d'adopter un outil informatique, il faut qu'il lui accorde sa confiance. Les outils doivent donc garantir au développeur d'applications que non seulement celles-ci expriment sans ambiguité ses intentions, mais qu'elles soient également fiables. La fiabilité des applications est critique pour les experts-métier, notamment lorsqu'elles mettent en jeu la sécurité (au sens général) des utilisateurs de ces applications. Pour cela, nous proposons une méthode de développement dirigée par l'expression de propriétés sur les comportements qu'un programme doit garantir. L'originalité de cette approche est que ces propriétés peuvent s'écrire dans le même langage que celui des applications d'orchestration. Nous mettons par ailleurs en évidence le fait que la vérification de ces propriétés est facilitée par la sémantique formelle du langage.

3 Faciliter la programmation

Dans le chapitre précédent, nous avons motivé l'élaboration d'une méthodologie outillée pour guider les utilisateurs (en particulier l'expert-métier) dans l'activité de programmation d'applications d'orchestration. L'expert-métier n'étant pas un programmeur par nature, il est nécessaire que les outils de programmation soient accessibles. Nous présentons ici deux procédés complémentaires destinés à faciliter la programmation : les langages dédiés et les supports de programmation visuels.

Pour faciliter la programmation à des experts-métiers d'un domaine particulier, la première étape consiste à leur proposer un langage dont les constructions sont spécialement conçues pour faciliter la réalisation de programmes pour le dit domaine.

C'est ce qu'apportent les *langages dédiés*. L'étape suivante est d'apporter un *support de programmation visuel* pour assister l'utilisateur dans l'activité de programmation. Dans cette section, nous faisons une présentation générale de ces deux étapes, après avoir précisé ce qu'on appelle l'utilisateur d'un langage.

3.1 Utilisateurs de langages

Il y a deux types d'utilisateurs de langages : les programmeurs professionnels, qui savent programmer, particulièrement ceux dont le métier inclut partiellement ou totalement l'activité de programmation, et les non-programmeurs. Sont définis comme des utilisateurs non-programmeurs les utilisateurs finaux et les programmeurs novices. A. J. Ko *et al.* ont proposé la définition suivante d'un utilisateur final [KAB+10] : "[*il*] *écrit des programmes pour accomplir un but dans le cadre de son domaine d'expertise*". Par défaut, un expert-métier appartient à cette catégorie d'utilisateurs. Les programmeurs novices sont à mi-chemin entre les utilisateurs finaux et les programmeurs : ce sont des gens qui n'ont pas de compétence en programmation, mais dont l'objectif est d'apprendre à programmer [KP05]. Cependant, de même que pour les utilisateurs finaux, ils n'ont pas un mode de pensée "programmatique". Il est par conséquent nécessaire de guider ces deux utilisateurs pour les amener à construire des applications fiables réalisant leurs besoins. Sauf indication contraire, nous amalgamons ces deux types d'utilisateurs, et parlons d'utilisateurs finaux dans la suite de ce document.

3.2 Les langages dédiés

Les langages généralistes, comme Java ou C, sont des langages très expressifs au sens où ils permettent de développer des applications pour n'importe quel domaine, mais ils ne sont pas adaptés à des utilisateurs non-programmeurs. Par opposition, les langages dédiés, que nous présentons maintenant, visent à améliorer l'accessibilité de la programmation.

3.2.1 Motivations

Les langages dédiés (ou *DSL*, pour *Domain-Specific Language*) ont pour objectif de proposer des abstractions et des notations spécifiques pour décrire une solution de manière concise dans un domaine particulier. Plus précisément, leur syntaxe capture le vocabulaire propre à ce domaine. Un langage dédié cible l'expert du domaine considéré, à qui ces notations et ces abstractions sont familières.

Ces langages sont utilisables par des utilisateurs qui ne sont pas forcément des programmeurs. Par exemple, le langage dédié SQL [Mel96] s'adresse à des utilisateurs devant manipuler des bases de données ; le langage DOT de la suite d'outils GraphViz [GN00] sert à définir des graphes et des propriétés de visualisation de ces graphes. De nombreux travaux présentent des langages dédiés dans des domaines variés ; la bibliographie annotée [vDKV00] présente un panorama d'applications de ces langages.

3.2.2 Apports des langages dédiés

Les langages dédiés facilitent la programmation, la maintenance, la réutilisation et la vérification du code. Nous présentons rapidement deux apports de ces langages, dont nous tirons profit pour le langage proposé dans cette thèse.

Facilité de programmation Les langages dédiés sont des langages conçus sur mesure pour faciliter le développement d'applications dans un domaine spécifique, en augmentant la lisibilité et la concision des programmes. Souvent, les langages dédiés sont déclaratifs, et permettent d'exprimer un programme dans les termes du problème qu'il résout (on dit qu'il exprime le "quoi") plutôt que dans les termes de la solution (on dit qu'il exprime le "comment"). Les langages dédiés permettent ainsi de s'abstraire des détails d'implémentation, et de se concentrer uniquement sur les problèmes du domaine. Enfin, leurs constructions syntaxiques restreintes réduisent le risque d'erreur dans le code, ce qui est d'autant plus important lorsque les développeurs sont des utilisateurs finaux.

Prenons l'exemple du langage Pantaxou [MPCL08]. Ce langage est dédié à la définition et à la coordination de services distribués. Pour cela, il fournit une syntaxe pour décrire et utiliser différents modes de communication (événements, commandes et sessions) offerts par les services. En particulier, Pantaxou offre des constructions typées pour déclarer les ressources que ces services doivent fournir ou peuvent recevoir (`requires event<`*`eventType`*`> from` *`serviceType`*, `provides command<`*`commandType`*`> to` *`serviceType`*, ...).

Fiabilité Un argument majeur pour la création de langages dédiés est qu'ils facilitent la vérification de propriétés sur les programmes. Par nature, les langages dédiés définissent une sémantique restreinte, permettant d'effectuer des analyses de code propres à leur domaine. En particulier, ils rendent certaines propriétés décidables (comme la terminaison d'un programme), contrairement aux programmes écrits dans un langage généraliste. Ces propriétés augmentent la fiabilité des programmes, qui est une condition importante de l'adoption des outils informatiques par des expert-métiers.

Par exemple, le langage Pantaxou permet de garantir des propriétés liées à l'interaction entre les entités, comme le fait qu'un événement publié par un service n'est jamais perdu. Ce genre de propriété peut difficilement être garanti avec des approches généralistes de la programmation distribuée.

3.2.3 Construction d'un langage dédié

Créer un langage dédié est une tâche difficile. Pour faciliter cette tâche, des méthodologies de développement ont été proposées [MHS05, TMC99, CM98]. L'une de ces méthodologies consiste en une classification de motifs (*patterns*) suivant les phases de développement du langage [MHS05].

Ces phases de développement sont la décision, l'analyse, la conception, l'implémentation. La phase de décision permet d'établir s'il est pertinent ou non de développer un langage dédié. La phase d'analyse est dédiée à l'identification du domaine, afin d'extraire les concepts clés capturant les connaissances du domaine. En phase de conception, le langage est conçu, soit en définissant un langage *embarqué* dans un langage généraliste – réutilisant sa syntaxe et sa sémantique –, soit en définissant un langage autonome. Une des approches pour définir un langage dédié consiste à spécifier le langage formellement (par exemple dans une sémantique dénotationnelle [Ten76]) au moyen d'une méthodologie, comme la méthodologie de développement de langages proposée par D. Schmidt [Sch86], et que nous utilisons pour formaliser notre langage. Une définition formelle est indispensable pour faciliter l'analyse de programmes. Enfin la phase d'implémentation donne lieu à un interprète ou à un compilateur vers un langage hôte existant. La compilation permet de déléguer l'implémentation de mécanismes complexes au langage cible, comme la gestion de la mémoire ou des mécanismes de communication dans le cas des langages de programmation distribuée.

Ces méthodologies souffrent néanmoins d'un défaut important : à notre connaissance, aucune de celles évoquées dans les travaux sur les DSLs n'aborde la question de l'évaluation de l'accessibilité d'un langage pour les utilisateurs qu'il cible. Or cette étape devrait presque être incontournable lorsque les langages sont destinés à des utilisateurs non-programmeurs. Cette évaluation, qualitative par nature, est d'ailleurs plutôt l'apanage de la communauté de recherche des interactions homme-machine (*Human-Computer Interaction*) [GP96, **?**, GB08], dont les langages visuels pour l'utilisateur final est l'une des thématiques. Dans la section 8.3, nous empruntons une technique classique de cette communauté pour évaluer notre langage.

3.3 Supports visuels pour la programmation

Nous présentons à présent les supports visuels qui sont un point de vue complémentaire aux langages dédiés pour l'aide à la programmation. Ces supports permettent d'utiliser des éléments graphiques lors du processus de programmation, auquel cas on emploie aussi l'expression de *programmation visuelle* [Shu99, BM95].

3.3.1 Motivations

Les supports visuels de programmation permettent de faciliter la programmation en apportant une aide visuelle au développeur lors du processus de programmation [RSB05]. Cette aide se matérialise de plusieurs manières : par des outils de visualisation de programmes, par des environnements de programmation visuels, ou par des langages visuels [Mye86]. Nous détaillons les deux dernières approches dans les sections suivantes.

Le but des supports visuels est d'offrir à l'utilisateur une relation interactive avec l'outil de programmation, facilitant l'exploration et l'apprentissage des éléments de programmation du langage. De plus, les représentations visuelles facilitent la compréhension d'un programme à des utilisateurs finaux [Shu99, BM95]. Prenons l'exemple des langages visuels : il est intéressant de noter qu'une cible favorite des langage visuels sont les enfants [RMMH$^+$09, BL93, GIL$^+$95, SCS94]. Un exemple typique de tels langages est Scratch [RMMH$^+$09], créé au MIT. Ce langage est désormais intégré au programme des lycées pour l'apprentissage de l'algorithmique [Nat]. Il permet de définir, à l'aide d'expressions visuelles représentées par des pièces de puzzle, des animations graphiques et des jeux. De nombreux enfants ont publié, sur le site officiel de Scratch, des programmes écrits à l'aide de ce langage, démontrant son accessibilité.

3.3.2 Environnements de programmation visuels

Les environnements de programmation sont des outils répandus facilitant l'écriture de code dans des langages traditionnels. Leurs fonctionnalités prennent la forme d'outils de complétion contextuelle automatique (mots clés, rappels de fonctions et de structures définies dans le code, ou encore motifs syntaxiques), de vérification de syntaxe et de typage à la volée, d'outils de débogage ou de test. La grande majorité des éditeurs textuels de code proposent désormais de telles fonctionnalités, y compris *emacs* et *vim* (pour *vi*-i*m*proved, l'éditeur Unix vi amélioré), deux des éditeurs les plus populaires au sein de la communauté des programmeurs.

Un environnement utilisant des expressions visuelles est appelé un environnement de programmation visuelle (*VPE* pour *Visual Programming Environment*) [Shu99] – on les appelle également des *éditeurs visuels*. Par exemple, l'environnement de développement intégré Eclipse propose un environnement de programmation visuel pour Java, et intègre de nombreuses fonctionnalités graphiques, dont différentes vues éditables d'un même programme facilitant l'accès aux informations relatives au code (structure, dépendances entre les fichiers, ou entre les éléments du programme, dont les méthodes, les appels de méthodes, les classes). D'autres environnements existent pour assister la programmation dans des langages textuels, comme ViPEr [SMA02] pour le langage généraliste Python ou Visifold [BA96] pour le langage de coordination Manifold [**?**].

3.3.3 Langages de programmation visuels

Dans un langage visuel, l'utilisateur exprime des programmes en utilisant des éléments syntaxiques combinant plusieurs dimensions, dont une dimension spatiale, et la plupart du temps une dimension textuelle. Les langages visuels partagent des objectifs proches des langages dédiés : ils fournissent une notation appropriée pour représenter un domaine particulier, et faci-

litent l'écriture de programmes corrects. La bibliographie [Mar] recense un très grand nombre de langages de programmation visuels, classifiés selon les paradigmes couramment utilisés par ces outils [BB94]. Beaucoup de ces langages visuels s'adressent à des utilisateurs qui ne sont pas experts en programmation, c'est-à-dire des utilisateurs finaux ou des programmeurs novices [KP05].

La frontière entre les environnements de programmation visuels et les langages de programmation visuels est poreuse. Un exemple illustratif de cette observation est l'environnement Squeak de Smalltalk [IKM+97], où le langage est parfaitement intégré avec les outils de l'environnement et l'environnement d'exécution. M. Burnett différencie ces deux approches ainsi : les premiers sont dédiés à des langages traditionnels ; les seconds sont des langages visuels à part entière qui sont complètement intégrés à leur environnement de programmation [BM95]. Nous conservons cette distinction.

Catégories de langages visuels Dans un des articles fondateurs sur la programmation visuelle, N.C. Shu [Shu99] classe les langages visuels en trois grandes catégories : les langages icôniques, les langages diagrammatiques et les langages orientés formulaires.

Les langages diagrammatiques définissent des connexions entre des éléments visuels représentant des objets, des processus, ou les différents états d'un programme [Har87]. Les langages empruntant le paradigme de flux ou de règles, que nous présentons dans la section suivante, appartiennent à cette catégorie. La représentation diagrammatique est aussi le mode de représentation de prédilection des langages de requêtes [DP05, BMR99, PSAC97]. Le langage Pantagruel que nous proposons dans cette thèse appartient également à cette catégorie.

Les langages icôniques privilégient l'utilisation d'icônes pour représenter des objets et des actions. On trouve de tels langages dans le domaine de la programmation de jeux [Mac09, TR86]. Sikuli [YCM09] propose une utilisation originale des icônes pour définir des applications de type macros dans le gestionnaire de fenêtres du système d'exploitation MacOS. Les icônes sont des captures d'écrans de boutons ou d'éléments de l'interface graphique de MacOS. Chaque icône représente à la fois l'élément et l'action qui leur est associée.

Enfin, la troisième catégorie regroupe les langages de type formulaire [BAD+01, Mye91], dont l'éditeur de feuilles de calcul Excel est l'exemple typique. Ces langages se caractérisent par l'utilisation de tableaux et de cellules pour éditer et représenter visuellement les programmes.

Paradigmes de programmation En informatique, un paradigme se définit par un ensemble de concepts permettant de décrire des programmes selon une école de pensée particulière. Par exemple, le paradigme orienté objet offre aux programmeurs les concepts permettant d'exprimer un programme comme un ensemble d'objets. Les langages visuels proposent des visualisations particulières de programmes centrées sur les concepts caractérisant un paradigme donné. Certains d'entre eux sont une adaptation visuelle des paradigmes connus pour les langages de programmation : les paradigmes de programmation fonctionnelle [CG90], orientée objets [BGL95], orientée flux de données ou de contrôle, à base de règles, ou encore orientée formule de logique [PAR96]. Le choix d'un paradigme est conditionné par le domaine d'applications que l'on souhaite exprimer et le type de développeurs ciblés.

Nous nous intéressons dans la suite à deux paradigmes particulièrement adaptés pour des langages d'orchestration d'entités : le paradigme à base de flux et le paradigme à base de règles.

3.3.4 Paradigmes visuels pour l'orchestration

Programmation à base de flux

Le paradigme à base de flux se décline sous deux formes correspondant à deux explicitations du comportement d'un programme : le paradigme de flux de données, le paradigme de flux de contrôle. Les flux de données décrivent la séquence des données qui sont échangées à chaque étape d'exécution d'une instruction d'un programme, en exhibant la relation de dépendance entre ces données. Les flux de contrôles décrivent la suite des appels d'un programme, en exhibant le chemin menant une instruction à l'autre. Ces deux points de vue offrent une vision d'ensemble du comportement d'un programme. Examinons à présent quelques approches visuelles basées sur le paradigme de flux.

La notation BPMN (Business Process Modelling Notation) apporte une couche visuelle audessus du langage d'orchestration de services web *BPEL* (Business Process Execution Language) [Whi04]. Les services web sont des composants logiciels déployés sur Internet. BPMN est un langage orienté flux de contrôle au sens où il spécifie les échanges de messages et les invocations de méthodes entre les services web. Une application BPMN est composée d'activités (aussi dénommées des processus), d'événements, déclenchant l'exécution d'activités, et de portes, permettant d'indiquer un embranchement indiquant plusieurs chemins d'exécution possibles (pouvant être parallèles ou conditionnels) ou d'indiquer une jointure où différents chemins d'exécution se rencontrent. Le contexte d'exécution d'un comportement est soit un événement, soit une condition exprimée à travers une porte.

Les langages Prograph et MVPL (Microsoft Visual Programming Language) sont tous les deux des langages d'orchestration généralistes et adoptant le paradigme à base de flux de données [Mic, SC95]. Tous deux combinent ce paradigme avec celui orienté objet, dont les concepts incluent les classes, les objets et l'échange de messages. Les entités qu'ils orchestrent sont des capteurs de robots et des actionneurs, ou encore des composants logiciels. Par exemple, MVPL décrit son processus d'orchestration comme "*la coordination d'information entre un ensemble connecté de processus*". En d'autres termes, une application est un ensemble d'activités définies par des entrées et des sorties dont l'orchestration consiste à connecter les entrées des unes aux sorties des autres. Bien que généraliste, MVPL est particulièrement adapté pour définir des applications robotiques (par exemple, pour les briques de base de Lego Mindstorms®).

Néanmoins, si les langages utilisant le paradigme de flux permettent d'avoir une vision d'ensemble du comportement d'un programme, leur lecture n'est pas forcément plus aisée que lorsque que ces programmes sont exprimés de manière textuelle [GP92].

Programmation à base de règles

Le paradigme de programmation à base de règles est largement répandu dans les domaines de la programmation visuelle, et en particulier pour l'orchestration d'entités. Une règle est définie par une condition d'entrée, et une action à exécuter lorsque cette condition est vérifiée.

Les langages à base de règles, visuels ou non, sont utilisés dans des domaines variés, par exemple le contrôle de robots (Altaira [Pfe98], LegoSheets [GIL+95]) ou la programmation de jeux (AgentSheets [Rep93], KidSim [SCS94], Blender [vG03], ou encore VBBL [CRS98]). Dans le premier cas, le contrôle du comportement logique d'un robot consiste à orchestrer les capteurs et les actionneurs qui le composent.Dans le second cas, il s'agit de faire réagir les personnages constitutifs d'un jeu lorsqu'ils se trouvent dans une situation particulière (par exemple, contourner un obstacle).

Voici deux des raisons justifiant l'utilisation de ce paradigme dans un langage visuel :

– lorsqu'il y a une correspondance naturelle entre un modèle à base de règles et les objets constituants du domaine de programmation. C'est particulièrement valable dans les domaines relatifs à l'orchestration des robots et à la programmation de jeux. Dans le cas des robots, les données des capteurs définissent les conditions d'une règle, et les effecteurs sont les actions [Pfe98]. Dans le cas des jeux, l'état des agents constitue une condition, et le comportement de l'agent qui en résulte constitue une action.
– parce que chaque règle peut être aisément comprise indépendamment des autres [SCS94, NMB09, HR85]. En conséquence, et comme mentionné dans [Pfe98], la représentation des règles dans une fenêtre est très adaptée, car celles-ci peuvent être visualisées indépendamment.

Un inconvénient majeur de la programmation par règles est qu'elles nécessitent de considérer toutes les alternatives [NMB09] et d'engendrer ainsi une explosion de règles. Dans cette thèse nous proposons un moyen de pallier cet inconvénient.

3.4 Synthèse

Dans ce chapitre, nous avons présenté deux techniques, les langages dédiés et les supports de programmation visuels, permettant de faciliter l'activité de programmation. Nous avons vu que les langages dédiés permettent de capturer un vocabulaire approprié pour la programmation d'applications dans un domaine particulier. En outre, ils permettent d'améliorer la fiabilité des programmes. Nous avons également vu que les supports visuels aident à la prise en main d'un langage de programmation. L'utilisation de ces deux approches complémentaires facilite la programmation et permet de vérifier les programmes, garantissant ainsi accessibilité et fiabilité.

Dans cette thèse nous proposons le langage Pantagruel, associant ces deux approches afin d'outiller la méthodologie motivée par le chapitre précédent. Il s'agit d'un langage visuel dédié au développement d'applications d'orchestration d'entités, basé sur le paradigme de règles. Ce paradigme est approprié pour l'orchestration d'entités dans un environnement ubiquitaire : comme pour les robots, un environnement ubiquitaire est composé d'entités qui sont des capteurs et des actionneurs (ou effecteurs). De plus, le paradigme de règles semble beaucoup plus fréquent dans les langages de programmation destinés aux enfants [KP05], ce qui est un bon indicateur de l'accessibilité d'un langage.

4 Vers une méthode dédiée pour faciliter l'orchestration d'entités

Contexte Nous avons vu que faciliter la programmation d'applications à des utilisateurs finaux est un problème difficile. Par ailleurs, quel que soit le langage, la programmation nécessite une démarche logique pour aboutir à des programmes bien structurés et apportant la solution escomptée. Une démarche habituelle du programmeur est une approche d'analyse (*top-down*) : il énonce le problème et le décompose en sous-problèmes afin de le résoudre. Pour faciliter le développement d'applications d'orchestration, nous proposons de guider la programmation en adoptant une démarche similaire : l'objet à décomposer est un but, la solution est l'application le réalisant. Dans le champ d'applications de l'assistance à la personne, ce but peut être la réalisation d'une tâche par une personne avec déficiences cognitives.

Contributions Ce chapitre présente une méthodologie dédiée à la construction d'applications d'orchestrations, fondée sur l'expérience des éducateurs spécialisés. Plus spécifiquement, nous avons étudié leur démarche de construction de stratégies d'assistance. Afin d'amener les experts-métier vers la programmation d'applications, nous outillons cette méthodologie avec un langage de programation visuel. En procédant ainsi, nous établissons une passerelle entre l'expression des besoins et l'application finale qui réalise ces besoins.

Plan du chapitre Dans la section 4.1, nous présentons le point de départ posant les fondations de notre méthodologie. Puis nous en détaillons les principes dans la section 4.2. Nous concluons dans la section 4.3.

4.1 Point de départ

Avant de définir notre méthodologie, nous avons conduit des entretiens avec deux praticiens du monde médico-social et une personne impliquée dans une association d'aide à l'autonomie des personnes avec déficiences cognitives (que nous appelons simplement *personnes déficientes* dans la suite de ce chapitre). Nous rapportons ces entretiens et leurs résultats ici.

4.1.1 Entretiens avec des experts-métier

Le métier des professionnels de l'assistance à la personne consiste à apporter un soutien social, psychologique et intellectuel aux personnes souffrant de déficiences cognitives et de troubles du comportement.

Nous avons interviewé un psychiatre ayant trente ans d'expérience dans le suivi de personnes déficientes ou perturbées mentalement. Nous nous sommes ensuite entretenus avec un éducateur spécialisé ayant vingt ans d'expérience dans ce métier. Cet éducateur est responsable d'un foyer accueillant des personnes déficientes. Enfin, nous avons discuté avec une personne impliquée dans la mise en place de résidences équipées de technologies d'assistance pour les personnes atteintes du Syndrome de Down. Chaque entrevue a duré environ trois heures, au cours de laquelle nous avons collecté les informations permettant d'élaborer des projets d'assistance aux personnes déficientes. Ces projets regroupent un ensemble de solutions d'assistance, chacune décrivant un moyen d'aider une personne dans un contexte particulier (communication, relations sociales, éducation). Nous nous sommes intéressés aux solutions d'assistance visant à apprendre à une personne déficiente à se débrouiller seule dans ses activités quotidiennes, appelées stratégies éducatives. Cette démarche est basée sur les techniques cognitives comportementales, dont la finalité est de proposer des aides concrètes au quotidien des personnes déficientes.

4.1.2 Démarche de construction d'une stratégie éducative

Une démarche conventionnelle pour construire une solution d'assistance comprend trois étapes. D'abord, les déficiences d'une personne sont identifiées et décrites de manière détaillée, donnant lieu à un *profil*. Ensuite, une *stratégie éducative* est proposée et testée in situ. Enfin, cette stratégie est périodiquement évaluée, afin d'être renforcée ou assouplie selon l'évolution de la personne, son comportement et sa satisfaction vis-à-vis de cette stratégie.

Définition d'un profil Afin de déterminer l'aide appropriée pour assister une personne déficiente, les praticiens évaluent d'abord son niveau de déficience. Seuls les praticiens spécifiquement formés aux techniques d'évaluation peuvent assurer ce travail. Pour cela, ils s'entretiennent avec la personne ainsi qu'avec sa sphère sociale et familiale. Les premiers entretiens durent au minimum deux heures et permettent d'obtenir une estimation de ses inaptitudes. Un questionnaire d'évaluation standard est ensuite rempli. L'un des questionnaires les plus utilisés est l'Echelle de Vineland [Spa89]. Les questions sont habituellement structurées selon les aptitudes cognitives et physiques de la personne, par exemple, la mémoire à court-terme et la praxie manuelle (l'habileté de l'usage des mains). Les résultats du questionnaire sont ensuite interprétés, et donnent lieu à la définition d'un profil. Ce profil recense la liste des compétences de la personne pour effectuer des tâches concrètes (s'habiller, manger, utiliser un téléphone), pour avoir des relations sociales, pour coordonner ses mouvements, et pour exécuter des tâches complexes requises pour pratiquer une activité professionnelle. Dans cette thèse, où nous ciblons les applications d'assistance à domicile, nous nous sommes intéressés aux questions liées à la réalisation de tâches concrètes.

Définition et mise en place d'une stratégie éducative La seconde étape de la démarche éducative est la définition puis le déploiement d'une stratégie d'assistance individuelle adaptée au profil de la personne. Cette étape est dirigée par un éducateur spécialisé qui est chargé de l'accompagner. La définition d'une stratégie se décompose en trois phases :

- l'identification d'un objectif global à atteindre dans une période de temps donnée (qui peut s'étaler sur quelques mois, voire une année, selon la personne). Cet objectif concerne l'acquisition d'une autonomie dans un ou plusieurs champs d'activités (toilette, cuisine, loisir) ;
- la décomposition progressive de l'objectif en sous-tâches jusqu'à atteindre des tâches élémentaires que la personne peut comprendre et achever rapidement. Le pas de décomposition dépend fortement du profil de la personne et consiste généralement à déterminer des séquences de tâches : alors que les personnes ordinaires ordonnent instinctivement leurs activités journalières, les personnes avec déficiences cognitives éprouvent souvent des difficultés à en séquencer les tâches sous-jacentes ;
- l'identification de moyens d'assistance pour accompagner la personne dans la réalisation de ces tâches. Ces moyens d'assistance peuvent être humains ou matériels. Les moyens humains sont l'éducateur spécialisé, les aides médico-psychologiques ou un membre de la famille assurant le suivi d'une tâche spécifique. Les moyens matériels sont de différentes natures et sont choisis selon les sensibilités ou aptitudes spatiales, auditives et visuelles de la personne. Par exemple, des images sont parfois utilisées pour définir visuellement une séquence de tâches à effectuer. Parmi les autres objets communs cités par les praticiens, on trouve les horloges adaptées, les chronomètres, les carnets de bord, les jouets (peluches). Ces objets, placés à des endroits stratégiques de l'espace de vie de la personne, peuvent servir à symboliser une tâche spécifique.

Observation et ajustement de la stratégie La démarche de l'éducateur spécialisé dans l'assistance à une personne est un processus par essai-erreur. Périodiquement, il ré-évalue le comportement de la personne déficiente face à la stratégie adoptée. Selon que la personne s'améliore ou régresse, ou dès que des situations imprévues surviennent (une chute, une maladie), l'éducateur spécialisé ajuste la stratégie. Ces ajustements consistent à modifier la décomposition initiale des buts de trois façons : décomposer plus finement un but, supprimer des niveaux de décomposition inutiles, ou étendre le but avec des sous-buts plus complexes. Les moyens d'assistance sont ensuite adaptés selon ces ajustements.

4.1.3 Le cas d'Henrick

Dans cette section, nous présentons un cas d'étude dont nous nous servons pour illustrer la méthodologie dans la suite de ce document. Ce cas d'étude rapporte la vie d'une personne déficiente appelée Henrick. Ce personnage fictif est extrait de la thèse de Arne Svensk [Sve03]. Ce rapport agrège un ensemble de situations auxquels Svensk a été confronté dans sa carrière d'éducateur spécialisé.

Henrick vit dans un appartement indépendant. Il a besoin d'être assisté tout au long de la journée, depuis l'heure où il se réveille jusqu'au moment où il se couche. En particulier, Henrick doit se lever à une heure spécifique selon l'activité prévue dans sa journée. Une activité peut être un travail (en l'occurrence, il s'occupe de l'entretien bi-hebdomadaire d'un terrain de football), ou un loisir (se rendre à une exposition artistique, pratiquer un sport). Pour l'aider à se lever, son réveil est configuré pour jouer une musique dont le genre est déterminé par l'activité du

jour.

Une fois qu'Henrick est réveillé, il a besoin de prendre une douche, puis de s'habiller, et enfin de prendre son petit-déjeuner. Comme il a des difficultés à réguler la température de l'eau, des mécanismes automatiques doivent être intégrés à la douche pour lui éviter les risques de brûlures. Svensk propose un système tel que, lorsque "[*il*] *ferme la porte, l'eau est automatiquement réglée à la bonne temperature*". Cet automatisme pallie les difficultés d'Henrick, faisant de la toilette une étape ordinaire de sa routine journalière. Après la douche, Henrick s'habille. Pour choisir les vêtements appropriés à la météo courante, il est assisté par un éducateur spécialisé, qui, chaque jour, prépare les vêtements en accord avec la météo relevée la veille. Puis il se rend à la cuisine pour prendre son petit-déjeuner. Cette tâche repose sur un ensemble d'appareils mis à sa disposition et dotés d'interfaces simplifiées afin qu'il les utilise sans assistance : une machine à café, un grille-pain, et un four micro-onde.

Henrick ne sait pas lire l'heure sur une horloge classique ; un mécanisme simplifié doit alors lui être proposé pour lui permettre de garder le fil du temps qui passe et des tâches qu'il doit réaliser à différents intervalles de temps. L'encadrement des tâches quotidiennes d'Henrick permet de limiter les situations de panique causées par une tâche oubliée ou réalisée pendant un temps trop long.

4.2 Une approche orientée buts

L'approche que nous proposons est basée sur la démarche de construction d'une stratégie éducative. La décomposition d'un but est centrale dans l'élaboration de cette stratégie, notre méthodologie sera donc centrée sur ce processus de décomposition. Dans la suite, nous présentons les critères de décomposition d'un but. Nous définissons ensuite une méthodologie outillée permettant de réaliser un but et formalisant la démarche adoptée par les éducateurs spécialisés. Nous illustrons notre propos en l'appliquant au cas d'Henrick décrit précédemment.

4.2.1 Critères de décomposition

La décomposition d'un but peut se faire selon deux critères : d'une part la *chronologie* des tâches constituantes de l'activité correspondante, et d'autre part les *situations* particulières auxquelles l'activité ou ses sous-tâches peuvent se rattacher. L'arborescence de la décomposition selon ces critères est montrée dans la figure 4.1.

Chronologie Le critère de chronologie est motivé par la difficulté qu'ont les personnes déficientes à séquencer les tâches. Un but peut alors être décomposé en mettant en évidence une dépendance chronologique entre les sous-buts obtenus. Pour exhiber cette dépendance, nous introduisons une notation d'*ordre* (+ ou ×), indiquant si un ensemble de sous-buts fait partie d'une chronologie (signe +) ou non (signe ×).

Par exemple, dans la figure 4.2, le but principal "faire sa toilette" est décomposé en deux sous-buts chronologiquement dépendants : "prendre une douche" et "nettoyer la cabine de douche". En revanche, le but "se brosser les dents" peut être fait à tout moment au cours de la toilette.

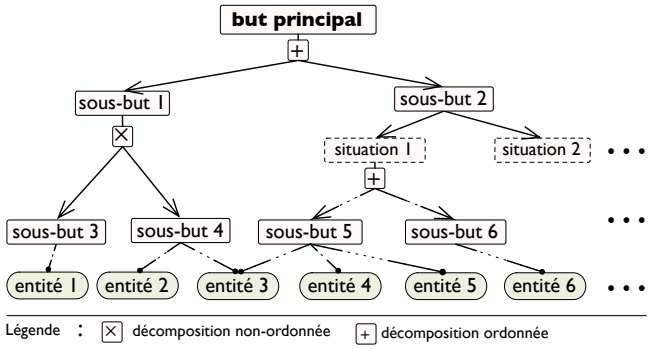

FIG. 4.1: Arborescence de la décomposition

En explicitant les séquences de tâches, la décomposition chronologique permet à l'éducateur d'organiser son activité d'assistance conformément aux besoins de la personne déficiente.

Situation Le critère de situation est motivé par la nécessité d'adapter l'assistance selon les situations anormales dans lesquelles la personne déficiente peut se trouver : par exemple, si la personne est malade, ou entre dans un état de panique, ou fait une chute, l'assistance habituelle n'est plus adéquate. Prenons le sous-but "prendre une douche" dans la figure 4.2, qui se décline en deux situations : une situation normale et une situation de chute. En situation normale, ce but est décomposé chronologiquement en quatre étapes, de "aller sous la douche" à "fermer le robinet". Si, en prenant sa douche, la personne fait une chute (par exemple suite à une crise d'épilepsie), il faut fermer le robinet du mitigeur et appeler un éducateur spécialisé.

La décomposition selon les situations correspond à une spécialisation de but. Elle permet à l'éducateur d'expliciter les différents cas particuliers survenant dans le quotidien d'une personne déficiente.

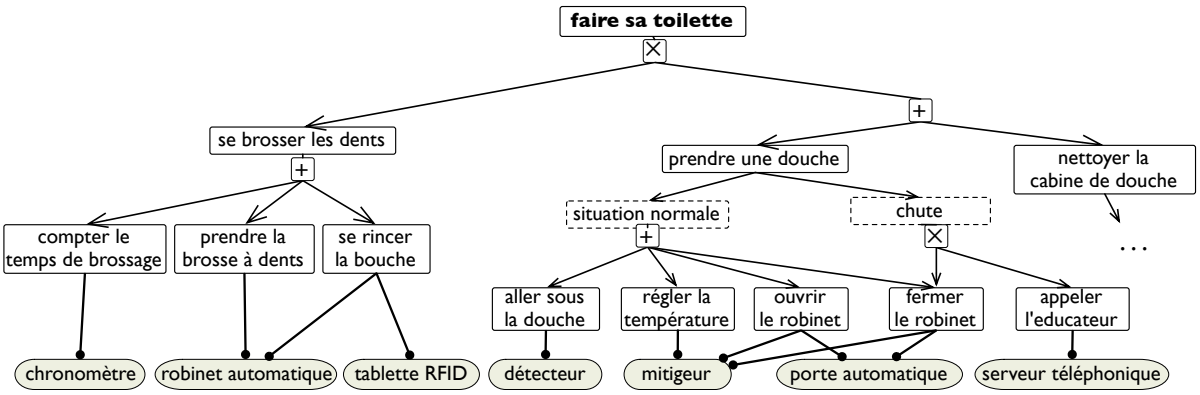

FIG. 4.2: Exemple de décomposition

4.2.1.1 Généralisation

L'approche orientée buts est valable pour d'autres champs que l'assistance à la personne : dans le champ de la domotique, les buts peuvent correspondre à des opérations automatisant le fonctionnement d'appareils électroménagers ; dans celui de la gestion d'information, ils peuvent être associés à des opérations de traitement de l'information et de mise à jour des écrans d'affichage. Une décomposition chronologique minimale est la configuration, puis le déclenchement d'une action selon cette configuration. Le critère de situation peut être utilisé pour identifier différentes adaptations d'un but donné. Par exemple, dans un établissement scolaire, un but visant à en contrôler l'accès peut être adapté différemment selon les journées "portes ouvertes", les périodes d'examen ou de congés au cours desquelles un système de surveillance doit être mis en place. En séparant les diverses préoccupations d'un but, le critère de situation permet d'en faire une description modulaire, aidant ainsi l'utilisateur à structurer les applications qui, à terme, en découlent.

4.2.1.2 Une relation avec la modélisation orientée traits

Il y a un parallèle intéressant à faire entre les critères de décomposition proposés et la technique d'analyse de domaine FODA (pour *Feature-Oriented Domain Analysis*) de Czarnecki [CE00]. Cette technique a pour objectif de déterminer les concepts d'un domaine (par exemple une voiture), et d'expliciter les propriétés (les traits ou *features*) de ces concepts. Elle donne lieu à un modèle dit modèle de traits (*feature model*), décrivant les variations de ces propriétés (par exemple, la propriété "transmission" d'une voiture peut être déclinée en mode automatique ou manuel), et les dépendances entre ces propriétés (par exemple, une voiture doit définir obligatoirement un mode de transmission et un type de moteur, et un moteur peut être diesel ou essence, mais pas les deux). Ces propriétés sont de trois types : alternatif – pour désigner le fait qu'une propriété peut avoir plusieurs variations mutuellement exclusives –, obligatoire, ou optionnel. Adapter ces propriétés pour compléter nos critères et raffiner le processus de décomposition est une perspective intéressante pour les travaux futurs à cette thèse.

4.2.2 Présentation générale

Notre méthodologie décrit la démarche à suivre pour la recherche et la mise en oeuvre d'une application, depuis l'analyse des besoins jusqu'à la construction de l'application. Elle combine deux phases : une phase d'analyse, consistant à décomposer un but en objets élémentaires, et une phase de synthèse, consistant à construire une application dont les objets élémentaires sont les briques de base. Cette démarche est résumée par la figure 4.3. L'étape charnière est la modélisation des entités (étape c.1), qui permet de décrire chaque sous-but sous la forme d'une application exécutable.

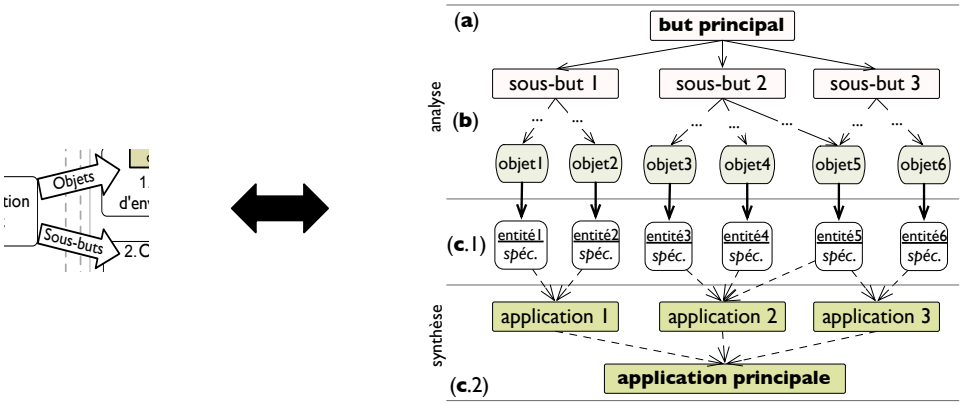

FIG. 4.3: Une approche combinant une phase d'analyse et une phase de synthèse

La méthodologie s'inscrit dans un processus complet allant jusqu'au déploiement des applications (par exemple, vers une plate-forme de simulation), et où chaque étape de la phase de synthèse est outillée. La figure 4.4 en présente les cinq étapes, chacune trouvant une correspondance dans le processus de construction d'une stratégie éducative :

(a) l'*expression des besoins* correspond à l'identification d'un profil. Elle donne lieu à des buts se référant à des activités que la personne déficiente est amenée à réaliser ;

(b) la *décomposition* correspond à la définition d'une stratégie éducative basée sur ces buts ;

(c) l'*application* correspond à la mise en place de la stratégie dans l'environnement de la personne déficiente ;

(d) le *test* correspond à l'observation du comportement de la personne vis-à-vis de la stratégie ;

(e) l'*ajustement* correspond à l'adaptation de la stratégie selon les observations précédentes.

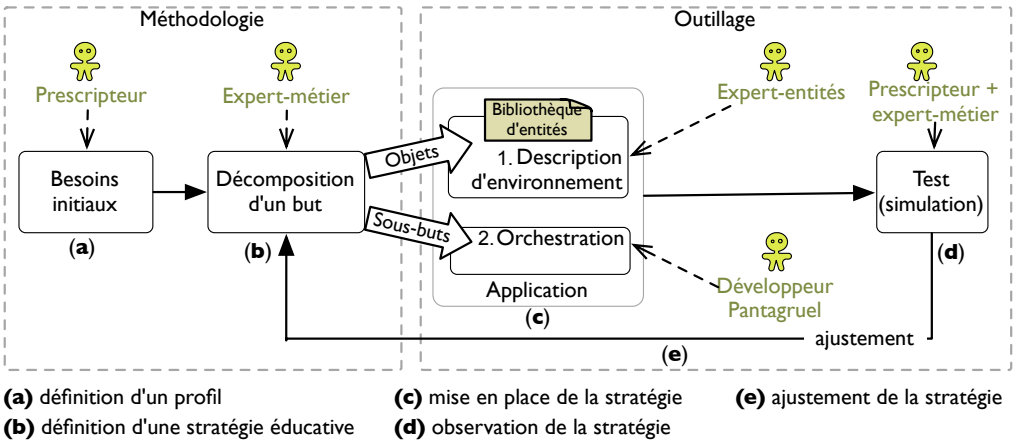

FIG. 4.4: Vue générale de la méthodologie outillée

Les étapes de la méthodologie décrivent la passerelle entre les besoins et le déploiement d'une application en environnement réel. Initialement, le demandeur exprime ses besoins (a), donnant lieu à des buts principaux qui sont analysés par l'expert-métier durant la phase de décomposition (b). Cette étape consiste à décomposer progressivement les buts identifiés, jusqu'à atteindre des buts élémentaires pouvant directement être associés à des objets. Le choix des objets peut être guidé en mettant à disposition une bibliothèque d'entités réelles, relevant d'un champ d'applications spécifique. Cette bibliothèque est extensible, c'est-à-dire qu'elle peut être enrichie avec de nouveaux objets ou de nouvelles fonctionnalités sur les objets existants. Un expert des entités réelles du champ considéré modélise ensuite ces entités afin d'accéder aux fonctionnalités requises par l'expert-métier (c.1). Ces objets sont les briques de base de la construction d'une application d'orchestration, dont la finalité est d'assister la réalisation du but principal (c.2). L'exécution de cette étape est le rôle d'un développeur Pantagruel. A terme, nous souhaitons que l'expert-métier prenne ce rôle. L'application est finalement testée, par exemple au moyen d'une plate-forme de simulation (d). Les tests peuvent faire apparaître la nécessité d'ajuster les applications (e). Cette dernière étape peut reposer sur l'arbre de décomposition issu de l'étape (b), en ajustant celui-ci par la suppression ou l'ajout de buts (par exemple en introduisant un pas de décomposition supplémentaire), ou en modifiant la collection d'objets identifiés.

4.2.3 Le cas d'Henrick : analyse

Nous illustrons ici la partie analyse de notre méthodologie sur le cas d'Henrick en choisissant comme but principal le bon démarrage de sa journée. Le premier pas de décomposition donne lieu à quatre sous-buts : (1) se lever, (2) prendre une douche, (3) s'habiller, (4) gérer la suite des tâches restantes. Le but (4) est transverse aux trois premiers et permet d'assurer le bon déroulement du but principal. Par souci de concision, nous en présentons maintenant uniquement une décomposition chronologique et en situation normale.

Se lever L'étape de lever d'Henrick est élémentaire et se décompose directement en deux objets : un calendrier logiciel annonce la journée qui démarre et un radio-réveil déclenche une sonnerie musicale (figure 4.5). Le genre musical joué pour réveiller Henrick est paramétré en fonction de l'événement envoyé par le calendrier, indiquant l'activité du jour. On peut envisager d'autres scénarios faisant usage du calendrier : afficher des pictogrammes correspondant à une activité (par exemple, le nettoyage de ses vêtements), ou transmettre un message vocal annonçant ses émissions de télévision préférées, où les activités et les émissions sont inscrites dans le calendrier.

