Approche langage au développement logiciel : application au domaine des systèmes d'informatique ubiquitaire

Julien Mercadal

► To cite this version:

Julien Mercadal. Approche langage au développement logiciel : application au domaine des systèmes d'informatique ubiquitaire. Langage de programmation [cs.PL]. Université Sciences et Technologies - Bordeaux I, 2011. Français. NNT : . tel-00654268

HAL Id: tel-00654268
https://theses.hal.science/tel-00654268

Submitted on 21 Dec 2011

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.

N° d'ordre : 4315

THÈSE

présentée à

l'Université de Bordeaux
École Doctorale de Mathématiques et Informatique

par **Julien Mercadal**

pour obtenir le grade de

Docteur
Spécialité : Informatique

Approche langage au développement logiciel :
application au domaine des systèmes d'informatique ubiquitaire

---

Soutenue le 10 octobre 2011

Après avis de :

M.	Pierre	Cointe	Professeur, École des Mines de Nantes	Rapporteur
M.	Renaud	Marlet	Directeur de recherche, École des Ponts ParisTech	Rapporteur

Devant la commission d'examen formée de :

M.	Xavier	Blanc	Professeur, Université de Bordeaux	Président
M.	Pierre	Cointe	Professeur, École des Mines de Nantes	Rapporteur
M.	Renaud	Marlet	Directeur de recherche, École des Ponts ParisTech	Rapporteur
Mme.	Isabelle	Demeure	Professeur, Télécom ParisTech	Examinatrice
M.	Charles	Consel	Professeur, ENSEIRB	Directeur de thèse

Structure d'accueil

Équipe-projet PHOENIX
INRIA Bordeaux – Sud-Ouest
Bâtiment A29
351, cours de la Libération
F-33405 Talence cedex

Équipe Langages, Systèmes et Réseaux
LaBRI – Laboratoire Bordelais de Recherche en Informatique
Unité Mixte de Recherche CNRS (UMR 5800)
351, cours de la Libération
F-33405 Talence cedex

À Anne-Maïmiti

Remerciements

Je tiens tout d'abord à remercier les membres du jury :

- Xavier Blanc, professeur à l'Université de Bordeaux, qui m'a fait l'honneur de présider le jury de cette thèse.
- Pierre Cointe, professeur à l'École des Mines de Nantes, et Renaud Marlet, directeur de recherche à l'École des Ponts ParisTech et responsable du projet IMAGINE, qui ont accepté la lourde tâche de rapporteur. Leurs commentaires avisés et constructifs m'ont permis d'améliorer encore davantage ce document.
- Isabelle Demeure, professeur à Telecom ParisTech, qui a accepté de faire partie de ce jury et qui a montré de l'intérêt pour ce travail.
- Charles Consel, professeur des Universités à l'ENSEIRB et responsable du projet PHOENIX, qui m'a donné l'opportunité de réaliser cette thèse. «C'est le rôle essentiel du professeur d'éveiller la joie de travailler et de connaître» a écrit Albert Einstein [Ein34]. Charles a été pour moi un excellent professeur avec qui j'ai eu plaisir à travailler et à apprendre tout au long de ces années, tant sur le plan humain que scientifique. Je tiens sincèrement à le remercier pour toute l'aide et le soutien qu'il m'a apportés, ainsi que la confiance dont il a fait preuve à mon égard. Je suis heureux d'avoir été l'un de ses *padawans*.

Je tiens aussi à remercier tout particulièrement Fabien Latry dont les conseils m'ont été des plus précieux. Il m'a fait bénéficier de ses compétences et de sa rigueur, et m'a encouragé à de nombreuses reprises. Je remercie également Zoé (Zoé, Zoé) Drey pour notre collaboration sur Pantagruel, son amitié et sa bonne humeur, ainsi que Nicolas Loriant pour notre travail sur la gestion des erreurs dans un système d'informatique ubiquitaire qui a abouti à une très *belle* publication.

Je tiens également à remercier tous les membres et anciens membres de l'équipe-projet PHOENIX qui ont de près ou de loin participé à cette aventure : Laurent Réveillère, Sapan Bhatia, Fabien Latry, Laurent Burgy, Nicolas Palix, Wilfried Jouve, Julien Lancia, Sylvie Embolla, Zoé Drey, David Bromberg, Damien Cassou, Alexandre Blanquart, Benjamin Bertran, Julien Bruneau, Henner Jakob, Nicolas Loriant, Hongyu Guan, Pengfei Liu, Ghislain Deffrasnes, Emilie Balland, Quentin Enard et Stéphanie Gatti.

Je tiens aussi à remercier deux de mes enseignants de l'Université de Bordeaux, Olivier Baudon qui m'a orienté vers Charles pour mon stage de recherche en Master, et Jean Bétréma pour son amitié et son soutien. Je voudrais de plus remercier certaines personnes rencontrées durant cette thèse. Je pense notamment à Olivier Danvy, Julia Lawall et Anne-Françoise Le Meur.

Je tiens également à remercier les docteurs Richard Beyssac, Alain Castinel, François Salon et Pascal Wintringer, les infirmières Betty Barrière, Martine Laroche et Isabelle Marie, ainsi que l'ensemble du personnel de l'hôpital Bagatelle qui m'ont permis de mener à bien cette aventure.

«Si tu m'apprivoises, nous aurons besoin l'un de l'autre.» [SE43]. Merci à Collyre, Copernic et Indy pour leur compagnie et toutes leurs léchouilles.

Je remercie mes zolis parents, Frédérique et Loïc, qui m'ont si gentiment accueilli au sein de leur famille. Je remercie également mes deux beaux-frères, Erwan et Thomas, ainsi que leur famille respective, pour tous les bons moments passés à Tarnac.
Je remercie ma famille pour son soutien indéfectible. Merci à Maman d'être toujours là pour moi et à Papa de m'avoir toujours poussé à faire de mon mieux. Merci à mes deux frères, Jérôme [1] et Laurent [2], et à ma petite sœur Marion, ma Chouchou, pour leurs encouragements et notre complicité qui me tient tant à cœur. Merci aussi à mes deux belles-sœurs, Aurore et Marion, et à mes deux petites nièces, Saria et Aliénor, pour tous leurs bisous et sourires.

Enfin et surtout, je remercie ma Soussou pour sa patience, sa compréhension et tout son amour. Elle est ma raison d'être, elle est toutes mes raisons. Je t'aime.

1. Créateur du jeu de société Lutinfernal
2. Administrateur du site Koopa.fr

Résumé

Face à l'augmentation de la taille et de la complexité des systèmes logiciels, il convient de les décrire à un plus haut niveau d'abstraction (*i.e.*, au-delà du code) avant de concrètement les implémenter. Toutefois, l'utilisation de ces descriptions de haut niveau dans les processus de construction et de vérification des systèmes reste très rudimentaire, ne permettant pas de véritablement guider et faciliter le développement logiciel.

Cette thèse propose une nouvelle approche pour rendre plus simple et plus sûr le développement de systèmes logiciels. Cette approche repose sur l'utilisation de langages dédiés et sur un couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation. Elle consiste tout d'abord à décrire à un haut niveau d'abstraction différents aspects, à la fois fonctionnels et non fonctionnels, d'un système dans la couche de spécification et d'architecture. Ces descriptions sont ensuite analysées et utilisées pour personnaliser la couche d'implémentation, afin de faciliter la construction et la vérification du système logiciel.

Nous illustrons notre approche dans le domaine de l'informatique ubiquitaire. À la suite d'une analyse complète du domaine, nous avons conçu deux langages dédiés à l'orchestration d'objets communicants, Pantaχou et Pantagruel.

Mots clés

Architecture logicielle, langages dédiés, génie logiciel, informatique ubiquitaire

Abstract

The sheer size and complexity of today's software systems pose challenges for both their programming and verification, making it critical to raise the level of abstraction of software development beyond the code. However, the use of high-level descriptions in the development process still remains rudimentary, improving and guiding this process marginally.

This thesis proposes a new approach to making software development simpler and safer. This approach is based on the use of domain-specific languages and a tight coupling between a specification and architecture layer, and an implementation layer. It consists of describing functional and non-functional aspects of a software system at a high level of abstraction, using the specification and architecture layer. These high-level descriptions are then analyzed and used to customize the implementation layer, greatly facilitating the programming and verification of the software system.

We have validated our approach in the domain of pervasive computing systems development. From a complete domain analysis, we have introduced two domain-specific languages, Pantaχou and Pantagruel, dedicated to the orchestration of networked smart devices.

Keywords

Software architecture, domain-specific language, software engineering, pervasive computing

Table des matières

1 Introduction **7**
- 1.1 Thèse . . . 8
- 1.2 Contributions . . . 8
- 1.3 Organisation du document . . . 9

I Contexte **11**

2 Architecture logicielle **13**
- 2.1 Introduction . . . 13
- 2.2 Motivations . . . 14
  - 2.2.1 Séparation des préoccupations . . . 15
  - 2.2.2 Analyses . . . 15
  - 2.2.3 Style architectural . . . 16
- 2.3 Approches existantes . . . 16
  - 2.3.1 Diagrammes en «boîtes et flèches» . . . 16
  - 2.3.2 Langages d'interconnexion de modules et de description d'interface . . . 16
  - 2.3.3 Langages de description d'architecture . . . 17
  - 2.3.4 Le cas particulier de UML . . . 17
- 2.4 Limitations actuelles . . . 18
  - 2.4.1 Communications difficiles . . . 18
  - 2.4.2 Manque de cohérence . . . 18
- 2.5 Évolution récente . . . 19
- 2.6 Bilan . . . 20

3 Langages dédiés **21**
- 3.1 Introduction . . . 21
- 3.2 Motivations . . . 22
  - 3.2.1 Programmation du domaine de problèmes . . . 22
  - 3.2.2 Sûreté améliorée . . . 22
  - 3.2.3 Réutilisation systématique . . . 23
- 3.3 Processus de développement . . . 23
  - 3.3.1 Conception . . . 23
  - 3.3.2 Implémentation . . . 26
- 3.4 Bilan . . . 28

4 Étude de cas : l'informatique ubiquitaire 29
4.1 Introduction 29
4.2 Systèmes d'informatique ubiquitaire 30
4.2.1 Contexte 32
4.2.2 Découverte de services 32
4.3 Enjeux du développement 32
4.3.1 Hétérogénéité 33
4.3.2 Caractère dynamique 33
4.3.3 Fiabilité 34
4.4 État de l'art 34
4.4.1 Intergiciels distribués et *frameworks* de programmation 34
4.4.2 Langages dédiés distribués 36
4.5 Bilan 36

5 Bilan 37
5.1 Étagement du développement logiciel 37
5.2 Problématiques 38

II Approche proposée 41

6 Présentation de l'approche 43
6.1 Développement logiciel plus simple et plus sûr 43
6.2 Approche 43

7 Couplage 47
7.1 Analyse de domaine et de famille 47
7.2 Exemple de travail 49
7.3 DiaSpec, un langage de spécification et d'architecture 49
7.3.1 Définition du langage 49
7.3.2 Génération d'un *framework* de programmation dédié 53
7.3.3 Couplage avec un langage généraliste 54
7.4 Pantaχou, un langage de script 55
7.4.1 Définition du langage 55
7.4.2 Compilation 59
7.4.3 Couplage avec un langage dédié de script 59
7.5 Pantagruel, un langage visuel 60
7.5.1 Définition du langage 60
7.5.2 Compilation 64
7.5.3 Couplage avec un langage dédié visuel 65
7.6 Bilan 65

8 Gestion des erreurs 67
8.1 Problématique 67
8.2 Modèle de gestion d'erreurs 68
8.2.1 Spécifier les erreurs 68
8.2.2 Architecturer la gestion des erreurs 70

8.2.3 Implémenter la gestion des erreurs 73
8.3 Implémentation 76
8.3.1 Signalisation des erreurs 76
8.3.2 Propagation des erreurs 77
8.4 Travaux connexes 78
8.5 Bilan 80

9 Conclusion **81**
9.1 Contributions 81
9.2 Perspectives 82

10 Publications **85**

Bibliographie **87**

Table des figures

2.1 Architecture logicielle comme point pivot 14
2.2 Architecture d'un compilateur 15

3.1 Processus de développement d'un langage dédié 24

4.1 Évolution de l'utilisation des ordinateurs 30
4.2 De l'informatique à l'intelligence ambiante 30
4.3 Fonctionnement d'un système d'informatique ubiquitaire 31

5.1 Problématiques de simplification du développement logiciel 38

6.1 Vue générale de notre approche 44

7.1 Extrait de la spécification des types de données (DiaSpec) 50
7.2 Extrait de la spécification des interfaces (DiaSpec) 51
7.3 Style architectural de DiaSpec 51
7.4 Extrait de l'architecture du système de gestion des incendies (DiaSpec) 52
7.5 Une implémentation de l'opérateur de contexte `SmokeDetected` (DiaSpec/Java) 53
7.6 Extrait de la spécification des interfaces (Pantaχou) 56
7.7 Une implémentation du service `SmokeDetected` (Pantaχou) 58
7.8 Processus de compilation de Pantaχou 59
7.9 Extrait de la spécification des interfaces (Pantagruel) 61
7.10 Extrait de la spécification des énumérations et des objet communicants du système de gestion des incendies (Pantagruel) 62
7.11 Extrait des règles d'orchestration du système de gestion des incendies (Pantagruel) 63
7.12 Processus de compilation de Pantagruel 64

8.1 Extrait de la spécification des interfaces avec des déclarations de gestion d'erreurs 69
8.2 Extrait de la hiérarchie des exceptions *built-in* 69
8.3 Extrait de l'architecture du système de gestion des incendies avec des déclarations de gestion d'erreurs (niveau application) 71
8.4 Extrait de l'architecture de gestion d'erreurs du système de gestion des incendies (niveau système) 72
8.5 Extrait de la classe *proxy* `TemperatureSensorProxy` générée pour l'opérateur de contexte `AverageTemperature` 74
8.6 Exemples de continuations 75
8.7 Extrait d'une implémentation de l'opérateur de contexte `FireExtinctionFailure` 76
8.8 Signalisation des erreurs 77

8.9 Propagation de l'exception de type ApplicationLevelException 78

Chapitre 1

Introduction

La taille et la complexité des applications et des systèmes logiciels n'ont eu de cesse d'augmenter, avoisinant ces dernières années parfois plusieurs millions de lignes de code (*e.g.*, quarante millions pour Windows XP ou encore plus d'un milliard pour l'Airbus A380). Cette situation pose un certain nombre de problèmes dans le développement logiciel [CE00]. Par exemple, les techniques d'ingénierie logicielle actuelles permettent de facilement détecter et éliminer toutes sortes d'anomalies dans le cas de petits systèmes. Toutefois, lorsqu'il s'agit de systèmes complexes de grande taille, la tâche est beaucoup plus ardue car il devient difficile de comprendre et d'analyser la manière dont le contrôle et les données circulent entre les différentes parties du système. Ces anomalies peuvent alors avoir des conséquences imprévisibles et désastreuses sur le système, allant des bogues fréquents des ordinateurs au crash du réseau téléphonique interurbain américain, en passant par l'explosion de la fusée Ariane 5 lors du vol 501. L'un des défis scientifiques majeurs de notre temps réside donc dans l'amélioration de la construction et de la vérification de tels systèmes.

Pour relever ce défi, il est nécessaire d'élever le niveau d'abstraction des systèmes bien au-delà du code et de les décomposer en modules (*i.e.*, en sous-problèmes) [Par72], afin de gérer toute la complexité et de faciliter le raisonnement. L'architecture logicielle est aujourd'hui un moyen reconnu pour répondre à ces attentes. Elle vise à fournir des descriptions de haut niveau des systèmes, représentant non seulement leur structure logique, mais également beaucoup d'autres aspects fonctionnels et non fonctionnels (*e.g.*, comportement, sûreté). Cependant, la plupart du temps, les approches existantes d'architecture logicielle ne garantissent pas la cohérence entre les différents aspects qu'elles permettent de décrire et leurs implémentations. Or, il ne s'agit pas seulement d'élever le niveau d'abstraction, mais surtout de tirer profit des descriptions de haut niveau dans les processus de construction et de vérification des systèmes, afin de simplifier et d'améliorer le développement logiciel.

Traditionnellement, les approches d'architecture logicielle cherchent à rester indépendantes des langages d'implémentation, augmentant de ce fait le fossé entre les abstractions architecturales et celles de programmation. Il devient alors difficile d'apporter aux développeurs une aide effective pour faciliter la construction et la vérification de systèmes. En outre, plus les abstractions sont généralistes, plus il est difficile de réduire ce fossé et de prendre en compte toutes les spécificités et les besoins du domaine d'application pour lequel un système est développé. Au contraire, des abstractions spécifiques aux problèmes du domaine apparaissent plus appropriées, car elles renferment davantage d'informations pertinentes pour guider et supporter rigoureusement le processus de développement logiciel.

1.1 Thèse

La thèse présentée dans ce document propose une approche langage pour rendre plus simple et plus sûr le développement de systèmes logiciels. Elle repose sur l'utilisation de langages dédiés et sur un couplage fort entre des descriptions architecturales et un langage de programmation. Ce couplage permet de fournir aux développeurs un support de programmation et des garanties de sûreté, conformément aux descriptions architecturales. Ainsi, les développeurs peuvent être guidés sûrement tout au long du processus de développement. Nous illustrons et validons notre approche par l'étude du domaine du développement de systèmes d'informatique ubiquitaire.

L'approche que nous proposons introduit différents couplages entre des couches de spécification et d'architecture et des couches d'implémentation, dédiées au développement de systèmes d'informatique ubiquitaire. Les couches de spécification et d'architecture permettent de décrire à un haut niveau d'abstraction certains aspects de ces systèmes. Les couches d'implémentation sont ensuite personnalisées en fonction des informations définies dans les descriptions de haut niveau, facilitant la construction et la vérification de systèmes d'informatique ubiquitaire. De plus, selon le degré de spécificité des couches d'implémentation, le couplage permet d'ouvrir de nouvelles perspectives, à la fois en termes d'abstractions de programmation et de vérifications.

1.2 Contributions

La contribution de cette thèse est double. Dans un premier temps, nous montrons comment, en couplant fortement une couche de spécification et d'architecture et une couche d'implémentation dédiées à un domaine d'application, il est possible de faciliter la construction et la vérification de systèmes logiciels. Dans un second temps, nous proposons une mise en œuvre de cette approche dans le domaine de l'informatique ubiquitaire et nous évaluons ses avantages en termes de facilité de programmation et de sûreté.

Analyse de domaine. Nous présentons une analyse complète du domaine du développement de systèmes d'informatique ubiquitaire. Nous identifions les objets de base du domaine avec leurs opérations, relations et contraintes, et nous introduisons deux langages dédiés, Pantaχou et Pantagruel, qui permettent de développer des systèmes d'informatique ubiquitaire à des niveaux d'abstraction différents.

Approche langage. Nous décrivons différents couplages entre des couches de spécification et d'architecture et des couches d'implémentation, dédiées au développement de systèmes d'informatique ubiquitaire et évoluant à des niveaux d'abstraction différents. Nous montrons comment, grâce au couplage, la programmation est rigoureusement dirigée par les descriptions de haut niveau de la couche de spécification et d'architecture, guidant le développement logiciel et permettant aux développeurs de se concentrer sur la logique du programme.

Développement logiciel plus simple. Les descriptions de la couche de spécification et d'architecture permettent de personnaliser la couche d'implémentation. Nous montrons comment le processus de personnalisation permet d'offrir aux développeurs des constructions et

des abstractions de programmation de haut niveau et dédiées pour guider et supporter l'implémentation de différents aspects d'un système, à la fois fonctionnels (*e.g.*, la découverte d'objets communicants) et non fonctionnels (*e.g.*, la gestion des erreurs).

Développement logiciel plus sûr. Le couplage fort permet d'assurer automatiquement et systématiquement que l'implémentation du système est conforme aux descriptions de la couche de spécification et d'architecture. Nous montrons alors comment certaines propriétés deviennent apparentes dans la structure du système, facilitant l'analyse statique du code et la vérification de propriétés (*e.g.*, l'intégrité des communications).

1.3 Organisation du document

Ce document est organisé en deux parties. Nous présentons tout d'abord le contexte scientifique dans lequel il s'inscrit. Nous décrivons ensuite notre approche et ses avantages, en l'illustrant dans le domaine du développement de systèmes d'informatique ubiquitaire.

Contexte. La première partie porte sur l'étude du contexte scientifique dans lequel nous nous plaçons. Les chapitres 2 et 3 présentent les deux composantes de base de notre travail : l'architecture logicielle et les langages dédiés. Le chapitre 4 présente le domaine d'application choisi pour illustrer notre thèse : le développement de systèmes d'informatique ubiquitaire. Enfin, le chapitre 5 dresse le bilan de cette étude et résume les problématiques soulevées par la complexité et la taille croissantes des systèmes logiciels.

Approche proposée. La seconde partie vise à illustrer l'approche proposée dans le domaine du développement de systèmes d'informatique ubiquitaire. Le chapitre 6 présente notre approche fondée un couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation. Le chapitre 7 introduit différents couplages entre des couches de spécification et d'architecture et des couches d'implémentation, dédiées au développement de systèmes d'informatique ubiquitaire et évoluant à des niveaux d'abstraction différents. Ces couplages permettent de faciliter la construction et la vérification de systèmes d'informatique ubiquitaire. Le chapitre 8 montre les avantages de notre approche pour guider et supporter la gestion des erreurs dans les systèmes d'informatique ubiquitaire. Pour conclure, le chapitre 9 récapitule nos contributions et donne quelques directions futures pour nos travaux.

Première partie

Contexte

Chapitre 2

Architecture logicielle

Face à la taille et à la complexité croissantes des systèmes logiciels, les programmeurs ont naturellement été amenés à étager le développement logiciel [DK76]. Ainsi, avant de programmer un système complexe, il apparaît important de bien comprendre, à un plus haut niveau d'abstraction, ce que le système doit faire. L'architecture logicielle [TMD09] cherche à répondre à cette attente. Le concept d'architecture logicielle a été introduit pour la première fois à la conférence NATO en 1969 [BR70]. Depuis, l'architecture logicielle a suscité une attention croissante, à la fois de la part de la communauté scientifique et de celle des industriels du domaine du génie logiciel [KOS06]. En effet, des diagrammes informels et *ad hoc* en «boîtes et flèches» aux *frameworks* et outils permettant de raisonner et de générer du support de programmation à partir de descriptions architecturales, en passant par des techniques et des méthodologies de conception architecturale (*e.g.*, *Rational Unified Process* [Kru03]), l'architecture logicielle connaît aujourd'hui son âge d'or [SC06].

Dans ce chapitre, nous présentons une introduction à l'architecture logicielle, ainsi qu'un aperçu des principaux avantages de son utilisation dans le développement logiciel. Nous donnons ensuite un éventail des approches existantes pour décrire une architecture logicielle. Enfin, nous nous penchons sur les limites actuelles de ces approches pour tirer pleinement profit des décisions architecturales au niveau de l'implémentation des systèmes.

2.1 Introduction

Il existe de nombreuses définitions d'une architecture logicielle [Sof10]. Chacune d'elles met l'accent sur certains aspects particuliers de l'architecture logicielle. Dans ce document, l'architecture d'une application ou d'un système logiciel définit ce dernier en termes de *composants* et d'*interactions* entre ces composants [CBB$^+$10]. Les composants peuvent être vus comme des unités de calcul ou de stockage. Ils effectuent chacun une partie du calcul global du système et interagissent pour combiner leur comportement. Composants et interactions constituent donc les briques de base à partir desquelles une architecture est construite. Une description architecturale peut comporter aussi bien des aspects fonctionnels (*e.g.*, comportement) que des aspects non fonctionnels (*e.g.*, sécurité, fiabilité, performance). Elle expose également un ensemble de propriétés et de contraintes que le système doit respecter. En offrant une vision globale et haut niveau de la structure et de l'organisation d'un système, l'architecture logicielle joue un rôle clé comme point pivot entre les exigences d'un système et sa mise en œuvre, comme illustré à la figure 2.1. Elle fournit ainsi un plan de construction partiel, possiblement

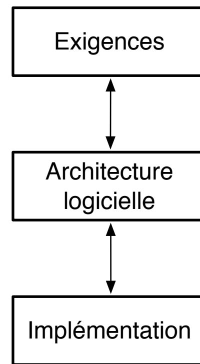

FIGURE 2.1 – Architecture logicielle comme point pivot

dans plusieurs vues [Kru95], dont les choix vont conditionner et guider le développement des systèmes logiciels.

Considérons l'exemple simple et familier d'un compilateur, illustré à la figure 2.2. L'architecture d'un compilateur [ALSU06] peut être décomposée en cinq composants [1] : un analyseur lexical, un analyseur syntaxique, un analyseur sémantique, un générateur de code et une table de symboles. Les quatre premiers sont responsables des phases du processus de compilation. Par exemple, l'analyseur syntaxique est chargé de transformer un flot d'unités lexicales en un arbre abstrait. Une fois le système décomposé en composants, ces derniers doivent être combinés afin de définir la structure du système dans son ensemble. Dans notre exemple, les interactions (représentées par des flèches) entre les différents composants sont simples. Les composants sont connectés dans une séquence linéaire dans laquelle le calcul d'un composant est utilisé par le composant suivant pour son propre calcul. Finalement, cette architecture implique certaines propriétés structurelles auxquelles les implémentations de compilateur doivent se conformer. Par exemple, elle indique que l'analyseur syntaxique n'interagit pas directement avec le générateur de code.

2.2 Motivations

L'architecture logicielle facilite la compréhension de systèmes complexes en les représentant à un haut niveau d'abstraction et en masquant certains détails technologiques. Ainsi, les descriptions architecturales ont rapidement et naturellement constitué un formidable vecteur de communication entre les différents intervenants d'un projet informatique. Aujourd'hui, la maîtrise des architectures logicielles offre de nombreux autres avantages dans le développement logiciel, notamment en termes de réutilisation, de vérification et d'évolution. En effet, l'architecture logicielle permet une meilleure séparation des préoccupations (*separation of concerns*), ainsi que de nouvelles opportunités d'analyse. Elle permet également de factoriser certaines expériences passées et éprouvées sous la forme d'un style architectural.

1. Les différentes phases d'optimisation ont été omises.

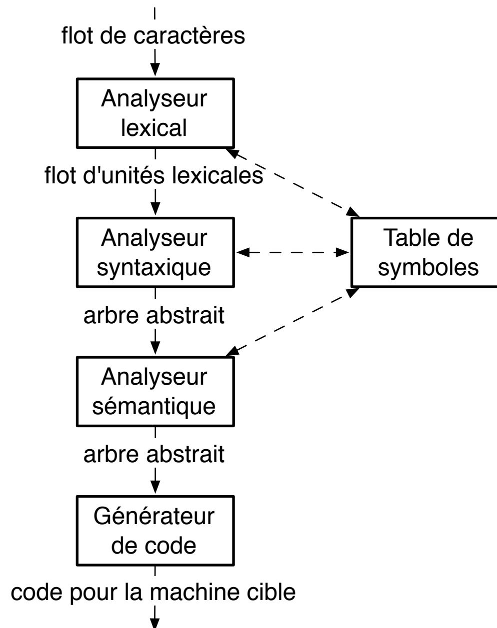

FIGURE 2.2 – Architecture d'un compilateur

2.2.1 Séparation des préoccupations

L'architecture logicielle s'appuie sur un principe fort du génie logiciel : la séparation des préoccupations [TOHS99]. En effet, afin de gérer la complexité d'un système, l'architecture logicielle sépare les calculs effectués par les composants des interactions dans lesquelles ils interviennent. Cette séparation augmente l'indépendance des composants du système, promouvant leur réutilisation (*e.g.*, dans des bibliothèques de composants) et facilitant leur implémentation. Elle permet également de changer facilement les implémentations des composants ainsi que les mécanismes d'interaction, ce qui favorise l'interopérabilité et l'évolution des systèmes. Par exemple, le générateur de code de la figure 2.2 peut être échangé indépendamment des autres composants afin de cibler une nouvelle machine spécifique. Finalement, en supprimant certains détails d'implémentation au niveau architectural, l'architecte peut se concentrer sur les analyses et les décisions qui sont les plus cruciales à la structure et l'organisation du système.

2.2.2 Analyses

L'architecture logicielle permet de raisonner sur un système à un haut niveau d'abstraction [AAG95, LKA+95]. Il existe un large spectre d'analyses architecturales permettant de vérifier aussi bien des propriétés structurelles que des propriétés comportementales. Ces analyses incluent la vérification de la conformité entre la spécification d'un composant du système et son implémentation ou encore différentes analyses de dépendance. Toutefois, du fait du fossé existant entre l'architecture d'un système et son implémentation, il peut être difficile de s'assurer que les propriétés et les contraintes vérifiées au niveau de l'architecture le sont

également au niveau de l'implémentation. Ainsi, garantir la cohérence entre l'architecture d'un système et son implémentation reste encore aujourd'hui un véritable défi. Cependant, dans le cadre d'un style architectural donné, il devient plus facile de faire le lien entre certains aspects architecturaux et leur implémentation.

2.2.3 Style architectural

Un aspect important des architectures logicielles est le concept de style architectural [AAG93]. Ce dernier caractérise une famille de systèmes qui utilise les même types de composants, d'interactions, de contraintes structurelles et sémantiques, et d'analyses [Gar95]. Parmi les styles architecturaux les plus répandus, nous pouvons citer le style *pipes and filters*, le style client/-serveur ou encore le style architecture en couches [SG96]. L'architecture de la figure 2.2 utilise par exemple le style *pipes and filters* pour chaîner les différentes phases d'un compilateur. Le but premier des styles architecturaux est de faciliter la conception architecturale ainsi que sa réutilisation. En effet, un style architectural capture et exploite certains motifs (*patterns*) et idiomes d'un domaine d'application particulier. De plus, en contraignant l'espace de conception, un style architectural permet des analyses spécifiques et puissantes qui vont au-delà de ce qu'il est possible de faire avec des éléments architecturaux généralistes.

2.3 Approches existantes

De nombreuses approches ont été proposées pour décrire et analyser, plus ou moins précisément, une architecture logicielle.

2.3.1 Diagrammes en «boîtes et flèches»

Les diagrammes en «boîtes et flèches» sont le moyen le plus simple pour décrire la structure d'un système. En effet, ils permettent de facilement identifier les éléments importants du système (*i.e.*, les boîtes) ainsi que leurs relations (*i.e.*, les flèches). Toutefois, ces diagrammes nécessitent d'être surchargés avec des informations informelles expliquant les calculs effectués par les boîtes et la nature des interactions représentées par les flèches. Par exemple, une flèche peut aussi bien représenter un lien de dépendance entre deux composants qu'un appel de procédure. Le manque de sémantique explicite limite sévèrement l'utilité de ces diagrammes dans le processus de développement logiciel.

2.3.2 Langages d'interconnexion de modules et de description d'interface

Les langages d'interconnexion de modules (*Module Interconnection Languages* ou MIL [2]) sont utilisés pour décrire formellement, au travers de services requis et fournis, les relations entre les différents modules d'un système [PDN86]. Des exemples de tels langages incluent MIL75 [DK76] ou encore INTERCOL [Tic79]. Les langages de description d'interface (*Interface Description Languages* ou IDL [2]) sont utilisés avec un but similaire dans la programmation orientée objet et composant [Sno89]. Par exemple, l'IDL de CORBA [3] [OMG95] a été développé

2. Dans ce document, nous utiliserons de manière indifférente le terme *langage d'interconnexion de modules* et *MIL*. De la même manière, nous utiliserons indifféremment le terme *langage de description d'interface* et *IDL*.

