UNIVERSITÉ DE BORDEAUX

Département de formation doctorale en informatique

École doctorale EDMI Bordeaux
N° d'ordre : XXXX

Développement logiciel orienté paradigme de conception : la programmation dirigée par la spécification

THÈSE

soutenue le 17/03/2011

pour l'obtention du

Doctorat de l'Université de Bordeaux

(spécialité informatique)

par

Damien Cassou

Jury

Président :	Xavier Blanc,	Professeur à l'Université de Bordeaux
Rapporteurs :	Laurence Duchien,	Professeur à l'Université de Lille
	Yves Ledru,	Professeur à l'Université Joseph Fourier de Grenoble
Examinateurs :	Charles Consel,	Professeur à l'Université de Bordeaux
	Paul Klint,	Professeur à l'Université d'Amsterdam
	Alexander L. Wolf,	Professeur à l'Imperial College de Londres

Laboratoire Bordelais de Recherche en Informatique — UMR 5800

[7 mars 2011 at 13:58]

RÉSUMÉ

Nombre d'applications ont pour comportement principal l'attente d'un événement venant d'un environnement extérieur, la préparation d'un résultat et l'exécution d'actions sur cet environnement. Les interfaces graphiques et les systèmes avioniques en sont des exemples. Le paradigme SCC, pour sense-compute-control, est particulièrement adapté à la description de ces applications. Le développement d'applications suivant ce paradigme est complexe à cause du manque de cadre conceptuel et d'outils de support.

Cette thèse propose un cadre conceptuel dédié au paradigme SCC et se concrétise par un langage de description d'architectures. À partir d'une description dans ce langage, un framework de programmation peut être généré. Il guide l'implémentation d'une application grâce à un support dédié et vérifie que cette implémentation est conforme à l'architecture décrite. Les contributions de cette thèse sont évaluées suivant des critères d'expressivité, d'utilisabilité et de productivité.

LISTE DES PUBLICATIONS

Les travaux discutés dans cette thèse ont été présentés précédemment.

CONFÉRENCES INTERNATIONALES

- « Improving design, programming and verification of software systems using a domain-specific ADL, » dans *ICSE'11 : Proceedings of the 33rd International Conference on Software Engineering*, 2011, Damien Cassou, Émilie Balland, Charles Consel et Julia Lawall (à paraître)
- « A generative programming approach to developing pervasive computing systems, » dans *GPCE'09 : Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, 2009, pages 137–146, Damien Cassou, Benjamin Bertran, Nicolas Loriant et Charles Consel

DÉMONSTRATIONS

- « A Tool Suite to Prototype Pervasive Computing Applications, » dans *PerCom'10 : Proceedings of the 8th International Conference on Pervasive Computing and Communications*, 2010, Damien Cassou, Julien Bruneau et Charles Consel

POSTERS

- « Towards a Tool-based Development Methodology for Sense/Compute/Control Applications, » dans *SPLASH'10 : Proceedings of the 1st International Conference on Systems, Programming, Languages, and Applications : Software for Humanity*, 2010, Damien Cassou, Julien Bruneau, Julien Mercadal, Quentin Enard, Emilie Balland, Nicolas Loriant et Charles Consel

RAPPORTS TECHNIQUES

- « Towards a Tool-based Development Methodology for Pervasive Computing Applications » soumis à *Transactions on Software Engineering*, Damien Cassou, Julien Bruneau, Charles Consel, Emilie Balland

TABLE DES MATIÈRES

I CONTEXTE 1

1 LE PARADIGME SCC 3
- 1.1 Périmètre du paradigme 3
- 1.2 Importance du paradigme SCC 7

2 DÉVELOPPEMENT D'UNE APPLICATION SCC 15
- 2.1 Langages de programmation généralistes 15
- 2.2 Bibliothèques de code 15
- 2.3 Extensions de langages et langages dédiés 16
- 2.4 Approches structurantes 17
- 2.5 Approches architecturales 18
- 2.6 Ingénierie des modèles 18

II APPROCHE PROPOSÉE 21

3 VUE D'ENSEMBLE DE L'APPROCHE 23
- 3.1 Contributions 23
- 3.2 Présentation de l'approche 24

4 DESCRIPTION D'UNE APPLICATION SCC 27
- 4.1 Présentation générale du langage 27
- 4.2 Taxonomie 29
- 4.3 Architecture 33
- 4.4 Interactions 39
- 4.5 Synthèse 44
- 4.6 Liens avec d'autres approches 46
- 4.7 Travaux en cours et futurs 48

5 IMPLÉMENTATION D'UNE APPLICATION SCC 51
- 5.1 Génération de support de programmation 51
- 5.2 Utilisation du support généré 60
- 5.3 Implémentation du compilateur 68
- 5.4 Synthèse 69
- 5.5 Liens avec d'autres approches 70
- 5.6 Travaux en cours et futurs 72

6 ÉVOLUTION D'UNE APPLICATION SCC 75
- 6.1 Changements dans la taxonomie 76
- 6.2 Changements dans l'architecture 76
- 6.3 Changements du système déployé 78
- 6.4 Synthèse 78
- 6.5 Liens avec d'autres approches 79
- 6.6 Travaux en cours et futurs 79

7 VÉRIFICATION D'UNE APPLICATION SCC 81
- 7.1 Analyses sur l'architecture 81
- 7.2 Analyses sur l'implémentation 85
- 7.3 Analyses sur l'exécution 87

7.4 Synthèse 91
7.5 Liens avec d'autres approches 92
7.6 Travaux en cours et futurs 93

III MISE EN OEUVRE 95

8 ÉVALUATION 97
8.1 Expressivité 97
8.2 Utilisabilité 107
8.3 Productivité 108

9 UNE PLATEFORME DE RECHERCHE 113
9.1 Simulateur pour l'informatique ubiquitaire 113
9.2 Ciblage des utilisateurs finaux 113
9.3 Couverture des besoins non fonctionnels 114
9.4 Adaptation multimédia 114
9.5 Déploiement chez les particuliers 115

10 CONCLUSION 117

IV APPENDICES 121

A QUESTIONNAIRE DE L'ÉVALUATION 123

B GRAMMAIRE DU LANGAGE DIASPEC 125

BIBLIOGRAPHIE 131

LISTE DES FIGURES

FIGURE 1 Les quatre couches d'une application SCC 4
FIGURE 2 Les quatre couches représentées sous la forme d'une boucle autour de l'environnement 5
FIGURE 3 Application graphique représentée comme une boucle de contrôle (adaptée de [110]) 9
FIGURE 4 La décomposition traditionnelle d'un système de contrôle de robot mobile en modules fonctionnels (adaptée de [13]) 10
FIGURE 5 Développement d'une application avec notre approche 25
FIGURE 6 Style architectural d'une application SCC 29
FIGURE 7 Modèle sémantique UML simplifié des composants de DiaSpec 34
FIGURE 8 Représentation graphique du moniteur de serveur web 35
FIGURE 9 Utilisation de l'extension d'Eclipse pour DiaSpec 38
FIGURE 10 Représentation graphique créée automatiquement par l'extension d'Eclipse pour DiaSpec 38
FIGURE 11 Interactions possibles dans DiaSpec 40
FIGURE 12 Exemple pour illustrer la synchronisation dans le moniteur de serveur web 43
FIGURE 13 Exemple pour illustrer la disjonction dans le moniteur de serveur web 44
FIGURE 14 Compilation de contrats d'interactions 55
FIGURE 15 Utilisation du motif de conception Composite pour interagir avec les entités Logger 66
FIGURE 16 Architecture du compilateur de DiaSpec 69
FIGURE 17 Évolution de la description de l'application après le début de l'implémentation 75
FIGURE 18 Arbre représentant ce que chaque élément de l'architecture peut définir 82
FIGURE 19 L'application cœur en marche 98
FIGURE 20 L'application cœur décrite avec UML 98
FIGURE 21 L'application cœur décrite avec DiaSpec 99

Figure 22 Représentation d'un avion et de ses différents axes 100
Figure 23 Extrait de l'application de pilote automatique décrite avec DiaSpec 101
Figure 24 Simulation de l'application de pilote automatique d'avion avec le simulateur FlightGear 101
Figure 25 Matériels dédiés aux expérimentations domotiques dans notre laboratoire 102
Figure 26 Architecture du système de sécurité 102
Figure 27 Simulateur du système de sécurité 103
Figure 28 Application de *newscast* décrite avec DiaSpec 104
Figure 29 Simulation de l'application de *newscast* dans une école d'ingénieurs 104
Figure 30 Exemple de dialogue entre un utilisateur et son domicile 105
Figure 31 Application de dialogue avec le domicile décrite avec DiaSpec 106
Figure 32 Les trois volets de l'application communautaire DiaStore 110
Figure 33 Première partie du scénario d'adaptation multimédia 115
Figure 34 Deuxième partie du scénario d'adaptation multimédia 115

LISTE DES TABLEAUX

Table 1 Contrats d'interactions des opérateurs du moniteur de serveur web 43
Table 2 Liste des opérateurs logiques pour les expressions logiques de la découverte d'entités 67
Table 3 Métriques de l'étude de gestion de l'école 105
Table 4 Mesures du code généré pour quelques applications 109

LISTE DES LISTINGS

Listing 1 Taxonomie complète du moniteur de serveur web 32
Listing 2 Architecture complète du moniteur de serveur web 37
Listing 3 Quelques contrats d'interactions associés aux opérateurs du moniteur de serveur web 45
Listing 4 La classe abstraite Java `AbstractAccessLogReader` générée par le compilateur de DiaSpec 53
Listing 5 Une implémentation écrite par un développeur de la classe d'entités `AccessLogReader` 61
Listing 6 Une implémentation écrite par un développeur de la classe d'entités `NSLookup` 62
Listing 7 Une implémentation écrite par un développeur de l'opérateur de contexte `AccessLogParser` 63
Listing 8 Une implémentation écrite par un développeur de l'opérateur de contexte `AccessingProfile` 64
Listing 9 Une implémentation écrite par un développeur de l'opérateur de contexte `IP2Profile` 65
Listing 10 Une implémentation écrite par un développeur de l'opérateur de contrôle `ProfileLogger` 65
Listing 11 Exemple d'utilisation d'une barrière dynamique 89
Listing 12 Grammaire du langage DiaSpec 125

Première partie

CONTEXTE

1

LE PARADIGME SCC

Un paradigme de programmation est un ensemble de règles, de concepts et de pratiques qui contraint et guide le développement d'une application, de la réalisation de son architecture au déploiement, en passant par l'implémentation et le test. Un tel paradigme est souvent associé à un style (ou patron) architectural qui est un ensemble de décisions réutilisables concernant l'architecture d'une application [106, chap. 3].

Taylor et al. [106] définissent une différence entre style et patron, celle-ci sort du cadre de cette thèse

Nombre d'applications ont pour comportement principal (1) l'attente d'un événement (un clic, la réception d'un message, *etc.*), (2) la préparation d'un résultat (une page web, la représentation d'un message, *etc.*) et (3) l'exécution d'actions en retour (renvoi de la page web au client, affichage d'une notification, *etc.*). Le paradigme SCC, pour *sense-compute-control*, est particulièrement adapté à la description de ces applications [106, p. 97]. Par la suite, nous appellerons les applications pour lesquelles le paradigme SCC est applicable les *applications SCC*.

Ce chapitre décrit le périmètre d'utilisation du paradigme SCC ainsi que son importance.

1.1 PÉRIMÈTRE DU PARADIGME

Nous définissons tout d'abord le périmètre du paradigme en listant ses caractéristiques. Ensuite, nous examinons des domaines d'applications qui peuvent faire levier sur ce paradigme.

1.1.1 *Caractéristiques*

Le paradigme SCC est applicable lorsque l'application à développer interagit avec un *environnement*. Cet environnement est soit *physique*, soit *logiciel*, soit une combinaison des deux. Un environnement physique peut, par exemple, être composé d'un bâtiment, d'utilisateurs ou de périphériques matériels. Un environnement logiciel peut par exemple être composé de services web, d'une tierce application ou d'une base de données.

Les applications SCC interagissent avec l'environnement de deux façons : en y capturant de l'information et en agissant dessus. Le paradigme SCC facilite le développement de ces applications en les séparant en quatre couches (Figure 1) :

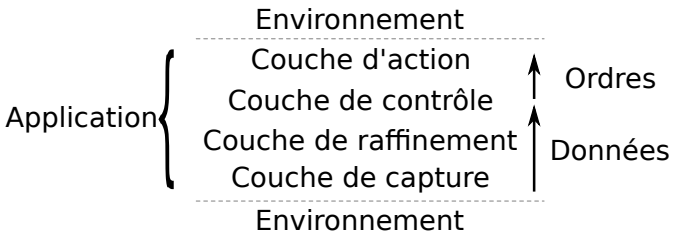

FIGURE 1: Les quatre couches d'une application SCC

COUCHE DE CAPTURE La couche basse de l'application capte l'information utile de l'environnement et rend ces *données* accessibles au reste de l'application. Par exemple, la couche de capture peut transmettre des températures mesurées.

COUCHE DE RAFFINEMENT La couche supérieure raffine les données de la couche de capture vers des données applicatives de plus haut niveau. Le raffinement peut être de l'agrégation de plusieurs données ou de la transformation de données. Par exemple, la couche de raffinement donne la température de la maison en faisant la moyenne de plusieurs valeurs mesurées.

COUCHE DE CONTRÔLE Dans la couche de contrôle, les données applicatives issues de la couche de raffinement sont utilisée pour prendre des décisions sur les actions à entreprendre. Ceci est fait en envoyant des ordres à la couche d'actions. Par exemple, la couche de contrôle peut décider d'allumer le chauffage si la température moyenne est trop faible.

COUCHE D'ACTION La couche d'action est la plus haute et reçoit des *ordres* de la couche de contrôle. Cette couche exécute les ordres de la couche inférieure en agissant sur l'environnement. Par exemple, la couche d'action est responsable du contrôle du thermostat des radiateurs.

Les couches de raffinement et de contrôle contiennent la logique applicative. Taylor et al. [106] réunissent ces deux couches au sein d'une même couche dont le rôle est de prendre les données bas-niveau de la couche de capture et d'actionner les éléments de la couche d'action. Bien que plus simple, ce découpage mélange les rôles de calcul de l'information et d'utilisation du résultat de ce calcul. Dans cette thèse nous utiliserons systématiquement le découpage en quatre couches.

De ce découpage en trois couches vient le nom sense–compute–control

L'interface entre l'application et l'environnement se fait au travers d'entités sur étagère (*off-the-shelf*) de types variés : capteurs ou actionneurs, logicielles ou matérielles, haut-niveau ou bas-niveau *etc.*. Ces entités utilisent des plateformes spécifiques, présentent des modèles d'interactions différents et fournissent des interfaces non standardisées. Pour que cette hétérogénéité ne

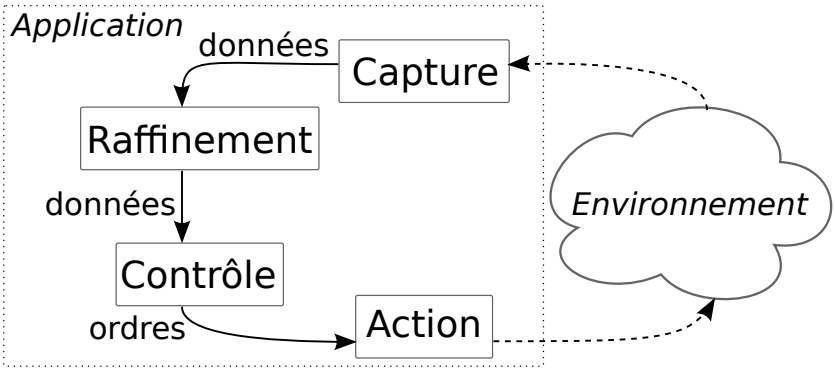

FIGURE 2: Les quatre couches représentées sous la forme d'une boucle autour de l'environnement

pollue pas l'ensemble du code des applications SCC, les couches de capture et d'action isolent l'interface entre l'application et l'environnement de la logique applicative.

Ces quatre couches peuvent aussi être représentées sous la forme d'une boucle autour de l'environnement comme dans la Figure 2. Nous verrons comment ce type de boucles est particulièrement utilisé dans le domaine des systèmes autonomiques qui se réparent et se configurent automatiquement.

Les applications SCC sont souvent en partie ou totalement distribuées. Dans le cas d'une application domotique par exemple, les éléments de la couche de capture sont placés au plus près de l'environnement : les capteurs sont déployés dans chaque pièce et utilisent le réseau pour communiquer avec le reste de l'application.

Le paradigme SCC se définit donc par une interaction avec un environnement, par ses quatre couches aux rôles distincts, par un besoin d'indépendance de la logique applicative vis-à-vis de l'environnement, et par la distribution de tout ou partie des éléments de l'application.

1.1.2 *Domaines d'applications concernés*

Après avoir défini le paradigme de façon absolue, nous le définissons maintenant de manière relative en dressant une liste non exhaustive de domaines qui utilisent déjà le paradigme SCC ou qui pourraient en bénéficier. Pour chaque domaine, nous établissons une liste d'applications types, et nous caractérisons l'environnement.

LA ROBOTIQUE Dans le domaine de la robotique, l'environnement est constitué d'un robot (une base de connaissances, des moteurs, un programme, *etc.*) et de son voisinage (les murs, le sol,

les personnes, *etc.*) [102]. Les applications de la robotique sont, par exemple, le divertissement des utilisateurs et l'accomplissement de tâches (construction, nettoyage, *etc.*).

L'INFORMATIQUE UBIQUITAIRE Dans la vision de Weiser [114], l'informatique ubiquitaire mêle des éléments logiciels et matériels qui s'associent pour que les utilisateurs ne décèlent par leur présence. Dans ce but, les applications de l'informatique ubiquitaire doivent avoir « conscience » de leur environnement, appelé le contexte, et s'y adapter sans que l'utilisateur n'intervienne. Dans l'informatique ubiquitaire, l'environnement est constitué entre autres des utilisateurs, de leur voisinage (comme la pièce ou le bâtiment), d'appareils mobiles (comme les téléphones portables et lecteurs multimédia) et fixes (comme les écrans, les lampes). Parmi les nombreux exemples d'applications de l'informatique ubiquitaire, citons le marché aux puces mobiles [46] qui alerte les acheteurs passant à proximité d'un vendeur susceptible de les intéresser. Citons également les applications qui proposent de transférer les appels téléphoniques en fonction de la localisation du destinataire ou qui proposent de changer de moyen de communication si le destinataire est occupé [54, 81].

L'AVIONIQUE Un avion, qu'il soit civile ou militaire, est composé d'un nombre très important de capteurs [19]. Ces capteurs donnent par exemple la position de l'avion, son angle d'incidence, la température extérieure ainsi que la vitesse et la direction du vent. L'information donnée par ces capteurs est transmise à plusieurs systèmes comme les systèmes de navigation, de communication, de pilote automatique et de radar. Pour ces systèmes, l'environnement est l'avion lui-même, l'équipage, les passagers, le voisinage de l'avion, et les tours de contrôle. Une des applications les plus évidentes est le pilote automatique, mais citons aussi les *in-flight entertaining systems* qui visent à informer et divertir les passagers.

L'AUTOMOBILE Le système mécatronique standard de l'automobile inclut un système de contrôle électronique également appelé architecture Électrique/Électronique [5, 67, 68]. Cette architecture est composée de capteurs (température, vitesse, *etc.*), d'actionneurs (climatisation, *airbag*, tableau de bord) et d'un ou plusieurs calculateurs (appelés *Electronic Control Unit* ou ECU). L'environnement est composé de la voiture, de ses passagers et du voisinage de la voiture. Les applications régulent la température dans l'habitacle et aident à la conduite par exemple.

LES APPLICATIONS WEB Une application web est basée sur une architecture client/serveur [41] : un navigateur web envoie des requêtes à un serveur, qui construit une page web en réponse et la renvoie au client. Certaines applications maintiennent leur état entre deux requêtes d'un utilisateur dans une session. Le paradigme SCC peut s'appliquer aux applications web : dans ce domaine l'environnement est principalement constitué de navigateurs web des utilisateurs, de bases de données, et de sessions. Les applications web de courriers électroniques, les systèmes de gestion de contenu et les magasins en ligne sont des exemples d'applications web.

LES INTERFACES UTILISATEURS De manière plus générale, toutes les applications qui proposent une interface interactive avec les utilisateurs sont susceptibles d'utiliser le paradigme SCC [110]. Dans ce cas, l'environnement est constitué de l'utilisateur et de moyens de communiquer avec lui : périphériques d'entrées (clavier, souris), périphériques de sorties (écran, enceinte), réseaux et logiciels (serveur graphique, navigateur web). Les applications de ce domaine sont les lecteurs audio et vidéo, les logiciels de messagerie, les logiciels de créations multimédia...

Dans cette première section nous avons défini le paradigme SCC en étudiant ses caractéristiques propres. Nous avons aussi constaté que ce paradigme peut être appliqué à des domaines variés et nombreux. La suite de ce chapitre décrit l'importance de ce paradigme.

1.2 IMPORTANCE DU PARADIGME SCC

Dans cette section nous constatons que plusieurs paradigmes existants sont suffisamment proches du paradigme SCC pour être considéré soit comme des cas particuliers soit comme ayant des différences mineures. Nous expliquons ensuite les bénéfices que les domaines d'applications cités plus haut peuvent tirer de l'utilisation du paradigme SCC.

1.2.1 *Relations avec d'autres paradigmes*

D'autres paradigmes de programmation ont été développés et utilisés pour faciliter le développement d'applications. Dans cette section nous étudions les différences et similitudes du paradigme SCC par rapport à d'autres paradigmes proches.

1.2.1.1 *Informatique ubiquitaires*

Dans l'informatique ubiquitaire [114], on trouve de nombreux travaux proposant une définition du *contexte* d'une application et comment rendre l'application adaptable à ce contexte. Une définition souvent reprise est celle apportée par Dey et al. [24] que nous pouvons traduire de l'anglais par : *"Le contexte correspond à n'importe quelle information qui peut être utilisée pour caractériser l'état d'une entité. Une entité est une personne, lieu, ou objet qui est considéré pertinent pour l'interaction entre un utilisateur et une application, incluant l'utilisateur et l'application eux-mêmes."*

Dans la suite nous évoquons les principaux travaux permettant d'adapter dynamiquement une application à son contexte.

CONTEXT TOOLKIT Les travaux autour de Context Toolkit de Dey et al. [24] ont permis de catégoriser les composants de calcul de contexte en trois types : les *widgets*, les *interprètes* et les *agrégateurs*. Les widgets capturent l'information de contexte et fournissent des données de bas niveau (par exemple un widget capteur de température). Les interprètes prennent une donnée bas-niveau et la traduisent vers une donnée de plus haut niveau (par exemple un interprète indique s'il fait chaud ou froid en fonction d'une température). Enfin, les agrégateurs regroupent diverses données de widgets en une seule donnée (par exemple un agrégateur donne la température d'une maison en regroupant les données de tous les capteurs de température de la maison). Les widgets se positionnent naturellement dans la couche de capture du paradigme SCC car ils permettent de récupérer des données de l'environnement. Les interprètes et agrégateurs raffinent les données des widgets et se placent donc naturellement dans la couche de raffinement. Ces travaux définissent aussi les *services* qui exécutent des actions pour le compte des *applications*. Les services se placent dans la couche d'action car ils interagissent directement avec l'environnement. Les applications utilisent les données des interprètes et agrégateurs pour décider des actions à prendre : les applications se placent donc dans la couche de contrôle.

OPERATOR GRAPH Les graphes d'opérateurs (*operator graph*) proposés par Chen et Kotz [16] introduisent une abstraction de graphe pour décrire la collection, l'agrégation et la dissémination d'informations de contexte. Les auteurs remarquent que les applications réagissant aux données de contexte ont une « structure dirigée par les événements », où les changements de contexte sont représentés par des événements. Un graphe est alors composé de *sources* et d'*opérateurs*. Les sources captent les informations de

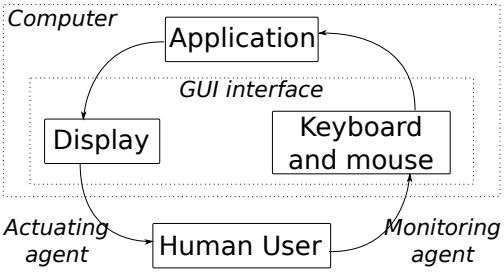

FIGURE 3: Application graphique représentée comme une boucle de contrôle (adaptée de [110])

contexte et publient des événements en conséquence. Les opérateurs s'abonnent aux événements envoyés par les sources ou par d'autres opérateurs et envoient des événements. Les arêtes du graphe représentent la relation entre émetteur et récepteur d'un événement. Le graphe contient aussi des *applications* qui s'abonnent et reçoivent des événements. Les sources se placent dans la couche de capture du paradigme SCC alors que les opérateurs se placent dans la couche de raffinement. Les applications se placent dans la couche de contrôle car elles reçoivent des données et non des ordres. La couche d'action du paradigme SCC n'a pas de représentant dans cette approche car les auteurs se sont surtout intéressés au traitement des données de l'environnement.

Ces travaux sont très proches du paradigme SCC, ce qui rend ce paradigme particulièrement adapté pour décrire des applications adaptables au contexte.

1.2.1.2 *Systèmes autonomiques*

Un système autonomique est un système qui assure sa configuration, sa réparation, son optimisation et sa protection [45]. Une approche fréquemment utilisée pour réaliser un système autonomique consiste à le construire autour d'une boucle de contrôle (*feedback loop*) [26, 57, 110]. Une boucle de contrôle utilise des capteurs pour récupérer l'état du système. Elle analyse cet état, planifie des changements à opérer, et agit sur le système.

Au-delà des systèmes autonomiques, ce genre de boucle de contrôle est omniprésent dans les systèmes logiciels. Van Roy [110] présente par exemple une application graphique réalisée comme une boucle de contrôle autour de l'utilisateur (voir Figure 3).

Le nombre et les noms des éléments de la boucle varient suivant les travaux. Par exemple, IBM [57] utilise des capteurs, un surveillant, un analyseur, un planificateur, un exécuteur et des effecteurs. Cependant il n'y a pas de consensus sur le contenu exact de la boucle. Van Roy [110] utilise par exemple un agent

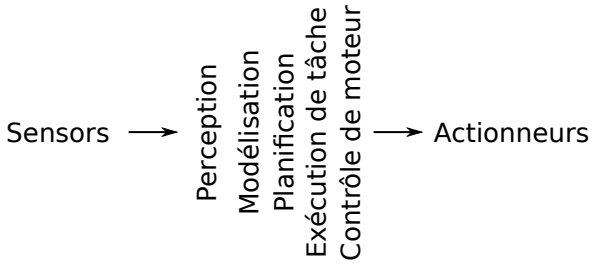

FIGURE 4: La décomposition traditionnelle d'un système de contrôle de robot mobile en modules fonctionnels (adaptée de [13])

de surveillance, un calculateur d'actions correctives et un agent d'actions. De leur côté, Dobson et al. [26] utilisent des noms d'actions dans leur boucle : collecter, analyser, décider et agir. Malgré leurs différences, ces boucles sont issues de concepts très proches. La boucle du paradigme SCC présentée dans la Figure 2 met en œuvre, elle aussi, des concepts très similaires. Le paradigme SCC peut donc prendre en compte les besoins des systèmes autonomiques.