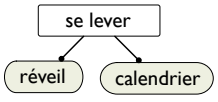

FIG. 4.5: Se lever

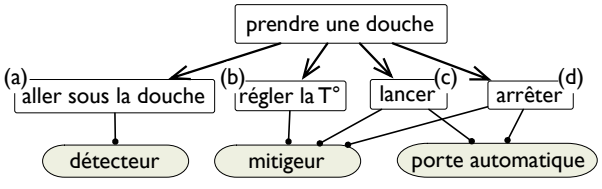

FIG. 4.6: Prendre une douche

Prendre une douche Pour faciliter la toilette, la salle de bain d'Henrick doit être équipée d'une douche déclenchant automatiquement l'écoulement de l'eau à la bonne température (figure 4.6, sous-buts (*b*) et (*c*)), une fois que la porte est fermée et qu'Henrick est détecté sous la douche (sous-but (*a*)). Si la porte est ouverte, l'eau cesse de couler (sous-but (*d*)). Nous pouvons raffiner ce mécanisme en supposant que la porte ne peut être fermée que de l'intérieur, et que par défaut elle est ouverte. Déclencher la douche dépend d'entités comme une porte automatique, un mitigeur, et un détecteur de mouvement. Notons que cette décomposition est propre au cas d'Henrick et que dans la réalité, le contrôle de la douche peut être largement assoupli. D'autres personnes déficientes ont les moyens de contrôler la douche et peuvent préférer un mode manuel qui leur donne un plus grand degré de liberté, par exemple régler eux-mêmes la température de l'eau. Dans ce cas les objets issus de la décomposition peuvent être différents.

Se vêtir Nous proposons à l'éducateur spécialisé de réduire son intervention dans la tâche d'habillement d'Henrick (figure 4.7), concernant le choix des vêtements. Pour cela on peut utiliser un service Web, recueillant la météorologie courante, et la technologie RFID, permettant d'identifier électroniquement des objets. Ainsi, Henrick peut être informé du dernier rapport météorologique (sous-but (*a*)), dans la pièce où ses vêtements sont rangés. Pour faciliter le choix du bon ensemble de vêtements, chacun est marqué d'une étiquette ou *tag* RFID (sous-but (*b*)). Ces tags sont ensuite lus lorsqu'Henrick passe un portique installé dans la pièce et muni d'un lecteur de tags RFID. La combinaison de ces entités – tag et portique – permet d'alerter Henrick si les vêtements qu'il porte ne conviennent pas à la météorologie du jour.

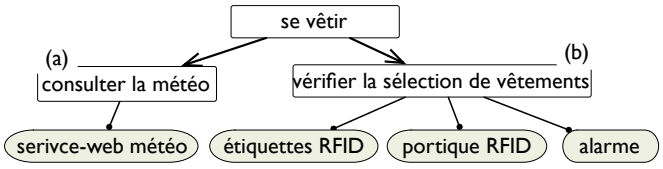

FIG. 4.7: Se vêtir

Gérer la suite de tâches Henrick est assisté à des moments spécifiques de la journée lorsqu'une liste de tâches doit être accomplie dans un intervalle de temps limité. Par exemple, le matin, il doit se préparer et quitter la maison à temps pour se rendre au travail, et le soir préparer son dîner et être prêt pour ne pas manquer le début de ses programmes télévisés favoris. Pour cela il est nécessaire de donner à Henrick des indications sur le temps moyen à passer pour effectuer une tâche (par exemple, s'habiller ou se doucher). Cette information peut être affichée à des endroits stratégiques de l'appartement de sorte qu'elles soient dans son champ de vision quand il en a besoin. Pour cela, nous proposons d'équiper son appartement avec un prompteur de tâches (figure 4.8). Ce composant logiciel détermine le temps restant pour accomplir une tâche et maintient la liste des tâches qu'il reste à effectuer. Le prompteur de tâches est couplé à des écrans pour afficher continuellement son état sous une forme appropriée (par exemple à l'aide d'images). Les tâches peuvent être suivies grâce à certaines entités physiques comme les détecteurs placés à divers endroits de l'appartement ou le mitigeur pour vérifier le fonctionnement de la douche.

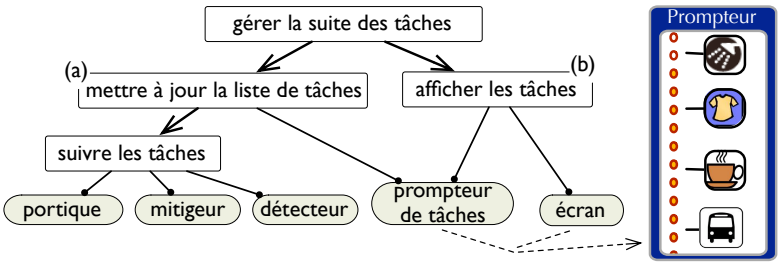

FIG. 4.8: Gérer la suite des tâches – Un prompteur de tâches

4.3 Synthèse

Dans ce chapitre, nous avons proposé une méthodologie définissant une passerelle entre l'expression des besoins et la réalisation d'applications ubiquitaires. Puisque ces applications doivent rendre service aux utilisateurs dans leurs activités quotidiennes, il est primordial de garantir cette corrélation. Notre méthodologie fait un premier pas vers cet objectif car elle permet de connecter l'étape d'analyse à l'étape de synthèse au moyen des entités, fournissant les fonctionnalités requises pour développer les applications.

Nous en avons illustré ici l'étape d'analyse en l'appliquant au cas d'étude Henrick. Cette méthodologie doit son originalité au fait qu'elle est *dédiée* au domaine de l'orchestration d'entités, et qu'elle combine deux stratégies de développement logiciel, analyse et synthèse, par l'intermédiaire des descriptions d'entités. Dans le chapitre 8, nous justifions cette originalité par un tour d'horizon des approches d'orchestration existantes dans l'informatique ubiquitaire.

Pour mettre en oeuvre l'étape de synthèse, nous associons à cette méthodologie le langage visuel Pantagruel. Il permet de modéliser des entités sous la forme d'une *taxonomie*, puis de construire des applications consistant à *orchestrer* ces entités pour réaliser les sous-buts résultant de l'étape d'analyse. La définition de ce langage est l'objet du chapitre suivant. Grâce à ce langage, nous complétons, dans le chapitre 6, l'illustration du cas d'Henrick par l'étape de synthèse. Notons que les applications qui y sont présentées ont été testées sur la plate-forme de simulation développée par notre équipe de recherche.

5 Un langage visuel paramétré par une taxonomie

Contexte La méthodologie que nous venons de définir propose une passerelle entre l'expression des besoins et les objets qui serviront de support aux applications. Pour concrétiser cette passerelle, il est nécessaire de fournir des outils appropriés permettant de construire ces applications à partir des éléments résultant de l'analyse.

Contribution Ce chapitre présente un tel outil, nommé Pantagruel. C'est un langage visuel paramétré par la définition d'un environnement. Cet environnement est défini grâce à une taxonomie, composée d'un ensemble de classes d'entités exposant des interfaces d'interaction. Ces entités sont orchestrées par un langage utilisant un paradigme nommé *capteur-contrôleur-actionneur* prenant la forme de règles "conditions → actions". Pantagruel a été formalisé dans une sémantique dénotationnelle pour faciliter la vérification de programmes (ce que voyons au chapitre 7). Nous montrons l'applicabilité de notre approche sur le champ de la gestion d'information. Une partie de ce chapitre a été publiée [DMC09].

Plan Après la présentation d'un autre champ d'applications illustratif dans la section 4.2.3, nous donnons, dans la section 5.2, une présentation générale du langage Pantagruel. Nous détaillons dans les sections 5.3 et 5.4 les deux sous-couches composant le langage : le langage de taxonomie et le langage d'orchestration. Puis nous décrivons le modèle d'exécution du langage, de manière informelle dans la section 5.5 puis formellement dans la section 5.6. Enfin, l'implémentation du langage est présentée dans la section 5.7.

5.1 Un autre champ d'applications

Nous avons abordé dans le chapitre précédent les applications de l'assistance à la personne. Notre proposition couvre d'autres champs d'applications. Nous illustrons ainsi notre approche, tout au long de ce chapitre, par les applications de gestion de réunions qui constituent une partie d'un autre champ : la gestion d'information (décrit dans la section 2.1.2, page 19). Ces applications visent à informer les employés d'une entreprise (ou les personnes d'un établissement d'éducation) sur les réunions ayant lieu au sein de ces bâtiments. Elles se basent sur les informations concernant les employés et les caractéristiques des bâtiments, comme le nombre de salles de réunion disponibles. Par exemple, lorsqu'une réunion est prévue pour trois personnes dans la salle de réunion d'une entreprise à 15h, ces trois personnes peuvent être prévenues cinq minutes avant, au moyen d'une alerte sur leur téléphone mobile ou un envoi d'e-mail vers leur ordinateur selon leur localisation.s

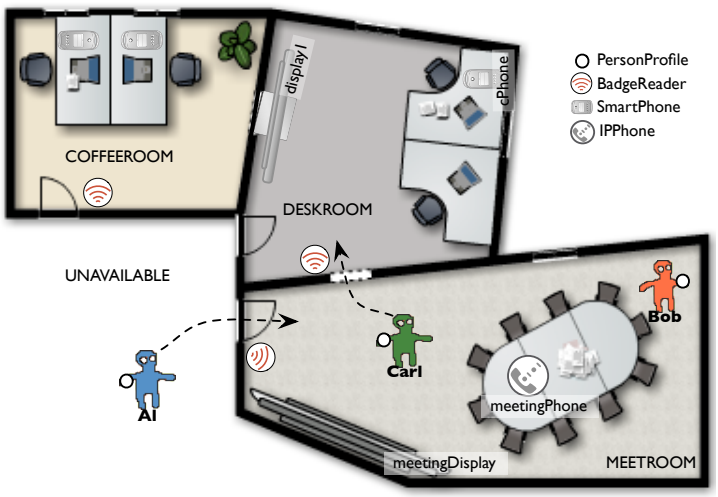

FIG. 5.1: Un espace professionnel

Ces applications orchestrent diverses entités : des lecteurs de badges RFID, des agents de présence, des écrans LCD, des ordinateurs portables, des téléphones mobiles intelligents (*smartphones*), un agenda partagé et un serveur de messagerie instantanée. Un exemple d'espace professionnel équipé de telles entités est présenté sur la figure 5.1.

Un lecteur de badge RFID est placé dans chaque pièce (COFFEEROOM, DESKROOM, et MEETINGROOM) pour détecter la localisation des personnes (Al, Bob et Carl) détenant un badge RFID. Un agenda partagé mémorise chaque réunion et les informations qui lui correspondent comme la date, l'heure de début et de fin, la salle et les participants.

Divers scénarios d'orchestration peuvent être imaginés pour gérer des réunions, comme le partage de notes dans les conférences participatives [LD99], ou leur planification selon les salles et l'emploi du temps des participants [KSD+03]. Prenons le cas d'un scénario destiné à assurer le bon démarrage des réunion et à en faciliter le suivi par les participants. Lorsqu'une réunion démarre, l'écran de l'ordinateur de l'organisateur s'affiche sur l'écran LCD de la salle de réunion où cette dernière a été planifiée, permettant le partage de documents et de présentations avec les participants. Les participants souhaitant prendre part à la réunion depuis un endroit distant et connectés à un serveur de messagerie instantanée sont invités à démarrer une session audio-visuelle depuis leur smartphone. Simultanément, la présentation de la réunion est alors affichée sur leur smartphone.

De tels scénarios d'orchestration doivent pouvoir être facilement adaptés selon les besoins et préférences des utilisateurs, mais également selon les retours qu'ils peuvent rapporter vis-à-vis du comportement de ces applications. Par exemple, certains utilisateurs souhaiteraient être alertés à l'aide d'un message vocal, alors que d'autres préféreraient recevoir un e-mail. D'autres utilisateurs souhaiteraient être prévenus par l'intermédiaire d'une tierce personne (leur assistant(e) de projet, par exemple). Ces scénarios peuvent avoir un impact considérable sur la vie des personnes et illustrent l'importance de faciliter la création et l'amélioration d'applications d'orchestration. Accomplir cet objectif peut rendre ces applications à la fois compréhensibles à

une large audience et proches des spécifications informelles fournies par les utilisateurs.

Le scénario que nous avons décrit ici met également en avant la richesse des entités disponibles aujourd'hui, nécessitant une approche expressive pour les orchestrer. Finalement, l'espace professionnel précédemment décrit est constitué d'entités pour lesquelles de nombreuses variations sont possibles, nécessitant une approche d'orchestration haut niveau qui fasse abstraction de ces variations. Dans les sections suivantes, nous présentons Pantagruel, un langage qui satisfait ces deux exigences, et nous l'illustrons à travers une modeste mise en application du scénario proposé.

5.2 Présentation générale de Pantagruel

Pantagruel est un langage visuel dédié au développement de logiques d'orchestration paramétrées par une *taxonomie* d'entités décrivant un champ d'applications particulier. Spécifiquement, Pantagruel est composé de deux sous-couches langages : un langage de taxonomie et un langage d'orchestration. L'approche de développement de Pantagruel comprend deux étapes : d'abord, un environnement ubiquitaire est décrit selon les entités qui le constituent. Cet environnement est conforme à une taxonomie d'entités. Ensuite, une application est développée : cette étape est dirigée par la taxonomie d'entités et consiste à orchestrer ces entités en utilisant des constructions de haut niveau.

La description d'un champ d'applications permet à notre approche d'être instanciée selon un *environnement* particulier. Cette description répertorie un ensemble de *classes* d'entités pertinentes pour le champ ciblé. Chacune de ces classes spécifie une interface donnant accès aux fonctionnalités des entités de ces classes. Puisque les applications d'orchestrations, ou *logiques d'orchestration*, sont écrites pour une description particulière d'environnement, les entités qu'elles orchestrent doivent être combinées conformément à cette description.

Pour faciliter la programmation de logiques d'orchestrations, nous avons développé un éditeur visuel utilisant le paradigme capteur-contrôleur-actionneur (que nous abrégeons SCA, pour Sensors-Controllers-Actuators – figure 5.2). Ce paradigme est adapté à des utilisateurs non-programmeurs, en témoigne l'évaluation que nous en faisons dans le chapitre 8, de même que son utilisation dans des domaines aussi variés que la programmation de jeux (par exemple, Blender [vG03] et KidSim [SCS94]) et le contrôle de robots (par exemple, Altaira [Pfe98], LegoSheets [GIL+95] pour les robots de LegoMindstorm [Leg]). De la même façon qu'une logique de jeux, une logique d'orchestration collecte des données contextuelles avec des capteurs (sensors), les combine avec un contrôleur (controller), et réagit en déclenchant des actionneurs (actuators). Chacune de ces combinaisons s'appelle une règle d'orchestration.

Notre éditeur de programmation visuel fournit en outre au développeur une interface paramétrée par la description de l'environnement. En particulier, il exploite les informations relatives aux entités de cet environnement pour guider le développeur dans la définition de règles d'orchestration. Nous examinons à présent en détail les deux sous-couches langage de Pantagruel.

FIG. 5.2: Le paradigme Capteur-Contrôleur-Actionneur

5.3 Un langage de taxonomie

Pour s'abstraire des variations entre les entités, Pantagruel est doté d'un langage déclaratif dont l'objectif est de décrire une *taxonomie* d'entités. Les classes d'entités déclarées dans cette taxonomie constituent un champ d'applications et peuvent être instanciées pour décrire un environnement particulier. Un extrait de la taxonomie, décrivant le champ de la gestion de réunions dans un espace professionnel, est présenté dans la figure 5.3. Pour faciliter la lecture, nous empruntons des éléments syntaxiques du langage UML [BRJ99].

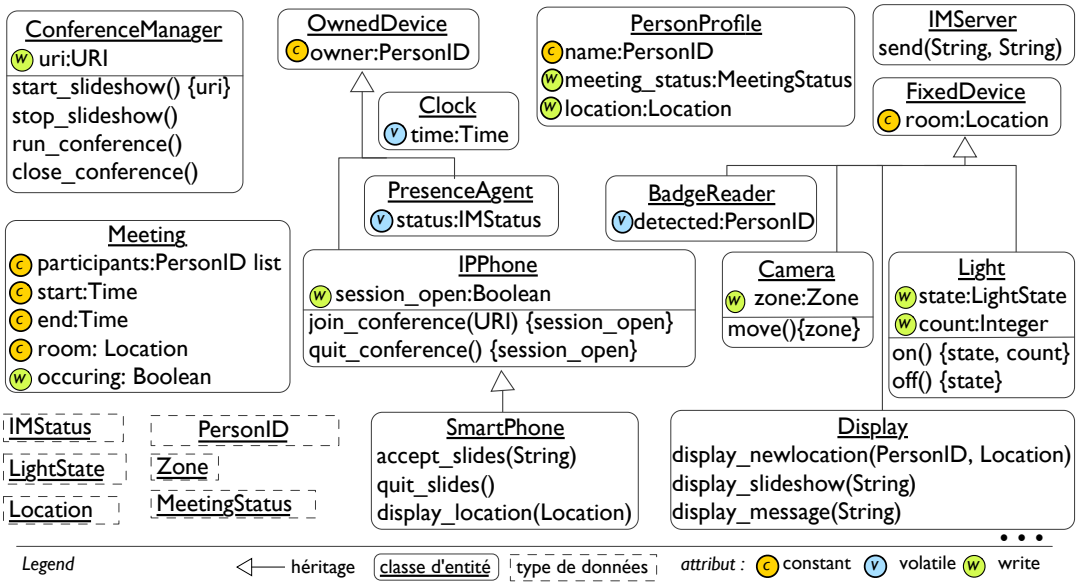

FIG. 5.3: Extrait d'une taxonomie pour la gestion de réunions

5.3.1 Description d'un champ d'applications

Décrire un champ d'applications consiste à déclarer un ensemble de classes d'entités, chacune caractérisant des entités ayant des fonctionnalités communes. La déclaration d'une classe d'entités indique la manière dont les entités de cette classe peuvent interagir. Elle permet de plus de faire abstraction des variations différenciant ces entités, facilitant la réutilisation d'une description d'un champ pour différents environnements. Comme dans l'approche orientée objets, une *entité* est définie comme une instance d'une classe particulière.

Une classe d'entités définit des attributs, représentant un *contexte* particulier, et des méthodes pour accéder aux fonctionnalités des entités. L'ensemble des attributs et des méthodes consti-

tuent une *interface d'interaction* des entités. Les classes peuvent être organisées de manière hiérarchique, permettant l'héritage des attributs et des méthodes et le raffinement des descriptions d'entités. A noter que des types de données sont également déclarés dans la taxonomie afin de typer les attributs et les méthodes.

Dans les paragraphes qui suivent, nous définissons l'interface d'interaction d'une entité, composée des interfaces de contexte et d'action. Ces concepts résultent de notre analyse du domaine de l'orchestration d'entités pour l'informatique ubiquitaire.

Une interface de contexte Le contexte représente l'état courant d'un environnement ubiquitaire, qui peut être capté ou calculé par les entités (par exemple par des capteurs physiques pour l'un ou des composants logiciels pour l'autre). Les informations de contexte jouent un rôle clé dans l'expression de conditions dans les applications d'orchestration. Leur niveau d'abstraction doit correspondre aux besoins et au vocabulaire du développeur d'applications qui va l'utiliser. Si une information de contexte est trop précise, une application devra manipuler des données de bas niveau, polluant le code avec des conditions et des opérations supplémentaires sur les structures de données. Un tel code ne pourra alors pas mettre en évidence les objectifs et les besoins que l'application réalise. Si, au contraire, l'information est trop abstraite, la réutilisation des entités risque d'être compromise car elle entraînera la définition d'une taxonomie trop détaillée ; en effet, pour pouvoir cibler une variété suffisamment riche d'applications, cette information imposera de décrire de nouvelles classes pour exprimer les variations d'entités pouvant figurer dans le champ d'applications. Définir une taxonomie nécessite donc d'analyser le champ d'applications, afin de faciliter sa réutilisation.

Pour favoriser cette analyse, nous avons identifié trois sortes d'éléments de contexte pour caractériser l'état d'un environnement : ils sont *constants*, *externes* ou *applicatifs*. Ces éléments, que nous présentons maintenant, peuvent être utilisées dans la partie capteur d'une règle d'orchestration.

Un élément de contexte constant est une information caractérisant une entité particulière et dont la valeur est définie une fois pour toutes lors du déploiement d'un environnement. Il peut s'agir de la pièce où une entité fixe est installée, du propriétaire d'une entité mobile comme un smartphone, ou de toute autre information qui ne change pas au cours du cycle de vie de l'environnement qui l'héberge. Nous modélisons ce contexte par un attribut *constant*  . Par exemple, la classe d'entités **FixedDevice** (entité fixe) déclare un attribut room (pièce) qui est constant pour une configuration d'environnement donnée. Ainsi, une entité appartenant à la classe **BadgeReader** (lecteur de badge) qui en hérite est supposée avoir une position constante.

Un élément de contexte externe représente une information provenant de l'extérieur et évoluant continuellement. Ce type d'information peut être recueilli par des capteurs physiques, comme un capteur de température ou un lecteur RFID renvoyant des informations de localisation à partir de badges RFID, ou par des composants logiciels (par exemple, un client de messagerie instantané indiquant le statut d'une personne sur son ordinateur). Pour modéliser ces informations, nous introduisons la notion d'attribut *volatile* . Une entité déclarant cet attribut met périodiquement à jour sa valeur afin de la communiquer à un programme Pantagruel et lui permettre ainsi de déclencher les actionneurs d'une règle d'orchestration. Par exemple,

considérons la classe **BadgeReader** dans la figure 5.3. Chacune de ses entités porte un attribut detected correspondant à l'identifiant d'une personne munie d'un badge RFID. Cet identifiant est mis à jour lorsqu'une personne passe son badge devant le lecteur.

Enfin, un élément de contexte applicatif correspond à des informations calculées par l'application d'orchestration. Pour modéliser un tel contexte, nous définissons des attributs *write* Ⓦ, qui peuvent être modifiés dans la partie Actionneur d'une règle d'orchestration. Prenons l'exemple de l'attribut meeting_status, défini sur la classe **PersonProfile** définissant le profil d'une personne. Un programme Pantagruel peut affecter cet attribut à une valeur caractérisant le profil de la personne selon sa participation à une réunion et sa présence dans la salle de réunion (par exemple, cette valeur peut être PRESENT, REMOTE ou ABSENT).

L'information de contexte étant explicitée comme une abstraction dans le langage Pantagruel, nous facilitons sa manipulation et sa lisibilité. Dans le chapitre 7, nous montrons également que cette abstraction facilite la vérification de la logique d'orchestration.

Une interface d'actions Les fonctionnalités fournies par une classe d'entités sont rendues accessibles par des déclarations de méthodes typées. Ces méthodes permettent de faire exécuter aux entités des actions affectant éventuellement l'environnement c'est-à-dire agissant sur le contexte applicatif. Cet interfaçage est notamment approprié lorsque l'implémentation de ces actions est hors de portée de Pantagruel. Par exemple, Pantagruel n'est pas adapté pour programmer la séquence d'opérations requise pour faire effectuer une rotation à une caméra. En conséquence, la classe **Camera** inclut une signature de méthode move pour accéder à cette fonctionnalité.

Les méthodes sont utilisées dans la partie Actionneur d'une règle d'orchestration. Lorsqu'elle est invoquée, une méthode peut affecter le contexte applicatif. Prenons comme exemple la classe **Light** (Lampe). Les lampes ont une espérance de vie qui dépend de la fréquence à laquelle elles sont allumées. Pour permettre au développeur de définir des règles qui contrôlent ces lampes et optimisent leur durée de vie, nous avons modélisé cette classe avec deux attributs write, state et count. Le premier porte le statut de la lampe (par exemple, ON ou OFF). Le deuxième compte les fois où la lampe est activée, modélisant l'élément de contexte applicatif "fréquence d'activation" de la lampe. La classe **Light** définit également deux signatures de méthodes, on() et off(). Si ces méthodes modifient le contexte applicatif, le développeur, lorsqu'il les utilise, doit pouvoir en être conscient. Nous complétons pour cela les signatures des méthodes avec des *effets*, exposant la liste des attributs de la classe qui peuvent être modifiés par ces méthodes. Ainsi, la méthode on() déclare dans ses effets les attributs write {state, count}.

5.3.2 Description d'un environnement

La taxonomie ainsi définie est ensuite utilisée pour définir des environnements concrets. La définition d'un environnement pour un champ d'applications particulier est un ensemble d'entités créées en instanciant les classes de la taxonomie décrivant ce champ. Ces entités peuvent être créées statiquement, c'est-à-dire avant exécution des applications d'orchestration, ou dynamiquement, c'est-à-dire une fois que ces applications sont déployées et lancées dans l'environnement. Lors de l'étape d'instanciation, tous les attributs constant des entités doivent être définis,

à l'inverse des informations de contexte externes et applicatifs qui sont initialement inconnues.

Instanciation statique La figure 5.4 donne un extrait des instances de ces classes, qui sont les entités utilisées dans notre exemple de gestion de réunions. En pratique, ces entités peuvent être définies et disposées graphiquement dans une représentation 2D de l'espace professionnel, à l'aide de l'éditeur visuel DiaSim-Wizard [BJC09] créé par les membres de notre équipe. Cet éditeur dédié à la définition d'environnements ubiquitaires est capable de charger une taxonomie Pantagruel, représentant les classes et les entités sous forme d'icônes.

Chaque entité est nommée et possède une référence à sa classe d'entités. Par exemple, nous avons créé l'instance meetingPhone de la classe **IPPhone**. Cette entité correspond au téléphone de la salle de réunion (MEETINGROOM). Elle joue un rôle spécifique dans notre scénario de gestion de réunions. Par conséquent, elle doit être créée avant l'exécution d'une application traitant ce scénario, afin de permettre au programmeur de l'utiliser lors de sa définition d'une logique d'orchestration.

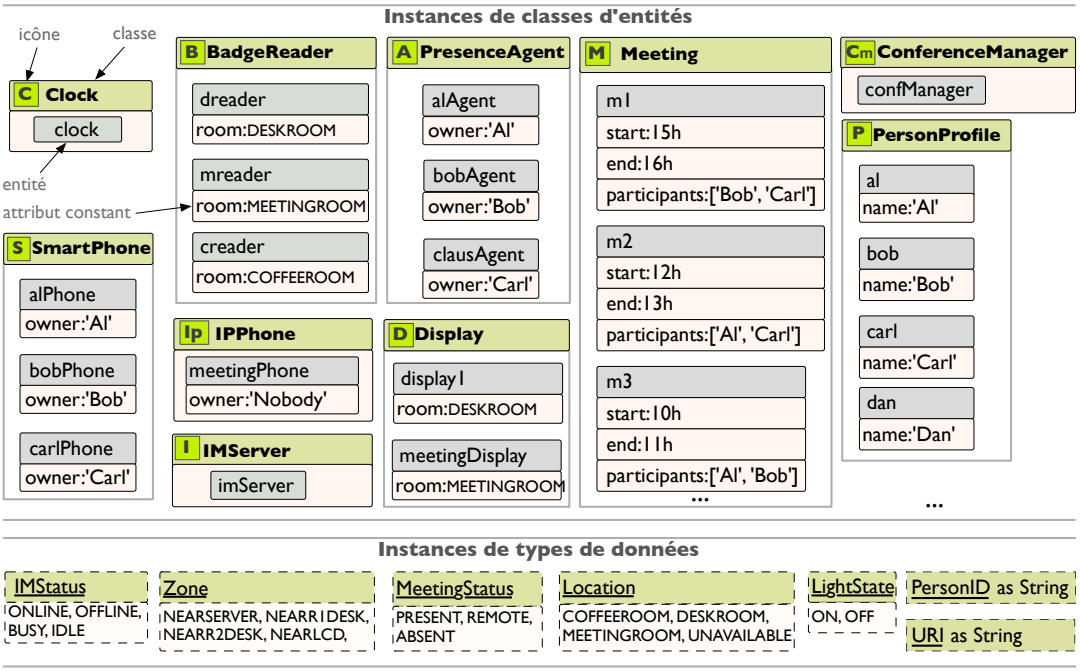

FIG. 5.4: Un environnement concret pour la gestion de réunions

Instanciation dynamique La figure 5.4 comprend des exemples d'événements de réunion pré-enregistrés dans l'agenda partagé de l'entreprise. Les événements de réunion sont définis par des informations complexes que nous modélisons par des entités de la classe **Meeting**. Leurs attributs sont constant. Ces entités peuvent être créées dynamiquement dans un environnement, c'est-à-dire lors de l'exécution des logiques d'orchestration. Pour permettre à Pantagruel de traiter ces entités, nous tirons profit d'un mécanisme de découverte d'entités mis en oeuvre par la

plate-forme vers laquelle Pantagruel est compilé. Nous présentons rapidement cette plate-forme dans la section 5.7.

Types de données concrets Pantagruel permet de définir des types de données simples, instanciant les déclarations de types de la taxonomie. Ces types désignent soit une énumération, soit une référence à un type primitif (chaîne de caractère, entier, booléen). Par exemple, l'énumération Location liste l'ensemble des pièces de l'espace professionnel de notre exemple. Les références nommées permettent d'expliciter des associations entre des attributs de type primitif. Par exemple, l'identifiant d'une personne, spécifié par le type PersonID, permet de s'assurer que les valeurs détectées par les instances de **BadgeReader** appartiennent à l'ensemble des personnes possédant un profil (modélisé par la classe **PersonProfile**).

5.4 Un langage visuel d'orchestration

Des applications orchestrant ces entités peuvent ensuite être définies en utilisant le langage d'orchestration de Pantagruel. Pour cela, ce langage est paramétré par une taxonomie et un environnement conforme à cette taxonomie. Il permet de développer des règles d'orchestration en utilisant les interfaces fournies par les classes de la taxonomie. Nous présentons ici les concepts visuels du langage permettant d'utiliser les abstractions de la taxonomie.

5.4.1 Présentation des concepts visuels

Dans Pantagruel, le paradigme SCA est visuellement représenté sous la forme d'un tableau divisé en trois colonnes : les capteurs *Sensors*, les contrôleurs *Controllers* et les actionneurs *Actuators*, comme illustré dans les figures 5.5 (scénario S1), 5.6 (scénario S2) et 5.7 (scénario S3). Ces trois scénarios présentent des règles d'orchestration pour notre exemple de gestion de réunions de la section 5.1. Le scénario S1 regroupe les règles destinées à la transmission d'une visio-conférence aux participants selon leur profil. Le scénario S2 regroupe les règles permettant de démarrer ou d'arrêter la transmission d'une visio-conférence en fonction des heures à laquelle elle est prévue. Enfin, le scénario S3 regroupe les règles servant à configurer le profil des participants selon leur localisation.

Pour programmer une application, le développeur commence par placer dans le tableau les entités pertinentes pour les scénarios. Par exemple, pour le scénario S1, ces entités sont (1) les instances de **Meeting**, (2) les instances de **PersonProfile**, (3) les instances de **SmartPhone** et (4) l'entité **meetingPhone**. Ensuite, il définit des conditions sur des informations de contexte dans la colonne des capteurs, les combine dans la colonne des contrôleurs, et déclenche des actions dans la colonne des actionneurs. Pour faciliter la lecture, les règles sont numérotées dans la colonne des contrôleurs (par exemple R1.1). Une caractéristique clé de notre approche est de guider le développement d'une logique d'orchestration selon une taxonomie et un environnement. Pour cela, l'éditeur visuel du langage fournit au développeur des menus contextuels paramétrés par les entités qu'il a placées dans le tableau.

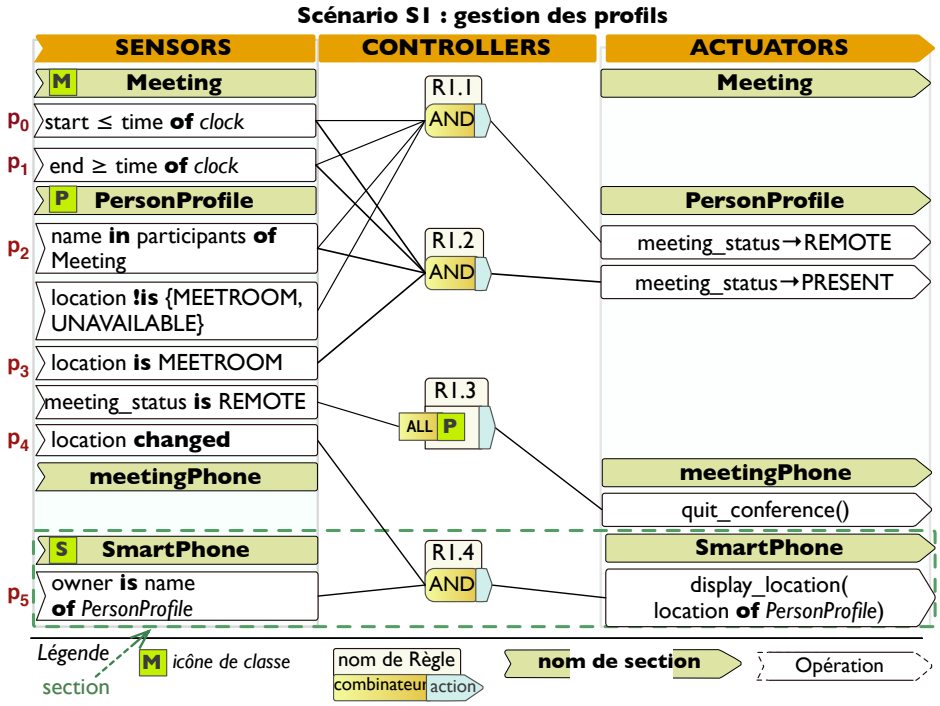

FIG. 5.5: (S1) Gestion de réunions - configuration et utilisation des profils

5.4.1.1 Sections comme éléments structurants d'une règle

Le paradigme de développement orienté entités de Pantagruel est symbolisé par une structuration visuelle des règles d'orchestration en fonction des entités concernées par ces règles. Précisément, le tableau est divisé en *sections* horizontales, chacune représentant une entité impliquée dans l'application d'orchestration. Ainsi la troisième section de la figure 5.5 est définie pour l'entité **meetingPhone** appartenant à la classe **IPPhone**. Une section définit une portée visuelle pour les opérations (capteurs ou actionneurs) utilisées sur l'entité correspondante. Par exemple, l'attribut session_open, défini sur la classe **IPPhone**, peut être manipulé dans la section **meetingPhone**. Une section est également requise pour déclencher une action sur cette même entité, par exemple, quitter la visio-conférence lorsqu'elle est terminée. Notons que le téléphone **meetingPhone** est démarré par l'action de la règle R2.3 de la figure 5.6.

Néanmoins, lorsqu'un attribut externe à une section est requis dans une opération sur l'entité de cette section, celui-ci doit être accédé explicitement comme c'est le cas de l'attribut time invoqué sur l'entité **clock**. Pour cela la notation "*attribute* **of** *entity*" est utilisée.

5.4.1.2 Manipuler des classes d'entités

Lorsqu'un grand nombre d'entités est déployé dans un environnement, des règles d'orchestration doivent pouvoir être définies et exécutées sur des ensembles d'entités. Opérer sur des entités spécifiques peut alors rapidement engendrer une explosion du nombre de règles nécessaires pour

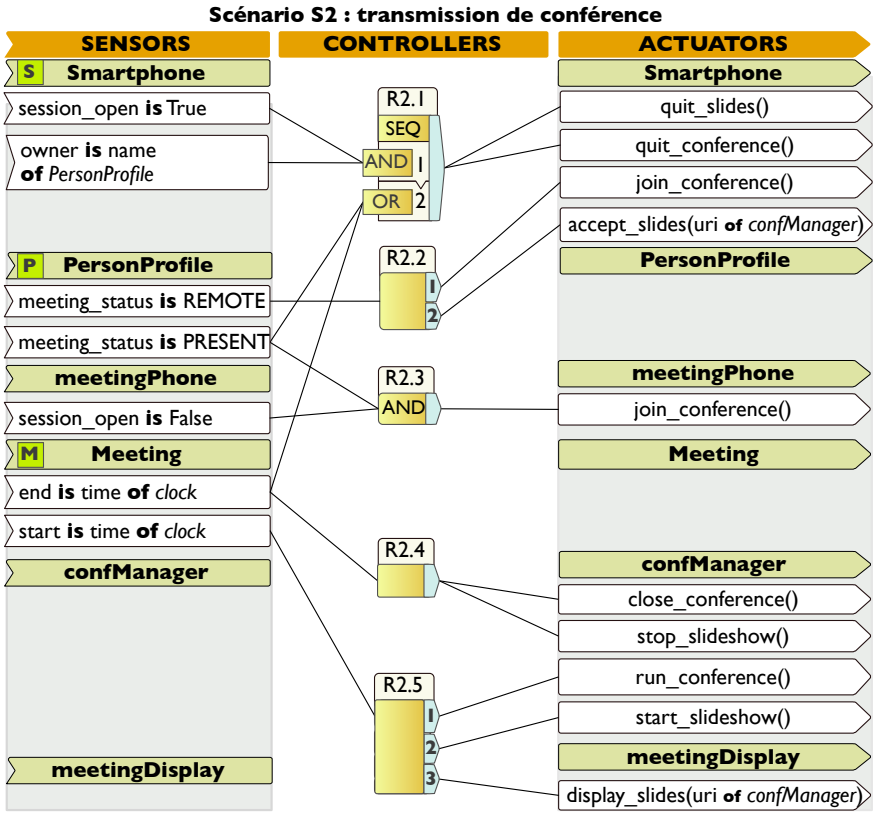

FIG. 5.6: (S2) Gestion de réunions - lancement d'une visio-conférence

définir une application. De plus, si des entités sont créées dynamiquement, il n'est plus possible d'y faire référence directement et elles doivent pouvoir être découvertes lorsqu'elles sont utilisées dans les règles. Il faut donc que les règles puissent être définies sur des entités appartenant à une classe donnée. Enfin, il peut être nécessaire d'écrire des règles supplémentaires pour raffiner (au sens, rendre plus précis) le traitement de ces ensembles d'entités.

Pour cela, une section peut faire référence à toutes les instances d'une classe donnée. Cette section porte alors le nom d'une classe (commençant par une lettre majuscule). Considérons par exemple les deux premières sections du scénario S1 de la figure 5.5 (page 55). Ces deux sections orchestrent la classe **Meeting** et la classe **PersonProfile**. La "portée visuelle" s'applique comme pour les entités spécifiques : l'attribut meeting_status, défini sur la classe **PersonProfile**, peut être manipulé dans la section **PersonProfile**.

Lorsqu'une règle d'orchestration inclut une opération venant d'une section de classe, elle est exécutée sur l'ensemble des instances de la classe correspondante. Par exemple, la figure 5.5 inclut la section **PersonProfile**. L'une des conditions définies dans cette section détermine si une personne est localisée dans la salle de réunion (c'est-à-dire "location **is** MEETINGROOM"). Elle permet à une règle de s'appliquer sur toutes les personnes présentes dans la salle de réunion.

Nous voyons à présent en détail la définition des conditions et des actions.

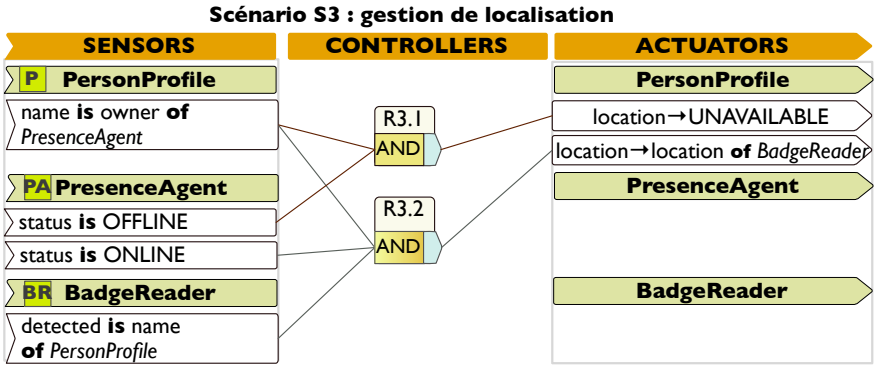

FIG. 5.7: (S3) Gestion de réunions - gestion de la localisation

5.4.2 Définir des conditions de contexte

Les capteurs sont des conditions définies sur des informations de contexte, qu'elle que soit leur nature (constantes, externes, applicatives). Pantagruel fournit des notations et des opérateurs permettant d'exprimer des conditions variées. Spécifiquement, les valeurs d'informations de contexte peuvent être testées à l'aide des opérateurs de comparaison classiques (<, >, **is** pour = et **!is** pour ≠) ou des opérateurs sur des ensembles (**has** et **in**, préfixés par **!** pour leur négation). Une construction spécifique nommée **changed** permet un traitement particulier sur les informations de contexte externes et applicatives. Cette construction devient vraie lorsque la valeur d'un attribut qui s'y rapporte change. Elle permet de définir une logique de réaction indépendante des détails d'implémentation relatifs au changement de contexte.

Lorsqu'une condition porte sur une classe d'entités, elle agit comme un filtre sur les entités appartenant à cette classe. Par exemple, la condition "location **is** MEETINGROOM" collecte l'ensemble des personnes présentes dans la salle de réunion. Si on souhaite imposer une condition à toutes les entités d'une classe, on utilise l'opérateur **ALL**, préfixé par une classe d'entités. Ce mot clé, utilisé dans la colonne des contrôleurs, est illustré dans la règle R1.3 du scénario S1 (figure 5.5) : l'action quit_conference de l'entité **meetingPhone** est exécutée uniquement quand la salle de réunion est vide, c'est-à-dire quand toutes les personnes participantes l'ont quittée.

Un combinateur de flux

On peut avoir besoin de définir des règles qui manipulent des sous-ensembles d'entités collectées. Dans ce cas, il est nécessaire de raffiner encore l'opération de filtrage des instances, par exemple, pour déclencher des actions particulières selon les entités collectées. Pour ce faire, les conditions de filtrage peuvent être combinées. Par exemple, lorsqu'une personne est présente, nous souhaitons également savoir si elle participe à la réunion qui est sur le point d'avoir lieu. Pour cela, nous définissons une autre condition sur la section **PersonProfile** du scénario S1 (figure 5.5) qui sélectionne le sous-ensemble des participants parmi les personnes qui sont dans la salle de réunion. Précisément, un profil de personne est collecté si (1) la personne est dans la salle de réunion (condition p_3 sur **PersonProfile**), (2) son nom appartient à la liste

des participants d'une réunion (condition p_2 sur **PersonProfile**), (3) la réunion en question est en cours (conditions p_0 et p_1 sur **Meeting**). Ce filtrage progressif est engendré par l'opérateur **AND** de la colonne des contrôleurs combinant ces quatre conditions. Les conditions de contexte peuvent également être groupées par les opérateurs **OR**, **SEQ** et **ALL** (**NONE** pour "aucun"). Ainsi, lorsque les conditions portent sur des classes, les contrôleurs agissent comme des combinateurs de flux. En particulier, l'opérateur **SEQ** permet d'expliciter ou d'imposer un ordre d'évaluation des conditions. L'utilité de cet opérateur est illustrée dans le contrôleur de la règle R4.1 (figure 5.8). Il permet de collecter d'abord les personnes dont le meeting_status est égal à REMOTE, puis de tester si *toutes ces* personnes possèdent un smartphone.