3. CORBA est l'acronyme anglais de *Common Object Request Broker Architecture*.

pour spécifier des interfaces d'objets distribués et leurs interactions. Un autre exemple d'IDL très répandu est WSDL [4] pour les services Web [CDK+02]. L'inconvénient principal des MIL et des IDL est qu'ils permettent seulement de capturer des relations d'implémentation (*e.g.*, de quel code un composant a besoin pour fonctionner) et non des relations d'interactions architecturales (*e.g.*, comment un composant compte interagir avec un autre), ce qui limite considérablement les analyses possibles.

2.3.3 Langages de description d'architecture

Les langages de description d'architecture (*Architecture Description Languages* ou ADL [5]) fournissent des notations formelles pour décomposer un système en *composants* et *connecteurs* (*i.e.*, des mécanismes d'interaction), et pour spécifier comment ces éléments sont combinés afin de former une *configuration* [MT00]. Les ADL couvrent de nombreux domaines d'application et la plupart d'entre eux se focalisent sur certains aspects spécifiques de l'architecture logicielle. Par exemple, Darwin [MK96] s'intéresse à la construction de systèmes distribués, MetaH [BEJV96] fournit un environnement complet pour concevoir des systèmes de contrôle avionique temps réel, et C2 [MORT96] cible le domaine des interfaces utilisateur. Rapide [LV95], quant à lui, se concentre sur la spécification et l'analyse par simulation de l'aspect dynamique des architectures logicielles. Wright [AG97] supporte la spécification et l'analyse comportementale des éléments architecturaux. En particulier, il permet de détecter la présence d'inter-blocages dans les systèmes concurrents. Unicon [SDK+95] est un ADL généraliste qui peut générer automatiquement le *glue code* (*i.e.*, le code assurant l'interopérabilité des composants) à partir de la spécification des connecteurs. Néanmoins, contrairement à Wright par exemple, Unicon ne supporte qu'un ensemble restreint de types de connecteur, non facilement extensible. Enfin, Aesop [GAO94] se concentre sur l'utilisation des styles architecturaux, SADL [MQR95] offre un mécanisme de raffinement architectural, et Acme [GMW97] vise à fournir un modèle architectural générique. En résumé, certains ADL (*e.g.*, Wright, Rapide, SADL) introduisent une notation s'appuyant sur des méthodes formelles, ce qui permet d'effectuer des analyses puissantes avec l'aide d'outils CASE (*Computer-Aided Software Engineering*) [Iiv96] par exemple. De manière complémentaire, d'autres ADL (*e.g.*, Darwin, C2, Unicon) sont accompagnés d'outils capables de générer certains artefacts logiciels à partir des descriptions architecturales.

La plupart des ADL sont en pratique peu utilisés, à l'exception de Koala [vOvdLKM00] et surtout de UML [6] [Fow03], bien que ce dernier ne soit pas considéré comme un véritable ADL par certains puristes.

2.3.4 Le cas particulier de UML

UML est un langage de modélisation généraliste pour décrire différents aspects et différentes parties d'un système, et notamment son architecture logicielle. Bien que la version 2.0 de UML [FGDTS06] marque une amélioration notable par rapport à son prédécesseur (*e.g.*, introduction du mécanisme de profil), elle souffre encore d'un manque de définition sémantique pour certaines de ses constructions [HR04] et ne supporte pas certains concepts architecturaux

4. WSDL est l'acronyme anglais de *Web Services Description Language.*

5. Dans ce document, nous utiliserons de manière indifférente le terme *langage de description d'architecture* et *ADL.*

6. UML est l'acronyme anglais de *Unified Modeling Language.*

de base importants, tels que par exemple la notion de connecteur. Ainsi, ces différents manques empêchent l'analyse précise des interactions architecturales, ainsi que la possibilité de faire un lien fort entre certaines décisions architecturales et le code qui les implémente. Le langage UML est également connu pour trop facilement mêler des concepts de conception avec des directives d'implémentation, minimisant les bénéfices attendus d'une architecture logicielle.

Toutes ces approches, des diagrammes en «boîtes et flèches» aux ADL, en passant par les MIL et les IDL, tentent, à leur manière, de supporter plus ou moins efficacement le processus de développement logiciel. Toutefois, certains problèmes et limites subsistent et représentent encore aujourd'hui un véritable frein à l'utilisation des architectures logicielles dans un cadre industriel.

2.4 Limitations actuelles

Bien que l'architecture logicielle ait récemment atteint son âge d'or, de nombreux problèmes ouverts demeurent [SC06, CS09]. En particulier, un problème majeur est le fossé qui existe entre le travail d'architecture et celui d'implémentation. Ce fossé rend difficile non seulement la communication entre l'architecte et le développeur d'un système, mais aussi la garantie que l'implémentation du système respecte bien toutes les décisions architecturales prises, à la fois fonctionnelles et non fonctionnelles.

2.4.1 Communications difficiles

L'architecte et le développeur d'un système sont généralement deux personnes différentes. Pourtant, les approches existantes ne permettent pas de les faire communiquer de manière efficace, chacun dans leurs termes. En effet, les langages de description d'architecture sont faiblement couplés avec les langages de programmation utilisés pour implémenter les systèmes logiciels. Ce faible couplage permet de favoriser la portabilité ainsi que de différer certains choix d'implémentation bas niveau. Toutefois, il ne permet pas de s'assurer que les développeurs d'un système ont bien compris tout ce que l'architecte avait à l'esprit, ni même de les aider ou de les guider dans ce but. À la place, les ADL attendent des développeurs qu'ils suivent les bonnes pratiques de programmation (*e.g.*, éviter d'utiliser des données partagées) et le plan de construction donné par l'architecture. Les développeurs sont alors amenés soit à deviner les intentions de l'architecte, essentiellement en refaisant le travail de conception, soit à consulter fréquemment l'architecte, faisant de ce dernier le goulot d'étranglement dans le processus de développement logiciel. Cette situation peut finalement conduire les développeurs à introduire involontairement des incohérences dans l'implémentation du système, rendant de surcroît l'architecture inutile.

2.4.2 Manque de cohérence

Le faible couplage entre les ADL et les langages d'implémentation rend très difficile de garantir la cohérence entre l'architecture d'un système et son implémentation. Un système se conforme à son architecture si cette dernière est une abstraction correcte du comportement du système à l'exécution. Autrement dit, les propriétés structurelles et sémantiques d'une description architecturale doivent aussi être vérifiées dans l'implémentation du système.

Sans cette cohérence, les bénéfices escomptés des architectures logicielles en termes d'analyse, d'implémentation, de compréhension et d'évolution des systèmes logiciels, présentés dans la section 2.2, sont sérieusement compromis. Garantir la cohérence entre l'architecture d'un système et son implémentation peut être fait soit par extraction, soit par construction.

Par extraction. Il s'agit d'analyser statiquement ou dynamiquement certaines parties du code du système afin d'extraire une architecture et déterminer si elle respecte bien les propriétés architecturales attendues. Alors que les descriptions architecturales utilisent des abstractions de haut niveau, les implémentations des systèmes correspondants sont le plus souvent développées avec des langages de programmation généralistes, tels que Java ou C++. Par conséquent, sans un couplage fort entre les abstractions architecturales et le code d'implémentation, les analyses d'extraction sur le code deviennent une tâche ardue.

Par construction. Il s'agit de raffiner l'architecture du système ou de générer du support de programmation à partir des descriptions architecturales. Toutefois, le support de programmation fourni par les ADL se limite généralement à de simples squelettes de code à trous ou encore à des *stubs* pour connecter les composants développés de manière indépendante. Ce faible support de programmation est insuffisant pour tracer et imposer certaines contraintes architecturales au niveau de l'implémentation, en particulier celles qui vont au-delà de la structure du système.

2.5 Évolution récente

Face aux limitations des solutions actuelles, quatre approches, Archface [UNT10], ArchJava [ACN02b], ACOEL [Sre02] et ComponentJ [SC00], ont récemment changé de stratégie en choisissant de coupler fortement des éléments architecturaux avec des langages d'implémentation. Par exemple, ACOEL et ArchJava intègrent directement un langage de description d'architecture dans un langage de programmation généraliste. Pour cela, ils étendent ce dernier avec de nouvelles constructions syntaxiques dédiées aux préoccupations architecturales. En particulier, ArchJava est une extension du langage Java qui unifie de manière transparente architecture logicielle et implémentation orientée objet. Archface, quant à lui, sépare les descriptions architecturales de l'implémentation. Il propose un nouveau mécanisme d'interface et utilise la programmation orientée aspect [EFB01] pour réduire le fossé entre l'architecture d'un système et son implémentation, et décrire les interactions entre les différents composants.

Ce fort couplage apporte des bénéfices considérables en termes d'analyse, d'implémentation et d'évolution des systèmes, comme le remarquent Alexandrescu *et al.* dans leur évaluation de ArchJava [AL03]. Par exemple, le système de type de ArchJava permet de garantir statiquement l'*intégrité des communications* [LV95] entre l'architecture d'un système et son implémentation [ACN02a], ce que les ADL généralistes, présentés dans la section 2.3, ne sont pas capables de faire. Cette propriété spécifie que les composants dans l'implémentation peuvent seulement communiquer avec les composants auxquels ils sont directement connectés dans l'architecture. L'intégrité des communications est une propriété fondamentale, liant architecture et implémentation, sur laquelle d'autres propriétés architecturales se reposent. En effet, en vérifiant quels composants communiquent, d'autres propriétés peuvent se concentrer sur comment ces composants communiquent (*e.g.*, un système de type décrivant quelles sortes de données sont échangées). Sans intégrité des communications, la vérification de propriétés au

niveau du code est très difficile car certains chemins de communication peuvent être omis ou non valides, et certains flots de données et de contrôle ignorés. L'intégrité des communications constitue donc un premier pas indispensable pour garantir la cohérence complète entre l'architecture d'un système et son implémentation.

Ces quatre approches offrent toutes des solutions généralistes, aussi bien au niveau du langage de description d'architecture que du langage d'implémentation. Cependant, spécialiser ces approches à un domaine d'application particulier pourrait permettre d'aller au-delà de la propriété d'intégrité des communications, et ainsi de garantir et supporter d'autres aspects architecturaux (*e.g.*, la sécurité, la sûreté) au niveau de l'implémentation des systèmes. En effet, contrairement à une approche généraliste, une approche spécifique peut prendre en compte toutes les spécificités d'un domaine d'application afin de fournir des analyses et du support d'implémentation qui sont dédiés et adaptés à ses besoins. L'approche proposée dans cette thèse cible des domaines d'applications particuliers, et plus précisément le domaine de l'informatique ubiquitaire.

2.6 Bilan

L'architecture logicielle permet de représenter et de raisonner sur différents aspects fonctionnels et non fonctionnels d'un système à un haut niveau d'abstraction. Aujourd'hui, il ne s'agit plus de considérer ces descriptions architecturales d'un simple point de vue «contemplatif» (*i.e.*, pour documenter, spécifier ou encore communiquer), mais au contraire de tirer profit des informations qu'elles fournissent pour supporter et guider le plus possible le processus de développement logiciel. En particulier, l'architecture logicielle permet d'offrir une meilleure séparation des préoccupations, de donner lieu à de puissantes analyses et de favoriser la réutilisation au travers de styles architecturaux. Il existe différents travaux pour décrire l'architecture d'un système, parmi lesquels les langages de description d'architecture sont les plus aboutis. Néanmoins, en cherchant à rester indépendants des langages d'implémentation, les ADL traditionnels rendent difficile la communication entre l'architecte et le développeur, et ne permettent pas de garantir la cohérence entre les décisions architecturales (*e.g.*, en ce qui concerne la sécurité ou la sûreté des systèmes) et leurs implémentations. Sans cette cohérence, tous les avantages et l'impact de l'utilisation des architectures logicielles dans le développement logiciel en deviennent amoindris. Récemment, plusieurs approches ont adopté une stratégie différente, en intégrant directement un langage de description d'architecture dans un langage de programmation généraliste. Cette intégration a contribué à améliorer la construction et la vérification de systèmes logiciels. Toutefois, la généricité de ces approches limite le support de programmation et les analyses qu'il serait possible de fournir dans le cadre d'une approche de développement spécifique à un domaine.

Chapitre 3

Langages dédiés

Les langages généralistes fournissent des solutions génériques pour traiter une large classe de problèmes. Toutefois, de telles solutions ne sont pas toujours optimales pour les problèmes considérés et montrent rapidement leurs limites face à l'ampleur et à la complexité des développements logiciels actuels [CE00]. Une approche différente consiste à utiliser des langages dédiés à un domaine de problèmes, offrant des solutions sur mesure à leurs besoins spécifiques [KT08]. Une telle approche est bien souvent meilleure et plus efficace car elle se concentre sur un plus petit ensemble de problèmes et permet de spécialiser le processus de développement logiciel.

Dans ce chapitre, nous présentons une introduction aux langages dédiés, ainsi qu'un aperçu des principaux avantages de leur utilisation par rapport aux langages généralistes. Nous étudions ensuite les différentes étapes du processus de développement d'un langage dédié.

3.1 Introduction

À l'instar de l'architecture logicielle, il existe de nombreuses définitions d'un langage dédié [Fow10, MHS05, Con04, vDKV00, vDK98]. Chacune d'elles dépend à la fois du domaine d'application et de la communauté d'utilisateurs qu'elle vise. La littérature présente souvent un langage dédié (*Domain-Specific Language* ou DSL [1]) comme un «petit langage», textuel ou visuel, plutôt déclaratif qu'impératif, et non Turing-complet. Toutefois, comme le montrent les langages dédiés listés dans le paragraphe suivant, ces critères ne permettent en aucun cas de définir si un langage est un DSL ou non. Dans ce document, un langage dédié est un langage de programmation restreint à un problème ou à un domaine particulier. Il fournit des notations appropriées et des abstractions spécifiques au domaine. Celles-ci possèdent une sémantique explicite, pouvant être exprimée sous forme d'expressions mathématiques ou bien donnée par un interprète ou un compilateur (*i.e.*, par une implémentation). De plus, un langage dédié propose une expressivité focalisée sur les concepts du domaine, cachant aux utilisateurs une grande partie des détails d'implémentation. Ainsi, dans un programme écrit avec un langage dédié, seule la logique du domaine est exposée et manipulée.

Les exemples de langages dédiés sont très nombreux. Certains d'entre eux sont largement répandus et quotidiennement utilisés depuis plusieurs décennies. L'exemple le plus connu est certainement le langage de formules d'Excel pour les feuilles de calcul qui compte aujourd'hui environ un demi-milliard d'utilisateurs. Les langages dédiés sont utilisés dans de nombreux domaines très variés tels que les bases de données (*e.g.*, SQL, Datalog), la chimie moléculaire (*e.g.*,

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

CML [MR97]), la mise en forme de documents (*e.g.*, HTML, LaTeX), les pilotes de périphériques (*e.g.*, Devil [MRC+00], GAL [TMC99]), les produits financiers (*e.g.*, RISLA [AvDR95]), la robotique (*e.g.*, Altaira [PJ97], LEGOsheets [GIL+95]) ou encore la téléphonie sur IP (*e.g.*, CPL [LS00], SPL [BCL+06], VisuCom [LMC07]). Les ADL, présentés au chapitre précédent, sont également des DSL, dédiés à la description d'architectures logicielles.

Les langages dédiés sont aujourd'hui un moyen reconnu pour augmenter la productivité et la qualité des développements logiciels [DSM10].

3.2 Motivations

En se focalisant sur un domaine particulier, les langages dédiés proposent des solutions de programmation sur mesure afin de répondre efficacement aux besoins spécifiques du domaine. Ainsi, ils offrent de nombreux avantages par rapport aux langages généralistes, notamment en termes de productivité, de vérification et de facilité de programmation [Ben86]. En effet, de par leur conception, les langages dédiés simplifient la programmation du domaine de problèmes, contribuent à augmenter la sûreté des applications et permettent de réutiliser de manière systématique une grande partie de l'expertise du domaine.

3.2.1 Programmation du domaine de problèmes

Le niveau d'abstraction offert par un langage dédié correspond directement à celui du domaine de problèmes, réduisant de manière significative la distance conceptuelle entre l'espace de problèmes et l'espace de solutions. Cette situation facilite grandement le développement logiciel et améliore la productivité, permettant de concevoir des systèmes plus complexes. Elle permet de développer des solutions plus lisibles et plus concises qu'avec un langage de programmation généraliste. Un langage dédié expose aux développeurs ce qui doit être calculé (*i.e.*, la logique du domaine) plutôt que comment le calculer (*i.e.*, les détails d'implémentation), ce qui leur permet de réfléchir, concevoir et implémenter des solutions dans les termes du domaine. Un langage dédié aspire alors à devenir un outil accessible aux experts du domaine, sans qu'ils aient nécessairement besoin de connaissances approfondies en programmation.

3.2.2 Sûreté améliorée

Les langages dédiés facilitent la vérification de propriétés du domaine en les rendant explicites dans la structure de l'application. Garantir ces propriétés avec un langage généraliste peut être difficile, voire impossible. Un langage dédié offre par conséquent des garanties de sûreté beaucoup plus fortes, visant à détecter les erreurs le plus tôt possible dans le processus de développement. Ces garanties de sûreté peuvent être obtenues soit par construction, soit par analyse.

La première approche consiste à restreindre ou à enrichir la sémantique du langage afin de rendre décidables certaines propriétés critiques du domaine. L'exemple typique est la suppression des instructions de boucle non contrôlée (*e.g.*, l'instruction `while` en Java) pour garantir la terminaison des programmes. Puisque seules des opérations valides vis-à-vis de la sémantique du domaine sont générées, il est possible de garantir par construction (*i.e.*, avant l'exécution) qu'une application écrite dans un langage dédié respecte certaines propriétés du domaine. Par exemple, la conception du langage VisuCom empêche de construire des services de téléphonie

incomplets ou incorrects qui pourraient corrompre, voire interrompre, la plate-forme toute entière [Lat07].

La seconde approche repose sur l'utilisation d'analyses de programme traditionnelles. De par la nature haut niveau des langages dédiés, il est possible de facilement réutiliser des outils de vérification, comme par exemple des vérificateurs de modèles, pour effectuer ces analyses. Ces dernières ont l'avantage de pouvoir détecter des anomalies plus subtiles dans les applications vérifiées. Par exemple, le langage Devil vérifie statiquement que tous les ports, les variables privées et les registres déclarés dans une spécification d'interfaces de programmation de périphériques sont utilisés au moins une fois [Ré01]. Par ailleurs, les langages dédiés permettent d'introduire des optimisations spécifiques au domaine, notamment au niveau de la compilation, qui se révèlent souvent plus efficaces que celles offertes par des compilateurs traditionnels. Par exemple, le compilateur SQL évalue les coûts des différents plans d'exécution possibles d'une requête et choisit le moyen le plus efficace de la traiter. Cette optimisation est rendue possible grâce à l'utilisation de la connaissance du domaine et de ses concepts clés, et parce que les moyens d'expression sont restreints.

3.2.3 Réutilisation systématique

Les langages de programmation généralistes permettent d'abstraire des opérations communes dans des bibliothèques ou dans des *frameworks*. Toutefois, la réutilisation de ces bibliothèques et de ces *frameworks* est laissée à la charge et à la discrétion du développeur. Au contraire, les langages dédiés permettent de systématiser la réutilisation. En effet, un langage dédié caractérise une collection d'abstractions spécifiques à un domaine, servant à résoudre les problèmes de ce domaine. Chacune de ces abstractions est associée à un motif de code qui sera automatiquement généré. Ainsi, programmer avec un langage dédié permet de réutiliser de manière systématique certains fragments de code. Mais au-delà de la réutilisation du code lui-même, c'est toute la connaissance d'un domaine, capturée dans la conception du langage dédié, qui est réutilisée. Par exemple, le langage SPL [BCL+06], dédié au développement de services de téléphonie, illustre parfaitement cette systématisation de la réutilisation. En effet, la syntaxe d'un service SPL reflète la structure des sessions d'un environnement SIP [2] [RSC+02], associant un gestionnaire d'événement à chaque type de message SIP.

3.3 Processus de développement

Le développement d'un langage dédié n'est pas une tâche facile car il requiert à la fois beaucoup de temps et d'efforts. Différents outils [GSCK04] et méthodologies [CM98] ont donc été proposés pour supporter la création de langages dédiés. La figure 3.1, adaptée de la thèse de Fabien Latry [Lat07], résume le processus de développement d'un langage dédié. Il se décompose principalement en deux phases : la conception et l'implémentation.

3.3.1 Conception

La phase de conception d'un langage dédié occupe une place critique dans le processus de développement. En effet, elle est responsable de déterminer l'expressivité, l'accessibilité, le

2. SIP est l'acronyme anglais de *Session Initiation Protocol*.

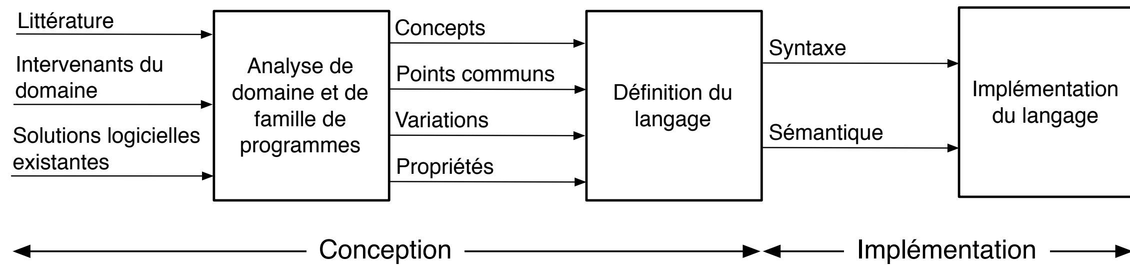

FIGURE 3.1 – Processus de développement d'un langage dédié

niveau d'abstraction, ainsi que les propriétés du langage dédié. Pour cela, il convient d'analyser minutieusement le domaine d'application cible afin de pouvoir définir la syntaxe et la sémantique du langage dédié.

Analyses de domaine et de famille de programmes

La première étape de la conception d'un langage dédié consiste à étudier et à délimiter précisément son domaine d'application. En particulier, il s'agit de définir les besoins et les objectifs du langage dédié. Cette étape est très difficile car un domaine trop étroit peut limiter l'utilisation du langage dédié, alors qu'un domaine trop large peut compliquer la vérification de propriétés. Différentes méthodes d'analyses peuvent être utilisées pendant cette étape : les analyses de domaine et de famille de programmes. Le concept d'analyse de domaine a été introduit pour la première fois par Neighbors. Il le définit comme suit :

> «*Une analyse du domaine tente d'identifier les objets, les opérations et les relations entre ce que les experts du domaine considèrent comme important pour ce domaine.*» [Nei80].

Depuis, cette définition a été revisitée de nombreuses fois, notamment par McCain [McC85], Arango [Ara89] ou encore Prieto-Díaz [PD90]. Le but d'une telle analyse est donc d'identifier et de structurer un maximum de points communs entre les différents éléments d'un domaine afin de les réutiliser de manière systématique. Toutefois, l'analyse de domaine n'est généralement pas suffisante pour caractériser entièrement le domaine d'application d'un langage dédié, car elle ne s'intéresse qu'aux points communs qui existent à l'intérieur d'un domaine, sans prendre en compte les variations. L'analyse de famille de programmes, quant à elle, sépare les points communs des points de variations. Parnas définit une famille de programmes comme suit :

> «*Nous considérons qu'un ensemble de programmes constitue une famille, lorsqu'il est utile d'étudier les programmes de cet ensemble, en étudiant d'abord ses propriétés communes puis en déterminant les particularités de chaque membre de la famille.*» [Par76].

Une famille de programmes représente donc un domaine tel que le définit Neighbors. Toutefois, contrairement à un domaine quelconque, une famille de programmes décrit un ensemble restreint dans lequel il est possible d'étudier les particularités de chaque membre. De plus, comme le montre Consel [Con04], la notion de famille de programmes permet de structurer le

développement d'un langage dédié, ainsi que de l'évaluer en termes de performance ou encore de robustesse.

Les analyses de domaine et de famille de programmes utilisent différentes sources d'information. Ces dernières incluent la littérature du domaine (*e.g.*, manuels, rapports techniques), les différents intervenants du domaine (*e.g.*, experts, clients) ou encore les solutions logicielles existantes du domaine (*e.g.*, prototypes, standards). À partir de ces sources d'information, les analyses de domaine et de famille de programmes permettent d'identifier non seulement les caractéristiques communes et variables des systèmes logiciels du domaine, mais aussi la terminologie et les concepts clés du domaine. Elles permettent également de formuler les contraintes et les propriétés critiques du domaine que le langage dédié doit garantir.

La plupart du temps, les analyses de domaine et de famille de programmes sont effectuées de manière informelle et *ad hoc.* Afin d'aider le concepteur du langage dédié, plusieurs méthodologies et outils ont été proposés. Parmi ceux-ci, nous pouvons citer DARE (*Domain Analysis and Reuse Environment*) [FPDF98], FAST (*Family-Oriented Abstractions, Specification, and Translation*) [WL99], FDL (*Feature Description Language*) [vDK02], FODA (*Feature-Oriented Domain Analysis*) [KCH+90] ou encore ODE (*Ontology-based Domain Analysis*) [FGD02].

Les informations recueillies par les analyses de domaine et de famille de programmes sont ensuite utilisées pour la définition du langage dédié.

Définition du langage

Une fois le domaine d'application délimité, il s'agit de définir les constructions syntaxiques du langage dédié ainsi que leur sémantique (*i.e.*, leur signification).

Syntaxe. Le choix de la syntaxe d'un langage dédié n'est pas une tâche anodine. En effet, elle doit non seulement être lisible et facilement compréhensible par les utilisateurs du langage dédié, mais également permettre de rendre décidables certaines propriétés du domaine et d'automatiser certaines optimisations. Pour cela, le concepteur du langage dédié utilise les informations provenant de l'analyse de domaine et de famille de programmes. La terminologie et les concepts clés du domaine servent à nommer les constructions et les abstractions du langage dédié (*e.g.*, les opérations, les types de données prédéfinis) introduites pour chaque point commun. Toutefois, la forme finale de ces dernières dépend des points de variations ainsi que des contraintes et des besoins en vérification du domaine. Pour représenter la syntaxe d'un langage, il existe plusieurs notations, telles que BNF (*Backus-Naur Form*) [Knu64] ou encore SDF (*Syntax Definition Formalism*) [HHKR89]. Le principe est de définir la grammaire du langage dédié sous la forme de règles de dérivation exprimant les variations du domaine et définissant la syntaxe concrète. L'analyse syntaxique transforme alors un programme en un arbre de syntaxe abstraite, conformément aux règles de la grammaire.

Une fois les constructions syntaxiques du langage dédié choisies, il s'agit de formellement définir leur sémantique. En effet, l'analyse syntaxique ne suffit pas pour déterminer si un programme, bien que syntaxiquement correcte, est ou non valide, c'est-à-dire s'il a du sens.

Sémantique. La sémantique formelle d'un langage de programmation permet de définir un modèle mathématique du sens de tout programme écrit dans ce langage [Win93]. Elle est généralement divisée en deux parties : une sémantique statique et une sémantique dynamique. La sémantique statique se concentre sur la signification d'un programme avant son exécution (*e.g.*, vérification de type), alors que la sémantique dynamique s'intéresse au comportement des

programmes à l'exécution. Il existe principalement trois styles de sémantiques formelles : la sémantique opérationnelle, la sémantique dénotationnelle et la sémantique axiomatique. Chaque style possède ses propres spécificités et son utilité. La sémantique opérationnelle [Plo04] décrit sous la forme de règles les effets de l'exécution de chaque construction syntaxique. Elle est adaptée à la vérification de propriétés sémantiques, ainsi qu'à la construction de compilateurs. La sémantique dénotationnelle [Sch86], quant à elle, décrit les programmes de manière plus abstraite et compositionnelle à l'aide de formalismes mathématiques. Enfin, la sémantique axiomatique [Hoa69] est adaptée à la preuve de programmes. Une sémantique formelle est donc à la fois essentielle pour la vérification de propriétés et l'implémentation du langage.

3.3.2 Implémentation

La dernière phase du processus de développement d'un langage dédié correspond à son implémentation. Il existe différentes techniques d'implémentation d'un DSL [MHS05, Spi01]. Chacune d'elles a ses avantages et ses inconvénients. Le choix d'une technique dépend en grande partie des contraintes et des besoins spécifiques du domaine d'application cible (*e.g.*, performance, portabilité). Afin de donner un aperçu des différentes techniques d'implémentation d'un langage dédié, nous utilisons la notion de *langage hôte*. Il s'agit du langage de programmation utilisé pour implémenter le langage dédié. Dans certains cas, le langage hôte ne sera pas directement visible par l'utilisateur du langage dédié. De tels langages dédiés sont dits *autonomes*. Dans d'autres cas, l'application sera un mélange entre le langage dédié et le langage hôte. De tels langages dédiés sont dits *enchâssés*.

Langages dédiés autonomes

Un langage dédié autonome est un langage complètement nouveau, développé à partir de zéro. L'avantage d'une telle implémentation est la maîtrise totale de toutes les constructions syntaxiques du langage dédié, permettant d'adapter entièrement son implémentation à son domaine d'application. Cependant, le problème reste le coût de construction élevé. En effet, dans le cas d'un langage dédié autonome, il n'est pas possible de profiter des constructions de base du langage hôte, ce qui nécessite d'écrire du code explicite dans l'implémentation du DSL afin de reconnaître par exemple les constructions arithmétiques ou encore les boucles. Un autre inconvénient est le manque de réutilisation pour d'autres implémentations de DSL. Un langage dédié autonome peut être implémenté soit par un interprète (*e.g.*, HTML), soit par un compilateur (*e.g.*, LaTeX).

Interprète. Un interprète traite directement chaque construction du langage dédié pour produire les résultats attendus. L'interprétation est appropriée pour les langages dédiés qui présentent un caractère dynamique, car elle permet un contrôle plus important sur l'environnement d'exécution. Une approche classique pour écrire un interprète est d'utiliser des outils standards de génération automatique d'analyseurs lexicaux (*e.g.*, Lex [LS90]) et syntaxiques (*e.g.*, Yacc [JS90]). Le reste de l'interprète est généralement écrit à la main dans un langage de programmation généraliste (*e.g.*, OCaml). Toutefois, la sémantique dynamique du langage dédié peut être utilisée pour guider l'écriture de l'interprète. De plus, plusieurs approches ont été proposées pour simplifier et améliorer le développement d'interprètes de langages dédiés en tirant profit des informations spécifiques au domaine. Par exemple, Scott Thibault [Thi98] propose d'écrire des interprètes structurés et génériques, et d'utiliser l'évaluation partielle [CD93]

comme technique d'optimisation pour supprimer le coût de l'interprétation et de la généricité. Sheard *et al.* [SBP99] proposent, quant à eux, d'utiliser les monades [Wad90] et une sémantique multi-niveau (*multi-stage programming*) [Tah99] pour écrire des interprètes modulaires.