1.2.1.3 *Capture–Planification–Action en robotique*

Depuis la fin des années 1960, le paradigme *sense-plan-act* (SPA) est utilisé pour décomposer les éléments fonctionnels des robots en trois sous-systèmes [102, Chapitre 8] :

- Le sous-système de capture (*sense*) récupère l'état du robot et de l'environnement à proximité grâce à des capteurs.
- Cet état de l'environnement est transmis au sous-système de planification (*plan*) qui en crée un modèle et définit une série d'actions visant à atteindre un but.
- Le sous-système d'action (*act*) envoie la série d'actions à exécuter au robot.

Ce paradigme est associé à un découpage en couches représenté Figure 4.

Comme le remarquent Taylor et al. [106, Chap. 11], les couches du paradigme SPA sont particulièrement proches de celles du paradigme SCC. Les mêmes auteurs discutent aussi d'une différence notable entre les deux paradigmes. Dans le paradigme SPA, le sous-système de planification a pour rôle principal de maintenir un modèle de l'environnement et d'accomplir une planification sophistiquée des actions du robot à venir. D'après les auteurs, ce ne serait pas le cas dans le paradigme SCC. Cependant, il est possible que ceci relève plus d'une utilisation différente des paradigmes que d'une réelle divergence.

Récemment, Edwards et al. [29] ont utilisé le paradigme SCC pour créer des systèmes robotiques qui s'adaptent et s'organisent

d'eux-mêmes comme des systèmes autonomiques. Ceci renforce le constat fait précédemment que le paradigme SCC intègre les concepts du paradigme SPA et de la boucle des systèmes autonomiques.

1.2.1.4 *Modèle–vue–contrôleur*

Le paradigme Modèle–Vue–Contrôleur, ou MVC, est utilisé pour développer des interfaces graphiques utilisateurs ou des applications web [64, 66]. Dans ce paradigme, une application se décompose en trois composants :

MODÈLE Le rôle du modèle est de représenter l'état de l'application. Ce composant est maintenu indépendant des composants vue et contrôleur.

VUE Le rôle de la vue est de représenter graphiquement le modèle pour l'utilisateur. La vue est informée des changements d'état à l'intérieur du modèle par un mécanisme de notifications et se remet à jour en fonction de ceux-ci.

CONTRÔLEUR Le rôle du contrôleur est de prendre en compte les requêtes de l'utilisateur et de mettre à jour le modèle en fonction. Le contrôleur est indépendant de la vue.

Nous pouvons remarquer que le paradigme MVC est bâti autour d'un environnement extérieur correspondant à l'utilisateur et qu'il possède un comportement similaire au paradigme SCC (attente d'un événement, préparation d'un résultat, et rafraîchissement de la vue). De plus, les composants du MVC jouent un rôle proche des couches du paradigme SCC :

- le composant contrôleur du MVC, en prenant en compte les requêtes des utilisateurs, joue le même rôle que la couche de capture ;
- le composant modèle, en représentant l'état de l'application, joue un rôle similaire à la couche de raffinement ;
- le composant vue, en renvoyant les données à l'utilisateur, joue le même rôle que la couche d'action.

Le paradigme SCC est donc capable, dans de nombreux cas, de représenter des applications graphiques en conservant les avantages du paradigme MVC. De même, les résultats des travaux autour des applications adaptables au contexte, des systèmes autonomiques et du paradigme SPA en robotique, sont plus proches des cas particuliers du paradigme SCC que de résultats avec des divergences fondamentales. Ceci nous amène à conclure que le paradigme SCC est fondamental et que son étude doit avoir des conséquences importantes dans de nombreux domaines d'applications. Nous terminons ce chapitre en étudiant les bénéfices d'utilisation du paradigme dans les applications SCC.

1.2.2 *Bénéfices d'utilisation du paradigme*

Un paradigme de programmation est un ensemble de règles, de concepts et de pratiques. Grâce à cela, l'utilisation d'un paradigme de programmation, lors du développement d'une application, permet de guider les architectes et les développeurs. En ce qui concerne le paradigme MVC, par exemple, les développeurs doivent répartir chaque élément de l'application dans un des trois composants (modèle, vue ou contrôleur). Chaque composant possède ses contraintes ce qui limite les risques pour les développeurs de se tromper. L'utilisation d'un paradigme facilite aussi la maintenance car la connaissance de celui-ci donne des indices pour la compréhension de l'application.

Listons pour finir, les bénéfices que l'on peut tirer de l'utilisation du paradigme SCC.

INDÉPENDANCE DU MOYEN DE CAPTURE La couche de capture assure une indépendance entre la façon dont sont récupérées les données et leur utilisation au sein de la logique applicative. Si des changements interviennent dans la manière dont les données doivent être récupérées dans l'environnement, la couche de capture limite les risques de devoir modifier la logique applicative. Cette indépendance facilite aussi la phase de déploiement de l'application. En effet, par des changements limités à la couche de capture il est possible de passer d'un environnement de test ou de simulation à un environnement déployé. De plus, ce passage peut se faire progressivement en introduisant des éléments d'un environnement déployé dans la couche de capture de test.

INDÉPENDANCE DU MOYEN D'ACTION De la même façon, la couche d'action permet à la logique applicative de s'abstraire des détails bas-niveau de l'exécution d'un ordre.

SÉPARATION DU CALCUL ET DE L'UTILISATION DES DONNÉES La couche de raffinement est dédiée au calcul des données alors que la couche de contrôle est dédiée à l'utilisation de ces données. La couche de raffinement est indépendante de la couche de contrôle. Cette séparation permet de réutiliser plus facilement les données produites. Cette séparation simplifie aussi le développement dans chacune des couches.

DÉCOUPAGE POUR UNE DISTRIBUTION FACILITÉE La séparation des couches de capture et des couches d'action de la logique applicative facilite la distribution des éléments de ces couches au plus près de l'environnement.

Alors que ce chapitre présentait le paradigme et son importance, le chapitre suivant explique pourquoi le développement d'applications suivant ce paradigme est difficile.

2

DÉVELOPPEMENT D'UNE APPLICATION SCC

Le développement d'applications SCC est complexe pour plusieurs raisons. Une des raisons principales est le lien avec l'environnement qui ne doit pas interférer avec la logique applicative. Une autre difficulté provient du besoin de sûreté de fonctionnement car la modification d'un environnement physique peut avoir des conséquences irréversibles pour la sécurité des biens ou des personnes.

Dans ce chapitre nous étudions les différents outils, langages et méthodologies permettant le développement d'applications SCC. Chaque section présente un ensemble de solutions existantes et quelques unes de leurs limitations pour faciliter ce développement.

Dans la deuxième partie de cette thèse, nous reviendrons sur toutes ces solutions pour les comparer avec notre approche.

2.1 LANGAGES DE PROGRAMMATION GÉNÉRALISTES

Les langages de programmation généralistes, tels que Java ou Lisp, permettent de créer tout type d'applications. Les applications SCC sont la plupart du temps développées avec ces langages connus des programmeurs.

Cependant, ces langages ne fournissent aucun support particulier pour le paradigme SCC et ne guident donc pas les développeurs.

2.2 BIBLIOTHÈQUES DE CODE

Tous les langages de programmation généralistes fournissent, cependant, des mécanismes pour regrouper des morceaux d'applications dans des bibliothèques de code de façon à pouvoir les réutiliser dans d'autres applications. Ces bibliothèques permettent d'extraire les motifs de code qui se répètent, par exemple lors de l'utilisation d'un paradigme.

Parmi les bibliothèques de code habituellement utilisées pour le développement d'applications SCC, citons celles qui permettent de distribuer du code comme par exemple Java RMI ou JAIN SIP. Java RMI (*Remote Method Invocation*) est une implémentation

Java du modèle RPC (*Remote Procedure Call*) [27]. Ce modèle RPC permet à un programme de faire exécuter du code à un processus distant sans avoir besoin d'implémenter explicitement les détails de cette interaction. SIP (*Session Initiation Protocol*) est un protocole de communication dédié à la téléphonie sur IP à base de messages [93]. JAIN SIP est une interface de programmation standardisée pour gérer les messages SIP.

Les *frameworks* de programmation permettent eux aussi aussi de réutiliser du code. Ils possèdent les deux caractéristiques suivantes qui les distinguent des bibliothèques de code : le *framework* de programmation est responsable du flot de contrôle de l'application et le comportement général du *framework* peut être étendu ou spécialisé suivant les besoins de l'application [96]. Olympus est un *framework* de programmation au-dessus de l'intergiciel Gaia qui fournit des abstractions haut-niveau pour la programmation d'espaces actifs [88]. One.world est un canevas de programmation pour les applications ubiquitaires [51].

Les intergiciels sont d'autres technologies permettant de réutiliser du code. Un intergiciel (ou *middleware*) est un logiciel qui assure la communication entre différentes applications sur un réseau [96]. CORBA et Gaia en sont deux exemples. CORBA (*Common Object Request Broker Architecture*) est un intergiciel générique qui se base sur un langage de description d'interfaces (IDL) pour générer du support pour la communication [7]. Gaia est un intergiciel dédié aux applications ubiquitaires [92] qui repose sur le concept d'espace actif (*active space*).

Les bibliothèques, *frameworks* de programmation et intergiciels sont des technologies nécessaires pour faciliter le développement d'applications dans un même domaine ou suivant un même paradigme. Ces technologies fournissent toutes des abstractions et des fonctionnalités qui rendent le code des applications plus court et donc plus simple à développer et maintenir. Pour couvrir toujours plus les besoins des utilisateurs, ces solutions incorporent plus d'abstractions et de fonctionnalités au fur et à mesure des versions [63]. Ceci les rend, cependant, de plus en plus difficiles à comprendre et donc à utiliser. En effet, les développeurs ont besoin de comprendre en grande partie ces technologies avant de pouvoir les utiliser efficacement.

2.3 EXTENSIONS DE LANGAGES ET LANGAGES DÉDIÉS

Comme nous l'avons vu, les langages de programmations généralistes permettent mais ne facilitent pas le développement d'applications SCC. Les technologies de réutilisation de code que nous venons d'étudier (bibliothèques, intergiciels et *frameworks*) sont

nécessaires mais complexes à mettre en oeuvre. Pour simplifier leur utilisation, des approches comme ContextJ ou ReactiveML étendent des langages généralistes par des constructions dédiées. ContextJ étend Java avec de nouvelles constructions syntaxiques pour permettre aux applications d'adapter leur comportement au contexte en cours d'exécution [20]. ReactiveML est une extension à ML pour programmer des applications réactives [75]. ReactiveML s'inspire du modèle synchrone d'Esterel [8] et permet la définition de processus qui réagissent aux événements.

Certains outils comme JastAdd [31], Silver [111] ou Helvetia [90] permettent de simplifier le développement de nouvelles extensions pour les langages généralistes. Helvetia permet aussi de pouvoir continuer à utiliser les outils existants du langage généraliste (éditeurs, débogueurs, *etc.*) [89].

Les langages dédiés (DSL pour *Domain-Specific Language*) vont plus loin que les extensions de langages en remplaçant complètement les langages généralistes par un langage créé pour un domaine particulier. Par exemple, AmbientTalk [23, 109] propose un langage dédié à l'informatique ambiante avec des primitives particulièrement adaptées aux applications SCC.

Les extensions de langages et les langages dédiés facilitent l'implémentation d'applications SCC en fournissant des constructions dédiées et haut-niveau aux développeurs. Cependant, ces approches ne fournissent pas de solution en ce qui concerne la phase de conception de l'application. Par exemple, ces solutions vont fournir des constructions pour implémenter les interactions entre composants, mais pas pour décrire les interactions autorisées.

2.4 APPROCHES STRUCTURANTES

Pour faciliter le développement d'applications SCC, il est nécessaire que les développeurs soient guidés dans leur utilisation des couches et des différents éléments qui composent ces couches. Les approches *Context Toolkit* [24] et *Operator Graph* [16] étudiées précédemment (Section 1.2.1, page 8) font partie de ces approches qui fournissent un support pour structurer les applications SCC.

En proposant des types de composants bien distincts, avec leurs caractéristiques propres, ces approches assurent par construction que les applications respectent bien le paradigme SCC. Ces approches mélangent, cependant, les concepts architecturaux (les différents éléments de l'application et leurs interactions) et le code. En mélangeant ces concepts, il devient plus difficile de raisonner sur le logiciel pour vérifier des propriétés de sûreté de fonctionnement par exemple. Il devient aussi plus difficile

de réutiliser les concepts architecturaux dans les phases suivant l'implémentation, comme le test ou le déploiement.

2.5 APPROCHES ARCHITECTURALES

Une approche architecturale repose sur un langage dédié à la description d'architectures (ADL pour *Architecture Description Language*) [1, 78, 80]. Un ADL est un langage qui propose des constructions spécifiques pour décrire les composants d'une architecture, leurs dépendances, ainsi que leurs configurations à l'exécution. Parmi les nombreux ADLs existants, citons C2 qui propose un paradigme architectural en couches pour les interfaces graphiques [105]. UML (*Unified Modelling Language*) dans sa version 2.0 [12] peut aussi être considéré comme un ADL d'après Medvidovic et al. [80]. Même si UML est un langage de description généraliste, il est possible d'en dériver des versions dédiées au paradigme SCC en utilisant des ensembles d'attributs et de contraintes, appelés *profils*, qui peuvent être appliqués aux éléments d'UML pour leur donner une sémantique particulière.

Les approches architecturales sont particulièrement utiles pour avoir une représentation haut-niveau d'une application indépendemment de son implémentation. Malheureusement, ces travaux apportent peu de réponses concernant la phase d'implémentation qui suit la réalisation de l'architecture [100]. Un des problèmes les plus importants est la vérification de la conformité d'une implémentation par rapport à une architecture [18]. Certaines approches permettent cela comme ArchJava [2], ComponentJ [97], ACOEL [104] et Archface [108]. Les trois premières ajoutent des constructions architecturales à Java et la quatrième utilise la programmation par aspects (AOP) pour définir son propre langage de description d'architectures. Cependant, comme pour les langages généralistes, ces approches ne sont pas dédiées au paradigme SCC ce qui limite le support apporté aux architectes et développeurs.

2.6 INGÉNIERIE DES MODÈLES

L'ingénierie dirigée par les modèles (IDM) est une approche de génie logiciel visant à générer tout ou partie d'une application à partir de *modèles*. Un modèle est *"une simplification d'un système, construite avec un but délibéré en tête"* (traduction de la définition de Bézivin et Gerbé [11]). La génération de l'application à partir d'un ou plusieurs modèles se fait en général en plusieurs étapes de transformation : d'un modèle très haut-niveau à un modèle très proche du code. L'*Object Management Group* (OMG) propose une vision de l'IDM, nommé *Model-Driven Architecture* (MDA [77]),

qui repose sur des standards tels que UML, OCL, QVT et MOF. Le but principal de l'approche IDM est de masquer la complexité du développement d'applications en proposant des technologies qui combinent des langages de modélisation dédiés, des outils de transformation de modèles et des générateurs de code [95].

Serral et al. [99] proposent d'utiliser l'IDM dans le cadre de l'informatique ubiquitaire. Leurs travaux reposent sur l'utilisation des diagrammes UML associés à OCL ainsi que sur une stratégie de compilation vers Java et OWL (*Ontology Web Language*). Dans ce cadre, les auteurs proposent d'utiliser des modèles pour définir toutes les caractéristiques d'une application ubiquitaire : ces caractéristiques incluent entre autres les services offerts par l'application, les pièces du bâtiment dans lequel l'application va être déployée ainsi que les politiques d'utilisation des services par les utilisateurs.

Ces approches basées sur l'IDM facilitent le développement d'applications SCC en adaptant des langages de modélisation généralistes au paradigme et en fournissant des outils pour manipuler les modèles produits. Cependant, utiliser ces approches requiert une expertise non négligeable dans toutes les technologies sous-jacentes (les diagrammes UML, OCL, QVT, *etc.*) : ces technologies, notamment UML, étant considérées comme de plus en plus démesurées, ambiguës et peu maniables [35, 86, 107]. De plus, en remplaçant totalement les langages de programmation généralistes par des diagrammes, le gain en abstraction se fait difficilement ressentir et le résultat n'est pas forcément plus simple [73]. Les développeurs sont ainsi privés des outils qu'ils connaissent, maîtrisent et qui ont également prouvé leur efficacité [36].

Nous avons vu dans ce chapitre que le développement d'applications SCC est possible avec les approches existantes mais n'est pas complètement facilité par celles-ci. Dans cette thèse nous proposons une approche qui permet de décrire et d'implémenter une application SCC avec des outils dédiés en réutilisant les outils et langages de programmation que les développeurs connaissent bien.

Deuxième partie

APPROCHE PROPOSÉE

3

VUE D'ENSEMBLE DE L'APPROCHE

Ce chapitre présente une vue d'ensemble du travail de cette thèse. Nous en listons tout d'abord les contributions puis en offrons une présentation générale.

3.1 CONTRIBUTIONS

Les contributions principales de cette thèse sont les suivantes.

UN LANGAGE DÉDIÉ AU PARADIGME SCC Cette thèse introduit DiaSpec, un langage de description d'architectures dédié au paradigme SCC (Chapitre 4). Ce langage est divisé en deux parties. La première partie permet de décrire une taxonomie de composants agissant dans les couches de capture et d'action. La deuxième partie permet de décrire l'architecture d'une application SCC, avec les composants des couches de raffinement et de contrôle. Ce langage fournit un cadre de travail pour guider le développement d'une application SCC en assignant des rôles aux membres du projet et en proposant une séparation des préoccupations. DiaSpec augmente la culture commune (le vocabulaire, les spécifications, le code, *etc.*) qui peut être partagée et réutilisée entre les membres du projet.

UN SUPPORT DE PROGRAMMATION DÉDIÉ À partir de descriptions écrites en DiaSpec, un *framework* de programmation dédié est généré (Chapitre 5). Ce *framework* de programmation guide l'implémentation d'une application SCC et remonte le niveau d'abstraction de cette implémentation grâce à des mécanismes haut-niveau. Ces mécanismes permettent de faire interagir les différents composants de l'application en respectant les contraintes énoncées dans la description de l'architecture. Ce *framework* de programmation est conçu pour être auto-documenté, nécessitant peu d'apprentissage préalable. Loin d'empêcher un développement itératif, notre approche générative en facilite l'usage en maintenant à tout moment une cohérence forte entre la description architecturale et l'implémentation (Chapitre 6).

DES ANALYSES À PLUSIEURS NIVEAUX L'approche présentée dans cette thèse permet à des outils d'analyse de s'appliquer

à l'architecture de l'application, son implémentation et son exécution (Chapitre 7). Grâce au langage dédié DiaSpec et à ses abstractions haut-niveau, des analyses peuvent être conduites automatiquement sur la description de l'architecture des applications. Le *framework* de programmation généré permet d'assurer que les analyses conduites sur l'architecture sont toujours valables au niveau de l'implémentation. Enfin, des analyses sont conduites automatiquement pendant l'exécution de l'application pour garantir sa cohérence à tout moment.

Ces contributions sont évaluées suivant trois aspects complémentaires (Chapitre 8) : (1) l'*expressivité*, évaluant la portée du paradigme SCC et du langage DiaSpec, (2) l'*utilisabilité*, estimant la facilité d'utilisation des outils et (3) la *productivité*, mesurant le temps de développement, la qualité du code ainsi que la capacité à réutiliser le code. Enfin, l'approche est présentée en tant que plateforme de recherche autour de laquelle de nombreux travaux gravitent (Chapitre 9).

3.2 PRÉSENTATION DE L'APPROCHE

Nous avons vu précédemment que le paradigme SCC est constitué de quatre couches : la couche de capture, la couche de raffinement, la couche de contrôle et la couche d'action. Parmi ces quatre couches, la couche de capture et la couche d'action jouent le rôle d'interface avec l'environnement, composante essentielle des applications SCC. Les couches de raffinement et de contrôle contiennent, quant à elles, la logique de l'application SCC. L'interface avec l'environnement et la logique de l'application sont deux éléments fondamentaux de notre approche et forment donc deux parties distinctes du langage de description d'architectures DiaSpec (Figure 5).

INTERFACE AVEC L'ENVIRONNEMENT L'interface avec l'environnement est décrite à l'aide d'une *taxonomie d'entités*. Chaque entité possède deux facettes : une facette *source* permettant d'envoyer des données capturées dans la couche de raffinement, et une facette *action* permettant de répondre à des ordres venant de la couche de contrôle. Les experts de domaines sont responsables de la spécification de la taxonomie ①. Les entités sont décrites de façon indépendante afin d'être facilement réutilisées dans plusieurs applications.

LOGIQUE DE L'APPLICATION La logique de l'application est décrite dans une architecture. L'architecture spécifie deux types de composants, les *opérateurs de contexte* et les *opérateurs de*

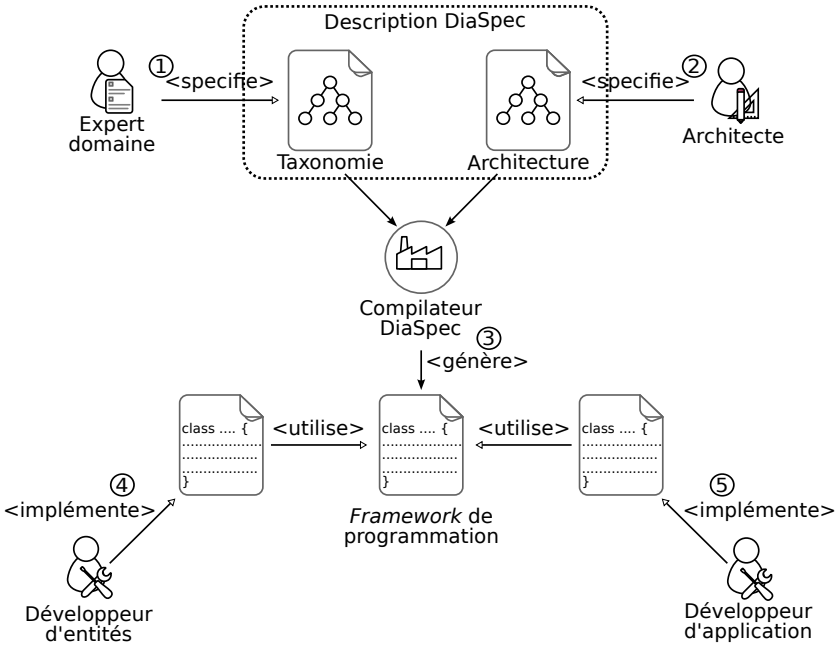

FIGURE 5: Développement d'une application avec notre approche

contrôle. Les opérateurs de contexte traitent les données envoyées par les entités et se placent dans la couche de raffinement. Les opérateurs de contrôle utilisent les données provenant des opérateurs de contexte et agissent sur les entités. Ils se placent dans la couche de contrôle. Les architectes sont responsables de la spécification de l'architecture de l'application ②.

À partir d'une description en DiaSpec, un compilateur génère un *framework* de programmation dans un langage généraliste ③. Ce *framework* de programmation contient une *classe abstraite* pour chaque déclaration de composant DiaSpec (entité ou opérateur). Cette classe abstraite contient des méthodes générées pour guider l'implémentation du composant associé. Cette classe abstraite possède aussi des déclarations de méthodes abstraites pour permettre aux développeurs d'entités ④ et d'applications ⑤ d'implémenter la logique de chaque composant. Implémenter un composant se fait en créant une sous-classe de la classe générée associée. Avec cette approche, les développeurs sont amenés à implémenter toutes les méthodes abstraites de la super-classe générée. Les développeurs ne modifient jamais les fichiers générés. Ainsi, les fichiers peuvent être générés à nouveau si la description de l'architecture ou de la taxonomie change sans impacter le code écrit manuellement.

Pour assurer les contraintes architecturales au niveau de l'implémentation, le *framework* de programmation généré utilise le typage statique fourni par le langage généraliste sur lequel il repose. De cette façon, une implémentation est toujours conforme à son architecture. Par exemple, une des propriétés vérifiables

sur l'implémentation est l'« intégrité de communication » : cette propriété indique que chaque composant doit seulement communiquer avec des composants auxquels il est connecté dans l'architecture. Cette propriétés est vérifiée automatiquement par la génération, dans la classe abstraite du composant, d'une méthode pour chaque interaction possible : une implémentation de composant n'a pas d'autre moyen pour interagir avec d'autres composants que de passer par les méthodes générées. De cette façon, toute implémentation correcte vis-à-vis du compilateur du langage généraliste est correcte vis-à-vis de la description de l'architecture. De plus, pour assurer la cohérence globale de l'application, des analyses à l'exécution sont effectuées automatiquement par le *framework* de programmation.

4

DESCRIPTION D'UNE APPLICATION SCC

Domain-specific languages lift the platform's level, reduce the underlying APIs' surface area, and let knowledgeable end users live in their data without complex software-centric models and the API field of dreams.

— Dave Thomas [107]

Dans ce chapitre, nous présentons le langage DiaSpec dédié à la description d'applications SCC. Nous décrivons tout d'abord les différents éléments architecturaux présents dans le langage, puis nous proposons un formalisme permettant de décrire les interactions entre ces éléments.

4.1 PRÉSENTATION GÉNÉRALE DU LANGAGE

Nous décrivons ici les principales caractéristiques du langage DiaSpec. En particulier, nous étudions ce qui en fait un langage dédié ainsi qu'un langage de description d'architectures. Enfin, nous présentons brièvement l'exemple d'application SCC qui servira de fil conducteur dans toute cette partie de la thèse.

4.1.1 *Un langage dédié*

Nous avons vu au chapitre 1 que le paradigme SCC sépare les applications en quatre couches : la couche de capture, de raffinement, de contrôle et d'action (Figure 1, page 4). Les éléments de chacune de ces couches ont des contraintes spécifiques en terme d'interactions et de déploiement. Le langage DiaSpec est un langage dédié au paradigme SCC dans le sens où il fournit des concepts haut-niveau directement liés au paradigme SCC. Plus précisément, DiaSpec propose des éléments architecturaux différents pour chacune des couches du paradigme :

COUCHE DE CAPTURE Une *source* capte des données de l'environnement ;

COUCHE DE RAFFINEMENT Un *opérateur de contexte* transforme les données fournies par les sources et par d'autres opérateurs de contexte pour produire des données de niveau applicatif ;

COUCHE DE CONTRÔLE Un *opérateur de contrôle* utilise les données venant de la couche de raffinement pour faire exécuter des ordres ;

COUCHE D'ACTION Une *action* propose d'exécuter les ordres de la couche de contrôle sur l'environnement.

Il arrive souvent qu'un même élément physique ou logiciel soit à la fois capable de fournir des données et de répondre à des actions. Par exemple, une caméra envoie son flux vidéo (une source) et son angle de vision peut être changé (une action). Pour rendre compte de ce fait, DiaSpec possède la notion d'*entité*. Une entité regroupe un ensemble de sources et d'actions. Ce regroupement est similaire à celui fait par Dey et al. [24] dans *Context Toolkit* où les *services* (équivalent des actions) sont incorporés dans les *widgets* (équivalent des sources). On dit qu'une entité possède deux facettes, une dans la couche de capture qui envoie des informations et une dans la couche d'action qui lui permet d'être manipulée. Bien évidemment, certaines entités ne se servent que d'une des deux facettes en ne proposant soit que des sources, soit que des actions.