Sous sa forme actuelle, la représentation visuelle des contrôleurs souffre d'une limitation : un contrôleur ne permet de combiner qu'un nombre limité d'opérateurs booléens. Un travail futur sur l'éditeur visuel sera d'améliorer l'élément visuel utilisé pour le contrôleur.

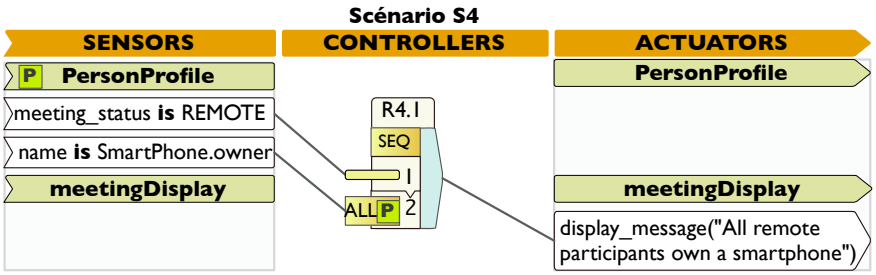

FIG. 5.8: Le combinateur de conditions **SEQ**

5.4.3 Définir des actions

Note préliminaire : dans les explications qui suivent, les règles dont l'identifiant est de la forme R1.x *sont contenues dans la figure 5.5 (page 55), celles de la forme* R2.x *sont contenues dans la figure 5.6 (page 56), et celles de la forme* R3.x *sont contenues dans la figure 5.7 (page 57).*

Lorsque l'évaluation d'un contrôleur retourne la valeur booléenne `vrai`, une action est exécutée, comme dans la règle R1.1, ou plusieurs actions. Dans ce cas, les actions sont soit exécutées dans un ordre indéterminé (règle R2.1), soit séquentiellement (règle R2.5). Ces actions correspondent à des invocations de méthodes (règle R1.1) ou à des affectations d'attributs (règle R3.2). Les invocations de méthodes sont des opérations effectuées sur des entités depuis Pantagruel et doivent être conformes à leur signature déclarée dans la taxonomie. Lorsque la méthode d'une instance est invoquée, elle peut avoir un effet sur le contexte applicatif, c'est-à-dire mettre à jour un ou plusieurs attributs write de l'instance suivant sa déclaration. Par exemple, la méthode join_conference invoquée sur l'entité **meetingPhone** peut mettre l'attribut session_open à la valeur booléenne `vrai` (règles R2.3), indiquant que ce téléphone a rejoint une session audio-visuelle. Les affectations permettent de mettre à jour l'état d'une entité, en modifiant explicitement la valeur d'un de ses attributs, comme dans la section **PersonProfile** du scénario S1 (figure 5.5).

Finalement, lorsqu'il souhaite exécuter des actions dans un ordre spécifique, le développeur définit dans le contrôleur l'ordre d'exécution souhaité. Les séquences d'actions sont utiles lorsqu'on souhaite expliciter l'échange de données entre deux actions. Un exemple de telles échanges est celui d'une photographie prise par une caméra puis envoyée à un serveur de mails, ou, comme dans les règles R2.2 et R2.5, l'échange d'une adresse `URI` pour transmettre la présentation d'une visio-conférence à un navigateur web. Ainsi, l'effet produit par une action affecte la suivante sans nécessiter une règle intermédiaire.

Dépendances de règles La mise à jour des attributs write est une mise en oeuvre du contexte applicatif. L'utilisation de ces attributs dans la partie Capteur d'une application Pantagruel permet d'expliciter une relation causale entre les actions des règles. Par exemple, les entités **SmartPhone** ne joignent une session audio que si leur propriétaire n'est pas dans la salle de réunion (règle R2.2 de la figure 5.6, page 56). Cette contrainte est assurée par les règles R1.1 et R1.2 (figure 5.5, page 55), qui ont antérieurement modifié le statut meeting_status des entités de **PersonProfile**, filtrant les personnes selon leur participation à la réunion en cours.

5.5 Modèle d'exécution

Pour comprendre comment les règles d'orchestration définies précédemment sont évaluées, nous introduisons à présent les concepts qui sous-tendent le modèle d'exécution du langage d'orchestration. Nous présentons ensuite de manière informelle leur mise en œuvre dans le modèle d'exécution de Pantagruel. Cette présentation informelle a pour but de faciliter la compréhension de la sémantique de Pantagruel, présentée dans la section 5.6.

5.5.1 Concepts clé

Pantagruel est un langage de programmation réactif pour décrire la logique d'orchestration d'entités communicantes. Les systèmes réactifs sont des programmes qui interagissent en permanence avec leur environnement, en *réagissant* à ses changements [HP85]. Nous proposons un modèle réactif simplifié par les concepts que nous présentons ici. Grâce à ces concepts, il est possible de raisonner sur une logique d'orchestration exprimée en Pantagruel sans se préoccuper des contraintes d'exécution qui dépassent notre propos.

Modèle centré sur le contexte La logique d'orchestration de Pantagruel repose sur les informations de contexte, qui sont également un élément central du modèle réactif de Pantagruel. Ainsi, nous nous concentrons sur les données disponibles à un temps donné et sur les entités par lesquelles ces données sont fournies, plutôt que sur la manière dont ces données sont acquises. Par conséquent, la réactivité d'un programme Pantagruel est conditionnée par les changements observés dans le contexte d'un environnement, qui représente aussi l'état du programme à l'exécution.

Temps discret et actions atomiques Le langage Pantagruel fait abstraction de préoccupations temporelles réelles : son objectif principal est d'exprimer les relations entre un contexte et le

comportement des entités, facilitant la vérification de propriétés concernant la logique d'orchestration, comme cela est illustré dans le chapitre 7. A cette fin, le temps d'exécution est discret : chaque "top" d'horloge t correspond à une étape d'évaluation, où toutes les règles sont évaluées avant le prochain top. Nous simplifions encore le modèle en déclarant que chaque méthode exécutée se termine immédiatement, correspondant au principe d'exécution atomique. Ces deux abstractions nous permettent de raisonner sur la logique d'orchestration indépendamment de contraintes de temps ou de l'implémentation des fonctionnalités des entités.

Mode parallèle Il y a deux modes d'exécution de base pour les systèmes réactifs [Rob00]. Dans une approche à base de règles, le mode d'itération séquentiel présuppose que l'exécution des règles est ordonnée. Dans ce mode, la première règle est évaluée, puis la seconde, selon le résultat renvoyé par la première règle (par exemple, l'état du système mis à jour), et ainsi de suite. Au contraire, le mode d'itération parallèle, que nous choisissons, suppose que toutes les règles dont les conditions sont vérifiées à un temps discret t sont exécutées *simultanément* (c'est-à-dire avec la même "photographie" de l'état au temps t). Nous pensons que le mode parallèle permet de donner à l'utilisateur final une meilleure intuition de l'exécution : un avantage de ce mode est que l'utilisateur crée des règles sans avoir à se soucier des dépendances entre ces règles. Il peut ainsi raisonner sur le même état global pour chaque temps discret, ce qui est facilité par la disposition visuelle de Pantagruel où l'ensemble des capteurs des règles d'un programme est représenté dans une même colonne.

Non-interférence La non-interférence désigne le fait que deux actions exécutées en même temps n'ont pas d'effet concurrent, c'est-à-dire que leur effet est le même que si ces actions étaient exécutées dans une séquence [Bou89]. En Pantagruel, cette propriété implique que lorsque des règles sont exécutées en même temps, chacune a sa propre copie de l'état courant du système, et leurs effets sont disjoints. A la fin de l'exécution, les effets sont joints, générant un nouvel état du programme en combinant ces effets. La non-interférence rend explicite le fait qu'une règle ne peut annuler l'effet de l'autre. Cette opération nécessite soit de définir un opérateur de combinaison approprié, soit de s'assurer que deux règles s'exécutant en même temps n'ont pas d'effets contradictoires. Nous choisissons la deuxième solution, qui est présentée dans le chapitre 7, en utilisant les déclarations des effets sur les méthodes. Cette vérification permet notamment de garantir le comportement déterministe des programmes, augmentant le degré de confiance de l'expert-métier.

Comportement basé sur les filtres En Pantagruel, le développeur peut exprimer des règles sur des entités spécifiques ou sur des ensembles d'entités en utilisant le nom de leur classe. L'abstraction *classe* est utilisée pour filtrer des entités sur lesquelles un comportement doit être défini. Elle permet également d'exprimer des conditions basées sur des opérateurs de quantification (par exemple, **ALL**) sur des ensembles d'entités. Nous conservons cette abstraction pour décrire le comportement des programmes car elle permet à la logique d'orchestration de fonctionner avec des instances de classes créées pendant l'exécution d'un programme.

5.5.2 Principe d'exécution général

Les concepts clés que nous venons de décrire sont mis en œuvre dans le processus d'exécution de Pantagruel décrit dans l'algorithme 1. Ce processus élémentaire utilise un ensemble de règles R, un environnement e_c composé d'une taxonomie et d'instances de classes, et des mémoires σ_t, $\sigma_{t\text{-}1}$ et σ_{t+1} (lignes 1 et 2 de l'algorithme 1). L'exécution d'un programme Pantagruel se déroule de la façon suivante : les règles sont évaluées selon la mémoire courante qui modélise le *contexte* de l'environnement concret et produisent une nouvelle mémoire, reflétant les actions exécutées. Plus précisément, la mémoire contient l'ensemble des attributs volatile, write et constant de toutes les entités. Désormais nous appellons ces attributs d'entités des *variables*.

Algorithme 1 L'exécution de Pantagruel

1: $R = \{r_1, \ldots, r_n\}$: ensemble de règles, e_c : taxonomie et instances, V : variables volatile ;
2: $\sigma_t = \sigma_{t\text{-}1} = \sigma_{t+1} = s_0$ où s_0 est la mémoire initiale, contenant les variables constantes.
3: **tant que vrai faire**
4: $\quad \sigma_t \leftarrow update\_volatile(V, \sigma_t)$
5: $\quad$ **pour chaque** $r_i \in R$ **faire** $\sigma_{t+1} \leftarrow eval\_rule(r_i, e_c, \sigma_{t\text{-}1}, \sigma_t, \sigma_{t+1})$ **fin pour**
6: $\quad \sigma_{t\text{-}1} \leftarrow copy(\sigma_t)$
7: $\quad \sigma_t \leftarrow copy(\sigma_{t+1})$
8: **fin tant que**

Le comportement réactif est modélisé par une boucle infinie (ligne 3) sur l'évaluation des règles d'orchestration. Chaque itération de la boucle correspond à un top d'horloge (ou temps discret) t, modélisant la nature *discrète* de Pantagruel. Une itération utilise trois "photographies" ou versions de la mémoire : $\sigma_{t\text{-}1}$, σ_t, et σ_{t+1} aux temps t-1, t et t+1. Nous expliquons l'usage de ces mémoires dans les paragraphes suivants.

Au début de chaque itération, le contexte externe modélisé par les attributs volatile est capturé et mis à jour dans la mémoire courante σ_t (ligne 4). Cette photographie de la mémoire modélise le *mode parallèle*, supposant une évaluation simultanée des règles à chaque itération (ligne 5).

Pour observer un changement de contexte survenant dans l'environnement, il faut pouvoir suivre l'évolution des variables modélisant les éléments de contexte. Pour cela nous utilisons les deux mémoires $\sigma_{t\text{-}1}$ et σ_t. Ces mémoires permettent de savoir si les valeurs des variables ont changé entre le temps t-1 et le temps t. Ainsi, lorsqu'une règle est active, c'est-à-dire lorsque ses conditions sont vérifiées avec la mémoire σ_t, et que celles-ci sont insatisfaites avec la mémoire $\sigma_{t\text{-}1}$, les actions de cette règle sont exécutées. Plus spécifiquement, la fonction d'évaluation d'une règle *filtre* les entités satisfaisant ses conditions, et exécute alors les actions sur ces entités sélectionnées. Lorsque les actions sont appelées au temps t, leurs effets sont immédiatement ajoutés à la mémoire tampon σ_{t+1}, ce qui correspond au principe d'*atomicité*.

Afin d'appliquer le principe de *non-interférence*, nous adoptons la stratégie de double-mémoire $\sigma_{t+1} - \sigma_t$: chaque règle est évaluée et comprise indépendamment des autres à l'aide de leur propre copie de la mémoire σ_t. Leurs effets sont ensuite accumulés dans la mémoire σ_{t+1}.

Lorsque l'itération courante est complétée, la mémoire courante σ_t devient la mémoire précédente $\sigma_{t\text{-}1}$ de l'itération suivante, et la mémoire tampon σ_{t+1} devient la mémoire courante σ_t de l'itération suivante (lignes 6 et 7).

Nous détaillons le processus d'évaluation d'une règle (c'est-à-dire la fonction *eval_rule*) dans la section suivante ainsi que le mécanisme de filtrage qui est un aspect important de notre langage.

5.5.3 Sémantique informelle des règles

Une règle $r = (P, A)$ est composée d'un ensemble de conditions P, que nous appelons maintenant prédicats, muni d'un opérateur logique (un contrôleur) et d'un ensemble d'actions A. Pour rappel, les actions sont des invocations de méthodes ou des affectations. Nous considérons uniquement les méthodes ici, car le processus s'applique de manière similaire aux affectations. Nous limitons ici la combinaison des prédicats à un contrôleur avec un seul opérateur de type **AND** ou **OR**, suffisant pour donner une intuition de l'évaluation d'une règle. Nous détaillons la sémantique complète des prédicats dans la section 5.6.

- $P = (c, \{p_1, ..., p_n\})$ est un ensemble de prédicats et c un opérateur logique combinant les p_i. L'opérateur logique est `OR` ou `AND`. Un prédicat p_i est une comparaison de deux expressions (variables ou constantes) ou un prédicat `changed`.
- $A = (q, \{a1, ..., an\})$ est un ensemble d'actions, où q indique si leur exécution dans la portée de la règle concernée est séquentielle ou non.

L'évaluation d'une règle par la fonction *eval_rule* utilisée dans l'Algorithme 1 se décompose en deux fonctions, une pour évaluer les prédicats P, l'autre pour déclencher les actions A.

5.5.3.1 Evaluation des prédicats

Nous avons vu dans la section 5.4.1 qu'un prédicat peut être une condition sur une entité spécifique, ou agir comme un filtre pour sélectionner des entités appartenant à une classe donnée. Examinons l'interprétation des prédicats selon ses deux usages.

Les prédicats : des conditions booléennes L'évaluation des prédicats est conditionnée par le changement de contexte, modélisé par les mémoires $\sigma_{t\text{-}1}$ et σ_t. Lorsqu'une règle est définie sur des entités spécifiques, son évaluation est élémentaire, retournant une valeur booléenne. La fonction *context* évaluant ces prédicats est définie par l'algorithme 2. L'évaluation d'un prédicat `x changed` est légèrement différente, nécessitant de comparer la variable x en utilisant les deux mémoires dans une même fonction (ce prédicat équivaut à la condition `x =` *valeur précédente de* `x`). Une sémantique intégrant les deux types de prédicats est présentée dans la section 5.6.

Algorithme 2 la fonction $context(P, \sigma_t, \sigma_{t\text{-}1}) : Booleen$ (note : ← signifie affectation)

$P = (c, P_c)$, avec $P_c = \{p_1, ..., p_n\}$
si $c =$ `AND` **alors** $b \leftarrow \forall p_i \in P_c, pred(p_i, \sigma_t) =$ `vrai` et $\exists p_a \in P_c, pred(p_a, \sigma_{t\text{-}1}) =$ `faux`
si $c =$ `OR` **alors** $b \leftarrow \exists p_a \in P_c, pred(p_a, \sigma_t) =$ `vrai` et $\forall p_i \in P_c, pred(p_i, \sigma_{t\text{-}1}) =$ `faux`
retourner b
où $pred(p, s)$ retourne `vrai` si p est `vrai` au regard de la mémoire s, `faux` sinon

Lorsqu'un prédicat est défini comme un filtre sur une classe, la fonction d'évaluation doit être légèrement adaptée, car elle ne peut plus simplement renvoyer une valeur booléenne.

Les prédicats : des filtres sur les classes L'évaluation des prédicats agissant comme des filtres (appelés *prédicats-filtres*) nécessite d'introduire les définitions suivantes :

- E_{C_i} est l'ensemble des entités appartenant à la classe C_i.
- $Tuple = (E_{C_1} \times \ldots \times E_{C_n})$ est l'ensemble des n-uplets d'entités ou *tuples-entités*, où, dans un tuple-entité donné $t \in Tuple$, chaque élément est une entité appartenant à une classe unique. Plus précisément, un tuple-entité correspond à une combinaison particulière d'entités qui vérifie un prédicat-filtre donné après en avoir remplacé chaque classe par une entité de ce tuple-entité conformément à la classe qu'elle instancie.
- $TupleSet = \mathcal{P}(Tuple)$ est l'ensemble des parties de $Tuple$, définissant toutes les combinaisons possibles d'ensembles de tuples-entités. Une partie $T \in TupleSet$ est un paramètre d'entrée de l'opération de filtrage.

Pour illustrer ces définitions, examinons la règle R1.4 de la figure 5.5 (page 55), qui fait intervenir deux classes. Les prédicats "location **changed**" et "owner **is** name **of** *PersonProfile*", notés p_4 et p_5 dans la figure, s'appliquent respectivement sur les sections **PersonProfile** et **SmartPhone**. Pour évaluer ces deux prédicats, nous construisons un ensemble initial T_0 de tuples-entités, chacun étant une combinaison unique d'une entité de **PersonProfile** et d'une entité de **SmartPhone**. On notera $T_0 \in \mathcal{P}(E_{\text{PersonProfile}} \times E_{\text{SmartPhone}}) = \{$(al, alPhone), (bob, alPhone), ..., (carl, bobPhone), (carl, carlPhone)$\}$[1].

La fonction *context* doit à présent retourner, au lieu d'une valeur booléenne, l'ensemble des tuples-entités qui satisfont les prédicats de la règle (voir l'algorithme 3).

Algorithme 3 la fonction $context(T_0, P, \sigma_t, \sigma_{t\text{-}1}) : TupleSet$

$P = (c, P_c)$, avec $P_c = \{p_1, ..., p_n\}$, $T_0 = \{t_1, ..., t_n\}$ et $T \leftarrow \{\}$
si $c =$ `AND` **alors**
  pour chaque $t_k \in T_0$,
    si $\forall p_i \in P, pred(t_k, p_i, \sigma_t) =$ `vrai` et $\exists p_a \in P, pred(t_k, p_a, \sigma_{t\text{-}1}) =$ `faux`, **alors** $T \leftarrow \{t_k\} \cup T$
(... adaptation similaire pour l'évaluation de* `OR` *... *)*
retourner T
où $pred(t, p, s)$ retourne `vrai` si p est `vrai` au regard de la mémoire s et en remplaçant chaque classe dans p par l'entité correspondante du tuple-entité t.

Nous enrichissons pour cela la fonction *context* précédente avec un paramètre T_0 de type *TupleSet*, et la fonction *pred* avec un paramètre t_k de type *Tuple*, qui est collecté s'il satisfait les prédicats P (opération $T \leftarrow \{t_k\} \cup T$). La fonction *context* retourne finalement l'ensemble des tuples-entités collectés par ce processus.

Illustrons l'application de cette fonction pour les prédicats de la règle R1.4 de la figure 5.5, en utilisant T_0 précédemment défini comme *TupleSet* initial. Supposons qu'entre deux itérations t-1 et t de la boucle réactive, les personnes Al et Carl se sont déplacées de la pièce COFFEEROOM à la salle MEETINGROOM : l'attribut location de leurs profils respectifs a changé (satisfaisant alors le prédicat p_4), tandis que Bob est resté dans la même pièce au même moment (ou n'a pas passé son badge). D'après l'environnement concret de la figure 5.4 (page 53), seuls les profils

[1] En pratique, T_0 est construit avec toutes les classes impliquées dans une taxonomie Pantagruel. Nous réduisons sa taille par souci de clarté.

al et carl satisfont p_5 avec les instances de **SmartPhone**, alphone et carlphone respectivement. L'évaluation des prédicats p_4 et p_5 produit donc la sélection de tuples-entités $T_1 = \{$(al, alphone), (carl, carlphone)$\}$.

5.5.3.2 Evaluation des actions

Une fois que les prédicats sont évalués, les actions d'une règle sont exécutées en conséquence. Si la règle opère uniquement sur des entités spécifiques, l'évaluation des actions est élémentaire (on exécute l'action sur ces entités spécifiques). Si en revanche, les prédicats ou les actions sont définis sur des classes, les actions doivent être évaluées en fonction des tuples-entités retournés par l'évaluation des prédicats. Dans ce dernier cas, une action appelée sur une classe s'applique alors aux entités de cette classe qui sont collectées par l'évaluation des prédicats.

Par suite, nous définissons une fonction *action* pour évaluer une action a de l'ensemble A des actions d'une règle (algorithme 4). Elle requiert un paramètre T_1 de type *TupleSet*, et la mémoire courante σ_t pour accéder aux valeurs des attributs éventuellement utilisés dans l'action (comme paramètres d'une méthode ou valeur d'affectation). Après avoir recueilli l'ensemble des classes dont l'action a dépend, la fonction *action* exécute a sur les entités de ces classes. Ces entités sont extraites de T_1 via un mécanisme de projection à la manière du langage de bases de données SQL (fonction *proj* dans l'algorithme 4).

Algorithme 4 La fonction $action(T_1, a, \sigma_t)$: Store

$C_a \leftarrow$ l'ensemble des classes dont a dépend
si $C_a \neq \emptyset$ **alors**
  pour chaque $u \in proj(C_a, T_1)$ **faire**
    $\sigma_{t+1} \leftarrow action_u(a_u, \sigma_t)$
  où $proj(X, Y)$ projette Y dans X
    Exemple : $proj(\{A, B\}, \{(d_A, d_B, d_C), (e_A, e_B, e_C)\}) = \{(d_A, d_B), (e_A, e_B)\}$
  et a_u est a, dont les noms de classes A, B sont substitués à leur entité correspondante dans u
sinon
  $\sigma_{t+1} \leftarrow action_u(a, \sigma_t)$
où $action_u$ évalue une action utilisant uniquement des entités spécifiques

Reprenons en guise d'exemple l'évaluation des actions avec la règle R1.4 qui a servi d'exemple dans le processus d'évaluation des prédicats. Dans cette règle, la méthode display_location dépend de deux classes. Spécifiquement, elle doit être appelée sur chaque entité de **SmartPhone** dont le propriétaire (attribut owner) – collecté par le prédicat "name **is** owner **of** *SmartPhone*" défini sur la classe **PersonProfile** – a changé de pièce. Dans cet exemple, les actions doivent s'appliquer sur chaque tuple-entité (personprofile, smartphone)$\in T_1$, où T_1 résulte de l'évaluation des prédicats de la règle R1.4 (*cf.* la section précédente).

La structure tuple-entité préserve la relation entre les entités impliquées dans une règle, comme dans cet exemple, la relation d'appartenance entre un téléphone et une personne particulière. Ainsi, en considérant le tuple T_1 établi précédemment, l'action display_location affiche la nouvelle localisation d'Al sur *son* téléphone **alPhone**, et de même pour Carl.

Notons enfin que lorsque les actions d'une règle sont séquentielles, l'exécution d'une action de la séquence ne dépend plus de σ_t mais de la mémoire tampon σ_{t+1} mise à jour par l'action

précédente.

5.6 Sémantique formelle

Après avoir rapidement présenté les notations utilisées pour définir notre sémantique, nous donnons la sémantique des deux sous-couches langages de Pantagruel. La sémantique du langage de taxonomie est complètement statique, celle du langage d'orchestration est essentiellement dynamique. La vérification statique des types dans les programmes d'orchestration étant élémentaire, nous ne la présentons pas (par ailleurs, l'utilisation de l'éditeur visuel garantit un typage correct des règles).

5.6.1 Notions préliminaires

La sémantique de Pantagruel est écrite dans le style dénotationnel. Nous suivons la méthodologie proposée par David Schmidt [Sch86]. En particulier, nous définissons le langage de taxonomie et le langage d'orchestration en suivant trois étapes :

- la définition d'une *syntaxe abstraite* : c'est la représentation textuelle abstraite d'un programme ;
- la définition des *algèbres sémantiques* : ce sont les domaines (d'un point de vue programmeur, les structures de données) et leurs opérations qui serviront à donner la signification d'un programme ;
- la définition de *fonctions de valuation* : ce sont les fonctions connectant la syntaxe abstraite du langage et l'algèbre sémantique, donnant ainsi la signification ou *dénotation* du langage. Elles sont décrites avec les notations du lambda-calcul.

Nous utilisons les conventions de notation suivantes :

$A \times B$	produit des domaines A et B, dont les éléments sont des paires (a, b) où $a \in A$ et $b \in B$
$A + B$	"union disjointe" des domaines A et B, dont les éléments (e, x) sont tels que $x \in A$ si $e = \texttt{A}$ et $x \in B$ si $e = \texttt{B}$; – une opération in$A : A \to A + B$ est telle que in$A(a) = (\texttt{A}, a)$ – une opération is$A : A + B \to \mathbb{B}$ est telle que si $a \in A$, $x = (\texttt{A}, a) \Rightarrow$ is$A(x)$.
$\mathcal{P}(A)$	ensemble des parties de A
$A \to B$	signature d'une fonction, qui à un élément de A associe un élément de B
$(f\ x)$	application d'une fonction f à un élément x
$\lambda x.g\ x$	définition d'une fonction prenant un argument x et dont le corps est $g\ x$
$[v \mapsto x]\rho$	substitution de la variable v à x dans ρ (ou extension de ρ à v associée à x)
$x\downarrow_i, i \in \mathbb{N}$	projection sur le $i^{ème}$ élément de x
$\mathbf{X}[\![\mathrm{X}]\!]$	dénotation du noeud syntaxique abstrait X.
$\perp$	élément *bottom* dénotant la valeur de variables non initialisées.
$f \circ g$	définition d'une composition de fonctions, équivalente à $\lambda x.(f \circ g)(x)$ ou $\lambda x.f(g(x))$

5.6.2 Sémantique statique du langage de taxonomie

Dans cette section, nous formalisons la sous-couche langage de taxonomie de Pantagruel. Pour cela, nous définissons la syntaxe textuelle faisant correspondre chaque élément visuel de la figure 5.3 (page 50) à un élément textuel, ses algèbres sémantiques, et enfin ses fonctions de valuation.

Par souci de concision et pour faciliter la compréhension de la sémantique, nous avons volontairement simplifié la syntaxe du langage de taxonomie, pour en garder uniquement le coeur : une méthode déclare un seul paramètre et un seul effet, et le domaine des valeurs de variables est restreint aux entiers naturels. En pratique, une méthode peut déclarer autant d'effets et de paramètres que nécessaires, et les valeurs peuvent être de type liste, booléen, ou énumération. La sémantique peut s'étendre facilement pour prendre en compte ces précisions.

5.6.2.1 Syntaxe abstraite

La syntaxe textuelle abstraite du langage de taxonomie Pantagruel est donnée dans la figure 5.9. La signification des métavariables est donnée dans la partie haute de la figure (par exemple les métavariables C_i identifient les déclarations de classes). Les règles de grammaire sont données dans la partie basse de la figure.

D ∈ Domain	W ∈ Instantiation	I_m ∈ Method-id
C ∈ Class	Q ∈ Entity	I_i ∈ Impl-id
A ∈ Attribute	Z ∈ Assignment	I_e ∈ Entity-id
M ∈ Method	N ∈ Numeral	I ∈ Identifier = Class-id ∪ Attribute-id ∪
U ∈ Modifier-name	I_c ∈ Class-id	Method-id ∪ Entity-id ∪ Impl-id ∪ Global-id
T ∈ Primitive-type	I_a ∈ Attribute-id	∪ Rule-id

D ::= C	T ::= **int** | **void**
C ::= C_1 ; C_2 | **class** I_{c1} (**extends** I_{c2})$^?$ A M **end**	W ::= Q
A ::= A_1 ; A_2 | U I_a **:** T	Q ::= Q_1 ; Q_2 | I_e **:** I_c [Z]
M ::= M_1 ; M_2 | I_m **(** T$^?$ **)** **→** $I_a$$^?$ {I_i}	Z ::= Z_1 ; Z_2 | I_a **:=** N
U ::= **volatile** | **write** | **constant**	

FIG. 5.9: Syntaxe abstraite du langage de taxonomie

Une taxonomie, ou un domaine D est une collection de définitions de classes. La définition **class** `Ic1 extends Ic2 A M` désigne une classe nommée I_{c1} héritant de la classe C_2. Elle a des attributs A et des méthodes M. Elle hérite de plus des attributs et des méthodes de C_2. Pantagruel utilise un mécanisme d'héritage simple et ne permet pas la surcharge de déclarations de méthodes. Un attribut possède un identifiant I_a, un type `T`, et un modificateur `U` désignant sa nature constante, volatile, ou write. Une méthode possède un identifiant I_m ; elle est définie avec un paramètre optionnel de type `T` et un effet optionnel I_a. Pour rappel, Pantagruel fait abstraction des implémentations de méthodes : ainsi une méthode fournit simplement un identifiant I_i faisant référence à son implémentation définie à l'extérieur de Pantagruel (une implémentation peut éventuellement être stockée dans un environnement spécifique qui leur associe cet identifiant).

Un environnement concret W instancie une taxonomie D : c'est un ensemble de définitions d'entités possédant un identifiant $\texttt{I}_e$, et dont chacune est une instance d'une classe de D (identifiée par $\texttt{I}_c$). Chaque entité est initialisée selon la règle de grammaire Z, avec une valeur pour chacun des attributs constants déclarés dans sa classe et dans les ascendants de sa classe.

La taxonomie et son instanciation doivent respecter quelques règles de typage élémentaires, que nous présentons rapidement ici. Par souci de concision nous ne les détaillerons pas dans les fonctions de valuation du langage.

- Héritage de classe : étant donné une déclaration $\texttt{I}_{c_1}$ `extends` $\texttt{I}_{c_2}$, la classe C_1 est bien formée si la classe C_2 est définie.
- Effet de méthode : étant donnée une déclaration de méthode $\texttt{I}_m$ `(T`$^?$`) ➔` $\texttt{I}_a^?$ $\{\texttt{I}_i\}$, I_a fait toujours référence à un attribut write.
- Instanciation : étant donnée une affectation Z de la forme $\texttt{I}_a$ `:=` N, I_a fait toujours référence à un attribut constant.

5.6.2.2 Algèbres sémantiques

Les domaines nécessaires à l'évaluation de la taxonomie sont définis dans la figure 5.10 pour les domaines des classes et dans la figure 5.11 pour les domaines de la mémoire.

L'évaluation d'une taxonomie et de son instanciation requiert des structures de données pour les attributs et les méthodes composant une classe d'entités. Ces structures utilisent quelques domaines de base (partie I de la figure 5.10), dont *Unit*, qui est un ensemble dont l'élément unique est "()" (et qui sert, entre autres, à définir des structures élémentaires) et *Type*, qui représente le domaine des types. Ce domaine permet d'identifier le type du paramètre d'une méthode, à savoir *Int* si la méthode possède un paramètre (de type "entier naturel"), et *Void* sinon. Nous l'utilisons aussi, par principe, pour identifier le type d'un attribut. Notons enfin que l'union d'un domaine et du domaine *None* (qui est égal à *Unit*) désigne un ensemble d'éléments optionnel.

Domaines des attributs et des méthodes Les domaines des attributs et des méthodes sont définis dans les parties II à V de la figure 5.10.

Un attribut, ou élément du domaine des attributs *Attribute-struct*, est le produit d'un élément du domaine *Modifier*, pour la nature de l'attribut, et d'un élément du domaine *Type*, pour le type de l'attribut. Ces deux domaines correspondent uniquement à des annotations et sont donc définis par le domaine singleton *Unit*. Une méthode du domaine *Method-struct* est le produit de trois éléments : le type de son paramètre t du domaine *Type*, un effet optionnel e du domaine *Effect* et un élément f du domaine *FunImpl* correspondant à l'implémentation de la méthode. Lorsque l'effet existe, il correspond à l'identifiant d'un attribut, du sous-domaine *Attribute-id* (défini dans la figure 5.9, page 66). Une implémentation f du domaine *FunImpl* est conforme à la déclaration de la méthode : elle prend un paramètre réel optionnel (*Param*) et une valeur de retour optionnelle (*EffectReturn*) correspondant à la valeur de l'effet éventuellement produit par l'exécution de la méthode.

Afin d'accéder aux attributs et aux méthodes d'une classe, ceux-ci sont stockés dans des *environnements*, définis par les domaines Env_a et Env_m (parties IV et V de la figure 5.10), et

I. Domaines de base

Domain $Unit$ (domaine singleton, dont l'unique opération est () : $Unit$)
Domain $n \in Nat = \mathbb{N}$
Domain $b \in Bool = \mathbb{B}$
Domain $t \in Type = Int + Void$ **where** $Int = Void = Unit$

II. Attribut

Domain $u \in Modifier = Volatile + Write + Constant$
where $Volatile = Write = Constant = Unit$
Domain $a \in Attribute\text{-}struct = Modifier \times Type$

III. Méthode

Domain $m \in Method\text{-}struct = Type \times Effect \times FunImpl$
where $Effect = (Attribute\text{-}id + None)$
and $FunImpl = (Param \to EffectReturn)$
where $Param = Nat + None$, $EffectReturn = Nat + None$, $None = Unit$

IV. Environnement des attributs

Domain $\rho_a \in Env_a = Attribute\text{-}id \to Attribute\text{-}struct_\perp$ (environnement initial : $\rho_a^0 = \lambda i.\perp$)
Operations
$modifier : Env_a \to Attribute\text{-}id \to Modifier$
$modifier = \lambda\rho_a.\lambda i_p.\rho_a(i_p)\downarrow_1$

V. Environnement des méthodes

Domain $\rho_m \in Env_m = Method\text{-}id \to Method\text{-}struct_\perp$ (environnement initial : $\rho_m^0 = \lambda i.\perp$)

VI. Classes

Domain $c \in Class\text{-}struct = (Class\text{-}id + None) \times Env_a \times Env_m \times \mathcal{P}(Entity\text{-}id)$
where $None = Unit$

VII. Environnement des classes

Domain $\rho_c \in Env_c = Class\text{-}id \to Class\text{-}struct_\perp$ (environnement initial : $\rho_c^0 = \lambda i.\perp$)
Operations
$lookup_c : (Class\text{-}id + Entity\text{-}id) \to Env_c \to Class\text{-}struct$
$lookup_c = \lambda i_z.\lambda\rho_c.$cases i_z of
is$Entity\text{-}id(i_e) \to \rho_c(one\ \{i_c \in Class\text{-}id \mid i_e \in \rho_c(i_c)\downarrow_4\})$
[] is$Class\text{-}id(i_c) \to \rho_c(i_c)$
where *one* retourne l'unique élément de l'ensemble donné en argument.
$update_c : Entity\text{-}id \to Class\text{-}id \to Env_c \to Env_c$
$update_c = \lambda i_e.\lambda i_c.\lambda\rho_c.([i_c \mapsto$ **let** $(i'_c, \rho_a, \rho_m, s) = \rho_c(i_c)$ **in** $(i'_c, \rho_a, \rho_m, (\{i_e\} \cup s))]\rho_c)$
$lookup_m : Method\text{-}id \to (Class\text{-}id + Entity\text{-}id) \to Env_c \to Method\text{-}struct_\perp$
$lookup_m = \lambda i_m.\lambda i_z.\lambda\rho_c.$**let** $(i'_c, \_, \rho_m, \_) = (lookup_c\ i_z\ \rho_c)$ **in**
$\rho_m(i_m) \neq \perp \to \rho_m(i_m)$
[] (cases i'_c of is$None() \to \perp$ [] is$Class\text{-}id(i_h) \to (lookup_m\ i_m\ i_h\ \rho_c)$)

FIG. 5.10: Domaines du langage de taxonomie

associant à leurs identifiants les structures correspondantes des domaines *Attribute-struct* et *Method-struct*. Notons que les environnements sont définis de façon classique par une fonction partielle associant un identifiant à une valeur (ici, une structure d'attribut ou de méthode), et peuvent être initialisés par les opérations ρ_a^0 et ρ_m^0 qui retournent alors des environnements "vides" (c'est-à-dire associant à tout identifiant la valeur non définie $\perp$).

Domaines des classes Nous disposons maintenant des domaines nécessaires pour définir une classe d'entités. Elle correspond à un élément du domaine *Class-struct* (partie VI de la figure 5.10) composé de quatre éléments : l'identifiant optionnel i'_c d'une classe héritée, un environnement d'attributs ρ_a, un environnement de méthodes ρ_m, et un ensemble d'identifiants s du sous-domaine *Class-id* faisant référence aux entités instanciant la classe c.

Finalement, comme pour les attributs et les méthodes, les classes sont stockées dans un environnement du domaine Env_c (partie VII de la figure 5.10), noté ρ_c, qui associe à chaque nom de classe une structure *Class-struct*. Cet environnement est muni de trois opérations facilitant l'accès et la modification des éléments de l'environnement des classes : $lookup_c$ donne accès à une classe, selon l'identifiant d'entité ou de classe donné en paramètre ; $update_c$ ajoute une instance à l'ensemble des entités de la classe donnée en paramètre ; enfin, $lookup_m$ est une fonction récursive qui donne accès à une méthode de la classe ou de l'un de ses ascendants, selon l'identifiant de la méthode et un identifiant de classe ou d'entité.

Mémoire : informations de contexte Les domaines relatifs aux informations de contexte sont définis dans la figure 5.11.

VIII. Variables et valeurs

Domain $i \in Variable = Entity\text{-}id \times Attribute\text{-}id$
Domain $x \in Value = Nat + Undefined$
where $Undefined = Unit$
Operations
$lth,\ gth,\ eq,\ neq : Value \to Value \to Bool$
$eq = \lambda x_1.\lambda x_2$ cases x_1, x_2 of $\text{is}Undefined(), \_ \to false\ [] \ \text{is}Nat(x_1), \text{is}Nat(x_2) \to x_1 = x_2$
(... opérateurs de comparaison classiques omis ...)

IX. Mémoire stockant les informations de contexte

Domain $\sigma \in Store = Variable \to Value$
Operations
$\sigma^0 : Store$
$\sigma^0 = \lambda i.\text{in}Undefined()$
$join : Store \to Store \to Store$
$join = \lambda\sigma_1.\lambda\sigma_2.(\lambda i.\sigma_1(i)\ combine\ \sigma_2(i))$
$join^*$: $\mathcal{P}(Store) \to Store$
$join^* = \lambda S.card(S) < 2 \to one\ S\ [] ((s_1\ join\ s_2)\ ...\ s_{n\text{-}1})\ join\ s_n$, where $s_i \in S$
where *one* retourne l'unique élément de l'ensemble donné en argument.

FIG. 5.11: Domaines pour la structure mémoire Store

Les informations de contexte (partie VIII de la figure 5.11) sont représentées par les attributs des entités, identifiés par une variable de la forme (*nom d'entité, nom d'attribut*). La valeur d'une variable est soit un entier naturel (*Nat*), soit la valeur du domaine singleton *Undefined*, alors attribuée à une variable qui n'a pas encore été mise à jour (cas des variables volatile et write, initialement). Par ailleurs, le domaine *Value* est muni d'opérateurs booléens de comparaison classiques. Notons toutefois que si au moins l'une des valeurs comparées par un de ces opérateurs est indéfinie, la valeur booléenne *False* est retournée.

Lors de l'instanciation de la taxonomie, les informations de contexte constant sont initialisées. Pour y accéder au moment d'évaluer les règles d'orchestration, nous les stockons dans une mémoire du domaine *Store* (défini dans la partie IX de la figure 5.11), qui associe à chaque variable une valeur. Afin d'accéder aux informations de contexte de manière homogène, nous regroupons dans une mémoire commune toutes les variables pouvant être utilisées dans un programme, qu'elles soient de nature constant, volatile ou write.

Finalement, le domaine *Store* est muni d'une opération *join*, qui prend deux mémoires σ_1 et σ_2 en paramètre et en retourne une nouvelle (voir aussi la section 5.4.3 du livre de D. Schmidt [Sch86]). Pour créer la nouvelle mémoire, cette opération combine les valeurs des variables définies dans σ_1 et σ_2 à l'aide de la fonction *combine*. Cette fonction permet d'expliciter le fait que deux commandes exécutées en parallèle n'interfèrent pas, en combinant leurs résultats sans les écraser. L'implémentation de la fonction *combine* est laissée à la discrétion du développeur qui implémente un compilateur basé sur la sémantique, avec pour seule contrainte la suivante : si une variable est indéfinie (*Undefined*) dans l'une des mémoires d'entrée, la fonction *combine* retourne alors la valeur qui lui est associée dans l'autre mémoire d'entrée. Nous généralisons par $join^*$ l'opération *join* appliquée à un ensemble de mémoires.

5.6.2.3 Fonctions de valuation

L'évaluation de la taxonomie D est une correspondance directe de la syntaxe aux structures, comme cela est défini dans la figure 5.12 (partie haute, page 71). Nous détaillons à présent son processus d'évaluation.