Compilateur. Le compilateur d'un langage dédié génère du code en langage hôte, laissant certains aspects bas niveau de la compilation au compilateur du langage hôte. Le code généré reprend des motifs de code identifiés pendant l'analyse de domaine et de famille de programmes, et peut cibler certaines couches d'implémentation, telles qu'un intergiciel ou un cadre de programmation spécifique à un domaine. La compilation est appropriée pour réaliser des analyses statiques complètes des applications afin de garantir certaines propriétés du domaine avant l'exécution. Comme dans le cas des interprètes, il est possible d'utiliser des outils standards de construction de compilateurs. Toutefois, là encore, la nature des langages dédiés permet de réutiliser des outils de génération de programmes haut niveau, simplifiant le développement de compilateurs de DSL. Par exemple, Stratego/XT [BKVV08] est un environnement de développement pour créer des systèmes de transformation autonomes. Il est destiné à l'analyse, la manipulation et la génération de programmes, et supporte un large éventail de transformations. Le langage Stratego [Kal06] permet d'écrire des règles de transformation sur des arbres de syntaxe abstraite, alors que XT offre une collection d'outils et de langages déclaratifs (*e.g.*, ASF+SDF [vdBHKO02], ATerms [vdBdJKO00]) pour réaliser d'autres aspects des systèmes de transformation, tels que l'analyse syntaxique (*parsing*) ou encore la mise en forme de code (*pretty-printing*). Hamey et Goldrei [HG08] montrent les avantages de l'utilisation de Stratego/XT pour implémenter un langage dédié par rapport aux outils traditionnels de construction de compilateurs. Latry *et al.* [CLRC05] proposent, quant à eux, de structurer le développement d'un compilateur de DSL en le décomposant en facettes représentant différentes dimensions de compilation, et d'utiliser des techniques de génération de programmes [CE00], telles que la programmation orientée aspect [EFB01], les annotations [CEI+07] ou encore la spécialisation de programmes [CD93], pour traiter de manière modulaire les informations spécifiques au domaine. Fabien Latry [Lat07] propose également une architecture en couches des langages afin de simplifier le processus de compilation des langages dédiés autonomes.

Langages dédiés enchâssés

Le concept de langage enchâssé a été introduit par Paul Hudak [Hud96]. Il s'agit d'une technique d'implémentation d'un langage dédié reposant sur l'utilisation systématique d'extensions syntaxiques du langage hôte. L'avantage d'une telle implémentation est que le langage dédié bénéficie de toutes les caractéristiques du langages hôte, c'est-à-dire ses constructions (*e.g.*, arithmétique, boucles) ou encore son infrastructure (*e.g.*, environnement de développement et de débogage, compilateur), tout en l'augmentant de constructions syntaxiques spécifiques au domaine. Cependant, le problème reste l'inadéquation entre l'expressivité du langage hôte et les restrictions sémantiques du langage dédié. En effet, le langage dédié enchâssé hérite de la puissance d'expression des mécanismes du langage hôte, ce qui peut considérablement limiter la vérification de propriétés du domaine. De plus, l'environnement d'exécution du langage hôte peut ne pas être optimisé pour les besoins spécifiques du DSL. Un langage dédié enchâssé peut être implémenté soit par un préprocesseur, soit par une extension du compilateur ou de l'interprète du langage hôte.

Préprocesseur. L'application passe d'abord par un préprocesseur qui traduit les constructions du langage dédié en instructions du langage hôte, puis le résultat est compilé (ou interprété) par le compilateur (ou l'interprète) du langage hôte. Ce type d'implémentation est simple et permet au concepteur du préprocesseur d'ignorer les détails d'implémentation du langage hôte afin de se concentrer sur l'expression des constructions du langage dédié sous la forme d'instructions du langage hôte. Toutefois, l'analyse statique est limitée à celle réalisée par le processeur du langage hôte. De plus, les messages d'erreurs retournés à l'utilisateur du langage dédié font référence à du code généré en langage hôte par le préprocesseur, et non aux constructions et abstractions spécifiques du DSL, ce qui complexifie considérablement leur gestion. Parmi les préprocesseurs les plus utilisés, nous pouvons citer le préprocesseur du langage C (*cpp*) qui permet l'inclusion de fichiers, la définition et l'expansion de macros, et la compilation conditionnelle. Le langage Lisp propose, quant à lui, un système de macros syntaxiques sophistiqué, permettant de manipuler le langage dans son ensemble afin de l'étendre selon les besoins spécifiques.

Extension de compilateurs et interprètes. Il s'agit d'étendre le compilateur ou l'interprète du langage hôte avec de nouvelles constructions spécifiques au domaine. Dans ce cas, le préprocesseur se retrouve inclus au sein même du compilateur ou de l'interprète. Il devient alors possible de réaliser des optimisations et des vérifications spécifiques au domaine plus abouties. Un exemple d'extension est l'interprète Tcl [Ous94] qui a été étendu pour de nombreux domaines d'application. Toutefois, étendre un compilateur est considérablement plus difficile et requiert une certaine expertise dans l'utilisation de compilateurs de compilateur, tels que Lex et Yacc [LS90, JS90], JavaCC (*Java Compiler Compiler*) [AP03] ou encore ANTLR (*ANother Tool for Language Recognition*) [Par07]. Récemment, de nouvelles approches ont émergé pour simplifier l'extension de compilateurs et d'interprètes. En particulier, nous pouvons citer JastAdd [EH07] et Silver [VWBGK10] qui permettent d'augmenter facilement la grammaire de Java. Par exemple, Van Vyk et Schwerdfeger [VWS07] montrent comment utiliser Silver pour enchâsser dans Java un langage de requêtes SQL et effectuer des vérifications de type.

3.4 Bilan

Les langages dédiés sont des langages de spécification ou de programmation restreints à un problème ou à un domaine particulier. Ils sont aujourd'hui une alternative intéressante et sérieuse aux approches traditionnelles de développement logiciel. En effet, ils offrent un haut niveau d'abstraction sur le domaine considéré, facilitant la construction et la vérification de systèmes logiciels. De par sa conception, un DSL rend la programmation plus simple et concise, permettant aux développeurs de penser les solutions dans les termes du domaine. De plus, il factorise toute la connaissance du domaine et en systématise la réutilisation, et permet également la vérification statique de propriétés spécifiques au domaine, améliorant la sûreté des applications. Néanmoins, tous ces avantages sont l'aboutissement d'un processus de développement difficile et coûteux. Différentes analyses et méthodologies de conception, ainsi que différentes techniques d'implémentation, peuvent alors être utilisées pour faciliter le développement d'un langage dédié. Nous nous appuyons sur celles-ci pour guider la conception des différents langages dédiés au développement de systèmes d'informatique ubiquitaire introduits dans la mise en œuvre de notre approche.

Chapitre 4

Étude de cas : l'informatique ubiquitaire

Le développement de systèmes d'informatique ubiquitaire illustre parfaitement les problématiques présentées dans ce document. En effet, ces systèmes sont au carrefour de plusieurs domaines technologiques (*e.g.*, multimédia, réseau, systèmes distribués et embarqués) et préoccupations sociales (*e.g.*, confidentialité, fiabilité, sécurité), ce qui les rend particulièrement difficiles à construire et à vérifier [DG02, Sat01, Abo99]. Les solutions de programmation actuelles offrent très peu d'abstractions et un support souvent générique et limité pour traiter ces différents aspects, faisant de la programmation de systèmes d'informatique ubiquitaire un processus compliqué et sujet à erreurs.

Dans ce chapitre, nous présentons le domaine de l'informatique ubiquitaire, puis nous caractérisons les systèmes d'informatique ubiquitaire pour en comprendre le fonctionnement. Nous montrons ensuite les enjeux relatifs au développement de tels systèmes. Enfin, nous donnons un état de l'art des solutions existantes.

4.1 Introduction

Le concept d'informatique ubiquitaire [1] a été introduit par Marc Weiser [Wei91] pour désigner sa vision futuriste de l'informatique du XXIe siècle. Il imaginait un monde abondamment peuplé d'objets informatiques et numériques qui seraient reliés en réseaux à très grande échelle et interagiraient de manière autonome et transparente afin d'accomplir diverses tâches de la vie quotidienne. L'idée novatrice de cette vision réside dans le fait de mettre les nouvelles technologies (*e.g.*, les technologies de l'information et de la communication) au service des utilisateurs, et non l'inverse. En particulier, il s'agit de permettre aux utilisateurs d'accéder aux différents services offerts par ces objets communicants, le plus naturellement possible, n'importe où, à tout instant et à partir de divers dispositifs. De plus, dans cette vision, les communications débordent du cadre classique homme vers homme ou homme vers machine pour inclure des communications directes entre machines.

La vision de Marc Weiser est devenue aujourd'hui possible grâce à la combinaison de trois facteurs principaux : la miniaturisation et la puissance des composants électroniques, la chute des coûts de production et l'omniprésence des technologies réseaux sans fil [WP05]. En effet, ces

1. L'informatique ubiquitaire est également appelée informatique diffuse ou encore intelligence ambiante.

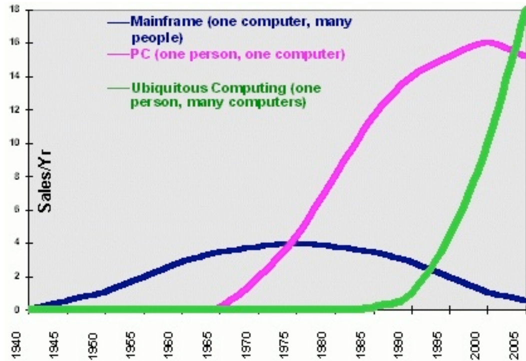

FIGURE 4.1 – Évolution de l'utilisation des ordinateurs

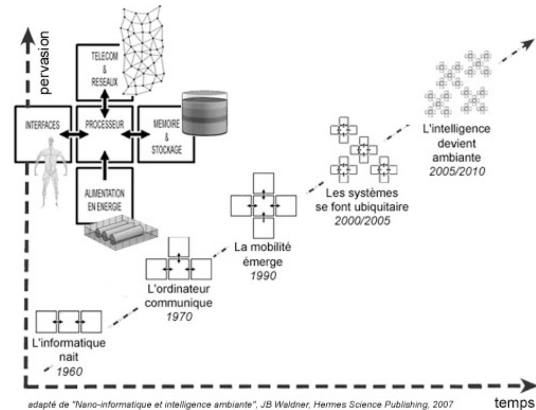

FIGURE 4.2 – De l'informatique à l'intelligence ambiante

dernières années ont vu la miniaturisation des dispositifs informatiques et électroniques dotés de capacités de capture, de traitement et de communication, pendant que leur puissance de calcul continuait d'augmenter. Beaucoup de progrès ont également été réalisés afin de réduire leur consommation d'énergie. Cette situation, associée à la chute des coûts de production, a favorisé la démocratisation de ces technologies et leur intégration dans de nombreux objets du quotidien. Comme le remarque Gérard Berry [Ber07], «*les téléphones, appareils photos et caméras vidéo, les lecteurs et instruments de musique, les contrôleurs enfouis dans les avions, les voitures ou encore l'électroménager deviennent des ordinateurs habillés autrement*». En outre, le développement des technologies réseaux sans fil (*e.g.*, Bluetooth, Wi-Fi, 3G+) a permis à ces nouveaux types d'objets communicants d'interagir spontanément avec l'utilisateur ou avec d'autres objets, aussi bien localement qu'à distance, et de favoriser leur mobilité. Les figures 4.1 et 4.2 illustrent cette évolution, en montrant respectivement comment la diminution de la taille et des coûts des calculateurs a permis de passer d'un ordinateur partagé par tous à de nombreux ordinateurs simultanément utilisables par une même personne, et l'introduction progressive de l'ordinateur dans les activités humaines.

Un ensemble d'objets communicants déployés dans une infrastructure réseau et électrique et orchestrés par diverses applications forme un système d'informatique ubiquitaire.

4.2 Systèmes d'informatique ubiquitaire

Un système d'informatique ubiquitaire permet d'automatiser certaines tâches quotidiennes grâce aux différents objets communicants disponibles [SM03]. La figure 4.3 représente schématiquement le fonctionnement d'un tel système. Le système collecte tout d'abord des informations de l'environnement physique (*e.g.*, température ambiante, lumière du soleil, bande passante, présence d'un utilisateur) à partir des objets communicants capables de les capturer. De tels objets communicants sont appelés des capteurs. Ces derniers peuvent être aussi bien physiques que logiciels. Les informations collectées sont ensuite interprétées, filtrées et agrégées par diverses applications, afin de les enrichir de données contextuelles dans le but d'obtenir et de partager des informations de plus haut niveau. À partir de ces dernières, certaines applications peuvent décider des actions à entreprendre (*e.g.*, allumer une lumière,

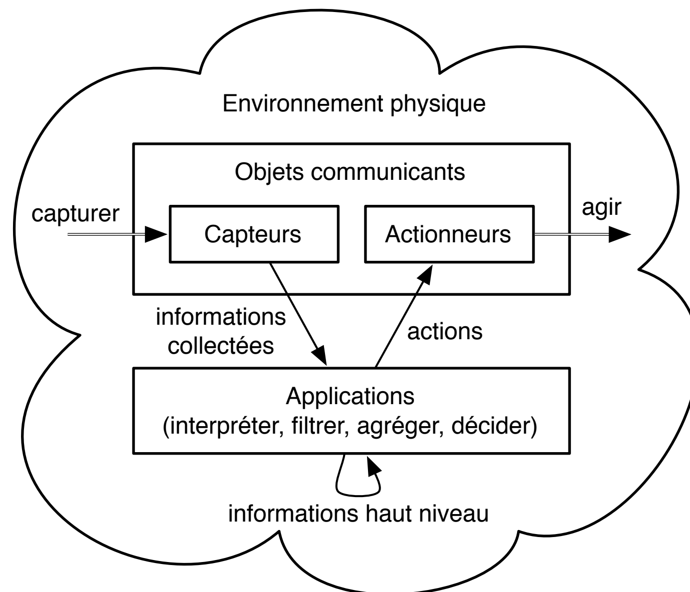

FIGURE 4.3 – Fonctionnement d'un système d'informatique ubiquitaire

déclencher une alarme, modifier un statut, afficher une information) par les objets communicants capables d'agir sur l'environnement physique. De tels objets communicants sont appelés des actionneurs. Un objet communicant peut être à la fois capteur et actionneur.

Les exemples de systèmes d'informatique ubiquitaire sont nombreux. Nous pouvons citer le traçage de l'ensemble des produits matériels, la détection précoce par réseaux de capteurs des accidents écologiques (*e.g.*, incendies), la gestion intégrée des bâtiments (*e.g.*, consommation d'énergie, sécurité) ou encore la surveillance des personnes âgées peu autonomes. Considérons l'exemple d'un système de sécurité pour se protéger contre les intrusions. Ce système est responsable de détecter et de signaler toutes les tentatives d'intrusion d'un individu dans un bâtiment. Pour cela, il peut nécessiter différents types d'objets communicants, tels que des détecteurs de mouvements pour collecter les coordonnées d'une intrusion, un agenda pour connaître les plages horaires de surveillance du bâtiment, diverses applications pour analyser les données collectées et déterminer si une intrusion a bien lieu afin d'y réagir, des alarmes sonores pour alerter les alentours, des caméras vidéo pour filmer l'intrusion, une plate-forme de téléphonie pour envoyer la vidéo de l'intrusion sur le téléphone portable du propriétaire, *etc.* Cet exemple donne une bonne idée de la taille et de la complexité qu'un système d'informatique ubiquitaire peut avoir, coordonnant une multitude d'objets communicants et requérant une expertise étendue dans de nombreux domaines, comme la programmation distribuée ou encore les télécommunications.

Les systèmes d'informatique ubiquitaire reposent sur deux notions centrales : le contexte à partir duquel ils réagissent et s'adaptent aux différents changements de l'environnement, et la découverte de services pour trouver quels objets communicants sont disponibles dans l'environnement.

4.2.1 Contexte

Il existe de nombreuses définitions, plus ou moins précises, de la notion de contexte en informatique ubiquitaire [Sch02, Dey00, Sch95]. Dans ce document, nous adoptons la définition de Dey *et al.* en considérant le contexte comme «*toutes les informations qui peuvent être utilisées pour caractériser la situation d'une entité (i.e., une personne, un lieu ou encore un objet) qui est considérée pertinente pour l'interaction entre un utilisateur et une application, y compris l'utilisateur et l'application eux-mêmes*» [DAS01]. Autrement dit, il s'agit de considérer toutes les informations susceptibles d'influencer le comportement du système. Typiquement, ces informations de contexte peuvent inclure des localisations, des identités, des statuts ou encore le temps. Toutefois, la nature du contexte dépend fortement du domaine d'application cible. Par exemple, dans le cas d'un système de détection automatique d'incendie, différentes informations de contexte peuvent être utiles pour caractériser une situation d'incendie, telles que le taux de dioxyde carbone, la température ambiante ou encore la présence ou non de fumée. Ainsi, à partir de plusieurs pièces d'information de contexte, il est possible d'inférer une nouvelle information de contexte qui pourra à son tour être utilisée par le système. Il apparaît donc important de bien comprendre quelles informations de l'environnement physique constituent le contexte dans un domaine d'application particulier, et de quelle manière elles sont représentées dans le système afin de pouvoir les collecter et les utiliser. Plusieurs approches ont été proposées pour représenter les informations de contexte d'un environnement d'informatique ubiquitaire, basées sur des ontologies [RWK+08, CFJ03], sur des bases de données virtuelles [JS03] ou encore sur d'autres modèles relationnels [HI06].

4.2.2 Découverte de services

Les systèmes d'informatique ubiquitaire nécessitent d'accéder aux différentes capacités offertes par les objets communicants disponibles. Pour cela, ils utilisent les mécanismes de découverte de services [ZMN05] afin de localiser dynamiquement les objets communicants dont ils ont besoin. La découverte de services permet aux développeurs de ne pas avoir à connaître où sont physiquement localisés les objets communicants. Ainsi, les systèmes ne sont pas statiquement liés à des objets communicants particuliers, ce qui leur permet de s'adapter plus facilement aux changements de l'environnement physique, puisque de nouveaux objets communicants peuvent apparaître et d'autres disparaître. La découverte de services peut être centralisée ou distribuée. Dans le premier cas, la découverte se fait à l'aide d'un annuaire qui centralise les descriptions des services et les adresses physiques associées des objets communicants. Dans le second cas, tous les objets communicants participent à la découverte de services, soit en envoyant périodiquement la description des services qu'ils fournissent, soit en envoyant des requêtes décrivant les caractéristiques des services dont ils ont besoin. Il apparaît donc important de pouvoir spécifier clairement quels types de services (*i.e.*, les capacités offertes par les objets communicants) sont disponibles dans un environnement, afin de les mettre à disposition des différents systèmes déployés.

4.3 Enjeux du développement

Le développement de systèmes d'informatique ubiquitaire est une tâche difficile, qui requiert à la fois de faire face à l'hétérogénéité des technologies et au caractère dynamique de l'environnement. De plus, ces systèmes sont généralement destinés à évoluer de manière

transparente dans un environnement humain, ce qui nécessite de s'intéresser à certaines préoccupations sociales, telles que la fiabilité des systèmes développés [EG01].

4.3.1 Hétérogénéité

Les systèmes d'informatique ubiquitaire sont fortement hétérogènes à plusieurs niveaux. Tout d'abord, ils sont déployés dans de nombreux et divers domaines d'application, tels que la domotique [JRS+09], la gestion des catastrophes routières [CCG+07], l'armée [AKY02] ou encore les soins hospitaliers [Jou06]. La diversité de ces domaines d'application implique toutes sortes d'objets communicants, aussi bien matériels (*e.g.*, téléphones portables, caméras vidéo, capteurs) que logiciels (*e.g.*, bases de données, agendas en ligne, clients de messagerie instantanée). Ces objets communicants ont des capacités riches et très variées, allant de la détection de mouvements à la réception d'un flux audio/vidéo, en passant par le stockage de données. De plus, afin de pouvoir coopérer, ils reposent sur des intergiciels spécifiques (*e.g.*, CORBA [OMG95], DCOM [2] [EE98], Java RMI [3] [Gro01], les services Web [CDK+02]) et supportent différents protocoles réseaux (*e.g.*, SIP [4] [RSC+02], UPnP [5], X10 [6]). Ils peuvent également proposer plusieurs modèles d'interaction : un vers un (*e.g.*, RPC [7] [BN84]) ou un vers plusieurs (*e.g.*, le modèle événementiel *publish/subscribe* [EFGK03]), synchrones ou asynchrones, proactifs ou réactifs. Les informations échangées entre les objets communicants sont elles aussi très hétérogènes. Elles peuvent différer non seulement par leur nature (*e.g.*, texte, image, flux audio) mais aussi par leur format (*e.g.*, GIF, JPEG ou encore PNG pour l'encodage des images). Toutefois, l'interopérabilité de toutes ces technologies n'est aujourd'hui pas assurée. L'enjeu consiste donc ici à abstraire toute la complexité et l'hétérogénéité technologiques afin de faciliter le développement des systèmes d'informatique ubiquitaire.

4.3.2 Caractère dynamique

Dans un système d'informatique ubiquitaire, la disponibilité des objets communicants peut varier au cours du temps. En effet, ces derniers peuvent être dynamiquement ajoutés ou retirés du système, de manière intentionnelle ou non. Par exemple, la portée limitée des technologies réseaux sans fil, associée à la mobilité des utilisateurs, peut placer hors de portée certains objets communicants. De plus, ces objets communicants ont généralement des ressources limitées en énergie et sont sujets à des défaillances matérielles et logicielles, ce qui peut les rendre indisponibles pour un temps indéfini. Le système doit alors s'adapter (*e.g.*, en découvrant de nouveaux objets communicants) afin de poursuivre son exécution et potentiellement reprendre certaines tâches interrompues. L'enjeu consiste donc ici à fournir un support de programmation pour découvrir les objets communicants disponibles dans l'environnement et pour coordonner leurs interactions malgré leur volatilité.

---

2. DCOM est l'acronyme anglais de *Distributed Component Object Model*.
3. RMI est l'acronyme anglais de *Remote Method Invocation*.
4. SIP est l'acronyme anglais de *Session Initiation Protocol*.
5. UPnP est l'acronyme anglais de *Universal Plug and Play*.
6. X10 est un protocole par courants porteurs.
7. RPC est l'acronyme anglais de *Remote Procedure Call*.

4.3.3 Fiabilité

L'utilisateur est au centre des systèmes d'informatique ubiquitaire. De ce fait, la fiabilité des systèmes avec lesquels il cohabite est essentielle. En effet, un mauvais fonctionnement pourrait entraîner des conséquences néfastes sur les activités de l'utilisateur. Par exemple, dans le cas d'un système d'aide à la personne, si un capteur (*e.g.*, de chute) venait à tomber en panne, la sécurité de la personne pourrait être compromise. De plus, chaque domaine d'application possède des propriétés critiques spécifiques, qu'il est essentiel que le système déployé garantisse. Par exemple, une propriété évidente du domaine de l'aide à la personne est qu'en présence d'une situation anormale, le système alerte automatiquement le personnel médical le plus rapidement possible. Vérifier un système d'informatique ubiquitaire avant son déploiement en environnement réel est une tâche très compliquée avec les techniques de vérification traditionnelles, notamment à cause de la taille des systèmes, pouvant impliquer un grand nombre d'objets communicants, du caractère dynamique de l'environnement et de certaines situations difficilement reproductibles (*e.g.*, un incendie). L'enjeu consiste donc ici à fournir des garanties de sûreté, soit par construction, soit par analyses, afin de développer et déployer des systèmes les plus robustes possibles vis-à-vis du domaine d'application cible.

4.4 État de l'art

Il existe de nombreuses solutions pour développer des systèmes d'informatique ubiquitaire. Ces solutions présentent des approches différentes du problème, reposant sur des intergiciels, des *frameworks* de programmation ou encore des langages dédiés.

4.4.1 Intergiciels distribués et *frameworks* de programmation

Une approche classique pour le développement de systèmes d'informatique ubiquitaire est d'utiliser des intergiciels distribués standardisés. Ces derniers facilitent le développement de systèmes distribués traditionnels, en masquant la complexité du réseau et en offrant divers mécanismes génériques (*e.g.*, annuaire de services, notification d'événements) pour supporter les interactions entre les différents objets communicants déployés. Les intergiciels distribués peuvent être regroupés en deux familles : les intergiciels synchrones et asynchrones [Men05].

Les premiers sont généralement fondés sur le paradigme des RPC [BN84]. Parmi les plus utilisés, nous pouvons citer CORBA [OMG95], DCOM [EE98], Java RMI [Gro01] ou encore les services Web [CDK+02]. En particulier, ces intergiciels fournissent un moyen (*e.g.*, un IDL pour CORBA ou une interface Java pour Java RMI) de définir non seulement une représentation abstraite des services offerts par les objets communicants (*i.e.*, leur signature), mais également une représentation intermédiaire des données échangées. Ces définitions sont ensuite utilisées pour générer du support générique de communication (*e.g.*, des *stubs*) et de découverte de services. Elles permettent donc d'abstraire une partie de l'hétérogénéité des systèmes distribués afin d'assurer l'interopérabilité des différents objets communicants.

Les interactions synchrones peuvent se révéler problématiques dans certaines situations. Une alternative est d'utiliser des communications asynchrones, *via* des échanges de messages ou d'événements. Contrairement aux intergiciels distribués synchrones, souvent proches d'un modèle de programmation client/serveur, les intergiciels distribués asynchrones permettent aux interactions entre les objets communicants de ne pas être liées par l'espace (*i.e.*, les connexions réseau) ou le temps. Ainsi, les objets communicants sont découplés les uns des autres, favori-

sant leur autonomie. Une famille représentative de tels intergiciels est les MOM [8] [BCSS99]. Il existe différents paradigmes de communication asynchrone : *message queuing* [Dic98, MHC00], *publish/subscribe* [CJ02, CCW03] ou encore *tuple space* [Gel85].

Toutefois, toutes ces solutions, synchrones ou asynchrones, restent peu adaptées pour faire face au caractère très dynamique des environnements d'informatique ubiquitaire. En effet, elles ont été développées pour des réseaux fixes et stables, et ne proposent par conséquent aucune abstraction de programmation pour gérer la mobilité des objets communicants ou encore la notion de contexte [MCE02]. De nouveaux intergiciels, capables de s'adapter en fonction de leur environnement d'exécution, ont donc été proposés. Ils s'appuient sur les travaux précédents, tout en élevant leur niveau d'abstraction. Parmi ceux-ci, nous pouvons citer Alice [HCC99], Dolmen [RB96], Aura [GSSS02], Gaia [RHC+02] one.world [Gri04], LIME [MPR06] ou encore TOTA [MZ09]. Ces approches fournissent des mécanismes et des abstractions de programmation dédiés pour répondre à certains besoins spécifiques des systèmes d'informatique ubiquitaire. Par exemple, le projet one.world simplifie la programmation des tâches de découverte de services, de migration ou encore de gestion des erreurs [GDL+04]. Le projet Aura, quant à lui, introduit un style architectural pour supporter la mobilité des utilisateurs [SG02]. En particulier, il permet de représenter les tâches des utilisateurs à un haut niveau d'abstraction et d'y associer dynamiquement les objets communicants disponibles les plus appropriés à leur réalisation. Un autre exemple est le projet Gaia qui utilise CORBA pour fournir du support dédié à la gestion des événements et du contexte, ou encore à la localisation des services. Il utilise également un langage de script impératif, nommé LuaOrb, pour simplifier la programmation des systèmes d'informatique ubiquitaire, et notamment la coordination entre les objets communicants [CCI99]. Toutefois, ce langage est peu expressif et interprété. Ranganathan *et al.* proposent alors un nouveau modèle de programmation haut niveau, nommé Olympus [RCAM+05]. Ce dernier utilise des ontologies [SS09] pour spécifier les types et les propriétés des différents objets communicants d'un système, et fournit certaines opérations courantes, telles que démarrer ou stopper un service. Les tâches des utilisateurs sont ensuite développées dans les termes des abstractions définies dans les descriptions ontologiques. Ainsi, le développeur est libéré des détails d'implémentation et de la complexité des technologies sous-jacentes, et peut se concentrer sur la logique même des tâches.

À l'instar d'Olympus, d'autres *frameworks* de programmation [Bar05] et boîtes à outils [EBF+08, GIM+04, BRSB03] ont été proposés pour aider au développement de systèmes d'informatique ubiquitaire. De telles approches se focalisent généralement sur un problème ou un domaine d'application particulier, tel que les environnements domestiques [HCH+03] ou encore la gestion des informations de contexte (*i.e.*, acquisition, représentation, diffusion, réaction) [HI06, CK02, DAS01]. De plus, la plupart du temps, elles fournissent un support de programmation générique, résultant dans des interfaces de programmation (*Application Programming Interface* ou API [9]) complexes qui peuvent nécessiter un processus d'apprentissage long et laborieux.

Un moyen de manipuler plus facilement les abstractions et les opérations définies dans une bibliothèque ou un *framework* est de les envelopper dans un langage dédié.

8. MOM est l'acronyme anglais de *Message Oriented Middleware.*
9. Dans ce document, nous utiliserons de manière indifférente le *terme interface de programmation* et *API.*

4.4.2 Langages dédiés distribués

Plusieurs langages dédiés au développement de systèmes d'informatique ubiquitaire ont été proposés, parmi lesquels nous pouvons citer AmbientTalk [DVCM+06], YABS [BC06], Habilitation [JRS+09] ou encore VisualRDK [WKU+07]. Ces langages sont fondés sur différents paradigmes de communication (*e.g.*, le modèle des acteurs [Agh86], *tuple space* [Gel85]) et de notation textuelle ou graphique. Ils offrent également des constructions syntaxiques et des abstractions de haut niveau qui sont proches des concepts du domaine de l'informatique ubiquitaire, et non plus des concepts traditionnels de programmation. Ainsi, de tels langages dédiés permettent de réduire de manière significative la taille et la complexité des programmes chargés de coordonner les différents objets communicants d'un système. Toutefois, aucun d'eux ne permet d'élever le niveau d'abstraction de la programmation des systèmes d'informatique ubiquitaire au-delà du code. Par conséquent, il est difficile pour les développeurs d'avoir une vision globale du système et ainsi de pouvoir garantir certaines propriétés du domaine cible.

Les solutions de développement de systèmes d'informatique ubiquitaire sont nombreuses et montrent toutes le besoin de programmer à un plus haut niveau d'abstraction, que ce soit en proposant des mécanismes et des notations dédiés, ou en introduisant des couches de spécification et d'architecture. Bien que ces solutions permettent de gérer une grande partie de la complexité des systèmes d'informatique ubiquitaire, elles n'exploitent pas suffisamment cette élévation du niveau d'abstraction pour réellement simplifier la programmation des systèmes, notamment en guidant les développeurs et en prenant en compte toutes les spécificités des domaines d'application pour lesquels les systèmes sont développés. Il en va de même pour le processus de vérification. En effet, certaines solutions offrent des mécanismes de tolérance aux fautes pour augmenter la robustesse des systèmes [CRC05, Gri04, PJKF03], d'autres, plus récemment, proposent de formellement spécifier un système d'informatique ubiquitaire afin de pouvoir garantir certaines propriétés [SW09, RC08], mais aucune des solutions existantes ne permet de facilement vérifier un système d'informatique ubiquitaire dans son ensemble.