4.1.2 *Un langage architectural*

Le langage DiaSpec permet de décrire les éléments architecturaux d'une application SCC en termes d'entités et d'opérateurs principalement et de leurs dépendances. En ce sens, DiaSpec est un langage de description d'architectures. Tel un ADL, DiaSpec ne permet pas de définir le comportement des éléments architecturaux décrits. La définition du comportement est laissée à d'autres outils tels que les langages de programmation. DiaSpec est donc un langage de description d'architectures dédié au paradigme SCC.

Comme nous l'avons vu, le paradigme SCC est composé de quatre couches, réparties en deux ensembles : les couches de capture et d'action qui gèrent l'interaction avec l'environnement et les couches de raffinement et de contrôle qui gèrent la logique applicative. Le langage DiaSpec est divisé en deux parties pour refléter ces deux ensembles : une partie *taxonomie* qui permet de décrire les entités, et donc le lien avec l'environnement, et une partie *architecture* qui permet de décrire la logique applicative. La Figure 6 présente le lien entre ces deux parties et les différents

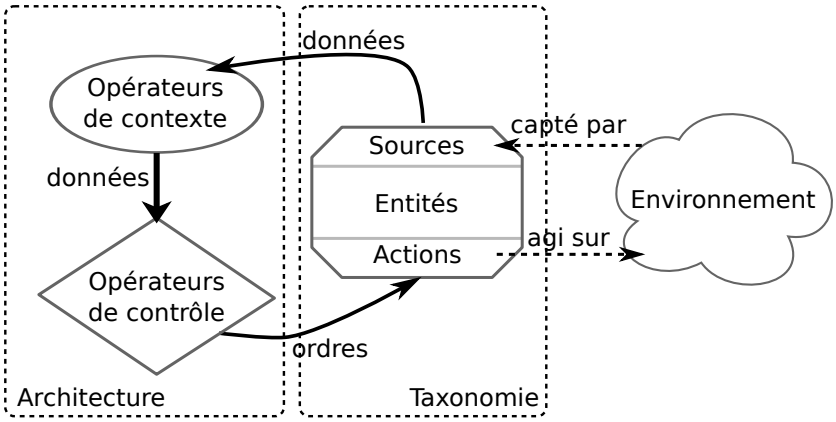

FIGURE 6: Style architectural d'une application SCC

éléments architecturaux. Cette figure est particulièrement semblable à la Figure 2 (page 5) ce qui montre le lien fort entre les éléments architecturaux et les couches du paradigme SCC. Les deux sections suivantes (4.2 et 4.3) décrivent ces deux parties en détail.

Pour illustrer cette thèse, nous utilisons l'exemple d'un moniteur de serveur web d'intranet. Dans cette application SCC, l'environnement est constitué d'un serveur web et d'outils réseaux. Les deux tâches que nous souhaitons implémenter sont (1) la mise à jour d'un journal de profils des utilisateurs du serveur web (nom et adresse IP de l'utilisateur) et (2) l'envoi de messages aux administrateurs système en cas d'intrusion.

4.2 TAXONOMIE

Le langage DiaSpec est donc composé d'une partie que l'on appelle taxonomie. Une taxonomie DiaSpec est un ensemble de descriptions de classes d'entités. Les classes d'entités sont indépendantes les unes des autres pour permettre leur réutilisation au sein de descriptions d'autres applications SCC.

4.2.1 *Les entités*

Comme nous l'avons vu, une classe d'entités décrit un ensemble de *sources* et d'*actions*. Une classe d'entités décrit aussi un ensemble d'*attributs*.

SOURCE Une entité produit des données pour l'application en capturant des informations de l'environnement. Pour chaque donnée que l'entité peut capturer, l'expert de domaine associe à la classe de l'entité une source avec un nom et un type. Par exemple,

dans le moniteur de serveur web, un capteur est nécessaire pour détecter les connexions de client sur le serveur.

ACTION Une entité fournit aussi à l'application des moyens d'agir sur l'environnement. L'expert de domaine associe à la classe de l'entité des ensembles de déclarations de méthodes représentant les différents moyens d'interaction avec l'environnement. Chaque déclaration de méthode contient un nom et des paramètres typés. Par exemple, une entité doit être capable d'envoyer des messages aux administrateurs système du serveur web en cas d'intrusion. Une déclaration de méthode n'a pas de type de retour car ceci fournirait un moyen détourné d'envoyer des valeurs à l'application et ceci est une tâche réservée aux sources.

ATTRIBUT Pour distinguer les instances d'une même classe d'entités, on associe à chaque classe des attributs. Un attribut a un nom et un type. Par exemple, un attribut de type booléen peut indiquer si une entité est en fonction ou pas. Un autre attribut possible pourrait fournir le coût d'utilisation de l'entité : ceci permettrait, dans le cas du moniteur de serveur web, de choisir le type de message à envoyer aux administrateurs (*email* ou SMS) en fonction de l'importance du message et du coût de l'envoi.

Comme dans une hiérarchie de classes d'un langage de programmation objets, une classe d'entités peut hériter d'une autre classe d'entités pour en faire une version plus spécifique. La classe plus spécifique hérite des sources, actions et attributs de sa super-classe et peut en ajouter de nouveaux. Dans l'exemple du moniteur de serveur web, la plupart des entités peuvent être démarrées et arrêtées : une action permettant ceci peut donc être associée à une classe d'entités, et il suffit ensuite de faire hériter les autres classes de celle-ci.

La gestion du cycle de vie des composants se retrouve dans de nombreuses plateformes à composants, telles que OSGi [84]

4.2.2 *Application à l'exemple*

Nous décrivons ici les classes d'entités nécessaires au moniteur de serveur web ainsi que la syntaxe du langage DiaSpec permettant de décrire ces classes. Dans cet exemple d'application, aucune entité ne possède à la fois des sources et des actions. Nous verrons plus tard que c'est un cas particulier et que le regroupement de sources et d'actions au sein d'une même entité est parfois utile.

La grammaire complète du langage DiaSpec est en Annexe B

LES SOURCES Les serveurs web tracent les accès qui leur sont faits dans un fichier journal (une ligne dans le journal représen-

tant un accès). Par exemple, le serveur Lighttpd [1] trace chaque accès dans le fichier `/var/log/lighttpd/access.log`. Chaque ligne contient des informations sur l'accès comme l'adresse IP du client, l'heure de l'accès, la page demandée et le modèle du navigateur utilisé. L'ajout d'une ligne dans ce journal est détecté par l'entité `AccessLogReader` qui transmet la ligne au travers de sa source `line`. L'entité `NSLookup` représente l'outil réseau du même nom qui associe une adresse IP à un nom de machine en interrogeant des serveurs de noms de domaines (DNS). Cette association se fait au travers de la source `ip2host`. Enfin, l'entité `LDAPServer` fait une requête sur un annuaire d'entreprise LDAP pour associer un nom de machine à un profil utilisateur. Cette association se fait au travers de la source `host2profile`.

Dans le cas de notre moniteur de serveur web intranet, toutes les adresses IP sont dans un réseau local

LES ACTIONS Pour pouvoir envoyer des messages aux administrateurs système en cas d'intrusion, l'entité `Mailer` fournit une action `Mail`. L'action `Mail` définit une seule méthode, appelée `send()`, qui possède trois paramètres permettant d'indiquer le destinataire, le titre ainsi que le contenu du message. L'entité `Logger` a pour charge de mettre à jour le journal. Elle utilise l'action `Log` qui définit la méthode `log()`. Cette méthode prend en paramètre un message à enregistrer dans le journal ainsi que le niveau d'importance de ce message.

Le Listing 1 présente la taxonomie complète du moniteur de serveur web écrite en DiaSpec. Entre les lignes 1 et 4, la classe d'entités `StoppableDevice` est définie. Le mot-clé `device` permet de définir des entités, qu'elles soient logicielles ou matérielles. Ce mot-clé est suivi du nom de la classe d'entités. La classe d'entités `StoppableDevice` est définie comme abstraite (mot-clé `abstract`) pour indiquer qu'il n'est pas possible d'en créer des instances. À la ligne 2, le mot-clé `attribute` est utilisé pour ajouter un attribut `isStarted` de type booléen. À la ligne 3, le mot-clé `action` associe l'ensemble d'actions `StartStop` à l'entité. Cet ensemble d'actions est défini entre les lignes 6 et 8 : il contient une action `start` et une action `stop`, toutes les deux sans paramètre. Entre les lignes 10 et 12, la classe d'entités `AccessLogReader` est définie comme héritant de la classe d'entités `StoppableDevice` : elle hérite donc de l'ensemble d'actions `StartStop` ainsi que l'attribut `isStarted`. À la ligne 11, une source est ajoutée : cette source est appelée `line` et est de type chaîne de caractères. Une source peut être liée à des données supplémentaires que nous nommerons indices : voir l'exemple à la ligne 15 où la donnée de type chaîne de caractères est liée à une adresse IP. Nous étudierons l'utilité de cette construction dans la Section 4.4. DiaSpec permet aussi de définir des types de données. Le type de données `IPAddress`

1. http://www.lighttpd.net/

```
abstract device StoppableDevice {
  attribute isStarted as Boolean;
  action StartStop;
}

action StartStop {
  start(); stop();
}

device AccessLogReader extends StoppableDevice {
  source line as String;
}

device NSLookup {
  source ip2host as String indexed by ip as IPAddress;
}

structure IPAddress {
  address as String;
}

device LDAPServer {
  source host2profile as Profile indexed by host as String;
}

structure Profile {
  name as String;
  ip as IPAddress;
}

device Logger extends StoppableDevice {
  attribute level as Level;
  action Log;
}

action Log {
  log(message as String, level as Level);
}

enumeration Level {ALL, INFO, DEBUG, ERROR, NONE}

device Mailer extends StoppableDevice {
  action Mail;
}

action Mail {
  send(to as String, title as String, content as String);
}
```

Listing 1: Taxonomie complète du moniteur de serveur web

est défini entre les lignes 18 et 20 et est composé d'un champ `address` de type chaîne de caractères. La ligne 40 définit une énumération permettant d'indiquer le niveau d'importance d'un message. Cette énumération est utilisée au niveau de la définition de la méthode `log` de l'ensemble d'actions `Log` (ligne 37) ainsi que pour l'attribut `level` de la classe d'entités `Logger` (ligne 32).

Une taxonomie décrit donc des classes d'entités, liées potentiellement par une relation d'héritage. Une taxonomie contient aussi des définitions de types de données. DiaSpec fournit les types de base habituellement rencontrés dans les langages de programmation comme les booléens, les entiers, les flottants, les chaînes de caractères et les tableaux. DiaSpec fournit aussi la possibilité de créer ses propres types de données qui peuvent être soit des *structures* soit des *énumérations*. Comme dans beaucoup de langages de programmation, une structure est un ensemble de champs nommés et typés alors qu'une énumération est un ensemble de noms.

Sauf dans le cas de l'héritage, les entités sont décrites de façon indépendante les unes des autres. Les entités sont aussi indépendantes de la logique applicative. Grâce à cette indépendance, les entités peuvent être placées dans des catalogues d'entités pour faciliter leur réutilisation dans d'autres applications. De plus, cette description indépendante permet aux entités d'être déployées dans un système et d'ajouter *a posteriori* à ce système une ou plusieurs applications qui les utilisent.

4.3 ARCHITECTURE

L'architecture de l'application est composée des opérateurs de contexte et des opérateurs de contrôle. L'architecture est construite au dessus de la taxonomie : les opérateurs de contexte dépendent des sources des entités et les opérateurs de contrôle dépendent des actions des entités.

4.3.1 *Les opérateurs*

La Figure 7 représente le modèle sémantique simplifié des composants de DiaSpec. Dans ce diagramme, les flèches représentent la dépendance d'un élément à un autre. Les entités sont indépendantes les unes des autres et des autres types de composants pour maximiser leur réutilisabilité et pour proposer des bibliothèques d'entités (entités sur étagère). Les opérateurs de contexte dépendent des données qu'ils utilisent (provenant de sources d'entités ou d'autres opérateurs de contexte). Les opérateurs de contrôle dépendent des opérateurs de contexte qu'ils utilisent

Le terme « modèle sémantique » dans le contexte des DSLs est défini par Fowler [39, Chap. 11]

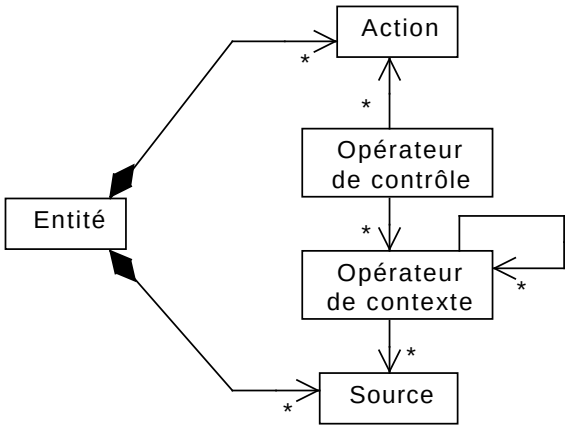

FIGURE 7: Modèle sémantique UML simplifié des composants de DiaSpec

et des actions d'entité nécessaires à la modification de l'environnement. Les données circulent des sources vers les opérateurs de contexte et des opérateurs de contexte vers les opérateurs de contrôle.

Contrairement à la taxonomie, la description d'un opérateur dans l'architecture ne décrit pas une classe mais un seul opérateur. En effet, le rôle d'un opérateur étant simplement de raffiner des données, il n'est pas nécessaire d'avoir ce même raffinement déployé plusieurs fois. [2]

OPÉRATEUR DE CONTEXTE Un opérateur de contexte construit une donnée en fonction de données fournies par les sources des entités et d'autres opérateurs de contexte. L'architecte de l'application associe donc à chaque opérateur de contexte un nom, un type de données, un ensemble de sources d'entités et un ensemble d'autres opérateurs de contexte. Dans le moniteur de serveur web, par exemple, un opérateur de contexte prend en entrée une ligne sous la forme d'une chaîne de caractères fournie par l'entité `AccessLogReader` et, produit en sortie une structure de donnée contenant l'adresse IP et le navigateur du client.

OPÉRATEUR DE CONTRÔLE Un opérateur de contrôle fait exécuter des actions à des entités en fonction de données d'entrées fournies par des opérateurs de contexte. Dans le moniteur de serveur web, par exemple, un opérateur de contrôle reçoit des

2. Pour une application où la sûreté de fonctionnement est importante certains opérateurs pourraient être répliqués, mais dans une première approche l'architecte n'a pas à considérer ce besoin.

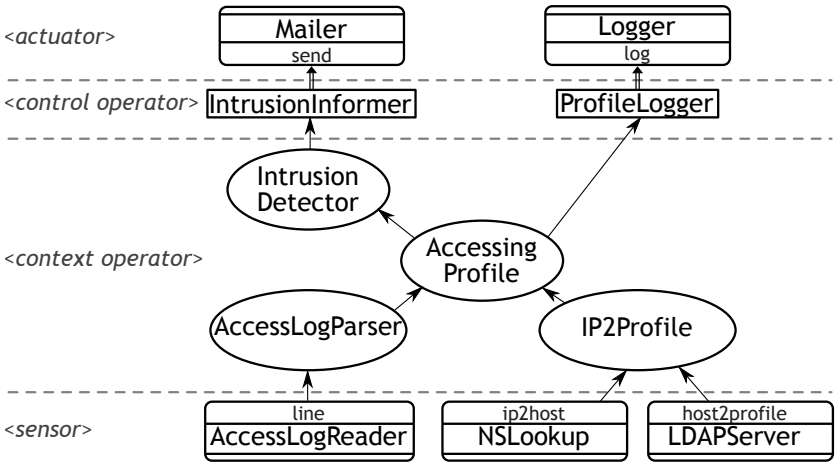

FIGURE 8: Représentation graphique du moniteur de serveur web

notifications d'intrusion et décide de faire envoyer des messages aux administrateurs système.

4.3.2 *Application à l'exemple*

La Figure 8 représente les composants du moniteur de serveur web dans les quatre couches du paradigme. La figure représente le flot de données entre ces composants avec une flèche simple, et le flot d'ordres avec une flèche double. Les paragraphes qui suivent présentent ces composants en détail. Les entités sont représentées par des rectangles aux coins ronds, les opérateurs de contexte par des ellipses, et les opérateur de contrôle par des losanges.

LES OPÉRATEURS DE CONTEXTE Au centre de la Figure 8, l'opérateur de contexte `AccessingProfile` calcule le profil de l'utilisateur qui vient de se connecter au serveur web. Cet opérateur est activé par l'opérateur de contexte `AccessLogParser` qui est lui-même activé par la source `line` de l'entité `AccessLogReader`. Quand une nouvelle ligne est ajoutée dans le journal du serveur web, l'opérateur `AccessLogParser` transforme la chaîne de caractères en une structure de données plus facilement utilisable grâce à des expressions rationnelles (*regular expressions*). Cette structure contient entre autres l'adresse IP de l'utilisateur et le navigateur utilisé. Cette information est envoyée à l'opérateur `AccessingProfile`, qui extrait l'adresse IP de la structure et demande à l'opérateur de contexte `IP2Profile` de l'associer à un profil utilisateur. Ce profil est obtenu en demandant aux sources des entités `NSLookup` et `LDAPServer`. L'opérateur `IntrusionDetector` utilise les données calculées par l'opérateur `AccessingProfile` et propage ces données si elles sont suspectées

de représenter des intrusions : par exemple, si `IP2Profile` n'a pas été capable d'associer un profil à une adresse IP, la donnée calculée par `AccessingProfile` sera incomplète ce qui peut être le signe d'une intrusion.

LES OPÉRATEURS DE CONTRÔLE Le moniteur de serveur web a deux tâches : la mise à jour d'un journal de profils des utilisateurs ainsi que la gestion des intrusions. Ces deux tâches sont respectivement déléguées à deux opérateurs de contrôle `ProfileLogger` et `IntrusionInformer`, qui invoquent les entités `Logger` et `Mailer` en utilisant les actions `Log` et `Mail`. Pour mettre à jour le journal de profils utilisateurs, l'opérateur `ProfileLogger` a besoin d'être informé des profils utilisateurs qui accèdent au serveur web. Pour avertir les administrateurs système d'une intrusion potentielle, l'opérateur `IntrusionInformer` a besoin de se voir notifier des intrusions par l'opérateur `IntrusionDetector`.

Le Listing 2 présente l'architecture complète du moniteur de serveur web décrite dans le langage DiaSpec. Le mot-clé `include` permet de faire référence à des éléments définis à l'intérieur d'un autre fichier. Par exemple, la ligne 1 permet à l'architecture du moniteur de faire référence à la taxonomie définie au Listing 1. Entre les lignes 3 et 6, l'opérateur de contexte `IP2Profile` est défini. Cet opérateur utilise deux sources (`ip2host` et `host-2profile`) venant de deux entités (`NSLookup` et `LDAPServer`). Entre les lignes 12 et 15 l'opérateur de contexte `AccessingProfile` est défini. Celui-ci dépend de deux autres opérateurs de contexte qui sont `AccessLogParser` et `IP2Profile`. Les lignes 21 à 24 définissent l'opérateur de contrôle `ProfileLogger`. Cet opérateur dépend de données venant de l'opérateur de contexte `Accessing-Profile`. La ligne 23 associe l'ensemble d'actions `Log` venant de l'entité `Logger` à cet opérateur de contrôle.

4.3.3 *Support à l'utilisation de DiaSpec*

Pour faciliter l'écriture de la taxonomie et de l'architecture, une extension à l'environnement de développement Eclipse [3] a été développée. Cette extension, basée sur XText [4], apporte des fonctions comme la coloration syntaxique, la complétion de code et les modèles de code (*template*) (Figure 9). L'extension est aussi capable de représenter graphiquement une architecture décrite en DiaSpec (Figure 10).

3. http://eclipse.org/
4. http://www.eclipse.org/Xtext/

```
include "taxonomie.diaspec";

context IP2Profile as Profile indexed by ip as IPAddress {
  source ip2host from NSLookup;
  source host2profile from LDAPServer;
}

context AccessLogParser as Access {
  source line from AccessLogReader;
}

context AccessingProfile as IdentifiedAccess {
  context AccessLogParser;
  context IP2Profile;
}

context IntrusionDetector as IdentifiedAccess {
  context AccessingProfile;
}

controller ProfileLogger {
  context AccessingProfile;
  action Log on Logger;
}

controller IntrusionInformer {
  context IntrusionDetector;
  action Mail on Mailer;
}

structure Access {
  host_ip as IPAddress;
  timestamp as String;
  request as String;
  status as Integer;
  line as String;
}

structure IdentifiedAccess {
  access as Access;
  profile as Profile;
}
```

Listing 2: Architecture complète du moniteur de serveur web

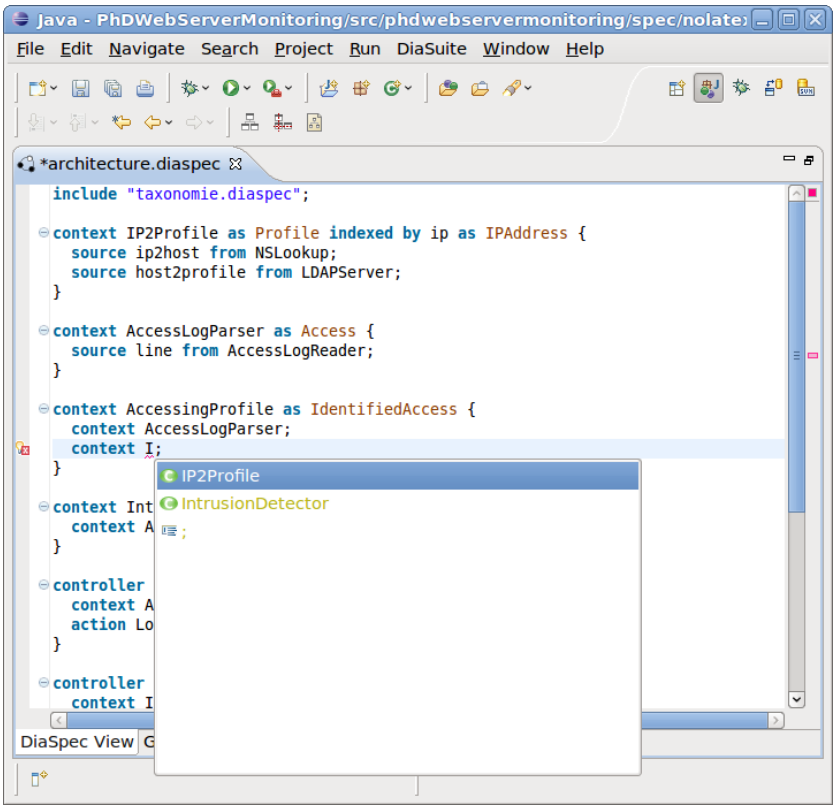

FIGURE 9: Utilisation de l'extension d'Eclipse pour DiaSpec

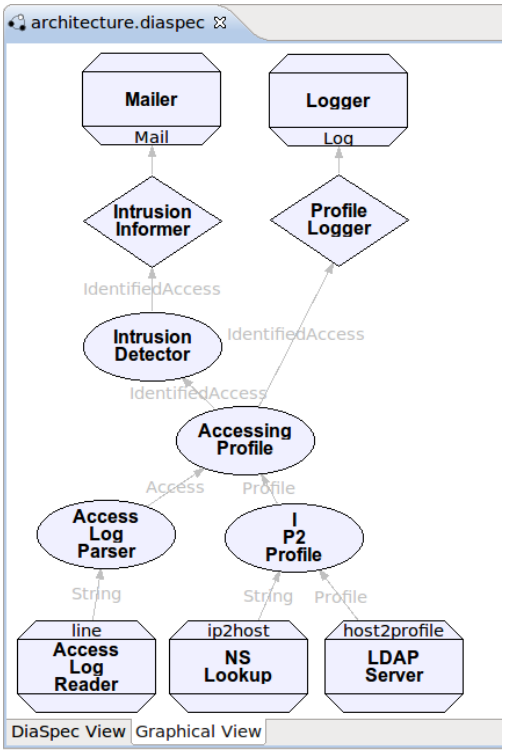

FIGURE 10: Représentation graphique créée automatiquement par l'extension d'Eclipse pour DiaSpec

4.4 INTERACTIONS

Nous avons vu jusqu'à maintenant comment les entités et les opérateurs sont définis au sein de la taxonomie et de l'architecture dans le langage DiaSpec. Nous voyons maintenant comment ces composants communiquent.

4.4.1 *Différents modes*

Il existe trois modes d'interactions en DiaSpec permettant aux composants d'échanger des donnés ou de faire exécuter des actions.

ÉVÉNEMENT Le mode d'interaction événement, ou *push*, permet à un composant d'envoyer une donnée à plusieurs destinataires sans avoir à en dépendre [34]. Les composants qui souhaitent recevoir des événements d'un autre composant doivent s'y abonner.

REQUÊTE Le mode d'interaction requête, ou *pull*, permet à un composant de demander une donnée à un autre composant. Le composant qui effectue une requête doit attendre la réponse avant de pouvoir poursuivre son traitement. Il est possible de passer des arguments à une requête. Ces arguments sont décrits dans la spécification de l'application comme les indices d'une valeur. Par exemple, l'opérateur de contexte `IP2Profile` a besoin d'un paramètre `ip` de type `IPAddress` pour fournir un profil (Listing 2, ligne 3).

COMMANDE Le mode d'interaction commande permet de faire exécuter des actions à une entité. Des paramètres peuvent être associés à une commande. L'appel à une commande est non bloquant et aucune donnée n'est renvoyé au composant qui a demandé l'exécution des actions.

Une source d'entité est soit *proactive*, soit *réactive*, soit les deux. Elle est proactive si elle transmet des données sans y avoir été invitée. Elle est réactive si elle répond à une demande ou à la réception d'une donnée. Un opérateur est toujours réactif : il ne fait rien sans y avoir été invité au préalable par la réception d'une donnée ou d'une demande de donnée. Ces propriétés assurent que chaque application SCC est réactive à l'état de l'environnement : chaque action effectuée sur l'environnement est initiée par l'environnement et suit un ensemble d'interactions des sources d'entités jusqu'aux opérateurs de contrôle.

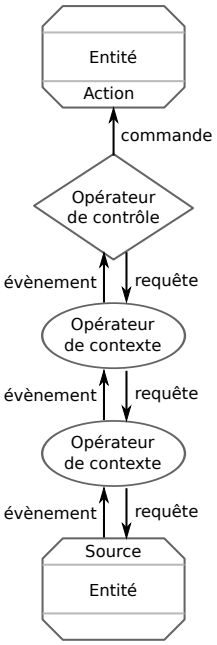

FIGURE 11: Interactions possibles dans DiaSpec. L'origine d'une flèche est du côté de l'initiateur de l'interaction. Les données circulent toujours du bas vers le haut

Les types d'interactions possibles dans le paradigme SCC sont limités. Quand une source est proactive, elle envoie des événements à un ou plusieurs opérateurs de contexte en fonction de ce qu'elle capte de l'environnement. Quand une source est réactive, elle répond à des requêtes d'un opérateur de contexte. Un opérateur de contexte reçoit des événements d'un autre opérateur de contexte ou d'une source. Un opérateur de contexte reçoit des requêtes venant d'un autre opérateur de contexte ou d'un opérateur de contrôle. Un opérateur de contrôle reçoit des événements d'opérateurs de contexte, envoie des requêtes à des opérateurs de contexte et commande des actions sur des entités. La Figure 11 montre les différentes interactions possibles entre types de composants.