Fonctions de valuation de la taxonomie

D: Domain → Env_c
$\mathbf{D}[\![\mathrm{C}]\!] = \mathbf{C}[\![\mathrm{C}]\!]\rho_c^0$

C: Entityclass → Env_c → Env_c
$\mathbf{C}[\![\mathrm{C}_1\ ;\ \mathrm{C}_2]\!] = \mathbf{C}[\![\mathrm{C}_2]\!] \circ \mathbf{C}[\![\mathrm{C}_1]\!]$
$\mathbf{C}[\![\textbf{class}\ \mathrm{I}_{C1}\ \textbf{extends}\ \mathrm{I}_{C2}\ \mathrm{A}\ \mathrm{M}\ \textbf{end}\ ]\!] = \lambda\rho_c.[[\![\mathrm{I}_{C1}]\!] \mapsto (\mathrm{in}\mathit{Class\text{-}id}([\![\mathrm{I}_{C2}]\!]), \mathbf{A}[\![\mathrm{A}]\!]\rho_a^0, \mathbf{M}[\![\mathrm{M}]\!]\rho_m^0)]\rho_c$
$\mathbf{C}[\![\textbf{class}\ \mathrm{I}_C\ \mathrm{A}\ \mathrm{M}\ \textbf{end}\ ]\!] = \lambda\rho_c.[[\![\mathrm{I}_C]\!] \mapsto (\mathrm{in}\mathit{None}(), \mathbf{A}[\![\mathrm{A}]\!]\rho_a^0, \mathbf{M}[\![\mathrm{M}]\!]\rho_m^0)]\rho_c$

A: Attribute → Env_a → Env_a
$\mathbf{A}[\![\mathrm{A}_1;\ \mathrm{A}_2]\!] = \mathbf{A}[\![\mathrm{A}_2]\!] \circ \mathbf{A}[\![\mathrm{A}_1]\!]$
$\mathbf{A}[\![\mathrm{U}\ \mathrm{I}_p\ \textbf{:}\ \mathrm{T}]\!] = \lambda\rho_a.[[\![\mathrm{I}_p]\!] \mapsto (\mathbf{U}[\![\mathrm{U}]\!], \mathbf{T}[\![\mathrm{T}]\!])]\rho_a$

M: Method → Env_m → Env_m
$\mathbf{M}[\![\mathrm{M}_1;\ \mathrm{M}_2]\!] = \mathbf{M}[\![\mathrm{M}_2]\!] \circ \mathbf{M}[\![\mathrm{M}_1]\!]$
$\mathbf{M}[\![\mathrm{I}_m\textbf{(}\ \mathrm{T}\ \textbf{)}\ \rightarrow \mathrm{I}_p\textbf{[}\ \mathrm{I}_m\textbf{]}]\!] = \lambda\rho_m.[[\![\mathrm{I}_m]\!] \mapsto (\mathbf{T}[\![\mathrm{T}]\!], \mathrm{in}\mathit{Nat}([\![\mathrm{I}_p]\!]), \mathbf{I'}[\![\mathrm{I}_i]\!])]\rho_m$
$\mathbf{M}[\![\mathrm{I}_a\textbf{(}\ \mathrm{T}\ \textbf{)}\ [\mathrm{I}_m]]\!] = \lambda\rho_m.[[\![\mathrm{I}_a]\!] \mapsto (\mathbf{T}[\![\mathrm{T}]\!], \mathrm{in}\mathit{None}(), \mathbf{I'}[\![\mathrm{I}_m]\!])]\rho_m$

U: Modifier-name → *Modifier* (élémentaire)
T: Primitive-type → *Type* (élémentaire)
I': Impl-id → *FunImpl* (accès à l'implémentation via un environnement extérieur)

Fonctions de valuation de l'instanciation d'une taxonomie dite "environnement concret"

W: Instantiation → Env_c → (Env_c × *Store*)
$\mathbf{W}[\![\mathrm{Q}]\!] = \lambda\rho_c.(\mathbf{Q}[\![\mathrm{Q}]\!]\ \rho_c\ \sigma^0\ )$

Q: Entity → Env_c → *Store* → (Env_c × *Store*)
$\mathbf{Q}[\![\mathrm{Q}_1;\ \mathrm{Q}_2]\!] = \lambda\rho_c.\lambda\sigma.\textbf{let}\ (\rho_c', \sigma') = (\mathbf{Q}[\![\mathrm{Q}_1]\!]\ \rho_c\ \sigma\ )\ \textbf{in}\ (\mathbf{Q}[\![\mathrm{Q}_2]\!]\ \rho_c'\ \sigma'\ )$
$\mathbf{Q}[\![\mathrm{I}_e\ \textbf{:}\ \mathrm{I}_C\textbf{[}\ \mathrm{Z}\ \textbf{]}]\!] = \lambda\rho_c.\lambda\sigma.(\mathit{update}_c\ [\![\mathrm{I}_C]\!]\ [\![\mathrm{I}_e]\!]\ \rho_c), (\mathbf{Z}[\![\mathrm{Z}]\!]\ [\![\mathrm{I}_e]\!]\ (\rho_c([\![\mathrm{I}_C]\!]))\ \sigma)$

Z: Assignment → *Entity-id* → *Class-struct* → *Store* → *Store*
$\mathbf{Z}[\![\mathrm{Z}_1;\ \mathrm{Z}_2]\!] = \lambda i_e.\lambda c.(\mathbf{Z}[\![\mathrm{Z}_2]\!]\ i_e\ c)\ \circ (\mathbf{Z}[\![\mathrm{Z}_1]\!]\ i_e\ c)$
$\mathbf{Z}[\![\mathrm{I}_p\textbf{=}\ \mathrm{N}]\!] = \lambda i_e.\lambda(\_, \rho_a, \_, \_).\lambda\sigma.\textbf{let}\ (u, t) = \rho_a([\![\mathrm{I}_p]\!])\ \textbf{in}$
cases m of
is$\mathit{Constant}(c) \rightarrow$ (cases t of is$\mathit{Int}() \rightarrow [(i_e, [\![\mathrm{I}_p]\!]) \mapsto \mathrm{in}\mathit{Nat}(\mathbf{N}[\![\mathrm{N}]\!])\sigma]\ [] _ \rightarrow \sigma$)
$[] \_ \rightarrow \sigma$

N: Numeral → *Nat*

FIG. 5.12: Fonctions de valuation du langage de taxonomie

La fonction de valuation **D** crée un environnement ρ_c contenant les structures de classes *Class-struct*. Cette fonction est initialisée avec un environnement vide représenté par ρ_c^0, et appelle la fonction de valuation **C** sur chaque élément syntaxique de classe. Nous examinons maintenant la définition de la fonction **C**.

Cette fonction prend en paramètre un environnement ρ_c qu'elle *étend* en associant, dans ρ_c, une structure de classe c à l'identifiant I_C du nœud de classe évalué. Cette opération d'extension est réalisée via l'opérateur de composition "∘" (voir la première ligne définissant **C**) : l'environnement renvoyé par l'évaluation de C_1 est donné en paramètre de la fonction évaluant C_2 (le paramètre ρ_c est implicite), produisant un environnement contenant les structures de classes

de C_1 et C_2.

Lorsqu'une classe C_1 hérite d'une classe identifiée par I_{C_2}, son évaluation produit une structure de classe contenant cet identifiant (in*Class-id*(I_{C_2})) ; sinon la structure contient in*None*(). Pour compléter la construction d'une structure de classe avec les attributs et les méthodes qui la concernent, la fonction **C** fait appel aux fonctions de valuation **A** et **M**. Celles-ci prennent en paramètre un environnement d'attributs et un environnement de méthodes initialement vides (ρ_a^0 et ρ_m^0). Chaque environnement est construit selon le même principe que la création de l'environnement des classes ρ_c. Notons que la création des structures de méthodes (par la fonction **M**) utilise une fonction de valuation **I'** qui, étant donné un identifiant, retourne un élément du domaine *FunImpl*. Elle peut être interprétée comme un environnement donnant accès aux implémentations des méthodes, dont nous laissons la construction au développeur du compilateur ou de l'interprète basé sur la sémantique.

Enfin, la fonction de valuation **W** instancie une taxonomie (figure 5.12, partie basse) ; elle consiste à compléter avec des entités l'environnement ρ_c retourné par la fonction **D**, et à créer une mémoire initiale σ (*Store*) de variables constantes. Plus précisément, elle appelle la fonction **Q** sur chaque entité qui est alors ajoutée à l'ensemble des entités de la structure de classe dont elle est une instance. Chacun des attributs constants de ces entités est associé à une valeur, via la fonction **Z** qui complète la mémoire qui lui est donnée en paramètre. Notons que l'initialisation de variables non constantes est ignorée. Nous pouvons à présent définir la sémantique du langage d'orchestration, paramétré par une taxonomie et son instanciation.

5.6.3 Sémantique dynamique du langage d'orchestration

De la même façon que précédemment, nous définissons la sémantique du langage d'orchestration après avoir précisé sa syntaxe abstraite et ses algèbres sémantiques. Comme pour le langage de taxonomie, nous faisons quelques simplifications pour ne garder que le coeur du langage d'orchestration : la syntaxe du langage d'orchestration ne considère que l'opérateur d'égalité pour les prédicats. De plus, en conséquence des simplifications de la syntaxe du langage de taxonomie, les appels de méthodes ne prennent qu'un argument et les valeurs des expressions sont de type *entier*. Enfin, nous ignorons le cas des méthodes ne prenant pas d'argument car leur évaluation n'est qu'une spécialisation du cas avec argument. Toutefois, de même que pour la sémantique du langage de taxonomie, celle du langage d'orchestration s'étend naturellement vers une définition complète à partir de la base présentée ici.

5.6.3.1 Syntaxe abstraite

La syntaxe abstraite du langage d'orchestration est donnée dans la figure 5.13 (page 73). La signification des métavariables est donnée dans la partie haute la figure, et les règles de grammaire dans la partie basse. Un programme, ou scénario d'orchestration `S` est composé d'un ensemble de règles, une règle `R` est composée d'un ensemble de prédicats `B` et d'actions `E`. Etant donnée une entité ou une classe d'entité identifiée par I_z, nous définissons un prédicat et une action comme suit. Un prédicat est une composition de prédicats simples de la forme X_1 **changed** ou $X_1 = X_2$ où X_1 et X_2 sont des expressions. Bien que la sémantique ne soit pas restreinte, les contraintes du langage visuel imposent que X_1 soit toujours une variable, X_2 pouvant être alors

une constante ou une variable. Dans la syntaxe textuelle, une variable est de la forme $I_z.I_a$, où I_a fait référence à un attribut d'entité ou de classe ; par exemple, le prédicat noté p_0 dans la règle R1.1 de la figure 5.6 (page 56) utilise la variable $I_z.I_a$ = Meeting.start.

Une action est soit l'appel d'une méthode identifiée par $I_z.I_m$ et dont le paramètre est une expression X, soit l'affectation à une expression X d'une variable (ou attribut d'entité) identifiée par $I_z.I_w$. Dans ce dernier cas, I_w doit toujours être une référence à un attribut write. Par exemple, la règle R2.3 de la figure 5.6 (page 56), définit une action affectant la variable $I_z.I_w$ = meetingPhone.session_open, où session_open est un attribut write de la classe IPPhone dont meetingPhone est une instance. Enfin, lorsqu'une règle contient une séquence d'actions, celle-ci est préfixée par le mot-clé **seq**.

S ∈ Scenario	X ∈ Expression	I_r ∈ Rule-id
R ∈ Rule	E ∈ Actuator	I_c ∈ Class-id
B ∈ Predicate	K ∈ MethodCall	I_z ∈ Ref-id = Class-id ∪ Entity-id

S ::= **scenario** R **end**
R ::= R_1 ; R_2 | **rule** I_r **cond :** B **do :** E **end**
B ::= (B_1 **and** B_2) | (B_1 **or** B_2) | (B_1 **seq** B_2) | X_1 **=** X_2 | **ALL** (I_c , B) | X **changed**
X ::= N | $I_z.I_a$
E ::= **seq**$^?$ E_1 ; E_2 | $I_z.I_w$ ← X | K
K ::= $I_z.I_m$ (X)

FIG. 5.13: Syntaxe abstraite du langage d'orchestration

5.6.3.2 Algèbres sémantiques

L'évaluation d'un scénario nécessite un ensemble de domaines définis dans la figure 5.14, que nous présentons maintenant.

I. Environnements des tuples-entités ou *configurations*

Domain $\omega \in TupleEnv = Class\text{-}id \rightarrow Entity\text{-}id_{\perp}$
Operations
$\omega^0 : TupleEnv$
$\omega^0 = \lambda i.\perp$
$restr : \mathcal{P}(Class\text{-}id) \rightarrow TupleEnv \rightarrow TupleEnv$
$restr = \lambda l.\lambda\omega.(l = \emptyset \rightarrow \lambda i.\perp \,[]\, [i_1 \mapsto \omega(i_1)]...[i_k \mapsto \omega(i_k)](\lambda i.\perp))$
where $\{i_1, ..., i_k\} = l$
Domain $\Omega \in TupleEnv^{\bullet} = \mathcal{P}(TupleEnv)$
Operations
$\Omega^0 : Env_c \rightarrow TupleEnv^{\bullet}$
$\Omega^0 = \lambda\rho_c.\{([i_{c_1} \mapsto i_{e_1}]...[i_{c_n} \mapsto i_{e_n}](\lambda x.\perp) \in TupleEnv \mid \{i_{c_1}, ..., i_{c_n}\} = Class\text{-}id, i_{e_j} \in \rho_c(i_{c_j})\downarrow_4\}$
$restr_{\Omega} : \mathcal{P}(Class\text{-}id) \rightarrow TupleEnv^{\bullet} \rightarrow TupleEnv^{\bullet}$
$restr_{\Omega} = \lambda l.\lambda\Omega.\{(restr\ l\ \omega) \mid \omega \in \Omega\}$

II. Fonctions abstraites d'évaluation des expressions de prédicats

Domain $f \in Cmpfun = (Store \rightarrow Value) \times (Store \rightarrow Value) \rightarrow Bool$
Domain $g \in Chgfun = (Store \rightarrow Value) \rightarrow Bool$

III. Environnement des règles

Domain $\rho_r \in RuleEnv = Rule\text{-}id \rightarrow Rule\text{-}struct$ (environnement initial : $\rho_r^0 = \lambda i.(\emptyset, \lambda s.s)$)
where $Rule\text{-}struct = (TupleEnv^{\bullet} \times (TupleEnv^{\bullet} \rightarrow Store \rightarrow Store))$

FIG. 5.14: Domaines du langage d'orchestration

Domaine des prédicats Dans la section 5.2, pour pouvoir traiter les prédicats, qu'ils soient des filtres ou des conditions booléennes, nous avions introduit la structure *Tuple*. Nous la définissons dans la figure 5.14 (partie I) sous la forme d'une fonction partielle du domaine *TupleEnv* (la structure *TupleSet* de la section 5.2 correspond alors à une partie de *TupleEnv*, aussi notée $TupleEnv^{\bullet}$). Plus précisément, un élément $\omega \in TupleEnv$ associe à chaque identifiant d'une classe de la taxonomie l'identifiant d'une entité. L'avantage de cette traduction est que la substitution d'une classe i_c à une entité s'exprime naturellement, en appliquant ω à i_c. Ainsi, un environnement ω est une *configuration* particulière pour laquelle un prédicat peut être satisfait.

Pour faciliter la compréhension de l'évaluation des prédicats indépendamment du processus d'évaluation de règles, la sémantique des prédicats fait abstraction de la nature spécifique d'un état du programme (ici, l'état du programme est défini par deux éléments mémoire *Store* que nous avions notés σ_t et $\sigma_{t\text{-}1}$ dans la section 5.5). Ainsi, l'évaluation des prédicats peut être transposée à d'autres représentations de l'état d'un programme. Pour ce faire, nous donnons dans la figure 5.14 (partie II) deux fonctions abstraites f et g encapsulant la structure *Store*. La première fonction $f \in Cmpfun$ sert à l'évaluation des prédicats de la forme $X_1{=}X_2$: c'est une fonction booléenne qui prend en argument deux fonctions x et x', représentant deux expressions dont les valeurs, selon un *Store* donné, sont comparées. Cette fonction La seconde fonction $g \in Chgfun$ sert à l'évaluation des prédicats de la forme X **changed** : c'est une fonction booléenne

qui prend en argument une fonction y qui représente une expression dont la valeur, selon un *Store* donné, est comparée à une valeur test (nous voyons plus loin que, dans la sémantique de Pantagruel, cette valeur test correspond à la valeur de la variable dans une mémoire précédente). Une représentation concrète des deux fonctions f et g sera donnée au moment de l'évaluation des prédicats (figure 5.16, page 76).

Domaine des règles La partie III de la figure 5.14 (page 74) définit un environnement de règles *RuleEnv*. Cet environnement contient les structures *Rule-struct* représentant les règles d'orchestration. Plus précisément, il associe à chaque identifiant de règle un couple de la forme (Ω, A), où Ω est une partie de *TupleEnv*, et où A est un ensemble de fonctions de *Store* dans *Store* correspondant aux actions non encore évaluées de la règle. Nous verrons dans le chapitre 7 que cet environnement est utile pour vérifier certaines propriétés d'un programme Pantagruel, par exemple la non-interférence entre deux règles.

5.6.3.3 Fonctions de valuation

Un programme Pantagruel est un scénario S paramétré par une taxonomie D et son instanciation W, comme défini dans la figure 5.15 (page 75, deuxième ligne). Nous examinons à présent en détail l'évaluation d'un scénario.

$\mathbf{P}$: Program $\rightarrow$ *Store*

$\mathbf{P}[\![\mathrm{D}\,;\,\mathrm{W}\,;\,\mathrm{S}]\!] = \textbf{let}\ \rho_c, \sigma_0 = (\mathbf{W}[\![\mathrm{W}]\!]\ (\mathbf{D}[\![\mathrm{D}]\!]))\ \textbf{in}\ (\mathbf{S}[\![\mathrm{S}]\!]\ \rho_c\ \sigma_0)$

$\mathbf{S}$: Scenario $\rightarrow$ $Env_c \rightarrow$ *Store* $\rightarrow$ *Store*

$\mathbf{S}[\![\mathrm{R}]\!] = \lambda\rho_c.\lambda\sigma.fix\big(\lambda f.\lambda\sigma_{t\text{-}1}.\lambda\sigma_t.\textbf{let}\ \rho_r = \mathbf{R}[\![\mathrm{R}]\!]\ \rho_c\ \sigma_{t\text{-}1}\ \sigma_t\ \rho_r^0\ \textbf{in}$
$\quad\textbf{let}\ \{\sigma_1, ..., \sigma_k\} = \{(a_i\ \Omega_i\ \sigma_t) \mid i \in \textit{Rule-id},\ \rho_r(i) = (\Omega_i, a_i),\ \textbf{such that}\ \Omega_i \neq \emptyset\}\ \textbf{in}$
$\quad\quad\textbf{let}\ \sigma'_{t+1} = \textit{join}^*\ \{\sigma_1, ..., \sigma_k\}\ \textbf{in}$
$\quad\quad\quad\textbf{let}\ \sigma_{t+1} = [v_n \mapsto (\textit{pullvol}\ v_n)] \ldots [v_1 \mapsto (\textit{pullvol}\ v_1)]\sigma'_{t+1}\ \textbf{in}$
$\quad\quad\quad\quad(f\ \sigma_t\ \sigma_{t+1})\big)\ \sigma\ \sigma$
$\quad\textbf{where}\ \{v_1, ..., v_n\} = \{(i_e, i_p) \in \textit{Variable} \mid$
$\quad\quad\quad\textbf{let}\ \rho_a = (\textit{lookup}_c\ i_e\ \rho_c)\downarrow_2\ \textbf{in}\ (\textit{modifier}\ i_p\ \rho_a) = \mathrm{is}\,\textit{Volatile}()\}$
$\quad\textbf{and}\ \textit{pullvol} : Volatile\text{-}id \rightarrow Nat$
$\quad\quad\textit{pullvol}$ = fonction externe récupérant les variables volatiles recueillies par les entités réelles

FIG. 5.15: Fonctions de valuation du programme principal et d'un scénario

Evaluation d'un scénario La fonction de valuation **S** est une boucle infinie, modélisée par une fonction de point fixe (*fix*) sans condition d'arrêt et prenant en paramètre deux mémoires $\sigma_{t\text{-}1}$ et σ_t, initialement égales à la mémoire σ_0 (à la première itération de la boucle) qui ne contient que des variables constantes. A l'intérieur de cette boucle, la fonction de valuation **R** – détaillée au paragraphe suivant – construit l'environnement de règles ρ_r, contenant toutes les règles du programme sous forme de structures *Rule-struct* (voir figure 5.14, page 74). Pour construire cet environnement, chaque règle est évaluée avec σ_t et $\sigma_{t\text{-}1}$. L'évaluation de chaque règle i renvoie un couple (Ω_i, a_i) : Ω_i est un ensemble de configurations ; a_i est un ensemble d'actions attendant

deux paramètres – Ω_i et la mémoire courante σ_t – pour être exécutées (voir deuxième **let** de la fonction **S**, figure 5.15). Pour chaque règle (Ω_i, a_i) dont l'ensemble Ω_i n'est pas vide, les actions a_i sont ensuite effectivement évaluées avec Ω_i et σ_t. Puis les mémoires produites par les règles sont jointes (avec la fonction $join^*$) pour donner une nouvelle mémoire σ'_{t+1}. Les variables volatile y sont injectées (via la fonction *pullvol*, laissée à la charge du développeur qui implémente un compilateur vers une plate-forme cible), donnant lieu à la mémoire σ_{t+1} qui sera la mémoire courante de l'itération suivante.

Evaluation d'une règle La fonction d'évaluation des règles est définie dans la figure 5.16.

$$
\begin{array}{l}
\mathbf{R}\text{: Rule} \rightarrow \mathit{Env_c} \rightarrow \mathit{Store} \rightarrow \mathit{Store} \rightarrow \mathit{RuleEnv} \rightarrow \mathit{RuleEnv} \\
\quad \mathbf{R}[\![\mathrm{R}_1;\ \mathrm{R}_2]\!] = \lambda\rho_c.\lambda\sigma_{t\text{-}1}.\lambda\sigma_t.(\mathbf{R}[\![\mathrm{R}_2]\!]\ \rho_c\ \sigma_{t\text{-}1}\ \sigma_t) \circ (\mathbf{R}[\![\mathrm{R}_1]\!]\ \rho_c\ \sigma_{t\text{-}1}\ \sigma_t) \\
\quad \mathbf{R}[\![\textbf{rule}\ \mathrm{I}_r\ \textbf{when :}\ \mathrm{B}\ \textbf{do :}\ \mathrm{E}\ \textbf{end}]\!] = \lambda\rho_c.\lambda\sigma_{t\text{-}1}.\lambda\sigma_t.\lambda\rho_r. \\
\qquad \textbf{let}\ f = \lambda\sigma.\lambda(x,x').((x\ \sigma){=}(x'\ \sigma)) \\
\qquad \textbf{in}\ \ \textbf{let}\ f_1 = (f\ \sigma_t) \\
\qquad\qquad\qquad f_2 = (f\ \sigma_{t\text{-}1}) \\
\qquad\qquad\qquad g_1 = \lambda y.((y\ \sigma_{t\text{-}1}) \neq (y\ \sigma_t)) \\
\qquad\qquad\qquad g_2 = \lambda y.\mathit{false} \\
\qquad\qquad \textbf{in}\ [[\![\mathrm{I}_r]\!] \mapsto ((\mathbf{B}[\![\mathrm{B}]\!]\ f_1\ g_1\ \Omega) \cap \complement_\Omega(\mathbf{B}[\![\mathrm{B}]\!]\ f_2\ g_2\ \Omega), \mathbf{E}[\![\mathrm{E}]\!]\ \rho_c\ )]\rho_r \\
\quad \textbf{where} \quad \complement_X(Y) \ = \text{l'ensemble complémentaire de Y dans X} \\
\quad \textbf{and} \qquad \Omega \qquad\ = \Omega^0\ \rho_c
\end{array}
$$

FIG. 5.16: Fonction de valuation d'une règle

Elle requiert l'environnement ρ_c servant à construire un ensemble initial de configurations, deux mémoires σ_t et $\sigma_{t\text{-}1}$, et un environnement de règles ρ_r. A l'issue de l'évaluation de chaque règle I_r, ρ_r est étendu par une nouvelle structure *Rule-struct* représentant cette règle. Le processus d'évaluation se déroule comme suit : les prédicats de chaque règle identifiée par I_r sont évalués deux fois, d'abord avec σ_t, ensuite avec $\sigma_{t\text{-}1}$. La première évaluation utilise deux fonctions concrètes $f_1 \in \mathit{Cmpfun}$ et $g_1 \in \mathit{Chgfun}$, encapsulant la mémoire σ_t. La fonction concrète f_1 compare deux expressions selon σ_t. La fonction g_1 sert à évaluer le prédicat **changed** : elle compare une expression évaluée selon σ_t avec sa valeur selon $\sigma_{t\text{-}1}$.

De la même façon, la mémoire $\sigma_{t\text{-}1}$ est encapsulée dans f_2 et g_2 pour la deuxième évaluation. g_2 est une fonction "fantôme" ou *dummy*, dont le corps est ($\lambda i.\mathit{false}$) : les prédicats **changed** ayant déjà opéré un filtrage des configurations selon $\sigma_{t\text{-}1}$ et σ_t lors de la première évaluation, cette fonction "fait semblant" de les évaluer et permet ainsi de les préserver.

Finalement, les configurations qui satisfont les prédicats d'une règle i sont collectées conformément au modèle de changement de contexte : elles correspondent à l'intersection Ω_i de l'ensemble des configurations satisfaites pour σ_t et de l'ensemble des configurations insatisfaites pour $\sigma_{t\text{-}1}$ – c'est-à-dire le complémentaire de l'ensemble satisfait pour $\sigma_{t\text{-}1}$ (d'où la définition de la fonction fantôme g_2).

Les actions de la règle sont quant à elles partiellement évaluées avec la fonction **E** en lui donnant en argument l'environnement des classes ρ_c. Dans les paragraphes suivants, nous précisons les deux sous-parties du processus d'évaluation d'une règle : l'évaluation des prédicats et l'évaluation des actions.

Evaluation des prédicats La figure 5.17 (partie haute) détaille la fonction d'évaluation **B** des prédicats.

$\mathbf{B}$: Predicate $\rightarrow$ *Cmpfun* $\rightarrow$ *Chgfun* $\rightarrow$ $TupleEnv^{\bullet} \rightarrow TupleEnv^{\bullet}$

$\mathbf{B}[\![\mathrm{X}_1 = \mathrm{X}_2]\!] = \lambda f.\lambda g.\lambda\Omega.\{\omega \in \Omega \mid f((\mathbf{X}[\![\mathrm{X}_1]\!]\ \omega)\ , (\mathbf{X}[\![\mathrm{X}_2]\!]\ \omega)\ ) = true\}$
$\mathbf{B}[\![\mathrm{X}\ \textbf{changed}]\!] = \lambda f.\lambda g.\lambda\Omega.\{\omega \in \Omega \mid g(\mathbf{X}[\![\mathrm{X}]\!]\omega) = true\}$
$\mathbf{B}[\![\mathrm{B}_1\ \textbf{and}\ \mathrm{B}_2]\!] = \lambda f.\lambda g.\lambda\Omega.(\mathbf{B}[\![\mathrm{B}_1]\!]\ f\ g\ \Omega) \cap (\mathbf{B}[\![\mathrm{B}_2]\!]\ f\ g\ \Omega)$
$\mathbf{B}[\![\mathrm{B}_1\ \textbf{or}\ \mathrm{B}_2]\!] = \lambda f.\lambda g.\lambda\Omega.(\mathbf{B}[\![\mathrm{B}_1]\!]\ f\ g\ \Omega) \cup (\mathbf{B}[\![\mathrm{B}_2]\!]\ f\ g\ \Omega)$
$\mathbf{B}[\![\mathrm{B}_1\ \textbf{seq}\ \mathrm{B}_2]\!] = \lambda f.\lambda g.(\mathbf{B}[\![\mathrm{B}_2]\!]\ f\ g\ \circ \mathbf{B}[\![\mathrm{B}_1]\!]\ f\ g\ )$

$\mathbf{B}[\![\textbf{ALL}\ \mathrm{I}_C,\ \mathrm{B}]\!] = \lambda f.\lambda g.\lambda\Omega.\textbf{let}\ \Omega' = \mathbf{B}[\![\mathrm{B}]\!]\ f\ g\ \Omega\ \textbf{in}\ (restr_{\Omega}\ \{[\![\mathrm{I}_c]\!]\}\ \Omega' = restr_{\Omega}\ \{[\![\mathrm{I}_c]\!]\}\ \Omega) \rightarrow \Omega'\ [\!]\ \emptyset$

$\mathbf{B}[\![\textbf{NOT}\ \mathrm{B}]\!] = \lambda f.\lambda g.\lambda\Omega.\complement_{\Omega}(\mathbf{B}[\![\mathrm{B}]\!]\ f\ g\ \Omega)$

$\mathbf{X}$: Expression $\rightarrow$ *TupleEnv* $\rightarrow$ *Store* $\rightarrow$ *Value*

$\mathbf{X}[\![\mathrm{N}]\!] = \lambda\omega.\lambda\sigma.\mathbf{N}[\![\mathrm{N}]\!]$
$\mathbf{X}[\![\mathrm{I}_e.\ \mathrm{I}_p]\!] = \lambda\omega.\lambda\sigma.\sigma([\![\mathrm{I}_e]\!], [\![\mathrm{I}_p]\!])$
$\mathbf{X}[\![\mathrm{I}_C.\ \mathrm{I}_p]\!] = \lambda\omega.\lambda\sigma.\sigma(\omega([\![\mathrm{I}_C]\!]), [\![\mathrm{I}_p]\!])$

$\mathbf{X'}$: Expression $\rightarrow \mathcal{P}(Class\text{-}id)$

$\mathbf{X}'[\![\mathrm{N}]\!] = \emptyset$
$\mathbf{X}'[\![\mathrm{I}_e.\ \mathrm{I}_p]\!] = \emptyset$
$\mathbf{X}'[\![\mathrm{I}_C.\ \mathrm{I}_p]\!] = \{[\![\mathrm{I}_C]\!]\}$

FIG. 5.17: Fonctions de valuation des prédicats et des expressions

La fonction **B** prend en argument deux fonctions concrètes $f \in$ *Cmpfun* et $g \in$ *Chgfun*. Afin de définir l'opération de combinaison de flux, **B** prend également en argument un ensemble de configurations Ω et renvoie un nouvel ensemble Ω'. Chaque élément de Ω' satisfait un prédicat conformément à l'évaluation de ses expressions avec les fonctions f et g.

Les prédicats sont des combinaisons de prédicats élémentaires de la forme X_1 **changed** ou X_1 **=** X_2, où X_i est une expression désignant une variable ou une constante. Sa fonction de valuation **X** est définie dans la partie basse de la figure 5.17. Les prédicats peuvent être associés à cinq types d'opérateurs, **and**, **or**, **seq**, **ALL** et **NOT**. Evaluer une combinaison de prédicats revient à combiner des flux de données, où les données sont des ensembles de configurations filtrés par chaque prédicat élémentaire. Ce principe, commun aux langages de requêtes, permet d'appliquer une variété de prédicats de filtrage et accroît l'expressivité de Pantagruel. Les combinateurs **and** et **or** sont traduits respectivement par une intersection et une union d'ensembles. Le prédicat **seq** est équivalent à une opération de composition, où l'ensemble (de sortie) renvoyé par un prédicat devient le paramètre (d'entrée) du prédicat suivant.

Cas particuliers : les préfixes ALL et NOT Voyons maintenant les opérateurs **ALL** et **NOT** préfixant un prédicat. Le préfixe **ALL** est un opérateur "tout ou rien". Par exemple, en représentation "statique", étant donné un ensemble $\{c_1, ..., c_n\}$ d'instances de la classe `C`, un prédicat de la forme **ALL** `I`$_c$`, I`$_c$`.x = 3` peut être développé en une combinaison-**and** de prédicats, à savoir `c`$_1$`.x = 3` **and** ... **and** `c`$_n$`.x = 3`. Nous exprimons cette combinaison en utilisant la fonction de restriction $restr_{\Omega}$ sur $TupleEnv^{\bullet}$ (figure 5.14, page 74) : on vérifie que l'ensemble des

instances c_i de `C` satisfaisant c_i`.x = 3` étant donné Ω' est égal à la restriction à `C` de Ω. Le cas échéant, Ω' est retourné, filtrant les entités des autres classes tout en préservant la condition-**ALL**. Enfin, le préfixe **NOT** permet d'exprimer des prédicats négatifs, et correspond simplement au complémentaire de l'ensemble des configurations de Ω satisfaisant le prédicat préfixé.

Illustration Nous illustrons par la figure 5.18 l'évaluation des prédicats par filtrage pour la règle R1.4 de l'exemple de gestion de réunion (figure 5.5, page 55), étant donné l'environnement concret de la figure 5.4 (page 53). Les prédicats de cette règle sélectionnent les personnes possédant un smartphone et dont la localisation a changé : elle contient des prédicats sur la classe SmartPhone (abrégée en *S*) et la classe PersonProfile (abrégée en *P*). Supposons que Al et Bob sont tous deux dans la pièce COFFEEROOM au temps `t-1` et changent de pièce au temps `t`. Les profils de ces deux personnes sont donc collectés, ainsi que le smartphone dont ils sont propriétaires (attribut owner), engendrant l'ensemble Ω dont nous montrons un extrait dans la figure. Pour chaque configuration de Ω, le prédicat *S*.owner = *P*.name est toujours vrai car il compare des constantes, et le prédicat P.location **changed** devient vrai de $\sigma_{t\text{-}1}$ à σ_t.

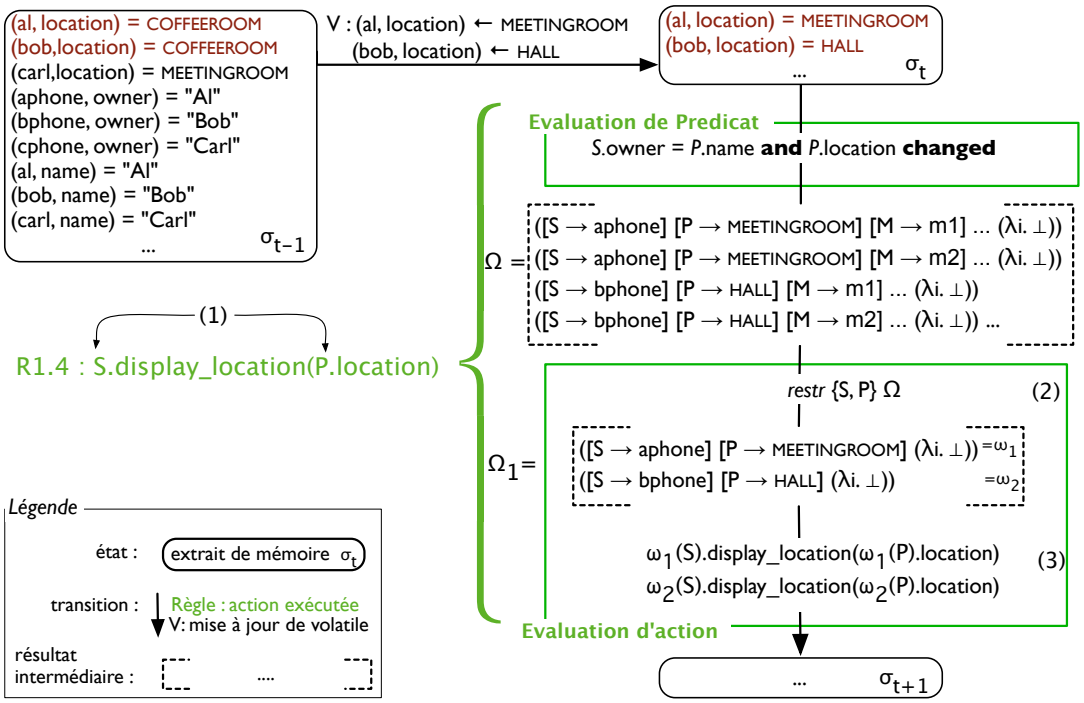

FIG. 5.18: Une évaluation basée sur le filtrage

Evaluation des actions La partie haute de la figure 5.19 détaille la fonction d'évaluation **E** des actions d'une règle, qui prend en argument l'environnement ρ_c, pour accéder à la déclaration des effets d'une méthode, et une mémoire σ, pour accéder aux variables utilisées dans les actions. **E** prend également en argument un ensemble de configurations Ω (celui renvoyé par l'évaluation

des prédicats). Si les actions d'une règle sont exécutées en parallèle, les mémoires renvoyées par l'évaluation de ces actions sont jointes de la même façon que le sont celles des règles au niveau d'un scénario. En mode séquentiel (mot-clé **seq** dans la syntaxe abstraite), la mémoire renvoyée par une action devient l'argument mémoire de l'action suivante.

$$
\begin{array}{l}
\mathbf{E}\text{: Actuator} \to \mathit{Env}_c \to \mathit{TupleEnv}^\bullet \to \mathit{Store} \to \mathit{Store} \\
\quad \mathbf{E}[\![\mathrm{E}_1\ ;\ \mathrm{E}_2]\!] = \lambda\rho_c.\lambda\Omega.\lambda\sigma.(\mathbf{E}[\![\mathrm{E}_1]\!]\ \Omega\ \rho_c\ \sigma)\ \mathit{join}\ (\mathbf{E}[\![\mathrm{E}_2]\!]\ \Omega\ \rho_c\ \sigma) \\
\quad \mathbf{E}[\![\textbf{seq}\ \mathrm{E}_1\ ;\ \mathrm{E}_2]\!] = \lambda\rho_c.\lambda\Omega.(\mathbf{E}[\![\mathrm{E}_2]\!]\ \Omega\ \rho_c)\ \circ\ (\mathbf{E}[\![\mathrm{E}_1]\!]\ \Omega\ \rho_c) \\
\quad \mathbf{E}[\![\mathrm{I}_z.\mathrm{I}_p \leftarrow \mathrm{X}]\!] = \lambda\rho_c.\lambda\Omega.\lambda\sigma.\text{cases}\ [\![\mathrm{I}_z]\!]\ \text{of} \\
\qquad \text{is}\mathit{Class\text{-}id}(i_c) \to \textbf{let}\ \Omega_1 = (\mathit{restr}_\Omega\ (\mathbf{X}'[\![\mathrm{X}]\!] \cup \{i_c\})\ \Omega) \\
\qquad\qquad\qquad \textbf{in}\ \mathit{join}^*\ \{[(\omega(i_c), [\![\mathrm{I}_p]\!]) \mapsto (\mathbf{X}[\![\mathrm{X}]\!]\ \omega\ \sigma)]\sigma \mid \omega \in \Omega_1\} \\
\qquad [\,]\ \text{is}\mathit{Entity\text{-}id}(i_e) \to \textbf{let}\ \omega = \mathit{first}\ (\mathit{restr}_\Omega\ \mathbf{X}'[\![\mathrm{X}]\!]\ \Omega) \\
\qquad\qquad\qquad\quad \textbf{in}\ [(i_e, [\![\mathrm{I}_p]\!]) \mapsto (\mathbf{X}[\![\mathrm{X}]\!]\ \omega\ \sigma)]\sigma \\
\qquad \textbf{where}\ \mathit{first}(X) = \text{une fonction choisissant arbitrairement le premier élément de } X \\
\quad \mathbf{E}[\![\mathrm{K}]\!] = \lambda\rho_c.\lambda\Omega.\lambda\sigma.(\mathbf{K}[\![\mathrm{K}]\!]\ \Omega\ \rho_c\ \sigma) \\
\hline
\mathbf{K}\text{: MethodCall} \to \mathit{Env}_c \to \mathit{TupleEnv}^\bullet \to \mathit{Store} \to \mathit{Store} \\
\quad \mathbf{K}[\![\mathrm{I}_z.\mathrm{I}_m\ (\ \mathrm{X}\ )]\!] = \lambda\rho_c.\lambda\Omega.\lambda\sigma.\text{cases}\ [\![\mathrm{I}_z]\!]\ \text{of} \\
\qquad \text{is}\mathit{Class\text{-}id}(i_c) \to \textbf{let}\ \Omega_1 = (\mathit{restr}_\Omega\ (\mathbf{X}'[\![\mathrm{X}]\!] \cup \{i_c\})\ \Omega)\ \textbf{in} \\
\qquad\quad \textbf{let}\ (t, e, f) = (\mathit{lookup}_m\ [\![\mathrm{I}_m]\!]\ i_c\ \rho_c)\ \textbf{in}\ \mathit{join}^*\ \{(\mathit{evalfun}\ f\ \omega(i_c)\ e\ (\mathbf{X}[\![\mathrm{X}]\!]\ \omega\ \sigma)\ \sigma) \mid \omega \in \Omega_1\} \\
\qquad [\,]\ \text{is}\mathit{Entity\text{-}id}(i_e) \to \textbf{let}\ \Omega_1 = \mathit{restr}_\Omega\ \mathbf{X}'[\![\mathrm{X}]\!]\ \Omega\ \textbf{in} \\
\qquad\quad \textbf{let}\ (t, e, f) = (\mathit{lookup}_m\ [\![\mathrm{I}_m]\!]\ i_e\ \rho_c)\ \textbf{in}\ \mathit{join}^*\ \{(\mathit{evalfun}\ f\ i_e\ e\ (\mathbf{X}[\![\mathrm{X}]\!]\ \omega\ \sigma)\ \sigma) \mid \omega \in \Omega_1\} \\
\quad \textbf{where}\ \mathit{evalfun} : \mathit{FunImpl} \to \mathit{Entity\text{-}id} \to \mathit{Effect} \to \mathit{Value} \to \mathit{Store} \to \mathit{Store} \\
\qquad \mathit{evalfun} = \lambda f.\lambda i_e.\lambda e.\lambda v.\lambda\sigma.\text{cases}\ ((f\ v), e)\ \text{of} \\
\qquad\quad (\text{is}\mathit{Nat}(n),\ \text{is}\mathit{Attribute\text{-}id}(i_p)) \to [(i_e, i_p) \mapsto \text{in}\mathit{Nat}(n)]\sigma \\
\qquad\quad [\,]\ (\_,\ \text{is}\mathit{None}()) \to \sigma
\end{array}
$$

FIG. 5.19: Fonction de valuation des actions

Voyons à présent l'évaluation **K** d'une invocation de méthode, définie dans la partie basse de la figure 5.19. Comme pour les prédicats, une invocation de méthode peut être définie sur des classes. Son évaluation peut se décomposer en trois phases : (1) les classes dont la méthode dépend sont extraites ; une méthode dépend d'une classe si elle s'applique sur la classe ou si son paramètre fait référence à un attribut de classe au lieu d'un attribut d'instance (une variable "réelle"). Pour obtenir les dépendances de classes, nous utilisons la fonction d'évaluation `X'` (partie basse de la figure 5.17, page 77) ; (2) on définit une restriction Ω_1 de Ω sur ces classes, en utilisant l'opération restr_ω du domaine TupleEnv$^\bullet$; (3) la méthode est invoquée via *evalfun* pour chaque configuration restreinte de Ω_1. Plus spécifiquement, on récupère, à l'aide de l'opération lookup_m appliquée sur la classe ou l'entité concernée I_z, la définition de la méthode invoquée , qui retourne la structure (de type *MethodStruct*) correspondante. Cette structure inclut l'implémentation f de cette méthode, qu'on exécute ensuite pour chaque entité sur laquelle elle est invoquée (il s'agit de l'application $(f\ v)$ dans la définition de l'opération $\mathit{evalfun}$).

Si la méthode déclare des effets, les variables write correspondantes sont directement mises à jour dans la mémoire σ. Dans la fonction de valuation **K** d'une méthode $\mathrm{I}_z.\mathrm{I}_m$, qui modifie une variable write notée (i_e, i_p) (avant-dernière ligne de la figure **??**), cette dernière est mise à

jour avec la valeur retournée par l'application de la méthode ; De la même façon, on définit la sémantique de l'affectation.

Cette technique d'évaluation basée sur les fonctions de restriction permet d'éviter élégamment la duplication de l'exécution d'une action dans une règle à base de classes : pour un ensemble $\mathcal{C}$ de classes dont l'action dépend, l'action est exécutée une seule fois pour chaque entité de la classe, étant donnée la restriction sur $\mathcal{C}$ d'un ensemble de configurations donné. Le défaut de cette technique est qu'elle nécessite le calcul d'un produit cartésien (pour l'ensemble des configurations de départ donné à la fonction d'évaluation $\mathbf{B}$). Cette opération peut générer un ensemble très grand de configurations si l'application Pantagruel utilise un grand nombre d'entités. Il est possible d'optimiser le traitement mais nous ne l'abordons pas dans cette thèse.

Illustration La figure 5.18 (page 78) illustre le processus global d'évaluation de la règle R1.4 de notre exemple. L'action de cette règle est l'affichage de la localisation d'une personne sur son téléphone mobile ; cette action implique deux classes, SmartPhone (S) et PersonProfile (P) (étape 1). Ainsi, l'ensemble Ω des configurations est restreint à ces deux classes (étape 2), renvoyant Ω_1. L'implémentation de la méthode display_location est récupérée, et est exécutée sur chaque instance de SmartPhone, ici aphone et bphone (étape 3). Dans cet exemple, la méthode display_location ne déclare pas d'effet, la mémoire σ_{t+1} reste donc égale à σ_t, modulo les variables volatile.

Notons que la sémantique des prédicats et des actions reste valable si une règle s'applique sur des entités spécifiques. En particulier, lorsqu'une action agit uniquement sur une entité spécifique, et que ses paramètres sont des constantes ou des variables (réelles c'est-à-dire préfixées par une entité et non une classe), elle est évaluée une fois sur cette entité : la fonction $restr_\Omega$ restreint l'ensemble des configurations (renvoyé par les prédicats) sur l'ensemble vide, puisqu'il n'y a pas de classes impliquée. Elle retourne alors le singleton $\{\lambda i.\bot\}$, tel que spécifié dans la figure 5.14 (page 74). Puisqu'il n'y a qu'un seul élément dans l'ensemble retourné, l'action est évaluée une et une seule fois.