4.5 Bilan

Les systèmes d'informatique ubiquitaire permettent d'orchestrer divers objets communicants afin d'automatiser certaines tâches quotidiennes des utilisateurs. Ils sont aujourd'hui déployés dans un nombre croissant de domaines d'application. Néanmoins, le développement de tels systèmes reste une tâche difficile qui demande à la fois une maîtrise de diverses technologies et une prise en compte des besoins spécifiques du domaine d'application cible. En particulier, l'hétérogénéité des objets communicants et le caractère dynamique des environnements compliquent considérablement leur processus d'implémentation et de vérification. Cependant, certains aspects des systèmes d'informatique ubiquitaire (*e.g.*, les informations de contexte, les activités des utilisateurs ou encore les types, les capacités et les interactions des objets communicants) peuvent être décrits à un haut niveau d'abstraction et utilisés pour faciliter et guider le processus de développement logiciel. En ce sens, de récentes approches ont émergé, à l'exemple du mécanisme de découverte de services de Olympus. Toutefois, leur exploitation de ces descriptions de haut niveau reste encore trop sommaire pour offrir un support de programmation adapté et certaines garanties de sûreté aux développeurs. Le domaine du développement de systèmes d'informatique ubiquitaire constitue donc un très bon candidat pour illustrer notre approche.

Chapitre 5

Bilan

Dans ce chapitre, nous présentons le bilan du contexte scientifique de cette thèse. Après avoir résumé les enjeux liés à l'étagement du développement logiciel, nous exposons les problématiques d'amélioration de la construction et de la vérification de systèmes logiciels à partir de descriptions de haut niveau.

5.1 Étagement du développement logiciel

L'augmentation de la taille et de la complexité des systèmes logiciels rend les processus de construction et de vérification beaucoup plus longs, coûteux et complexes. Afin de réduire les efforts des développeurs, il est indispensable d'élever le niveau d'abstraction des systèmes bien au-delà du code. Ainsi, il devient possible de se concentrer sur la définition du *quoi*, c'est-à-dire ce que le système doit être et faire, avant de passer au *comment*, c'est-à-dire la phase d'implémentation.

Dans le chapitre 2, nous avons mis en évidence l'importance de l'architecture logicielle dans le processus de développement logiciel. Elle vise à décrire à un haut niveau d'abstraction différents aspects fonctionnels et non fonctionnels d'un système (*i.e.*, le *quoi*), allant de sa structure logique au traitement des erreurs, en passant par la gestion de la sécurité. Ainsi, une description architecturale masque certains détails d'implémentation (*i.e.*, le *comment*), afin de faciliter la compréhension et le raisonnement (section 2.2). Toutefois, pour réellement tirer profit de ces descriptions de haut niveau dans les processus de construction et de vérification des systèmes, nous avons justifié le besoin de combler le fossé existant entre l'architecture d'un système (*i.e.*, l'intention de l'architecte) et son implémentation concrète (section 2.4). Nous avons également souligné la nécessité de prendre en compte tous les besoins et toutes les spécificités du domaine d'application pour lequel un système est développé, afin de faciliter de manière significative sa construction et sa vérification (section 2.5).

Dans le chapitre 3, nous avons introduit les langages dédiés qui apportent de nouvelles solutions à certains problèmes de développement logiciel. En particulier, ils permettent de restreindre l'espace de conception et de capturer toute la connaissance d'un domaine, offrant ainsi des abstractions de l'espace de problèmes (*i.e.*, du domaine), et non plus de l'espace de solutions (*i.e.*, des technologies informatiques). De ce fait, les langages dédiés présentent de nombreux avantages en termes de facilité de programmation et de vérification (section 3.2). Dans ce document, nous avons choisi de nous intéresser au domaine de l'informatique ubiquitaire.

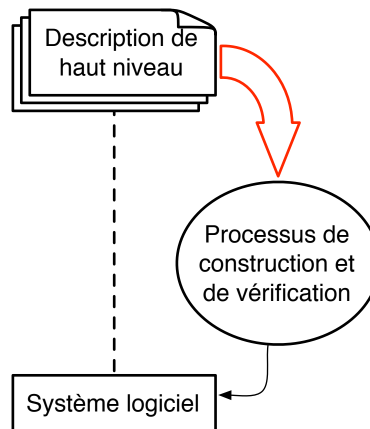

Figure 5.1 – Problématiques de simplification du développement logiciel

Dans le chapitre 4, nous avons montré que les systèmes d'informatique ubiquitaire sont des systèmes difficiles à construire et à vérifier avec les techniques existantes de programmation. Toutefois, différents aspects de ces systèmes (*e.g.*, les objets communicants qui composent le système, leurs capacités, leurs interactions ou encore les informations de contexte) peuvent être spécifiés à un haut niveau d'abstraction. L'enjeu consiste donc à exploiter ces différentes descriptions de haut niveau dans le processus de développement des systèmes d'informatique ubiquitaire, afin de le rendre plus simple et plus sûr. Ainsi, le domaine de l'informatique ubiquitaire permet d'illustrer parfaitement les problématiques de simplification du développement logiciel à partir de descriptions de haut niveau.

5.2 Problématiques

La figure 5.1 résume schématiquement les problématiques soulevées par la complexité et la taille croissantes des systèmes logiciels. Pour appréhender ces dernières, les développeurs disposent de vues simplifiées décrivant certains aspects des systèmes à un haut niveau d'abstraction. Ces descriptions de haut niveau permettent aux développeurs de bien comprendre ce qu'un système devrait faire, indépendamment des détails d'implémentation. Les informations fournies par ces descriptions doivent être prises en compte dans les processus de construction et de vérification des systèmes, afin de simplifier le développement logiciel et augmenter la productivité. Dans cette optique, les outils de développement nécessitent d'offrir aux développeurs un support de programmation adapté et des garanties de sûreté, conformément aux informations spécifiées dans les descriptions de haut niveau.

Support de programmation. Afin de faciliter le développement de systèmes logiciels, les développeurs nécessitent un support de programmation adapté à leurs besoins. L'enjeu consiste donc à leur fournir des abstractions de programmation spécifiques pour guider et supporter l'implémentation des différents aspects définis dans les descriptions de haut niveau. Or, les approches de description d'architecture traditionnelles proposent un support de programmation très sommaire, générique et surtout statique (section 2.3). Ainsi, ces approches ne permettent

pas d'aider les développeurs à résoudre tous les problèmes d'un domaine d'application efficacement. Dans le domaine de l'informatique ubiquitaire, certains travaux de recherche que nous avons présentés dans la section 4.4 utilisent des descriptions de haut niveau afin de simplifier certaines tâches des systèmes d'informatique ubiquitaire, comme la gestion des informations de contexte (section 4.2.1) ou la découverte de services (section 4.2.2). Toutefois, ces approches permettent seulement de spécialiser certains mécanismes et opérations génériques, mais en aucun cas d'assister rigoureusement les développeurs dans le processus de développement de systèmes d'informatique ubiquitaire.

Garanties de sûreté. Les descriptions de haut niveau peuvent exprimer facilement des propriétés et des contraintes, à la fois architecturales et du domaine, auxquelles les développeurs nécessitent de se conformer pour augmenter la sûreté des systèmes. L'enjeu consiste donc à assurer que les systèmes sont développés dans le respect de ces propriétés et contraintes. Or, comme indiqué dans la section 2.4.2, garantir la cohérence entre l'architecture d'un système et son implémentation est un problème délicat, compte tenu du fossé d'abstraction qui existe. Aussi, contrairement aux analyses architecturales (section 2.2.2), le raisonnement sur le code d'implémentation peut s'avérer très difficile, du fait de la complexité des technologies existantes et des solutions de développement. Nous avons illustré ce point dans le domaine de l'informatique ubiquitaire. Les nouveaux défis que pose le développement de systèmes d'informatique ubiquitaire (section 4.3) rendent particulièrement difficiles une analyse statique complète de ces systèmes, ainsi que la vérification de propriétés.

Approches inadaptées. Les langages de description d'architecture (section 2.3.3) n'apportent pas de réponses satisfaisantes pour rendre plus simple et plus sûr le développement de systèmes logiciels. Nous avons montré les limites de ces solutions, dues à leur indépendance vis-à-vis des langages d'implémentation (section 2.4), mais aussi à leur approche généraliste du problème (section 2.5). Ainsi, les développeurs sont obligés de tenir compte manuellement des différentes informations spécifiées dans les descriptions de haut niveau, ce qui peut les amener à introduire involontairement certaines incohérences et anomalies dans l'implémentation des systèmes.

Il apparaît donc que simplifier les processus de construction et de vérification de systèmes logiciels en tirant profit de diverses informations décrites à un haut niveau d'abstraction est d'un réel intérêt. La partie II de ce document présente notre approche pour y parvenir.

Deuxième partie

Approche proposée

Chapitre 6

Présentation de l'approche

Un changement drastique d'échelle s'est opéré ces dernières années dans la taille et la complexité des systèmes logiciels, les rendant de plus en plus difficiles à construire et à vérifier. Maîtriser la taille et la complexité croissantes des systèmes, tout en leur assurant une meilleure sûreté et fiabilité, est aujourd'hui un véritable défi scientifique. Les travaux présentés dans cette thèse visent à rendre le développement logiciel plus simple et plus sûr. Dans ce chapitre, nous présentons la vision globale de notre approche pour répondre aux problèmes posés par le développement de systèmes complexes de grande taille.

6.1 Développement logiciel plus simple et plus sûr

Comme nous l'avons vu dans le chapitre précédent, élever le niveau d'abstraction de la programmation des systèmes logiciels au-delà du code est nécessaire pour en faciliter la compréhension et simplifier le raisonnement. Toutefois, cela n'est pas suffisant pour assister significativement les développeurs dans les processus de construction et de vérification des systèmes. En effet, un fossé subsiste entre l'implémentation écrite par les développeurs et les descriptions de haut niveau de cette implémentation, utilisées *a priori* pour la construction, et *a posteriori* pour la vérification. Pour répondre à ce problème, nous proposons de nous placer dans un domaine d'application particulier pour lequel nous identifions clairement les notions de problème et de solution, puis d'utiliser une approche générative qui tire profit des diverses informations spécifiques définies dans les descriptions de haut niveau afin de fournir aux développeurs le support de programmation et les garanties de sûreté nécessaires pour programmer plus simplement et plus sûrement. Ainsi, la programmation devient rigoureusement dirigée par les descriptions de haut niveau, guidant le processus de développement et permettant aux développeurs de se concentrer sur la logique du programme. Il devient également possible d'assurer automatiquement et systématiquement que les systèmes développés sont bien conformes à l'ensemble des directives spécifiées dans les descriptions de haut niveau.

6.2 Approche

L'approche que nous proposons pour rendre plus simple et plus sûr le développement de systèmes logiciels repose sur l'utilisation de langages dédiés et sur un couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation. Une vue générale de notre approche est illustrée à la figure 6.1. La couche de spécification et d'architecture fournit

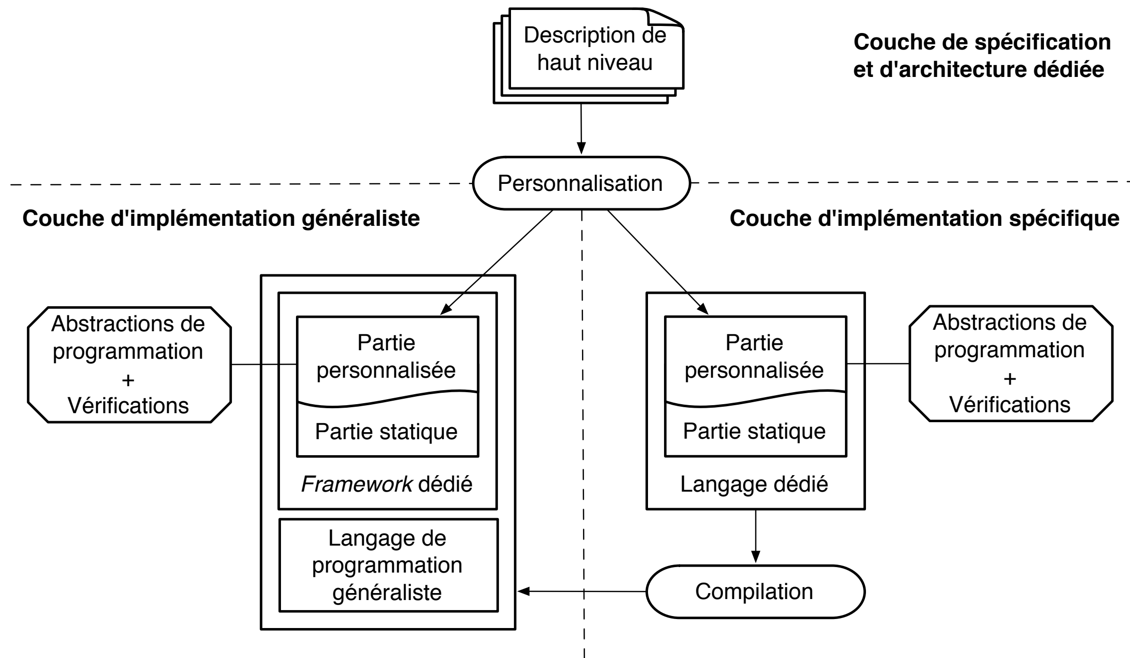

FIGURE 6.1 – Vue générale de notre approche

tout d'abord des constructions dédiées pour décrire à un haut niveau d'abstraction différents aspects (*e.g.*, structure logique, comportement, gestion des erreurs) du système à développer. Ces descriptions de haut niveau sont ensuite analysées et utilisées pour personnaliser la couche d'implémentation associée. Il peut s'agir par exemple d'une API ou d'un *framework* de programmation dédié au-dessus d'un langage de programmation généraliste, ou encore d'un langage dédié. Le processus de personnalisation consiste à enrichir et spécialiser une partie de la couche d'implémentation (*e.g.*, les structures et les opérations fournies par une API ou un *framework*, ou encore la syntaxe et la sémantique d'un DSL), afin de fournir le support de programmation et les garanties de sûreté appropriés pour aider et guider les développeurs dans le développement de systèmes logiciels pour le domaine considéré.

Ce couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation, ainsi que la nature dédiée des couches, permettent d'offrir aux développeurs des abstractions de programmation haut niveau et dédiées à l'implémentation des différents aspects du système décrits. De telles abstractions de programmation permettent de résoudre plus facilement les problèmes du domaine cible, simplifiant ainsi la construction de systèmes. De plus, elles peuvent rendre certaines propriétés critiques du domaine apparentes dans la structure des programmes, facilitant ainsi la vérification de systèmes. Il devient alors possible de réaliser des vérifications statiques qui seraient difficiles, voire impossibles, dans le cadre de développements traditionnels, améliorant ainsi la sûreté et la fiabilité des systèmes développés. Ces vérifications garantissent, entre autres, la cohérence entre les descriptions de haut niveau fournies par la couche de spécification et d'architecture (*i.e.*, les intentions et les choix de l'architecte) et les implémentations des systèmes, ou encore le respect des propriétés du domaine d'application.

Tous ces bénéfices de simplicité de programmation et de sûreté concernent aussi bien l'implémentation des aspects fonctionnels (*e.g.*, les interactions entre les différentes parties du système) que non fonctionnels (*e.g.*, la gestion des erreurs) d'un système. Ainsi, l'approche proposée permet de couvrir différentes préoccupations d'implémentation, à la fois fonctionnelles et non fonctionnelles, selon les besoins du domaine. De plus, les bénéfices de ce couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation augmentent avec le degré de spécificité de la couche d'implémentation. En effet, plus cette dernière est haut niveau (*i.e.*, proche des concepts du domaine), plus les abstractions de programmation et les vérifications fournies peuvent être fortes, rendant ainsi le processus de développement logiciel encore plus simple et plus sûr.

Les couches d'implémentation couplées à des couches de spécification et d'architecture similaires sont amenées à partager un certain nombre de caractéristiques, ce qui ouvre des perspectives intéressantes de compilation. En effet, les couches d'implémentation de plus haut niveau peuvent être facilement compilées dans des couches d'implémentation de plus bas niveau personnalisées à partir des mêmes informations, comme illustré à la figure 6.1.

Dans le reste de ce document, nous appliquons l'approche que nous venons de présenter au domaine du développement de systèmes d'informatique ubiquitaire et évaluons son impact sur les processus de construction et de vérification des systèmes. Pour cela, nous introduisons dans le chapitre 7 différents couplages entre des couches de spécification et d'architecture et des couches d'implémentation, dédiées au développement de systèmes d'informatique ubiquitaire et évoluant à des niveaux d'abstraction différents. Coupler fortement de telles couches contribue d'une part à faciliter et guider la programmation des systèmes d'informatique ubiquitaire, fournissant des abstractions de programmation de haut niveau et dédiées (*e.g.*, pour la découverte de services), et d'autre part à améliorer la sûreté et la fiabilité de ces systèmes, garantissant statiquement des propriétés critiques du domaine. Enfin, nous présentons dans le chapitre 8 les avantages de notre approche pour simplifier la gestion des erreurs dans les systèmes d'informatique ubiquitaire.

Chapitre 7

Couplage

Dans ce chapitre, nous illustrons notre approche dans le domaine de l'informatique ubiquitaire. Nous présentons une analyse de domaine et de famille des systèmes d'informatique ubiquitaire. Nous décrivons ensuite trois déclinaisons de notre approche, développées à partir des résultats de cette analyse et fondées sur un couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation. Ces déclinaisons permettent de programmer des systèmes d'informatique ubiquitaire à des niveaux d'abstraction différents et présentent aussi un degré de spécificité différent au niveau de leur couche de spécification et d'architecture. Nous introduisons tout d'abord un langage dédié à la description d'architecture de systèmes d'informatique ubiquitaire, nommé DiaSpec [1] [CBCL11, CBM+10, CBLC09], qui permet de générer un *framework* de programmation dédié dans un langage de programmation généraliste. Puis, nous introduisons deux langages dédiés que nous avons développés, Pantaχou [MPCL08], un langage de script, et Pantagruel [DMC09], un langage visuel, qui permettent de faciliter l'orchestration des différents objets communicants d'un système d'informatique ubiquitaire. Nous illustrons ces différents couplages à l'aide d'un exemple concret.

7.1 Analyse de domaine et de famille

Cette section présente l'élaboration d'un cahier des charges. Notre objectif est d'obtenir les données d'entrée permettant de guider la conception à la fois des couches de spécification et d'architecture et des couches d'implémentation, dédiées au développement de systèmes d'informatique ubiquitaire. Pour cela, nous avons décrit dans la section 3.3.1 différentes analyses et méthodologies qui visent à identifier, à partir de diverses sources d'information, les points communs et les variations des systèmes logiciels d'un domaine, mais aussi les propriétés et les concepts importants pour ce domaine. Ainsi, afin de mener à bien notre analyse de domaine et de famille, nous avons présenté dans le chapitre 4 une étude sur les systèmes d'informatique ubiquitaire. En particulier, nous avons étudié leur fonctionnement, les défis à relever et différentes approches de développement existantes. Cette étude a été étayée par des informations récoltées dans la littérature. Nous avons notamment recensé et étudié un certain nombre d'exemples de systèmes d'informatique ubiquitaire développés avec les approches existantes. Cette étude montre finalement la terminologie utilisée dans le domaine des systèmes d'informatique ubiquitaire, ainsi que les objets de base du domaine (*i.e.*, les objets communicants, les interactions et les informations de contexte) avec leurs opérations, relations et contraintes.

1. Le langage DiaSpec est développé au sein de l'équipe-projet PHOENIX.

Nous présentons maintenant les notions et les abstractions jugées nécessaires pour déclarer et manipuler ces objets du domaine au niveau d'une couche de spécification et d'architecture et d'une couche d'implémentation dédiées au développement de systèmes d'informatique ubiquitaire.

Couche de spécification et d'architecture. Cette couche vise à masquer la complexité et l'hétérogénéité des systèmes d'informatique ubiquitaire (section 4.3.1). Pour cela, il convient de décrire de manière abstraite quels types d'objets communicants et quels types d'informations de contexte sont pertinents pour un domaine donné (*e.g.*, la domotique, l'aide à la personne), comment ces objets communicants interagissent et comment ils sont liés les uns aux autres. Cette couche doit donc introduire un ensemble de constructions et de notations pour spécifier à la fois les types de données échangées et les interfaces à travers lesquelles les objets communicants sont considérés et manipulés. Une interface représente un type d'objet communicant en termes de propriétés, de capacités à capturer les informations de contexte et de capacités à agir sur l'environnement. Les propriétés permettent de caractériser les objets communicants et les capacités permettent de fournir respectivement les données et les actions intervenant dans la logique du système d'informatique ubiquitaire. Une interface peut également inclure d'autres informations, notamment relatives à des préoccupations non fonctionnelles telles que la sécurité ou encore la gestion des erreurs. Un objet communicant peut avoir plusieurs interfaces, chacune modélisant une facette particulière de l'objet communicant et pouvant être exploitée indépendamment des autres. Ainsi, les objets communicants deviennent polymorphes : selon l'interface considérée, ils sont utilisés de manière différente, se focalisant sur certains aspects particuliers de l'objet communicant. Ensuite, une interface peut étendre une autre interface, afin d'en créer une version plus spécifique. Elle hérite alors des propriétés, des capacités et de toute autre information de l'interface qu'elle étend, et peut en ajouter de nouvelles. L'organisation hiérarchique des interfaces permet à une interface plus spécifique d'être utilisée là où une interface moins spécifique est attendue, offrant ainsi un certain degré de généricité. De même, les types de données échangées doivent pouvoir être organisés hiérarchiquement. Enfin, cette couche doit permettre d'architecturer un système d'informatique ubiquitaire, en déclarant les interactions autorisées entre les objets communicants, et plus précisément entre les interfaces des objets communicants, dans la couche d'implémentation.

Couche d'implémentation. Cette couche vise à implémenter l'orchestration des différents objets communicants d'un système d'informatique ubiquitaire. Il est donc nécessaire d'introduire dans cette couche des constructions et des opérateurs spécifiques pour découvrir et faire interagir les objets communicants en fonction des informations définies dans la couche de spécification et d'architecture. Ces opérateurs doivent, par exemple, permettre d'accéder aux propriétés ou aux capacités d'un objet communicant. Cette couche doit également gérer de manière transparente la nature distribuée et le caractère dynamique des systèmes d'informatique ubiquitaire (section 4.3.2), de manière à n'exposer que des constructions et des opérations de haut niveau au développeur. Ainsi, ce dernier est libéré de la complexité des technologies sous-jacentes et peut se concentrer uniquement sur la logique d'orchestration. Enfin, les systèmes d'informatique ubiquitaire nécessitent d'être développés avec des contraintes particulières de fiabilité (section 4.3.3). Cette couche doit donc permettre la vérification de propriétés à différents moments du processus de développement, que ce soit statiquement ou dynamiquement. Il peut s'agir, par exemple, de garantir la conformité d'une implémentation

par rapport aux déclarations de la couche de spécification et d'architecture. Un autre exemple concerne l'intégrité des communications (section 2.5), assurant que seuls les objets communicants connectés au niveau de l'architecture peuvent communiquer entre eux au niveau de l'implémentation. Il doit également être possible de vérifier certaines propriétés critiques du domaine d'application pour lequel le système d'informatique ubiquitaire est développé.

Les résultats de notre analyse de domaine et de famille des systèmes d'informatique ubiquitaire forment un cahier des charges pour les couches de spécification et d'architecture et les couches d'implémentation, dédiées au développement de systèmes d'informatique ubiquitaire. De plus, ces couches doivent reprendre la terminologie du domaine que nous avons identifiée.

7.2 Exemple de travail

Pour illustrer notre approche, nous introduisons un système de gestion des incendies au sein d'un bâtiment. Ce système permet de détecter un incendie en analysant les données produites par les détecteurs de fumée et les capteurs de température du bâtiment. Lorsqu'un incendie est détecté, le système est chargé de déclencher les alarmes et les sprinklers, et de libérer les portes coupe-feu. D'autres mesures d'urgence peuvent être prises, comme par exemple prévenir les pompiers. Ce système fait partie d'un projet plus large visant à automatiser la gestion de l'ENSEIRB [2]. Il s'agit, par exemple, de gérer la consommation d'énergie du bâtiment ou encore la diffusion d'informations aux étudiants.

Dans les sections suivantes, nous montrons comment spécifier, architecturer et implémenter ce système dans trois solutions de programmation : DiaSpec, Pantaχou et Pantagruel. Pour chacune de ces solutions, nous évaluons l'impact du couplage sur le développement d'un tel système. D'autres exemples de systèmes d'informatique ubiquitaire ont été développés avec ces solutions dans des domaines d'application très différents, tels que les télécommunications [MPCL08] ou encore l'aide à la personne [DMC09]. Ces exemples illustrent eux aussi les avantages de notre approche.

7.3 DiaSpec, un langage de spécification et d'architecture

DiaSpec [CBCL11, CBM+10, CBLC09] est un langage dédié à la description d'architecture de systèmes d'informatique ubiquitaire. À partir d'une spécification DiaSpec, un *framework* de programmation dédié peut être généré en Java. Ainsi, il s'agit d'un couplage entre une couche de spécification et d'architecture très spécifique et une couche d'implémentation généraliste.

7.3.1 Définition du langage

Le langage DiaSpec se décompose en deux couches. La première couche permet de spécifier les types de données échangées et les interfaces des objets communicants pour un domaine d'application donné (*e.g.*, la domotique). La seconde couche permet de décrire l'architecture d'un système d'informatique ubiquitaire (*e.g.*, un système de gestion des incendies), étant donnée une définition de types de données et d'interfaces. DiaSpec offre des constructions de

2. L'ENSEIRB (École Nationale Supérieure d'Électronique, d'Informatique et de Radiocommunications de Bordeaux) est une école d'ingénieurs spécialisée dans les technologies de l'information et de la communication.

```
1  enumeration LockedStatus {LOCKED, UNLOCKED}
2  enumeration AlarmType {FIRE, INTRUSION}
3  enumeration TemperatureUnit {CELSIUS, FAHRENHEIT}
4  enumeration Accuracy {LOW, NORMAL, HIGH}

6  structure Temperature {
7    value as Integer;
8    unit as TemperatureUnit;
9  }
10 structure Smoke {
11   isDetected as Boolean;
12 }
```

FIGURE 7.1 – Extrait de la spécification des types de données (DiaSpec)

haut niveau et dédiées dans ces deux couches. Une syntaxe complète de DiaSpec est disponible dans la thèse de Damien Cassou [Cas11].

Couche de spécification

La couche de spécification de DiaSpec fournit la possibilité d'utiliser des types primitifs (*e.g.*, les booléens, les entiers ou encore les chaînes de caractères), ainsi que de créer ses propres types de données. Ces derniers peuvent être soit des énumérations, soit des structures, comme illustré à la figure 7.1. Une énumération est un ensemble fini de constantes (*e.g.*, le type énuméré `LockedStatus`, ligne 1) et une structure est un ensemble de champs nommés et typés (*e.g.*, la structure `Temperature`, lignes 6 à 9).

Une fois les types de données définis, il s'agit de déclarer les différentes interfaces des objets communicants. En DiaSpec, les interfaces sont introduites par le mot clé **device**, comme illustré à la figure 7.2. Les propriétés d'une interface sont déclarées en utilisant le mot clé **attribute**. Par exemple, l'interface `Device` définit l'attribut `location` de type `Location` (ligne 2). Les valeurs des attributs sont ensuite utilisées pour la découverte d'objets communicants (*e.g.*, pour découvrir un capteur de température dans une pièce particulière). Les capacités à capturer les informations de contexte d'une interface sont déclarées en utilisant le mot clé **source**. Une interface inclut une source de données nommée et typée pour chaque information de contexte qu'elle est capable de capturer. Par exemple, l'interface `TemperatureSensor` définit la source de données `temperature` de type `Temperature` (ligne 11). Notons que la déclaration d'une source de données ne tient pas compte de la manière dont les données sont fournies par les objets communicants, c'est-à-dire soit par publication d'un événement (*i.e.*, mode *push*), soit en réponse à une requête (*i.e.*, mode *pull*). Les capacités à agir sur l'environnement d'une interface sont déclarées en utilisant le mot clé **action**. Par exemple, l'interface `Sprinkler` définit l'action `OnOff` (ligne 14), permettant d'actionner les sprinklers. Une action correspond à un ensemble de déclarations de méthodes (lignes 23 à 26). Ces méthodes n'ont pas de type de retour, assurant une séparation claire entre le contexte à partir duquel le système d'informatique ubiquitaire réagit et les actions qu'il entreprend sur l'environnement.

```
1  device Device {
2    attribute location as Location;
3  }
4  device FireSensor extends Device {}
5  device FireActuator extends Device {}
6  device SmokeDetector extends FireSensor {
7    source smoke as Smoke;
8  }
9  device TemperatureSensor extends FireSensor {
10   attribute accuracy as Accuracy;
11   source temperature as Temperature;
12 }
13 device Sprinkler extends FireActuator {
14   action OnOff;
15 }
16 device FireDoor extends FireActuator {
17   action Release;
18 }
19 device Alarm extends Device {
20   action Activation;
21 }

23 action OnOff {
24   on();
25   off();
26 }
```

Figure 7.2 – Extrait de la spécification des interfaces (DiaSpec)

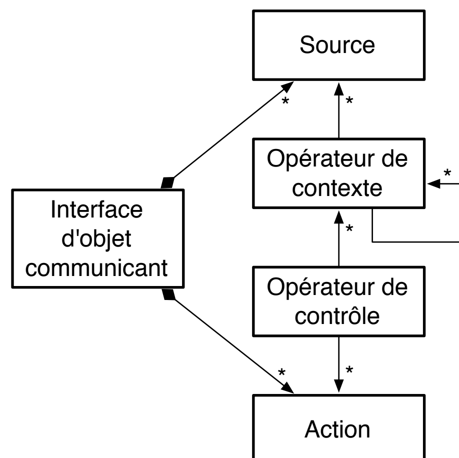

Figure 7.3 – Style architectural de DiaSpec

```
1  context SmokeDetected as Boolean
2    indexed by location as Location {
3    source smoke from SmokeDetector;
4  }
5  context AverageTemperature as Temperature
6    indexed by location as Location {
7    source temperature from TemperatureSensor;
8  }
9  context FireState as Boolean
10   indexed by location as Location {
11   context SmokeDetected;
12   context AverageTemperature;
13 }

15 controller FireController {
16   context FireState;
17   action Release on FireDoor;
18   action OnOff on Sprinkler;
19   action Activation on Alarm;
20 }
```

FIGURE 7.4 – Extrait de l'architecture du système de gestion des incendies (DiaSpec)

Couche d'architecture

La couche d'architecture de DiaSpec est fondée sur un style architectural, illustré à la figure 7.3. Ce style architectural décompose la couche de logique applicative d'un système d'informatique ubiquitaire (figure 4.3) en opérateurs de contexte et de contrôle. Un opérateur de contexte construit une donnée à partir de données fournies par les sources des objets communicants et d'autres opérateurs de contexte. Les implémentations de ces opérateurs interprètent, filtrent et agrègent les données afin de les adapter aux besoins du système. Un opérateur de contrôle reçoit ensuite des données d'opérateurs de contexte et détermine les actions à faire exécuter aux objets communicants du système d'informatique ubiquitaire.