ENFANTS ET PARENTS D'UN OPÉRATEUR Dans un graphe où les flèches représentent le flot de données et d'ordres comme celui de la Figure 8, on définit les *enfants* et les *parents* d'un opérateur comme respectivement les origines des flèches qui entrent et les destinations des flèches qui sortent. Dit autrement, les enfants d'un opérateur sont les sources d'entités ou les opérateurs de contexte qui fournissent une donnée à cet opérateur. Les parents d'un opérateur sont les opérateurs ou les actions d'entités qui reçoivent les données de cet opérateur.

Les opérateurs de contexte et de contrôle sont protégés des exécutions concurrentes grâce à des files de messages. Chaque

opérateur possède une file de messages (donnée ou requête) et les messages sont traités un par un par l'opérateur.

4.4.2 *Contrats d'interactions*

Jusqu'à maintenant, nous avons vu que les opérateurs déclarent des dépendances vers d'autres opérateurs ou vers des sources ou actions d'entités. Cette simple déclaration de dépendance ne permet pas à l'architecte d'exprimer complètement le flux de données et d'ordres. En effet, il existe plusieurs modes d'interactions entre les opérateurs (événement et requête) et le choix d'un mode plutôt qu'un autre a un impact sur le flux des données. Par exemple, l'architecture du moniteur de serveur web, telle que représentée dans la Figure 8 et le Listing 2, n'indique pas que l'opérateur de contexte `IP2Profile` ne peut être accédé que par une requête. Ce manque d'information peut conduire à différentes interprétations de la même architecture et donc à différentes implémentations, dont certaines en dehors des souhaits originaux de l'architecte.

Pour pallier ce manque d'information, nous avons conclu que les trois informations suivantes sont nécessaires :

CONDITIONS D'ACTIVATION Les opérateurs étant réactifs, il est nécessaire d'indiquer quelles sont les interactions capables d'activer un opérateur.

DONNÉES REQUISES Pour chaque activation possible d'un opérateur, il est nécessaire d'indiquer les interactions supplémentaires autorisées.

ACTION À ENTREPRENDRE En réponse à chaque activation, il est nécessaire d'indiquer les actions qui pourront être entreprises (émission d'une valeur dans le cas d'un opérateur de contexte ou commande d'une action dans le cas d'un opérateur de contrôle).

Nous déduisons de ces informations que l'on peut associer un ensemble de triplets à chaque opérateur, nommé *contrat d'interactions*. Chaque triplet, nommé *contrat d'interactions basique*, est de la forme $\langle A; U; E \rangle$ où A, U et E sont respectivement la condition d'activation, la liste des données requises et l'action à entreprendre. Ces éléments sont définis comme suit :

- $A = \Uparrow (A_1, \ldots, A_n) \mid \Downarrow self$, où $n > 0$, A_i est le nom d'un enfant de l'opérateur courant ou une disjonction de ces noms. $self$ indique l'opérateur lui-même. $\Uparrow (A_1, \ldots, A_n)$ correspond aux données poussées (*push*) par tous les enfants $A_1, \ldots, A_n$. Si un A_i est une disjonction de noms $(A_{i1} \vee \ldots \vee A_{im})$, alors une donnée venant de n'importe

lequel peut être utilisée. $\Downarrow$ self correspond à une requête d'un des parents. Une requête retourne toujours une donnée au parent appelant. La forme $\Downarrow$ self n'est pas autorisée pour un opérateur de contrôle, ceux-ci ne calculant pas de donnée.
- $U = \Downarrow (B_1, \ldots, B_n)$ où $n \geqslant 0$ et B_i est le nom d'un enfant de l'opérateur courant. Une requête est utilisée pour cette interaction et c'est à la logique applicative de décider d'envoyer ou pas cette requête .
- $E = \Uparrow$ self | $\Uparrow$ self? | $\emptyset$ pour un opérateur de contexte indique respectivement que l'opérateur envoie toujours, parfois ou jamais une nouvelle valeur à tous les parents qui sont abonnés. Quand $A = \Downarrow$ self, une valeur est toujours renvoyée au parent qui a fait la requête, quelque soit E. $E = \Uparrow (C_1, \ldots, C_n)$ pour un opérateur de contrôle, où $n > 0$ et C_i le nom d'une action de classe d'entités, indique que l'opérateur doit exécuter au moins une action parmi $C_1, \ldots, C_n$.

Des vérifications doivent être conduites sur ces contrats d'interactions pour vérifier qu'ils sont valides. Par exemple, lorsqu'un contrat d'interactions basique a pour données requises $U = \Downarrow B$ et que B est un opérateur de contexte, il faut que B ait un contrat d'interactions basique dont la condition d'activation est $A = \Downarrow$ self. Ces vérifications sont étudiées au Chapitre 7.

4.4.3 *Application à l'exemple*

La Table 1 spécifie les contrats d'interactions pour le moniteur de serveur web. Par exemple, le contrat d'interactions de l'opérateur `IntrusionDetector` indique, de par son action à entreprendre $E = \Uparrow$ self?, que quand cet opérateur reçoit une nouvelle donnée venant de `AccessingProfile`, il peut, mais c'est facultatif, pousser une nouvelle donnée. En pratique, `IntrusionDetector` pousse une nouvelle donnée seulement lorsque la donnée qu'il prend en entrée est suspectée de correspondre à une intrusion. Au contraire, l'action à entreprendre du contrat d'interactions associé à l'opérateur `IP2Profile` est $\emptyset$. Quand cet opérateur reçoit une requête, il retourne le profil calculé au parent qui a fait la requête, mais il n'informe pas les autres parents, s'ils existent, en poussant la donnée.

SYNCHRONISATION Une séquence $\Uparrow (A_1, \ldots, A_n)$ dans la condition d'activation d'un contrat d'interactions indique la synchronisation de plusieurs enfants (source d'entité ou opérateur de contexte). Supposons que la donnée calculée par l'opérateur de contexte `AccessLogParser` soit raffinée en deux données : la localisation géographique de l'utilisateur (calculée à partir de

Opérateur	Contrat d'interactions associé
`AccessLogParser`	$\langle \Uparrow (\texttt{AccessLogReader.line}); \emptyset; \Uparrow \mathit{self} \rangle$
`AccessingProfile`	$\langle \Uparrow (\texttt{AccessLogParser}); \Downarrow (\texttt{IP2Profile}); \Uparrow \mathit{self} \rangle$
`IP2Profile`	$\langle \Downarrow \mathit{self}; \Downarrow (\texttt{NSLookup.ip2host}, \texttt{LDAPServer.host2profile}); \emptyset \rangle$
`IntrusionDetector`	$\langle \Uparrow (\texttt{AccessingProfile}); \emptyset; \Uparrow \mathit{self}? \rangle$
`IntrusionInformer`	$\langle \Uparrow (\texttt{IntrusionDetector}); \emptyset; \Uparrow (\texttt{Mailer.send}) \rangle$
`ProfileLogger`	$\langle \Uparrow (\texttt{AccessingProfile}); \emptyset; \Uparrow (\texttt{Logger.log}) \rangle$

TABLE 1: Contrats d'interactions des opérateurs du moniteur de serveur web

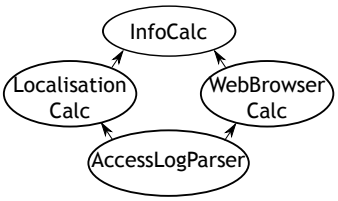

FIGURE 12: Exemple pour illustrer la synchronisation dans le moniteur de serveur web

l'adresse IP) et le navigateur utilisé. Supposons aussi que ces deux données soient calculées dans deux opérateurs de contexte différents, respectivement `LocalizationCalc` et `WebBrowserCalc` comme représenté dans la Figure 12. L'information calculée par ces deux opérateurs peut être ensuite combinée dans un autre opérateur de contexte nommé `InfoCalc` pour, par exemple, faire des statistiques sur l'utilisation des navigateurs en fonction de la localisation. Le contrat d'interactions de `InfoCalc` serait $\langle \Uparrow (\texttt{WebBrowserCalc}, \texttt{LocalizationCalc}); \emptyset; \Uparrow \mathit{self} \rangle$, ce qui assure que l'opérateur reçoive des informations synchronisées de `LocalizationCalc` et `WebBrowserCalc`.

DISJONCTION Une disjonction de noms dans une condition d'activation indique qu'un opérateur peut utiliser les données des enfants sans avoir besoin de les distinguer. Par exemple, dans le moniteur de serveur web, un accès dangereux peut être le résultat soit d'une intrusion soit d'une injection SQL. Les deux informations ont le même type, `IdentifiedAccess`. Supposons que nous ayons un nouvel opérateur de contexte nommé `SQLInjDetector` qui pousse des données quand une injection SQL est détectée comme indiquée sur la Figure 13. Nous pouvons alors définir un opérateur de contexte `DangerDetector` qui abstrait la notion de danger provenant soit de `IntrusionDetector` soit de `SQLInjDetector`. Cet opérateur serait associé au contrat d'interactions $\langle \Uparrow (\texttt{IntrusionDetector} \vee \texttt{SQLInjDetector}); \emptyset; \Uparrow \mathit{self} \rangle$.

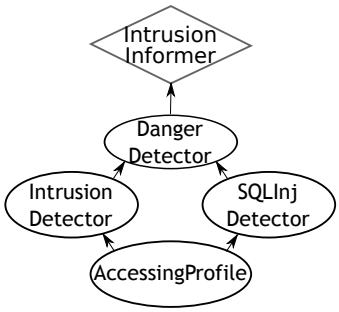

FIGURE 13: Exemple pour illustrer la disjonction dans le moniteur de serveur web

COMPOSITION DE CONTRATS D'INTERACTIONS Comme un opérateur peut être activé par plusieurs conditions, nous introduisons l'opérateur $\|$ qui permet de combiner plusieurs contrats d'interactions basiques. Une composition de contrats d'interactions basiques est, par exemple, nécessaire quand un opérateur de contexte peut être activé soit par un événement, soit par une requête. Par exemple, l'opérateur `AccessingProfile` pourrait avoir un second contrat d'interactions basique $\langle \Downarrow self; \emptyset; \emptyset \rangle$ qui permettrait d'accéder à la donnée la plus récente calculée par ce contexte.

Le langage DiaSpec permet d'exprimer les contrats d'interactions grâce à une syntaxe dédiée, illustrée Listing 3. Un contrat d'interactions basique est déclaré avec le mot-clé `behavior` et est placé au sein de la déclaration de l'opérateur. La condition d'activation est déclarée grâce au mot-clé `when`. Ce mot-clé est suivi soit par le mot-clé `provided` soit par le mot-clé `required` pour indiquer respectivement que l'activation de l'opérateur vient d'une donnée poussée par un enfant (ligne 5) ou d'une requête d'un parent (ligne 15). Le mot-clé `get` introduit les données requises comme montré à la ligne 16. Le contrat d'interactions d'un opérateur de contexte doit se terminer par `always`, `maybe` ou `no` suivi du mot-clé `publish` ce qui correspond respectivement à $E = \Uparrow self$, $E = \Uparrow self?$ et $E = \emptyset$ (voir exemple lignes 6 et 17). Le contrat d'interactions d'un opérateur de contrôle doit se terminer par le mot-clé `do` suivi d'ensembles d'actions d'entités comme illustré à la ligne 27.

4.5 SYNTHÈSE

Dans ce chapitre nous avons décrit comment le langage DiaSpec permet de définir les éléments essentiels du paradigme SCC. Ces éléments sont les entités ainsi que les opérateurs de contexte et de contrôle. Une entité est un composant qui fait l'interaction avec l'environnement. Une entité possède une facette source dans

```
context AccessLogParser as Access {
  source line from AccessLogReader;

  behavior {
        when provided line from AccessLogReader
        always publish
  }
}

context IP2Profile as Profile indexed by ip as IPAddress {
  source ip2host from NSLookup;
  source host2profile from LDAPServer;

  behavior {
        when required self
        get ip2host from NSLookup, host2profile from LDAPServer
        no publish
  }
}

controller ProfileLogger {
  context AccessingProfile;
  action Log on Logger;

  behavior {
        when provided AccessingProfile
        do any Log on Logger
  }
}
```

Listing 3: Quelques contrats d'interactions associés aux opérateurs du moniteur de serveur web

la couche de capture du paradigme SCC et une facette action dans la couche d'action. Les opérateurs sont des composants responsables de la logique applicative. Un opérateur de contexte raffine des données de la couche de capture et se place dans la couche de raffinement. Un opérateur de contrôle utilise les données raffinées pour contrôler des actions et se place dans la couche de contrôle.

Le langage DiaSpec est divisé en deux parties : la partie taxonomie déclare des classes d'entités indépendantes les unes des autres, tandis que la partie architecture déclare des opérateurs qui dépendent de la taxonomie. Les interactions possibles entre types de composants sont décrites au sein d'ensembles de triplets appelés des contrats d'interactions. Ces contrats d'interactions s'appliquent aux opérateurs et définissent les conditions d'activation, les données requises ainsi que les actions à entreprendre.

Dans le chapitre suivant nous expliquons comment les concepts du langage DiaSpec permettent la génération d'un *framework* de programmation, qui guide la réalisation de l'implémentation et qui en masque ses complexités.

4.6 LIENS AVEC D'AUTRES APPROCHES

Le langage DiaSpec peut être comparé à d'autres langages de description d'architectures logicielles. Les ADLs sont des langages dédiés à la description d'architectures logicielles. Parmi les ADLs les plus connus on trouve UML [12], *Unicon* [101], *Darwin* [74], *Rapide* [69], *Wright* [4], C2 [21, 79, 105] et *ACME* [48]. Ces langages sont comparés en détails par Medvidovic et al. [78, 80] ainsi que par le projet ACCORD [1]. Ces langages ont principalement été créés dans le but de représenter des architectures logicielles et de conduire des analyses sur ces architectures.

Les ADLs permettent de définir des architectures logicielles sous la forme de composants, de connecteurs et de leur configuration [78]. La configuration décrit comment les composants et connecteurs sont instanciés et liés. Une description DiaSpec a le même rôle, mais dans un contexte dédié aux applications SCC. Voici comment les notions de composant, de connecteur et de configuration sont représentées en DiaSpec :

COMPOSANT Plutôt que de décrire un ensemble de composants, une spécification DiaSpec décrit des ensembles d'entités et d'opérateurs de contexte et de contrôle. DiaSpec fournit donc des composants dédiés au paradigme SCC, guidant la réalisation de l'architecture.

CONNECTEUR Les connecteurs n'ont pas besoin d'être décrits par une spécification DiaSpec car ils sont fournis par l'approche.

CONFIGURATION Enfin, la configuration est répartie à divers endroits en DiaSpec en fonction de ce qui est configuré. La déclaration d'une connexion entre deux opérateurs est représentée par les liens de dépendance dans l'architecture. La déclaration d'une connexion entre une entité et un opérateur se fait en partie dans l'architecture, dans laquelle on lie un opérateur et une classe d'entités, et en partie dans l'implémentation de l'opérateur, dans laquelle on le lie avec une ou plusieurs instances de la classe d'entités. Enfin, la création d'instances des composants se fait dans une ou plusieurs classes Java dédiées.

Certains ADLs sont eux aussi dédiés à un paradigme particulier. L'ADL C2 par exemple est dédié à la description d'interfaces graphiques [105]. En C2, les composants sont organisés dans autant de couches que nécessaire et les connecteurs sont placés entre ces couches. Les composants communiquent toujours au travers des connecteurs et en utilisant des messages qui peuvent être de deux types : les *notifications*, qui vont des couches hautes vers les basses, et les *requêtes*, qui vont des couches basses vers les hautes. Un composant peut seulement être conscient des composants au-dessus de lui. Cet ADL est très similaire à DiaSpec. Une différence notable, cependant, est la présence d'un seul type de composant alors que DiaSpec en fournit trois, adaptés aux besoins du paradigme SCC. C2 n'a pas de mécanisme similaire aux contrats d'interactions.

Sens inverse à celui de DiaSpec

Il existe des modèles proches des contrats d'interactions. Par exemple, les modèles basés sur les automates tels que les automates IO [72] et les automates d'interfaces [22] sont couramment utilisés pour modéliser des interactions et des actions dans les systèmes concurrents et distribués. Ces approches ont été utilisées pour décrire des interactions de composants dans les ADLs [113]. Les contrats d'interactions sont plus simples dans la mesure où ils ne décrivent pas des séquences d'interactions mais seulement des contraintes d'interactions. Notre objectif est de spécifier seulement ce qui peut être garanti par le *framework* de programmation généré. Des modèles plus complets, tels que les automates, peuvent difficilement être garantis au niveau de l'implémentation. De plus, les modèles basés sur les automates sont une solution généraliste qui ne capturent pas les propriétés particulières du paradigme SCC. Par exemple, les opérateurs doivent être réactifs, ce qui nécessite une vérification avec les automates, alors que les contrats d'interactions l'assurent syntaxiquement.

4.7 TRAVAUX EN COURS ET FUTURS

BESOINS NON FONCTIONNELS Jakob et al. [59] proposent d'étendre les ADLs avec des annotations exprimant les besoins non fonctionnels et prennent une version précédente de DiaSpec comme exemple. Ces annotations sont ensuite traduites vers des déclarations d'aspects qui sont tissées dans le code généré par le compilateur DiaSpec. Ces travaux décrivent le langage DiaAspect qui est un langage orienté aspects pour les ADLs et un modèle de points d'insertion (*joint point*) applicable aux ADLs en général.

GESTION D'ERREURS Mercadal et al. [82] remarquent que la gestion des erreurs devient de plus en plus difficile à maîtriser au fur et à mesure que les applications agrègent des entités réseaux et des composants logiciels. Les auteurs proposent de remonter une partie de la gestion des erreurs de l'implémentation vers l'architecture. Pour ce faire, ils étendent DiaSpec avec des déclarations de gestion d'erreurs qui se matérialisent par l'ajout de mots-clés dédiés. Enfin, Mercadal et al. décrivent comment l'implémentation de la gestion des erreurs est facilitée par des changements dédiés apportés au *framework* de programmation généré par le compilateur de DiaSpec.

QUALITÉ DE SERVICE De nombreuses approches proposent des langages déclaratifs pour la gestion de Qualité de Service (QoS) dans des domaines variés, notamment les systèmes temps réel critiques. Cependant, Gatti et al. [49] remarquent que ces approches ne permettent pas la spécification de QoS à tout niveau du cycle de développement, empêchant la traçabilité des besoins. Les auteurs proposent donc une approche déclarative intégrée pour spécifier, raffiner et propager les besoins non fonctionnels, depuis la phase de capture des exigences jusqu'au déploiement de l'application. Les auteurs ont développé un support outillé, intégré à DiaSpec, permettant la traçabilité des exigences, tout en s'assurant de la cohérence de ces exigences lors du raffinement. Ces travaux permettent la génération automatique de mécanismes de surveillance des propriétés de QoS à l'exécution à partir des spécifications. Le langage DiaSpec est ainsi enrichi avec des propriétés de performance temporelle qui sont ensuite utilisées pour guider l'implémentation et la vérification.

COMPOSANTS COMPOSITES Beaucoup d'ADLs possèdent la notion de *composant composite* : un composant composite est un composant qui renferme d'autres composants, soit composites soit « primitifs ». Un composant composite permet de voir une architecture selon plusieurs niveaux de détails, en fonction des

besoins. Il pourrait être intéressant de voir ce qu'un composant composite en DiaSpec pourrait représenter. Par exemple, est-ce qu'un composant composite devrait contenir tous les types de composants (entités et opérateurs) ou seulement un type ? Là où nous pouvons facilement envisager des composites d'opérateurs de contexte, il semble moins intéressant d'envisager des composites d'opérateurs de contrôle car ceux-ci n'interagissent jamais entre eux. Si un composant composite contient tous les types de composants, il est crucial de se demander si celui-ci peut-être proactif ou pas. Si un composant composite peut être proactif, un tel composant composite ne peut être considéré que comme une source d'entité, la seule à pouvoir être proactive dans le paradigme SCC. En revanche, si un composant composite peut être seulement réactif il ne peut contenir aucune source d'entité proactive.

INFORMATION SUR LES SOURCES Actuellement, la description des sources d'entités ne prend pas en compte le fait qu'une source puisse être accédée seulement avec des événements ou seulement avec des requêtes. Par exemple, une source de type base de données peut ne jamais envoyer d'événements, et pourtant un contrat d'interactions basique peut actuellement avoir cette source comme condition d'activation. Dans ce cas, l'implémentation de la méthode abstraite générée ne sera jamais appelée car aucun événement n'arrivera jamais. Pour éviter ces situations, l'architecte pourrait indiquer, quand une source doit être accessible seulement en *push*, seulement en *pull* ou les deux.

5

IMPLÉMENTATION D'UNE APPLICATION SCC

The best architecture is worthless if the code doesn't follow it.

— Paul Clements and Mary Shaw [18]

Le compilateur de DiaSpec génère un *framework* de programmation Java à partir d'une spécification DiaSpec, constituée d'une taxonomie et d'une architecture. Ce *framework* est généré pour satisfaire deux propriétés importantes : il doit guider les développeurs dans l'implémentation de l'application et il doit les limiter à ce que la spécification autorise.

Dans ce chapitre nous commençons par présenter le *framework* de programmation généré pour la spécification du moniteur de serveur web. Cette présentation est ensuite utilisée pour expliquer comment un développeur implémente les entités et la logique applicative en s'appuyant sur ce *framework*. Nous montrons dans ce chapitre comment le *framework* de programmation guide les développeurs. Nous expliquons dans le chapitre suivant comment il les limite.

5.1 GÉNÉRATION DE SUPPORT DE PROGRAMMATION

Le *framework* de programmation généré contient une *classe abstraite* pour chaque composant de la spécification DiaSpec (entité, opérateur de contexte et opérateur de contrôle). Cette classe abstraite générée contient des méthodes pour faciliter l'implémentation des composants (découverte d'entités, interactions, *etc.*). Cette classe abstraite inclut aussi des déclarations de méthodes abstraites pour permettre aux développeurs d'implémenter la logique applicative (calculer une valeur, envoyer des commandes aux entités, *etc.*).

Implémenter un composant DiaSpec nécessite de créer une sous-classe de la classe abstraite générée correspondante. En faisant cela, le développeur doit implémenter chaque méthode abstraite. Le développeur écrit le code applicatif dans les sous-classes, jamais au sein du code généré. Cette stratégie contraste avec la génération de code incomplet (ou « à trous ») que le développeur doit remplir. En conséquence, dans notre approche,

un architecte peut changer la spécification DiaSpec et générer un nouveau *framework* de programmation sans écraser le code des développeurs. Les changements dans la spécification qui ont un effet sur le code déjà implémenté sont révélés par le compilateur Java.

cf. *Chapitre 6*

Le *framework* de programmation généré contient aussi des *proxies* et des interfaces pour permettre aux développeurs d'interagir avec les entités de façon transparente, sans gérer les détails des technologies de systèmes distribués. Enfin, le *framework* de programmation généré contient du support haut-niveau pour manipuler facilement des ensembles d'entités, suivant le motif de conception Composite [44].

5.1.1 *Support généré pour une entité*

Chaque déclaration faite dans l'architecture DiaSpec a un impact sur le *framework* de programmation généré. Nous détaillons cet impact en prenant comme exemple le moniteur de serveur web du chapitre précédent et notamment les Listings 1 page 32 et 2 page 37.

La compilation d'une déclaration d'entité de la taxonomie produit une classe abstraite du même nom que l'entité et préfixé par `Abstract`, telle que celle présentée dans le Listing 4.

ATTRIBUTS Les entités sont caractérisées par des attributs. À l'exécution, des valeurs sont assignées aux attributs. Les attributs sont gérés par les *getters* et *setters* générés dans la classe abstraite. Par exemple, l'entité `AccessLogReader` possède un attribut `isStarted` (hérité de l'entité `StoppableDevice`, Listing 1, ligne 2, page 32) à l'origine de la génération des méthodes `getIsStarted()` et `setIsStarted()` (Listing 4, lignes 10 à 11). Dans chaque sous-classe, le développeur utilisera la méthode `setIsStarted()` pour indiquer si l'entité est en fonction ou pas. La valeur initiale de chaque attribut doit être transmise au constructeur généré (Listing 4, ligne 3) qui va utiliser la méthode `setIsStarted()` (ligne 5).

SOURCES Une source d'entité produit des données pour les opérateurs de contexte. Le support pour cette propagation est fourni par le *framework* de programmation sous la forme de méthodes implémentées qui permettent au développeur d'entités d'alimenter le processus. Par exemple, la déclaration de l'entité `AccessLogReader` et de sa source `line` (Listing 1, ligne 11, page 32) impliquent la génération d'une méthode `publishLine()` dans la classe `AbstractAccessLogReader` (Listing 4, ligne 14). Cette

```
1  // de la ligne 10
2  public abstract class AbstractAccessLogReader {
3      protected AccessLogReader(ServiceConfiguration conf,
           Boolean isStarted) {
4          super(conf);
5          setIsStarted(isStarted);
6      }
7
8      // de StoppableDevice, ligne 2
9      private Boolean isStarted;
10     public Boolean getIsStarted() {return isStarted;}
11     protected void setIsStarted(Boolean isStarted) {...}
12
13     // de AccessLogReader, ligne 11
14     protected void publishLine(String line) {...}
15
16     // de StoppableDevice, ligne 3
17     public abstract void start();
18
19     // de StoppableDevice, ligne 3
20     public abstract void stop();
21     ...
22 }
```

Listing 4: La classe abstraite Java `AbstractAccessLogReader` générée par le compilateur de DiaSpec à partir de la déclaration de l'entité `AccessLogReader` (Listing 1, lignes 10 à 12, page 32). Les commentaires dans le code référencent les lignes dans la taxonomie, Listing 1.

méthode doit être appelée par les implémentations de l'entité `AccessLogReader`.

ACTIONS Une action correspond à un ensemble d'opérations supporté par l'entité. Cela prend la forme de méthodes abstraites générées dans la classe abstraite correspondante à l'entité. Chaque opération doit être implémentée par le développeur d'entité dans les sous-classes. Par exemple, la déclaration de l'entité `AccessLogReader` et de l'action `StartStop` (héritée de l'entité `StoppableDevice`, Listing 1, ligne 3, page 32) est à l'origine de la génération des méthodes abstraites `start()` et `stop()` (Listing 4, lignes 16 à 20).

5.1.2 *Support généré pour un opérateur*

Un opérateur peut avoir plusieurs implémentations, mais au plus une instance est déployée à tout moment

La compilation d'une déclaration d'opérateur dans l'architecture produit une classe abstraite du même nom préfixé par `Abstract`. Cette classe abstraite possède une *méthode abstraite* et une *méthode appelante* pour chaque contrat d'interactions basique. La méthode abstraite doit être implémentée par les développeurs dans les sous-classes et la méthode appelante est utilisée par le *framework* pour appeler l'implémentation de la méthode abstraite avec les bons arguments. Pour faciliter l'implémentation du comportement d'un opérateur, tout ce dont le développeur peut avoir besoin est fourni en paramètre à la méthode abstraite par la méthode appelante. Par exemple, si un contrat d'interactions indique qu'une interaction avec un autre opérateur est possible, un objet est passé en paramètre permettant d'initier cette interaction. Ceci limite grandement les besoins en documentation externe au *framework* généré : le développeur n'a pas à chercher le nom de la classe ou de la méthode à utiliser car tout est en paramètre.