Ce principe, basé sur les restrictions, bénéficie d'un autre avantage, mis en valeur par ce "cas particulier" : une règle contenant des prédicats-filtres, mais appelant une action dépendant uniquement d'entités spécifiques s'interprète comme une formule existentielle. Par exemple, la règle suivante : `when C.x = 42 do` $\mathtt{e}_2$`.m()` s'interprète ainsi : "s'il existe une entité appartenant à `C`, telle que `C.x = 42`, alors, exécuter l'action $\mathtt{e}_2$`.m()`". Cette interprétation est valable de manière générale dès que la règle contient parmi ces prédicats une classe qui n'est pas utilisée dans ses actions (ici, `C`).

5.7 Implémentation

Afin d'augmenter l'accessibilité de Pantagruel, nous avons développé un éditeur visuel pour développer des applications qui peuvent ensuite être compilées afin d'être testées dans un environnement réel ou simulé.

5.7.1 Un éditeur visuel

Le langage Pantagruel est intégré à un éditeur visuel paramétré par une taxonomie (figure 5.20) et muni d'une palette d'outils (à droite de la figure). Il se présente sous la forme d'un *plug-in* dans l'environnement de développement intégré Eclipse [Ecl]. Les fonctionnalités que nous y avons apportées permettent de construire des programmes d'orchestration syntaxiquement corrects et bien typés par construction : grâce à la taxonomie, l'éditeur met à disposition de l'utilisateur les classes d'entités, les entités, leurs attributs et leurs méthodes au moyen de menus de sélection adaptés au contexte d'édition. Par exemple, lorsque l'utilisateur passe la souris sur un contrôleur, une représentation sommaire en langage naturel de la règle associée à ce contrôleur est affichée (fenêtre contextuelle Rule R0, apparaissant sous la règle R0 dans la figure). Lorsqu'il double-clique sur une entité dans la colonne des capteurs, une liste déroulante avec les attributs de l'entité apparaît, ainsi qu'un ensemble de valeurs autorisées par chaque attribut (fenêtre bathdoor - Sensor, apparaissant sur l'attribut status de l'entité bathdoor dans la figure). Nous avons appliqué un principe similaire pour guider la définition d'actions, ce qui garantit notamment que seules les variables *write* sont affectées et qu'une méthode est appelée avec le(s) bon(s) paramètre(s).

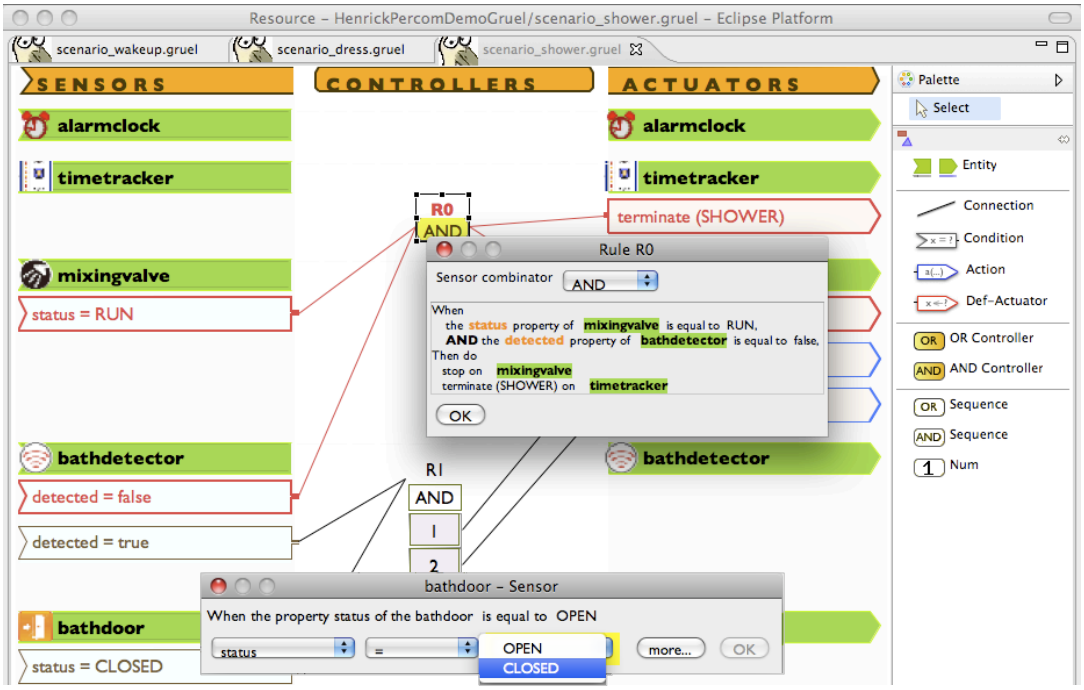

FIG. 5.20: Environnement de programmation visuel de Pantagruel

5.7.2 Une couche d'abstraction au-dessus d'une approche générative

Une fois édités, les programmes Pantagruel peuvent être déployés sur une plateforme dédiée aux applications de l'informatique ubiquitaire. L'implémentation de Pantagruel est basée sur la sémantique dénotationnelle du langage. Ainsi, les programmes compilés bénéficient des propriétés pouvant être vérifiées grâce à la sémantique et qui sont l'objet du chapitre 7. L'implémentation du langage Pantagruel est basée sur la plate-forme DiaSuite, que nous présentons maintenant.

5.7.2.1 Présentation de DiaSuite

DiaSuite est une suite d'outils développée par l'équipe Phoenix, incluant un intergiciel (*middleware*) dédié à la construction et au déploiement d'applications de l'informatique ubiquitaire. Elle permet de couvrir le cycle de développement logiciel d'une application de l'informatique ubiquitaire. Elle est composée d'un langage de haut-niveau de description d'architecture nommé DiaSpec, d'un générateur de code nommé DiaGen, d'un simulateur 2D d'environnements ubiquitaires nommé DiaSim [BJC09], et d'une infrastructure logicielle permettant de faire communiquer des entités via divers protocoles. DiaSpec sert à spécifier des interfaces d'entités et leurs modes d'interactions, décrivant les ressources qu'elles peuvent fournir (les informations qu'elles peuvent capter et leurs fonctionnalités).

A partir d'une spécification de taxonomie et d'architecture d'application, un support de développement logiciel est généré, facilitant l'implémentation des fonctionnalités des entités et d'applications respectant une architecture spécifique. DiaSpec et son générateur permettent de s'abstraire des technologies propres aux systèmes distribués, et notamment des mécanismes permettant aux entités d'interagir via un réseau. DiaSuite intègre en outre un mécanisme de découverte d'entités, permettant de gérer l'apparition de nouvelles entités. Ce mécanisme est assorti d'un langage de requêtes pour interroger l'environnement lorsque des entités d'une classe donnée sont créées ou disparaissent. Le lecteur pourra se référer à [CBLC09] pour une explication détaillée de l'approche générative de DiaSuite.

5.7.2.2 Compilation

Pour compiler les programmes Pantagruel, notre stratégie consiste à tirer profit de DiaSuite en traduisant une taxonomie écrite en Pantagruel en spécification DiaSpec. Plus spécifiquement, les interfaces d'interaction fournies par Pantagruel pour communiquer avec les entités sont traduites en composants DiaSpec. Les attributs volatile sont traduits en sources d'événements. Ainsi, un événement publié par le composant qui le déclare porte la valeur d'un attribut volatile qui est alors mis à jour dans la mémoire de Pantagruel. Les méthodes sont traduites en commandes DiaSpec. Celles-ci peuvent ensuite être invoquées par l'infrastructure de DiaSuite via un composant *noyau* qui fournit une interface de communication entre les composants DiaSpec. Les attributs write sont traduits en attributs d'entités pouvant être accédés et modifiés par des commandes de type *getter* et *setter* générés par le générateur de DiaSuite.

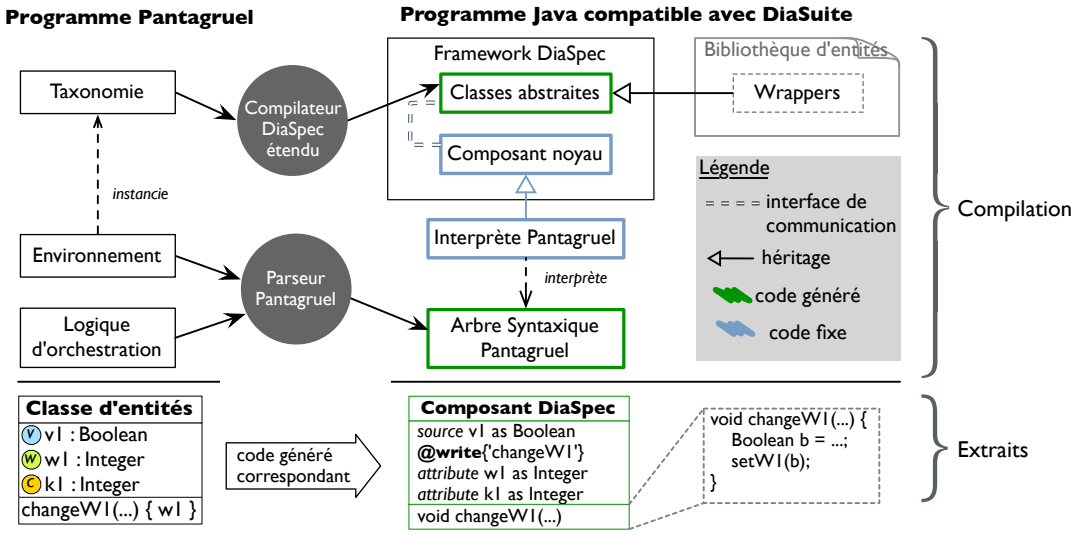

FIG. 5.21: Compilation de Pantagruel vers DiaSuite et extraits de la compilation

Le processus de compilation de Pantagruel est résumé par la figure 5.21 (partie supérieure). La première étape consiste à traduire une description d'environnement en Pantagruel vers une description DiaSpec (*cf.* figure 5.21 – partie inférieure). Une classe d'entités est compilée en

une déclaration de composant DiaSpec, faisant correspondre les modes d'interaction propres à Pantagruel à ceux de DiaSpec (basés sur des événements et commandes). Une fois cette correspondance effectuée, la description est passée à DiaGen, générant un cadre de programmation ou *framework* dédié à la taxonomie. Ce framework inclut un support de programmation pour les événements et les commandes. Ensuite, les règles d'orchestration de Pantagruel sont données à un parseur qui les traduit en arbre syntaxique abstrait en Java. Ces règles sont interprétées en Java, au moyen d'un composant spécifique héritant du composant noyau de DiaSpec. Ce composant contient l'infrastructure nécessaire pour manipuler les commandes et traiter les événements. L'implémentation d'une application Pantagruel est complétée en fournissant des *wrappers* qui implémentent les fonctionnalités des entités, c'est-à-dire l'interface d'interaction requise par une classe d'entités Pantagruel. Plus précisément, elle correspond à une classe Java implémentant (1) les méthodes recevant des données de certaines entités et les publiant sous forme d'événements, (2) les méthodes permettant d'accéder aux fonctionnalités des entités et (3) les méthodes permettant d'accéder et de modifier leurs attributs write.

Pour générer les wrappers, nous permettons à la taxonomie d'être annotée avec des déclarations d'implémentation. Ces déclarations permettent de faire correspondre des éléments de la taxonomie avec des composants DiaSpec existants et leurs modes d'interaction (événements ou commandes). Lors de la compilation de la taxonomie, la spécification DiaSpec est combinée à des composants DiaSpec existants. Ce mécanisme nous permet d'exploiter les spécifications existantes pour lesquelles des implémentations de wrappers existent.

5.7.2.3 Adaptation de l'éxecution

Une fois la compilation terminée, une application Pantagruel est déployée dans l'environnement ubiquitaire cible. Spécifiquement, l'interprète Pantagruel est lancé, découvrant l'ensemble des entités déployées dans l'environnement. Il intègre un *thread* Java qui implémente la boucle d'itération infinie de l'algorithme 1. A chaque itération, les règles sont exécutées en utilisant les mémoires $\sigma_{t\text{-}1}$ et σ_t et en modifiant la mémoire tampon σ_{t+1}. L'interprète Pantagruel est notifié dès que des événements sont publiés par les entités, et met à jour les valeurs des attributs volatile correspondants dans la mémoire tampon.

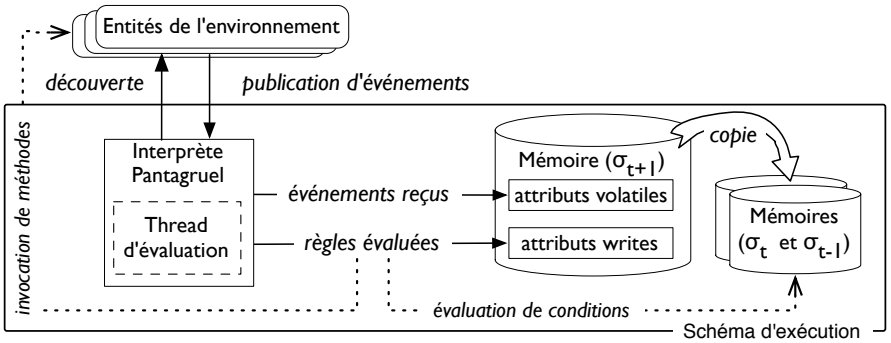

FIG. 5.22: Schéma d'exécution de Pantagruel dans DiaSuite

5.7.2.4 Simulation avec DiaSim

Pour faciliter le test des programmes Pantagruel, nous exploitons également DiaSim, le simulateur intégré à la suite d'outils DiaSuite, couplé à un logiciel de rendu 2D. Il permet de définir un environnement ubiquitaire graphique équipé d'entités spécifiées par le langage de description DiaSpec. Nous avons étendu DiaSim afin de lui permettre de charger une taxonomie Pantagruel. DiaSim permet d'ajouter visuellement des entités à l'environnement ubiquitaire, comme le montre la figure 5.23. L'exécution d'une application Pantagruel dans un environnement composé d'entités simulées est visualisée dans DiaSim. La simulation est commandée par des stimuli pouvant être envoyés depuis l'interface de rendu de DiaSim (déplacement d'agents visuels, entrées de valeurs numériques).

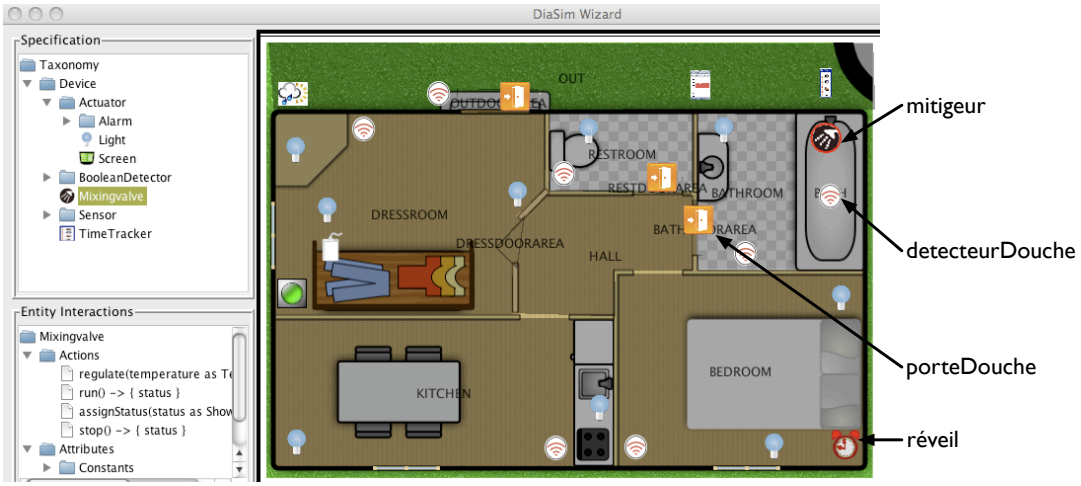

FIG. 5.23: L'éditeur DiaSim : la taxonomie (gauche) et l'éditeur d'environnement (droite)

5.8 Synthèse

Dans ce chapitre, nous avons présenté Pantagruel, un langage dédié qui permet de développer des programmes pour une grande variété de champs d'applications. Le langage de taxonomie permet de décrire, sous forme d'un ensemble de classes, un champ d'applications qui sert ensuite de paramètre au langage d'orchestration visuel. La taxonomie expose les interfaces d'interaction des entités, qui englobent trois sortes d'information de contexte, couvrant un large périmètre de données caractérisant un environnement. Nous argumentons l'expressivité apportée par ces interfaces d'interaction dans le chapitre 8. Le langage d'orchestration utilise ensuite ces interfaces pour développer facilement des règles d'orchestration centrées sur les entités. Nous avons mesuré son accessibilité auprès de programmeurs novices, et nous en reportons les résultats dans le chapitre 8.

La sémantique de Pantagruel repose sur cinq concepts clés focalisés sur la logique d'orchestration des entités. Elle a servi de base pour implémenter le compilateur vers une plate-forme dédiée au développement et au déploiement d'applications ubiquitaires. Nous couvrons ainsi le

cycle de développement de l'étape de synthèse de la méthodologie, que nous pouvons maintenant illustrer, ce qui est l'objet du chapitre suivant.

En plus de clarifier la signification du langage, la sémantique peut être implémentée directement dans un langage fonctionnel comme OCaml, ce que nous avons fait pour pouvoir tester rapidement les premiers programmes Pantagruel. Enfin, la sémantique constitue également une base fondamentale pour définir et vérifier des propriétés sur les programmes Pantagruel. Nous donnons un aperçu de son utilité dans le chapitre 7.

6 Orchestration d'entités guidée par la méthodologie

Contexte Nous avons présenté dans le chapitre 5 le langage Pantagruel qui outille notre méthodologie. La sous-couche langage de taxonomie permet de modéliser, sous la forme d'entités, les objets identifiés à l'issue de la première phase de la méthodologie (la phase d'analyse). La sous-couche langage d'orchestration permet de développer des applications dont les entités sont les briques de base et servent de passerelle vers la phase de synthèse.

Contribution Dans ce chapitre, nous appliquons la phase de synthèse au cas d'Henrick (voir chapitre 4). Dans cette phase, il s'agit d'écrire des applications d'orchestration à l'aide des entités identifiées lors de la phase d'analyse. Cet exemple complète ainsi l'application de la méthodologie et illustre la corrélation entre ses deux phases, en montrant comment la décomposition des buts guide la programmation des applications. Les applications décrites dans ce chapitre ont fait l'objet d'une démonstration [DC10].

Plan du chapitre Dans la section 6.1 nous appliquons l'étape de synthèse au cas d'Henrick. Dans la section 6.2 nous présentons les résultats de l'évaluation de la méthodologie effectuée auprès d'éducateurs spécialisés. Enfin, en guise de conclusion, nous proposons quelques perspectives dans la section 6.3.

6.1 De la méthodologie au langage (applications pour Henrick)

Après un rapide rappel du processus global de la méthodologie sur le cas d'Henrick, nous présentons le développement de la phase de synthèse.

6.1.1 Rappel de la démarche

Lors de la phase d'analyse, nous avons décomposé le but "démarrer la journée" en quatre sous-buts, eux-mêmes encore décomposés pour finalement donner lieu à des objets. Il s'agit lors de la phase de synthèse de développer des applications réalisant les sous-buts (voir figure 6.1).

Afin de développer ces applications, les objets sont d'abord modélisés sous forme d'entités à l'aide du langage de taxonomie de Pantagruel. Cette taxonomie est ensuite instanciée pour former un environnement, par exemple, l'appartement d'Henrick. Les interfaces d'interaction de ces entités contiennent les informations exprimées dans les buts élémentaires ; elle mettent ainsi à disposition les informations de contexte et les fonctionnalités nécessaires pour développer des applications d'orchestration.

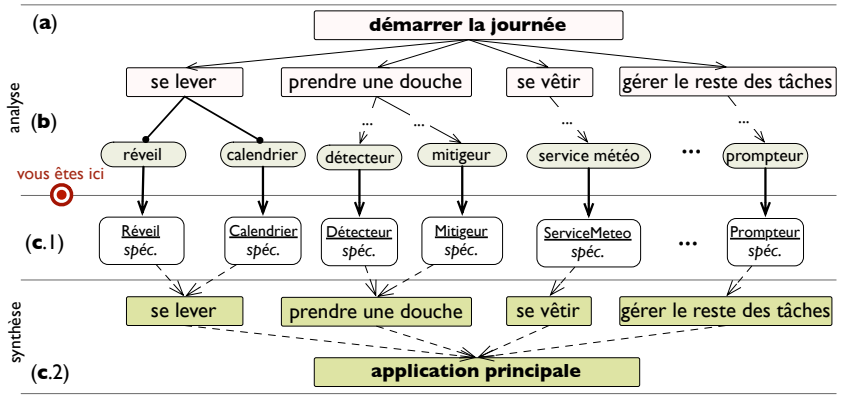

FIG. 6.1: Rappel de la démarche

Dans la section suivante, nous présentons la mise en œuvre des étapes (c.1) et (c.2) de la figure 6.1.

6.1.2 Le cas d'Henrick : synthèse

Nous présentons un extrait de la taxonomie d'entités pouvant équiper l'appartement d'Henrick, puis nous montrons comment leur orchestration permet de réaliser les quatres sous-buts identifiés dans la figure 6.1.

6.1.2.1 Taxonomie

Chaque objet nécessaire au but "démarrer la journée" est modélisé par une entité de la taxonomie de la figure 6.2, dont les attributs et les méthodes sont définis par des types de données pertinents pour le champ d'applications visé.

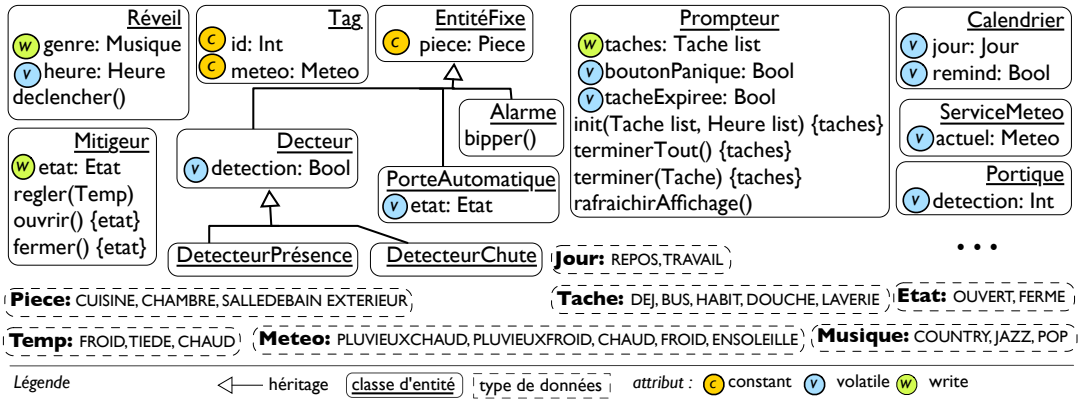

FIG. 6.2: Extrait de la taxonomie du champ d'applications d'assistance

Par exemple, nous avons défini la classe **Réveil** avec un attribut write, nommé genre pour désigner le genre de musique que le radio-réveil peut jouer. Cet attribut est de type Musique, donnant le choix entre JAZZ, COUNTRY et POP. Nous avons aussi défini la classe **Prompteur** qui correspond à l'objet prompteur de tâches, intégrant un écran permettant d'afficher, via la méthode rafraîchirAffichage, les tâches restant à effectuer par Henrick.

Cette taxonomie peut ensuite être instanciée pour décrire l'appartement d'Henrick. La figure 5.23 du chapitre précédent (page 85) est une définition de cet environnement avec l'éditeur DiaSim. La plupart des classes sont instanciées une seule fois, comme le **réveil** et le **mitigeur** pour les classes **Réveil** et **Mitigeur**. En revanche, les classes **Tag** et **PorteAutomatique** sont instanciées autant de fois qu'il y a de tenues vestimentaires et de portes nécessitant un contrôle.

6.1.2.2 Applications d'orchestration

Nous pouvons à présent construire les règles d'orchestration. Pour commencer, on peut choisir de définir un scénario pour chaque sous-but initial (issu du premier pas de décomposition du but principal). On place ensuite dans l'éditeur Pantagruel toutes les entités correspondant aux objets connectés au but initial. Puis on réexamine chaque but élémentaire issu du but afin d'identifier les informations et/ou les fonctionnalités permettant de le réaliser. Un but élémentaire peut donc se traduire en une condition, en une action, ou les deux, formant alors une règle d'orchestration complète. Nous détaillons ici la traduction des quatre buts d'Henrick en applications. Toutes ces applications sont définies dans la figure 6.3. Ainsi, chaque règle d'orchestration à laquelle nous faisons référence dans la suite est contenue dans cette figure.

Notons que nous avons choisi de ne pas alourdir ces applications pour faciliter la compréhension de la démarche de développement. Ces applications nécessitent d'importantes précisions avant de pouvoir être déployées dans un environnement réel. En effet, l'assistance à la personne est en pratique plus complexe que ce que nous avons présenté dans la description du cas d'Henrick.

Se lever Ce but implique deux entités : un calendrier et un réveil. Il s'agit d'associer un genre musical à l'activité de la journée. Pour cela, nous ajoutons deux actionneurs dans la section **reveil** : un pour configurer la musique (par exemple à JAZZ ou à COUNTRY), un pour la faire jouer (en utilisant la méthode déclencher). Ces deux actions sont effectuées séquentiellement (règles R1.1 et R1.2). Elles sont déclenchées lorsque le calendrier publie un événement indiquant l'activité de la journée. Cette information provient d'un capteur dans la section **calendrier** testant la nature de la journée (capteur "jour **is** TRAVAIL"). Nous supposons que l'événement est publié précisément à l'heure spécifiée dans le calendrier : le moment de la publication est abstrait par la nature volatile de l'attribut jour).

Prendre une douche Ce but consiste à réaliser des actions sur le mitigeur et implique les entités suivantes : un détecteur de mouvement, une porte automatique et un mitigeur. Le but élémentaire "aller sous la douche" issu de sa décomposition repose sur le détecteur de mouvement et la position ouverte ou fermée de la porte de la salle de bain. Il en résulte les règles R2.1 et R2.2. Pour limiter les risques de panique lorsque la douche est déclenchée automatiquement,

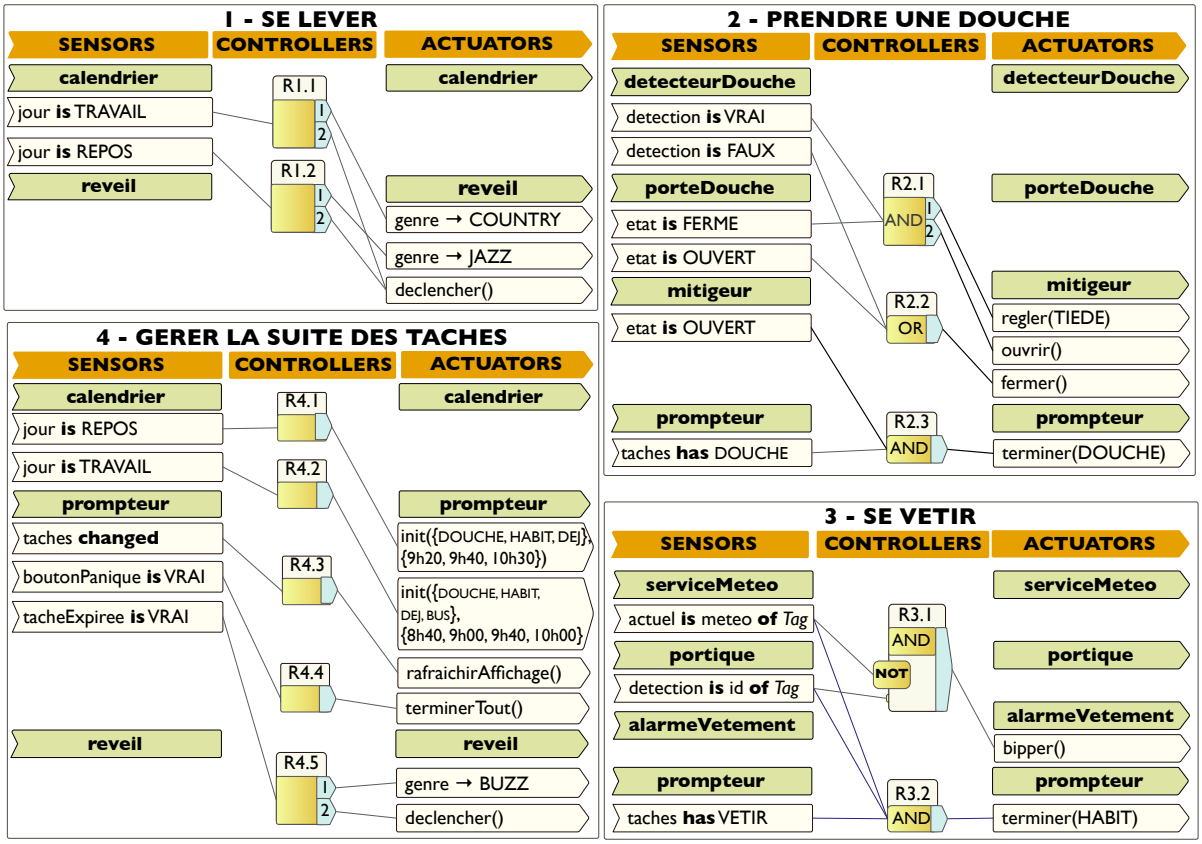

FIG. 6.3: Des applications d'orchestration pour l'appartement d'Henrick

on peut supposer que le mitigeur peut également être contrôlé manuellement par Henrick. Enfin, la règle R2.3 supprime la tâche DOUCHE de la liste des tâches à effectuer lorsque le mitigeur a été ouvert. Pour définir une application plus réaliste, l'ajout d'entités et de règles supplémentaires peut être envisagé, par exemple pour chronométrer le temps passé sous la douche et éteindre le mitigeur pour qu'Henrick puisse se savonner.

Se vêtir Ce but requiert le prompteur de tâches, le portique, le service météo, et une alarme pour alerter Henrick en cas de choix de vêtements inadaptés à la météo actuelle. Les tags doivent être filtrés selon des contraintes spécifiques liées aux informations fournies par le service météo. Pour réaliser ce filtrage, l'attribut meteo de ces tags est comparé à la météo actuelle. Ainsi, un filtre est défini avec deux conditions de contexte "detection **is** id **of** Tag", dans la section **portique**, et "actuel **is** meteo **of** Tag" dans la section **serviceMeteo**. Ces deux conditions sont combinées dans la règle R3.1 par le contrôleur AND et un opérateur NOT, conditionnant le déclenchement de l'alarme alarmeVetement si Henrick quitte la pièce en portant une tenue inappropriée. Enfin, la règle R3.2 supprime la tâche HABIT de la liste des tâches du prompteur en appelant l'action

terminer sur celui-ci lorsqu'Henrick a choisi la bonne tenue vestimentaire.

Gérer la suite des tâches Ce but nécessite un calendrier, un prompteur de tâches, et un réveil. Le calendrier permet d'initialiser le prompteur avec les tâches qu'Henrick doit effectuer conformément à l'activité annoncée pour la journée. Son initialisation inclut également l'heure à laquelle ces tâches devraient être terminées. L'attribut tâches est mis à jour à chaque fois qu'une tâche est complétée. Les règles R4.1 et R4.2 contrôlent le prompteur selon ces spécifications. Lorsque l'attribut tâches change, la liste des tâches affichées par le prompteur est rafraîchie (règle R4.3). Rappelons que cet attribut est modifié par les applications précédentes.

Afin d'éviter qu'Henrick ne panique lorsque le temps estimé pour une tâche est dépassé, le prompteur de tâches intègre un bouton de panique, qu'Henrick peut utiliser pour annuler le processus de suivi des tâches. Ce bouton est modélisé par l'attribut volatile boutonPanique ; l'annulation est alors effectuée par la méthode terminerTout, appelée dans la règle R4.4. Cette méthode supprime toutes les tâches enregistrées dans le prompteur.

6.1.3 Un exemple d'exécution

Nous avons déployé toutes ces applications dans le logiciel de simulation DiaSim. Celles-ci ont été testées en simulant des comportements d'Henrick au moyen d'un agent se déplaçant dans un environnement 2D. Une trace d'exécution de l'application "prendre une douche" est modélisée par la figure 6.4.

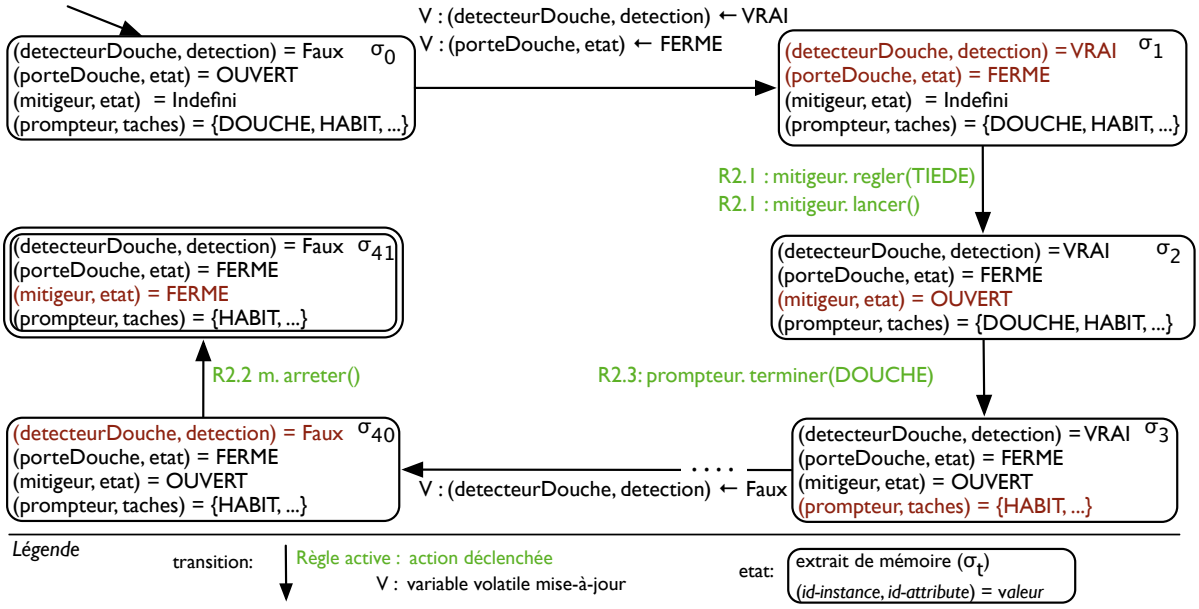

FIG. 6.4: Exécution de l'application "Prendre une douche"

Imaginons qu'Henrick entre dans la salle de bain et que la porte de la douche se ferme derrière

lui comme indiqué par les attributs volatiles dans le passage de la mémoire σ_0 à σ_1. Ces attributs changent l'environnement d'Henrick, et provoquent le déclenchement des règles R2.1 puis R2.2 agissant sur le mitigeur. La règle R2.2 modifie l'état du mitigeur (mémoire σ_2), qui passe à VRAI. Ce changement d'état provoque le déclenchement de la règle R2.3 qui supprime la tâche DOUCHE du prompteur, donnant lieu à une mémoire mise à jour, indiquée par σ_3. Au moment où Henrick sort de la douche, l'attribut detection du détecteur de présence detecteurDouche passe à FAUX, provoquant le déclenchement de la règle R2.2 qui arrête le mitigeur.

6.2 Evaluation de la pertinence de la méthodologie outillée

Pour évaluer la pertinence de notre méthodologie, nous avons mis à contribution deux éducatrices spécialisées. En particulier, nous souhaitions vérifier que la méthodologie dirigée par les buts élémentaires, correspond au processus d'élaboration de stratégies éducatives par l'éducateur spécialisé. Au cours de deux entrevues individuelles, nous avons d'abord présenté nos objectifs d'évaluation aux éducatrices. Puis nous leur avons demandé d'appliquer la méthodologie sur un profil de personne déficiente, préalablement défini à l'aide d'un questionnaire standard fourni par un médecin. Afin de poser le contexte de cette évaluation, nous détaillons dans cette section le profil professionnel de ces deux éducatrices et le support matériel des entrevues. Nous détaillons ensuite la démarche suivie pour mener cette évaluation. Cette démarche, consistant en une session individuelle avec chaque éducatrice, nous a permis de valider la phase d'analyse de la méthodologie ; nous avons également pu recueillir des retours intéressants sur l'adéquation de Pantagruel à leur métier, et plus généralement sur leurs attentes vis-à-vis d'outils permettant de construire des applications d'assistance à la personne.

6.2.1 Participants

Anne et Catherine sont deux éducatrices spécialisées âgées de 30 ans. Anne exerce dans différents domaines de l'assistance : elle accompagne des enfants en situation familiale instable ainsi que des adolescents et des adultes déficients intellectuels combinant parfois plusieurs handicaps physiques. A cette fin, elle détermine régulièrement des stratégies pour les aider à acquérir une autonomie dans leurs activités quotidiennes. Elle interagit pour cela avec des membres du corps médical et possède une connaissance des techniques d'évaluation des déficiences d'une personne. Catherine est spécialisée dans l'accompagnement des adolescents souffrant de troubles psychologiques et d'adultes déficients intellectuels et physiques. Elle interagit également avec des médecins.

Toutes deux interviennent dans l'assistance aux activités des personnes, qu'il s'agisse d'activités de base comme le lever, l'habillement, ou d'activités impliquant des interactions sociales, comme l'usage des transports en commun ou les activités proposées par les foyers occupationnels.[1]

[1] Les foyers occupationnels sont des structures d'accueil de jour aux personnes déficientes, dont l'objet est de les accompagner dans la pratique d'activités ludiques, éducatives et sociales

6.2.2 Support matériel

Le support de l'évaluation est le suivant :

- un entretien préliminaire : il a permis de collecter les compétences des éducatrices rapportées ci-dessus, ainsi que leurs compétences à l'utilisation de logiciels ;
- un questionnaire standard prérempli : il énumère les aptitudes d'une personne déficiente fictive basée sur le personnage d'Henrick, notre cas d'étude principal dans cette thèse. Ces aptitudes, présentées sous forme d'assertions associées à une réponse Oui/Non, sont groupées selon les buts pour lesquels elles sont requises. En particulier, nous avons sélectionné les buts suivants : se lever, faire sa toilette, se vêtir et quitter l'appartement. Par exemple, le but "faire sa toilette" comprend les aptitudes "se brosser les dents", "prendre une douche", et le but "se vêtir" comprend l'aptitude "mettre les vêtements dans le bon ordre". Les éducatrices étaient invitées à interpréter ces aptitudes pour choisir une décomposition adaptée au profil de la personne.
- une collection de technologies d'assistance existantes ou disponibles dans un futur proche : ces technologies sont présentées sous la forme d'une bibliothèque d'objets explorés au moyen d'une petite interface logicielle. Chaque technologie correspond à un objet associé à une représentation visuelle et à une description informelle de son utilisation, ainsi que d'une liste d'objets apparentés (par exemple, une étiquette RFID est liée à un lecteur de badge portatif et à un portique RFID).
- le prototype d'éditeur visuel Pantagruel : nous l'avons paramétré par la taxonomie de la figure 6.2 afin de présenter une utilisation de Pantagruel pour l'assistance à la personne.

6.2.3 Sessions d'évaluation

Avant d'évaluer notre méthodologie, nous avons d'abord interrogé Anne et Catherine sur leurs connaissances de l'outil informatique afin d'adapter la présentation du contexte d'évaluation. Toutes deux sont familières avec les suites logicielles de bureautique, telles que Microsoft Word, Excel, les clients de messagerie instantanée et de courrier électronique.

Nous avons ensuite expliqué l'objectif de notre évaluation et leur avons demandé d'examiner le questionnaire décrivant le profil d'une personne. Enfin, nous leur avons présenté la bibliothèque d'objets, et les avons invitées à poser toute question concernant l'usage de ces objets ou à proposer d'autres objets issus de leur imagination. La présentation du profil et l'exploration de la bibliothèque ont duré 25 minutes. Nous avons ensuite fixé la durée de l'étape de décomposition à 45 minutes, au cours de laquelle nous avons observé le procédé de décomposition suivi par les éducatrices. Chaque éducatrice a dû choisir et décomposer un but particulier parmi ceux proposés dans le questionnaire-profil. Pour les guider dans cette démarche, nous leur avons suggéré de se poser la question suivante à chaque pas de décomposition : ce but peut-il être directement associé à un objet élémentaire ? Si oui, elles y attachent l'un des objets de la bibliothèque, et sinon, elles poursuivent la décomposition. Nous ne sommes intervenus que lorsqu'elles ont demandé de l'aide. A suivi une discussion de 20 minutes sur le résultat de cette étape, permettant d'en saisir certains choix spécifiques, les difficultés rencontrées et leur cause. Nous avons finalement consacré 30 minutes à la présentation de Pantagruel et son utilisation pour outiller l'étape de décomposition, afin d'avoir leur point de vue sur l'adéquation de l'outil

à leur profession.

6.2.4 Analyse des résultats

L'une des questions de cette évaluation était de savoir si les participantes parviennent à envisager facilement la manière dont les technologies d'assistance peuvent les aider pour renforcer le support actuel apporté aux personnes déficientes. Nous avons donc accordé une importance particulière à leur façon d'utiliser les objets pour accomplir les buts élémentaires. Nous détaillons ce point après avoir rapporté les stratégies de décomposition adoptées par chacune.

6.2.4.1 Stratégies de décomposition

Anne a choisi deux buts : le premier est "se lever" et le second est "faire sa toilette" ; Catherine a choisi le second but. Parmi les quatre buts proposés, toutes deux ont désigné l'activité de toilette comme la plus importante parmi celles nécessitant une assistance.

Anne a naturellement suivi la démarche de décomposition suggérée avant l'évaluation. Catherine s'est sentie plus à l'aise en effectuant une décomposition détaillée en buts élémentaires dès le premier pas de décomposition. Plus précisément, chaque but correspondait à des tâches élémentaires qu'une personne doit effectuer chronologiquement pour réaliser le but "faire sa toilette". Bien qu'étant à un seul niveau, cette décomposition se restructurait naturellement en sous-buts. A l'issue de cette étape de l'évaluation, toutes deux ont indiqué que la méthodologie est indiscutablement proche de leur démarche habituelle pour élaborer des stratégies éducatives.

6.2.4.2 Objets élémentaires et applications d'assistance

Anne est familière avec les technologies de la domotique (capteurs, étiquettes RFID) et avec l'idée d'interaction à distance entre les entités, par exemple via Internet ; il lui a paru naturel de faire correspondre des buts à des objets. En revanche, Catherine était au départ perplexe lorsqu'il a fallu associer les sous-buts aux objets, car elle ne parvenait pas à se sentir à l'aise avec les concepts de domotique et d'interaction distante. Nous lui avons alors suggéré quelques applications basiques utilisant un détecteur de présence, des étiquettes électroniques et des scanneurs d'étiquettes, en soulignant la possibilité de les contrôler à distance pour les faire interagir. Un déclic s'est produit et elle est ensuite parvenue sans hésitation à connecter à chaque but élémentaire des objets appropriés.

Les deux éducatrices ont emprunté une démarche similaire pour faire cette connexion : après avoir sélectionné quelques objets, elles ont formulé des phrases décrivant la manière dont ceux-ci peuvent assister un but. Par exemple, pour le sous-but "se brosser les dents", Anne a imaginé une salle de bain équipée d'un haut-parleur, d'un lavabo surmonté d'une tablette RFID (objet qu'elle a inventé), ainsi que des diodes de localisation pour chaque élément nécessaire à cette activité. Les étiquettes RFID, fixées sur la tablette, seraient capables de détecter si un élément (la brosse à dent, le gant et le verre de rinçage) est posé ou retiré. La description d'une application était dirigée par des suppositions sur le comportement de la personne : si la personne prend le verre avant la brosse à dent, la voix du haut-parleur lui rappelle de prendre la brosse à dent d'abord. Après l'activité de brossage, la diode associée au verre clignote pour indiquer la tâche de rinçage.