L'architecture de notre système de gestion des incendies est illustrée à la figure 7.4. Dans cet exemple, les objets communicants de type `SmokeDetector` détectent la présence de fumée et ceux de type `TemperatureSensor` estiment la température ambiante. Ces informations sont respectivement utilisées par les opérateurs de contexte `SmokeDetected` (lignes 1 à 4) et `AverageTemperature` (lignes 5 à 8), déclarés en utilisant le mot clé **context**. En particulier, considérons l'opérateur de contexte `SmokeDetected`. Cet opérateur est chargé de déterminer si une pièce est enfumée. Pour cela, il agrège et traite les sources de données `smoke` produites par des objets communicants de type `SmokeDetector` (ligne 3). À partir de ces données, l'opérateur `SmokeDetected` peut fournir une valeur de type `Boolean`, signalant la présence ou l'absence de fumée dans une pièce. Pour faciliter l'utilisation de cette information par d'autres opérateurs, celle-ci est liée à un index `location` de type `Location`, indiquant où la fumée est détectée (ligne 2). Un index permet donc d'enrichir d'informations contextuelles la donnée fournie par un opérateur de contexte. Ensuite, l'opérateur de contexte `FireState` combine les données fournies par les opérateurs de contexte `SmokeDetected` et `AverageTemperature`, afin de déterminer si un incendie a lieu dans la pièce considérée. Cette information est transmise à l'opérateur de contrôle `FireController` (lignes 15 à 20), déclaré en utilisant le mot

```
1  public class MySmokeDetected extends SmokeDetected {
2    public MySmokeDetected() {
3      allSmokeDetectors().subscribeSmoke(this);
4    }
5    @Override
6    public void smokeChanged(SmokeDetector smokeDetector, Smoke smokeDetected) {
7      Location location = smokeDetector.getLocation();
8      if (smokeDetected.isDetected) {
9        setSmokeDetected(location, new Boolean(true));
10       return;
11     }
12     SmokeDetectorComposite detectors = select(smokeDetectorWhere().location(location));
13     for (SmokeDetector sd : detectors) {
14       try {
15         if (sd.getSmoke().isDetected) {
16           setSmokeDetected(location, new Boolean(true));
17         }
18       } catch (DiaGenCommunicationException e) { [...] }
19     }
20     setSmokeDetected(location, new Boolean(false));
21   }
22 }
```

FIGURE 7.5 – Une implémentation de l'opérateur de contexte `SmokeDetected` (DiaSpec/Java)

clé **controller**. Cet opérateur est chargé d'éteindre l'incendie et de prendre certaines mesures d'urgence. Pour cela, il invoque les actions `Release`, `OnOff` et `Activation` fournies respectivement par les objets communicants de type `FireDoor`, `Sprinkler` et `Alarm` (lignes 17 à 19).

7.3.2 Génération d'un *framework* de programmation dédié

À partir des déclarations de la couche de spécification et d'architecture, le compilateur de DiaSpec génère un *framework* de programmation Java qui fournit un support d'implémentation et d'exécution dédié au développement de systèmes d'informatique ubiquitaire. Ce *framework* dédié contient une classe abstraite pour chaque déclaration de composant DiaSpec (*i.e.*, interface, opérateur de contexte et opérateur de contrôle). Ces classes abstraites comprennent des méthodes générées pour supporter l'implémentation des composants (*e.g.*, des opérations pour la découverte d'objets communicants ou encore pour la gestion des événements). Elles comprennent également des déclarations de méthodes abstraites pour permettre aux développeurs d'implémenter la logique applicative (*e.g.*, réagir au changement d'une source de données ou encore invoquer une action). Implémenter un composant DiaSpec nécessite de créer une sous-classe de la classe abstraite générée correspondante, obligeant les développeurs à implémenter chaque méthode abstraite. Ainsi, les développeurs écrivent le code des composants dans les sous-classes, et non dans les classes abstraites générées. Ceci permet de guider précisément les développeurs dans le processus de développement et aussi d'éviter la programmation par *stub*, toujours un peu pénible, notamment en cas de régénération. Le *framework* de programmation généré contient également des *proxies* (*i.e.*, une vue locale d'un objet communicant) et s'appuie sur des technologies de systèmes distribués existantes (*e.g.*, RMI, SIP), afin de masquer et gérer les détails de bas niveau.

La figure 7.5 représente un extrait d'une implémentation de l'opérateur de contexte `SmokeDetected`. Cette implémentation étend la classe abstraite générée correspondante. Parce que l'opérateur de contexte `SmokeDetected` dépend de sources de données `smoke` fournies par des objets communicants de type `SmokeDetector`, le *framework* généré fournit les méthodes nécessaires pour découvrir, sélectionner et interagir avec les objets communicants de type `SmokeDetector`. Par exemple, la méthode `allSmokeDetectors` permet de découvrir tous les objets communicants de type `SmokeDetector` disponibles dans le bâtiment (ligne 3). La méthode `subscribeSmoke`, quant à elle, permet de souscrire à la source de données `smoke`. La méthode `smokeChanged` est automatiquement appelée par le *framework* lorsqu'un objet communicant de type `SmokeDetector` publie sa source de données `smoke` (ligne 6). La méthode `getSmoke`, quant à elle, permet d'accéder à la source de données `smoke` des objets communicants de type `SmokeDetector` (ligne 15). La méthode `setSmokeDetected` permet de publier un événement de type `Boolean`, indiquant la présence ou l'absence de fumée dans une pièce donnée (ligne 16).

7.3.3 Couplage avec un langage généraliste

DiaSpec permet de coupler fortement des descriptions architecturales avec le langage de programmation généraliste Java. Grâce à ce couplage, l'implémentation d'un système d'informatique ubiquitaire est rigoureusement guidée par les descriptions de la couche de spécification et d'architecture de DiaSpec, définissant quels objets communicants peuvent être découverts et comment interagir avec eux. Pour illustrer ce couplage, reprenons l'exemple de l'opérateur de contexte `SmokeDetected` donné à la figure 7.5. Conformément à sa déclaration, son implémentation peut seulement interagir avec des objets communicants de type `SmokeDetector` *via* la source de données `smoke` (figure 7.4, ligne 3). Ces déclarations personnalisent le support fourni par le *framework* de programmation généré, afin de guider l'implémentation de l'opérateur de contexte. En effet, les opérations de découverte d'objets communicants et d'interaction sont statiquement typées par rapport aux déclarations de la couche de spécification et d'architecture. Par exemple, lorsque un développeur implémente l'opérateur de contexte `SmokeDetected`, la méthode `select` permet seulement de rechercher des objets communicants de type `SmokeDetector` (figure 7.5, ligne 12). Elle est également paramétrée par les attributs de l'interface `SmokeDetector` (*i.e.*, l'attribut `location` hérité de l'interface `Device`). Cette méthode retourne une collection de *proxies* pour les objets communicants sélectionnés. Dans notre cas, ces *proxies* donnent uniquement accès à la source de données `smoke`, empêchant les développeurs d'accéder aux autres sources de données et actions des objets communicants sélectionnés.

Ce couplage fort permet donc de guider et de faciliter la construction de systèmes d'informatique ubiquitaire. Il permet aussi à la découverte d'objets communicants et aux interactions d'être statiquement vérifiées par le compilateur de DiaSpec, augmentant la sûreté des systèmes d'informatique ubiquitaire. En outre, il vérifie *par génération* que l'implémentation d'un système d'informatique ubiquitaire est conforme à l'architecture décrite. Toutefois, l'utilisation d'un langage de programmation généraliste comme Java dans la couche d'implémentation limite les possibilités offertes par le couplage, à la fois en termes de facilité de programmation et de vérification. Une solution consiste alors à utiliser un langage dédié dans la couche d'implémentation. Pour étudier cette solution, nous avons développé deux langages dédiés, Pantaχou et Pantagruel.

7.4 Pantaχou, un langage de script

Pantaχou[3] [MPCL08] couple fortement une couche de spécification et d'architecture, moins spécifique que DiaSpec, mais avec un langage de script dédié à l'orchestration d'objets communicants, donc de plus haut niveau que Java. La sémantique de Pantaχou a été formellement définie et un compilateur a été développé. La définition formelle du langage [MPCL08] sert de fondation pour la définition d'analyses de programme (*e.g.*, intégrité des communications).

7.4.1 Définition du langage

Le langage Pantaχou se décompose en deux couches. La première couche permet de spécifier les types de données échangées, les interfaces des objets communicants ainsi que leurs dépendances. La seconde couche permet d'orchestrer les différents objets communicants d'un système d'informatique ubiquitaire, dans le respect des informations définies dans la première couche. Pantaχou offre des constructions de haut niveau et dédiées dans ces deux couches. Une syntaxe complète de Pantaχou est disponible dans la thèse de Nicolas Palix [Pal08].

Couche de spécification et d'architecture

Pantaχou permet de spécifier les types de données échangées de manière similaire à DiaSpec. Les interfaces, quant à elles, sont introduites par le mot clé **service**, comme illustré à la figure 7.6. À l'instar de DiaSpec, les propriétés d'une interface sont décrites par des attributs. Par exemple, l'interface `Device` définit l'attribut `location` de type `Location` (ligne 2). Les valeurs des attributs sont ensuite utilisées pour la découverte d'objets communicants (*e.g.*, pour découvrir un capteur de température dans une pièce particulière). En Pantaχou, les objets communicants peuvent interagir suivant trois modes d'interaction : les commandes, les événements et les sessions[4]. Les commandes sont déclarées en utilisant le mot clé **command**. Par exemple, l'interface `Sprinkler` fournit la commande `OnOff` (ligne 16). Une commande correspond à un ensemble de déclarations de méthodes (lignes 54 à 57). Contrairement à DiaSpec, les méthodes d'une commande peuvent retourner des valeurs. Les événements, quant à eux, sont déclarés en utilisant le mot clé **event**. Par exemple, l'interface `TemperatureSensor` produit des événements de type `Temperature` (ligne 12). Les événements ne sont pas nommés comme dans DiaSpec, rendant impossible la déclaration de plusieurs événements de même type mais de sémantique différente dans une même interface. Les capacités à capturer les informations de contexte d'une interface correspondent donc aux événements (mode *push*) et aux commandes dont les méthodes ne retournent pas **void** (mode *pull*), alors que les capacités à agir sur l'environnement d'une interface correspondent aux commandes dont les méthodes retournent **void**.

En Pantaχou, la couche de spécification et la couche d'architecture sont entrelacées. En effet, la déclaration d'une interface intègre des éléments architecturaux permettant de préciser si une interaction est requise ou fournie, ainsi que les interfaces avec lesquelles l'interaction est autorisée. Par exemple, l'interface `SmokeDetected` requiert des événements de type `Smoke` produits par des objets communicants de type `SmokeDetector`, et fournit des événements de type `SmokeDetected` aux objets communicants de type `FireState` (lignes 25 à 30). Cette

3. Pantaχou signifie *partout* en grec ancien.
4. Les sessions ne sont pas présentées dans ce document mais sont détaillées dans [Pal08].

```
1  service Device {
2   Location location;
3  }
4  service FireSensor extends Device {}
5  service FireActuator extends Device {}
6  service SmokeDetector extends FireSensor {
7    provides event<Smoke> to SmokeDetected;
8    provides command<GetSmoke> to SmokeDetected;
9  }
10 service TemperatureSensor extends FireSensor {
11   Accuracy accuracy;
12   provides event<Temperature> to AverageTemperature;
13   provides command<GetTemperature> to AverageTemperature;
14 }
15 service Sprinkler extends FireActuator {
16   provides command<OnOff> to FireController;
17 }
18 service FireDoor extends FireActuator {
19   provides command<Release> to FireController;
20 }
21 service Alarm extends FireActuator {
22   provides command<Activation> to FireController;
23 }

25 service SmokeDetected {
26   requires event<Smoke> from SmokeDetector;
27   requires command<GetSmoke> from SmokeDetector;
28   provides event<SmokeDetected> to FireState;
29   provides command<GetSmokeDetected> to FireState;
30 }
31 service AverageTemperature {
32   requires event<Temperature> from TemperatureSensor;
33   requires command<GetTemperature> from TemperatureSensor;
34   provides event<AverageTemperature> to FireState;
35   provides command<GetAverageTemperature> to FireState;
36 }
37 service FireState {
38   requires event<SmokeDetected> from SmokeDetected;
39   requires event<AverageTemperature> from AverageTemperature;
40   requires command<GetSmokeDetected> from SmokeDetected;
41   requires command<GetAverageTemperature> from AverageTemperature;
42   provides event<FireState> to FireController;
43 }
44 service FireController {
45   requires event<FireState> from FireState;
46   requires command<OnOff> from Sprinkler;
47   requires command<Release> from FireDoor;
48   requires command<Activation> from Alarm;
49 }

51 command GetSmokeDetected {
52   Boolean getSmokeDetected(Location location);
53 }
54 command OnOff {
55   void on();
56   void off();
57 }
```

FIGURE 7.6 – Extrait de la spécification des interfaces (Pantaχou)

couche est en fait directement inspirée des langages d'architecture logicielle traditionnels. En outre, contrairement à DiaSpec, les objets communicants ne sont pas explicitement distingués des applications du système d'informatique ubiquitaire (figure 4.3). Néanmoins, les interfaces qui fournissent uniquement des commandes ou des événements peuvent être considérées comme faisant partie de la couche de spécification des objets communicants (*e.g.*, l'interface `SmokeDetector`), alors que celles qui requièrent au moins un événement peuvent être considérées comme faisant partie de la couche d'architecture d'une application (*e.g.*, l'interface `SmokeDetected`). En Pantaχou, ces dernières sont appelées *services*. Cet entrelacement entre la couche de spécification et la couche d'architecture ne favorise donc pas la réutilisation des différentes descriptions, comme dans DiaSpec. Il est important de noter que DiaSpec a été développé après Pantaχou, ce qui lui a permis de profiter de l'expérience acquise lors de la conception de Pantaχou pour gagner en abstraction et en spécificité, afin d'offrir des concepts beaucoup plus dédiés à la description d'architecture de systèmes d'informatique ubiquitaire.

Couche d'implémentation

La couche d'implémentation de Pantaχou est dédiée à l'orchestration des objets communicants d'un système d'informatique ubiquitaire (*i.e.*, la logique applicative). Elle ne permet pas d'implémenter les interfaces des objets communicants qui nécessitent généralement l'utilisation de bibliothèques logicielles (*e.g.*, pour récupérer la valeur d'un capteur de température). Par conséquent, les interfaces des objets communicants sont implémentées dans un langage de programmation généraliste, ici en Java en utilisant le *framework* de programmation généré par DiaSpec.

La figure 7.7 représente un extrait d'une implémentation du service `SmokeDetected`[5]. Cette implémentation est une instance de l'interface `SmokeDetected` (ligne 1). De ce fait, ce service interagit avec des objets communicants de type `SmokeDetector` à partir desquels il reçoit des événements de type `Smoke`. Le développeur commence donc par définir quel sous-ensemble d'objets communicants de type `SmokeDetector` l'intéresse, en exprimant entre accolades les valeurs des attributs (ligne 3). Puis, il définit de la même façon quels événements de type `Smoke` produits par le sous-ensemble d'objets communicants l'intéressent (ligne 4). Dans ces deux définitions, aucun filtre n'est défini, signifiant que tous les événements de type `Smoke` produits par n'importe quel objet communicant de type `SmokeDetector` (indiqué par **[*]**) sont considérés. Ensuite, le développeur introduit un ou plusieurs gestionnaires (*handler*) pour traiter ces événements, en utilisant le mot clé **onReceive**. Par exemple, le gestionnaire `smokeReception` (lignes 5 à 24) publie un événement de type `SmokeDetected` (lignes 8, 17 et 22), indiquant si la pièce dans laquelle se trouve l'objet communicant qui a produit l'événement reçu est enfumée. L'opérateur **new** permet de créer de nouveaux événements, mais aussi de découvrir les objets communicants disponibles dans l'environnement (ligne 15). Puis, conformément à sa déclaration, le service `SmokeDetected` définit la méthode `getSmokeDetected` (ligne 25), fournissant une implémentation de la commande `GetSmokeDetected`. Enfin, le développeur définit un bloc **initial** (lignes 28 à 31), jouant le rôle d'un constructeur. Ce bloc utilise la méthode **adopt** pour désigner quel gestionnaire utiliser lors de la réception d'un événement de type `Smoke`, et crée la table de hachage `smokeDetectedHT` déclarée à la ligne 2. Cette table de hachage permet de stocker et d'associer à une localisation particulière la donnée calculée par le service `SmokeDetected`, permettant à d'autres services d'y avoir accès ultérieurement *via* la méthode

5. Pour des raisons de lisibilité, la gestion des exceptions a été omise.

```
1  'sip:mysmokedetected@enseirb.fr' instanciates SmokeDetected {
2    var HashTable SmokeDetectedHT;
3    service<SmokeDetector> { } smokeDetector;
4    event<Smoke> from smokeDetector[∗] { } smokeEvent;
5    onReceive smokeReception(smokeEvent e) {
6      var Location location = e.src.location;
7      if (e.data.isDetected) {
8        publish(new event<SmokeDetected>{bool=true; location=location;});
9        smokeDetectedHT.put(location, true);
10       return;
11     }
12     service<SmokeDetector> {
13       location = location;
14     } selectedSmokeDetector;
15     for (selectedSmokeDetector sd : new selectedSmokeDetector[∗]) {
16       if (sd.getSmoke().isDetected) {
17         publish(new event<SmokeDetected>{bool=true; location=location;});
18         smokeDetectedHT.put(location, true);
19         return;
20       }
21     }
22     publish(new event<SmokeDetected>{bool=false; location=location;});
23     smokeDetectedHT.put(location, false);
24   }
25   Boolean getSmokeDetected(Location location) {
26     return smokeDetectedHT.get(location);
27   }
28   initial {
29     adopt(smokeReception);
30     smokeDetectedHT = new HashTable<Location, Boolean>();
31   }
32 }
```

FIGURE 7.7 – Une implémentation du service `SmokeDetected` (Pantaχou)

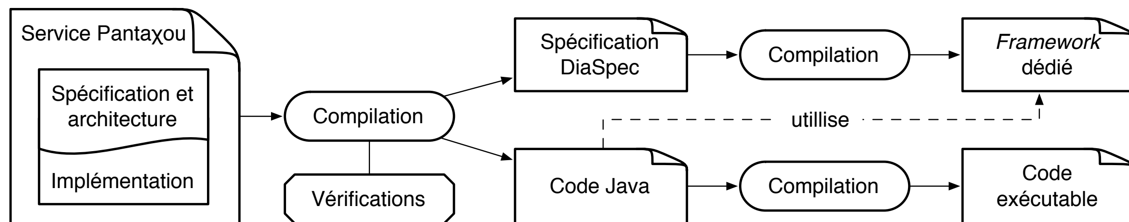

FIGURE 7.8 – Processus de compilation de Pantaχou

`getSmokeDetected` (mode *pull*). Le développeur doit donc explicitement gérer cette donnée calculée, ce qui n'est pas de l'orchestration d'objets communicants à proprement parler. Au contraire, dans DiaSpec, cette gestion est rendue transparente et est aussi facilitée par l'utilisation des index. Là encore, le *framework* de programmation généré par le compilateur de DiaSpec a profité de l'expérience acquise lors de la conception de Pantaχou.

7.4.2 Compilation

La figure 7.8 montre les principales étapes de la compilation d'un service Pantaχou. Tout d'abord, le service est passé au compilateur de Pantaχou qui réalise différentes vérifications [MPCL08], puis transforme d'une part les descriptions de la couche de spécification et d'architecture dans des spécifications DiaSpec, et d'autre part l'implémentation du service en Java. La spécification DiaSpec est ensuite passée au compilateur de DiaSpec qui génère un *framework* de programmation dédié. Il est important de noter que le code Java généré s'appuie sur ce *framework* de programmation. Enfin, le code Java est compilé pour produire le code exécutable. En examinant cette chaîne de compilation, nous pouvons noter qu'une partie de la complexité du compilateur de Pantaχou est déplacée dans le compilateur de DiaSpec. En effet, ce dernier est dédié aux aspects liés à la distribution des objets communicants, alors que le compilateur de Pantaχou est dédié à la traduction des abstractions du langage dans des motifs de code Java fondés sur le *framework* de programmation généré. En particulier, il s'agit d'exprimer les événements et les commandes dont les méthodes ne retournent pas **void** dans des sources de données DiaSpec (indexées par les paramètres des méthodes), et les commandes dont les méthodes retournent **void** dans des actions DiaSpec. Ainsi, de par la réduction du pas de compilation, le processus de compilation de Pantaχou devient plus simple.

7.4.3 Couplage avec un langage dédié de script

Pantaχou couple fortement une couche de spécification et d'architecture généraliste et un langage de script dédié à l'orchestration d'objets communicants. Bien que la couche de spécification et d'architecture de Pantaχou soit moins spécifique que DiaSpec, la nature dédiée de la couche d'implémentation permet un couplage fort. Grâce à ce couplage, l'implémentation d'un service est rigoureusement guidée par ses déclarations dans la couche de spécification et d'architecture, définissant quels objets communicants peuvent être découverts et comment interagir avec eux. Pour illustrer ce couplage, reprenons l'exemple du service `SmokeDetected` donné à la figure 7.7. Conformément à sa déclaration, son implémentation peut seulement interagir avec des objets communicants de type `SmokeDetector` *via* la commande `GetSmoke`

et *via* des événements de type `Smoke`, et publier des événements de type `SmokeDetected` (figure 7.6, lignes 25 à 30). Ces déclarations personnalisent les constructions du langage pour guider l'implémentation du service. Par exemple, considérons la construction de découverte d'objets communicants de la forme **service**<I>{...}. Lorsqu'un développeur implémente le service `SmokeDetected`, cette construction de découverte d'objets communicants peut seulement faire référence à l'interface `SmokeDetector`. De plus, elle est paramétrée par les attributs de cette interface (*i.e.*, l'attribut `location` hérité de l'interface `Device`). Ceci est illustré à la figure 7.7, lignes 12 à 14, où des objets communicants de type `SmokeDetector` localisés dans une pièce particulière sont recherchés. De même, la construction **event**<T>{...} peut seulement faire référence au type `Smoke` ou `SmokeDetected` pour respectivement recevoir (*e.g.*, ligne 4) ou publier (*e.g.*, ligne 8) des événements de ces types. Elle est également paramétrée par les attributs de ces types de données définis dans la couche de spécification et d'architecture. Ainsi, les méthodes **adopt** et **publish** sont statiquement typées par rapport aux différents types d'événements que le service peut respectivement recevoir et publier.

Ce couplage fort permet donc de guider et de faciliter la construction de systèmes d'informatique ubiquitaire. Il permet aussi à la découverte d'objets communicants et aux interactions d'être statiquement vérifiées par le compilateur de Pantaχou, augmentant la sûreté des systèmes d'informatique ubiquitaire. En outre, il vérifie *par construction*, et non *par génération* comme dans l'approche DiaSpec, que l'implémentation d'un système d'informatique ubiquitaire est conforme aux descriptions de la couche de spécification et d'architecture. Contrairement à un langage de programmation généraliste, Pantaχou possède des constructions syntaxiques dédiées et une sémantique simple qui permettent d'exprimer la logique d'orchestration à un haut niveau d'abstraction, rendant le développement d'un service encore plus simple et plus sûr.

7.5 Pantagruel, un langage visuel

Pantagruel [DMC09] couple fortement une couche de spécification dédié et un langage visuel dédié à l'orchestration d'objets communicants. La sémantique de Pantagruel a été formellement définie et un compilateur a été développé [6]. La définition formelle du langage [DMC09] sert de fondation pour la définition d'analyses de programme. Ces analyses concernent le langage mais aussi l'intention du développeur et la sémantique du programme dans son ensemble.

7.5.1 Définition du langage

Le langage Pantagruel se décompose en deux couches. La première couche permet de spécifier les types de données échangées et les interfaces des objets communicants pour un domaine d'application donné (*e.g.*, la domotique). La seconde couche permet d'orchestrer visuellement sous forme de règles les différents objets communicants d'un système d'informatique ubiquitaire (*e.g.*, un système de gestion des incendies), étant donnée une définition de types de données et d'interfaces. Pantagruel offre des constructions de haut niveau et dédiées dans ces deux couches. Une syntaxe complète de Pantagruel est disponible dans la thèse de Zoé Drey [Dre10].

6. Le compilateur de Pantagruel a été développé par Alexandre Blanquart, Ghislain Deffrasnes et Zoé Drey.

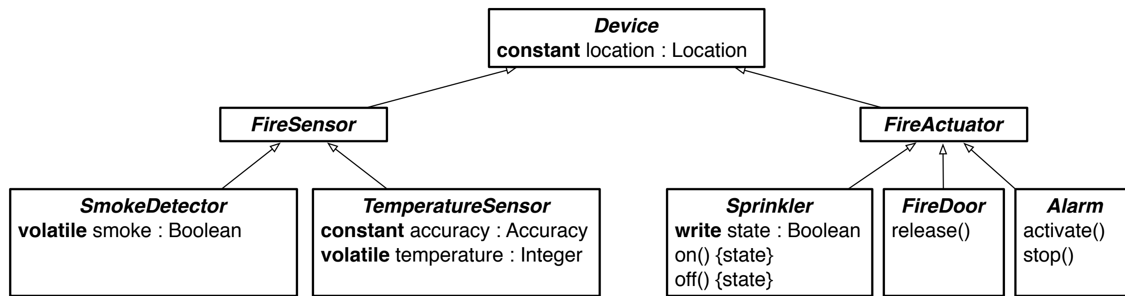

FIGURE 7.9 – Extrait de la spécification des interfaces (Pantagruel)

Couche de spécification

En Pantagruel, les types de données échangées peuvent être soit des types primitifs (*e.g.*, booléens, entiers ou encore chaînes de caractères), soit des types énumérés. Les interfaces, quant à elles, sont un ensemble d'attributs et de méthodes, comme illustré à la figure 7.9. Les attributs sont déclarés en utilisant trois mots clés : **constant**, **write** ou **volatile**. Les attributs **constant** sont des attributs dont la valeur ne change pas dans le temps. Par exemple, l'interface `Device` définit l'attribut `location` de type `Location`. Cet attribut indique que les objets communicants de type `Device` ont une localisation fixe. Les attributs **write** représentent des données dont la valeur est calculée et mise à jour par les règles d'orchestration. Par exemple, l'interface `Sprinkler` définit l'attribut `state` de type `Boolean`. Cet attribut est mis à jour lorsque les méthodes `on` et `off` sont invoquées. Les attributs **constant** et **write** permettent donc de décrire les propriétés d'une interface. Les attributs **volatile** représentent des données de l'environnement dont la valeur est susceptible de changer à n'importe quel moment et mise à jour périodiquement par les objets communicants. Par exemple, l'interface `TemperatureSensor` définit l'attribut `temperature` de type `Integer`. Cet attribut est mis à jour par les objets communicants de type `TemperatureSensor` lorsque la température ambiante change. Les attributs, quels qu'ils soient, sont ensuite utilisés par le développeur pour exprimer les conditions d'activation d'une règle d'orchestration. Les méthodes, quant à elles, sont similaires à celles de DiaSpec, dans le sens où elles n'ont pas de type de retour. Par exemple, l'interface `Sprinkler` définit les méthodes `on` et `off`, permettant respectivement de déclencher et d'arrêter les sprinklers. En Pantagruel, les méthodes peuvent être associées à des attributs **write** sur lesquels elles ont un effet de bord. Par exemple, les méthodes `on` et `off` déclarent entre accolades l'attribut `state` qu'elles devraient mettre à jour pour indiquer si le sprinkler est en activité. Les capacités à capturer les informations de contexte d'une interface correspondent donc aux attributs **volatile**, alors que les capacités à agir sur l'environnement d'une interface correspondent aux méthodes et aux attributs **write**.

Une fois les types de données et les interfaces définis, il est possible de spécifier certaines énumérations et certains objets communicants par rapport au système d'informatique ubiquitaire à développer. La figure 7.10 représente un extrait de la spécification des énumérations et des objets communicants pour notre système de gestion des incendies. Par exemple, le type énuméré `Location` liste l'ensemble des pièces du bâtiment. Chaque objet communicant a un nom unique et implémente une ou plusieurs interfaces. Par exemple, l'objet communi-

sd11 : *SmokeDetector*
location = ROOM1

sd12 : *SmokeDetector*
location = ROOM1

sd2 : *SmokeDetector*
location = ROOM2

sd3 : *SmokeDetector*
location = ROOM3

alarm : *Alarm*
location = ROOM1

ts1 : *TemperatureSensor*
accuracy = HIGH
location = ROOM1

ts2 : *TemperatureSensor*
accuracy = HIGH
location = ROOM2

ts3 : *TemperatureSensor*
accuracy = NORMAL
location = ROOM3

s11 : *Sprinkler*
location = ROOM1

s12 : *Sprinkler*
location = ROOM1

s2 : *Sprinkler*
location = ROOM2

s3 : *Sprinkler, Alarm*
location = ROOM3

fd1 : *FireDoor*
location = ROOM1

fd12 : *FireDoor*
location = ROOM1

fd2 : *FireDoor*
location = ROOM2

fd3 : *FireDoor*
location = ROOM3

Location {ROOM1, ROOM2, ROOM3, ...}
Accuracy {LOW, NORMAL, HIGH}

FIGURE 7.10 – Extrait de la spécification des énumérations et des objet communicants du système de gestion des incendies (Pantagruel)

cant `sd11` est de type `SmokeDetector`. Les attributs **constant** sont initialisés, alors que les attributs **volatile** et **write** sont initialement indéfinis. Bien évidemment, d'autres objets communicants peuvent être dynamiquement ajoutés au système et pris en compte par la couche d'implémentation.

Couche d'implémentation

À l'instar de Pantaχou, la couche d'implémentation de Pantagruel est dédiée à l'orchestration des objets communicants d'un système d'informatique ubiquitaire. Elle fournit un langage de règles, fondé sur le paradigme *capteur-contrôleur-actionneur*. Ce paradigme décompose une règle d'orchestration en capteurs, contrôleur et actionneurs. Les capteurs représentent les conditions d'activation de la règle. Le contrôleur spécifie comment les capteurs et les actionneurs sont respectivement combinés. Les actionneurs déterminent les actions à invoquer sur les objets communicants. Ainsi, le paradigme capteur-contrôleur-actionneur convient particulièrement bien aux programmeurs novices, comme en témoigne son utilisation dans différents domaines d'application tels que le jeu vidéo [vG03] ou encore la robotique [PJ97, GIL$^+$95].