Nous expliquons maintenant le lien entre le contrat d'interactions et la méthode abstraite. Pour des raisons de simplicité, nous commençons par présenter la compilation d'un contrat d'interactions pour un opérateur avec au plus un enfant dans la condition d'activation et au plus une donnée requise. Quand cet opérateur est un opérateur de contexte, celui-ci possède exactement un indice. Quand cet opérateur est un opérateur de contrôle, il contrôle exactement une action d'entité. L'enfant dans la condition d'activation et dans la donnée requise est un opérateur de contexte. Nous expliquerons ensuite les autres cas.

Soient les éléments suivants :

- C un opérateur ;
- TC le type de la donnée associée à C si C est un opérateur de contexte ;

Type de retour :

$T(\langle \Uparrow A; \_; \Uparrow self \rangle)$	=	TC	op. de contexte
$T(\langle \Uparrow A; \_; \Uparrow self? \rangle)$	=	void	op. de contexte
$T(\langle \Uparrow A; \_; \emptyset \rangle)$	=	void	op. de contexte
$T(\langle \Downarrow self; \_; \_ \rangle)$	=	TC	op. de contexte
$T(\langle \_; \_; \Uparrow C \rangle)$	=	Actions	op. de contrôle

Nom de la méthode :

$meth(\langle \Uparrow A; \_; \_ \rangle)$	=	onNew<A>	
$meth(\langle \Downarrow self; \_; \_ \rangle)$	=	get	op. de contexte

Information de l'activateur :

$param_1(\langle \Uparrow A; \_; \_ \rangle)$	=	TA valeur	
$param_1(\langle \Downarrow self; \_; \_ \rangle)$	=	TX indice	op. de contexte

Fonction de donnée requise :

$param_2(\langle \_; \Downarrow A; \_ \rangle)$	=	#TA(...) getA	

Fonction d'émission :

$param_3(\langle \_; \_; \Uparrow self? \rangle)$	=	#void(TC) publish	op. de contexte
$param_3(\langle \_; \_; \Uparrow C \rangle)$	=	... *découverte*	op. de contrôle

FIGURE 14: Compilation de contrats d'interactions

- TX le type de l'indice de C si C est un opérateur de contexte ;
- K un des contrats d'interactions basiques de C ;
- A un opérateur de contexte, enfant de C ;
- TA le type de la donnée associée à A ;
- D une action d'entité donc C dépend si C est un opérateur contrôle ;

Alors, la méthode abstraite générée correspondante est de la forme :

$$T(K)meth(K)(param_1(K), param_2(K), param_3(K))$$

où $T(K)$, $meth(K)$, $param_1(K)$, $param_2(K)$, et $param_3(K)$ sont définis comme dans la Figure 14 et décrits comme suit.

TYPE POUR UN OPÉRATEUR DE CONTEXTE Dans le cas où C est un opérateur de contexte et où une donnée est poussée par un enfant, le type de retour de la méthode générée dépend de

l'émission. Si une émission est nécessaire ($E = \Uparrow self$), le type de retour est le type de l'opérateur de contexte TC. Sinon, le type de retour est `void`. Si une émission est optionnelle ($E = \Uparrow self?$), la donnée à publier est fournie au travers d'une fonction passée en paramètre plutôt que par retour de valeur. Dans le cas d'une donnée requise par un parent, le type de retour est toujours le type du contexte. Même quand il n'y a pas d'émission, la donnée calculée doit toujours être retournée au parent qui a fait la requête.

TYPE POUR UN OPÉRATEUR DE CONTRÔLE Le type de retour de la méthode générée pour un opérateur de contrôle est toujours le type *Actions*. Ce type permet d'encapsuler toutes les commandes à invoquer sur les entités.

NOM DE LA MÉTHODE Dans le cas d'une donnée poussée par un enfant, le nom de la méthode générée commence par `onNew`, illustrant le fait qu'un enfant fourni une nouvelle valeur. Le nom de la méthode est suivi du nom de l'enfant. Dans le cas d'une donnée requise par un parent, pour un opérateur de contexte, le nom de la méthode est `get()`, traduisant le fait que le but est de récupérer la donnée de l'opérateur de contexte. Un opérateur de contrôle n'a pas de méthode `get()` car un opérateur de contrôle ne calcule pas de donnée.

PREMIER PARAMÈTRE : ACTIVATEUR Dans le cas d'une donnée poussée par un enfant, le premier argument de la méthode est la donnée poussée par l'enfant, de type `TA`. Dans le cas d'une donnée requise par un parent, le premier argument de la méthode est l'indice de l'opérateur de contexte, de type `TX`.

DEUXIÈME PARAMÈTRE : DONNÉE REQUISE Le deuxième argument de la méthode est une fonction qui permet d'exécuter l'interaction avec l'enfant qui fournit une donnée requise. Quand l'enfant requiert un paramètre, la fonction est paramétrée en conséquence. La fonction peut être appelée autant de fois que nécessaire, avec différentes valeurs pour le paramètre si besoin.

TROISIÈME PARAMÈTRE : ÉMISSION Dans le cas d'un opérateur de contexte, si l'émission est optionnelle, le troisième argument est une fonction qui permet de publier une nouvelle donnée, de type T, à tous les parents. Dans le cas d'un opérateur de contrôle, le troisième argument permet d'initier la découverte d'entités (voir plus bas). Dans tous les autres cas, le troisième paramètre est omis.

Nous venons d'étudier comment se compile un contrat d'interactions pour les cas simples. Les exemples qui suivent nous permettent d'illustrer ces cas simples ainsi que les cas plus complexes.

5.1.2.1 *Exemples de compilation de contrats d'interactions*

Analysons quelques méthodes abstraites générées à partir des contrats d'interactions du moniteur de serveur web (voir Table 1, page 43).

Le contrat d'interactions de l'opérateur `AccessingProfile` est compilé vers la méthode abstraite suivante :

```
abstract IdentifiedAccess
onNewAccessLogParser(Access newAccess,
    PullFromIP2ProfileCallback ip2Profile);
```

Le nom de cette méthode abstraite commence par `onNew`, illustrant le fait qu'un enfant fournit une nouvelle donnée. Le premier paramètre contient la donnée fournie par l'enfant. Le second paramètre représente une fonction qui permet d'exécuter une requête (un *pull*) vers l'opérateur `IP2Profile`. Cette fonction prend une adresse IP en paramètre et retourne un profil correspondant. Le type de retour de la méthode abstraite `onNewAccessLogParser()` force l'implémentation de cette méthode à retourner un profil qui est alors envoyé automatiquement (un *push*) par la méthode appelante.

L'opérateur `IP2Profile` est une sorte de base de données qui peut être accédée seulement par des requêtes avec une adresse IP en argument. Son contrat d'interactions est compilé vers la méthode abstraite suivante :

```
abstract Profile
get(IPAddress newIPAddress, PullFromNSLookupCallback nsLookups,
    PullFromLDAPServerCallback ldapServers);
```

Le nom de cette méthode abstraite est `get`, illustrant le fait qu'une donnée est requise (par un parent). Une implémentation de cette méthode abstraite peut utiliser la source `ip2host` venant de l'entité `NSLookup` ainsi que la source `host2profile` de l'entité `LDAPServer`. Ces deux interactions possibles sont représentées par deux objets en paramètre qui permettent ces interactions. Comme `NSLookup` et `LDAPServer` sont deux classes d'entités, il est nécessaire d'indiquer avec quelles instances l'interaction doit se faire. C'est le rôle de la découverte d'entités qui sera étudiée plus bas, dans la Section 5.2.3. Parce que l'émission E du contrat d'interactions est $\emptyset$, `IP2Profile` retourne une valeur seulement au parent qui a fait la requête.

Le contrat d'interactions de `IntrusionDetector` est compilé vers la méthode abstraite suivante :

```
abstract void
onNewAccessingProfile(IdentifiedAccess newIdentifiedAccess,
    PublishCallback publish);
```

Le premier paramètre représente la donnée transmise par l'opérateur de contexte `AccessingProfile`. Comme il n'y a pas de donnée requise dans le contrat d'interactions de `IntrusionDetector`, aucun paramètre ne permet de faire une requête. Tous les profils arrivant à l'opérateur `IntrusionDetector` ne sont pas nécessairement des intrusions. Une fonction `publish` est donc passée en paramètre pour permettre à la logique applicative de décider d'avertir quand une intrusion est détectée. Cette fonction prend un profil (de type `IdentifiedAccess`) en paramètre, le stocke, et ne retourne rien. Une fois que l'implémentation de la méthode abstraite retourne, la méthode appelante utilise ce paramètre comme valeur à transmettre aux parents abonnés. Si cette fonction n'est jamais appelée, aucune valeur n'est transmise aux parents abonnés.

Le contrat d'interactions de `AccessLogParser` est compilé vers la méthode abstraite suivante :

```
abstract Access
onLineFromAccessLogReader(LineFromAccessLogReader lineProxy);
```

Le paramètre `lineProxy` est une structure qui contient à la fois la donnée transmise par une instance de `AccessLogReader` (de type `String`) et les valeurs des attributs de cette instance (un booléen pour l'attribut `isStarted`). Pour qu'un opérateur de contexte puisse recevoir des donnés poussées par une source d'entité telle que `line`, il faut avant tout qu'il s'y abonne. Nous verrons plus loin qu'une méthode `postInitialize()` est déclarée abstraite pour laisser à l'opérateur la possibilité de s'abonner.

Le contrat d'interactions de l'opérateur de contrôle `ProfileLogger` est compilé vers la méthode abstraite suivante :

```
abstract Actions
onNewAccessingProfile(IdentifiedAccess newIdentifiedAccess,
    ActionOnLogger loggers);
```

Le paramètre `loggers` permet d'initier la découverte d'entités de type `Logger` puis d'exécuter la méthode `log()` sur le résultat. Cette méthode `log()` retourne un objet de type `Actions` qui encapsule toutes les commandes à effectuer sur les entités sélectionnées par la découverte d'entités. Ces commandes sont stockées dans les objets de type `Actions` et sont exécutées par la méthode appelante lorsque l'implémentation de la méthode abstraite retourne.

SYNCHRONISATION Le contrat d'interactions de `InfoCalc` (Figure 12) est compilé vers la méthode abstraite suivante :

```
abstract Info
onNewWebBrowserCalcAndLocalizationCalc(WebBrowser newWebBrowser,
    Localization newLocalization);
```

Le nom de la méthode générée contient l'ensemble des noms des opérateurs prenant part à la synchronisation. Cette méthode doit être appelée avec le navigateur et la localisation dès que les deux sont présentes. Plusieurs stratégies peuvent être utilisées pour implémenter ce genre de synchronisation : une approche est de garder seulement la plus récente des données venant de chaque opérateur, une autre approche est d'ajouter toutes les valeurs dans des files. Notre implémentation utilise une file pour chaque opérateur. Quand chaque file a au moins une donnée, le *framework* consume une donnée dans chacune et invoque la méthode abstraite en passant ces données en paramètre. Cette implémentation peut être changée par le développeur.

DISJONCTION Le contrat d'interactions de `DangerDetection` (Figure 13) est compilé vers la méthode abstraite suivante :

```
abstract IdentifiedAccess
onNewIntrusionDetectorOrSQLInjDetector(
    IdentifiedAccess newIdentifiedAccess);
```

Le nom de la méthode générée contient l'ensemble des noms des opérateurs prenant part à la disjonction. Cette méthode abstraite a un paramètre qui représente la disjonction. Le *framework* généré appelle cette méthode pour chaque donnée envoyée soit de l'opérateur `SQLInjDetector` soit de l'opérateur `IntrusionDetector`.

FONCTIONS D'INTERACTION Comme indiqué plus haut, des fonctions sont passées en paramètre des méthodes abstraites pour représenter des interactions optionnelles. Le langage Java ne disposant pas des clôtures, chacune de ces fonctions est implémentée comme une classe Java interne de la classe abstraite avec une seule méthode. Les instances de ces classes internes sont créées par la méthode appelante, qui les passe à l'implémentation de la méthode abstraite. Par exemple, `PullFromIP2ProfileCallback` est défini de la façon suivante :

```
public abstract class AbstractAccessingProfile {
  ...
  protected class PullFromIP2ProfileCallback {
    ...
    public Profile get(IPAddress ipAddress) {
        ...
        // transfère la requête à l'instance déployée de
        // l'opérateur IP2Profile en passant par DiaGenCore
        return ...;
    }
```

```
  }
}
```

Nous avons vu jusqu'à maintenant comment le compilateur de DiaSpec utilise les descriptions d'entités et d'opérateurs pour générer un *framework* de programmation dédié.

5.2 UTILISATION DU SUPPORT GÉNÉRÉ

Nous allons maintenant décrire la méthode pour implémenter une entité, un opérateur de contexte et un opérateur de contrôle, en s'appuyant sur le *framework* généré.

5.2.1 *Implémentation d'une entité*

L'implémentation d'une entité décrite dans la spécification se fait en créant une sous-classe de la classe dédiée générée dans le *framework* de programmation. Les Listings 5 et 6 montrent deux exemples d'implémentation d'entités tirés du moniteur de serveur web.

La gestion des exceptions n'est pas montrée dans les listings suivants car elle fait partie d'un travail annexe [82]

Le Listing 5 montre une implémentation de l'entité `AccessLogReader` qui publie au fur et à mesure les lignes ajoutées dans un fichier : en pratique, ce fichier est un journal des accès à un serveur web. La classe concrète `FileAccessLogReader` (Listing 5) hérite de la classe abstraite `AbstractLogReader` (Listing 4). Ce faisant, `FileAccessLogReader` doit implémenter toutes les méthodes de sa super-classe. Par exemple, les méthodes `start()` et `stop()` sont implémentées dans la classe `FileAccessLogReader` (Listing 5, lignes 10 à 25), sont déclarées dans la classe `AbstractAccessLogReader` (Listing 4, lignes 16 à 20) et sont originaires de la description de la classe d'entités `AccessLogReader` qui hérite de la classe d'entités `StartStop` (Listing 1, lignes 6 à 8, page 32). Les méthodes `setIsStarted()` et `getIsStarted()`, utilisées dans le Listing 5, sont implémentées dans la super-classe (Listing 4).

Cet exemple montre la publication d'une source de données *via* une méthode `publish`. L'appel à la méthode `publishLine()` va déclencher une réaction dans les opérateurs de contexte qui ont souscrit, `AccessLogParser` dans ce cas.

Le Listing 6 montre une implémentation possible de l'entité `NSLookup`. Une implémentation d'entité peut répondre à des requêtes venant d'opérateurs de contexte en surchargeant la méthode `get()` associée à la source (Listing 6, lignes 7 à 11). Cette méthode reçoit en paramètre l'adresse IP à convertir en nom d'hôte grâce à la déclaration de l'indice `ip` (Listing 1, ligne 15,

```java
public class FileAccessLogReader extends AbstractAccessLogReader {

    private final File log;

    public FileAccessLogReader(ServiceConfiguration conf, File
        log) {
        super(conf, false);
        this.log = log;
    }

    @Override
    public void start() {
        setIsStarted(true);
        BufferedReader br = openFile();
        while (getIsStarted()) {
            String line = br.readLine();
            if (line == null) { sleep(); }
            else { publishLine(line); }
        }
        br.close();
    }

    @Override
    protected void stop() {
        setIsStarted(false);
    }

    private BufferedReader openFile() {
        BufferedReader br;
        br = new BufferedReader(new FileReader(log));
        while (br.readLine() != null)
            ; // go to end
        return br;
    }

    private void sleep() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            setIsStarted(false);
        }
    }
}
```

Listing 5: Une implémentation écrite par un développeur de la classe d'entités `AccessLogReader`

```java
public class INetNSLookup extends AbstractNSLookup {

    public INetNSLookup(ServiceConfiguration conf) {
        super(conf);
    }

    @Override
    protected String getIp2host(IPAddress ip) throws Exception {
        InetAddress inetAddress =
            InetAddress.getByName(ip.getAddress());
        return inetAddress.getHostName();
    }

}
```

Listing 6: Une implémentation écrite par un développeur de la classe d'entités `NSLookup`

page 32). La classe utilisée `InetAddress` est fournie par Java dans le paquetage `java.net`.

5.2.2 *Implémentation d'un opérateur*

L'implémentation d'un opérateur de contexte ou de contrôle se fait de la même façon que pour une entité : il suffit de créer une nouvelle sous-classe de la classe abstraite générée correspondante à l'opérateur. Les Listings 7, 8, 9 et 10 montrent quelques exemples d'implémentations d'opérateurs tirés du moniteur de serveur web.

Le Listing 7 montre une implémentation de l'opérateur de contexte `AccessLogParser`. Cette implémentation considère que les lignes transmises par la source `line` de l'entité `AccessLogReader` sont au format du serveur web *Lighttpd*.[1] La méthode `onNewLineFromAccessLogReader()` est appelée automatiquement par le *framework* lorsqu'une entité `AccessLogReader` publie une ligne. Cette implémentation utilise un ensemble d'expressions rationnelles permettant de récupérer les informations dans la chaîne de caractères fournie par `AccessLogReader`. Une fois les données récupérées et regroupées dans une structure `Access`, l'instruction Java `return` (Listing 7, ligne 22) va rendre le contrôle à la méthode appelante qui va transférer cette structure à tous les parents qui ont souscrit.

La méthode `postInitialize()` (Listing 7, lignes 11 à 13) est appelée par le *framework* lors du déploiement de l'opérateur. Surcharger cette méthode permet, par exemple, aux opérateurs de contexte de souscrire à une ou plusieurs instances d'entités.

1. http://redmine.lighttpd.net/wiki/lighttpd/Docs:ModAccessLog

```
1  public class LighttpdAccessLogParser extends
       AbstractAccessLogParser {
2
3      private static final Pattern REMOTE_HOST_IP =
           Pattern.compile("^([^ ]+) ");
4      [...]
5
6      public LighttpdAccessLogParser(ServiceConfiguration conf) {
7          super(conf);
8      }
9
10     @Override
11     protected void postInitialize(SubscribeOnAccessLogReader
           accessLogReaders) {
12         accessLogReaders.all().subscribeLine();
13     }
14
15     @Override
16     public Access
           onNewLineFromAccessLogReader(LineFromAccessLogReader
           lineProxy) {
17         String line = lineProxy.value();
18         Access access = new Access();
19         access.setLine(line);
20         access.setHost_ip(new IPAddress(parseRemoteHostIP(line)));
21         [...]
22         return access;
23     }
24
25     public String parseRemoteHostIP(String line) {
26         Matcher m = REMOTE_HOST_IP.matcher(line);
27         m.find();
28         return m.group(1);
29     }
30 }
```

Listing 7: Une implémentation écrite par un développeur de l'opérateur de contexte `AccessLogParser`

C'est ce qui est fait dans le Listing 7, ligne 12, où le paramètre `accessLogReaders` est utilisé pour découvrir toutes les instances de l'entité `AccessLogReader` et souscrire à leurs sources `line`.

La découverte d'entités est étudiée plus loin dans ce chapitre

Le Listing 8 correspond à une implémentation de l'opérateur de contexte `AccessingProfile`. La méthode `onNewAccessLogParser()` est appelée automatiquement quand une nouvelle donnée arrive de l'opérateur `AccessLogParser`. Le premier paramètre, `access`, contient la donnée. Le deuxième paramètre, `ip2Profile`, permet de faire une requête vers l'opérateur de contexte `IP2Profile`. Cette interaction se fait au moment de l'appel à la méthode `get()` (Listing 8, ligne 9) qui nécessite une adresse IP en paramètre.

```
1  public class AccessingProfile extends AbstractAccessingProfile {
2
3      public AccessingProfile(ServiceConfiguration conf) {
4          super(conf);
5      }
6
7      @Override
8      protected IdentifiedAccess onNewAccessLogParser(Access access,
           PullFromIP2ProfileCallback ip2Profile) {
9          Profile profile = ip2Profile.get(access.getHost_ip());
10         return new IdentifiedAccess(access, profile);
11     }
12 }
```

Listing 8: Une implémentation écrite par un développeur de l'opérateur de contexte `AccessingProfile`

Le Listing 9 contient un exemple d'implémentation de l'opérateur de contexte `IP2Profile`. La méthode `get()` est appelée par le *framework* lorsque l'opérateur `AccessingProfile` fait une requête. Le premier paramètre, une adresse IP, est l'indice associé à `IP2Profile` (Listing 2, ligne 3, page 37). Cette adresse IP est fournie par `AccessingProfile`. Les deux entités `NSLookup` et `LDAPServer` sont mises à contribution pour convertir cette adresse IP en un profil utilisateur. À la ligne 10 du Listing 9, la découverte d'entités est utilisée pour récupérer une instance de l'entité `NSLookup` et faire une requête sur sa source `ip2host`. De la même façon, la ligne 13 recherche une entité `LDAPServer` et fait une requête sur sa source `host2profile`.

Le Listing 10 montre un exemple d'implémentation de l'opérateur de contrôle `ProfileLogger`. La méthode `onNewAccessingProfile()` est automatiquement appelée par le *framework* lorsque `AccessingProfile` publie une donnée. Le premier paramètre est le profil publié tandis que le deuxième permet de découvrir des entités `Logger`. La variable `allLoggers` représente toutes les instances de l'entité `Logger` actuellement déployées. Cette variable `allLoggers` et sa méthode `log()` sont utilisées pour demander à tous les `Loggers` d'enregistrer une nouvelle information dans leur journal. La méthode `log()` retourne une instance de la classe `Action` qui est utilisée pour sauvegarder les actions à effectuer. Ces actions sont ensuite retournées au *framework* pour que les commandes correspondantes soient exécutées sur les entités.

Le *framework* de programmation guide le développeur par rapport à la description de l'architecture. En créant des sous-classes des classes abstraites générées, le développeur est obligé d'implémenter toutes les méthodes abstraites. Pour faciliter ce processus, la plupart des environnements de développement pour langages orientés objets (tels que *Eclipse*) génèrent des squelettes de codes

```
1  public class IP2Profile extends AbstractIP2Profile {
2
3      public IP2Profile(ServiceConfiguration conf) {
4          super(conf);
5      }
6
7      @Override
8      public Profile get(IPAddress ip, PullFromNSLookupCallback
           nsLookups, PullFromLDAPServerCallback ldapServers) {
9
10         String hostname = nSLookups.anyOne().getIp2host(ip);
11         if (hostname == null)
12             return null;
13         Profile profile =
               lDAPServers.anyOne().getHost2profile(hostname);
14         return profile;
15     }
16 }
```

Listing 9: Une implémentation écrite par un développeur de l'opérateur de contexte `IP2Profile`

```
1  public class ProfileLogger extends AbstractProfileLogger {
2
3      public ProfileLogger(ServiceConfiguration conf) {
4          super(conf);
5      }
6
7      @Override
8      public Actions onNewAccessingProfile(IdentifiedAccess profile,
           ActionOnLogger loggers) {
9          LoggerComposite allLoggers = loggers.all();
10         if (profile.getProfile() != null) {
11             return allLoggers.log(profile.getProfile().getName() +
                   " has just connected", Level.DEBUG);
12         } else {
13             return allLoggers.log("anonymous connection",
                   Level.ERROR);
14         }
15     }
16 }
```

Listing 10: Une implémentation écrite par un développeur de l'opérateur de contrôle `ProfileLogger`

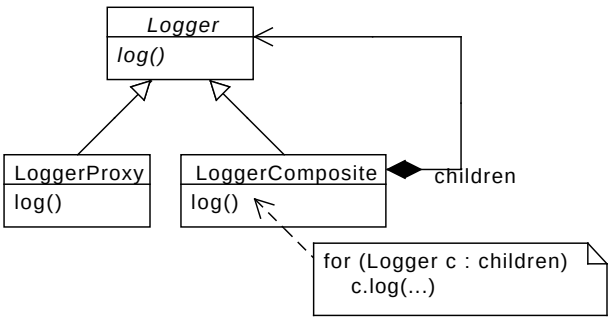

FIGURE 15: Utilisation du motif de conception Composite pour interagir avec les entités Logger

en fonction des super-classes. De plus, le *framework* de programmation passe chaque information nécessaire comme argument aux méthodes à implémenter dans les opérateurs. Ces arguments libèrent le développeur d'avoir à chercher des noms de classes ou de méthodes dans une documentation.

5.2.3 *Découverte d'entités*

Le *framework* de programmation dédié fournit du support pour la découverte d'entités de la taxonomie. Cette découverte d'entités est utile aux opérateurs de contexte pour pouvoir s'abonner et faire des requêtes sur les entités. Elle est aussi utile aux opérateurs de contrôle pour exécuter des commandes sur les entités. L'architecture faisant le lien entre les opérateurs et les classes d'entités, la découverte d'entités permet d'instancier ce lien en choisissant quelles entités de la classe vont participer à l'interaction.

La découverte d'entités retourne une collection de *proxies* pour les entités sélectionnées. Cette collection est encapsulée dans un objet Composite [44]. Par exemple, la Figure 15 montre comment ce motif de conception Composite est utilisé pour que l'implémentation de l'opérateur `ProfileLogger` puisse agir implicitement sur un ensemble d'entités `Logger` (Listing 10, lignes 11 et 13).

Les méthodes des implémentations qui sont autorisées à faire de la découverte d'entités ont en paramètre un objet permettant d'initier cette découverte. Par exemple, l'implémentation de l'opérateur `ProfileLogger` utilise le paramètre `loggers` pour récupérer toutes les instances de `Logger` (Listing 10, ligne 9). Cet objet est dédié à une classe d'entités et possède des méthodes qui permettent d'ajouter des filtres sur les attributs de cette classe. La méthode `all()` n'utilise aucun filtre et permet donc de récupérer toutes les instances déployées. La méthode `anyOne()` permet de récupérer une entité, choisie au hasard parmi toutes les entités dé-

Opérateur	*Paramètre(s)*	*Résultat*
`or`	2 expressions	Une des deux expressions en paramètre doit convenir
`and`	2 expressions	Les deux expressions en paramètre doivent convenir
`not`	1 expression	L'expression en paramètre ne doit pas convenir
`between`	2 valeurs	La valeur de l'attribut doit se trouver entre les 2 paramètres
`gt`, `lt`, `eq`	1 valeur	La valeur de l'attribut doit être respectivement supérieure, inférieure ou égale au paramètre

TABLE 2: Liste des opérateurs logiques pour les expressions logiques de la découverte d'entités

ployées (Listing 9, ligne 10). Pour ne récupérer que les instances de l'entité `Logger` qui sont démarrées, il est possible d'utiliser le filtre pour l'attribut `isStarted` de la façon suivante :

```
LoggerComposite runningLoggers = loggers.whereIsStarted(true);
```

L'argument que l'on passe au filtre, ici `true`, est soit une valeur du même type que l'attribut, ici `Boolean`, soit une expression logique. Si l'argument est une valeur, les entités découvertes sont celles qui ont la même valeur pour l'attribut filtré. Le morceau de code ci-dessus récupère donc seulement les entités dont l'attribut `isStarted` vaut `true`. Si une expression logique est choisie, la valeur de l'attribut des entités sélectionnées doit être une solution de l'expression logique. Voici une façon compliquée de sélectionner toutes les entités `Logger` :

```
LoggerComposite runningLoggers =
    loggers.whereIsStarted(or(eq(true), eq(false)));
```

Cette expression logique choisit toutes les instances de `Logger` qui ont un attribut `isStarted` soit à `true` soit à `false`.