Nous avons également observé que les éducatrices ont substitué naturellement les objets aux supports d'assistance habituels : les diodes ont remplacé les images, le haut-parleur a tenu lieu de rappel oral par l'éducateur lorsque la personne oublie une tâche. Pour assister l'ordonnancement des tâches, un prompteur de tâches était proposé dans la bibliothèque. Cet objet était omniprésent dans les applications proposées par les éducatrices. Par exemple, Catherine a utilisé le prompteur de tâches pour les activités de brossage dentaire et de toilette faciale. En supposant que les compétences de la personne déficiente le permettent, Catherine a proposé de lui laisser contrôler le prompteur.

6.2.4.3 Discussions sur Pantagruel

Nous avons finalement fait une démonstration de Pantagruel pour programmer quelques unes des applications envisagées à l'étape précédente. Le propos de cette démonstration était de recueillir leurs impressions sur l'accessibilité de l'outil, et leurs attentes vis-à-vis d'un tel outil.

Bien que l'outil leur ait paru accessible en tant que non-programmeuses (Catherine a pu créer quelques règles cohérentes pour un des buts de l'évaluation), leur première impression est que l'outil n'est pas adapté à la démarche des éducatrices. En particulier, Anne a indiqué que "*l'approche de développement de Pantagruel est centrée sur les objets, alors que celle de l'éducateur spécialisé est centrée sur l'action à réaliser*". Pour améliorer l'outil, d'intéressantes suggestions ont été formulées par les éducatrices, que nous rapportons maintenant.

Programmation par configuration ou modification Les deux éducatrices ont proposé un processus de programmation basé sur des modèles de règles d'orchestration prédéfinies selon un catalogue de buts usuels. L'idée est que l'éducateur choisisse un but du catalogue pour accéder à un modèle de règles, qu'il modifie ou configure pour l'adapter aux besoins d'un individu. Anne a ainsi proposé que, plutôt que de créer les règles en partant de zéro, l'on supprime des entités d'un programme Pantagruel prédéfini, ou l'on complète des règles existantes avec de nouvelles conditions ou actions, selon que la stratégie éducative doit être assouplie ou renforcée. Anne a notamment trouvé pratique l'idée que la suppression d'une entité entraîne celle de tous les éléments l'impliquant. Une technique de programmation similaire, basée sur la composition et la modification, a été proposée par S. Carmien dans sa thèse sur les technologies d'assistance [Car02].

Pour une représentation visuelle adaptée L'éditeur de Pantagruel fournit une aide contextuelle à l'utilisateur ainsi qu'une représentation sommaire en langage naturel de chaque règle, centrée sur les entités (nous reviendrons sur cette représentation dans le chapitre 8, au moment de parler de l'accessibilité du langage). Cette représentation a inspiré les deux éducatrices qui ont alors suggéré une interprétation des règles centrée sur les actions à réaliser par la personne. Par exemple, pour l'un des programmes présentés dans la démonstration (figure 6.3 de la page 90), Anne a proposé la traduction suivante : lorsqu'Henrick va sous la douche, la porte de la salle de bain doit être fermée et la douche réglée à la bonne température. Cette formulation masque les attributs des objets (par exemple, le statut de la porte), mais également

l'objet détecteur de présence placé sous la douche. Pour permettre une telle traduction, un travail important est nécessaire concernant l'accessibilité de la taxonomie et son intégration totale dans un processus de développement orienté buts. Pour cela, les objets pourraient être mis à disposition par un moteur de recherche centré sur les buts plutôt que sur des familles d'objets. Ensuite, l'interprétation des règles pourrait reposer sur les informations de buts fournies par ce moteur de recherche en utilisant les techniques de représentation de connaissances basées sur les ontologies [ISM97]. Ce travail peut faire l'objet d'une nouvelle direction de recherche pour améliorer l'accessibilité de Pantagruel.

Pour un programme fiable Catherine et Anne ont signalé la nécessité de développer des programmes fiables. Notamment, au cours de la démonstration, toutes deux ont évoqué la question du conflit entre deux règles : par exemple, à propos des règles d'orchestration du mitigeur, Anne a posé la question : "*que se passe-t-il lorsqu'une règle ferme le mitigeur et qu'une autre l'ouvre au même moment ?*". Ces remarques nous ont poussé à implémenter la vérification de quelques propriétés que nous exposons dans le chapitre 7.

Bilan

Cette évaluation montre que l'étape d'analyse de notre méthodologie correspond à la démarche usuelle d'élaboration de stratégies éducatives. En particulier, elle montre l'importance de la décomposition pour assister une personne déficiente dans les tâches les plus élémentaires de ses activités quotidiennes. Bien que les éducatrices soient sceptiques quant à la représentation visuelle de Pantagruel, elles sont intéressées par l'intégration d'un tel outil dans leur démarche professionnelle. Toutefois, cet outil a aussi soulevé de nombreuses questions d'ordre éthique ou pratique, comme le risque d'obtenir un effet inverse à celui escompté, c'est-à-dire l'acquisition par la personne déficiente d'une dépendance aux supports technologiques. Ces questions ont également mis en évidence la nécessité de faciliter l'adaptation des applications par un expert-métier. En résumé, les discussions que nous avons eues avec les éducatrices spécialisées font entrevoir de nouvelles directions de recherche très intéressantes sur l'accessibilité de la programmation.

6.3 Synthèse

Dans le chapitre 4 nous avions présenté la méthodologie outillée et nous en avions illustré la phase d'analyse. Dans ce chapitre nous en avons illustré la phase de synthèse, montrant ainsi le processus complet de développement d'applications à l'aide de la méthodologie et de Pantagruel. Nous avons également évalué la pertinence de cette démarche auprès d'éducateurs spécialisés. Cette évaluation a montré que l'étape d'analyse est adaptée à la démarche des éducateurs spécialisés, qui se sont montrés particulièrement intéressés par notre méthodologie. Cependant, bien qu'étant a priori accessible à des utilisateurs finaux, Pantagruel reste un langage encore trop programmatique pour des experts-métiers.

Il reste plusieurs points à traiter dans la méthodologie. D'une part, la connexion entre la phase d'analyse et la phase de synthèse, consistant à modéliser les objets, est une activité

difficile. En pratique, elle requiert l'intervention d'un expert spécialisé dans les composants matériels et logiciels d'un champ d'applications afin de développer les bonnes interfaces pour accéder à ces composants. Pour cela, il est nécessaire de mettre en place une collaboration entre celui-ci et l'expert-métier. Dow *et al.* ont proposé des principes de conception d'applications ubiquitaires mettant en évidence la nécessité de faire collaborer les différents acteurs du cycle de développement d'une application [DSLL06].

Il y a également un travail important à continuer sur l'accessibilité de la programmation pour l'expert-métier. En particulier, Pantagruel impose une représentation visuelle qui, si elle est adaptée au prototypage d'applications, ne reflète pas forcément la manière dont un expert-métier formule une solution. Pantagruel pourrait néanmoins servir de représentation pivot, à partir de laquelle on dériverait différentes formulations selon le champ d'applications et le mode de pensée de l'expert-métier (par exemple, par une traduction centrée sur les actions pour l'éducateur spécialisé).

7 Vers une approche de programmation dirigée par les vérifications

Contexte Une des caractéristiques du langage Pantagruel est de coupler une sous-couche langage de taxonomie avec une sous-couche langage d'orchestration paramétré par une taxonomie. Le développement d'applications d'orchestration est ainsi rigoureusement dirigé par les descriptions d'entités, ce qui permet aux experts-métiers de définir des règles d'orchestration en utilisant les termes de son métier. Cependant, il reste à assurer à l'expert-métier que les applications d'orchestration sont fiables et qu'elles se comportent conformément à ses attentes.

Contribution Ce chapitre propose une approche de programmation guidée par l'expression d'invariants sur des applications d'orchestration Pantagruel. Les invariants peuvent être exprimés dans le langage Pantagruel. Nous présentons également une définition de la propriété de non-interférence qui est un concept clé du langage d'orchestration. Ces propriétés sont fondées sur les concepts de Pantagruel formalisés par sa sémantique, ce qui permet alors de mettre en oeuvre des techniques de vérification. Ce travail est le résultat d'une collaboration avec Julien Mercadal.

Plan Dans la section 7.1 nous présentons la problématique liée à la vérification des programmes Pantagruel. Dans la section 7.2, nous proposons un angle d'attaque pour aborder cette problématique. Nous donnons ensuite dans la section 7.3 une définition de deux types d'invariants pouvant être exprimés sous forme de règles Pantagruel. Nous y formalisons également la notion de non-interférence. Enfin, la section 7.4 conclut ce chapitre.

7.1 Problématique

Dans Pantagruel, le langage d'orchestration de règles est paramétré par la définition d'une taxonomie. Grâce à l'éditeur visuel de Pantagruel, ce paramétrage guide l'expert-métier dans la définition de règles d'orchestration à l'aide d'un vocabulaire pertinent formulé par la taxonomie. Cependant, ce paramétrage n'est pas suffisant pour convaincre les experts-métier d'adopter un outil de programmation : il faut leur assurer que les programmes qu'ils développent sont fiables et se comportent à l'exécution comme ils s'y attendent. Garantir cette fiabilité est d'autant plus important que ces programmes peuvent avoir un impact important sur la vie des personnes. A notre connaissance, les approches visuelles pour l'utilisateur final ne prennent que partiellement en compte cette exigence : elles se limitent à la détection de conflits. Nous justifions ce propos dans le chapitre 8, dans le cadre de l'informatique ubiquitaire.

Il est difficile de rendre confiant l'utilisateur final, et plus particulièrement l'expert-métier, dans l'écriture des programmes d'orchestration. Cette situation est dûe à deux raisons : d'une part les outils de vérification existants lui sont inaccessibles ; d'autre part la spécificité du paradigme à base de règles favorise l'écriture de programmes non fiables.

Inaccessibilité des outils de vérification Les experts-métier ne disposent pas de moyens pour exprimer des propriétés spécifiques que doivent vérifier les programmes d'orchestration (par exemple, s'assurer que l'eau ne coule pas lorsqu'Henrick fait une chute sous la douche). La nature dédiée et haut niveau de Pantagruel facilite l'utilisation d'outils de vérification existants. En particulier, il existe des techniques reposant sur la logique temporelle [CES86, Lam94] (particulièrement adaptée pour la vérification des systèmes réactifs) ou les assistants de preuve [The, PVS, BLO98]. Cependant ces outils nécessitent d'avoir des connaissances spécifiques pour pouvoir exprimer des propriétés dans leurs formalismes. De plus, les connaissances de l'expert-métier peuvent être indispensables pour identifier les propriétés que les programmes doivent respecter, et plus particulièrement les comportements que ces programmes doivent systématiquement garantir. Le processus de vérification devrait donc pouvoir reposer sur son expertise ; en d'autres termes, l'expert-métier doit être un acteur important dans ce processus.

Manque de fiabilité des programmes à base de règles Les langages à base de règles souffrent d'un inconvénient connu : il est facile d'introduire des conflits entre les règles d'un programme donné. Détecter de telles situations requiert de la part des experts-métier de considérer l'ensemble des contextes possibles (c'est-à-dire les combinaisons de capteurs) dans lesquels les règles d'orchestration peuvent être déclenchées. Cette tâche est compliquée, d'une part parce que les informations de contexte extérieur ne peuvent être prédites, et d'autre part parce qu'un grand nombre d'entités peut être mis en jeu dans les programmes. En conséquence, des analyses avant exécution du programme sont nécessaires pour détecter, dans un ensemble de règles, les risques de conflits et offrir à l'expert une assistance appropriée pour l'aider à les résoudre.

7.2 Proposition

Afin d'améliorer la fiabilité des programmes d'orchestration écrits en Pantagruel, nous proposons à l'expert-métier une approche de programmation guidée par les propriétés. Cette approche s'appuie sur le paradigme à base de règles et sur la taxonomie pour faciliter l'expression de propriétés spécifiques à un programme. Ces propriétés, que nous présentons maintenant, peuvent guider l'expert dans l'écriture de programmes d'orchestration concordant avec ses intentions.

7.2.1 Propriétés spécifiques au programme

Dans ce chapitre, nous nous concentrons sur un type de propriétés spécifiques à un programme : les invariants d'actions. Un invariant est une propriété qui doit être vérifiée quelle que soit l'exécution d'un programme. Nous appelons invariant d'action une propriété qui impose qu'une action soit exécutée dans le programme étant donnée une condition exprimée par ce programme. Les invariants d'action peuvent être vus comme un *motif* particulier de règles

d'orchestration. Ils peuvent être écrits par l'expert-métier et servent ainsi de base à des programmes d'orchestration qui vont les intégrer à chaque fois que les conditions ou les actions de ces propriétés y apparaissent. De cette manière, le processus de vérification est incorporé au cycle de développement des règles d'orchestration et guide les experts tout en augmentant leur confiance vis-à-vis des programmes qu'ils créent.

En informatique ubiquitaire, les invariants d'actions peuvent permettre de s'assurer qu'une application n'est pas intrusive et ne se comporte pas de manière contraire à ce qu'on en attend. Ils peuvent ainsi améliorer la sécurité des personnes dans un environnement, en contraignant l'application à garantir certains comportements dans une situation de danger (un incendie, une intrusion, un accident, ou encore la chute d'une personne).

D'autres types de propriétés liées à la sûreté et à la vivacité des programmes sont envisageables, comme une propriété imposant qu'une règle s'exécute au moins une fois, qu'un état soit atteignable, ou encore qu'une action en précède toujours une autre. Ceci pourra être l'objet d'un travail ultérieur dont les fondations sont posées dans ce chapitre.

7.2.2 Propriétés spécifiques au langage

Dans ce chapitre, nous nous intéressons à la propriété de non-interférence. Ce terme est essentiellement utilisé dans le domaine de la sécurité [GM82], mais nous l'utilisons au sens indiqué par D. Schmidt [Sch86] pour les commandes d'un programme exécutées en parallèle. Cette définition correspond à celle que nous avons énoncée dans le chapitre 5. La propriété de non-interférence, dite spécifique au langage, peut être énoncée indépendamment d'un programme particulier. La dépendance des applications aux interactions extérieures rend délicate la vérification des propriétés spécifiques au langage Pantagruel. Cependant nous pouvons donner au développeur quelques informations permettant de les garantir grâce à la sémantique du langage. Ces informations peuvent à terme être précisées en faisant des suppositions sur l'implémentation des méthodes et sur l'information de contexte externe recueillie par les variables volatiles.

D'autres propriétés des systèmes réactifs concernent Pantagruel comme l'existence d'états stables ou l'absence de cycles dans une application, mais nous ne les abordons pas ici.

7.3 Invariants d'actions et non-interférence

Nous formalisons deux types de propriétés que nous illustrons à l'aide des applications d'orchestration d'assistance à Henrick définies dans la figure 6.3 (page 90) du chapitre précédent.

7.3.1 Invariants d'actions

Les invariants d'actions spécifient des contraintes de comportement sur le programme. Comme pour les règles d'orchestration classiques, on les définit en utilisant les briques de base fournies par le langage de taxonomie. En pratique, ces invariants correspondent à des règles d'orchestration mais se différencient des règles classiques par leur importance dans le processus de développement d'un programme : ils ont un rôle spécifique pour l'utilisateur final car ils permettent de mettre en évidence les règles qu'un programme ou une famille de programmes doit

systématiquement comporter. Ainsi, les invariants peuvent être envisagés comme un point de vue de programmation, où le programme est vu comme un ensemble de contraintes au lieu d'être présenté sous forme d'une solution à un problème. La programmation dirigée par les invariants peut alors s'intégrer dans un processus de développement incrémental, ce qui correspond notamment à l'approche employée par les éducateurs spécialisés.

Nous proposons deux sortes d'invariants d'actions : les invariants restrictifs et les invariants extensifs. Pour simplifier, nous parlons désormais de prédicat au singulier pour désigner les combinaisons de prédicats définies dans une règle Pantagruel.

Invariant restrictif Un invariant restrictif contraint une (ou plusieurs) action – si elle apparaît dans les règles d'un programme Pantagruel – à être appelée *uniquement* si le prédicat qu'il définit est satisfait. La vérification d'un invariant restrictif consiste à sélectionner l'ensemble des règles Pantagruel contenant la ou les actions spécifiées par cet invariant. Le prédicat de chacune de ces règles doit alors impliquer celui de l'invariant. En d'autres termes, ces règles ne doivent être déclenchées que si le prédicat de l'invariant est satisfait. Sinon, on en conclut qu'il n'est pas respecté par le programme.

Invariant extensif Un invariant extensif correspond à une propriété de vivacité : l'action (ou les actions) définie dans l'invariant doit apparaître dans au moins une des règles du programme ; de plus, le prédicat de l'invariant doit impliquer le prédicat de l'une de ces règles. En d'autres termes, il doit exister une règle (1) dont les actions contiennent l'action de l'invariant, et (2) qui est forcément déclenchée si le prédicat de cet invariant est satisfait. On peut donc voir l'invariant extensif comme une règle d'orchestration qui doit obligatoirement être définie dans le programme. S'il n'existe pas de telle règle, alors l'invariant n'est pas respecté par le programme.

7.3.1.1 Deux invariants pour les applications d'Henrick

Nous définissons respectivement dans les figures 7.1 et 7.2 un invariant restrictif et un invariant extensif pour l'application "2 - PRENDRE UNE DOUCHE" de la figure 6.3 (page 90).

Lorsque l'expert-métier souhaite imposer, dans l'application de gestion de la douche, que la douche soit activable uniquement lorsque la porte est fermée (nous avions supposé dans la section 4.2.3 que la porte ne se ferme que de l'intérieur), il peut commencer par définir un invariant restrictif, tel que celui proposé par la règle Re1 (figure 7.1). Pour des raisons de sécurité, l'expert peut en outre imposer que la douche s'arrête (c'est-à-dire que le mitigeur soit fermé) au moins lorsqu'Henrick fait une chute. Pour cela il pourra ajouter un détecteur de chute **detecteurChute** dans la douche, et définir l'invariant extensif Ex1 (figure 7.2), qui exprime cette contrainte.

7.3.1.2 Notations

Pour définir ces deux types d'invariants, nous désignons une règle Pantagruel (d'invariant ou d'orchestration) par le couple $(p,\ A)$: p est une formule booléenne sur les variables (attributs précédés d'un nom d'entité) et A est l'ensemble des actions associées à p. Pour simplifier, nous

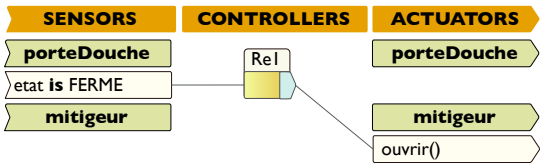

FIG. 7.1: Un invariant restrictif

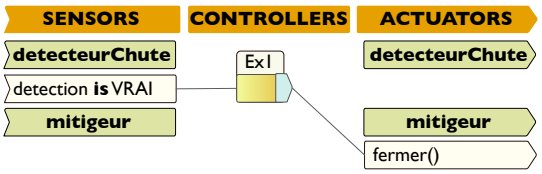

FIG. 7.2: Un invariant extensif

ne prenons en compte que les ensembles d'actions non ordonnés (dans les définitions qui vont suivre, un ensemble d'actions ordonné peut être vu comme une seule action). Nous utilisons de plus les notations suivantes : $s \vdash p$ signifie que p est satisfait avec les valeurs de variables stockées dans la mémoire s ; S_p désigne l'ensemble des mémoires qui satisfont le prédicat p. On note également *Variable* l'ensemble des variables utilisées dans un programme Pantagruel, et $s(v)$ la valeur d'un élément v de *Variable* dans la mémoire s. Nous disons que la mémoire s' "étend" la mémoire s, noté $s \sqsubseteq s'$, si toute variable définie dans s (c'est-à-dire dont la valeur est différente de $\perp$) l'est aussi dans s' avec la même valeur.

$$s \sqsubseteq s' \iff (\forall v \in \mathit{Variable}, (s(v) \neq \perp \Rightarrow (s'(v) \neq \perp \ \mathit{et}\ (s(v) = s'(v))))$$

De la même façon, pour deux ensembles de mémoires S et S', nous notons $S \sqsubseteq S'$ si toute mémoire de S est étendue par une mémoire de S' :

$$S \sqsubseteq S' \iff (\forall s \in S, \exists s' \in S', s \sqsubseteq s')$$

Remarquons alors que si p et p' sont deux prédicats (et respectivement S_p et $S_{p'}$ les mémoires qui les satisfont)[1] :

$$p \Rightarrow p' \iff S_p \sqsubseteq S_{p'}$$

Ce formalisme nous permettrait de baser la vérification d'un programme Pantagruel sur les mémoires du domaine *Store* défini dans la sémantique, par exemple en fournissant si nécessaire à l'utilisateur, les valeurs des variables pour lesquelles une propriété donnée n'est pas respectée.

7.3.1.3 Définitions

Nous définissons à présent les deux types d'invariants à l'aide de ces notations.

Définition 7.3.1 (Invariant restrictif) *Etant donné un ensemble de règles d'orchestration $\mathcal{R}$ et un invariant restrictif $Re = (p, A)$, $\mathcal{R}$ satisfait Re si :*

$$\forall a \in A, \ \mathit{et}\ \forall (p', A') \in \mathcal{R}, \mathit{tel\ que}\ a \in A', \mathit{on\ a}\ S_{p'} \sqsubseteq S_p \quad (p' \Rightarrow p)$$

[1] Les notations "$\forall s, s \vdash p \Rightarrow s \vdash p'$" et "$p \Rightarrow p'$" sont équivalentes.

D'après cette définition, un invariant restrictif est toujours respecté si, dans un programme, il n'existe pas de règle manipulant les actions de cet invariant. Lorsqu'une règle du programme utilise une action de l'invariant mais ne satisfait pas ses prédicats, un moyen de corriger le programme est d'ajouter au prédicat de la règle celui de l'invariant en les combinant avec l'opérateur **AND**. Lorsque les prédicats impliquent des variables write, il est sans doute possible de raffiner la correction, car elles permettent de définir des relations de dépendance entre les règles : on pourrait donc en déduire des relations entre les mémoires satisfaisant leurs prédicats et celles produites par l'exécution des actions. Il serait intéressant d'explorer les techniques de vérification permettant d'offrir des informations de correction optimale.

D'après la définition 7.3.1, l'invariant restrictif Re1 est respecté dans l'ensemble des règles de l'application "2 - **PRENDRE UNE DOUCHE**" (page 90). En effet, seule la règle R2.1 appelle l'action ouvrir, et cette règle vérifie le prédicat de Re1 : on a bien (porteDouche.etat **is** FERME $\wedge$ detecteurDouche.detection **is** VRAI) $\Rightarrow$(detecteurDouche.detection **is** VRAI).

Définition 7.3.2 (Invariant extensif) *Etant donné un ensemble de règles d'orchestration* $\mathcal{R}$ *et un invariant extensif* $Ex = (p, A)$, $\mathcal{R}$ *satisfait* Ex *si :*

$$\forall a \in A, \exists (p', A') \in \mathcal{R} \text{ telle que } a \in A', \text{ avec } S_p \sqsubseteq S_{p'} \quad (p \Rightarrow p')$$

Montrer à l'utilisateur qu'un invariant extensif n'est pas respecté dans un programme Pantagruel consiste à lui indiquer qu'il existe une action de l'invariant qui est soit inexistante dans les règles du programme, soit systématiquement appelée avec des mémoires ne vérifiant pas les prédicats de l'invariant. Dans ce cas, on peut lui proposer de rajouter l'invariant comme une règle d'orchestration du programme, ou de compléter une ou plusieurs règles contenant les actions de l'invariant, en leur ajoutant le prédicat de cet invariant.

D'après la définition 7.3.2, l'invariant extensif Ex1 n'est pas respecté, car il y a bien dans la même application un appel à l'action fermer sur le mitigeur (R2.2), mais il n'est pas conditionné par le capteur detection **is** VRAI sur le détecteur de chute. En effet, comme aucune prévision n'est possible sur ce capteur (car il teste une variable volatile), on peut trouver une mémoire s telle que (detecteurChute.detection **is** VRAI)$\not\Rightarrow$(porteDouche.etat **is** OPEN $\vee$ detecteurDouche.detection **is** FAUX). Pour respecter Ex1, il suffit d'ajouter cet invariant comme règle d'orchestration dans l'application, ou d'ajouter son prédicat à l'ensemble des conditions de la règle R2.2 avec l'opérateur **OR** (écrit "$\vee$" ici).

Puisque ces propriétés doivent être évaluées sur l'ensemble des règles d'orchestration d'un programme, leur procédé de vérification pourrait reposer sur l'environnement des règles *RuleEnv*. Il correspond à $\mathcal{R}$ dans les définitions, et peut être construit avant que les actions des règles ne soient exécutées, ce qui permet de faire une vérification statique des propriétés.

Rendre explicite le respect de ces invariants en modifiant le programme peut complexifier à outrance celui-ci et rendre sa lecture difficile. Pour éviter cet écueil, une idée serait d'utiliser les définitions des invariants pour construire une relation d'ordre entre les règles d'orchestration. Nous souhaitons explorer cette idée dans le futur.

7.3.2 Non-interférence

La propriété de non-interférence impose que lorsque deux règles sont exécutées en parallèle, elles n'ont pas d'effet contradictoire sur l'état d'un programme : soit leurs effets sont identiques sur une même variable, soit elles ne modifient pas la même variable. Sinon, on dit que ces règles sont en conflit ou interfèrent. La non-interférence est rendue abstraite dans la sémantique de Pantagruel par la fonction *join* définie sur la structure mémoire *Store*. La sémantique ignore donc le risque de conflit, induisant un comportement indéterministe. En particulier, l'implémentation de la fonction *combine* utilisée par *join* pour combiner les effets portant sur une variable est laissée à la discrétion du développeur qui développe le compilateur ou l'interprète fondé sur la sémantique (voir le domaine *Store* défini dans la figure 5.10, page 68).

Pour aider l'expert-métier à construire des règles d'orchestration non-interférentes, nous nous basons sur les déclarations de la taxonomie concernant les variables write, c'est-à-dire les déclarations d'attributs write et de méthodes avec effets. Notons que les méthodes ne donnent pas d'information sur la manière dont les variables qu'elles déclarent sont modifiées. Cela impose une définition très restrictive de non-interférence (qui peut être allégée à terme avec des hypothèses sur les effets des méthodes). Ainsi, deux règles R et R' peuvent interférer si l'un des quatre cas suivants est observé : premièrement, R et R' affectent à la même variable (write) deux valeurs différentes. Deuxièmement, R affecte une valeur à une variable, et R' invoque une méthode ayant un effet sur cette même variable. Troisièmement, R et R' invoquent deux méthodes différentes ayant un effet sur la même variable. Enfin, R et R' invoquent la même méthode déclarant un effet, mais avec des paramètres différents. Si les méthodes invoquées ne déclarent pas d'effet, R et R' ne sont pas interférentes : leur effet n'est pas observable, l'état du système n'est donc pas altéré.

7.3.2.1 Définition

Pour formaliser ces définitions, nous introduisons quelques notations. Etant donnée une action a, on note $\mathsf{effect}(a)$ la variable write modifiée par l'action, c'est-à-dire soit par un appel de méthode, soit par une affectation. Notons que lorsque a est une méthode ne déclarant pas d'effet, $\mathsf{effect}(a)$ renvoie la valeur fantôme none. On note $\mathsf{avalue}(a)$ la valeur d'affectation si a est une affectation, et $\mathsf{pvalue}(a)$ et $\mathsf{method}(a)$ le paramètre réel et le nom de la méthode invoquée si a est un appel de méthode.

Définition 7.3.3 (Non-interférence) *Etant données deux règles $R = (p, A)$ et $R' = (p', A')$, R et R' interfèrent s'il existe $(a_1, a_2) \in A \times A'$, et une mémoire $s \in S_p \cap S'_p$ (c'est-à-dire vérifiant $(s \vdash p) \wedge (s \vdash p')$), et telles que :*

- $(\mathsf{effect}(a_1) = \mathsf{effect}(a_2)) \wedge (\mathsf{avalue}(a_1) \neq \mathsf{avalue}(a_2))$*, si a_1 et a_2 sont des affectations ;*
- $(\mathsf{effect}(a_1) = \mathsf{effect}(a_2)) \wedge (\mathsf{effect}(a_2) \neq \mathsf{none})$ *si a_1 est une affectation et a_2 un appel de méthode ;*
- $(\mathsf{effect}(a_1) = \mathsf{effect}(a_2)) \wedge (\mathsf{effect}(a_1) \neq \mathsf{none}) \wedge (\mathsf{effect}(a_2) \neq \mathsf{none})$*, si a_1 et a_2 sont des appels de méthodes avec $\mathsf{method}(a_1) \neq \mathsf{method}(a_2)$;*
- $(\mathsf{effect}(a_1) \neq \mathsf{none}) \wedge (\mathsf{pvalue}(a_1) \neq \mathsf{pvalue}(a_2))$*, si a_1 et a_2 sont des appels de méthodes avec $\mathsf{method}(a_1) = \mathsf{method}(a_2)$.*

Comme pour la vérification des invariants, la sémantique permettrait de représenter $\mathcal{R}$ par un environnement *RuleEnv* afin de vérifier la non-interférence en comparant deux à deux les règles stockées dans l'environnement.

Le procédé de vérification comprendrait deux étapes : premièrement, utiliser les informations fournies par la taxonomie sur les variables write, pour savoir si leurs actions ont un effet potentiellement différent sur la même variable. Deuxièmement, vérifier si les prédicats de ces deux règles peuvent être satisfaits en même temps (ce qui engendrerait *potentiellement* l'exécution parallèle de ces deux règles).

La recherche d'une mémoire "contre-exemple" engendrant un risque d'interférence reposerait sur les variables utilisées dans les prédicats, ainsi que sur la nature de ces variables. Il s'agirait, entre autres, de regrouper l'ensemble des variables comparées par les deux prédicats. Si cet ensemble n'est pas vide, il serait possible, au moyen d'outils de vérifications existants (par exemple, des solveurs de contraintes [GSV08]), de générer les espaces de valeurs pour lesquels les prédicats des deux règles peuvent être tous deux satisfaits. Si cet ensemble est vide, deux cas de figures se présentent, suivant que les prédicats utilisent des variables volatile ou write. Dans le premier cas, la nature imprévisible des variables impliquerait par sécurité de systématiquement supposer l'existence d'une mémoire satisfaisant les prédicats des deux règles. Si ces prédicats utilisent des variables write, il serait possible, en explicitant les dépendances entre les règles (en l'occurrence entre celles dont les actions ont des effets sur des variables, et celles dont les prédicats utilisent ces variables), de préciser les conditions pour lesquelles les règles interfèrent. Il conviendrait de combiner les deux traitements lorsque les prédicats utilisent des variables volatile et write.

7.3.2.2 Un exemple

En guise d'exemple, nous ajoutons à l'application "2 - PRENDRE UNE DOUCHE" la règle R2.4 définie dans la figure 7.3. Cette règle commande la fermeture du mitigeur lorsqu'Henrick fait une chute alors qu'il est sous la douche. En ajoutant cette règle, le développeur introduit un conflit potentiel avec la règle R2.1 de la figure 6.3.

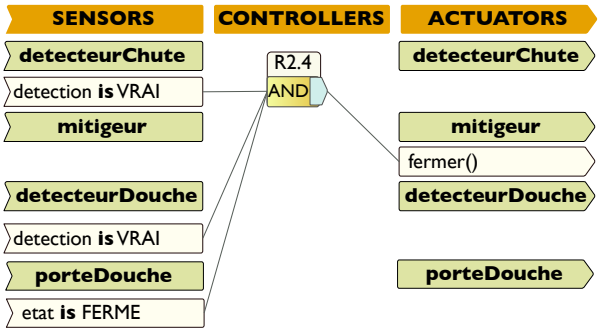

FIG. 7.3: Une règle introduisant une interférence dans l'application d'Henrick

Ces deux règles utilisent les méthodes ouvrir et fermer sur l'entité mitigeur, chacune déclarant un effet sur l'attribut etat de l'entité. Elles peuvent s'exécuter en parallèle, pour la mé-

moire $s = [\mathsf{detecteurDouche.detection} \mapsto \mathsf{VRAI}][\mathsf{detecteurChute.detection} \mapsto \mathsf{VRAI}][\mathsf{porteDouche.etat} \mapsto \mathsf{FERME}](\lambda v.\bot)$, qui, de plus, implique uniquement des variables volatiles. De ce fait, le comportement réel de l'application est imprévisible et peut donc devenir dangereux pour Henrick.

De même que pour les invariants, l'expert-métier peut corriger le programme en rajoutant des conditions dans les règles interférentes afin de les rendre mutuellement exclusives, c'est-à-dire que l'une ne s'exécute que si les conditions de l'autre ne sont pas vérifiées. Dans notre exemple, pour faire cette correction il s'agira d'ajouter à la règle R2.1 la condition detecteurChute.detection **is** FAUX. Néanmoins, cette fois encore, la définition d'une relation d'ordre entre les règles permettrait d'éviter d'engendrer des programmes pollués par un excès de conditions.

7.4 Synthèse

Dans ce chapitre, nous avons présenté une approche de développement de programmes Pantagruel basée sur les invariants d'actions. Ces invariants d'actions sont définis comme des motifs de règles d'orchestration Pantagruel. Ils ont une sémantique légèrement différente de celle des règles, et permettent de guider le développeur dans l'écriture de programmes fiables. Une telle démarche ouvre en outre la voie à une programmation modulaire des applications Pantagruel. Nous avons également formalisé la propriété de non-interférence d'un programme. Celle-ci permet de fournir à l'expert-métier les informations nécessaires à la correction des règles interférentes, par exemple en déterminant une mémoire qui peut déclencher l'exécution parallèle de deux règles dont les actions ont des effets différents sur une même variable write. Pour vérifier ces propriétés, nous avons commencé l'implémentation d'un petit prototype reposant sur la sémantique de Pantagruel et capable de générer un ensemble fini de mémoires "abstraites" à partir des prédicats des règles et des spécifications de leurs actions (il ne fait donc pas d'exécution en tant que telle et se contente d'analyser les définitions des règles). Ce prototype est une mise en oeuvre directe des définitions que nous avons données et basées sur les mémoires satisfaisant des prédicats. Il produit des informations qui peuvent être utilisées par le développeur pour corriger ses programmes afin qu'ils respectent les invariants. Cette implémentation est pour le moment très sommaire : notamment, dans le cas de la gestion des conflits, dont la définition implique fortement les variables write, elle engendre un ensemble d'états du programme (c'est-à-dire de mémoires) dont certains ne sont peut-être jamais atteignables (elle génère beaucoup de résultats "faux-positifs") à l'exécution. C'est également vrai pour les invariants.

Pour définir un processus de vérification optimal, nous pourrions utiliser les techniques d'interprétation abstraite [CC77]. Ces techniques permettent de définir une approximation de la sémantique d'un programme en définissant notamment des domaines abstraits permettant de raisonner sur un ensemble d'états (il s'agirait par exemple de mémoires dans Pantagruel) finis. On pourrait alors par exemple générer des séquences d'états, à partir d'une exécution abstraite des règles sur un nombre fini de pas d'itération, ainsi que d'hypothèses sur les états (par exemple en définissant des fonctions modélisant l'évolution des variables *volatile*).

Comme nous l'avons évoqué dans ce chapitre, l'approche guidée par les invariants souffre également d'une limitation : le programmeur est obligé d'"injecter" ces invariants dans le programme, ce qui peut entraîner une complexification de la logique d'orchestration du programme et la rendre difficile à lire. Au lieu de les intégrer dans les programmes, les invariants pourraient

être vus comme des règles d'orchestration à part entière s'exécutant en parallèle avec le programme d'orchestration "normal". Nous discutons de cette proposition et des problématiques de recherche qui en découlent dans la conclusion de cette thèse.

8 Evaluation globale

Contexte Nous avons mis en évidence au début de cette thèse la difficulté de développer des applications ubiquitaires, et par conséquent la nécessité de proposer une approche qui soit à la fois expressive, accessible, et fiable. Nous venons d'achever la présentation de notre méthodologie outillée, une approche qui tente de répondre à ces trois exigences. Nous avons validé la méthodologie en effectuant une évaluation auprès d'éducateurs spécialisés. Nous avons ensuite montré que Pantagruel favorisait la vérification de logiques d'orchestration grâce à sa sémantique. Nous devons à présent évaluer l'expressivité et l'accessibilité du langage Pantagruel.

Contribution Ce chapitre propose d'évaluer l'expressivité et l'accessibilité de Pantagruel. Pour cela nous évaluons d'abord les apports de notre approche en la comparant aux approches existantes dans le domaine de l'orchestration et plus particulièrement dans l'informatique ubiquitaire. Ensuite nous évaluons l'expressivité du langage par rapport aux champs d'applications présentés dans le chapitre 2. Cette évaluation repose sur trois catégories d'entités que nous avons identifiées à l'issue d'une analyse de ces champs d'applications. Enfin, nous évaluons l'accessibilité du langage d'orchestration auprès de programmeurs novices, et plus particulièrement le caractère intuitif du paradigme visuel de Pantagruel.

Plan La section 8.1 présente une évaluation comparative de Pantagruel selon quatre caractéristiques de notre approche. Dans la section 8.2, nous montrons que Pantagruel couvre les trois champs d'applications proposés dans le chapitre 2, et qu'il permet d'exprimer une grande variété d'applications de l'informatique ubiquitaire. Dans la section 8.3, nous faisons une évaluation de l'accessibilité du langage d'orchestration en suivant une démarche basée sur les méthodes d'évaluations existantes d'interfaces homme-machine et de langages visuels pour l'informatique ubiquitaire. Enfin, nous concluons dans la section 8.4.

8.1 Evaluation comparative dans le domaine de l'orchestration

Dans cette section, nous comparons Pantagruel aux approches partageant quatre de ses caractéristiques :

- les approches à base de taxonomie ;
- les métaphores visuelles dans l'informatique ubiquitaire ;
- les méthodologies de développement d'applications ubiquitaires ;
- les approches de vérification dans l'informatique ubiquitaire et les langages à base de règles pour l'utilisateur final.

L'objectif de ces comparaisons est de valider la position de Pantagruel en tant que langage visuel dédié conciliant l'accessibilité de la programmation et son expressivité, comme illustré dans la

figure 8.1. En particulier, cette figure présente sur quatres colonnes différents moyens de programmation, des intergiciels qui sont très expressifs mais non accessibles à un non-programmeur, aux approches destinées à l'utilisateur qui sont très accessibles, mais peu expressives parce que restreintes à un champ d'applications ou peu flexibles.

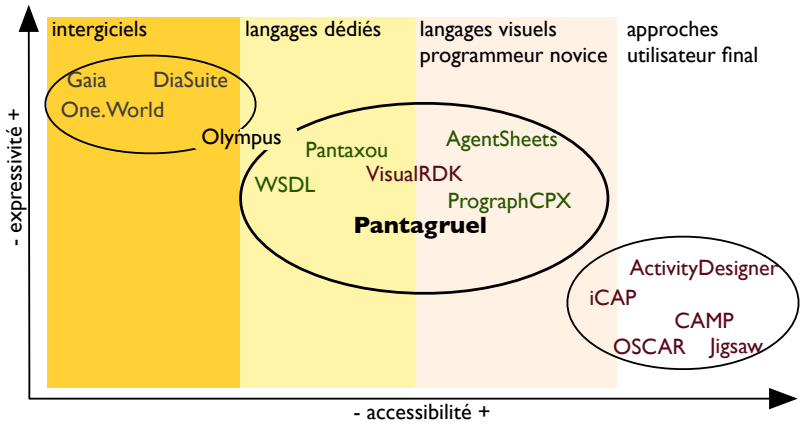

FIG. 8.1: Expressivité et accessibilité des approches existantes

8.1.1 Approches à base de taxonomie

Des approches à base de taxonomie proposent de gérer la diversité des champs d'applications, dans différents domaines dont l'informatique ubiquitaire, les services webs, la programmation de jeux ou de contrôle de robots.

Approches reposant sur des descriptions de services Dans le domaine de l'informatique ubiquitaire, Olympus [RCAM+05] est un cadre de programmation ou *framework* développé pour l'intergiciel Gaia, et permettant de définir une ontologie de services destinés à être orchestrés. A partir de cette ontologie le framework Olympus permet de découvrir des services pertinents pour une application particulière. Cependant, il ne fournit pas un modèle d'interaction pour décrire les informations offertes par les entités. Plus généralement, les approches à base d'ontologies, populaires dans la communauté des services web et de la gestion des connaissances, permettent de décrire des relations entre les entités. Contrairement à ces approches où la description des relations est généraliste, c'est-à-dire que les types d'informations représentés sont de natures variées, Pantagruel se focalise sur la description de leur interface d'interaction.

Dans le domaine des services web il existe également des langages plus spécifiques pour définir les interfaces d'interaction des services. Par exemple, WSDL [CCMW01] est un langage de spécification de contrats de services web, via la déclaration des fonctionnalités et des données fournies par un service. Cependant, cette déclaration ne fournit pas d'information précise sur les données au delà de leur type. Pantagruel diffère de ce langage en rendant explicite la nature – externe, applicative, ou constante – de l'information de contexte abstraite par les attributs.

Approches visuelles reposant sur les classes d'objets Dans le domaine du contrôle de robots, on trouve des approches visuelles orientées objets pour spécifier un ensemble de classes caractérisant un robot, puis programmer des applications à partir de cette spécification. Par exemple, le langage visuel généraliste Prograph CPX [SC95] permet de définir une hiérarchie de classes d'objets (en particulier, des capteurs et des effecteurs) pouvant être manipulés dans un robot. Le développeur bénéficie ainsi de l'avantage de la réutilisation apporté par le paradigme orienté objets. AgentSheets [Rep93] est une boîte à outils pour la définition d'environnements dits *orientés domaines*, où un domaine est défini par un environnement graphique et des agents qui sont des entités interagissant entre elles ou avec un utilisateur.[1] Le langage LegoSheets [GIL+95] a été ainsi conçu avec AgentSheets. Dans AgentSheets, un agent est modélisé par une classe et des tâches spécifiant son rôle, à l'aide des sous-composants suivants : capteurs, effecteurs, comportements, états (lesquels sont conditionnés par un comportement, qui peut être l'exécution d'une méthode) et images.

Cependant, AgentSheets ne donne pas d'indications particulières sur la manipulation d'information de contexte ou sur le comportement des méthodes, comme c'est le cas dans Pantagruel, via les interfaces d'interaction des entités. L'avantage de cette information est qu'elle facilite le raisonnement sur la logique d'orchestration.

8.1.2 Métaphores visuelles dans l'informatique ubiquitaire

Il existe un grand nombre d'approches visuelles pour permettre aux programmeurs novices et utilisateurs finaux de développer des applications ubiquitaires. Ces approches offrent une représentation intuitive des programmes d'orchestration, et permettent ainsi de rapprocher le développement des programmes des besoins de l'utilisateur final. Cependant, à notre connaissance, aucun des outils proposés dans la littérature de l'informatique ubiquitaire ne propose une approche suffisamment flexible pour couvrir un vaste champ d'applications contrairement aux approches à base de taxonomie dont Pantagruel fait partie.