En Pantagruel, le paradigme capteur-contrôleur-actionneur est représenté sous la forme d'un tableau divisé en trois colonnes : **sensors**, **controllers** et **actuators**. Ce tableau est illustré à la figure 7.11. Pour structurer visuellement encore davantage la définition d'une règle d'orchestration, le tableau est divisé en sections horizontales, faisant référence soit à un objet communicant (*e.g.*, `alarm` de type `Alarm`), soit à une interface (*e.g.*, `SmokeDetector`) du système d'informatique ubiquitaire. Les sections faisant référence à une interface permettent d'orchestrer facilement tous les objets communicants implémentant cette interface, suivant le patron de conception Composite [GHJV95]. À l'intérieur d'une section, la portée des capteurs et des actionneurs se limite à l'interface correspondante. Toutefois, il est possible d'accéder explicitement à un attribut d'un autre objet communicant ou d'une autre interface en utilisant le mot clé **of**. Par exemple, considérons la règle `R1` qui orchestre cinq sections. Cette règle libère les portes coupe-feu et déclenche les sprinklers et l'alarme du bâtiment lorsqu'un incendie est détecté. Pour cela, elle combine quatre capteurs avec le contrôleur **AND** [7] (*i.e.*, la conjonc-

7. Pantagruel fournit également le contrôleur **OR** (*i.e.*, la disjonction logique).

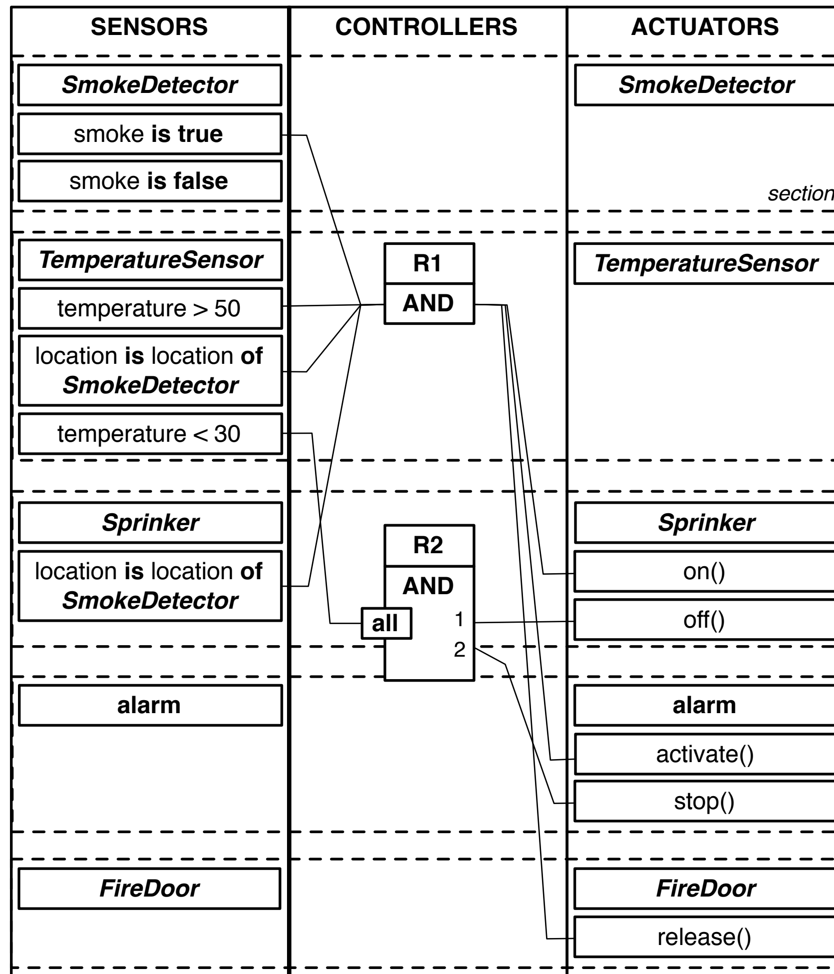

FIGURE 7.11 – Extrait des règles d'orchestration du système de gestion des incendies (Pantagruel)

tion logique). Les capteurs correspondent à des prédicats exprimés à partir des attributs des interfaces des objets communicants déclarés dans la couche de spécification. Pour tester les valeurs des attributs, Pantagruel fournit des opérateurs de comparaison (*e.g.*, **is**, $<$, $>$), des opérateurs ensemblistes (*e.g.*, **in**, **has**) et l'opérateur unaire **changed** qui permet de tester si la valeur d'un attribut a changé. Les capteurs à l'intérieur d'une section faisant référence à une interface agissent comme un filtre pour collecter l'ensemble des objets communicants satisfaisant ces capteurs. Ainsi, la règle `R1` collecte l'ensemble des objets communicants de type `SmokeDetector` qui ont détecté la présence de fumée (*i.e.*, `smoke` **is true**), l'ensemble des objets communicants de type `TemperatureSensor` localisés dans la même pièce qu'un des objets communicants de type `SmokeDetector` collectés (*i.e.*, `location` **is** `location` **of** `Smoke-Detector`) et qui mesurent une température ambiante supérieure à 50°C (*i.e.*, `temperature` $>$ 50), et l'ensemble des objets communicants de type `Sprinkler` localisés dans la même pièce qu'un des objets de type `SmokeDetector` collectés (*i.e.*, `location` **is** `location` **of** `Smoke-`

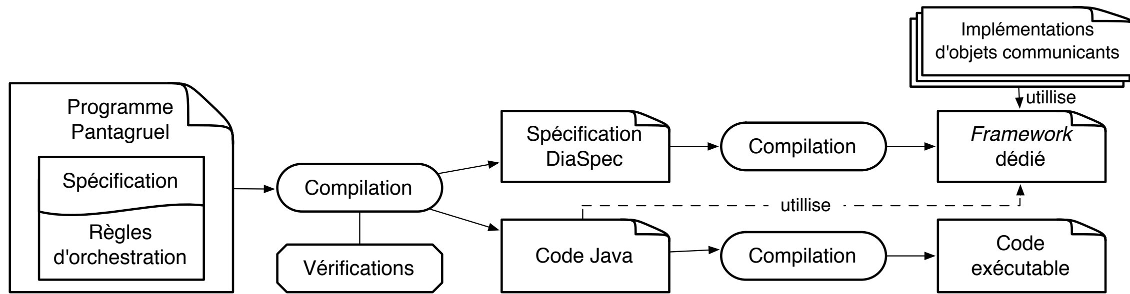

Figure 7.12 – Processus de compilation de Pantagruel

`Detector`). Lorsque toutes ces conditions sont satisfaites, la règle `R1` déclenche les actionneurs correspondants (*i.e.*, `on()`, `activate()` et `release()`) sur les objets communicants collectés. Les actionneurs peuvent correspondre soit à une mise à jour d'attributs **write**, soit à un appel de méthode, déclarés dans la couche de spécification. Une règle d'orchestration peut déclencher plusieurs actionneurs sans tenir compte de l'ordre (*e.g.*, la règle `R1`) ou dans un ordre prédéfini (*e.g.*, la règle `R2`). Enfin, dans une section faisant référence à une interface, une condition peut nécessiter d'être satisfaite pour tous les objets communicants correspondants. Une telle condition est exprimée en connectant le capteur au contrôleur *via* l'opérateur **all**, comme illustré dans la règle `R2`. Cette règle stoppe les sprinklers, puis l'alarme dès que tous les capteurs de température mesurent une température ambiante inférieure à 30°C.

7.5.2 Compilation

La figure 7.12 montre les principales étapes de la compilation d'un programme Pantagruel. Tout d'abord le programme Pantagruel est passé au compilateur de Pantagruel qui réalise différentes vérifications [DMC09], puis transforme d'une part les descriptions de la couche de spécification dans des spécifications DiaSpec, et d'autre part les règles d'orchestration en Java. La spécification DiaSpec est ensuite passée au compilateur de DiaSpec qui génère un *framework* de programmation dédié. Il est important de noter que le code Java généré s'appuie sur ce *framework* de programmation. De plus, des implémentations des objets communicants définis dans la couche de spécification doivent être fournies. Afin de respecter la sémantique d'exécution des règles d'orchestration, le code Java généré s'organise en deux parties. La première partie est chargée de souscrire à toutes les sources de données nécessaires et de stocker les événements reçus (*i.e.*, les valeurs des attributs **volatile** et **write**) dans une table de hachage. La seconde partie correspond à un *thread* qui exécute une boucle infinie dans laquelle toutes les règles d'orchestration sont évaluées et exécutées avec une copie courante de la table de hachage, chaque fois que la valeur d'un attribut **volatile** ou **write** change. Enfin, le code Java est compilé pour produire le code exécutable. À l'instar du compilateur de Pantaχou, une partie de la complexité du compilateur de Pantagruel est déplacée dans le compilateur de DiaSpec. En effet, le compilateur de Pantagruel est dédié à la traduction des abstractions du langage dans des motifs de code Java fondés sur le *framework* de programmation généré. En particulier, il s'agit d'exprimer les attributs **volatile** dans des sources de données DiaSpec. Ainsi, lorsqu'un objet communicant produit un événement, l'attribut **volatile** correspondant

est mis à jour. Les attributs **write**, quant à eux, sont transformés à la fois dans un attribut et une action DiaSpec. Cette dernière permet de mettre à jour l'attribut. À chaque mise à jour, un événement indiquant sa nouvelle valeur doit être publié. Enfin, les méthodes sont traduites dans des actions DiaSpec. Ainsi, de par la réduction du pas de compilation, le processus de compilation de Pantagruel est plus simple.

7.5.3 Couplage avec un langage dédié visuel

Pantagruel couple fortement une couche de spécification et un langage de règles visuel dédié à l'orchestration d'objets communicants. Grâce à ce couplage, la définition des règles d'orchestration est guidée par les descriptions de la couche de spécification, fournissant au développeur des menus contextuels et des vérifications à la volée. Pour illustrer ce couplage, reprenons les règles d'orchestration données à la figure 7.11. Lorsqu'un développeur construit une règle d'orchestration, il commence par sélectionner et placer dans le tableau les interfaces et les objets communicants déclarés dans la couche de spécification qui sont impliqués dans l'interaction. Ensuite, il définit certaines conditions dans la colonne **sensors**, les combine dans la colonne **controllers**, et détermine les actions à déclencher dans la colonne **actuators**. Les capteurs et les actionneurs sont personnalisés par rapport aux informations définies dans la couche de spécification. Par exemple, les capteurs de la section `SmokeDetector` permettent seulement de tester la valeur des attributs `location` et `smoke`. De plus, d'après leur type, ces attributs ont un ensemble fini de valeurs possibles. Par exemple, l'attribut `location` peut être testé avec chacune des constantes du type énuméré `Location` et avec les attributs `location` des autres objets communicants placés dans le tableau. Il est alors possible de présenter au développeur ces différents choix dans une liste déroulante lors de la définition du capteur.

Ce couplage fort permet donc de guider et de faciliter la construction de systèmes d'informatique ubiquitaire. De plus, il vérifie *par construction* que l'implémentation d'un système d'informatique ubiquitaire est conforme aux descriptions de la couche de spécification. Enfin, la nature haut niveau et dédiée de Pantagruel permet non seulement d'élever le niveau d'abstraction de la programmation, mais aussi de faciliter l'expression et la vérification de propriétés spécifiques au domaine, augmentant la sûreté des systèmes d'informatique ubiquitaire. La programmation de ces systèmes devient alors accessible à des non programmeurs.

7.6 Bilan

Le couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation, ainsi que la nature dédiée des couches, permettent de faciliter la construction et la vérification d'un système d'informatique ubiquitaire. Grâce à ce couplage, l'implémentation d'un tel système est rigoureusement dirigée par les descriptions de la couche de spécification et d'architecture, permettant aux développeurs de se concentrer sur la logique applicative. À partir de ces descriptions, la couche d'implémentation peut être personnalisée, offrant aux développeurs des constructions et des abstractions de programmation de haut niveau et dédiées pour guider et supporter le développement logiciel. Ce couplage permet également d'assurer automatiquement et systématiquement que l'implémentation du système est conforme aux descriptions de la couche de spécification et d'architecture, ce qui facilite grandement la vérification de propriétés (*e.g.*, l'intégrité des communications). Nous avons validé notre approche avec un langage de programmation généraliste, Java, et deux langages dédiés que nous avons (co)développés, Pantaχou, un langage de script, et Pantagruel, un langage visuel.

Chapitre 8

Gestion des erreurs

Dans le chapitre précédent, nous avons montré comment un couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation permettait de rendre plus simple et plus sûr le développement de systèmes d'informatique ubiquitaire. Jusqu'à présent, les couches de spécification et d'architecture décrivaient des aspects fonctionnels du système (*e.g.*, les interactions entre les composants). Dans ce chapitre, nous nous intéressons à la description d'un aspect non fonctionnel, la gestion des erreurs. En particulier, nous montrons comment notre approche permet de guider et supporter la gestion des erreurs dans un système d'informatique ubiquitaire. Après avoir résumé la problématique relative à la gestion des erreurs dans un système d'informatique ubiquitaire, nous introduisons un modèle original de gestion d'erreurs. Nous décrivons ensuite l'implémentation du modèle. Enfin, nous présentons divers travaux connexes à notre modèle de gestion d'erreurs.

8.1 Problématique

Les systèmes d'informatique ubiquitaire orchestrent une large gamme d'objets communicants, à la fois matériels et logiciels, communiquent en utilisant une variété de protocoles réseaux, s'appuient sur des technologies de systèmes distribués compliquées et invoquent une multitude d'API complexes. Par conséquent, un système d'informatique ubiquitaire doit gérer un large spectre d'erreurs afférentes à divers niveaux, incluant l'infrastructure physique (*e.g.*, coupure de courant), le matériel (*e.g.*, dysfonctionnement d'un périphérique), le système d'exploitation (*e.g.*, épuisement de ressources), le réseau (*e.g.*, perte de signal), l'intergiciel (*e.g.*, échec d'une invocation à distance) et les API (*e.g.*, données d'entrée incorrectes). Détecter et recouvrer de telles erreurs est donc essentiel pour rendre robuste un système d'informatique ubiquitaire. Pour cela, une approche systématique et rigoureuse pour gérer les erreurs est nécessaire.

Beaucoup de progrès ont été réalisés afin d'abstraire la complexité et la diversité des technologies existantes, mais pas des erreurs. Les erreurs, quelles qu'elles soient, nécessitent d'être propagées pour déclencher des traitements spécifiques à l'application. Un traitement systématique des erreurs se fait au prix de «polluer» l'ensemble du code de l'application avec du code de gestion d'erreurs. Typiquement, un bloc **try-catch** est introduit chaque fois que le développeur fait appel à une opération (ou un groupe d'opérations) qui peut échouer. Le code résultant est alors alourdi et entrelacé, comme démontré dans l'étude de Lippert et Lopes [LL00]. Même si les erreurs sont systématiquement traitées, le code de gestion d'erreurs est souvent *ad hoc*

et local, empêchant un raisonnement sur l'ensemble du système. Du fait d'un manque de support de programmation de haut niveau, écrire le code de gestion d'erreurs est une tâche ardue, négligée par la plupart des développeurs. Cette situation conduit à l'implémentation de stratégies de gestion d'erreurs triviales (*e.g.*, garder une trace de l'erreur et l'ignorer ou encore signaler une erreur différente), allant à l'encontre du but de ces erreurs.

Une caractéristique inhérente au bloc **try-catch** est d'aborder le recouvrement d'erreurs à deux niveaux différents : localement, en permettant de reprendre l'exécution de l'application lorsque c'est possible, et globalement en prenant des mesures pour préserver la cohérence globale du système. Dans le premier cas, l'exception n'est pas propagée et, si une valeur est manquante, elle peut être remplacée par une valeur par défaut. Cette gestion des erreurs est dite *au niveau de l'application*. Sinon, le développeur peut choisir une gestion des erreurs *au niveau du système*. Dans ce cas, le calcul erroné est interrompu et le contrôle est transféré à un gestionnaire de recouvrement d'erreurs qui traite l'erreur d'un point de vue du système. Un tel gestionnaire peut par exemple remplacer un capteur défectueux par un capteur de secours. Le conflit entre la gestion des erreurs au niveau de l'application et au niveau du système est exacerbée dans un système d'informatique ubiquitaire en raison du grand nombre d'objets communicants qui interagissent et de la multitude d'erreurs potentielles auxquelles il doit faire face. Il devient donc indispensable d'élever le niveau d'abstraction de la gestion des erreurs au-delà du code et du bloc **try-catch**.

8.2 Modèle de gestion d'erreurs

Nous présentons un modèle original de gestion d'erreurs dans un système d'informatique ubiquitaire [MECL10]. Ce modèle aborde la gestion des erreurs au niveau de l'architecture logicielle. Dans ce modèle, une même erreur est traitée à la fois au niveau de l'application et au niveau du système, lui permettant d'être compensée localement et gérée globalement par des stratégies de réparation au niveau de la plate-forme. Ce modèle est introduit dans l'approche DiaSpec (section 7.3) et est illustré par notre système de gestion des incendies (section 7.2). Tout d'abord, nous décrivons comment les erreurs sont caractérisées dans la couche de spécification de DiaSpec. Puis, nous expliquons comment étendre la couche d'architecture de DiaSpec pour architecturer la gestion d'erreurs, à la fois au niveau de l'application et au niveau du système. Enfin, nous examinons le support de programmation généré pour faciliter l'implémentation du code de gestion d'erreurs.

8.2.1 Spécifier les erreurs

Dans un système d'informatique ubiquitaire, les erreurs se produisent typiquement lors des interactions avec les objets communicants. Nous avons donc étendu la couche de spécification de DiaSpec (section 7.3.1) avec de nouvelles déclarations pour spécifier si une source de données ou une méthode est susceptible de lever une exception. Les exceptions sont levées en utilisant le mot clé **raises**, comme illustré à la figure 8.1. Par exemple, une exception de type `MeasureException` est levée par les objets communicants de type `TemperatureSensor` lorsqu'une erreur survient lors de la mesure de la température ambiante (ligne 3). De même, une exception de type `WaterPressureException` est levée par les objets communicants de type `Sprinkler` lorsqu'une erreur survient lors de l'aspersion (ligne 10).

D'autres exceptions qui ne sont pas directement liées aux capacités à capturer les informations de contexte ou à agir sur l'environnement des objets communicants, peuvent être levées

```
1  device TemperatureSensor extends FireSensor {
2    attribute accuracy as Accuracy;
3    source temperature as Temperature raises MeasureException;
4  }
5  device Sprinkler extends FireActuator {
6    action OnOff;
7  }

9  action OnOff {
10   on() raises WaterPressureException;
11   off();
12 }
```

FIGURE 8.1 – Extrait de la spécification des interfaces avec des déclarations de gestion d'erreurs

par l'environnement d'exécution lors d'un accès à une source de données ou lors d'un appel de méthode. Ces exceptions permettent de signaler des erreurs provenant de la plate-forme d'informatique ubiquitaire. Par exemple, lorsqu'un objet communicant devient indisponible à cause d'une panne réseau, une exception de type `NetworkException` est levée. Un autre exemple est l'exception de type `UnregisterException` qui est levée lorsque le bail d'un objet communicant a expiré. De telles exceptions sont dites *built-in* et forment une hiérarchie illustrée à la figure 8.2.

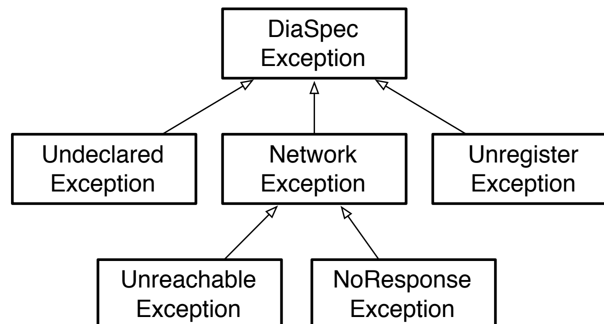

FIGURE 8.2 – Extrait de la hiérarchie des exceptions *built-in*

Notre modèle de gestion d'erreurs permet de signaler une même erreur à la fois au niveau de l'application et au niveau du système (figure 4.3). Au niveau de l'application, le code de gestion des erreurs s'occupe uniquement de compenser l'erreur afin de poursuivre, plus ou moins normalement, l'exécution du programme, et non de réparer la cause du problème (*e.g.*, le fait qu'un capteur soit défectueux). Ceci est réalisé au niveau du système où le code de gestion des erreurs s'occupe d'analyser la cause de l'erreur et de déterminer les stratégies de réparation qui préservent la cohérence globale du système. D'un point de vue conceptuel, le code de gestion d'erreurs au niveau de l'application et celui au niveau du système sont exécutés en parallèle, lorsque les deux niveaux sont touchés par une même erreur. Par ailleurs, lorsqu'une erreur est détectée de manière proactive (*e.g.*, expiration du bail d'un objet communicant), elle

peut être prise en charge par un gestionnaire traitant les erreurs au niveau du système, avant qu'elle n'affecte les applications. Ainsi, il est possible de réaliser un recouvrement d'erreurs qui est transparent aux applications.

8.2.2 Architecturer la gestion des erreurs

En s'intéressant à la gestion des erreurs dès l'architecture, notre approche permet d'abstraire le large spectre d'erreurs pouvant se produire dans un système d'informatique ubiquitaire, et fournit des informations pour guider et supporter l'implémentation du code de gestion d'erreurs. Nous avons étendu la couche d'architecture de DiaSpec (section 7.3.1) avec de nouvelles déclarations pour architecturer les erreurs à la fois au niveau de l'application et au niveau du système. Il est important de noter que notre modèle peut tout à fait s'appliquer à d'autres langage de description d'architecture fondés sur le même style architectural que DiaSpec (figure 7.3).

Au niveau de l'application

Architecturer la gestion des erreurs au niveau de l'application consiste à spécifier l'impact d'une exception sur le flot de contrôle des composants de l'architecture. Autrement dit, il s'agit de définir comment les erreurs sont gérées par les composants de l'architecture. Plus précisément, l'architecte détermine dès la conception du système si le traitement d'une exception dans un composant donné est (1) obligatoire, permettant de poursuivre l'exécution de manière transparente, (2) facultatif, similaire au traitement d'une exception non vérifiée (*unchecked*) en Java, (3) délégué au composant appelant, lui propageant automatiquement l'exception, ou (4) inutile car aucune exception ne peut arriver. Ces différentes politiques sont respectivement exprimées en utilisant les mots clés **mandatory catch**, **optional catch**, **skipped catch** et **no catch**, comme illustré à la figure 8.3.

Ces déclarations de gestion d'erreurs permettent à l'architecte d'exiger qu'une erreur soit traitée différemment en fonction de son caractère critique dans le système d'informatique ubiquitaire. Par exemple, dans cette description, l'architecte considère l'opérateur de contexte `FireState` comme l'élément central pour déterminer une situation d'incendie dans le bâtiment. Par conséquent, la gestion des erreurs provenant des objets communicants de type `SmokeDetector` n'est pas déléguée à l'opérateur de contexte `SmokeDetected`. Cette décision architecturale est exprimée à la ligne 3, où l'opérateur de contexte `SmokeDetected` est déclaré comme n'étant pas autorisé à traiter les erreurs provenant des objets communicants de type `SmokeDetector`. Cette tâche est assignée à l'opérateur de contexte `FireState`, où le traitement des erreurs est déclaré comme étant obligatoire (ligne 11).

En revanche, l'opérateur de contexte `AverageTemperature` est utilisé à différentes fins, incluant la régulation du chauffage. Son rôle est donc secondaire dans le système de gestion des incendies, puisqu'il est seulement utilisé pour corroborer les informations fournies par l'opérateur de contexte `SmokeDetected`. De ce fait, l'opérateur de contexte `AverageTemperature` est déclaré comme celui qui compense les erreurs provenant des objets communicants de type `TemperatureSensor` (ligne 7), fournissant toujours une valeur de température, même au détriment de l'exactitude de l'information. Parce que toutes les erreurs provenant des objets communicants de type `TemperatureSensor` sont nécessairement traitées par l'opérateur de contexte `AverageTemperature`, l'opérateur de contexte `FireState` n'a plus à se soucier de la fiabilité de cet opérateur, comme indiqué à la ligne 12.

```
1  context SmokeDetected as Boolean
2    indexed by location as Location {
3    source smoke from SmokeDetector [skipped catch];
4  }
5  context AverageTemperature as Temperature
6    indexed by location as Location {
7    source temperature from TemperatureSensor [mandatory catch];
8  }
9  context FireState as Boolean
10   indexed by location as Location {
11   context SmokeDetected [mandatory catch];
12   context AverageTemperature [no catch];
13 }

15 controller FireController {
16   context FireState [no catch];
17   action Release on FireDoor [optional catch];
18   action OnOff on Sprinkler [optional catch];
19   action Activation on Alarm [mandatory catch];
20 }
```

FIGURE 8.3 – Extrait de l'architecture du système de gestion des incendies avec des déclarations de gestion d'erreurs (niveau application)

Dans les opérateurs de contrôle, les déclarations de gestion d'erreurs permettent d'exprimer des décisions architecturales sur ce que l'implémentation de l'opérateur devrait faire au cas où l'invocation d'une action échouerait. La déclaration **skipped catch** est utilisée lorsque l'échec d'une action ne peut pas être compensé au niveau de l'application, nécessitant d'interrompre l'exécution du code. L'exception peut alors être ignorée par les développeurs. Au contraire, la déclaration **mandatory catch** est utilisée lorsque des actions alternatives pour compenser l'échec d'une action peuvent être choisies au niveau de l'application. Le développeur doit alors systématiquement implémenter un code de gestion d'erreurs (ligne 19). Enfin, la déclaration **optional catch** est utilisée lorsqu'aucune décision ne peut ou ne veut être prise par l'architecte concernant la gestion de l'échec d'une action, déléguant cette décision aux développeurs (lignes 17 et 18). Il est important de noter que la déclaration **no catch** peut être inférée : lorsqu'un composant déclare traiter toutes les erreurs susceptibles de se produire dans son implémentation (ligne 7), les interactions avec ce composant peuvent automatiquement être déclarées en utilisant la déclaration **no catch** (ligne 12). De plus, une analyse est effectuée pour vérifier la cohérence des déclarations de gestion d'erreurs.

Au niveau du système

Architecturer les erreurs au niveau du système consiste à définir des opérateurs de contexte et de contrôle qui traitent les événements signalant des erreurs. De tels événements sont appelés *événements exceptionnels*[1]. Ils ont le même type que les exceptions Java dont ils sont issus, et fournissent toutes les informations sur les causes de l'erreur. Les opérateurs de contexte

1. Cette distinction est purement conceptuelle car les événements exceptionnels sont implémentés comme des événements DiaSpec, comme détaillé à la section 8.3.

```
1  context FireDetectionFailure as Boolean
2    indexed by location as Location {
3    exception MeasureException from TemperatureSensor;
4    exception PowerException from FireSensor;
5    exception NeworkException from FireSensor;
6    exception UnregisterException from FireSensor;
7  }
8  context FireExtinctionFailure as Boolean
9    indexed by location as Location {
10   exception WaterPressureException from Sprinkler;
11   exception BatteryException from Sprinkler;
12   exception PowerException from FireActuator;
13   exception NeworkException from FireActuator;
14   exception UnregisterException from FireActuator;
15 }
16
17 controller EmergencyController {
18   context FireExtinctionFailure;
19   context OpeningHours;
20   action Activation from Alarm;
21   ...
22 }
23 controller DetectionController {
24   context FireDetectionFailure;
25   ...
26 }
```

Figure 8.4 – Extrait de l'architecture de gestion d'erreurs du système de gestion des incendies (niveau système)

traitant les erreurs au niveau du système diffèrent des opérateurs de contexte de l'architecture fonctionnelle dans la mesure où ils prennent en entrée des événements exceptionnels. Ils traitent et raffinent ces événements dans des valeurs spécifiques à l'application. Les opérateurs de contrôle reçoivent ensuite ces valeurs raffinées et exécutent une stratégie de réparation en invoquant différentes actions. Ainsi, les stratégies de réparation au niveau du système sont rendues explicites dans l'architecture, décomposées en opérateurs de contexte et de contrôle. Cette approche permet de raisonner sur la gestion des erreurs au niveau du système d'informatique ubiquitaire.

L'architecture de gestion d'erreurs de notre système de gestion des incendies est illustrée à la figure 8.4. Les opérateurs de contexte de cette architecture sont dédiés à la gestion des erreurs d'un point de vue du système. Par exemple, l'opérateur de contexte `FireDetection-Failure` (lignes 1 à 7) est dédié à la gestion des erreurs provenant des dispositifs de détection d'incendie (*e.g.*, des capteurs de température) et l'opérateur de contexte `FireExtinction-Failure` (lignes 8 à 15) à celles provenant des dispositifs d'extinction d'incendie (*e.g.*, des sprinklers). Ces erreurs peuvent être signalées par des exceptions définies dans la couche de spécification (*e.g.*, `WaterPressureException`, ligne 10) ou des exceptions *built-in* (*e.g.*, `NetworkException`, ligne 13), déclarées en utilisant le mot clé **exception**. Les opérateurs de contexte `FireDetectionFailure` et `FireExtinctionFailure` analysent les erreurs reçues et publient un événement de type `Boolean` lorsque des mesures doivent être prises. Par exemple, les occupants du bâtiment peuvent nécessiter d'être évacués si le nombre de dispositifs d'ex-

tinction d'incendie défectueux devient trop important. Pour cela, l'implémentation de l'opérateur de contexte `FireExtinctionFailure` doit, entre autres, garder une trace des erreurs provenant des dispositifs d'extinction d'incendie. Lorsque le nombre d'erreurs atteint un certain seuil, il publie un événement d'alerte. Cette information est transmise à l'opérateur de contrôle `EmergencyController` (ligne 18). Cet opérateur est chargé de déclencher l'alarme du bâtiment (ligne 20), si l'événement d'alerte a été reçu pendant les heures d'ouverture du bâtiment (ligne 19). Il est intéressant de noter qu'un opérateur de contexte ou de contrôle de l'architecture de gestion d'erreurs peut traiter à la fois des événements exceptionnels et des événements ordinaires produits par des composants de l'architecture fonctionnelle ou non. Ceci est illustré par l'opérateur de contrôle `EmergencyController` dont la logique dépend de l'opérateur de contexte `OpeningHours` (ligne 19) déclaré dans l'architecture fonctionnelle.

De manière similaire, l'opérateur de contexte `FireDetectionFailure` détermine si toutes les pièces du bâtiment possèdent des dispositifs de détection d'incendie opérationnels en traitant les événements exceptionnels provenant des objets communicants de type `FireSensor` (lignes 1 à 7). Dans cet opérateur, certaines déclarations de gestion d'erreurs tirent profit de la hiérarchie des interfaces des objets communicants (*e.g.*, `FireSensor`). Ainsi, un gestionnaire d'erreurs peut être défini à un niveau de granularité approprié et permettre de centraliser le traitement d'un type d'erreurs pour un type d'objets communicants donné.

8.2.3 Implémenter la gestion des erreurs

À partir de descriptions DiaSpec, un *framework* de programmation dédié est généré (section 7.3.2). Nous avons étendu le compilateur [2] de DiaSpec pour prendre en charge les déclarations de gestion des erreurs introduites dans la couche de spécification et d'architecture. Ainsi, le compilateur étendu de DiaSpec génère un *framework* de programmation pour faciliter et guider l'implémentation du code de gestion d'erreurs, à la fois au niveau de l'application et au niveau du système.

Au niveau de l'application

Au niveau de l'application, la gestion des erreurs consiste à compenser, soit la valeur manquante d'une source de données, soit l'échec d'une action, sans considérer la nature de l'erreur. Cette gestion d'erreurs est déclenchée par une unique exception de type `ApplicationLevelException`. Cette exception ne comporte aucune information sur les causes de l'erreur. Elle reste néanmoins une exception classique qui peut être propagée dans la pile d'appel.

Pour traiter l'exception de type `ApplicationLevelException`, nous proposons d'utiliser des continuations. Lorsqu'un composant de l'architecture déclare traiter les erreurs d'une interaction (*i.e.*, **mandatory catch**), le développeur doit passer une continuation comme paramètre supplémentaire à cette interaction. En revanche, lorsqu'un composant déclare ne pas traiter les erreurs d'une interaction (*i.e.*, **skipped catch** ou **no catch**), aucune continuation n'est requise. Cette manière de procéder oblige le développeur à fournir une continuation à chaque interaction qu'il utilise. Ceci contraste avec la construction **try-catch** qui peut couvrir un groupe d'opérations.