Les opérateurs logiques de la Table 2 sont définis et peuvent être utilisés pour la construction d'expressions logiques. Certains opérateurs logiques ne peuvent être utilisés que lorsque l'attribut a un type totalement ordonné. Par exemple, l'opérateur logique `gt` n'a pas de sens si nous l'utilisons sur un objet type `Profile` (Listing 1, ligne 26, page 32). Le compilateur Java vérifie cette contrainte et empêche l'utilisation de ces opérateurs logiques sur des attributs incompatibles. De nouveaux opérateurs logiques

un tel type doit implémenter l'interface Java `java.lang.Comparable`

peuvent être définis pour augmenter l'expressivité du langage de requête :

```
<Attribute extends Comparable<Attribute>>
    AttributeFilter<Attribute> gte(Attribute value) {
    return or(gt(value), eq(value));
}
```

Le code précédent définit un nouvel opérateur `gte` avec un paramètre, qui vérifie qu'une valeur d'attribut est supérieure ou égale au paramètre. Cet opérateur est valide seulement lorsque l'attribut est d'un type totalement ordonné.

Notre approche permet aussi de définir des requêtes sur plusieurs attributs :

```
loggers.whereIsStarted(true).andLevel(Level.ALL);
```

Cette requête sélectionne toutes les instances de `Logger` qui sont démarrées et qui ont l'attribut `level` à la valeur `ALL`. L'implémentation ne permet pas pour l'instant d'exprimer des expressions logiques utilisant plusieurs attributs. Ainsi, il n'est pas possible qu'une requête spécifie qu'une entité doit avoir une certaine valeur pour un attribut *ou* une autre valeur pour un autre attribut.

5.3 IMPLÉMENTATION DU COMPILATEUR

L'architecture du compilateur est représentée Figure 16. Celle-ci est composée principalement de trois éléments : (1) un analyseur syntaxique, (2) un analyseur sémantique, (3) un générateur de code.

ANALYSEUR SYNTAXIQUE La grammaire du langage DiaSpec est décrite dans le langage ANTLR,[2] un dérivé de EBNF (*Extended Backus-Naur Form*). À partir de cette description, ANTLR produit un analyseur syntaxique qui prend une déclaration DiaSpec en entrée et produit un arbre de syntaxe abstrait (AST). En cas de description non conforme au langage DiaSpec, un message indiquant la nature et l'emplacement de l'erreur est retourné à l'architecte.

ANALYSEUR SÉMANTIQUE Cet AST est ensuite parcouru pour vérifier que la spécification DiaSpec est cohérente relativement à un ensemble de règles. Par exemple, une de ces règles impose qu'un même nom ne peut être utilisé pour décrire à la fois une entité et un opérateur. Une autre règle impose que le contrat d'interactions d'un opérateur ne peut faire référence qu'à

2. http://antlr.org/

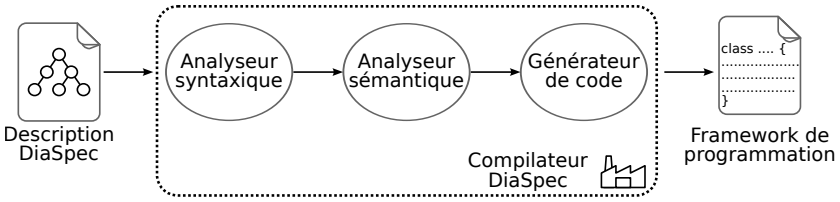

FIGURE 16: Architecture du compilateur de DiaSpec

des éléments architecturaux dont l'opérateur dépend. Les règles sémantiques sont étudiées plus en détail au chapitre suivant, page 81. Si une règle n'est pas respectée, un message indiquant la nature et l'emplacement de l'erreur est retourné à l'architecte.

GÉNÉRATEUR DE CODE Une fois l'AST vérifié, un générateur de code produit des fichiers Java. Le générateur de code est implémenté en utilisant StringTemplate,[3] un langage de *template* permettant de représenter facilement le code généré. Ces fichiers sont ensuite placés soit dans un dossier soit dans une archive JAR, au choix de l'utilisateur.

5.4 SYNTHÈSE

Dans ce chapitre nous avons vu comment les descriptions exprimées dans le langage DiaSpec sont traduites vers un *framework* de programmation dédié. Les fichiers de ce *framework* généré ne sont jamais modifiés par les développeurs, ce qui permet de faire évoluer les descriptions DiaSpec après que l'implémentation ait commencée. Cette séparation entre le code écrit par les développeurs et le code généré se fait grâce au mécanisme d'héritage du langage hôte : le code est généré dans les super-classes, adapté et complété par les développeurs dans les sous-classes.

Le Chapitre 6 détaille le support des évolutions possibles

Cette technique s'appelle Generation Gap *[39, Chap.57]*

Bien que simple, le langage DiaSpec est suffisamment précis pour permettre la génération d'un *framework* de programmation qui assiste le développeur dans l'implémentation de l'architecture. Pour une entité, cette assistance se matérialise par des méthodes abstraites à implémenter et des méthodes concrètes à appeler. Pour un opérateur, cette assistance se matérialise par des méthodes abstraites à implémenter. Chaque méthode abstraite pour un opérateur contient en paramètre tout ce qui peut être nécessaire au développeur : les valeurs transmises ainsi que les moyens d'interactions avec les autres éléments de l'architecture.

Pour permettre aux opérateurs d'interagir avec des entités, un mécanisme de découverte d'entités est généré dans le *framework* de programmation. Ce mécanisme est fourni sous la forme d'un

3. http://stringtemplate.org/

DSL enchâssé dans Java à la fois expressif et extensible. Grâce à lui, les développeurs sont capables de gérer les environnements dynamiques présents dans certains domaines comme celui de l'informatique ubiquitaire.

5.5 LIENS AVEC D'AUTRES APPROCHES

Nous décrivons ici quelques liens entre le travail présenté dans ce chapitre et d'autres approches.

LANGAGES DE DESCRIPTION D'ARCHITECTURES Les ADLs se sont historiquement concentrés sur la description d'architectures logicielles et sur leurs analyses [1, 78]. En ce qui concerne l'implémentation, certains ADLs, comme *Aesop* [47] et C2 [21], fournissent un *framework* de programmation générique composé de classes abstraites telles que `Component` et `Connector`. Les développeurs doivent implémenter les composants de l'application en créant des sous-classes du *framework*. Cette implémentation se fait indépendamment de la description de l'architecture qui est utilisée simplement comme référence par les développeurs. Un simulateur, ou une plateforme, d'exécution s'occupe ensuite de créer des instances des composants et connecteurs implémentés, de connecter ces instances, et d'exécuter l'application en se référant à la description de l'architecture. Notre approche est différente car le *framework* de programmation généré est dédié à l'application contrastant avec les *frameworks* génériques des ADLs. Par exemple, dans notre approche, un développeur implémente une classe qui hérite d'une classe `AccessLogParser` avec des méthodes abstraites dédiées et pas d'une classe générique `Component` valide pour toutes les applications.

GÉNÉRATION DE CODE Le *framework* généré par le compilateur de DiaSpec contient des classes abstraites qui doivent être implémentées par des sous-classes écrites manuellement par les développeurs. Cette technique s'appelle *Generation Gap* et a été étudiée initialement par Vlissides [112] et aussi très récemment par Fowler [39]. Cette technique permet de garder une séparation claire entre code généré et code écrit manuellement car ces deux parties sont écrites dans des fichiers et classes différents. Cette technique permet aussi les générations successives, même après que l'écriture de code manuelle ait commencé. Il existe d'autres approches de génération de code qui ont ce dernier avantage. Ainsi, l'ingénierie dirigée par les modèles, notamment avec MDA [77], a un objectif de générer tout le code à partir de modèles [99]. Ceci a l'avantage de ne plus requérir de code écrit manuellement. Cependant, cela présente l'inconvénient de néces-

siter que toute la logique applicative soit implémentée en utilisant des notations graphiques telles que le langage UML, ce qui n'est pas toujours plus efficace que d'implémenter avec des notations textuelles [36, 73]. Il n'est pas non plus toujours possible, ou souhaitable, que tout le code se situe dans des modèles. Quand c'est le cas, une autre approche de génération de code possible est d'utiliser des marqueurs dans le code généré pour indiquer où les développeurs peuvent ajouter le code écrit manuellement. Ces marqueurs sont appelés *protected areas* ou *modifiable regions* [108] et sont placés au sein du code généré grâce à des commentaires, ignorés par le compilateur. Lors d'une nouvelle génération, le générateur de code prend soin de replacer ce code écrit entre ces marqueurs au sein du nouveau code généré. Bien que cette approche supporte les générations successives de code après le début de l'implémentation, elles souffrent de deux problèmes majeurs [112] : les développeurs doivent insérer leur code au milieu du code généré, ce qui n'est pas toujours lisible et les marqueurs ne sont que des conventions que le compilateur est incapable de faire respecter. Certains langages de programmation généralistes permettent de couper une classe en plusieurs fichiers : c'est notamment le cas des classes partielles de C# et des classes ouvertes de Ruby. Avec ces langages, le code généré et le code écrit manuellement peuvent être séparés dans des fichiers différents, tout en faisant partie de la même classe [39].

DÉCOUVERTE D'ENTITÉS Pour permettre aux développeurs d'exprimer des requêtes de découverte d'entités, le compilateur de DiaSpec génère un DSL typé et enchâssé dans Java. Ce DSL s'inspire des travaux de Kabanov et Raudjärv [61] ainsi que des *fluent interfaces* de Fowler [38]. Beaucoup d'approches existantes pour la découverte d'entités, ou plus généralement de services, utilisent des chaînes de caractères pour décrire la requête [27, 83]. Le problème est que les erreurs dans les chaînes de caractères ne peuvent être détectées qu'à l'exécution du programme [61]. D'autres approches augmentent la grammaire du langage hôte [111], nécessitant la modification du compilateur ainsi que des environnements de développement. La technique que nous utilisons assure à la compilation que la requête est bien formée et ne nécessite pas de changement dans le compilateur du langage hôte ou dans les environnements de développement. La technique que nous utilisons est cependant utilisée par d'autres approches telles que l'API Criteria de Java EE 6 [50] et jMock [42].

5.6 TRAVAUX EN COURS ET FUTURS

Nous détaillons ici quelques perspectives et travaux en cours concernant le *framework* de programmation généré et l'implémentation s'appuyant dessus.

CHOIX DE LA TECHNOLOGIE DE DISTRIBUTION La distribution du code d'une application sur un réseau peut être utile dans certains des domaines couverts par le paradigme SCC. Par exemple, l'informatique ubiquitaire, et notamment le sous domaine de la domotique, requiert que des capteurs et actionneurs soient distribués dans les habitations et bâtiments [114]. Il existe de nombreuses technologies permettant de distribuer une application (RMI [27], CORBA [83], SIP [93], *etc.*). Sans abstraction supplémentaire, le choix de la technologie doit se faire dès le début de la conception de l'application car de ce choix va dépendre une grande partie du code. En effet, ces technologies sont invasives dans le sens où elles requièrent que la plupart du code dédié soit écrit par les développeurs à de nombreux emplacements dans le code applicatif. Pour pallier ce problème, le *framework* généré par le compilateur DiaSpec masque totalement la technologie de distribution. Ceci est réalisé grâce à une couche d'abstraction nommée DiaGenCore dont le *framework* généré dépend. Cette couche d'abstraction est ensuite reliée à la technologie choisie par l'administrateur système : la technologie de distribution peut donc être changée à tout moment du cycle de développement. La couche d'abstraction DiaGenCore est pour le moment compatible avec les technologies RMI, WebService et SIP. Plusieurs travaux ont déjà été publiés montrant l'intérêt de notre approche comme abstraction de la technologie SIP [9, 10, 60]. Il est possible d'adapter DiaGenCore à d'autres technologies en fonction des besoins. Il serait ainsi intéressant d'étudier les avantages que peuvent apporter certaines plateformes d'exécution comme FraSCAti [98] ou FUSE [58] en terme de support à l'exécution d'une application implémentée avec DiaSpec. Ces plateformes sont en effet capables de gérer des mécanismes de reconfiguration de l'application à l'exécution mais aussi des problèmes réseaux, ce qui n'est actuellement pas le cas avec DiaGenCore.

GÉNÉRATION DES PARTIES MANUELLES Comme nous l'avons vu, la logique de l'application SCC doit être implémentée en Java. Ceci permet à l'approche d'être flexible : en effet, la logique peut être implémentée manuellement comme nous l'avons vu ci-dessus. Elle peut aussi utiliser des *frameworks* tiers, ce qui n'est pas possible avec une approche totalement générative telle que PervML [99], ou peut même être générée à partir

de modèles sans nécessiter d'adaptation dans le compilateur de DiaSpec. Il existe par exemple des calculs récurrents sur les flots de données qui se prêtent plus à la description qu'à l'implémentation : calculs de moyenne, de maximum, de majorité, *etc.* C'est ce que Nicolas Loriant propose avec DiaEsper, un langage de requêtes basé sur Esper [4] qui permet de décrire un opérateur de contexte plutôt que de l'implémenter. Esper est un projet libre permettant de traiter des flux d'événements (*Complex Event Processing* ou CEP [70]) grâce à un langage dédié proche de SQL. DiaEsper permet de décrire comment un opérateur de contexte doit gérer les événements qu'il reçoit. Ceci prend en compte le filtrage des événements à écarter, les analyses à appliquer aux événements acceptés, les calculs à effectuer et la donnée à renvoyer. À partir d'une description en DiaEsper, un compilateur dédié, qui réutilise l'AST de DiaSpec, génère une implémentation pour cet opérateur de contexte qui utilise le projet Esper pour gérer les événements. Nous pouvons aussi imaginer que certains opérateurs puissent avoir une logique proche de celle d'une machine à états. Dans ce cas, il pourrait être intéressant d'utiliser un langage graphique, tel qu'un diagramme états-transitions d'UML, pour représenter les différents états et transitions et de générer la plupart du code d'implémentation. Par sa flexibilité, notre approche permet de gérer ces différents cas sans avoir à modifier le compilateur de DiaSpec.

Le calcul de majorité est particulièrement utile pour la sûreté de fonctionnement et le n-version programming

CHOIX DU LANGAGE CIBLE Nous avons vu que le compilateur de DiaSpec génère du code Java. Java dispose de nombreux outils et bibliothèques de code qui facilitent son utilisation dans beaucoup de contextes, notamment pour le paradigme SCC. Cependant, nous pouvons nous demander quelles caractéristiques doit avoir un langage pour être une bonne cible pour le compilateur de DiaSpec. La caractéristique la plus importante est la présence de typage statique dans le langage cible. En effet, sans les types statiques, le compilateur peut vérifier beaucoup moins de propriétés sur le code des développeurs et ceux-ci peuvent facilement écrire du code qui ne respecte pas l'architecture. La présence de typage statique est aussi essentielle pour guider les développeurs dans l'implémentation de la logique. Une autre caractéristique importante est que le langage cible doit être suffisamment haut-niveau et pratique pour rester plus simple que l'utilisation exclusive de modèles. Nous nous sommes aussi demandés si le paradigme objet était le seul capable d'être utilisé dans le *framework* généré. Un travail a commencé [5] pour étudier l'impact d'un langage purement fonctionnel sur l'approche DiaSpec et le langage Haskell a été choisi. Par manque de temps

Nous étudions au chapitre suivant les vérifications possibles sur le code

4. http://esper.codehaus.org
5. https ://github.com/DamienCassou/Functional-DiaSpec

cette expérimentation n'a pas encore abouti à des conclusions sur les caractéristiques utiles à l'approche.

6

ÉVOLUTION D'UNE APPLICATION SCC

La maintenance et l'évolution sont des étapes importantes du développement de n'importe quelle application : elles représentent plus de 85% du coût total d'une application [33]. Ces étapes sont encore plus importantes dans les domaines où les entités peuvent être déployées ou enlevées à n'importe quel moment, et où les utilisateurs peuvent avoir des besoins changeants. C'est la cas notamment de certains domaines du paradigme SCC comme l'informatique ubiquitaire.

En outre, les méthodologies de développement agiles promeuvent le développement itératif et incrémental des applications [6, 40]. Ceci impose aux approches, qui visent à faciliter le développement, de fournir du support pour revenir en arrière dans les étapes du cycle de développement.

Dans ce chapitre nous montrons comment notre approche permet, et même facilite, le développement itératif d'applications. Ceci est illustré dans la Figure 17. Nous montrons aussi comment certains changements peuvent être effectués pendant l'exécution.

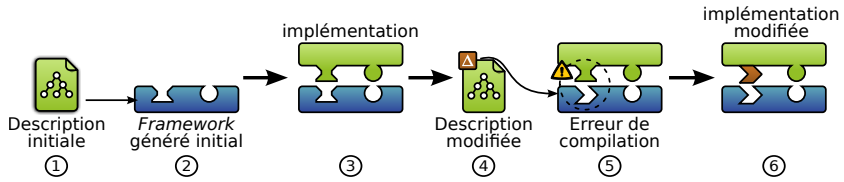

FIGURE 17: Les changements dans la description de l'application sont possibles, même après la génération du *framework* de programmation et le début de l'implémentation. Ces changements vont parfois déclencher des erreurs de compilation Java que l'environnement de développement du développeur rapportera. À l'étape ①, l'architecte écrit une description initiale. À l'étape ②, un *framework* de programmation est généré depuis la description et un développeur commence à implémenter en s'appuyant sur le *framework* à l'étape ③. À l'étape ④, l'architecte modifie la description. Un nouveau *framework* de programmation est généré à l'étape ⑤. Ce nouveau *framework* remplace le précédent, ce qui déclenche des erreurs de compilation Java dans l'implémentation existante. À l'étape ⑥, le développeur est guidé par les erreurs de compilation pour appliquer les changements à l'implémentation.

6.1 CHANGEMENTS DANS LA TAXONOMIE

Changer la taxonomie après que l'implémentation ait commencée, ou même après le déploiement, est possible et requiert une nouvelle génération du *framework*. Quand le nouveau *framework* est transmis aux développeurs, leur environnement de développement indique automatiquement les emplacements du code où des changements sont nécessaires. Voyons les principaux changements possibles.

AJOUT D'ENTITÉS Une nouvelle entité peut être déclarée dans la taxonomie à n'importe quel moment. Cette évolution ne nécessite aucun changement au-delà de l'implémentation de la classe abstraite nouvellement générée correspondante et du déploiement d'instances.

EXTENSION D'ENTITÉS Une déclaration de classe d'entités peut être étendue avec de nouvelles fonctionnalités (sources ou actions) et attributs. Comme dans une hiérarchie de classes, ces extensions ne requièrent pas de changement dans le code existant, mis à part l'implémentation de ces nouvelles fonctionnalités.

SUPPRESSION D'ENTITÉS OU DE FONCTIONNALITÉS Une déclaration de classe d'entités, ou une fonctionnalité, peut être supprimée. Le nouveau *framework* de programmation va faire déclencher des erreurs de compilation Java dans les parties du code qui utilisent ou implémentent l'élément supprimé. Ces erreurs concernent les développeurs d'entités et potentiellement les développeurs de l'application.

6.2 CHANGEMENTS DANS L'ARCHITECTURE

De la même façon, un architecte peut vouloir changer les descriptions des opérateurs de l'application. Cela nécessite aussi une nouvelle génération du *framework* qui peut mener à des erreurs de compilation. Celles-ci devront être corrigées par les développeurs de l'application.

AJOUT D'OPÉRATEURS De nouvelles déclarations d'opérateurs de contexte ou de contrôle peuvent être ajoutées. Ceci ne nécessite pas de changement de code au-delà de l'implémentation des nouvelles classes abstraites générées.

AJOUT DE CONTRATS D'INTERACTIONS Une déclaration d'opérateur peut être étendue par un nouveau contrat d'interac-

tions. Ceci va déclencher la génération d'une nouvelle méthode abstraite qu'un développeur devra implémenter.

MODIFICATION DE CONDITIONS D'ACTIVATION La condition d'activation d'un contrat d'interactions peut être changée. Si une disjonction est changée ou si un enfant est ajouté à la liste des valeurs poussées, aucun changement n'est nécessaire de la part des développeurs au-delà du changement dans le nom et les paramètres de la méthode correspondante. Si un enfant est supprimé de la liste des valeurs poussées, la donnée n'est plus accessible et une erreur de compilation apparaît à l'endroit où cette donnée est utilisée. Dans ce cas aussi, le nom de la méthode correspondante au contrat d'interactions peut changer, ce qui provoque une autre erreur de compilation. Enfin, si le changement concerne la façon dont les données arrivent (passage du mode de donnée poussée par les enfants vers le mode requête de parents), le nom de la méthode va changer ainsi que les données accessibles. Ceci provoquera des erreurs de compilation au niveau du nom de la méthode ainsi que de l'utilisation des données.

Cette fonctionnalité dépend de l'annotation `@Override` introduite dans Java 5

MODIFICATION DE DONNÉES REQUISES Une donnée requise peut être ajoutée ou supprimée à un contrat d'interactions. L'ajout d'une donnée requise à un opérateur n'affecte pas l'implémentation existante, mais de nouvelles fonctions d'interactions deviennent disponibles pour être appelées. La suppression d'une donnée requise engendre des erreurs de compilation là où l'interaction correspondante est invoquée.

MODIFICATION D'ACTIONS À ENTREPRENDRE L'action à entreprendre d'un contrat d'interactions peut être changée. Pour un opérateur de contexte, ce changement va affecter le type de la méthode correspondante au contrat d'interactions modifié et donc une erreur de compilation sera rapportée au niveau de l'instruction `return`. Ce changement peut également faire apparaître ou disparaître l'argument représentant la fonction de publication optionnelle. Pour un opérateur de contrôle, modifier l'action à entreprendre revient à ajouter ou supprimer des actions vers les entités. L'ajout d'une action à un opérateur de contrôle n'affecte pas l'implémentation existante, mais de nouvelles méthodes deviennent disponibles pour être appelées. La suppression d'une action engendre des erreurs de compilation là où l'action est invoquée.

SUPPRESSION DE CONTRATS D'INTERACTIONS La suppression d'un contrat d'interactions va rendre une partie du code mort. Cette situation est détectée par le compilateur Java qui

Cette fonctionnalité dépend de l'annotation `@Override` introduite dans Java 5

le rapporte comme une erreur et qui force le développeur à supprimer ce code mort.

CHANGEMENT DE TYPE D'OPÉRATEURS DE CONTEXTE Le type d'un opérateur de contexte peut être changé. Dans ce cas, l'implémentation de ce contexte et de tous ses parents doit être changée. Là encore, le compilateur Java affiche des messages d'erreur aux endroits à changer.

6.3 CHANGEMENTS DU SYSTÈME DÉPLOYÉ

Notre approche supporte quelques changements à l'exécution.

DÉPLOIEMENT OU SUPPRESSION D'ENTITÉS Des instances d'entités peuvent être déployées ou supprimées pendant l'exécution sans qu'il soit nécessaire de faire un quelconque changement. Les nouvelles instances s'enregistrent automatiquement et deviennent immédiatement disponibles à la découverte d'entités. La suppression d'une entité, par exemple par débranchement du matériel correspondant, va être détectée et celle-ci deviendra inaccessible à la découverte. En effet, le système de découverte d'entités vérifie périodiquement la présence des entités (Section 7.3.2, page 90).

DÉPLOIEMENT OU SUPPRESSION D'OPÉRATEURS De la même façon, de nouveaux opérateurs de contexte et de contrôle peuvent être déployés ou supprimés pendant l'exécution de l'application.

IMPLÉMENTATION ET DÉPLOIEMENT D'ENTITÉS Un développeur d'entités peut écrire une nouvelle implémentation d'une entité existante, c'est-à-dire créer une nouvelle sous-classe d'une classe abstraite générée, sans nécessiter d'autre changement. L'administrateur système pourra déployer de nouvelles instances de cette classe sans qu'il y ait lieu de redémarrer l'application. Ces instances vont s'enregistrer automatiquement pour pouvoir être découvertes.

6.4 SYNTHÈSE

Ce chapitre a présenté comment notre approche supporte les changements dans la taxonomie et l'architecture tard dans la phase de développement. Dans la plupart des cas, les changements ont un impact mineur sur le code existant. Dans tous les

cas, le compilateur Java indiquera les endroits à changer dans le code.

Nous avons également vu que certains changements sont possibles lors de l'exécution : l'ajout et la suppression de nouveaux éléments dans l'application se fait dynamiquement sans qu'il soit nécessaire de redémarrer l'application.

6.5 LIENS AVEC D'AUTRES APPROCHES

Certaines approches basées sur les modèles génèrent des squelettes de code que les développeurs peuvent remplir. Ces approches utilisent en général des marqueurs pour indiquer les zones dans lesquelles le développeur peut écrire son code [108]. Ces marqueurs sont insérés dans le code sous forme de commentaires. Ces approches n'écrasent pas le code qui se trouve entre les marqueurs quand elles génèrent de nouveau du code, après une modification de l'architecture. Notre approche remplace les marqueurs dans le code par l'utilisation de concepts de la programmation objets comme l'héritage et les méthodes abstraites. L'inconvénient principal de notre approche est que c'est au développeur de savoir quelles classes générées doivent être héritées et c'est aussi au développeur d'organiser ses sous-classes dans des paquetages si besoin. Cependant, un des avantages de notre approche est la possibilité d'avoir plusieurs implémentations pour une même description : il suffit dans ce cas d'implémenter plusieurs sous-classes de la même classe abstraite générée. Un autre avantage est que les fichiers sur lesquels travaillent les développeurs contiennent seulement leur code et pas du tout de code généré : ceci rend les fichiers plus compacts et plus clairs.

6.6 TRAVAUX EN COURS ET FUTURS

Dans notre approche, lorsque le *framework* de programmation est généré et que l'implémentation doit être changée, le compilateur Java indique les endroits à modifier. Les développeurs doivent alors effectuer manuellement les changements. Nous pensons qu'il est possible de faciliter un peu plus la tâche des développeurs en fournissant à l'expert de domaine et à l'architecte des outils de *refactoring*. Ceci pourrait se faire de plusieurs façons. Une des solutions les plus simples serait de générer des rapports de migration qui expliqueraient les changements de façon plus haut-niveau que le compilateur. Ce genre de rapports est fréquemment utilisé par les auteurs de bibliothèques de code qui veulent aider leurs utilisateurs à migrer vers une version plus récente. Certaines approches utilisent des rustines (ou *patchs*) sémantiques

pour propager un changement d'API dans tous les clients de cette API. L'outil Coccinelle a été spécialement développé pour propager automatiquement des changements dans l'API de Linux vers l'implémentation des pilotes de périphériques [85]. Une telle approche pourrait s'appliquer aussi au développement d'applications avec DiaSpec. Par exemple, lorsque l'architecte renomme un opérateur, un outil pourrait renommer les références à cet opérateur dans l'implémentation.

7

VÉRIFICATION D'UNE APPLICATION SCC

This automated transformation process is often referred to as "correct-by-construction," as opposed to conventional handcrafted "construct-by-correction" software development processes that are tedious and error prone.