Par exemple, CAMP [THA04] est un outil à base de règles qui cible l'utilisateur final. Il est basé sur la métaphore de la poésie magnétique : l'utilisateur compose visuellement des phrases de la forme "quand ... alors" en combinant des mots appartenant au vocabulaire de la domotique. Son champ d'applications est donc restreint, d'autant plus que CAMP ne repose pas sur une modélisation préalable d'un domaine particulier, ce qui réduit la flexibilité de cet outil. iCAP [DSSK06] est un autre outil fournissant un ensemble prédéfini et fixé de classes d'entités génériques : les objets (*things*), les activités, le temps, la localisation et les personnes. Néanmoins iCAP ne propose pas une approche uniforme pour exprimer des règles sur un groupe d'entités au delà du groupe de personnes. A l'inverse, notre approche fournit un support syntaxique pour définir des filtres sur les instances d'une classe d'entités quelles qu'elles soient. Ainsi, on peut créer des règles d'orchestration génériques et donc réutilisables dans une variété d'environnements instanciant une taxonomie donnée. D'autres outils basés sur le paradigme de *story-board* comme Topiary [LHL04] expriment la logique d'orchestration sous

[1] La définition précise dans la thèse de Repenning sur AgentSheets [Rep93], est la suivante : "*Agents can be viewed as active "objects to think with", equipped with sensors and effectors to interact with users or other agents*"

forme de flux de contrôle. Cependant, Topiary ne permet de définir que des applications basées sur la localisation. Enfin, les outils OSCAR [NES08] et Jigsaw [HCH+03] (ce dernier est basé sur la métaphore du puzzle) ciblent les applications domotiques et proposent une approche pour découvrir, connecter et contrôler un ensemble de services et d'entités matérielles. En plus d'être restreintes à un champ d'applications, ces approches ont une expressivité limitée car elles consistent à composer des entités et ceci à travers un accès limité à leurs fonctionnalités.

Le langage VisualRDK [WKU+07] se distingue des outils précédents par sa cible de programmeurs, qui sont les programmeurs novices ou expérimentés. C'est un langage de programmation qui offre des constructions inspirées des langages généralistes comme les processus, les cascades conditionnelles et les signaux. Ces constructions imitent celles de la programmation traditionnelle, sans spécialement cibler les aspects spécifiques au domaine de l'orchestration. Pantagruel diffère de cette approche en centrant le processus de programmation de règles sur la sélection d'entités, dont l'orchestration est réalisée en les interconnectant visuellement pour invoquer leurs fonctionnalités.

8.1.3 Méthodologies dans l'informatique ubiquitaire

Dans la communauté de recherche sur l'informatique ubiquitaire et les interfaces homme-machine, il existe des méthodologies destinées au développement d'applications ubiquitaires.

Par exemple, le cadre conceptuel *Context Toolkit* [DAS01] intègre une méthodologie pour guider le développement d'une taxonomie d'entités dédiée à une application. Cependant, elle donne lieu à une représentation programmatique des entités. De plus, elle concerne le niveau taxonomique, alors que notre méthodologie concerne le niveau applicatif.

D'autres méthodologies ont été proposées pour développer des applications ubiquitaires [ST06, KSS07, HKB+06, HS08]. Néanmoins, soit elles ne sont pas outillées [ST06, KSS07], laissant un fossé entre les besoins des utilisateurs et les applications finales, soit elles sont restreintes à un champ particulier (comme la domotique [ST06], les équipements sportifs [KSS07], les interfaces physiques [HKB+06], ou les applications pour téléphones mobiles [HS08]). A l'inverse, notre méthodologie est outillée par un langage permettant de modéliser divers champs d'applications.

Comme notre approche, l'outil *ActivityDesigner* [LL08] propose un couplage fort entre une méthodologie et un outil, à l'aide d'un paradigme basé sur le concept d'activités [Wil09]. Dans cette approche, une application est définie par des éléments appelés des pages, permettant de représenter l'état de l'activité d'un utilisateur (par exemple, s'il est en train de faire de la course à pied, ou s'il s'est entraîné pendant dix minutes sur son vélo d'appartement). Les pages sont connectées entre elles par des transitions, lesquelles peuvent être déclenchées par une entrée utilisateur (par exemple un bouton appuyé sur une interface de suivi d'activité) ou par l'état d'une activité (par exemple le temps pendant lequel une activité est pratiquée). ActivityDesigner offre une approche de développement des applications dirigée par les besoins de l'utilisateur, contrairement à notre approche où le développement est dirigé par les entités. De cette façon, la distance entre les besoins et l'application est plus petite qu'avec notre approche. Cependant, ce choix a pour contrepartie une expressivité limitée aux champs d'applications basés sur les activités humaines.

8.1.4 Vérification des langages d'orchestration

Nous comparons l'approche de vérification de Pantagruel aux approches de développement d'applications dans l'informatique ubiquitaire, puis aux approches visuelles à base de règles, du domaine de la programmation de jeux ou de robots.

8.1.4.1 Approches de l'informatique ubiquitaire

Dans de récents travaux de l'informatique ubiquitaire, des modèles formels ont été proposés pour spécifier et vérifier des environnements ubiquitaires [NYT+06, RC08]. Le modèle d'*Ubireal* [NYT+06] est basé sur le langage CTL de logique temporelle [CES86], celui proposé pour les applications de l'intergiciel Gaïa [RC08] repose sur le calcul ambiant [CG00]. Ces procédés de vérification nécessitent des connaissances pointues et demandent un investissement important. Notre approche tente de limiter cet investissement en guidant l'expert-métier dans l'activité de programmation au moyen de l'écriture et de la vérification d'invariants en Pantagruel. De plus, la sémantique de Pantagruel peut faciliter le processus de vérification.

De plus, comme mentionné par Ranganathan and Campbell [RC08], l'une des limitations de la vérification formelle est dûe au fossé entre la spécification et l'implémentation. En incorporant le processus de vérification dans le cycle de développement des programmes Pantagruel, notre approche fait un pas vers la réduction de ce fossé.

8.1.4.2 Approches visuelles à base de règles

Aucun des langages visuels pour l'utilisateur final que nous avons cités précédemment ne propose de moyens de vérification formelle des programmes crées par l'utilisateur. Dans les langages visuels à base de règles comme iCAP [DSSK06], CAMP [THA04], ou encore dans Topiary [LHL04], les règles conflictuelles sont détectées et indiquées à l'utilisateur final. Si l'utilisateur ne les résout pas manuellement, ces langages exécutent par défaut les dernières règles créées parmi celles qui sont conflictuelles. Cependant, la vérification des programmes dans ces approches se limite à la détection de conflits.

On trouve cette limitation dans d'autres langages d'orchestration visuels à base de règles. En particulier, nous avons exploré les domaines de la programmation de jeux ou de contrôle de robots. Par exemple, KidSim [SCS94] est un langage de programmation de jeux adoptant le modèle d'exécution séquentiel, qui lui permet d'imposer un ordre d'exécution des règles et d'éviter ainsi le risque de conflits propre aux modèles d'exécutions parallèles. Un autre moyen de résoudre les conflits est de définir des priorités entre les règles, comme le font Altaira [Pfe98] pour les robots, et Blender [vG03] et ChemTrains [BL93] pour la programmation de jeux. Néanmoins, contrairement à Pantagruel, aucun de ces langages ne tire profit du paradigme à base de règles pour intégrer un développement dirigé par les propriétés.

8.2 Périmètre d'expressivité de Pantagruel

Nous avons défini dans le chapitre 2 l'expressivité d'une approche comme sa capacité à définir facilement des applications pour de nombreux champs d'applications, selon les besoins expri-

més par les utilisateurs de ces champs. Nous avions alors identifié trois champs d'applications pouvant servir de référence pour évaluer une approche de programmation d'applications d'orchestration. L'évaluation qualitative présentée dans cette section a pour finalité de déterminer la capacité de Pantagruel à couvrir ces champs. En particulier, nous évaluons Pantagruel selon deux critères : (1) modéliser une grande variété de champs d'applications en utilisant les concepts du langage de taxonomie ; (2) modéliser un large panel d'applications dans un champ donné.

Dans la littérature en informatique ubiquitaire, les approches visuelles valident également leur expressivité selon une classification des applications ubiquitaires, soit existantes, soit définies selon leurs objectifs. Cependant, cette classification n'intègre pas la richesse des champs d'applications. Par exemple, l'expressivité du langage visuel CAMP [THA04] valide trois *modèles* d'applications issus d'une étude menée auprès d'utilisateurs finaux sur les applications qu'ils attendent dans le domaine de la domotique. De même, A.K. Dey *et al.* montrent que leur approche couvre les applications de la domotique imaginées par des utilisateurs finaux.

Contrairement à ces approches Pantagruel vise plusieurs champs d'applications. Nous proposons donc de compléter les démarches d'évaluation de ces approches selon les deux critères évoqués plus haut et précisés ici, qui concernent à la fois l'expressivité du langage de taxonomie et celle du langage d'orchestration :

Richesse des champs. Puisque Pantagruel définit un champ d'applications au travers des entités qui le constituent, notre étude d'expressivité consiste à définir l'*espace de modélisation des entités* qui couvre une grande variété de champs, et à étudier la capacité de Pantagruel à spécifier les entités caractérisées par cet espace.

Richesse des applications. Comme iCAP [DSSK06], nous utilisons la classification d'applications ubiquitaires proposée par B. Schilit *et al.* [SAW94]. A travers cette classification, nous montrons également que Pantagruel facilite la définition de variations d'une application suivant les préférences des utilisateurs ou leurs besoins.

L'évaluation de Pantagruel s'appuie essentiellement sur un travail expérimental : nous avons développé des applications pour les trois champs présentés dans le chapitre 2 qui sont : la domotique, la gestion d'information et l'assistance à la personne. Le point de départ de ce travail est la collecte et l'analyse d'un ensemble de besoins pour chacun de ces champs. Nous avons ensuite développé des taxonomies d'entités qui rassemblent les connaissances relatives à ces champs. Nous détaillons à présent cette démarche.

8.2.1 Support matériel et démarche

Avant d'identifier un ensemble de besoins relatifs à chaque champ, nous avons défini une bibliothèque d'objets trouvés dans la littérature de l'informatique ubiquitaire, et dont certains sont disponibles dans notre laboratoire sous forme d'entités simulées ou réelles. Nous avons ensuite appliqué la méthodologie pour analyser ces besoins : chaque besoin correspondait à un but (ou une familles de buts), décomposé en buts élémentaires correspondant à un comportement escompté pour une application. Par exemple, le besoin de gestion d'énergie dans une maison se décompose en buts incluant "contrôler les lampes", lui-même se décomposant en "détecter l'occupation de la maison", "capter la luminosité ambiante" et "ajuster l'intensité lumineuse".

Chaque but élémentaire était ensuite associé à des objets de la bibliothèque de départ. Par exemple, le but "détecter la luminosité ambiante" était associé à un capteur de luminosité.

Cette analyse a conduit à la définition de taxonomies d'entités pour chaque groupe d'applications listé dans le Tableau 8.1. Nous y avons inclus les deux exemples des chapitres 5 et 6 : la gestion de conférence et l'assistance de routines à domicile. Parmi les classes définies, une dizaine était partagée par les différents champs, en particulier les détecteurs (de mouvement, de présence ou de luminosité) et les entités telles que calendriers ou agendas. Nous avons développé et compilé toutes ces applications, et simulé la moitié d'entre elles (gestion de plantes, de routines à domicile, de lumières et personnalisation musicale) à l'aide du simulateur DiaSim.

Groupe d'applications (champ)	**Taxonomie**		
	classes	**attributs**	**méthodes**
Gestion de lumières et personnalisation musicale (d.)	10	10	5
Gestion de conférence (i.)	15	19	15
Gestion de plantes (d.)	11	9	5
Assistance de routines à domicile (d., a.)	15	15	9
Gestion d'information scolaire (d., i.)	10	16	7
Gestion d'ouverture d'école (d., i.)	10	10	6
Gestion d'interphones résidentiels (i., a.)	9	19	11

TAB. 8.1: Cas d'étude – (d. : domotique ; i. : gestion d'information ; a : assistance à la personne)

8.2.2 Espace de modélisation des entités

Suite à l'analyse des taxonomies que nous avons développées, nous avons déterminé trois catégories d'entités couvrant les champs étudiés : les capteurs physiques et virtuels, les capteurs de configuration et de suivi, et les composants de calcul. La présentation de ces catégories met en évidence l'expressivité du langage de taxonomie, et plus particulièrement la pertinence des concepts choisis pour définir les interfaces d'interaction des entités.

8.2.2.1 Capteurs physiques et virtuels

La plupart des applications ubiquitaires dépendent fortement des changements aléatoires survenant dans l'environnement où elles sont déployées. Par exemple, une application gérant la consommation d'énergie doit réagir aux changements de température. Nous modélisons la nature versatile de l'environnement au moyen de capteurs physiques et virtuels, par exemple, un capteur physique de luminosité ou de température. Ces informations sont naturellement représentées en Pantagruel par des attributs volatile sur des classes d'entités représentant de tels capteurs.

Lorsque l'environnement est virtuel, par exemple s'il s'agit d'un ordinateur ou d'un composant logiciel avec lequel l'utilisateur peut interagir, nous parlons de capteurs virtuels. Ceux-ci expriment alors une situation captée par un ordinateur, provoquée par l'utilisateur ou par une application logicielle tournant sur l'ordinateur. Par exemple, la classe PresenceAgent de notre exemple du chapitre 4 est un capteur virtuel qui modélise un client de messagerie instantané capable de détecter si une personne utilise son ordinateur ou pas (via l'attribut volatile status,

dans la mesure où la personne permet l'accès à cette information). De la même façon, une application logicielle de calendrier envoyant des événements selon les conférences prévues est un capteur virtuel. Enfin, les éléments constituants d'interfaces (logicielles ou non) avec lesquels l'utilisateur peut interagir peuvent également être vus comme des capteurs virtuels. Nous avons ainsi modélisé un bouton de panique sur le capteur virtuel Prompteur pour Henrick, ainsi que les interrupteurs permettant de contrôler manuellement l'alimentation des plantes, dans l'application de gestion des plantes.

8.2.2.2 Composants de configuration et de suivi

Configurer un environnement selon l'information recueillie par les capteurs, ou selon les préférences de l'utilisateur, est une caractéristique clé des applications ubiquitaires. En particulier, dans le champ de la domotique, le comportement des applications doit être configuré selon les habitants et leurs préférences. Nous appelons composants de configuration les entités destinées à effectuer ce type de réglage. Prenons l'application de personnalisation musicale de la maison développée pour iCAP [DSSK06] : la musique jouée par les radios installées dans une maison doit être configurée selon les préférences musicales des personnes se déplaçant de pièce en pièce. Dans cet exemple, que nous avons aussi utilisé pour motiver les participants de l'évaluation d'accessibilité, la classe Radio est un composant de configuration, tout comme la classe Réveil de l'exemple d'Henrick. Pour modéliser une entité représentant un composant de configuration, nous utilisons les attributs write ; ainsi, le genre musical music est un attribut write sur la classe Radio. Il peut être configuré selon les préférences d'une personne, représentée par exemple par des attributs constant, en affectant l'attribut write à ces derniers. On peut ainsi imaginer une classe Badge qui rassemble les préférences d'une personne (musiquePréférée) et les informations de sa localisation. Configurer la musique revient alors à affecter l'attribut musique d'une radio à la valeur de musiquePréférée d'une personne entrant dans la pièce où cette radio est installée.

La configuration de l'environnement peut aussi dépendre de méthodes appelées antérieurement par l'application. Prenons le cas des applications d'assistance à une personne pour accomplir différentes tâches de sa journée (se lever, prendre une douche et s'habiller) dans le bon ordre. Pour que l'assistance soit adaptée à la personne, il faut configurer l'environnement selon les tâches effectuées ou à accomplir. Par exemple, si l'environnement est composé d'un prompteur de tâches et d'un écran affichant les tâches restantes, ces applications doivent mettre à jour l'affichage de l'écran lorsqu'une tâche est supprimée dans le prompteur. Une manière de procéder est de suivre les différentes méthodes déclenchées sur les entités suite à la perception du comportement de la personne par les différents capteurs installés dans la maison. Nous appelons composants de suivi les entités destinées à faire ce suivi ou *monitoring*. Par exemple, lorsque la douche a coulé un certain temps, l'application peut en déduire (mais non garantir) que la personne a bien pris sa douche et ainsi lui suggérer de passer au séchage puis à l'habillement. Ces composants peuvent également être modélisés au moyen des attributs write sur les entités, servant alors d'indicateurs des méthodes déclenchées. Par exemple, l'attribut write nommé etat et défini sur la classe d'entités Mitigeur permet de savoir si la méthode ouvrir déclenchant le débit d'eau a été exécutée. D'après cette spécification, cet attribut est supposé être mis à jour dès que la méthode en question est invoquée, permettant à l'application de l'utiliser pour modifier

la liste des tâches enregistrées dans le prompteur de tâches.

8.2.2.3 Composant de calcul

Lorsqu'elles mettent en jeu un grand nombre d'entités, les applications doivent parfois rassembler des informations de contexte captées par ces entités et effectuer des calculs pour accomplir un but particulier. Le calcul d'information peut être explicité en établissant une relation entre les informations des entités et leur calcul, par exemple via les paramètres d'une méthode effectuant cette opération. Nous appelons composants de calcul les entités chargées d'effectuer de tels calculs.

Considérons des applications de gestion d'énergie qui contrôlent la consommation d'énergie en manipulant des données telles que la chaleur ou la luminosité ambiantes. La chaleur moyenne que doit produire un appareil de climatisation dans une pièce peut dépendre de la présence ou non de personnes dans la pièce et de l'heure de la journée (par exemple, il n'est pas nécessaire de faire fonctionner la climatisation à pleine puissance la nuit). Pour calculer cette chaleur, on peut imaginer une classe d'entités ComposantTemperature, utilisée comme un composant de calcul qui collecte ces informations et ajuste périodiquement le réglage de l'appareil. La chaleur moyenne est alors représentée par un attribut write nommé chaleurMoyenne sur ce composant, et son calcul est effectué par une méthode dont la signature est calculerTemperature(Heure h, Booleen occupé), et déclarant un effet sur chaleurMoyenne. De même, une classe d'entités Compteur, calculant le nombre de personnes dans une pièce (par exemple, qui se sont identifiées en utilisant un badge), peut être vue comme un composant de calcul dont l'effet est une variable nombre incrémentée à chaque fois qu'une personne entre (ou décrémentée dans le cas contraire). La déclaration des paramètres rend explicite les données requises pour la modification de l'effet calculé. La méthode peut ainsi être invoquée dans un programme Pantagruel selon le changement de contexte capturé par des entités de type capteur.

Résumé et limitations L'analyse des champs d'applications que nous avons faite nous a permis d'identifier trois catégories d'entités pouvant les caractériser. Nous avons montré que ces entités peuvent être spécifiées à l'aide des concepts du langage de taxonomie, et que ces concepts permettent d'expliciter certaines informations par le choix appropriés d'attributs, de méthodes, d'effets et de paramètres d'entrée.

Néanmoins, l'expressivité de Pantagruel est mise en défaut lorsqu'il s'agit de spécifier *comment* l'information de contexte applicative (les attributs write) doit être calculée. Cette information peut notamment être utile pour raffiner le processus de vérification des programmes, car elle pourrait fournir ou permettre de déduire des espaces de valeurs possibles pour les effets. Le cadre conceptuel *ContextToolkit* proposé par Dey *et al.* [DAS01] et que nous avons évoqué dans la sous-section 8.1.3 propose des abstractions de contexte pour spécifier certaines de ces informations. En particulier, l'utilisation de deux de ces abstractions pourraient bénéficier à Pantagruel : les *interprètes*, qui transforment des données en données de plus haut niveau, et les *aggrégateurs*, qui rassemblent des données propres à un élément particulier de l'environnement (par exemple, un "aggrégateur" de pièces, permettant de stocker le nombre de personnes entrant dans une pièce). Combiner ces abstractions avec notre langage de taxonomie permet-

trait d'enrichir significativement le rôle des classes d'entités, et de raffiner ainsi la vérification de programmes.

8.2.3 Espace de définition des applications

Nous montrons dans cette section que l'expressivité du langage de taxonomie favorise la concision des applications d'orchestration et l'adaptation des applications selon les préférences de l'utilisateur. Pour cela, nous proposons une évaluation qualitative reposant sur la classification d'applications ubiquitaires de B. Schilit *et al.* [SAW94]. Cette classification est centrée sur la notion de localisation de personnes, d'entités logicielles mobiles et de proximité des ressources. Nous examinons ici la capacité de Pantagruel à exprimer chaque catégorie d'applications de cette classification et nous montrons que, dans certains cas, cette classification peut s'étendre au delà des applications de localisation.

8.2.3.1 Sélection basée sur la proximité

Cette catégorie concerne la capacité d'une application à accéder aux ressources selon leur proximité, permettant ainsi de faire interagir des entités proches les unes des autres dans l'environnement. Ce type d'applications s'appuie essentiellement sur des capteurs physiques et virtuels. Le mécanisme de filtrage proposé par Pantagruel tente de généraliser cette fonctionnalité de sélection à des informations dépassant les contraintes de localisation.

L'information de contexte basée sur la proximité se modélise par des types de données et des attributs bien choisis tels que l'énumération Location et des attributs constant ou write, tels que room et location (voir figure 5.3 de la page 50). Le filtrage par classe des entités fournit un moyen concis de sélectionner les entités selon ces informations. Par exemple, les entités BadgeReader de la figure 5.5 (page 55) détectent la localisation de toute personne munie d'un badge, et permettent d'adapter le lancement de la conférence sur les écrans à proximité de ces personnes. Dans Pantagruel, ce mécanisme de sélection concerne aussi bien la localisation que d'autres types d'information de contexte. Par exemple, dans l'application de gestion de conférence, nous l'avons utilisée pour sélectionner une conférence sur le point de démarrer.

8.2.3.2 Reconfiguration contextuelle automatique

Les applications de cette catégorie gèrent les aspects dynamiques d'un environnement ubiquitaire, c'est-à-dire l'apparition ou la disparition d'entités, et la création de nouvelles connexions entre les composants. La reconfiguration automatique s'appuie principalement sur les capteurs physiques ou virtuels et les composants de configuration.

Un sous-ensemble d'applications appartenant à cette catégorie sont les applications dites "suivez-moi" (ou *follow-me*), où les interfaces utilisateur fournies par les applications *suivent* l'utilisateur lors de ses déplacements [HHS+99]. Un exemple d'application est un système de gestion de conférence chargeant une vidéo-conférence sur le téléphone mobile de l'utilisateur, sur son ordinateur, ou sur l'écran géant d'une pièce, selon que l'utilisateur est en déplacement, à son bureau, ou dans une salle commune. Dans l'exemple du chapitre 5, les règles R2.2 et R2.3 de la figure 5.6 implémentent partiellement cette application, créant une connexion entre

l'écran de la salle de conférence (meetingDisplay) et le gestionnaire de conférence (confManager) ou entre un téléphone mobile (SmartPhone) et le gestionnaire de conférence selon le contexte des participants. L'ajustement des lumières selon le déplacement de pièce en pièce d'un utilisateur constitue un autre exemple d'applications de reconfiguration contextuelle. Cet exemple s'exprime de manière très concise en Pantagruel, en connectant des entités Light et des entités PresenceDetector via une condition sur leur position (Light.room **is** PresenceDetector.room) et une condition sur l'état d'un capteur de présence représenté par un attribut volatile (detected). Deux règles suffisent pour définir une telle application de suivi de lumière : une pour les allumer et une pour les éteindre.

Enfin, le mécanisme de découverte d'entités, dont l'implémentation de Pantagruel tire profit, permet de gérer dans une certaine mesure l'apparition ou la disparition d'entités, toujours au moyen de règles utilisant les classes. Le processus d'exécution de Pantagruel interroge périodiquement l'environnement, collectant les entités correspondant aux conditions sur les classes. Néanmoins, il faut noter que Pantagruel ne propose pas de constructions pour demander explicitement à l'environnement quelles entités disparaissent ou apparaissent.

8.2.3.3 Informations et commandes contextuelles

Les applications de cette catégorie ont pour objectif d'exécuter des commandes paramétrées par les informations de contexte. Une commande est dite contextuelle lorsque la commande utilise des paramètres représentant des informations de contexte, ou lorsque son exécution est conditionnée par ces informations, ou encore lorsque ces deux cas surviennent. Les composants de configuration et de calcul sont donc les entités clés de ce type d'applications.

Les attributs typés, ainsi que la signification que nous associons aux paramètres d'entrée des méthodes, servent à définir ces applications. Utilisés en paramètres des méthodes, les attributs explicitent la relation entre une commande et le contexte avec lequel elle doit s'exécuter. Par exemple, dans la règle R1.4 de la figure 5.5 (page 55), la méthode d'affichage du téléphone d'une personne est paramétrée par la localisation de cette personne. Les commandes contextuelles peuvent aussi produire des résultats calculés selon les paramètres de la méthode. Ces résultats peuvent alors être modélisés par les effets (attributs write) d'une méthode.

8.2.3.4 Actions déclenchées par un contexte

Cette catégorie est définie comme une extension de la catégorie précédente : certaines commandes dépendent non seulement du contexte mais aussi de l'exécution de commandes précédemment exécutées. B. Schilit définit ce type d'actions ainsi : "*context-triggered action commands* [...] *are invoked automatically according to previously specified if-then rules*". [2] Ce type d'applications est directement lié aux composants de configuration. En Pantagruel, cette fonctionnalité est apportée par l'utilisation des attributs write dans la partie capteur d'une règle : ces attributs, en indiquant qu'une action a été exécutée, permettent de déclencher une règle suite à l'exécution d'une action provoquée par une autre règle, ce qui revient à décrire un système de transition. Ce type d'applications est illustré par notre exemple du chapitre 5 pour les

[2] "*les commandes correspondant à des actions déclenchées par un contexte* [...] *sont invoquées automatiquement selon des règles si-alors spécifiées antérieurement.*"

règles R2.1, R2.2 et R2.3 (figure 5.6, page 56) : ces règles définissent un capteur sur l'attribut write meeting_status des entités PersonProfile. Ces trois règles sont déclenchées lorsque les règles R1.1 ou R1.2 (figure 5.5, page 55) ont modifié cet attribut.

Résumé Nous venons de montrer que les concepts de Pantagruel et l'utilisation du paradigme à base de règle permettent de couvrir les quatre catégories d'applications présentées. De plus, nos abstractions permettent d'exprimer des applications qui vont au delà des contextes de localisation sur lesquelles ces catégories se focalisent. Enfin, les applications écrites en Pantagruel sont concises et emploient une notation homogène pour orchestrer un nombre d'entités potentiellement grand. La concision est notamment assurée par l'utilisation des classes, qui permet de s'abstraire des variations d'une application et de réutiliser une taxonomie pour déployer ces applications dans un grand nombre d'environnements instanciant cette taxonomie.

8.3 Accessibilité du langage d'orchestration

Les discussions que nous avons eues avec deux éducatrices spécialisées, rapportées dans le chapitre 6 nous ont apportés quelques retours sur les fonctionnalités nécessaires à Pantagruel pour être adapté à des utilisateurs finaux. Nous avons néanmoins pu mener une étude plus poussée de l'accessibilité de Pantagruel à des programmeurs novices, qui sont à mi-chemin entre utilisateurs finaux et programmeurs : ils n'ont pas d'expérience mais préparent une formation en programmation. En particulier, nous avons mené une évaluation auprès d'étudiants débutant leur première année d'une école d'ingénieurs en informatique et télécommunications. Pour cela, nous nous sommes basés sur des méthodes d'évaluations trouvées dans la littérature relative aux langages visuels de programmation d'applications ubiquitaires [NES08, THA04, DSSK06, LL08]. En particulier, nous avons souhaité évaluer deux aspects du langage d'orchestration :

Intuition du paradigme visuel Le paradigme visuel de Pantagruel est à base de règles et orienté entités. Nous avons évalué la pertinence du principe de programmation visuel dirigé par les entités pour créer des règles d'orchestration.

Accessibilité des abstractions Pantagruel définit les abstractions spécifiques pour orchestrer des ensembles d'entités. Nous nous sommes intéressés à la facilité de prise en main des concepts de classes et d'attributs pour filtrer des entités et exprimer des dépendances entre des classes d'entités.

8.3.1 Participants

Nous avons recruté dix-huit étudiants volontaires (quatre femmes et quatorze hommes âgés de 18 à 21 ans) venant d'écoles préparatoires en mathématiques et en physique. D'après un questionnaire préliminaire que nous leur avons fait remplir, la plupart des étudiants ont été peu confrontés à des langages visuels mais ont utilisé des logiciels graphiques dans le cadre scolaire. En particulier, huit étudiants ont manipulé des éditeurs de musique, de vidéos, ou de graphisme (Photoshop, SolidWorks, Pinnacles Studio, MotionWorks), et huit d'entre eux ont utilisé Maple, un logiciel de calcul. Seuls trois d'entre eux étaient autodidactes dans l'utilisation de langages

(Visual Basic, PHP, ou HTML). Cependant, ces expériences semblent ne pas avoir eu d'impact sur leur confiance à l'utilisation de l'éditeur Pantagruel (certains étudiants sans expérience ont montré des résultats plus convaincants que ceux avec expérience). Enfin, aucun des étudiants n'était familier à la notion d'informatique ubiquitaire ou de domotique.

8.3.2 Support matériel

L'étude s'est appuyée sur les éléments matériels suivants : un questionnaire préliminaire pour recueillir leur expérience de l'outil informatique, une courte documentation didacticielle, un ensemble de règles, et enfin un questionnaire d'appréciation. Avant de commencer les sessions d'évaluation, nous avons présenté aux participants une courte documentation didacticielle d'utilisation de l'outil. Nous avons ensuite observé leur manière de décrire un ensemble de règles à l'aide de Pantagruel. A la fin de l'évaluation, nous avons recueilli leur opinion sur l'outil via un questionnaire basé sur une échelle d'appréciation standard : le *System Usability Scale* [Bro96] (Echelle d'Accessibilité d'un Système).

Pour chaque session d'évaluation, nous avons également mis à disposition deux écrans (figure 8.2). Le premier affichait, via l'éditeur DiaSim [BJC09], un modèle en deux dimensions d'une maison composée d'entités d'une taxonomie prédéfinie. Cet écran servait de support visuel pour sélectionner les entités à orchestrer. Le deuxième affichait l'éditeur Pantagruel, présenté dans la figure 5.20 (page 82).

Pour accompagner chaque participant au cours d'une session, nous avons recruté deux observateurs dans notre équipe de recherche. Le premier avait pour tâche d'aider le participant s'il avait des problèmes avec l'interface graphique (par exemple pour connecter deux objets visuels en utilisant l'élément de connection). Le second notait les réactions du participant face à l'interface visuelle (par exemple, lorsqu'il effectuait une opération graphique invalide ou lorsqu'il réfléchissait à voix haute) et les interventions du premier observateur.

FIG. 8.2: Deux écrans pour l'évaluation de l'accessibilité (DiaSim et Pantagruel)

8.3.3 Sessions d'évaluation

Chaque participant est venu dans notre laboratoire pour une session individuelle de 60 minutes. Durant les 15 premières minutes, nous leur avons présenté le langage. Cette introduction comprenait une présentation du champ de la domotique et une utilisation de l'éditeur Pantagruel, à travers un exemple couvrant les concepts du langage. Nous avons également présenté

N°	Enoncés de règles simples	Temps
R1	Si Bruno ou Anna est dans le salon, régler le genre musical de la radio à JAZZ.	5.6 (2.6)
R2	S'il fait jour, éteindre la lampe extérieure.	4.5 (2.5)
R3	Si Anna entre dans la cuisine, allumer les lampes de la cuisine.	6.4 (3.8)

TAB. 8.2: Règles d'évaluation du paradigme visuel, et temps de complétion moyen (et écart type) en minutes

N°	Enoncés de règles avancées	Temps
R4	Si une personne entre dans une pièce, alors faire allumer toutes les lampes de cette pièce.	7.8 (3.2)
R5	Si une personne est dans une pièce, alors régler le genre musical de la radio située dans cette pièce au genre préféré de cette personne.	7.3 (2.9)
R6	S'il n'y a personne dans une pièce, alors éteindre les lampes de cette pièce.	8.0 (2.4)

TAB. 8.3: Règles d'évaluation des abstractions, temps de complétion moyen (et écart type), en minutes.

aux participants la démarche de construction d'une règle Pantagruel :

- analyser chaque énoncé de règle exprimé en langage naturel ;
- en extraire les entités impliquées en s'aidant du modèle 2D de l'environnement domotique prédéfini ;
- en extraire les conditions requises pour déclencher la règle ;
- en extraire les actions que la règle doit exécuter.

A chaque étape du processus, les éléments visuels correspondants doivent être créés dans l'éditeur Pantagruel. Pour cette évaluation, l'environnement prédéfini était composé des entités suivantes : des lampes (définies par une classe Lampe), des badges (Badge), des capteurs de luminosité (CapteurLumière) et des radios (Radio). Nous avons demandé aux participants de créer six règles de complexité variable pour orchestrer ces entités. Ces règles, présentées dans les tableaux 8.2 et 8.3, visaient à évaluer d'une part le paradigme visuel et d'autre part l'accessibilité des abstractions du langage (classes et prédicats-filtres).

Le premier ensemble de règles implique essentiellement des entités spécifiques. La règle R2 se distingue des deux autres par le fait que l'une des entités (le capteur de lumière) n'est pas explicitée dans l'énoncé. Le second ensemble nécessite l'utilisation de classes d'entités. La règle R4 généralise R3, nécessitant de créer un capteur filtrant les entités d'une classe (Lampe) pour ne pas avoir à itérer sur les lampes (la cuisine n'était équipée que de trois lampes et pouvait donc facilement être "orchestrée" en énumérant les entités Lampe spécifiques, alors que l'environnement comptait au total dix lampes). La règle R4 requiert en outre d'exprimer une dépendance entre deux classes d'entités, Badge et Lampe via leur attribut pièce). La règle R5 impose de paramétrer une action sur les entités Badge filtrées selon leur localisation (en l'occurrence le paramètre musiquePréférée d'une personne identifiée par son badge). Enfin, la règle R6 requiert d'exprimer une condition que doivent satisfaire toutes les entités d'une classe donnée (la classe Badge).

8.3.4 Analyse des résultats

Nous exposons à présent les opinions des participants recueillies à l'aide du questionnaire d'appréciation. Nous analysons ensuite le résultat des sessions d'évaluations selon le succès ou l'échec des participants à construire les règles des tableaux 8.2 et 8.3.

8.3.4.1 Résultats subjectifs

Le questionnaire *System Usability Scale* ou SUS [Bro96] est un questionnaire standard dont l'objectif est de recueillir l'opinion des utilisateurs sur l'ergonomie d'une interface graphique. Il compte dix questions – que nous avons adaptées à l'éditeur visuel Pantagruel – chacune associée à cinq réponses possibles : *vraiment pas d'accord* , *pas d'accord*, *indécis*, *d'accord* et *vraiment d'accord* (voir figure 10.1 de l'annexe pour la liste des questions). Via un calcul spécifique au questionnaire, ces résultats qualitatifs sont transformés en un résultat global compris entre 0 et 100. Le résultat moyen de nos questionnaires est de 70,3 (Ecart type=11.8, min=47.5, max=90.0) (la figure 10.2 de l'annexe détaille les résultats individuels). D'après l'évaluation empirique de ces questionnaires menée par Bangor *et al.* [LS09, BKM08] sur un ensemble d'interfaces utilisateurs, ce résultat place l'accessibilité de Pantagruel dans la catégorie "acceptable", minimum en dessous duquel l'interface requiert des améliorations avant d'être considérée comme accessible.

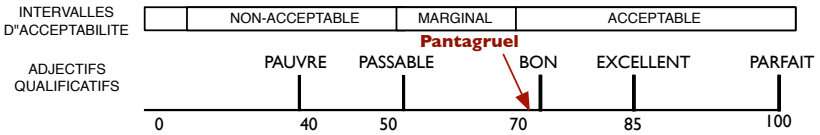

FIG. 8.3: Interprétation d'un Questionnaire SUS (extrait de [BKM08])

Dans ce questionnaire, quinze participants ont répondu *d'accord* ou *vraiment d'accord* à l'assertion "*Je pense que j'aimerais utiliser ce logiciel fréquemment pour prototyper des applications ubiquitaires*". Néanmoins, bien que personne n'ait exprimé son désaccord quant à la facilité d'utilisation de l'outil Pantagruel, sept d'entre eux ont donné la réponse *indécis*. Un participant ne s'est pas "*senti à l'aise à l'utilisation du logiciel*", contre douze d'entre eux qui étaient au moins *d'accord* avec cette assertion. Pour résumer, l'opinion générale pour chaque assertion était prononcée, sauf pour les assertions "*je pense que j'aurai besoin d'assistance pour être capable de créer d'autres applications*" et "*j'imagine que la plupart des gens pourraient apprendre très rapidement à utiliser ce logiciel*" : autant de personnes étaient d'accord et pas d'accord avec celles-ci.

Le questionnaire se terminait sur des remarques libres. Deux des participants ont indiqué qu'ils auraient eu besoin de quelques heures de formation avant de se sentir à l'aise avec l'éditeur Pantagruel. Deux autres ont mentionné le besoin d'avoir un simulateur pour tester leurs réponses, lequel n'était pas encore disponible à ce moment. Bien que l'appréciation de l'accessibilité de l'outil soit positive, il semblerait d'après ces résultats subjectifs qu'au premier abord, le paradigme visuel et les abstractions de Pantagruel ne soient pas intuitifs.

8.3.4.2 Analyse des performances dans la réalisation des règles

Nous venons d'analyser l'accessibilité subjective de Pantagruel. Nous analysons à présent son accessibilité au vu du succès ou de l'échec des participants à construire les règles que nous leur avons énoncées. En particulier, nous abordons ces deux points : le caractère intuitif des concepts visuels et l'accessibilité des abstractions.

Caractère intuitif des concepts visuels

Succès Après que les observateurs ont rappelé aux participants comment utiliser l'interface visuelle pour la première règle, tous les participants sont parvenus à appliquer naturellement le processus de création de règles associé au pardigme visuel. En particulier, ils ont choisi et positionné les bons éléments depuis la palette d'outils de l'éditeur vers le tableau d'édition à trois colonnes. La plupart d'entre eux a ainsi réussi à définir correctement les règles R2 et R3 en 3 à 7 minutes sans nécessiter d'intervention de la part des testeurs.

Défauts Seuls deux participants sont parvenus à saisir les concepts de l'interface sans intervention et rapidement (2 à 4 minutes). Quelques participants ont au départ confondu les capteurs et les actionneurs avec le concept entité et ont alors tenté de relier directement une section entité avec un actionneur. Deux des participants ont également tenté, pour la première règle, de glisser-et-déposer (*drag-and-drop*) les entités depuis l'environnement 2D vers l'éditeur Pantagruel. Enfin, la résolution moyenne des premières règles en 5.5 minutes montrent que, par rapport au nombre réduit d'éléments graphiques à manipuler pour cette règle, Pantagruel montre une faible ergonomie.

Enseignements tirés Les concepts visuels de Pantagruel sont faciles à saisir, mais sa structure orientée entités n'est pas aussi naturelle que nous l'attendions. Pour en donner une meilleure intuition, un moyen pourrait être de renforcer la relation entre l'environnement 2D et l'éditeur Pantagruel pour mettre en valeur le concept d'orchestration d'entités. Cette amélioration permettrait de se diriger vers un style de programmation plus intuitif connu dans la communauté des langages visuels sous le nom de programmation avec l'exemple ou *programming-with-example* [Mye86].

Accessibilité des abstractions Pantagruel

Succès La règle R3 pouvait être résolue rapidement avec ou sans abstractions. Quatre participants ont naturellement utilisé une classe pour la définir, à l'aide d'un capteur pièce **is** CUISINE sur la classe Lampe. Pour la règle R4 (second ensemble), la moitié des participants ont eu besoin d'un rappel sur le concept de dépendance de classes (la possibilité de mettre en relation deux classes dans les opérations d'une règle – capteur ou actionneur – pour réaliser une action sur un sous-ensemble d'entités) pour filtrer des entités. La plupart d'entre eux est ensuite parvenue à analyser correctement la règle R4 et à exprimer une dépendance entre les entités sans aide pour la règle suivante R5 (par exemple pour définir le prédicat badge.musiquePréférée **is** radio.genreMusical pour le réglage de la radio).

Défauts La création de règles du second ensemble a demandé plus de temps que pour le premier (7.7 minutes au lieu de 5.5 en moyenne). L'un des participants ne s'est pas du tout senti à l'aise avec l'opération de filtrage par les capteurs sur la classe Lampe de la règle R4 : bien qu'il eût créé une règle correcte, lorsqu'on lui a demandé de la relire, son interprétation ne correspondait pas à la signification de la règle. De plus, tous les participants ont demandé que le concept de dépendances de classes leur soit réexpliqué au moment d'aborder la règle R4.

Enfin, aucun participant n'est parvenu à construire la règle R6. Celle-ci nécessitait l'usage du mot-clé ALL ou NONE, qui, dans la version du prototype évalué, était placé sur un capteur et non sur un contrôleur comme c'est à présent le cas. Cet opérateur sélectionne toutes les entités d'une classe ou aucune, selon le capteur de la classe sur lequel il est placé. Les participants l'ont trouvé contre-intuitif, et beaucoup d'entre eux ont confondu la signification "pour tout" de cet opérateur avec la sélection d'entités qui s'interprète avec "pour chaque".

Enseignements tirés Les abstractions de Pantagruel demandent une formation plus poussée qu'une courte présentation didactielle, ainsi qu'un support visuel plus adapté. Ce constat nous a amenés à améliorer l'éditeur Pantagruel, en complétant l'assistance fournie par l'éditeur avec une traduction sommaire en langage naturel des conditions, des actions et des règles. Par exemple, en passant la souris sur un contrôleur de règle, l'utilisateur accède à une traduction de la règle qui explicite l'effet de filtrage des capteurs et les dépendances en combinant des mots tels que *ce/cette/ces*, *certain(e)s*, *n'importe quel*, *une*. Ainsi, la règle Lampe.pièce **is** Badge.pièce ⊣[ET]⊢→ Lampe.allumer() se traduit en : "quand la **pièce** de *certaines* **lampes** est égale à la **pièce** *d'un* **Badge**, faire **allumer** *sur ces* **Lampes**". Ainsi, cinq des six participants ayant bénéficié de cette amélioration ont été capables de compléter les règles R1 à R5 sans intervention et plus rapidement que les précédents (gain moyen de 2 minutes). Cependant, cette traduction rudimentaire doit être améliorée (par exemple en prenant en compte la signification des entités au moyen d'une ontologie, comme évoqué dans la section 6.2.4.3). Enfin, suite aux difficultés de compréhension de l'opérateur ALL, nous l'avons déplacé dans la colonne des contrôleurs, comme présenté dans le chapitre 5. Bien que sa position visuelle nous semble plus cohérente avec la sémantique de cet opérateur, une nouvelle évaluation reste nécessaire.

8.3.4.3 Résumé

Ces résultats montrent que Pantagruel tend vers l'accessibilité pour des programmeurs novices. Cependant, iCAP [DSSK06], que nous avons évoqué dans la section 8.1.2, et qui est basé sur des concepts visuels similaires, démontre une meilleure accessibilité. En particulier, les participants à l'évaluation d'iCAP ont résolu deux fois plus rapidement des règles similaires à celles du tableau 8.2. Cependant, iCAP n'introduit pas les abstractions de classe proposées par Pantagruel. De plus, comme les participants n'ont bénéficié que d'une introduction succincte au langage, nous pensons que nos résultats restent plutôt encourageants.

Néanmoins, des améliorations doivent être envisagées pour rendre les abstractions de Pantagruel plus intuitives au programmeur novice, et à terme à l'utilisateur final. L'une de ces améliorations est de fournir un support de test. Le compilateur vers DiaSim que nous avons implémenté postérieurement à cette évaluation d'accessibilité permet désormais de tester dy-

namiquement les règles créées. Pour compléter ce support, une visualisation des opérations de filtrage pourrait être également proposée, étant donné un ensemble de valeurs “test” particulières pour les attributs des entités. Pour cela, il semble naturel de tirer profit de la visualisation 2D fournie par l'interface d'édition de DiaSim.