Parce que le *framework* de programmation est généré dans le respect des descriptions architecturales, les interfaces des objets communicants fournies aux développeurs sont conformes

2. La version étendue du compilateur de DiaSpec a été développée par Quentin Enard.

```
1  public class TemperatureSensorProxy {
2    [...]
3    // only generated if declared mandatory or optional
4    public Temperature getTemperature(TemperatureContinuation tc) {
5      [...]  // code generated by the DiaSpec compiler
6    }
7    public interface TemperatureContinuation {
8      public Temperature onError();
9    }
10 }
```

FIGURE 8.5 – Extrait de la classe *proxy* `TemperatureSensorProxy` générée pour l'opérateur de contexte `AverageTemperature`

à leurs déclarations de gestion d'erreurs. La figure 8.5 représente un extrait de la classe *proxy*[3] `TemperatureSensorProxy` générée pour l'opérateur de contexte `AverageTemperature`. Cette classe fournit une méthode `getTemperature` (ligne 4) qui permet d'accéder à la source de données `temperature` fournie par les objets communicants de type `TemperatureSensor`. Parce que cette source a été déclarée en utilisant la déclaration **mandatory catch** dans l'opérateur de contexte `AverageTemperature`, la classe *proxy* générée demande au développeur de compenser une valeur manquante en cas d'échec. Pour cela, une continuation doit être passée en paramètre de la méthode `getTemperature`. Elle correspond au code de recouvrement à exécuter en cas d'échec. Puisque le langage Java ne supporte pas les fermetures (*closures*), ces dernières sont simulées par des interfaces générées (lignes 7 à 9) déclarant une méthode `onError` dont le type de retour est le même que celui de la méthode surchargée (*i.e.*, `Temperature`). La figure 8.6 présente des exemples de continuation pour la méthode `getTemperature`. La première continuation (lignes 2 à 8) fournit une valeur par défaut (*i.e.*, 21). La seconde continuation (lignes 11 à 23) tente d'accéder à nouveau à la source `température` de l'objet communicant concerné. En cas de nouvel échec, une valeur par défaut est retournée.

Si la source de données `temperature` avait été déclarée en utilisant la déclaration **skipped catch** dans l'opérateur de contexte `AverageTemperature`, la classe *proxy* générée inclurait uniquement une méthode `getTemperature` sans le paramètre pour la continuation, empêchant le développeur de traiter les erreurs. Dans ce cas, lorsqu'une erreur se produit, l'exception de type `ApplicationLevelException` est automatiquement propagée aux composants appelants jusqu'à ce qu'elle soit attrapée. Parce que les systèmes d'informatique ubiquitaire reposent sur un modèle d'exécution réactif, les exceptions de type `ApplicationLevelException` ignorées par le composant le plus haut dans la pile d'appel, peuvent être sainement interceptées par notre modèle de gestion d'erreurs, sans interrompre l'application. Ainsi, le développeur n'a pas besoin d'écrire du code de gestion d'erreurs, ni de modifier chaque signature de méthode où une exception n'est pas traitée pour déclarer sa propagation, comme en Java.

Sinon, si la source de données `temperature` avait été déclarée en utilisant la déclaration **optional catch** dans l'opérateur de contexte `AverageTemperature`, les deux méthodes, avec et sans le paramètre pour la continuation, seraient générées dans la classe *proxy*, déléguant au développeur le choix de traiter ou non les erreurs.

3. Une classe *proxy* est générée pour chaque interaction entre composants déclarée dans la couche d'architecture.

```
// default value
Temperature temperature = t.getTemperature(
  new TemperatureContinuation() {
    public Temperature onError() {
      return new Temperature(21, TemperatureUnit.CELCIUS);
    }
  }
);

// retry + default value
Temperature temperature = t.getTemperature(
  new TemperatureContinuation() {
    public Temperature onError() {
      return t.getTemperature(
        new TemperatureContinuation() {
          public Temperature onError() {
            return new Temperature(21, TemperatureUnit.CELCIUS);
          }
        }
      );
    }
  }
);

// do nothing
sprinkler.on(
  new OnContinuation() {
    public void onError() {
      // do nothing
    }
  }
);
```

FIGURE 8.6 – Exemples de continuations

Considérons un autre exemple, l'interaction de l'opérateur de contrôle `FireController` avec les objets communicants de type `Sprinkler`. Cette interaction est déclarée en utilisant la déclaration **optional catch** (figure 8.3, ligne 18). Par conséquent, le développeur peut, par exemple, soit invoquer les méthodes de l'action `OnOff` sans continuation, laissant l'exception se propager en cas d'échec, soit les invoquer avec une continuation vide, ignorant l'erreur et poursuivant l'exécution. La dernière option est illustrée à la figure 8.6, lignes 26 à 32.

Au niveau du système

Au niveau du système, la gestion des erreurs consiste à implémenter les opérateurs de contexte et de contrôle déclarés dans l'architecture de gestion d'erreurs. À l'instar des opérateurs de contexte et de contrôle de l'architecture fonctionnelle, leur implémentation nécessite de créer une sous-classe de la classe abstraite correspondante générée par le compilateur étendu de DiaSpec. La figure 8.7 représente un extrait d'une implémentation de l'opérateur de contexte `FireExtinctionFailure`, chargée de détecter les pièces du bâtiment dans lesquelles il n'y a aucun dispositif d'extinction d'incendie opérationnel. Cette implémentation étend la

```
1  public class MyFireExtinctionFailure extends FireExtinctionFailure {
2    public MyFireExtinctionFailure() {
3      allFireActuators().subscribePowerException();
4      allSprinklers().subscribeWaterPressureException();
5      [...]
6    }
7    @Override
8    public void onNewPowerException(FireActuator fireActuator, PowerException power) {
9      setFireExtinctionFailure(fireActuator.getLocation(),
            noMoreCorrectFireActuator(fireActuator.getLocation()));
10   }
11   @Override
12   public void onWaterPressureException(Sprinkler sprinkler, WaterPressureException waterPressure) {
13     setFireExtinctionFailure(sprinkler.getLocation(),
            noMoreCorrectFireActuator(sprinkler.getLocation()));
14   }
15   [...]
16 }
```

Figure 8.7 – Extrait d'une implémentation de l'opérateur de contexte `FireExtinction-Failure`

classe abstraite générée correspondante. Parce que l'opérateur de contexte `FireExtinction-Failure` déclare gérer les exceptions de type `PowerException` et `WaterPressureException` provenant respectivement d'objets communicants de type `FireActuator` et `Sprinkler` (figure 8.4, lignes 8 à 15), le *framework* généré fournit les méthodes nécessaires pour découvrir et interagir avec de tels objets communicants, ainsi que pour attraper les exceptions qu'ils lèvent. Par exemple, la méthode `allFireActuators` permet de découvrir tous les objets communicants de type `FireActuator` disponibles dans le bâtiment et est utilisée pour souscrire aux événements exceptionnels de type `PowerException` provenant des objets communicants de type `FireActuator` (figure 8.7, ligne 3). De manière similaire, la méthode `allSprinklers` est utilisée pour souscrire aux événements exceptionnels de type `WaterPressureException` provenant des objets communicants de type `Sprinkler` (ligne 4). Enfin, le développeur doit implémenter chaque méthode abstraite, telle que par exemple la méthode `onNewPowerException` (ligne 8). Cette dernière est automatiquement appelée par le *framework* lorsqu'une exception de type `PowerException` est levée.

8.3 Implémentation

Nous décrivons brièvement l'implémentation de notre modèle de gestion d'erreurs. Tout d'abord, nous examinons comment une erreur est transformée à la fois dans l'exception de type `ApplicationLevelException` et dans un événement exceptionnel. Puis, nous discutons comment ils sont propagés à travers les composants du système.

8.3.1 Signalisation des erreurs

Les exceptions Java, quelle que soit leur origine, sont interceptées par une couche intermédiaire, entre le *proxy* du client de l'objet communicant et l'implémentation de cet objet com-

```
1 try {
2   return temperatureSensor.getTemperature();
3 } catch (MeasureException e) { // user−defined exception
4   publishMeasureException(e);
5   throw new ApplicationLevelException();
6 } catch (Exception e) { // built−in exception
7   publishUndeclaredException(new UndeclaredException(e));
8   throw new ApplicationLevelException();
9 }
```

Figure 8.8 – Signalisation des erreurs

municant. Cette couche fait partie du *framework* de programmation généré par le compilateur étendu de DiaSpec. Elle fournit un traitement uniforme des exceptions Java, les convertissant à la fois dans l'exception de type `ApplicationLevelException` et un événement exceptionnel.

Une exception peut être soit définie par l'utilisateur, soit *built-in*. Les exceptions définies par l'utilisateur correspondent à celles déclarées dans la couche de spécification de DiaSpec. Elles sont levées par les implémentations des objets communicants, en utilisant l'instruction **throw** de Java. Une exception qui n'est pas déclarée dans la couche de spécification est considérée comme étant *buit-in*. Par exemple, lorsqu'un objet communicant devient indisponible, une exception est levée par le *proxy* du client. De plus, le compilateur étendu de DiaSpec génère certains mécanismes de détection d'erreurs (*e.g.*, *watchdog* et *heartbeat*). Ces mécanismes permettent aux opérateurs de contexte de l'architecture de gestion d'erreurs de traiter les erreurs de manière proactive (*i.e.*, avant qu'elles n'affectent les applications), améliorant ainsi la fiabilité de la plate-forme d'informatique ubiquitaire.

La couche entre le *proxy* d'un client et l'implémentation d'un objet communicant de type `TemperatureSensor` est illustrée à la figure 8.8. L'exception Java de type `MeasureException` est transformée à la fois dans un événement exceptionnel de type `MeasureException` (ligne 4) et l'exception de type `ApplicationLevelException` (ligne 5). De manière similaire, l'exception *built-in* Java de type `Exception` est transformée dans un événement exceptionnel de type `UndeclaredException` (ligne 7) et l'exception de type `ApplicationLevelException` (ligne 8). Les événements exceptionnels sont donc des événements DiaSpec publiés comme des sources de données (lignes 4 et 7).

8.3.2 Propagation des erreurs

Une fois convertie dans un événement exceptionnel, l'erreur est raffinée par les opérateurs de contexte de l'architecture de gestion d'erreurs. En particulier, ces opérateurs interprètent, agrègent et transforment les informations relatives à la cause de l'erreur, et les combinent avec d'autres sources de données pour produire des valeurs spécifiques à l'application. Ces valeurs sont ensuite utilisées par des opérateurs de contrôle pour déclencher des stratégies de réparation au niveau du système.

Au niveau de l'application, l'exception de type `ApplicationLevelException` est soit propagée, soit traitée dans une continuation. Elle est propagée dans la pile d'appel lorsque la gestion des erreurs a été déclarée en utilisant la déclaration **skipped catch**. Dans le cas d'une déclaration **optional catch** ou **mandatory catch**, le compilateur étendu de DiaSpec génère une méthode qui attrape l'exception de type `ApplicationLevelException` et exécute la

continuation, comme illustré à la figure 8.9.

```
1 public Temperature getTemperature(TemperatureContinuation tc) {
2   try {
3     return this.getTemperature();
4   } catch (ApplicationLevelException e) {
5     return tc.onError();
6   }
7 }
```

Figure 8.9 – Propagation de l'exception de type `ApplicationLevelException`

8.4 Travaux connexes

Nous présentons diverses approches existantes pour gérer les erreurs dans un système logiciel, qu'il soit ou non d'informatique ubiquitaire. À notre connaissance, aucune approche ne permet de spécifier, architecturer et implémenter la gestion des erreurs, ainsi que de décomposer le traitement d'une erreur d'un point de vue de l'application et d'un point de vue du système.

Architecture. Les langages de description d'architecture (chapitre 2) spécifient la structure et le comportement d'un système logiciel, qu'il soit ou non distribué, afin de garantir certaines propriétés à la fois statiquement et dynamiquement. Bien que la plupart d'entre eux cible certains aspects spécifiques d'un système [AG97, MK96, LV95], peu d'ADL s'intéressent à la spécification de la propagation des erreurs dans des descriptions architecturales données. Filho *et al.* proposent un «*framework* conceptuel», nommé Aereal, pour décrire et analyser les exceptions qui se propagent entre les composants de l'architecture dans un style architectural donné [FBR06]. Au contraire, notre approche va au-delà de la vérification dans la mesure où un *framework* de programmation est généré à partir de descriptions architecturales. Ce *framework* de programmation implémente les déclarations de gestion d'erreurs, garantissant que le système développé est conforme à ces déclarations. Notre approche générative est rendue possible grâce à la nature dédiée de DiaSpec.

Patrons d'exception. *Portland Pattern Repository* [Cun95] propose un ensemble de patrons d'exception pour des langages de programmation généralistes. Ce dépôt s'organise en plusieurs catégories : définition des types d'exception, levée des exceptions, traitement des exceptions, *etc.* Ces patrons d'exception sont des travaux en cours qui peuvent être librement étendus et discutés. Notre modèle combine, adapte et automatise deux d'entre eux aux besoins de la gestion des erreurs dans les systèmes d'informatique ubiquitaire.

L'exception de type `ApplicationLevelException`, proposée dans notre modèle, peut être implémentée avec le patron d'exception `BottomPropagation` dans lequel une exception ne comporte aucune information. Elle est également similaire à la monade `Maybe`[4] dans Haskell. Les événements exceptionnels, quant à eux, sont implémentés avec le patron d'exception `ExceptionReporter`. Ce dernier permet de découpler le traitement des erreurs du flot de

4. http://www.haskell.org/haskellwiki/Maybe

contrôle, offrant par exemple la possibilité aux exceptions d'être traitées par plusieurs gestionnaires. Il adapte le modèle événementiel *publish/subscribe* pour signaler des erreurs.

Non seulement notre approche s'articule autour de ces deux patrons d'exception (*i.e.*, `BottomPropagation` et `ExceptionReporter`), mais elle automatise également leur implémentation dans le *framework* de programmation généré.

Intergiciels et langages de programmation. Beaucoup de couches logicielles dédiées au domaine de l'informatique ubiquitaire ont été proposées pour fournir du support de programmation pour la gestion des erreurs [CRC05, Gri04, PJKF03]. Par exemple, le projet one.world [Gri04] propose un mécanisme de point de contrôle (*check-pointing*) qui permet aux développeurs de sauvegarder l'état d'un composant et de le restaurer ultérieurement afin de poursuivre l'exécution du composant après une défaillance (*e.g.*, une coupure de courant). Il permet également d'améliorer la robustesse des systèmes d'informatique ubiquitaire en fournissant de la persistance des données et des transactions. Notre modèle de gestion d'erreurs est complémentaire à cette approche. Nous prévoyons d'enrichir DiaSpec en intégrant des déclarations de stratégies de tolérance aux fautes, générant des mécanismes de recouvrement d'erreurs dans le *framework* de programmation.

Dedecker *et al.* proposent un langage dédié au développement d'applications dans un réseau mobile, nommé AmbientTalk [DVCM+06]. En particulier, ils introduisent un mécanisme dédié de gestion des exceptions [MDB+06] pour faire face à certaines spécificités du matériel mobile [DVCM+06], telles que la volatilité des connexions réseau ou encore la concurrence. Ce mécanisme se compose d'un ensemble de constructions syntaxiques qui permettent de traiter les exceptions à différents niveaux de granularité dans le code de l'application : message, bloc et collaboration. Une différence fondamentale avec notre approche est que ce mécanisme s'appuie encore sur une forme de bloc **try-catch** qui ne permet pas une séparation claire entre la gestion d'erreurs au niveau de l'application et celle au niveau du système.

Différents paradigmes de communication ont été utilisés pour développer des applications d'informatique ubiquitaire, fondés sur des données (*e.g.*, *tuple space* [Gel85]) ou sur le contrôle (*e.g.*, le modèle des acteurs [Agh86]). Ces paradigmes permettent de découpler les communications entre les objets communicants de leur dimension *spatiale* et *temporelle*. Cette stratégie améliore la résilience d'un système à travers l'isolation d'erreurs et empêche les erreurs d'être propagées. Toutefois, ces approches sont limitées aux erreurs réseau. Au contraire, notre approche s'attaque à une variété d'erreurs, allant des objets communicants défectueux aux erreurs de la plate-forme. De plus, elle permet de gérer les erreurs dès l'architecture, élevant le niveau d'abstraction de la gestion d'erreurs.

Demsky et Dash proposent un langage de tâches, nommé Bristlecone, pour développer des systèmes logiciels robustes [DD08]. Leur approche privilégie la poursuite de l'exécution et peut tolérer certaines dégradations à partir d'un comportement spécifié. L'environnement d'exécution de Bristlecone utilise les spécifications des tâches pour déterminer la tâche à exécuter. Parce que les tâches dans Bristlecone ont une sémantique transactionnelle, lorsqu'une tâche échoue à cause d'une erreur logicielle ou matérielle, l'environnement d'exécution de Bristlecone interrompt la transaction de la tâche. Pour éviter de déclencher à nouveau la même faute sous-jacente, l'environnement de Bristlecone enregistre les combinaisons de tâche qui ont causé l'échec, et exécute d'autres tâches. À cet égard, Bristlecone repose sur une stratégie unique qui traite les erreurs à un seul niveau de granularité : la tâche. Au contraire, notre approche peut recouvrer une erreur causée par un objet communicant défectueux sans

interrompre l'exécution des applications. Le code de recouvrement est défini à la fois au niveau de l'application et au niveau du système, ce qui permet l'implémentation d'une variété de stratégies de recouvrement.

Programmation orientée aspect (*Aspect-Oriented Programming* ou AOP). L'entrelacement entre le code fonctionnel et exceptionnel dans un système logiciel est une des conséquences du manque de support de programmation pour gérer les exceptions. La programmation orientée aspect est une technique de programmation pour modulariser les préoccupations transversales des programmes [EFB01], telles que la gestion des exceptions. Lippert et Lopes [LL00] ont étudié la réingéniérie du code de gestion d'exception en utilisant le langage orienté aspect AspectJ [Lad03]. Leur étude démontre que l'AOP réduit l'entrelacement du code lié à la gestion d'exceptions, ainsi que la portion de code correspondante.

Par comparaison, notre approche modularise et factorise le code de gestion des erreurs au niveau du système. Elle conserve également le code de gestion d'erreurs spécifique à l'application dans le code de l'application. De plus, l'approche AOP est spécifique à un programme cible, rendant difficile d'appliquer le tissage des actions à une variété de programmes.

Cacho *et al.* proposent une approche orientée aspect, nommé EJFlow, qui étend AspectJ et permet aux développeurs de modulariser le code de gestion d'erreurs au niveau du système [CFGF08]. Pour cela, des mécanismes sont introduits afin d'associer des gestionnaires d'exceptions avec des flux exceptionnels dans le code Java. Contrairement à notre approche, l'AOP ne fournit pas aux développeurs de support de programmation ni les guident dans l'implémentation du code de gestion d'exceptions.

8.5 Bilan

Notre approche permet de spécifier, architecturer et supporter la gestion des erreurs dans un système d'informatique ubiquitaire. Nous avons étendu une couche de spécification et d'architecture avec des déclarations de gestion d'erreurs. À partir de ces déclarations, un support de programmation peut être généré pour signaler, propager et traiter les erreurs issues d'exceptions Java. Ce support rend l'implémentation de la gestion d'erreurs plus rigoureuse et systématique. Ainsi, l'implémentation est dirigée par les déclarations de gestion d'erreurs. De plus, notre approche permet une séparation claire entre la gestion d'erreurs au niveau de l'application et celle au niveau du système.

Chapitre 9

Conclusion

Face à l'augmentation de la taille et de la complexité des systèmes logiciels, il convient de les décrire à un plus haut niveau d'abstraction (*i.e.*, au-delà du code) avant de concrètement les implémenter. Toutefois, l'utilisation de ces descriptions de haut niveau dans les processus de construction et de vérification des systèmes reste très rudimentaire, ne permettant pas de véritablement guider et faciliter le développement logiciel.

Nous avons présenté dans cette thèse une nouvelle approche pour rendre plus simple et plus sûr le développement de systèmes logiciels. Cette approche repose sur l'utilisation de langages dédiés et sur un couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation. Elle consiste tout d'abord à décrire à un haut niveau d'abstraction différents aspects, à la fois fonctionnels et non fonctionnels, d'un système dans la couche de spécification et d'architecture. Ces descriptions sont ensuite analysées et utilisées pour personnaliser la couche d'implémentation, afin de faciliter la construction et la vérification du système logiciel. Nous avons illustré notre approche dans le domaine de l'informatique ubiquitaire. Pour cela, nous avons conçu deux langages dédiés à l'orchestration d'objets communicants, Pantaχou et Pantagruel.

Dans ce chapitre, nous dressons un bilan des différentes contributions de cette thèse et présentons quelques perspectives de recherche.

9.1 Contributions

La contribution de cette thèse est double. Dans un premier temps, nous avons montré comment, en couplant fortement une couche de spécification et d'architecture et une couche d'implémentation dédiées à un domaine d'application, il est possible de faciliter la construction et la vérification de systèmes logiciels. Dans un second temps, nous avons proposé une mise en œuvre de cette approche dans le domaine de l'informatique ubiquitaire et nous avons évalué ses avantages en termes de facilité de programmation et de sûreté.

Analyse de domaine. Nous avons effectué une analyse complète du domaine des systèmes d'informatique ubiquitaire. À partir des résultats de cette analyse, nous avons conçus deux langages dédiés, Pantaχou, un langage de script, et Pantagruel, un langage visuel. Ces deux langages dédiés permettent de développer des systèmes d'informatique ubiquitaire à des niveaux d'abstraction différents.

Approche langage. Nous avons décrit différents couplages entre des couches de spécification et d'architecture et des couches d'implémentation, dédiées au développement de systèmes d'informatique ubiquitaire et évoluant à des niveaux d'abstraction différents. Nous avons montré comment, grâce au couplage, la programmation est rigoureusement dirigée par les descriptions de haut niveau de la couche de spécification et d'architecture, guidant le développement logiciel et permettant aux développeurs de se concentrer sur la logique du programme. Nous avons également montré que les bénéfices d'un tel couplage augmentent avec le degré de spécificité de la couche d'implémentation.

Développement logiciel plus simple. Les descriptions de la couche de spécification et d'architecture permettent de personnaliser la couche d'implémentation. Nous avons montré comment le processus de personnalisation permet d'offrir aux développeurs des constructions et des abstractions de programmation de haut niveau et dédiées pour guider et supporter l'implémentation de différents aspects d'un système, à la fois fonctionnels et non fonctionnels. En particulier, nous avons montré comment notre approche permet de simplifier la découverte d'objets communicants, ainsi que la gestion des erreurs dans un système d'informatique ubiquitaire.

Développement logiciel plus sûr. Le couplage fort permet d'assurer automatiquement et systématiquement que l'implémentation du système est conforme aux descriptions de la couche de spécification et d'architecture. Ainsi, nous avons montré comment certaines propriétés deviennent apparentes dans la structure du système, facilitant l'analyse statique du code et la vérification de propriétés. Il devient par exemple possible de garantir par construction l'intégrité des communications. De plus, le typage fort des couches d'implémentation permet d'effectuer statiquement des vérifications de cohérence (*e.g.*, pour les événements publiés et reçus ou encore pour les requêtes de découverte d'objets communicants). Enfin, nous avons montré comment notre approche permet de développer des systèmes d'informatique ubiquitaire plus fiables grâce à une gestion d'erreurs facilitée et systématisée.

9.2 Perspectives

Les travaux présentés dans cette thèse décrivent une nouvelle approche pour rendre plus simple et plus sûr le développement logiciel. Cette approche, fondée sur l'utilisation de langages dédiés et sur un couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation, ouvre de nouvelles perspectives de développement en termes de facilité de programmation et de vérification de systèmes logiciels.

Nous avons illustré et validé notre approche dans le domaine du développement de systèmes d'informatique ubiquitaire. De la même manière, nous pensons qu'il serait tout à fait possible d'appliquer notre approche à d'autres domaines d'application. Par exemple, le développement d'applications pour téléphone mobile (*i.e.*, les *smartphones*) constituerait un bon candidat. En effet, différents aspects d'une application pour téléphone mobile pourraient être spécifiés à un haut niveau d'abstraction, tels que le type d'application (*e.g.*, jeux, bureau), les fonctionnalités du téléphone utilisées (*e.g.*, gyroscope, écran tactile) ou encore les données de l'utilisateur requises (*e.g.*, informations personnelles, carnet d'adresses). Ces informations seraient ensuite utilisées pour personnaliser la couche d'implémentation fournie aux développeurs, garantissant par exemple que l'application développée ne transmet pas d'informations

sensibles. De telles vérifications sont difficiles, voire impossibles à réaliser dans le cas d'un support de programmation générique et statique comme ceux actuellement proposés.

À l'instar de nos travaux sur la gestion des erreurs présentés dans le chapitre 8, plusieurs travaux ont débuté au sein de l'équipe-projet PHOENIX pour faciliter la gestion d'autres aspects non fonctionnels d'un système d'informatique ubiquitaire, tels que la gestion de conflits entre les ressources du système [JCL11] ou encore la gestion de la qualité de service (*QoS*) [GBC11]. D'autres aspects non fonctionnels, tout aussi importants, devraient être considérés, tels que la protection de la vie privée des utilisateurs ou encore la tolérance aux fautes. Toutefois, la gestion de ces différents aspects non fonctionnels soulève la question de leur cohabitation, aussi bien au niveau de la couche de spécification et d'architecture qu'au niveau de la couche d'implémentation. Par exemple, il est important de s'assurer que le support de programmation généré pour la gestion des conflits entre les ressources et celui pour la gestion d'erreurs au niveau de l'application peuvent s'intégrer dans la couche d'implémentation sans interférer ni compliquer la tâche des développeurs.

Enfin, nous avons montré dans le chapitre 7 que le couplage fort entre une couche de spécification et d'architecture et une couche d'implémentation, dédiées à un domaine d'application, introduisait de nouvelles perspectives de vérification. Dans de précédents travaux [LMC07, LMC06, Mer06], nous avons également montré que, de par la nature haut niveau et dédiée de certains DSL (*e.g.*, Pantagruel), le processus de vérification devenait aussi haut niveau et à la portée d'outils de vérification formelle, tels qu'un vérificateur de modèles (*model checker*). Ainsi, nous prévoyons d'utiliser la vérification de modèles (*model checking*) pour garantir certaines propriétés dans un programme Pantagruel. Ces propriétés concernent le langage, mais également l'intention du développeur et la sémantique des règles d'orchestration dans son intégralité.

Chapitre 10

Publications

Conférences internationales

1. J. Mercadal, Q. Enard, C. Consel et N. Loriant. A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing. Dans *Proceedings of the 25th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 47–61, Reno/Tahoe, Nevada, USA. ACM Press, 2010
2. Z. Drey, J. Mercadal et C. Consel. A Taxonomy-Driven Approach to Visually Prototyping Pervasive Computing Applications. Dans *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages (DSL'09)*, pages 78–99, Oxford, UK. Springer-Verlag, 2009
3. J. Mercadal, N. Palix, C. Consel et J. Lawall. Pantaxou: a Domain-Specific Language for Developing Safe Coordination Services. Dans *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*, pages 149–160, Nashville, TN, USA. ACM Press, 2008
4. F. Latry, J. Mercadal et C. Consel. Staging Telephony Service Creation: a Language Approach. Dans *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'07)*, pages 99–110, New York, NY, USA. ACM Press, 2007

Workshops internationaux

1. F. Latry, J. Mercadal et C. Consel. Processing Domain-Specific Modeling Languages: A Case Study in Telephony Services. Dans *Proceedings of the 1st GPCE Workshop for QoS Provisioning in Distributed Systems (GPCE4QoS)*, Portland, USA. 2006

Posters

1. D. Cassou, J. Bruneau, J. Mercadal, Q. Enard, E. Balland, N. Loriant et C. Consel. Towards a Tool-Based Development Methodology for Sense/Compute/-Control Applications. Dans *Proceedings of the 25th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 247–248, Reno/Tahoe, Nevada, USA. Poster. ACM Press, 2010

Bibliographie

[AAG93] G. D. Abowd, R. Allen et D. Garlan. Using Style to Understand Descriptions of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 18(5):9–20. ACM Press, 1993.

[AAG95] G. D. Abowd, R. Allen et D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Tansactions on Software Engineering and Methodology*, 4(4):319–364. ACM Press, 1995.

[Abo99] G. D. Abowd. Software Engineering Issues for Ubiquitous Computing. Dans *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 75–84, Los Angeles, California, USA. ACM Press, 1999.

[ACN02a] J. Aldrich, C. Chambers et D. Notkin. Architectural Reasoning in ArchJava. Dans *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*, pages 334–367. Springer-Verlag, 2002.

[ACN02b] J. Aldrich, C. Chambers et D. Notkin. ArchJava : Connecting Software Architecture to Implementation. Dans *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 187–197, Orlando, Florida. ACM Press, 2002.

[AG97] R. Allen et D. Garlan. A Formal Basis for Architectural Connection. *ACM Tansactions on Software Engineering and Methodology*, 6(3):213–249. ACM Press, 1997.

[Agh86] G. Agha. *Actors : a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[AKY02] M. Adkins, J. Kruse et R. Younger. Ubiquitous Computing : Omnipresent Technology in Support of Network Centric Warfare. Dans *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, page 40. IEEE Computer Society Press, 2002.

[AL03] A. Alexandrescu et K. Lorincz. ArchJava : an Evaluation. Rapport technique, University of Washington, 2003.

[ALSU06] A. V. Aho, M. S. Lam, R. Sethi et J. D. Ullman. *Compilers : Principles, Techniques, and Tools.* Addison-Wesley, 2006.

[AP03] A. W. Appel et J. Palsberg. *Modern Compiler Implementation in Java (2nd ed.).* Cambridge University Press, 2003.

[Ara89] G. Arango. Domain Analysis : From Art Form To Engineering Discipline. *ACM SIGSOFT Software Engineering Notes*, 14(3):152–159. ACM Press, 1989.

[AvDR95] B. R. T. Arnold, A. van Deursen et M. Res. An Algebraic Specification of a Language for Describing Financial Products. Dans *Proceedings of the 17th ICSE Workshop on Formal Methods Application in Software Engineering Practice*, pages 6–13. IEEE Computer Society Press, 1995.

[Bar05] J. E. Bardram. The Java Context Awareness Framework (JCAF) – A Service Infrastructure and Programming Framework for Context-Aware Applications. Dans *Proceedings of the 3rd International Conference on Pervasive Computing (PERVASIVE'05)*, pages 98–115. Springer-Verlag, 2005.

[BC06] P. Barron et V. Cahill. YABS : a Domain-Specific Language for Pervasive Computing based on Stigmergy. Dans *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 285–294, Portland, Oregon, USA. ACM Press, 2006.

[BCL+06] L. Burgy, C. Consel, F. Latry, J. Lawall, N. Palix et L. Réveillère. Language Technology for Internet-Telephony Service Creation. Dans *Proceedings of the 41st IEEE International Conference on Communications (ICC'06)*, pages 1795–1800, Istambul, Turkey. IEEE Computer Society Press, 2006.

[BCSS99] G. Banavar, T. D. Chandra, R. E. Strom et D. C. Sturman. A Case for Message Oriented Middleware. Dans *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, pages 1–18. Springer-Verlag, 1999.