— Douglas C. Schmidt [95]

L'approche présentée dans cette thèse permet aux outils de conduire des analyses au niveau de l'architecture, de l'implémentation ainsi que pendant l'exécution. Grâce au langage dédié DiaSpec et à ses abstractions haut-niveau, des analyses peuvent être conduites automatiquement sur la description de l'architecture des applications. Le *framework* de programmation généré permet d'assurer que les analyses conduites sur l'architecture sont toujours valables au niveau de l'implémentation. Enfin, des analyses sont conduites automatiquement pendant l'exécution de l'application pour garantir sa cohérence à tout moment.

7.1 ANALYSES SUR L'ARCHITECTURE

Le compilateur de DiaSpec conduit un ensemble d'analyses sur la description de l'architecture avant de générer du code. Nous ne parlons ici que des analyses sémantiques, les analyses syntaxiques étant automatiquement conduites à partir de la grammaire du langage DiaSpec décrite en ANTLR. Par exemple, la grammaire interdit à une entité d'utiliser le mot-clé `context`.

Voir Annexe B pour une définition complète de la grammaire

UNICITÉ DES NOMS Pour qu'il n'y ait aucune ambiguïté lorsqu'un composant de l'architecture dépend d'un autre, chaque nom donné par l'architecte doit être unique relativement à la portée de ce nom. La Figure 18 représente tout ce qu'une architecture DiaSpec et ses éléments définissent sous la forme d'un arbre. Un noeud de l'arbre représente un élément de l'architecture alors qu'un arc de l'arbre représente une relation de définition. Pour chaque noeud de l'arbre, tous les enfants doivent avoir un nom unique. Par exemple, les ensembles d'actions, les entités, les opérateurs de contexte et les opérateurs de contrôle doivent tous avoir des noms uniques. De même, une source d'entité possède

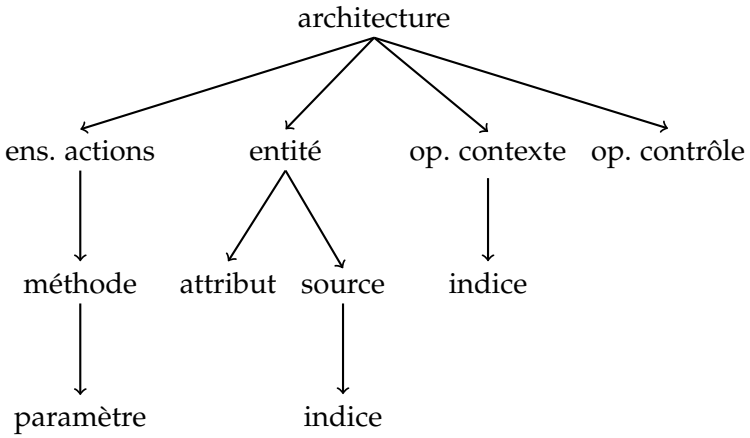

FIGURE 18: Arbre représentant ce que chaque élément de l'architecture peut définir

des indices de noms distincts. Cette unicité de noms doit aussi prendre en compte l'héritage entre entités : une entité ne peut pas définir une source ou un attribut avec le même nom qu'une source ou un attribut hérité.

UNICITÉ DES DÉPENDANCES Chaque opérateur ne doit déclarer qu'une fois une dépendance vers un autre opérateur ou une entité. De même, chaque entité ne doit déclarer qu'une fois une dépendance vers un ensemble d'actions.

EXISTENCE DES RÉFÉRENCES Lorsqu'un élément de l'architecture référence un autre élément de l'architecture, il faut que ce deuxième élément soit défini dans cette architecture. Par exemple, lorsqu'un opérateur de contexte déclare une dépendance avec une source s d'entité E, il faut que E existe et que E définisse une source s, ou en hérite.

PAS DE CYCLE DE DÉPENDANCE Pour empêcher les situations d'inter-blocages (*dead-locks*), le graphe des dépendances entre composants ne doit pas avoir de cycle. Grâce à la grammaire du langage DiaSpec, l'analyse syntaxique interdit tous les cycles sauf ceux entre contextes. Les cycles entre contextes sont interdits par l'analyse sémantique.

PAS DE CYCLE D'HÉRITAGE Une entité ne peut pas hériter d'elle-même, directement ou indirectement.

7.1.1 *Analyse des interactions*

Les contrats d'interactions basiques d'un opérateur ne peuvent référencer que des éléments de l'architecture avec lesquels l'opérateur déclare une dépendance.

7.1.1.1 *Extension des contrats d'interactions*

Pour simplifier l'expression des propriétés du reste de cette section, nous généralisons la définition des contrats d'interactions aux sources d'entités. Soit S une source d'entité, alors S possède deux contrats d'interactions basiques :

$$\begin{aligned} \mathit{Push} &: \langle\ \emptyset \quad ;\ \emptyset\ ;\ \Uparrow \mathtt{self?}\ \rangle \\ \mathit{Pull} &: \langle\ \Downarrow \mathtt{self}\ ;\ \emptyset\ ;\ \Uparrow \mathtt{self?}\ \rangle \end{aligned}$$

Le premier contrat indique que la source d'entité peut publier des événements sans y avoir été invité, c'est-à-dire qu'une source peut être proactive (Section 4.4.1). Le deuxième contrat indique que la source d'entité peut répondre à des requêtes.

Dans la suite, nous utilisons aussi la notation $\mathtt{self}[?]$ pour indiquer que l'émission d'un contrat d'interactions basique est égale soit à self soit à self?.

7.1.1.2 *Absence de contrat d'interactions*

Lorsqu'un opérateur ne définit pas de contrat d'interactions, le compilateur lui en associe automatiquement un très permissif. Pour un opérateur de contexte qui dépend de sources d'entités $S_1, \ldots, S_m$ et d'autres opérateurs de contexte $C_1, \ldots, C_n$ avec $m \geqslant 0$ et $n \geqslant 0$, le contrat d'interactions associé automatiquement contient exactement les $m + n + 1$ contrats d'interactions basiques suivants :

$$\begin{aligned}
&\langle \Downarrow \mathtt{self}\ ;\ \Downarrow (S_1, \ldots, S_m, \ldots, C_i, \ldots)\ ;\ \Uparrow \mathtt{self?}\ \rangle \\
&\langle \Uparrow S_1 \quad ;\ \Downarrow (S_1, \ldots, S_m, \ldots, C_i, \ldots)\ ;\ \Uparrow \mathtt{self?}\ \rangle \\
&\langle \ \ldots \quad ;\ \ldots \qquad\qquad\qquad\qquad ;\ \ldots \quad \rangle \\
&\langle \Uparrow S_m \quad ;\ \Downarrow (S_1, \ldots, S_m, \ldots, C_i, \ldots)\ ;\ \Uparrow \mathtt{self?}\ \rangle \\
&\langle \Uparrow C_1 \quad ;\ \Downarrow (S_1, \ldots, S_m, \ldots, C_i, \ldots)\ ;\ \Uparrow \mathtt{self?}\ \rangle \\
&\langle \ \ldots \quad ;\ \ldots \qquad\qquad\qquad\qquad ;\ \ldots \quad \rangle \\
&\langle \Uparrow C_n \quad ;\ \Downarrow (S_1, \ldots, S_m, \ldots, C_i, \ldots)\ ;\ \Uparrow \mathtt{self?}\ \rangle
\end{aligned}$$

Dans ces contrats, l'opérateur de contexte C_i appartient à la liste des données requises si et seulement si il a un contrat d'interactions basique pour lequel la condition d'activation est $\Downarrow self$.

Pour un opérateur de contrôle qui dépend d'actions d'entités $A_1, \ldots, A_m$ et d'opérateurs de contexte $C_1, \ldots, C_n$ avec $m > 0$ et $n \geqslant 0$, le contrat d'interactions associé automatiquement contient exactement les n contrats d'interactions basiques suivants :

$$\begin{array}{l} \langle \Uparrow C_1 \; ; \Downarrow (\ldots, C_i, \ldots) ; \Uparrow (A_1, \ldots, A_m) \rangle \\ \langle \ldots \quad ; \ldots \qquad\qquad ; \ldots \qquad\qquad \rangle \\ \langle \Uparrow C_n ; \Downarrow (\ldots, C_i, \ldots) ; \Uparrow (A_1, \ldots, A_m) \rangle \end{array}$$

Dans ces contrats, l'opérateur de contexte C_i appartient à la liste des données requises si et seulement si il a un contrat d'interactions basique pour lequel la condition d'activation est $\Downarrow self$.

7.1.1.3 *Utilité des contrats d'interactions*

Une définition de contrat d'interactions basique peut être inutile au sein d'une architecture : c'est le cas lorsque aucun flot de données ne satisfait la condition d'activation. On dit dans ce cas que le contrat d'interactions basique n'est pas *activable*. De plus, un contrat d'interactions basique peut ne pas être *exécutable* s'il propose de faire des requêtes sur des opérateurs qui ne fournissent pas cette interaction.

Soit un opérateur O qui a pour contrat d'interactions basique K de la forme $\langle A_o ; U_o ; E_o \rangle$, alors K est dit activable si une des deux conditions suivantes est satisfaite :

- $A_0 = \Uparrow C_1, \ldots, C_n$ et $\forall i \in [1, n]$, soit
  - C_i est le nom d'un enfant de O qui a un contrat d'interactions basique *activable* de la forme $\langle \_ ; \_ ; \Uparrow self[?] \rangle$, soit
  - C_i est une disjonction $(C_{i1} \vee \ldots \vee C_{im})$ et $\exists j \in [1, m]$, C_{ij} est le nom d'un enfant de O qui a un contrat d'interactions basique *activable* de la forme $\langle \_ ; \_ ; \Uparrow self[?] \rangle$
- $A_0 = \Downarrow self$ et $\exists C$ un parent de O qui a un contrat d'interactions basique *activable* de la forme $\langle \_ ; \Downarrow (\ldots, O, \ldots) ; \_ \rangle$

Un contrat d'interactions basique non activable peut être supprimé sans affecter l'architecture : en effet, l'implémentation de la méthode générée correspondante ne sera jamais appelée. On dit qu'un opérateur est activable, si et seulement si au moins un de ses contrats d'interactions basiques est activable. Si un opérateur n'est pas activable, il peut être supprimé sans affecter l'architecture.

Soit un opérateur O qui a pour contrat d'interactions basique K de la forme $\langle A_o; U_o; E_o \rangle$, alors K est dit exécutable si :

- $U_0 = \Downarrow C_1, \ldots, C_n,\ n \geqslant 0$ et $\forall i \in [1, n],\ C_i$ a un contrat d'interactions basique *exécutable* $\langle \Downarrow \mathtt{self}; \_; \_ \rangle$

Un contrat d'interactions basique non exécutable est susceptible de déclencher des erreurs à l'exécution : en effet, l'implémentation de la méthode générée correspondante peut faire une requête vers une source ou un opérateur de contexte qui ne permet pas cette interaction. Un contrat d'interactions basique non exécutable représente donc une incohérence et doit être supprimé pour éviter des erreurs à l'exécution.

En pratique, le compilateur de DiaSpec notifie l'architecte si un contrat d'interactions basique est non activable ou non exécutable.

7.1.1.4 *Non interférence des contrats d'interactions basiques*

Deux contrats d'interactions basiques d'un opérateur ne doivent pas être activés par une même entrée (événement ou requête) car une telle entrée rendrait l'exécution de l'application non déterministe. Soit O un opérateur qui possède deux contrats d'interactions basiques $\langle A_1; \_; \_ \rangle$ et $\langle A_2; \_; \_ \rangle$ et C une dépendance de O, on dit que deux contrats interfèrent si :

- $A_1 = A_2 = \Downarrow \mathtt{self}$ ou
- O apparaît dans A_1 et dans A_2

On dit qu'un contrat d'interactions est *déterministe* s'il n'existe pas deux de ses contrats d'interactions basiques qui interfèrent. En pratique, le compilateur s'assure que tous les contrats d'interactions sont déterministes. Cette propriété permet d'assurer qu'une même entrée activera toujours de la même façon un opérateur. De plus, cette propriété est indispensable à l'unicité des noms des méthodes abstraites générées car ceux-ci sont basés sur la condition d'activation.

Nous venons de voir que le compilateur de DiaSpec effectue un ensemble de vérifications permettant à l'architecte d'avoir confiance en l'intégrité et la cohérence de son architecture.

7.2 ANALYSES SUR L'IMPLÉMENTATION

Vérifier qu'une implémentation est conforme à une architecture est un travail complexe et maintenir cette conformité l'est encore plus [18, 25, 100, 106]. Le *framework* généré par le compilateur de DiaSpec permet de garantir que l'implémentation est conforme à l'architecture décrite en DiaSpec. On dit que l'implémentation est

« correcte par construction », à l'opposé de « construite par correction » [95]. Luckham et Vera [71] distinguent trois critères de conformité d'une implémentation par rapport à une architecture :

DÉCOMPOSITION Pour chaque composant dans l'architecture, il doit y avoir un composant dans l'implémentation correspondant ;

CONFORMITÉ D'INTERFACE Chaque composant dans l'implémentation doit se conformer à son interface architecturale ;

INTÉGRITÉ DE COMMUNICATION Chaque composant dans l'implémentation doit seulement communiquer avec des composants auxquels il est connecté dans l'architecture.

Dans le cadre de DiaSpec, nous appelons « composant dans l'implémentation » une classe concrète qui hérite d'une classe abstraite générée dans le *framework*, ou une instance d'une telle classe concrète.

DÉCOMPOSITION Ce critère est vérifié dans le sens où une classe abstraite est générée pour chaque composant de l'architecture. Cependant, le *framework* n'impose pas que toutes les classes abstraites soient implémentées ni que des instances soient créées : ceci permet de déployer les applications de façon incrémentale et en fonction des besoins.

CONFORMITÉ DE L'INTERFACE Le compilateur de DiaSpec génère, pour chaque description de composant, une classe abstraite qui est conforme par construction à cette description (Section 5.1.1, page 52, et Section 5.1.2, page 54). En héritant de cette classe abstraite, l'implémentation doit se conformer à la description : le compilateur Java vérifie cela automatiquement. Par exemple, lorsqu'une classe d'entités déclare un ensemble d'actions, la classe abstraite correspondante contient une méthode abstraite par action. Autre exemple, lorsqu'un contrat d'interactions basique d'un opérateur de contexte déclare une émission $\Uparrow$ `self`, le type de retour de la méthode abstraite généré impose à l'implémentation de retourner un objet de type compatible. Cet objet est ensuite publié automatiquement par le *framework*.

INTÉGRITÉ DE COMMUNICATION Pour satisfaire ce critère, le *framework* de programmation fournit aux implémentations les seuls moyens d'interactions autorisés. Par exemple, la classe abstraite associée à une déclaration d'entité ne contient aucun moyen d'initier une communication avec un autre composant de l'architecture autrement que par l'envoi d'événements : de plus, les événements que l'entité peut envoyer sont uniquement ceux décrits

dans l'architecture. Pour un opérateur, toutes les interactions se font durant l'exécution des méthodes `onNew` et `get`, et seulement au travers des fonctions passées en paramètre. La découverte d'entités n'est accessible qu'aux opérateurs et chaque opérateur ne peut découvrir que les classes d'entités dont il dépend (Section 5.2.3, page 66). Cette contrainte est assurée statiquement par le compilateur Java grâce au *framework* de programmation généré. De plus, le *framework* de programmation permet d'appliquer des filtres sur les requêtes de découverte d'entités mais seulement pour les attributs des classes de ces entités.

Il est important de noter que le *framework* généré garantit que l'implémentation est conforme à l'architecture, seulement lorsque les développeurs utilisent « proprement » le *framework*. Malgré toutes les protections placées dans le *framework* pour éviter une mauvaise utilisation, il est toujours possible à un développeur de passer outre en utilisant certains mécanismes de Java comme la réflexion.

7.3 ANALYSES SUR L'EXÉCUTION

Le langage Java et son compilateur ne sont pas capables de vérifier toutes les contraintes architecturales statiquement. Des vérifications dynamiques de l'implémentation sont donc nécessaires pour assurer que l'implémentation ne contient pas des fautes non détectables à la compilation. De plus, garantir qu'une implémentation est conforme à une architecture n'est pas suffisant : en effet, un composant peut être correctement implémenté, mais s'il n'est jamais déployé l'application peut ne pas fonctionner comme attendu. Il est donc nécessaire d'ajouter des vérifications à l'exécution pour détecter au plus tôt les problèmes pouvant survenir et en avertir l'administrateur.

7.3.1 *Vérifications dynamiques de l'implémentation*

Le langage Java et son système de types nous ont permis de concevoir un *framework* de programmation qui permet de garantir que la plupart des contraintes exprimées par l'architecte sont respectées dans l'implémentation. Cependant, le langage Java est toujours trop permissif et il ne permet pas de tout assurer statiquement. Par exemple, le *framework* généré vérifie qu'un opérateur de contrôle exécute toujours au moins une action lorsqu'il est activé. Ceci se fait en utilisant le type `Actions` comme type de retour des méthodes abstraites générées. Cependant, rien n'empêche une implémentation d'utiliser la valeur spéciale `null`, qui satisfait le compilateur mais pas les contraintes du paradigme

SCC. Pour pallier les limitations du langage Java, nous avons mis en place quelques vérifications dynamiques de l'implémentation. Ces vérifications sont amenées à être remplacées au fur et à mesure de l'évolution du langage Java. Elles ne seraient pas forcément nécessaires en présence d'un autre langage de programmation.

RÉUTILISATION DE FONCTION Les fonctions passées en paramètre aux opérateurs permettent d'initier des interactions optionnelles. Ces fonctions sont construites par les méthodes appelantes qui les transmettent aux implémentations des méthodes abstraites. Pour que les contrats d'interactions soient respectés, il est nécessaire de vérifier qu'une telle fonction n'est pas appelée après le retour de l'exécution de la méthode : cela signifie qu'un développeur ne doit pas stocker la fonction pour l'utiliser plus tard. Cette propriété ne peut pas être vérifiée par le compilateur Java. En contrepartie, le *framework* généré fournit une *barrière dynamique* qui assure cette propriété à l'exécution. Intuitivement, la barrière est levée lorsque la méthode appelante appelle l'implémentation de la méthode abstraite et rabaissée immédiatement après. La fonction d'interaction exécute l'interaction si la barrière est levée et, à défaut, fait échouer l'application. Le Listing 11 montre un exemple d'utilisation des barrières dynamiques.

EXÉCUTION D'ACTION Le paradigme SCC indique que la couche de contrôle doit faire exécuter des actions à la couche d'action. Le *framework* généré doit donc contraindre chaque opérateur de contrôle à exécuter au moins une action sur une entité chaque fois qu'il est activé. Cette contrainte est en partie assurée statiquement par le type de retour des méthodes abstraites générées à partir des contrats d'interactions basiques : en effet, chaque méthode abstraite pour un opérateur de contrôle doit retourner un objet de type `Actions` et les instances de ce type ne peuvent être créées que lors d'exécution d'actions. Par exemple, le Listing 10 montre que l'implémentation retourne le résultat de l'exécution de l'action `log()`. Cependant, le compilateur `Java` ne permet pas, pour l'instant, de vérifier qu'une implémentation retourne une instance de la classe `Actions` et pas simplement la valeur `null`. Une barrière dynamique est là encore utilisée pour empêcher la valeur `null` d'être retournée. Cette barrière est initialement levée permettant à l'opérateur de contrôle de commander des actions. Si à une occasion l'implémentation de la méthode abstraite retourne `null`, la barrière est définitivement baissée empêchant toute activation suivante de l'opérateur. Enfin, une exception est lancée pour prévenir l'administrateur.

```
1  public abstract class AbstractAccessingProfile {
2    ...
3    // classe interne contenant la définition de la fonction
4    protected class PullFromIP2ProfileCallback {
5      // la barrière dynamique
6      private boolean isActive = false;
7      // la fonction
8      public Profile get(IPAddress ipAddress) {
9        if (!isActive)
10         throw new IllegalAccessError("Illegal call");
11       // transfère la requête à l'instance déployée
12       // de l'opérateur IP2Profile
13       return ...;
14     }
15   }
16
17   // la méthode appelante
18   private void callOnAccessLogParser(Access value) {
19     PullFromIP2ProfileCallback c =
20       new PullFromIP2ProfileCallback();
21
22     IdentifiedAccess newValue;
23     try {
24       c.isActive = true; // levée de la barrière
25       // appelle la méthode implémentée
26       newValue = onNewAccessLogParser(value, c);
27     } finally {
28       c.isActive = false; // abaissement de la barrière
29     }
30     // publie la valeur retournée 'newValue'
31     ...
32   }
33
34   protected abstract IdentifiedAccess onNewAccessLogParser(
35           Access newAccessLogParser,
36           PullFromIP2ProfileCallback getIP2Profile);
37   ...
38 }
```

Listing 11: Classe abstraite générée à partir de la description de l'opérateur de contexte AccessingProfile (Listing 2, lignes 12 à 15, page 37 et Table 1, page 43) montrant l'utilisation des barrières dynamiques

7.3.2 *Vérification dynamiques de l'exécution*

Comme nous l'avons vu au début de cette thèse, les applications SCC sont souvent, en partie ou totalement, distribuées. Cette distribution peut rendre certaines interactions entre composants impossibles, par exemple, lorsqu'un problème réseau se produit. De plus, lors d'un déploiement incrémental de l'application, certains composants peuvent apparaître et d'autres disparaître en fonction des besoins. Il est donc nécessaire de faire des vérifications durant l'exécution pour connaître les composants disponibles pour les interactions.

ÉTAT D'UN COMPOSANT Les instances d'entités et d'opérateurs pouvant être déployées en réseau sur des machines différentes, il est possible à tout moment qu'une partie de l'application cesse de fonctionner tandis qu'une autre est toujours en fonction. Ces situations doivent être détectées car en l'absence de certains composants, c'est l'intégrité de l'application complète qui peut être mise en cause. Pour détecter cela chaque composant envoie, à un intervalle prédéfini, un message à un serveur dédié pour indiquer qu'il est toujours en vie : nous appelons ces messages un battement de cœur (ou *heartbeat*). Si le serveur ne reçoit pas de message d'un composant pendant une période paramétrable, il déclare ce composant comme ayant cessé de fonctionner.

EXISTENCE D'ENTITÉ La découverte d'entités permet d'interagir avec des instances d'entités sans qu'il soit nécessaire de connaître leur emplacement physique sur le réseau (Section 5.2.3, page 66). Pour prendre en compte le déploiement incrémental d'entités, cette découverte doit se faire le plus tard possible. En effet, la découverte doit se faire au moment où l'interaction avec les entités doit se produire et non au moment où la découverte est exécutée. Ceci est particulièrement important lors de la souscription aux événements envoyés par une entité. Par exemple, dans l'implémentation de l'opérateur de contexte `AccessLogParser`, le développeur exprime que l'opérateur est intéressé par tous les événements venant des instances d'entités `AccessLogReader` (Listing 7, ligne 12, page 63) : cette souscription est valide pour les entités actuellement déployées mais aussi pour les entités qui seront déployées plus tard. Grâce à cela, si une nouvelle instance de la classe d'entités `AccessLogReader` est déployée, l'opérateur de contexte souscrira automatiquement à ses événements sans qu'il soit nécessaire de faire une nouvelle découverte d'entités.

7.4 SYNTHÈSE

Nous avons vu dans ce chapitre que des analyses sont nécessaires à la fois sur l'architecture, l'implémentation et l'exécution de l'application. Ces analyses sont nécessaires pour garantir que les contraintes du paradigme SCC sont respectées.

Ces contraintes sont garanties sur l'architecture en grande partie grâce à la syntaxe dédiée du langage DiaSpec. L'analyseur sémantique du compilateur de DiaSpec conduit des analyses supplémentaires pour les contraintes qui ne sont pas garanties par la syntaxe.

En ce qui concerne l'implémentation, le *framework* généré permet de garantir qu'à la fois les contraintes du paradigme SCC et celles de l'architecture de l'application sont respectées. Ce dernier point est un des atouts majeurs de notre approche générative. En effet, les approches qui visent à fournir des garanties architecturales dans l'implémentation n'en sont souvent capables que pour les contraintes du paradigme et pas pour les contraintes de l'architecture. Par exemple, dans le cadre du paradigme SCC, les approches seraient capables de vérifier qu'un opérateur de contrôle ne prend pas de données d'une source d'entités. Cependant, elles ne seraient pas capables de vérifier qu'un opérateur de contrôle ne prend ses données que d'un opérateur de contexte donné et pas d'un autre. Grâce à la génération d'un *framework* de programmation dédié à la description d'une architecture d'une application, ce genre de garantie est possible.

Au niveau de l'exécution, deux types d'analyses sont nécessaires. Il faut tout d'abord vérifier les contraintes architecturales que le langage généraliste sous-jacent n'est pas capable d'assurer statiquement. Par exemple, Java n'est pas capable, actuellement, de garantir qu'une valeur est non nulle à la compilation. Il faut donc tester certaines propriétés à l'exécution, notamment grâce à des instructions de branchement et des barrières. Un autre type d'analyses est nécessaire lors de l'exécution, il s'agit d'analyses qui concernent le déploiement des composants. Dans un domaine où la mise en réseau des composants est nécessaire, certaines parties de l'application peuvent cesser de fonctionner à tout moment. De plus, dans le cadre d'un déploiement incrémental, l'administrateur peut décider d'ajouter ou supprimer des composants pendant que l'application est en fonction. Dans ces cas, des analyses sont nécessaires pour établir la liste des composants en fonction à un instant donné.

7.5 LIENS AVEC D'AUTRES APPROCHES

UML, reconnu comme un ADL par Medvidovic et al. [80], est utilisé pour décrire des architectures logicielles. Grâce à sa notion de profils, UML peut même être utilisé en tant que langage de description dédié à un domaine. Quand la description de l'application est suffisamment détaillée, comme dans les approches de l'ingénierie dirigée par les modèle [95], il peut être possible de générer toute l'implémentation de l'application. Dans la plupart des cas cependant, UML est seulement utilisé pour décrire certains aspects d'une application [37] : un diagramme de classes avec les classes les plus importantes, par exemple, ou un diagramme de séquence pour décrire une interaction particulière entre quelques objets. Dans ces cas là, il n'est pas possible de générer toute l'application. Les développeurs doivent alors considérer les diagrammes UML comme des artefacts contemplatifs, et s'assurer manuellement de la cohérence entre l'implémentation et la description de l'architecture . Maintenir cette correspondance dans le temps est une tâche reconnue difficile [25, 106]. Elle est cependant nécessaire pour maintenir les qualités initiales de l'application et ainsi éviter l'érosion du logiciel (*software erosion*). Grâce à notre compilateur qui génère un *framework* de programmation dédié à une description d'architecture, les développeurs écrivent toujours du code cohérent avec l'architecture.