8.4 Synthèse

Dans ce chapitre, nous avons comparé Pantagruel à des approches existantes partageant tout ou partie de ses concepts : la taxonomie, les concepts visuels, la méthodologie et la vérification. Nous avons ensuite établi le périmètre d'expressivité de Pantagruel qui englobe les quatre champs d'applications que nous souhaitions couvrir. Enfin, nous avons mené une étude d'accessibilité auprès de programmeurs novices. Ces deux travaux complètent ainsi les arguments fournis par les comparaisons aux approches existantes pour justifier la position de Pantagruel, en tant que langage offrant un compromis entre expressivité et accessibilité.

Cependant, ces évaluations montrent également qu'il est difficile d'attendre des utilisateurs finaux une prise en main immédiate des outils dont les paradigmes sont d'une certaine façon programmatiques. Pour faciliter l'apprentissage de ces outils, des recherches récentes sur les approches de génie logiciel pour l'utilisateur final ont été faites [BCR04, KAB+10]. Ces travaux présentent différentes techniques (tests interactifs, ou outils de débogage appropriés) pour aider l'utilisateur final à créer des applications correctes avec des outils visuels. Un autre moyen de faciliter la prise en main des outils serait de proposer différents niveaux de programmation, mis à disposition de l'utilisateur au fur et à mesure qu'il se familiarise avec ces outils. Enfin, une autre piste que nous avons déjà évoquée dans la section 8.3.4.2 est la programmation par l'exemple. Le langage Chimera [Kur93] en fait une utilisation judicieuse : il utilise le principe des macros pour programmer des applications d'édition graphique (par exemple, une opération de déplacement sur un objet particulier) basées sur des exemples. Ces exemples sont ensuite généralisés pour s'appliquer à une classe d'objets. Ce mécanisme pourrait largement profiter à Pantagruel pour la définition de règles à base de classes.

9 Conclusion

L'un des défis émergents de l'informatique ubiquitaire est de faciliter l'adaptation des applications aux nombreux besoins des utilisateurs concernant l'amélioration de leur vie quotidienne. Nous avons proposé de relever ce défi en concevant une méthodologie outillée par un langage visuel dédié à l'orchestration d'entités, qui soit expressif, accessible, et qui permette d'améliorer la fiabilité des applications développées.

9.1 Contributions

Une méthodologie outillée pour l'expert-métier

La contribution principale de cette thèse est de poser les bases d'une méthodologie outillée pour guider l'expert-métier dans le développement d'applications. Cette méthodologie, dédiée au domaine de l'orchestration d'entités, propose une passerelle entre l'expression des besoins et le déploiement des applications. Cette passerelle est basée sur la décomposition d'un but en sous-buts et en objets élémentaires. Elle permet de faciliter l'adaptation des applications à de nouveaux besoins, notamment en explicitant les relations entre les sous-buts et les applications qui les réalisent.

Un langage visuel paramétré par une taxonomie

Pour outiller notre méthodologie, nous avons créé un langage visuel à base de règles, paramétré par une taxonomie. Cette taxonomie permet de modéliser les informations de contexte caractérisant un environnement ubiquitaire. Ce langage permet d'exprimer des logiques d'orchestration de manière concise sur les entités d'un environnement ubiquitaire. Pour cela, nous avons proposé une représentation visuelle originale du paradigme à base de règles et centrée sur les entités. Une autre contribution a été de définir la sémantique dénotationnelle de ce langage et de permettre ainsi de raisonner sur la logique d'orchestration.

Une approche de programmation dirigée par la vérification

Pour améliorer la fiabilité des programmes Pantagruel, nous avons proposé une approche de programmation dirigée par l'expression d'invariants qui correspondent à des motifs de règles d'orchestration. Pour cela, nous avons défini deux types d'invariants, les invariants restrictifs et extensifs. Ces invariants sont dotés d'une sémantique particulière permettant de vérifier s'ils sont garantis par un programme d'orchestration donné. Une autre contribution a été d'utiliser une sémantique abstraite pour vérifier que les règles d'un programme ne sont pas interférentes.

La vérification de cette propriété est rendue possible grâce aux informations fournies dans la taxonomie qui sert de paramètre au langage d'orchestration.

Une évaluation de l'expressivité et de l'accessibilité du langage

Nous avons évalué deux facettes de Pantagruel : son expressivité et son accessibilité. Nous en résumons ici les résultats.

Expressivité Nous avons développé, compilé, et testé des applications Pantagruel sur trois champs de l'informatique ubiquitaire : la domotique, la gestion d'information, et l'assistance à la personne. A l'issue de cette expérimentation, nous avons identifié un espace de modélisation d'entités couvert par le langage de taxonomie. Cet espace de modélisation nous a permis de valider l'expressivité de Pantagruel vis-à-vis de ces trois champs. En particulier, nous avons montré que l'expressivité du langage d'orchestration est favorisée par celle du langage de taxonomie.

Accessibilité Un langage dédié est destiné à l'utilisateur du domaine considéré, lequel n'est pas systématiquement un expert en programmation : les constructions d'un tel langage visent à rendre accessible l'activité de programmation. Nous avons mesuré l'accessibilité du langage d'orchestration en menant une étude auprès de programmeurs novices. Cette étude s'est avérée être positive, et nous a également poussés à améliorer le langage d'orchestration et son éditeur visuel.

9.2 Travaux futurs

A moyen terme, nous souhaitons améliorer la vérification des programmes Pantagruel. A plus long terme, nous voudrions nous intéresser à la modularité du langage et à l'amélioration de son accessibilité, à la fois au travers d'une représentation plus intuitive du langage d'orchestration et d'un travail plus approfondi sur la méthodologie. En particulier, il s'agirait d'explorer les moyens de faciliter la construction d'une taxonomie.

9.2.1 Analyse

Définition d'une sémantique abstraite de Pantagruel

Afin de définir un procédé de vérification fondé sur la sémantique de Pantagruel nous souhaitons utiliser les techniques d'interprétation abstraite [CC77]. Ces techniques permettent de définir une sémantique abstraite d'un langage et une relation formelle de celle-ci avec la sémantique concrète. Cette relation (définie au moyen de ce qu'on appelle des connexions de Galois), permet notamment de prouver que la sémantique abstraite couvre bien l'ensemble des exécutions pouvant être engendrées par la sémantique concrète. Pour ce faire nous pourrions également nous baser sur les récents travaux de D.Schmidt [Sch09], dans lesquels il définit une dérivation mécanique d'une sémantique dénotationnelle vers une interprétation abstraite.

Cette sémantique permettrait de préciser les conditions pour lesquelles les propriétés que nous avons définies ne sont pas garanties. Il s'agirait notamment d'intégrer la notion de changement

de contexte : en effet, puisqu'une règle ne peut être exécutée qu'une fois jusqu'à ce que ses prédicats soient à nouveau insatisfaits (lorsqu'ils n'impliquent pas l'opérateur changed), cette information est essentielle pour par exemple détecter un nombre minimal de mémoires pour lesquelles on est sûr que la propriété de non-interférence n'est pas respectée.

De plus, il s'agira de définir, en utilisant les informations fournies dans la taxonomie sur les variables write, les dépendances entre deux états ou mémoires d'un programme. Ces dépendances permettraient également de préciser la vérification des propriétés sur un programme.

En plus d'intégrer le changement de contexte, un moyen supplémentaire de préciser les informations de vérification serait de compléter la taxonomie en faisant des hypothèses sur l'implémentation des méthodes, à savoir la manière dont leurs effets sont calculés, en fonction ou non de leurs paramètres d'entrée.

Utilisation du model-checking

Le *model checking* est une technique populaire de vérification de systèmes dynamique. Pour vérifier les propriétés d'invariants de Pantagruel, nous pourrions aussi utiliser TLA+ [Lam02], un langage de spécification pour décrire et raisonner sur des systèmes réactifs et concurrents à états finis (en particulier, vérifier des propriétés de vivacité et de sûreté). TLA+ est basé sur la théorie des ensembles et la logique temporelle des actions (TLA) [Lam94]. Pantagruel étant un langage à base de règles, la traduction des programmes en TLA+ se ferait assez naturellement, de même que l'expression des invariants d'actions. Un type de résultat que peuvent produire les outils manipulant TLA+ (comme TLC ou *Temporal Logic Compiler*) est un contre-exemple pour lequel une propriété n'est pas satisfaite.

Afin raisonner sur des états finis et de limiter le risque d'explosion d'états, en particulier du fait de la nature imprévisible des variables volatile, il serait en outre nécessaire de définir des *fonctions d'abstraction* modélisant une évolution spécifique de ces variables. Quelque soit la technique de vérification, ces fonctions permettraient de produire des résultats plus précis.

9.2.2 Langage

Modularité du langage d'orchestration

A l'heure actuelle, la modularité de Pantagruel est uniquement structurelle : les règles d'orchestration peuvent être regroupées en scénarios mais, à l'exécution, ces programmes sont considérés comme une application monolithique.

A long terme, nous souhaiterions intégrer la modularité introduite dans la méthodologie au travers de la description de situations, afin de pouvoir exprimer des stratégies d'exécution capables de coordonner les applications associées à ces situations. Une idée intéressante à explorer serait l'intégration d'une architecture à plusieurs niveaux, comme la *subsumption architecture* proposée dans le domaine de la programmation du contrôle des robots [Bro03]. Dans une telle architecture, chaque niveau est associé à un niveau de compétence, ou à un but relatif au comportement du robot (contourner des obstacles, explorer l'environnement, ou exécuter une tâche particulière). Une telle architecture présente plusieurs avantages : elle permet de définir des relations d'ordre non plus au niveau des règles, mais au niveau des programmes. Ces

relations d'ordre ou priorités peuvent être utilisées pour coordonner les programmes définis à chaque niveau, par exemple en inhibant certains comportements, ou en les différant (ce que nous appellerions stratégies).

Dans Pantagruel, chaque niveau pourrait correspondre à une *situation* décrite dans la méthodologie. La séparation de niveaux basique consisterait à distinguer les situations exceptionnelles des situations normales ; par exemple, l'application de la douche d'Henrick, pourrait être définie par deux niveaux : un niveau pour le cas d'une chute, et un niveau pour le fonctionnement normal. Un autre exemple serait une application de sécurité dans un bâtiment public, où le déverrouillage des portes serait conditionné par des priorités de type incendie, intrusion, ou par des conditions d'accès restreint.

Cette recherche pose de nouveaux défis concernant la sémantique, les techniques d'interprétation du langage, ou encore l'analyse de programmes Pantagruel : comment ordonnancer l'exécution de ces applications en s'assurant que leur comportement reste cohérent ? Dans le cas contraire, par exemple lorsque deux applications interfèrent, comment traiter cette interférence ? Une autre piste intéressante pour traiter ces problèmes serait d'explorer le principe des tours d'interprètes [DM88], où chaque étage pourrait correspondre à un niveau d'application spécifique, associé à une stratégie d'exécution propre.

Modes d'exécution

La sémantique de Pantagruel repose sur un mode d'itération parallèle et synchrone. Dans la réalité, les systèmes réactifs ne sont pas toujours en mesure de réagir de manière simultanée aux informations de contexte apparaissant dans l'environnement. Il serait alors intéressant de définir une sémantique alternative de Pantagruel, modélisant le mode d'exécution asynchrone, plus complexe mais également plus réaliste dans certaines situations.

9.2.3 Accessibilité et méthodologie

Accessibilité à l'utilisateur final : un langage pivot ?

D'après notre évaluation, le langage d'orchestration de Pantagruel s'avère être assez accessible au programmeur novice. Cependant, il reste programmatique, notamment par l'utilisation des classes et des méthodes avec paramètres. De plus, nos discussions auprès de deux éducatrices spécialisées ont montré que le développement centré sur les entités n'est pas toujours conforme au mode de pensée d'un expert-métier. En particulier, il n'est pas toujours intuitif d'imposer systématiquement de se référer aux attributs utilisés dans les capteurs pour exprimer l'état de l'environnement, alors que l'état pourrait être défini en dehors de l'"action" de recueillir les données par les capteurs. L'exemple basique est l'information accédée via les capteurs physiques : ainsi, il est plus intuitif de parler directement de chaleur que d'interroger un capteur sur la température détectée.

Afin d'éviter d'introduire un pas de compilation trop grand entre l'expression des applications et leur déploiement, une idée serait d'utiliser Pantagruel comme langage pivot pour préserver le lien entre les briques de bases que constituent les entités et les outils de développement beaucoup plus accessibles dédiés aux différents types d'experts-métier. De cette manière, on

pourrait continuer à tirer profit des vérifications apportées par le langage et de sa compilation vers la plate-forme DiaSpec [CBLC09].

Intégration de la construction de la taxonomie dans la méthodologie

Un point difficile que nous n'avons pas abordé dans la méthodologie est la construction de la taxonomie. Dans la méthodologie, cette étape est accomplie par un *expert des entités*, qui dispose de l'expertise nécessaire pour manipuler les composants logiciels, mais aussi pour développer les pilotes de périphériques matériels permettant de donner accès à leurs fonctionnalités et à leurs informations. Pour cette étape de construction, il est nécessaire de favoriser la collaboration entre les différents acteurs de la méthodologie : les interfaces d'interaction des entités doivent fournir les données requises pour développer les applications conformes aux objectifs des experts-métiers, et doivent pouvoir également être facilement manipulées via le langage d'orchestration Pantagruel. Dow *et al.* ont proposé des principes de conception d'applications ubiquitaires mettant en évidence l'importance de la collaboration entre les différents acteurs du cycle de développement d'une application [DSLL06]. L'exploration de cette voie permettrait probablement d'apporter de nouvelles pistes de recherche en génie logiciel.

Un tel travail aurait des répercussions sur l'accessibilité de la programmation des applications. En effet, une collaboration entre experts des entités et experts-métier permettrait entre autres d'identifier de nouveaux moyens d'accéder aux informations de la taxonomie ; puisque l'expert des entités a une connaissance précise des fonctionnalités, il pourrait, avec l'aide de l'expert-métier et de besoins génériques exprimés par ce dernier, enrichir une bibliothèque d'entités avec des informations "sémantiques" (documentation, dépendances entre les entités, conditions d'utilisation). Il serait alors intéressant d'étendre la taxonomie et d'utiliser les outils de raisonnement associés aux ontologies pour offrir des procédés de programmation non plus dirigés par les entités, mais par les informations sémantiques structurées via un langage d'ontologies.

Bibliographie

[BA96] Pascal Bouvry et Farhad Arbab : Visifold : A visual environment for a coordination language. *In COORDINATION '96 : Proceedings of the First International Conference on Coordination Languages and Models*, pages 403–406, London, UK, 1996. Springer-Verlag.

[BAD+01] Margaret M. Burnett, J. William Atwood, Rebecca Walpole Djang, James Reichwein, Herkimer J. Gottfried et Sherry Yang : Forms/3 : A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 11(2):155–206, 2001.

[BB94] Margaret M. Burnett et Marla Baker : A classification system for visual programming languages. *Journal of Visual Languages & Computing*, 5:287–300, 1994.

[BCR04] Margaret M. Burnett, Curtis Cook et Gregg Rothermel : End-user software engineering. *Communications of the ACM*, 47(9):53–58, 2004.

[BGL95] Margaret M. Burnett, Adele Goldberg et Ted G. Lewis, éditeurs. *Visual object-oriented programming : concepts and environments*. Manning Publications Co., Greenwich, CT, USA, 1995.

[BJC09] Julien Bruneau, Wilfried Jouve et Charles Consel : Diasim, a parameterized simulator for pervasive computing applications. *In Mobiquitous'09 : Proceedings of the 6th International Conference on Mobile and Ubiquitous Systems : Computing, Networking and Services*, Toronto, CAN, 2009. ICST/IEEE.

[BKM08] Aaron Bangor, Philip T. Kortum et James T. Miller : An empirical evaluation of the system usability scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, 2008.

[BL93] Brigham Bell et Clayton Lewis : Chemtrains : A language for creating behaving pictures. *In VL'93 : Proceedings of the IEEE Symposium on Visual Languages*, pages 188–195, 1993.

[BLO98] Saddek Bensalem, Yassine Lakhnech et Sam Owre : Invest : A tool for the verification of invariants. *In CAV'98 : Computer Aided Verification*, pages 505–510, 1998.

[BM95] Margaret M. Burnett et David W. McIntyre : Visual programming - guest editors' introduction. *IEEE Computer*, 28(3):14–16, 1995.

[BMR99] Francesca Benzi, Dario Maio et Stefano Rizzi : Visionary : a viewpoint-based visual language for querying relational databases. *Journal of Visual Languages & Computing*, 10(2):117 – 145, 1999.

[Bou89] Gérard Boudol : Atomic actions. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 38:136–144, 1989.

[BRJ99] Grady Booch, James E. Rumbaugh et Ivar Jacobson : The unified modeling language user guide. *J. Database Manag.*, 10(4):51–52, 1999.

[Bro96] John Brooke : Sus-a quick and dirty usability scale. *Jordan, P., Thomas, B. and Weerdmeester, B. (eds.). Usability Evaluation in Industry.*, 1996.

[Bro03] Rodney A. Brooks : A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, January 2003.

[Car02] Stefan Parry Carmien : *Socio-Technical Environments Supporting Distributed Cognition for Persons with Cognitive Disabilities.* Thèse de doctorat, University of Colorado at Boulder, Boulder, CO, USA, 2002.

[CBLC09] Damien Cassou, Benjamin Bertran, Nicolas Loriant et Charles Consel : A generative programming approach to developing pervasive computing systems. *In GPCE '09 : Proceedings of the eighth international conference on Generative programming and component engineering*, pages 137–146, New York, NY, USA, 2009. ACM.

[CC77] Patrick Cousot et Radhia Cousot : Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith et Sanjiva Weerawarana : Web service definition language (wsdl). Rapport technique, World Wide Web Consortium, March 2001.

[CDM+00] Keith Cheverst, Nigel Davies, Keith Mitchell, Adrian Friday et Christos Efstratiou : Developing a context-aware electronic tourist guide : some issues and experiences. *In CHI '00 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 17–24, New York, NY, USA, 2000. ACM.

[CE00] Krzysztof Czarnecki et Ulrich W. Eisenecker : *Generative programming : methods, tools, and applications.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CES86] Edmund. M. Clarke, E. Allen Emerson et A. Prasad Sistla : Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languanges and Systems*, 8(2):244–263, 1986.

[CG90] J.R. Cordy et T.C.N. Graham : GVL : a graphical, functional language for the specification of output in programming languages. *In ICCL'90 : International Conference on Computer Languages*, pages 11 –22, mar. 1990.

[CG00] Luca Cardelli et Andrew D. Gordon : Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[CM98] Charles Consel et Renaud Marlet : Architecturing software using a methodology for language development. *In Proceedings of the 10 th International Symposium on Programming Language Implementation and Logic Programming, number 1490 in Lecture Notes in Computer Science*, pages 170–194, 1998.

[CRS98] Philip T. Cox, Christopher C. Risley et Trevor J. Smedley : Toward concrete representation in visual languages for robot control. *Journal of Visual Languages & Computing*, 9(2):211 – 239, 1998.

[DAS01] Anind K. Dey, Gregory D. Abowd et Daniel Salber : A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2):97–166, 2001.

[DC10] Zoé Drey et Charles Consel : A visual, open-ended approach to prototyping ubiquitous computing applications. *In PerCom Workshops of the Eigth Annual IEEE International Conference on Pervasive Computing and Communications*, pages 817–819, Mannheim, Germany, 2010. IEEE.

[DM88] Olivier Danvy et Karoline Malmkjaer : Intensions and extensions in a reflective tower. *In LFP '88 : Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 327–341, New York, NY, USA, 1988. ACM.

[DMC09] Zoé Drey, Julien Mercadal et Charles Consel : A taxonomy-driven approach to visually prototyping pervasive computing applications. *In DSL-WC'09 : Proceedings of the 1st IFIP Working Conference on Domain-Specific Languages*, Oxford, GB, feb 2009.

[DP05] Dolev Dotan et Ron Y. Pinter : Hyperflow : An integrated visual query and dataflow language for end-user information analysis. *In VL/HCC'05 : Proceedings of the IEEE Symposium on Visual Languages and Human Computer Communication*, pages 27–34, 2005.

[DSLL06] Steven Dow, T. Scott Saponas, Yang Li et James A. Landay : External representations in ubiquitous computing design and the implications for design tools. *In DIS'06 : Conference on Designing Interactive Systems*. ACM, 2006.

[DSSK06] Anind K. Dey, Timothy Sohn, Sara Streng et Justin Kodama : iCAP : Interactive prototyping of context-aware applications. *In Pervasive'06 : 4th International Conference on Pervasive Computing*, pages 254–271. Springer, 2006.

[Ecl] Eclipse Foundation : Eclipse. `http://www.eclipse.org`.

[FNY09] Gerhard Fischer, Kumiyo Nakakoji et Yunwen Ye : Metadesign : Guidelines for supporting domain experts in software development. *IEEE Software*, 26(5):37–44, 2009.

[GB08] Saul Greenberg et Bill Buxton : Usability evaluation considered harmful (some of the time). *In CHI '08 : Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 111–120, New York, NY, USA, 2008. ACM.

[GIL+95] Jim Gindling, Andri Ioannidou, Jennifer Loh, Olav Lokkebo et Alexander Repenning : Legosheets : A rule-based programming, simulation and manipulation environment for the leg0 programmable brick. *In VL'95 : Proceedings of the IEEE Symposium on Visual Languages*, pages 172–179, 1995.

[GM82] Joseph A. Goguen et José Meseguer : Security policies and security models. *In IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GN00] Emden R. Gansner et Stephen C. North : An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.

[GP92] Thomas R.G. Green et Marian Petre : When visual programs are harder to read than textual programs. *In Human-Computer Interaction : Tasks and Organisation*, pages 167–180. G.C. Van der Veer, M. J. Tauber, S. Bagnarola and M. Antavolits (Eds.), 1992.

[GP96] T. R. G. Green et M. Petre : Usability analysis of visual programming environments : a 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7:131–174, 1996.

[Gri04] Robert Grimm : One.world : Experiences with a pervasive computing architecture. *IEEE Pervasive Computing*, 3(3):22–30, 2004.

[GSS02] David Garlan, Dan P. Siewiorek et Peter Steenkiste : Project Aura : Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1:22–31, 2002.

[GSV08] Sumit Gulwani, Saurabh Srivastava et Ramarathnam Venkatesan : Program analysis as constraint solving. *In PLDI'08 : Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 281–292. ACM, 2008.

[Har87] David Harel : Statecharts : A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HCH+03] Jan Humble, Andy Crabtree, Terry Hemmings, Karl-Petter. Åkesson, Boriana Koleva, Tom Rodden et Pär Hansson : "Playing with the Bits" user-configuration of ubiquitous domestic environments. *In UbiComp'03 : 5th International Conference on Ubiquitous Computing*, volume 2864, pages 256–263. Springer, 2003.

[HHS+99] Andy Harter, Andy Hopper, Pete Steggles, Andy Ward et Paul Webster : The anatomy of a context-aware application. *In MobiCom '99 : Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 59–68, New York, NY, USA, 1999. ACM.

[HKB+06] Björn Hartmann, Scott R. Klemmer, Michael Bernstein, Leith Abdulla, Brandon Burr, Avi Robinson-Mosher et Jennifer Gee : Reflective physical prototyping through integrated design, test, and analysis. *In UIST'06 : Proceedings of the 19th Symposium on User interface software and technology*, pages 299–308, New York, NY, USA, 2006. ACM.

[HP85] David Harel et Amir Pnueli : On the development of reactive systems. *Nato Asi Series F : Computer And Systems Sciences*, pages 477–498, 1985.

[HR85] Frederick Hayes-Roth : Rule-based systems. *Communications of the ACM*, 28(9):921–932, 1985.

[HS08] Paul Holleis et Albrecht Schmidt : Makeit : Integrate user interaction times in the design process of mobile applications. *In Pervasive'08 : Proceedings of the 6th International Conference on Pervasive Computing*, volume 5013 de *LNCS*, pages 56–74. Springer-Verlag, 2008.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace et Alan Kay : Back to the future : the story of squeak, a practical smalltalk written in itself. *SIGPLAN Not.*, 32(10):318–326, 1997.

[ISM97] Mitsuru Ikeda, Kazuhisa Seta et Riichiro Mizoguchi : Task ontology makes it easier to use authoring tools. *In IJCAI'97 : Proceedings of the 15th international joint conference on Artifical intelligence*, pages 342–347, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[KAB+10] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret M. Burnett, Martin Erwig, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Chris Scaffidi, Mary Shaw et Susan Wiedenbeck : The state of the art in end-user software engineering, 2010. to appear.

[KP05] Caitlin Kelleher et Randy Pausch : Lowering the barriers to programming : A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2):83–137, 2005.

[KRM+06] Matthias Kranz, Radu Bogdan Rusu, Alexis Maldonado, Michael Beetz et Albrecht Schmidt : A player/stage system for context-aware intelligent environments. *In UbiSys '06 : Proceedings of the System Support for Ubiquitous Computing Workshop*, pages 1–7, 2006.

[KSD+03] Mohan Kumar, Behrooz A. Shirazi, Sajal K. Das, Byung Y. Sung, David Levine et Mukesh Singhal : Pico : A middleware framework for pervasive computing. *IEEE Pervasive Computing*, 2:72–79, 2003.

[KSS07] Matthias Kranz, Wolfgang Spiessl et Albrecht Schmidt : Designing ubiquitous computing systems for sports equipment. *In PerCom'07 : Proceedings of the 5th International Conference on Pervasive Computing and Communications*, pages 79–86. IEEE Computer Society, 2007.

[Kur93] David Joshua Kurlander : *Graphical editing by example.* Thèse de doctorat, Columbia University, New York, NY, USA, 1993. Advisor-Feiner, Steven.

[Lam94] Leslie Lamport : The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam02] L. Lamport : *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[LBK+02] Anthony LaMarca, Waylon Brunette, David Koizumi, Matthew Lease, Stefan B. Sigurdsson, Kevin Sikorski, Dieter Fox et Gaetano Borriello : Plantcare : An investigation in practical ubiquitous systems. *In UbiComp'02 : 4th International Conference on Ubiquitous Computing*, pages 316–332, 2002.

[LD99] James A. Landay et Richard C. Davis : Making sharing pervasive : ubiquitous computing for shared note taking. *IBM Systems Journal*, 38(4):531–550, 1999.

[Leg] Lego : Lego mindstorms NXT 2.0. `http://mindstorms.lego.com/`.

[LHL04] Yang Li, Jason I. Hong et James A. Landay : Topiary : a tool for prototyping location-enhanced applications. *In UIST'04 : Proceedings of the 17th Symposium on User Interface Software and Technology*, pages 217–226. ACM, 2004.

[LL08] Yang Li et James A. Landay : Activity-based prototyping of ubicomp applications for long-lived, everyday human activities. *In CHI'08 : Proceedings of the 2008 Conference on Human Factors in Computing Systems*, pages 1303–1312. ACM, 2008.

[LS09] James R. Lewis et Jeff Sauro : The factor structure of the system usability scale. *In HCD'09 : Proceedings of the 1st International Conference on Human Centered Design*, pages 94–103, Berlin, Heidelberg, 2009. Springer-Verlag.

[Mac09] Matt MacLaurin : Kodu : end-user programming and design for games. *In FDG '09 : Proceedings of the 4th International Conference on Foundations of Digital Games*, pages xviii–xix, New York, NY, USA, 2009. ACM.

[Mar] Margaret M. Burnett : Visual language research bibliography. `http://web.engr.oregonstate.edu/~burnett/vpl.html`.

[McK99] Dorothy McKinney : Impact of commercial off-the-shelf (cots) software on the interface between systems and software engineering. *In ICSE '99 : Proceedings of the 21st international conference on Software engineering*, pages 627–628, New York, NY, USA, 1999. ACM.

[Mel96] Jim Melton : Sql language summary. *ACM Computing Surveys*, 28(1):141–143, 1996.

[MHS05] Marjan Mernik, Jan Heering et Anthony M. Sloane : When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[Mic] Microsoft Corporation : The Microsoft Visual Programming Language. `http://msdn.microsoft.com/en-us/library/bb483088.aspx`.

[MPCL08] Julien Mercadal, Nicolas Palix, Charles Consel et Julia L. Lawall : Pantaxou : a domain-specific language for developing safe coordination services. *In GPCE'08 : Proceedings of the eighth international conference on Generative programming and component engineering*, pages 149–160, 2008.

[Mye86] Brad A. Myers : Visual programming, programming by example, and program visualization : a taxonomy. *SIGCHI Bulletin*, 17(4):59–66, 1986.

[Mye91] Brad A. Myers : Graphical techniques in a spreadsheet for specifying user interfaces. *In CHI '91 : Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 243–249, New York, NY, USA, 1991. ACM.

[Nat] Education NATIONALE : Ressources pour le lycée - algorithmique. `http://media.eduscol.education.fr/file/Programmes/17/8/Doc_ress_algo_v25_109178.pdf`.

[NES08] Mark W. NEWMAN, Ame ELLIOTT et Trevor F. SMITH : Providing an integrated user experience of networked media, devices, and services through end-user composition. *In Pervasive'08 : 6th International Conference on Pervasive Computing*, pages 213–227. Springer, 2008.

[NMB09] Christoph NEUMANN, Ronald A. METOYER et Margaret M. BURNETT : End-user strategy programming. *Journal of Visual Languages & Computing*, 20(1):16–29, 2009.

[NYT+06] Hiroshi NISHIKAWA, Shinya YAMAMOTO, Morihiko TAMAI, Kouji NISHIGAKI, Tomoya KITANI, Naoki SHIBATA, Keiichi YASUMOTO et Minoru ITO : Ubireal : Realistic smartspace simulator for systematic testing. *In Ubicomp'06 : 8th International Conference on Ubiquitous Computing*, pages 459–476, 2006.

[PAR96] Jordi PUIGSEGUR, Jaume AGUSTI et Dave ROBERTSON : A visual logic programming language. *In VL '96 : Proceedings of the 1996 IEEE Symposium on Visual Languages*, page 214, Washington, DC, USA, 1996. IEEE Computer Society.

[Pfe98] Joseph J. PFEIFFER JR. : Altaira : A rule-based visual language for small mobile robots. *Journal of Visual Languages & Computing*, 9(2):127–150, 1998.

[PSAC97] Jordi PUIGSEGUR, W. Marco SCHORLEMMER et Jaume AGUSTÍ-CULLELL : From queries to answers in visual logic programming. *In Proceedings of the IEEE Symposium on Visual Languages*, pages 102–109, 1997.

[PVS] PVS : PVS Specification and Verification System. `http://pvs.csl.sri.com/`.

[RC08] Anand RANGANATHAN et Roy H. CAMPBELL : Provably correct pervasive computing environments. *PerCom'08 : Proceedings of the 6th International Conference on Pervasive Computing and Communications*, 0:160–169, 2008.

[RCAM+05] Anand RANGANATHAN, Shiva CHETAN, Jalal AL-MUHTADI, Roy H. CAMPBELL et M. D. MICKUNAS : Olympus : A high-level programming model for pervasive computing environments. *In PerCom'05 : Proceedings of the 3rd International Conference on Pervasive Computing and Communications*, pages 7–16. IEEE Computer Society, 2005.

[Rep93] Alexander REPENNING : *Agentsheets : a tool for building domain-oriented dynamic, visual environments*. Thèse de doctorat, University of Colorado at Boulder, Boulder, CO, USA, 1993.

[RHC+02] Manuel ROMÁN, Christopher HESS, Renato CERQUEIRA, Roy H. CAMPBELL et Klara NAHRSTEDT : Gaia : A middleware infrastructure to enable active spaces. *IEEE Pervasive Computing*, 1:74–83, 2002.

[RMMH+09] Mitchel RESNICK, John MALONEY, Andrés MONROY-HERNÁNDEZ, Natalie RUSK, Evelyn EASTMOND, Karen BRENNAN, Amon MILLNER, Eric ROSENBAUM, Jay SILVER, Brian SILVERMAN et Yasmin KAFAI : Scratch : Programming for all. *Communications of the ACM*, 52(11):60–67, 2009.

[Rob00] François Robert : *Les Systèmes Dynamiques Discrets*. Springer-Verlag, 2000.

[RSB05] Barbara G. Ryder, Mary Lou Soffa et Margaret Burnett : The impact of software engineering research on modern progamming languages. *ACM Transactions on Software Engineering and Methodology*, 14(4):431–477, 2005.

[SAW94] Bill Schilit, Norman Adams et Roy Want : Context-aware computing applications. *In Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, 1994.

[SC95] Scott B. Steinman et Kevin G. Carver : *Visual Programming with Prograph CPX*. Manning Publications Co., Greenwich, CT, USA, 1995.

[Sch86] David A. Schmidt : *Denotational semantics : a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.

[Sch09] David A. Schmidt : Abstract interpretation from a denotational-semantics perspective. *Electronic Notes in Theoretical Computer Science*, 249:19–37, 2009.

[SCS94] David Canfield Smith, Allen Cypher et James C. Spohrer : Kidsim : Programming agents without a programming language. *Communications of the ACM*, 37(7):54–67, 1994.

[Shu99] Nan C. Shu : Visual programming : Perspectives and approaches. *IBM Systems Journal*, 38(2/3):199–221, 1999.

[Shu06] Boris Shulman : RFID for Patient Flow Management in Emergency Unit. Rapport technique, IBM Corporation, `http://www.ibm.com/news/fr/fr/2006\discretionary{-}{}{}/03/\discretionary{-}{}{}cp1851.html`, 2006.

[SMA02] Stoffler D. Sanner M.F. et Olson A.J. : Viper, a visual programming environment for python. *In 10th International Python conference*, pages 103–115, February 2002.

[Spa89] Domenic V. Sparrow, Sara S. ; Cicchetti : *Major psychological assessment instruments, Vol. 2*, chapitre The Vineland Adaptive Behavior Scales. Allyn & Bacon, 1989.

[ST06] Albrecht Schmidt et Lucia Terrenghi : Methods and guidelines for the design and development of domestic ubiquitous computing applications. *In SAC'06 : Proceedings of the 21th Symposium on Applied Computing*, pages 1928–1929. ACM, 2006.

[Sta02] Vince Stanford : Using pervasive computing to deliver elder care. *IEEE Pervasive Computing*, 1(1):10–13, 2002.

[Sve03] Arne Svensk : Design for cognitive assistance. *In Human Factors and Ergonomics Society Europe Annual Meeting (HFES)*, 2003.

[Ten76] R. D. Tennent : The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, 1976.

[THA04] Khai N. Truong, Elaine M. Huang et Gregory D. Abowd : CAMP : A magnetic poetry interface for end-user programming of capture applications for the home. *In UbiComp'04 : 6th International Conference on Ubiquitous Computing*, pages 143–160. Springer, 2004.

[The] The Coq Development TEAM : The Coq Proof Assistant. `http://coq.inria.fr/`.

[TMC99] Scott Thibault, Renaud Marlet et Charles Consel : Domain-specific languages : From design to implementation application to video device drivers generation. *IEEE Transactions in Software Engineering*, 25(3):363–377, 1999.

[TR86] Steven L. Tanimoto et Marcia S. Runyan : Play : An iconic programming system for children. *In Visual Languages*, pages 191–205, 1986.

[vDKV00] Arie van Deursen, Paul Klint et Joost Visser : Domain-specific languages : an annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[vG03] Jason van Gumster : Blender as an educational tool. *In SIGGRAPH Educators Program*, 2003.

[Whi04] Stephen A. White : Business Process Modeling Notation (BPMN), Version 1.0. Business Process Management Initiative, `http://www.bpmi.org`, May 2004.

[Wil09] Ashley Williams : User-centered design, activity-centered design, and goal-directed design : a review of three methods for designing web applications. *In SIGDOC '09 : Proceedings of the 27th ACM international conference on Design of communication*, pages 1–8, New York, NY, USA, 2009. ACM.

[WKU+07] Torben Weis, Mirko Knoll, Andreas Ulbrich, Gero Muhl et Alexander Brandle : Rapid prototyping for pervasive applications. *IEEE Pervasive Computing*, 6(2):76–84, 2007.

[YCM09] Tom Yeh, Tsung-Hsiang Chang et Robert C. Miller : Sikuli : using GUI screenshots for search and automation. *In UIST'09 : Proceedings of the 22th Symposium on User interface software and technology*, pages 183–192, 2009.

10 Annexe

System Usability Scale

Instructions: Pour chacune des affirmations ci-dessous, cochez une case qui décrit le mieux vos réactions à l'exercice d'utilisation de Pantagruel que vous venez d'effectuer

		Vraiment pas d'accord				Vraiment d'accord
1.	Je pense que j'aimerais utiliser ce logiciel fréquemment pour prototyper des applications ubiquitaires.	☐	☐	☐	☐	☐
2.	J'ai trouvé les concepts du logiciel (sensor/actuator/etc.) inutilement complexes ou trop nombreux.	☐	☐	☐	☐	☐
3.	J'ai trouvé ce loigiciel facile à utiliser.	☐	☐	☐	☐	☐
4.	Je pense que j'aurais besoin d'assistance pour être capable de créer d'autres applications.	☐	☐	☐	☐	☐
5.	J'ai trouvé les fonctionnalités offertes par ce logiciel bien intégrées (e.g adaptées aux concepts.)	☐	☐	☐	☐	☐
6.	J'ai trouvé que ce logiciel présentait beaucoup d'incohérences à l'utilisation.	☐	☐	☐	☐	☐
7.	J'imagine que la plupart des gens pourraient apprendre très rapidement à utiliser ce logiciel.	☐	☐	☐	☐	☐
8.	J'ai trouvé ce logiciel lourd/peu commode à utliser.	☐	☐	☐	☐	☐
9.	Je me suis senti à l'aise en utilisant ce logiciel.	☐	☐	☐	☐	☐
10.	J'ai eu l'impression de devoir apprendre beaucoup de choses pour démarrer ce logiciel.	☐	☐	☐	☐	☐

FIG. 10.1: Echelle d'accessibilité de Pantagruel

SCORE SUS																					
Q° | Etudiant →	e1	e2	e3	e4	e7	e13	e5	e6	e8	e9	e11	e12	e10	e14	e15	e16	e17	e18	SUS<3	SUS>3	SUS=3
1	3,0	5,0	4,0	4,0	4,0	5,0	3,0	4,0	3,0	4,0	4,0	5,0	5,0	5,0	4,0	3,0	5,0	4,0	0	15	3
2	4,0	1,0	3,0	1,0	3,0	1,0	2,0	1,0	3,0	1,0	2,0	1,0	2,0	3,0	2,0	2,0	2,0	1,0	13	1	4
3	5,0	5,0	4,0	4,0	3,0	3,0	3,0	3,0	2,0	3,0	4,0	4,0	4,0	4,0	3,0	5,0	4,0	3,0	1	10	7
4	4,0	3,0	2,0	1,0	5,0	1,0	4,0	4,0	4,0	1,0	5,0	1,0	3,0	4,0	2,0	3,0	3,0	4,0	6	8	4
5	4,0	5,0	4,0	4,0	4,0	4,0	4,0	3,0	5,0	4,0	3,0	5,0	4,0	5,0	3,0	5,0	4,0	4,0	0	15	3
6	3,0	1,0	4,0	1,0	2,0	2,0	2,0	1,0	2,0	1,0	3,0	3,0	2,0	2,0	2,0	1,0	3,0	1,0	13	1	4
7	5,0	4,0	1,0	2,0	3,0	3,0	2,0	4,0	2,0	3,0	5,0	4,0	1,0	2,0	2,0	5,0	4,0	4,0	7	8	3
8	3,0	1,0	2,0	2,0	1,0	2,0	2,0	2,0	3,0	2,0	2,0	1,0	1,0	1,0	2,0	2,0	1,0	1,0	16	0	2
9	4,0	4,0	3,0	4,0	4,0	4,0	3,0	3,0	2,0	4,0	3,0	5,0	4,0	5,0	4,0	4,0	5,0	3,0	1	12	5
10	3,0	1,0	3,0	1,0	4,0	2,0	2,0	2,0	3,0	1,0	2,0	2,0	2,0	2,0	1,0	2,0	1,0	2,0	14	1	3
Moyenne impaire	16,0	18,0	11,0	13,0	13,0	14,0	10,0	12,0	9,0	13,0	14,0	18,0	13,0	16,0	11,0	17,0	17,0	13,0			
Moyenne paire	8,0	18,0	11,0	19,0	10,0	17,0	13,0	15,0	10,0	19,0	11,0	17,0	15,0	13,0	16,0	15,0	15,0	16,0			
Score SUS	60,0	90,0	55,0	80,0	57,5	77,5	57,5	67,5	47,5	80,0	62,5	87,5	70,0	72,5	67,5	80,0	80,0	72,5	70,3		
	Ecart type e1 > e18										11,8	Ecart type (6 derniers) e10, 14, 15, 16, 17, 18							5,2		
	Score moyen (après suppression du meilleur : 69,1)										70,3	Score (6 derniers)							74,0		

FIG. 10.2: Resultats du Questionnaire pour l'evaluation de Pantagruel

Résumé

Les technologies omniprésentes dans notre environnement intègrent désormais des éléments logiciels facilitant leur utilisation. Ces technologies offrent un vaste laboratoire d'expérimentation pour la recherche et en particulier pour l'informatique appliquée. Ces technologies sont un support évident pour rendre des services aux personnes dans leur vie quotidienne. Ces services concernent divers champs d'applications, chacun servant des objectifs spécifiques : confort, sécurité, accès à l'information ou encore assistance à la personne. Puisque les applications offrant ces services sont intimement liées aux besoins des utilisateurs, il est indispensable qu'elles s'adaptent facilement à leurs besoins. Une manière de répondre à ce défi est de proposer à l'utilisateur des outils pour programmer lui-même ses applications.

Notre contribution consiste non seulement à définir un tel outil, sous la forme d'un langage visuel paramétré par un champ d'applications, mais aussi à proposer une méthodologie dont l'objectif est de guider un utilisateur dans la programmation d'applications à l'aide de ce langage. Cette méthodologie est dédiée à l'orchestration d'entités communicantes : elles représentent les technologies déployées dans nos environnements. Notre approche, associant une méthodologie à un langage, est accessible à un programmeur novice et suffisamment expressive pour traiter divers champs d'applications. Afin d'augmenter la confiance de l'utilisateur dans le développement de ses applications, nous étendons la méthodologie en proposant une approche de développement dirigée par la vérification de quelques propriétés. Cette vérification est permise par la sémantique du langage, formellement définie.
Mots clés : méthodologie, langages visuels, langages dédiés, accessibilité, assistance à la personne

Abstract

Networked technologies, omnipresent in our surroundings, have increasingly more computing power, offering interfaces to easily access their functionalities. These technologies offer a wide testing ground for research, especially in applied computer science. They form an evident assistive support to help people in their daily activities. Applications that address people needs are found in various application areas, each related to specific goals : comfort, security, information management, or assisted-living.

The goal of this thesis is to propose an approach that bridges the gap between a user requirements and the applications that satisfy them. To do so, we define Pantagruel, an expressive and accessible visual language that is parameterized by an application area. To further reconcile accessibility with expressiveness, we provide the user with a domain-specific methodology to guide the development of applications. This methodology draws a bridge between the user needs and the applications, and is strongly coupled with the language concepts. In doing so, satisfying the requirements and evolving the applications according to new requirements is facilitated. To increase the user confidence in using Pantagruel language, and because the applications aim at being seamlessly integrated in his everyday life, Pantagruel programs need to be reliable. To this end, we extend our methodology with a programming approach driven by properties. These properties can be verified using the language semantics, which is formally defined.
Keywords : methodology, visual language, domain-specific language, usability, assisted-living