[BEJV96] P. Binns, M. Engelhart, M. Jackson et S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation, and Control. *Journal of Software Engineering and Knowledge Engineering*, 6(2):201–227. 1996.

[Ben86] J. Bentley. Programming Pearls : Little Languages. *Communications of the ACM*, 29(8):711–721. ACM Press, 1986.

[Ber07] G. Berry. Pourquoi et comment le monde devient numérique. http://www.college-de-france.fr/default/EN/all/inn_tec2007/, 2007.

[BKVV08] M. Bravenboer, K. T. Kalleberg, R. Vermaas et E. Visser. Stratego/XT 0.17. A Language and Toolset for Program Transformation. *Science of Computer Programming*, 72(1-2):52–70. Elsevier North-Holland, Inc., 2008.

[BN84] A. D. Birrell et B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59. ACM Press, 1984.

[BR70] J. N. Buxton et B. Randell, éditeurs. *Software Engineering Techniques : Report of a Conference Sponsored by the NATO Science Committee.* Rome, Italie, 27-31 octobre 1969, Brussels, Scientific Affairs Division, NATO, 1970.

[BRSB03] R. Ballagas, M. Ringel, M. Stone et J. Borchers. iStuff : a Physical User Interface Toolkit for Ubiquitous Computing Environments. Dans *Proceedings of the 21st International Conference on Human Factors in Computing Systems (CHI'03)*, pages 537–544, Ft. Lauderdale, Florida, USA. ACM Press, 2003.

[Cas11] D. Cassou. *Développement logiciel orienté paradigme de conception : la programmation dirigée par la spécification.* Thèse de doctorat, Université de Bordeaux, France, 2011.

[CBB+10] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord et J. Stafford. *Documenting Software Architectures : Views and Beyond (2nd ed.)*. Addison-Wesley, 2010.

[CBCL11] D. Cassou, E. Balland, C. Consel et J. Lawall. Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications. Dans *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 431–440, Honolulu, HI, USA. ACM Press, 2011.

[CBLC09] D. Cassou, B. Bertran, N. Loriant et C. Consel. A Generative Programming Approach to Developing Pervasive Computing Systems. Dans *Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE'09)*, pages 137–146, Denver, Colorado, USA. ACM Press, 2009.

[CBM+10] D. Cassou, J. Bruneau, J. Mercadal, Q. Enard, E. Balland, N. Loriant et C. Consel. Towards a Tool-Based Development Methodology for Sense/Compute/Control Applications. Dans *Proceedings of the 25th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 247–248, Reno/Tahoe, Nevada, USA. Poster. ACM Press, 2010.

[CCG+07] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe et S. Zachariadis. The RUNES Middleware for Networked Embedded Systems and its Application in a Disaster Management Scenario. Dans *Proceedings of the 5th IEEE International Conference on Pervasive Computing and Communications (PerCom'07)*, pages 69–78. IEEE Computer Society Press, 2007.

[CCI99] R. Cerqueira, C. Cassino et R. Ierusalimschy. Dynamic Component Gluing Across Different Componentware Systems. Dans *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'99)*, page 362–371. IEEE Computer Society Press, 1999.

[CCW03] M. Caporuscio, A. Carzaniga et A. L. Wolf. Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications. *IEEE Transactions on Software Engineering*, 29(12):1059–1071. IEEE Computer Society Press, 2003.

[CD93] C. Consel et O. Danvy. Tutorial Notes on Partial Evaluation. Dans *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*, pages 493–501, Charleston, South Carolina, USA. ACM Press, 1993.

[CDK+02] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi et S. Weerawarana. Unraveling the Web Services Web : an Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2):86–93. IEEE Educational Activities Department, 2002.

[CE00] K. Czarnecki et U. W. Eisenecker. *Generative Programming : Methods, Tools, and Applications*. Addison-Wesley, 2000.

[CEI+07] A. Chander, D. Espinosa, N. Islam, P. Lee et G. C. Necula. Enforcing Resource Bounds via Static Verification of Dynamic Checks. *ACM Transactions on Programming Languages and Systems*, 29(5):28. ACM Press, 2007.

[CFGF08] N. Cacho, F. C. Filho, A. Garcia et E. Figueiredo. EJFlow : Taming Exceptional Control Flows in Aspect-Oriented Programming. Dans *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 72–83, Brussels, Belgium. ACM Press, 2008.

[CFJ03] H. Chen, T. Finin et A. Joshi. An Ontology for Context-Aware Pervasive Computing Environments. *Knowledge Engineering Review*, 18(3):197–207. Cambridge University Press, 2003.

[CJ02] G. Cugola et H.-A. Jacobsen. Using Publish/Subscribe Middleware for Mobile Systems. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):25–33. ACM Press, 2002.

[CK02] G. Chen et D. Kotz. Context Aggregation and Dissemination in Ubiquitous Computing Systems. Dans *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA '02)*, pages 105–114. IEEE Computer Society Press, 2002.

[CLRC05] C. Consel, F. Latry, L. Réveillère et P. Cointe. A Generative Programming Approach to Developing DSL Compilers. Dans *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676, pages 29–46, Tallinn, Estonia. ACM Press, 2005.

[CM98] C. Consel et R. Marlet. Architecturing Software Using a Methodology for Language Development. Dans *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming (PLILP'98)*, pages 170–194, Pisa, Italy. Lecture Notes in Computer Science, 1998.

[Con04] C. Consel. *Domain-Specific Program Generation ; International Seminar, Dagstuhl Castle*, chapitre From A Program Family To A Domain-Specific Language, pages 19–29. Numéro 3016 de Lecture Notes in Computer Science. Springer-Verlag, 2004.

[CRC05] S. Chetan, A. Ranganathan et R. H. Campbell. Towards Fault Tolerant Pervasive Computing. *IEEE Technology and Society Magazine*, 24(1):38–44. IEEE Computer Society Press, 2005.

[CS09] P. Clements et M. Shaw. "The Golden Age of Software Architecture" Revisited. *IEEE Software*, 26(4):70–72. IEEE Computer Society Press, 2009.

[Cun95] Cunningham & Cunningham. Portland Pattern Repository. http://c2.com/cgi/wiki?ExceptionPatterns, 1995.

[DAS01] A. K. Dey, G. D. Abowd et D. Salber. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction*, 16(2):97–166. L. Erlbaum Associates Inc., 2001.

[DD08] B. Demsky et A. Dash. Bristlecone : a Language for Robust Software Systems. Dans *Proceedings of the 22nd European Conference on Object-Oriented (ECOOP'08)*, pages 490–515, Paphos, Cypress. Springer-Verlag, 2008.

[Dey00] A. K. DEY. *Providing Architectural Support for Building Context-Aware Applications.* Thèse de doctorat, Georgia Institute of Technology, 2000.

[DG02] N. DAVIES et H.-W. GELLERSEN. Beyond Prototypes : Challenges in Deploying Ubiquitous Systems. *IEEE Pervasive Computing*, 1(1):26–35. IEEE Educational Activities Department, 2002.

[Dic98] A. DICKMAN. *Designing Applications with MSMQ : Message Queuing for Developers.* Addison-Wesley, 1998.

[DK76] F. DEREMER et H. KRON. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, 2(2):80–86. IEEE Computer Society Press, 1976.

[DMC09] Z. DREY, J. MERCADAL et C. CONSEL. A Taxonomy-Driven Approach to Visually Prototyping Pervasive Computing Applications. Dans *Proceedings of the IFIP TC 2 Working Conference on Domain-Specific Languages (DSL'09)*, pages 78–99, Oxford, UK. Springer-Verlag, 2009.

[Dre10] Z. DREY. *Vers une méthodologie dédiée à l'orchestration d'entités communicantes.* Thèse de doctorat, Université de Bordeaux, France, 2010.

[DSM10] DSM FORUM. Domain-Specific Modeling. http://www.dsmforum.org/, 2010.

[DVCM+06] J. DEDECKER, T. VAN CUTSEM, S. MOSTINCKX, T. D'HONDT et W. DE MEUTER. Ambient-Oriented Programming in AmbientTalk. Dans *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*, pages 230–254, Nantes, France. Springer-Verlag, 2006.

[EBF+08] A. ERBAD, M. BLACKSTOCK, A. FRIDAY, R. LEA et J. AL-MUHTADI. MAGIC Broker : a Middleware Toolkit for Interactive Public Displays. Dans *Proceedings of the 6th IEEE International Conference on Pervasive Computing and Communications (PerCom'08)*, pages 509–514. IEEE Computer Society Press, 2008.

[EE98] G. EDDON et H. EDDON. *Inside Distributed COM.* Microsoft Press, 1998.

[EFB01] T. ELRAD, R. E. FILMAN et A. BADER. Aspect-Oriented Programming : Introduction. *Communications of the ACM*, 44(10):29–32. ACM Press, 2001.

[EFGK03] P. T. EUGSTER, P. A. FELBER, R. GUERRAOUI et A.-M. KERMARREC. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131. ACM Press, 2003.

[EG01] W. K. EDWARDS et R. E. GRINTER. At Home with Ubiquitous Computing : Seven Challenges. Dans *Proceedings of the 3rd International Conference on Ubiquitous Computing (UbiComp'01)*, pages 256–272, Atlanta, Georgia, USA. Springer-Verlag, 2001.

[EH07] T. EKMAN et G. HEDIN. The JastAdd System - Modular Extensible Compiler Construction. *Science of Computer Programming*, 69(1-3):14–26. Elsevier North-Holland, Inc., 2007.

[Ein34] A. EINSTEIN. *Comment je vois le monde.* Flammarion, 1934.

[FBR06] F. C. FILHO, P. H. BRITO et C. M. F. RUBIRA. Specification of Exception Flow in Software Architectures. *Journal of Systems and Software*, 79(10):1397–1418. Elsevier Science Inc., 2006.

[FGD02] R. A. Falbo, G. Guizzardi et K. C. Duarte. An Ontological Approach to Domain Engineering. Dans *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02)*, pages 351–358, Ischia, Italy. ACM Press, 2002.

[FGDTS06] R. B. France, S. Ghosh, T. Dinh-Trong et A. Solberg. Model-Driven Development Using UML 2.0 : Promises and Pitfalls. *IEEE Transactions on Computers*, 39(2):59–66. IEEE Computer Society Press, 2006.

[Fow03] M. Fowler. *UML Distilled : a Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003.

[Fow10] M. Fowler. *Domain-Specific Languages*. Addison-Wesley, 2010.

[FPDF98] W. Frakes, R. Prieto-Díaz et C. Fox. DARE : Domain Analysis and Reuse Environment. *Annals of Software Engineering*, 5:125–141. J. C. Baltzer AG, Science Publishers, 1998.

[GAO94] D. Garlan, R. Allen et J. Ockerbloom. Exploiting Style in Architectural Design Environments. *ACM SIGSOFT Software Engineering Notes*, 19(5): 175–188. ACM Press, 1994.

[Gar95] D. Garlan. What is Style ? Dans *Proceedings of the Dagstuhl Workshop on Software Architecture*, 1995.

[GBC11] S. Gatti, E. Balland et C. Consel. A Step-Wise Approach for Integrating QoS throughout Software Development. Dans *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering (FASE'11)*, pages 217–231, Saarbrücken, Germany. Springer-Verlag, 2011.

[GDL+04] R. Grimm, J. Davis, E. Lemar, A. Macbeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble et D. Wetherall. System Support for Pervasive Applications. *ACM Transactions on Computer Systems*, 22(4): 421–486. ACM Press, 2004.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112. ACM Press, 1985.

[GHJV95] E. Gamma, R. Helm, R. Johnson et J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GIL+95] J. Gindling, A. Ioannidou, J. Loh, O. Lokkebo et A. Repenning. LEGOsheets : a Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick. Dans *Proceedings of the 11th IEEE International Symposium on Visual Languages (VL'95)*, pages 172–179, Darmstadt, Germany. IEEE Computer Society Press, 1995.

[GIM+04] C. Greenhalgh, S. Izadi, J. Mathrick, J. Humble et I. Taylor. ECT : a Toolkit to Support Rapid Construction of Ubicomp Environments. Dans *Proceedings of the Workshop on System Support for Ubiquitous Computing (UbiSys'04), at the 6th International Conference on Ubiquitous Computing (UbiComp'04)*, pages 207–234. Springer-Verlag, 2004.

[GMW97] D. Garlan, R. Monroe et D. Wile. Acme : an Architecture Description Interchange Language. Dans *Proceedings of the 7th Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'97)*, pages 169–183, Toronto, Ontario, Canada. IBM Press, 1997.

[Gri04] R. GRIMM. One.world : Experiences with a Pervasive Computing Architecture. *IEEE Pervasive Computing*, 3(3):22–30. IEEE Educational Activities Department, 2004.

[Gro01] W. GROSSO. *Java RMI.* O'Reilly & Associates, Inc., 2001.

[GSCK04] J. GREENFIELD, K. SHORT, S. COOK et S. KENT. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools.* John Wiley & Sons, 2004.

[GSSS02] D. GARLAN, D. SIEWIOREK, A. SMAILAGIC et P. STEENKISTE. Project Aura : Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31. IEEE Educational Activities Department, 2002.

[HCC99] M. HAAHR, R. CUNNINGHAM et V. CAHILL. Supporting CORBA Applications in a Mobile Environment. Dans *Proceedings of the 5th International Conference on Mobile Computing and Networking (MobiCom'99)*, pages 36–47, Seattle, Washington, USA. ACM Press, 1999.

[HCH+03] J. HUMBLE, A. CRABTREE, T. HEMMINGS, K.-P. ÅKESSON, B. KOLEVA, T. RODDEN et P. HANSSON. "Playing with the Bits" User-Configuration of Ubiquitous Domestic Environments. Dans *Proceedings of the 5th International Conference on Ubiquitous Computing (UbiComp'03)*, pages 256–263, Seattle, WA, USA. Springer-Verlag, 2003.

[HG08] L. G. C. HAMEY et S. N. GOLDREI. Implementing a Domain-Specific Language Using Stratego/XT : an Experience Paper. *Electronic Notes in Theoretical Computer Science*, 203(2):37–51. Elsevier Science Publishers B. V., 2008.

[HHKR89] J. HEERING, P. R. H. HENDRIKS, P. KLINT et J. REKERS. The Syntax Definition Formalism SDF - reference manual. *ACM SIGPLAN Notices*, 24(11):43–75. ACM Press, 1989.

[HI06] K. HENRICKSEN et J. INDULSKA. Developing Context-Aware Pervasive Computing Applications : Models and Approach. *Pervasive and Mobile Computing*, 2(1):37–64. Elsevier Science Publishers B. V., 2006.

[Hoa69] C. A. R. HOARE. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580. ACM Press, 1969.

[HR04] D. HAREL et B. RUMPE. Meaningful Modeling : What's the Semantics of "Semantics" ? *IEEE Transactions on Computers*, 37(10):64–72. IEEE Computer Society Press, 2004.

[Hud96] P. HUDAK. Building Domain-Specific Embedded Languages. *ACM Computing Surveys*, 28(4es):196. ACM Press, 1996.

[Iiv96] J. IIVARI. Why Are CASE Tools Not Used ? *Communications of the ACM*, 39(10):94–103. ACM Press, 1996.

[JCL11] H. JAKOB, C. CONSEL et N. LORIANT. Architecturing Conflict Handling of Pervasive Computing Resources. Dans *Proceedings of the 11th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'11)*, pages 92–105, Reykjavik, Iceland. Lecture Notes in Computer Science, 2011.

[Jou06] Journal du CHU de Nice. Accueil des Urgences le parcours du patient optimisé. http://www.chu-nice.fr/site_CHU/file/site/magazines/MAG_39.pdf, 2006.

[JRS+09] M. Jimenez, F. Rosique, P. Sanchez, B. Alvarez et A. Iborra. Habitation : A Domain-Specific Language for Home Automation. *IEEE Software*, 26(4):30–38. IEEE Computer Society Press, 2009.

[JS90] S. C. Johnson et R. Sethi. *UNIX Vol. II : research system (10th ed.)*, chapitre Yacc : a Parser Generator, pages 347–374. W. B. Saunders Company, 1990.

[JS03] G. Judd et P. Steenkiste. Providing Contextual Information to Pervasive Computing Applications. Dans *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom'03)*, page 133. IEEE Computer Society Press, 2003.

[Kal06] K. T. Kalleberg. Stratego : a Programming Language for Program Manipulation. *Crossroads*, 12(3):4–4. ACM Press, 2006.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak et S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Rapport technique CMU/SEI-90-TR-21, Software Engineering Institute, novembre 1990.

[Knu64] D. E. Knuth. Backus Normal Form vs. Backus Naur Form. *Communications of the ACM*, 7(12):735–736. ACM Press, 1964.

[KOS06] P. Kruchten, H. Obbink et J. Stafford. The Past, Present, and Future for Software Architecture. *IEEE Software*, 23(2):22–30. IEEE Computer Society Press, 2006.

[Kru95] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6): 42–50. IEEE Computer Society Press, 1995.

[Kru03] P. Kruchten. *The Rational Unified Process : an Introduction.* Addison-Wesley, 2003.

[KT08] S. Kelly et J.-P. Tolvanen. *Domain-Specific Modeling : Enabling Full Code Generation.* Wiley - IEEE Computer Society Press, 2008.

[Lad03] R. Laddad. *AspectJ in Action : Practical Aspect-Oriented Programming.* Manning Publications Co., 2003.

[Lat07] F. Latry. *Approche langage au développement logiciel : Application au domaine des services de Téléphonie sur IP.* Thèse de doctorat, Université de Bordeaux, France, 2007.

[LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan et W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355. IEEE Computer Society Press, 1995.

[LL00] M. Lippert et C. V. Lopes. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. Dans *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 418–427, Limerick, Ireland. ACM Press, 2000.

[LMC06] F. Latry, J. Mercadal et C. Consel. Processing Domain-Specific Modeling Languages : A Case Study in Telephony Services. Dans *Proceedings of the 1st*

GPCE Workshop for QoS Provisioning in Distributed Systems (GPCE4QoS), Portland, USA. 2006.

[LMC07] F. Latry, J. Mercadal et C. Consel. Staging Telephony Service Creation : a Language Approach. Dans *Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm'07)*, pages 99–110, New York, NY, USA. ACM Press, 2007.

[LS90] M. E. Lesk et E. Schmidt. *UNIX Vol. II : research system (10th ed.)*, chapitre Lex - a Lexical Analyzer Generator, pages 375–387. W. B. Saunders Company, 1990.

[LS00] J. Lennox et H. Schulzrinne. CPL : A Language for User Control of Internet Telephony Services. Internet Engineering Task Force, IPTEL WG, novembre 2000.

[LV95] D. C. Luckham et J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734. IEEE Computer Society Press, 1995.

[McC85] R. McCain. Reusable Software Component Construction : a Product-Oriented Paradigm. Dans *Proceedings of the 5th AiAA/ACM/NASA/IEEE Computers in Aerospace Conference*, pages 125–135, 1985.

[MCE02] C. Mascolo, L. Capra et W. Emmerich. Mobile Computing Middleware. *Advanced Lectures on Networking*, 2497:20–58. Springer-Verlag, 2002.

[MDB+06] S. Mostinckx, J. Dedecker, E. G. Boix, T. van Cutsem et W. De Meuter. Ambient-Oriented Exception Handling. Dans *Advanced Topics in Exception Handling Techniques*, pages 141–160, 2006.

[MECL10] J. Mercadal, Q. Enard, C. Consel et N. Loriant. A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing. Dans *Proceedings of the 25th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*, pages 47–61, Reno/Tahoe, Nevada, USA. ACM Press, 2010.

[Men05] D. A. Menasce. MOM vs. RPC : Communication Models for Distributed Applications. *IEEE Internet Computing*, 9(2):90–93. IEEE Educational Activities Department, 2005.

[Mer06] J. Mercadal. Modélisation et Langages Métiers : Application aux Services de Téléphonie. Mémoire de D.E.A., Université de Bordeaux, France, 2006.

[MHC00] R. Monson-Haefel et D. Chappell. *Java Message Service*. O'Reilly & Associates, Inc., 2000.

[MHS05] M. Mernik, J. Heering et A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344. ACM Press, 2005.

[MK96] J. Magee et J. Kramer. Dynamic Structure in Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6):3–14. ACM Press, 1996.

[MORT96] N. Medvidovic, P. Oreizy, J. E. Robbins et R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. *ACM SIGSOFT Software Engineering Notes*, 21(6):24–32. ACM Press, 1996.

[MPCL08] J. Mercadal, N. Palix, C. Consel et J. Lawall. Pantaxou : a Domain-Specific Language for Developing Safe Coordination Services. Dans *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*, pages 149–160, Nashville, TN, USA. ACM Press, 2008.

[MPR06] A. L. Murphy, G. P. Picco et G.-C. Roman. LIME : a Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Tansactions on Software Engineering and Methodology*, 15(3):279–328. ACM Press, 2006.

[MQR95] M. Moriconi, X. Qian et R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372. IEEE Computer Society Press, 1995.

[MR97] P. Murray-Rust. Chemical Markup Language. *World Wide Web Journal*, 2(4):135–147. O'Reilly & Associates, Inc., 1997.

[MRC+00] F. Mérillon, L. Réveillère, C. Consel, R. Marlet et G. Muller. Devil : an IDL for Hardware Programming. Dans *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI'00)*, pages 17–30, San Diego, California, USA. 2000.

[MT00] N. Medvidovic et R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93. IEEE Computer Society Press, 2000.

[MZ09] M. Mamei et F. Zambonelli. Programming Pervasive and Mobile Computing Applications : the TOTA Approach. *ACM Tansactions on Software Engineering and Methodology*, 18(4):1–56. ACM Press, 2009.

[Nei80] J. M. Neighbors. *Software Construction Using Components*. Thèse de doctorat, University of California, Irvine, 1980.

[OMG95] OMG. The Common Object Request Broker : Architecture and Specification. Rapport technique, Object Management Group, 1995.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Pal08] N. Palix. *Langages dédiés au développement de services de communications*. Thèse de doctorat, Université de Bordeaux, France, 2008.

[Par72] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058. ACM Press, 1972.

[Par76] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1):1–9. IEEE Computer Society Press, 1976.

[Par07] T. Parr. *The Definitive ANTLR Reference : Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[PD90] R. Prieto-Díaz. Domain Analysis : an Introduction. *ACM SIGSOFT Software Engineering Notes*, 15(2):47–54. ACM Press, 1990.

[PDN86] R. Prieto-Díaz et J. M. Neighbors. Module Interconnection Languages. *Journal of Systems and Software*, 6(4):307–334. Elsevier Science Inc., 1986.

[PJ97] Joseph J. Pfeiffer Jr. A Rule-Based Visual Language for Small Robotic Applications. Dans *Proceedings of the 13th IEEE International Symposium on Visual Languages (VL'97)*, page 162. IEEE Computer Society Press, 1997.

[PJKF03] S. R. Ponnekanti, B. Johanson, E. Kiciman et A. Fox. Portability, Extensibility and Robustness in iROS. Dans *Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom'03)*, page 11. IEEE Computer Society Press, 2003.

[Plo04] G. D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139. 2004.

[RB96] P. Reynolds et R. Brangeon. Service Machine Development for an Open Longterm Mobile and Fixed Network Environment, DOLMEN Consortium, 1996.

[RC08] A. Ranganathan et R. H. Campbell. Provably Correct Pervasive Computing Environments. Dans *Proceedings of the 6th IEEE International Conference on Pervasive Computing and Communications (PerCom'08)*, pages 160–169. IEEE Computer Society Press, 2008.

[RCAM+05] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell et M. D. Mickunas. Olympus : a High-Level Programming Model for Pervasive Computing Environments. Dans *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom'05)*, pages 7–16. IEEE Computer Society Press, 2005.

[RHC+02] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell et K. Nahrstedt. A Middleware Infrastructure for Active Spaces. *IEEE Pervasive Computing*, 1(4):74–83. IEEE Educational Activities Department, 2002.

[RSC+02] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley et E. Schooler. SIP : Session Initiation Protocol, Request for Comments 3261, The Internet Engineering Task Force, 2002.

[RWK+08] R. Reichle, M. Wagner, M. U. Khan, K. Geihs, M. Valla, C. Fra, N. Paspallis et G. A. Papadopoulos. A Context Query Language for Pervasive Computing Environments. Dans *Proceedings of the 6th IEEE International Conference on Pervasive Computing and Communications (PerCom'08)*, pages 434–440. IEEE Computer Society Press, 2008.

[Ré01] L. Réveillère. *Approche langage au développement de pilotes de périphériques robustes.* Thèse de doctorat, Université de Bordeaux, France, 2001.

[Sat01] M. Satyanarayanan. Pervasive Computing : Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17. 2001.

[SBP99] T. Sheard, Z. Benaissa et E. Pasalic. DSL Implementation Using Staging and Monads. Dans *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99)*, pages 81–94, Austin, Texas, USA. ACM Press, 1999.

[SC00] J. C. Seco et L. Caires. A Basic Model of Typed Components. Dans *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'00)*, pages 108–128. Springer-Verlag, 2000.

[SC06] M. Shaw et P. Clements. The Golden Age of Software Architecture. *IEEE Software*, 23(2):31–39. IEEE Computer Society Press, 2006.

[Sch86] D. A. Schmidt. *Denotational Semantics : a Methodology for Language Development.* William C. Brown Publishers, 1986.

[Sch95] W. N. Schilit. *A System Architecture for Context-Aware Mobile Computing.* Thèse de doctorat, Columbia University, 1995.

[Sch02] A. Schmidt. *Ubiquitous Computing – Computing in Context.* Thèse de doctorat, Lancaster University, 2002.

[SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young et G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4):314–335. IEEE Computer Society Press, 1995.

[SE43] A. Saint-Exupéry. *Le Petit Prince.* Gallimard, 1943.

[SG96] M. Shaw et D. Garlan. *Software Architecture : Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[SG02] J. P. Sousa et D. Garlan. Aura : an Architectural Framework for User Mobility in Ubiquitous Computing Environments. Dans *Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA'02)*, pages 29–43. Kluwer, B.V., 2002.

[SM03] D. Saha et A. Mukherjee. Pervasive Computing : a Paradigm for the 21st Century. *IEEE Computer*, 36(3):25–31. IEEE Computer Society Press, 2003.

[Sno89] R. Snodgrass. *The Interface Description Language : Definition and Use.* Computer Science Press, Inc., 1989.

[Sof10] Software Engineering Institute. Community Software Architecture Definitions. http://www.sei.cmu.edu/architecture/start/community.cfm, 2010.

[Spi01] D. Spinellis. Notable Design Patterns for Domain-Specific Languages. *Journal of Systems and Software*, 56(1):91–99. Elsevier Science Inc., 2001.

[Sre02] V. C. Sreedhar. Mixin'Up Components. Dans *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, pages 198–207, Orlando, Florida. ACM Press, 2002.

[SS09] S. Staab et R. Studer. *Handbook on Ontologies.* Springer-Verlag, 2009.

[SW09] H. R. Schmidtke et W. Woo. Towards Ontology-Based Formal Verification Methods for Context Aware Systems. Dans *Proceedings of the 7th International Conference on Pervasive Computing (PERVASIVE'09)*, pages 309–326, Nara, Japan. Springer-Verlag, 2009.

[Tah99] W. Taha. *Multi-Stage Programming : Its Theory and Applications.* Thèse de doctorat, Oregon Graduate Institute of Science and Technology, 1999.

[Thi98] S. Thibault. *Langage Dédiés : Conception, Implémentation et Application.* Thèse de doctorat, Université de Rennes, France, 1998.

[Tic79] W. F. Tichy. Software Development Control Based on Module Interconnection. Dans *Proceedings of the 4th International Conference on Software Engineering (ICSE'79)*, pages 29–41, Munich, Germany. IEEE Computer Society Press, 1979.

[TMC99] S. A. Thibault, R. Marlet et C. Consel. Domain-Specific Languages : From Design to Implementation Application to Video Device Drivers Generation. *IEEE Transactions on Software Engineering*, 25(3):363–377. IEEE Computer Society Press, 1999.

[TMD09] R. N. Taylor, N. Medvidovic et E. M. Dashofy. *Software Architecture : Foundations, Theory, and Practice*. John Wiley & Sons, 2009.

[TOHS99] P. Tarr, H. Ossher, W. Harrison et Stanley M. Sutton, Jr. N degrees of separation : multi-dimensional separation of concerns. Dans *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 107–119, Los Angeles, California, United States. ACM Press, 1999.

[UNT10] N. Ubayashi, J. Nomura et T. Tamai. Archface : a Contract Place Where Architectural Design and Code Meet Together. Dans *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, pages 75–84, Cape Town, South Africa. ACM Press, 2010.

[vdBdJKO00] M. G. J. van den Brand, H. A. de Jong, P. Klint et P. A. Olivier. Efficient Annotated Terms. *Software : Practice and Experience*, 30(3):259–291. John Wiley & Sons, 2000.

[vdBHKO02] M. G. J. van den Brand, J. Heering, P. Klint et P. A. Olivier. Compiling Language Definitions : the ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368. ACM Press, 2002.

[vDK98] A. van Deursen et P. Klint. Little Languages : Little Maintenance. *Journal of Software Maintenance*, 10(2):75–92. John Wiley & Sons, 1998.

[vDK02] A. van Deursen et P. Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10: 1–17. 2002.

[vDKV00] A. van Deursen, P. Klint et J. Visser. Domain-specific languages : an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36. ACM Press, 2000.

[vG03] J. van Gumster. Blender as an Educational Tool. Dans *ACM SIGGRAPH 2003 Educators Program*, page 1, San Diego, California, USA. ACM Press, 2003.

[vOvdLKM00] R. van Ommering, F. van der Linden, J. Kramer et J. Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Transactions on Computers*, 33(3):78–85. IEEE Computer Society Press, 2000.

[VWBGK10] E. R. Van Wyk, D. Bodin, J. Gao et L. Krishnan. Silver : an Extensible Attribute Grammar System. *Science of Computer Programming*, 75(1-2):39–54. Elsevier North-Holland, Inc., 2010.

[VWS07] E. R. Van Wyk et A. C. Schwerdfeger. Context-Aware Scanning for Parsing Extensible Languages. Dans *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE'07)*, pages 63–72, Salzburg, Austria. ACM Press, 2007.

[Wad90] P. Wadler. Comprehending Monads. Dans *Proceedings of the 6th ACM Conference on Lisp and Functional Programming (LFP'90)*, pages 61–78, Nice, France. ACM Press, 1990.

[Wei91] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3): 66–75. 1991.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages : an Introduction.* MIT Press, 1993.

[WKU+07] T. Weis, M. Knoll, A. Ulbrich, G. Muhl et A. Brandle. Rapid Prototyping for Pervasive Applications. *IEEE Pervasive Computing*, 6(2):76–84. IEEE Educational Activities Department, 2007.

[WL99] D. M. Weiss et C. T. R. Lai. *Software Product-Line Engineering : a Family-Based Software Development Process.* Addison-Wesley, 1999.

[WP05] R. Want et T. Pering. System Challenges for Ubiquitous & Pervasive Computing. Dans *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 9–14, St. Louis, MO, USA. ACM Press, 2005.

[ZMN05] F. Zhu, M. W. Mutka et L. M. Ni. Service Discovery in Pervasive Computing Environments. *IEEE Pervasive Computing*, 4(4):81–90. IEEE Educational Activities Department, 2005.