Plus récemment, d'autres ADLs tels que ArchJava [2, 3], ComponentJ [97], ACOEL [104] et Archface [108] ont permis d'utiliser les architectures logicielles pour vérifier les implémentations. Ces approches font le lien entre les architectures logicielles et leurs implémentations grâce à un couplage fort. ArchJava, ACOEL et ComponentJ modifient le langage Java en lui ajoutant les concepts de composant et connecteur. Grâce à ce couplage fort, ces approches permettent de vérifier qu'une implémentation est bien conforme à son architecture : ArchJava vérifie par exemple la propriété d'intégrité de communication [71], qui indique que toutes les interactions entre composants doivent être décrites dans l'architecture. Cependant, ces approches impliquent le mélange des concepts d'architecture et d'implémentation ce qui rend les deux plus difficiles à utiliser. Pour regagner une séparation entre l'architecture et l'implémentation, Archface [108] propose un nouveau mécanisme d'interface qui fait levier sur les concepts de la programmation orientée aspects (AOP) pour décrire les interactions de composants. Les points de coupure de l'AOP sont utilisés comme abstraction de la structure de l'implémentation. Cependant, les points de coupures ne permettent pas de s'abstraire totalement de l'implémentation car ils sont dépendants de la structure de celle-ci. Les abstractions proposées par DiaSpec sont complètement indépendantes de la structure

de l'implémentation : grâce à cela, l'architecte peut se concentrer sur la description de l'application sans prendre en compte des considérations d'implémentation.

7.6 TRAVAUX EN COURS ET FUTURS

Nous détaillons ici quelques perspectives et travaux en cours concernant les analyses faites sur l'architecture, l'implémentation et l'exécution.

VÉRIFICATION DES FLOTS DE DONNÉES La sûreté de fonctionnement est un besoin important dans plusieurs domaines couverts par le paradigme SCC, comme l'avionique ou l'informatique ubiquitaire. Ceci vient du fait qu'une application SCC peut impacter directement l'environnement et les utilisateurs au travers d'actionneurs. Les contrats d'interactions rendent explicites certaines informations importantes sur le flot de données dès la description de l'architecture. Grâce à cette explicitation, des vérifications sur les flots de données sont possibles. Par exemple, l'architecte pourrait souhaiter être assuré que l'opérateur `ProfileLogger` soit toujours activé lorsqu'un client se connecte sur le serveur web (Figure 8, page 35). Ce type de garanties peut être exprimé grâce, par exemple, à des formules LTL (*Linear Temporal Logic* [56]), puis vérifié avec un vérificateur de modèles tel que SPIN [55].

VÉRIFICATIONS DE L'IMPLÉMENTATION Nous avons vu dans ce chapitre que certaines vérifications de l'implémentation ne peuvent se faire qu'à l'exécution de l'application. Dans un cadre idéal, ces vérifications devraient être faites avant l'exécution pour ne pas permettre l'exécution d'applications incorrectes. Nous pouvons envisager plusieurs solutions à ce problème. La plus simple des solutions est d'attendre que le langage Java évolue et de changer le *framework* pour qu'il prenne en compte les dernières évolutions. Par exemple, il a été proposé [1] que Java inclut un mécanisme pour indiquer qu'une valeur ne peut pas être nulle. Avec un tel mécanisme, le *framework* n'a plus besoin d'utiliser une barrière pour s'assurer qu'un opérateur de contrôle ne retourne jamais une action nulle. Une autre solution est d'utiliser un langage de programmation qui possède déjà un compilateur avec les vérifications nécessaires. Haskell étant reconnu pour son système de types, nous avons débuté l'implémentation d'un prototype qui génère du code Haskell au lieu de code Java. Comme indiqué précédemment (Chapitre 5, page 73), nous n'avons pas suffisamment avancé le prototype pour tirer la moindre conclu-

1. JSR-305 http://jcp.org/en/jsr/detail?id=305

sion. Une solution moins radicale est d'utiliser un outil dédié pour analyser le code avant son exécution. Des approches comme JastAdd [30] ou Julia [103] permettent ce type d'analyse sur du code Java et sont, par exemple, capables d'indiquer si une variable est garantie d'être non nulle.

Troisième partie

MISE EN OEUVRE

8 ÉVALUATION

Dans cette section, nous conduisons une évaluation de DiaSpec et des outils associés. Pour se faire, nous explorons trois aspects : (1) l'*expressivité*, évaluant la portée du paradigme SCC et du langage DiaSpec, (2) l'*utilisabilité*, estimant la facilité d'utilisation des outils, et (3) la *productivité*, mesurant le temps de développement, la qualité du code ainsi que la capacité à réutiliser le code.

8.1 EXPRESSIVITÉ

Au Chapitre 1, nous avons vu que le paradigme SCC pouvait être appliqué à de nombreux domaines. Dans cette section, nous vérifions que le langage DiaSpec est capable de couvrir ces domaines en présentant certaines des applications que nous avons développées.

8.1.1 *Les interfaces utilisateurs*

Pour illustrer son cours d'introduction sur les architectures logicielles, Schmidt [94] présente le paradigme MVC ainsi qu'une application graphique représentant un cœur. Ce cœur, le modèle, est soit « heureux », soit « triste ». Il peut aussi être de deux tailles différentes : « gros » ou « petit ». Deux fois par seconde, le cœur « bat » et change de taille. La Figure 19 montre l'application en fonction. Nous pouvons y voir à gauche la fenêtre principale avec une représentation du cœur et un bouton. Le cœur est rouge quand il est heureux et bleu s'il est triste. Le bouton permet de passer d'un état à un autre.

Deux fois par seconde, le coeur change de taille. À droite de la Figure 19, deux fenêtres plus petites montrent le compte des battements dans chacun des deux états.

La Figure 20 présente une vue UML de l'application cœur telle que décrite par Schmidt et utilisant le paradigme MVC. Le modèle est composé d'une classe `Heart` et d'une classe `Counter`. Une instance de la classe `Heart` possède deux instances de la classe `Counter`, une pour chaque état. Ces deux classes sont représentées graphiquement par deux vues nommées `HeartViewer` et `MoodViewer`. La classe `Hearbeat` est un contrôleur qui rythme

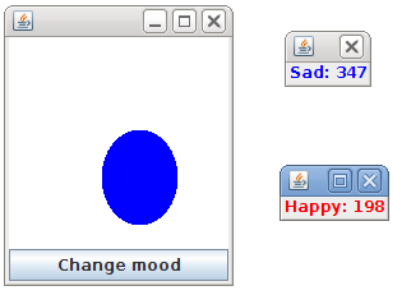

FIGURE 19: L'application cœur en marche

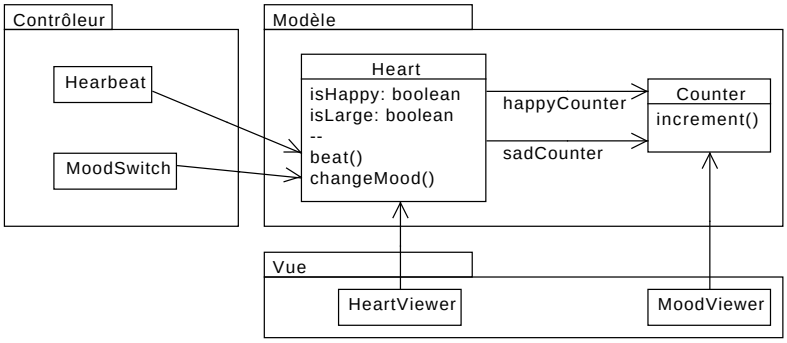

FIGURE 20: L'application cœur décrite avec UML (adapté de [94])

les battements cardiaques du cœur. En Java, cette classe est implémentée par un *thread* qui exécute une boucle infinie, appelant la méthode `beat()` sur le cœur puis s'endormant pendant une demi seconde. Le contrôleur `MoodSwitch` est un bouton qui permet à l'utilisateur de changer l'état du cœur quand il le désire. Comme le paradigme MVC le recommande, les classes du modèle ne dépendent ni des contrôleurs ni des vues. Ceci est implémenté en partie grâce au motif de conception *Observateur* [44] : les vues s'abonnent au modèle qui les informe quand il change.

La même application a été implémentée en DiaSpec suivant le paradigme SCC (Figure 21). La couche de capture s'occupant de récupérer des informations de l'environnement, et donc de l'utilisateur, c'est là que se placent les contrôleurs du paradigme MVC. Nous retrouvons donc, en bas de la Figure 21, les deux sources `MoodSwitch` et `Heartbeat`, qui fournissent respectivement l'information du clic sur le bouton de changement d'état ainsi que les battements cardiaques. La source `beat` de l'entité `Heartbeat` est utilisée par l'opérateur de contexte `HeartSize` qui gère la taille du cœur. L'opérateur de contexte `HeartMood` gère l'état courant du cœur tandis que l'opérateur de contexte `HeartCounter` compte les battements cardiaques pour chaque état. L'opérateur `HeartController` reçoit l'état du cœur, le nombre de battements dans

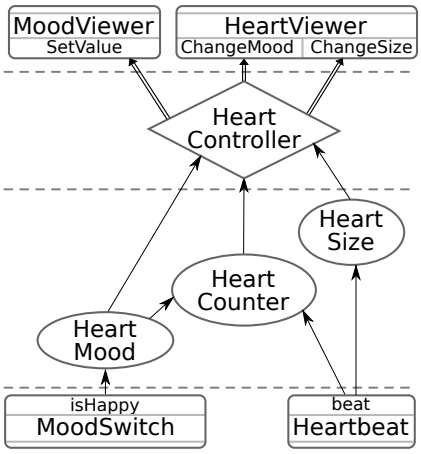

FIGURE 21: L'application cœur décrite avec DiaSpec

cet état ainsi que la taille actuelle du cœur et demande aux vues de se mettre à jour.

Une des différences notables entre l'implémentation en MVC et l'implémentation en DiaSpec est la séparation de l'état et de la taille du cœur en deux opérateurs de contexte. Cette propension à « éclater » les parties d'un même modèle en plusieurs opérateurs de contexte se retrouve aussi dans d'autres applications développées avec DiaSpec. Le paradigme SCC encourage ce découpage à grains fins qui favorise la réutilisation. L'architecte devra néanmoins faire attention à ne pas concevoir des composants à grains trop fins qui obligent les développeurs à implémenter beaucoup de classes très petites. Malgré cette différence, le paradigme SCC est suffisamment proche du paradigme MVC pour que le passage d'une architecture MVC à une architecture SCC soit simple.

8.1.2 *L'avionique*

Dans le cadre d'un partenariat avec l'entreprise Thales [1], nous avons développé une application de pilote automatique d'avions en DiaSpec. La Figure 22 représente un avion avec ses axes de rotation. L'altitude de l'avion peut être changée en modifiant l'angle vertical, ou *pitch*, qui est lui-même contrôlé par des élévateurs.

Pour une introduction à l'avionique, consulter [19]

La Figure 23 présente un extrait de l'application de pilote automatique. Les entités `InertialUnit` et `AirDataUnit` fournissent des informations sur l'état actuel de l'avion (le *pitch*, la vitesse verticale et l'altitude). L'entité `AutoPilotGui` fournit une interface graphique permettant au pilote de changer le mode d'utilisation du *pitch* ainsi que l'altitude désirée. L'entité `RouteManager` gère le plan de vol de l'avion et fournit à tout moment le prochain point de passage (*Way Point*) à atteindre. À partir de ces informations

1. http://www.thalesgroup.com/

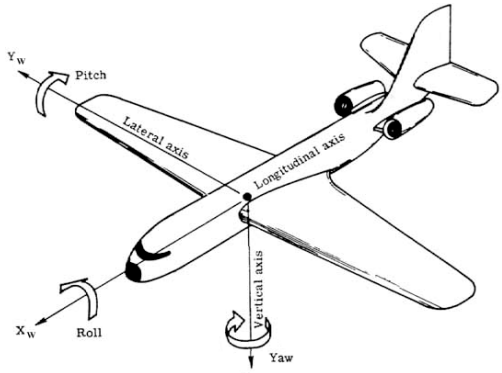

FIGURE 22: Représentation d'un avion et de ses différents axes

les opérateurs de contexte calculent (1) l'altitude à atteindre, (2) la vitesse verticale pour atteindre cette altitude, (3) le *pitch* nécessaire pour atteindre cette vitesse verticale et (4) la commande à appliquer à l'élévateur pour atteindre ce *pitch*. L'opérateur de contrôle transmet la commande aux élévateurs.

L'application complète de pilote automatique consiste en 9 entités, 33 opérateurs de contexte ainsi que 8 opérateurs de contrôle. Cette application est simulée grâce au simulateur d'avionique libre FlightGear [2] (Figure 24). Pour cette simulation, l'environnement réel de l'avion est remplacé par l'application FlightGear. Les sources d'entités transmettent les données de vol calculées par FlightGear et émises au travers d'une *socket* réseau. Les actions d'entités sont transférées à FlightGear au travers d'autres *sockets* réseau.

8.1.3 *L'informatique ubiquitaire*

Nous avons implémenté et déployé plusieurs applications domotiques dans une salle dédiée de notre laboratoire (Figure 25). Nous donnons quelques détails sur chaque application.

SYSTÈME DE SÉCURITÉ Nous avons implémenté un système de sécurité responsable d'un bâtiment. Le système de sécurité peut être activé ou désactivé à partir d'un boîtier, le *locker*, et à l'aide d'un mot de passe. Quand le système est activé, celui-ci détecte les intrusions à partir de détecteurs de mouvements déployés dans le bâtiment. Si une intrusion est détectée, des alarmes dans le bâtiment se déclenchent. De plus, les caméras présentes sur le lieux de l'intrusion prennent des photos qui sont transmises au gardien. L'architecture DiaSpec associée à cette application est représentée Figure 26. La Figure 27 montre

2. http://www.flightgear.org/

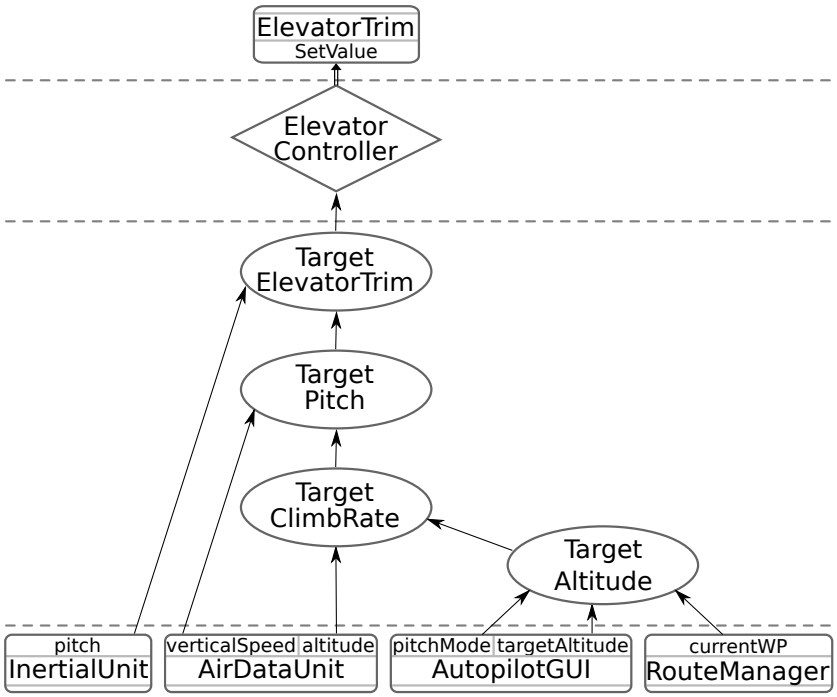

FIGURE 23: Extrait de l'application de pilote automatique décrite avec DiaSpec

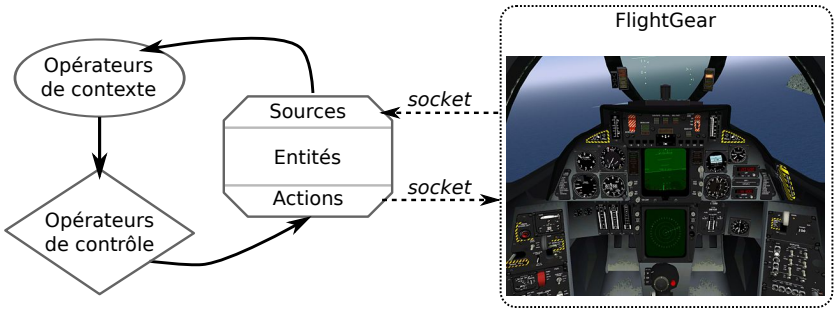

FIGURE 24: Simulation de l'application de pilote automatique d'avion avec le simulateur FlightGear

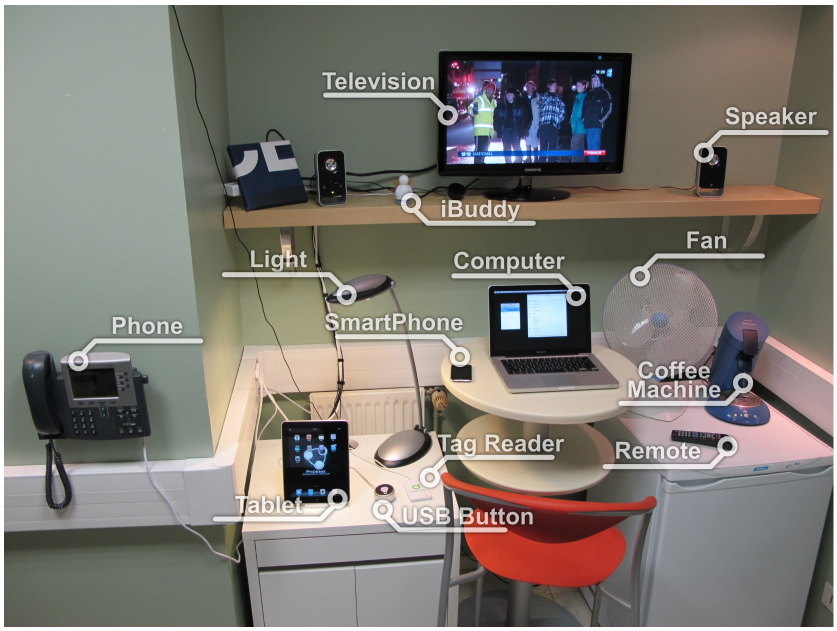

FIGURE 25: Matériels dédiés aux expérimentations domotiques dans notre laboratoire

une simulation de cette application réalisée avec DiaSim, un simulateur dédié aux applications développées avec DiaSpec [14]. Cette application a servi d'exemple à des démonstrations de notre approche [15].

DIFFUSION D'INFORMATIONS CONTEXTUELLES La diffusion d'informations contextuelles, vise à fournir des informations générales aux utilisateurs et à annoncer des événements prochains en fonction de leurs préférences ; un exemple de ce type d'applications est donné par Ranganathan et Campbell [87], dans le cadre de la publicité. Ce domaine nécessite des entités qui

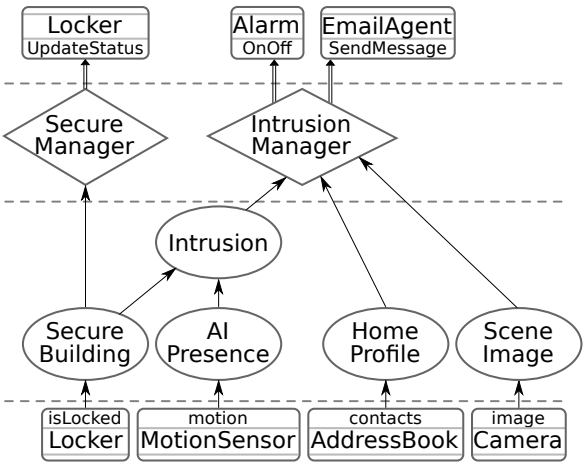

FIGURE 26: Architecture du système de sécurité

FIGURE 27: Simulateur du système de sécurité

diffusent des messages (comme des haut-parleurs et des écrans). En outre, les utilisateurs doivent être identifiés pour déterminer leurs préférences. Cette identification peut se faire par le biais de différentes techniques, telles que des lecteurs de badges à courte portée. Dans notre implémentation, l'application, dite de *news-cast*, est déployée dans une école et possède deux fonctions. Tout d'abord, l'application annonce les cours prochains à l'aide des haut-parleurs. La seconde fonction est l'affichage d'informations personnalisées aux étudiants se situant près d'écrans déployés dans l'école. Les informations affichées sur les écrans sont les dernières nouvelles de l'école ainsi que les emplois du temps du jour. Ces informations sont affichées en fonction des intérêts des étudiants se situant à côté de chaque écran. Ainsi, l'information affichée sur un écran dépend de la langue parlée, de la spécialité, des cours et des activités extra-scolaires des étudiants autour. La Figure 28 présente l'architecture de cette application en DiaSpec tandis que la Figure 29 montre une simulation de cette application avec le simulateur DiaSim dans les locaux de l'école d'ingénieurs ENSEIRB. [3]

DIALOGUE AVEC LE DOMICILE Cette application permet à un utilisateur de contrôler sa maison à distance grâce à un logiciel de messagerie instantanée standard. La Figure 30 présente un dialogue entre un utilisateur et son domicile. La commande `all lights off` permet d'éteindre les lumières, tandis que la

3. http://www.enseirb-matmeca.fr/

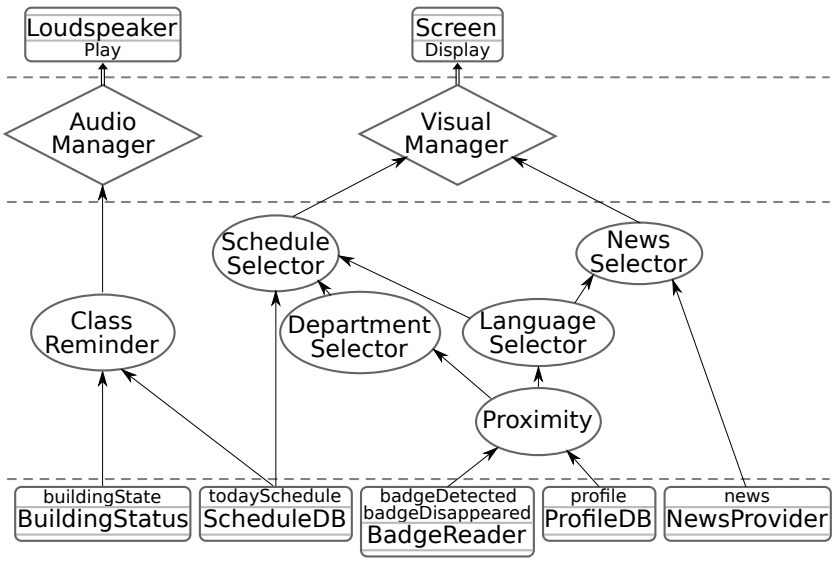

FIGURE 28: Application de *newscast* décrite avec DiaSpec

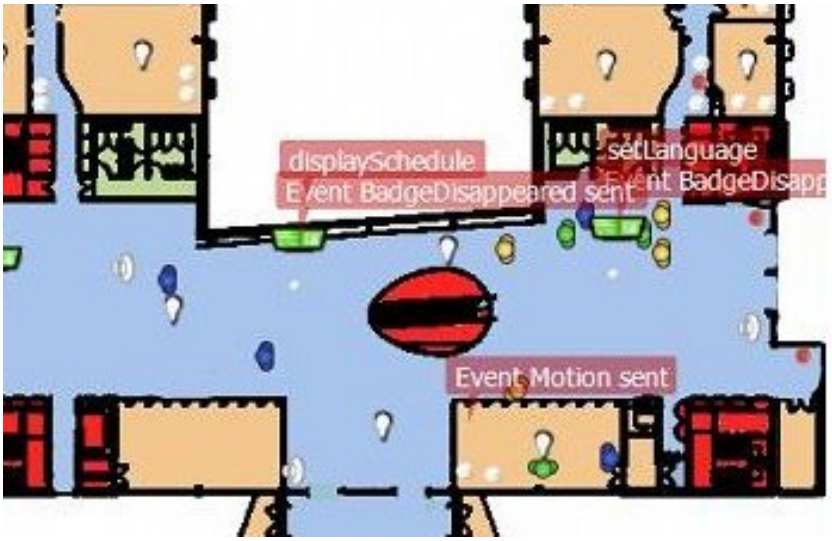

FIGURE 29: Simulation de l'application de *newscast* dans une école d'ingénieurs

FIGURE 30: Exemple de dialogue entre un utilisateur et son domicile

	Entité			*Opérateur*	
	classe	*source*	*action*	*contexte*	*contrôle*
Newscast	7	6	2	6	2
Lumière	5	4	2	3	1
Air	7	5	4	6	2
Feu	7	4	3	2	1
Accès	4	5	1	3	1
Sécurité	6	4	2	4	2

TABLE 3: Métriques de l'étude de gestion de l'école

commande `help` affiche la liste de toutes les commandes compréhensibles. La Figure 31 présente l'architecture de cette application en DiaSpec.

8.1.4 *Étude réelle*

Nous avons appliqué le langage DiaSpec et les outils associés présentés dans cette thèse à un cas d'étude réel : la gestion de l'école d'ingénieurs ENSEIRB de 13 500 m². Six applications, incluant l'application de *newscast*, ont été développées avec DiaSpec pour cette étude. Les autres applications gèrent l'air, la lumière, le feu, la sécurité ainsi que les accès. La Table 3 donne, pour chaque application de cette étude, le nombre d'éléments pour chaque type de composants.

Au total, l'étude compte 36 classes d'entités, 28 sources, 14 actions, 24 opérateurs de contexte et 9 opérateurs de contrôle. La

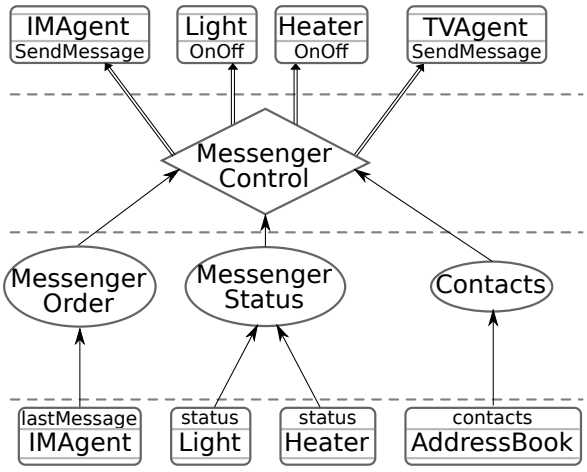

FIGURE 31: Application de dialogue avec le domicile décrite avec DiaSpec

taxonomie tient en 200 lignes de codes (LoC pour *Lines of Code*), l'architecture en 250 LoC, le *framework* généré en 7000 LoC et l'implémentation Java par les développeurs en 3000 LoC. Nous pouvons observer que pour une application réaliste, le nombre de classes d'entités est faible : 7 au maximum. Ceci rend les artefacts de notre approche maîtrisables pour les membres du projet. Ces artefacts forment un véhicule efficace pour exposer et partager les décisions architecturales. Cette étude montre que la plupart de la logique applicative est implémentée dans les opérateurs de contexte. En effet, les couches du paradigme SCC isolent le calcul de données de leurs utilisations pour contrôler l'environnement. Cette architecture en couches simplifie l'implémentation des opérateurs de contrôle et permet au calcul d'évoluer indépendemment du contrôle. En conséquence, il y a peu d'opérateurs de contrôle par application (entre un et deux), en comparaison du nombre d'opérateurs de contexte (quatre en moyenne).

L'école est simulée grâce à DiaSim. Dans cette simulation, plus de 400 instances d'entités sont déployées et 300 occupants sont simulés avec des comportements différents. Nous pouvons noter que, bien que le nombre de déclarations dans les applications soit assez faible, l'étude permet de gérer des scénarios de simulation de taille conséquente.

Pendant le développement de l'approche présentée dans cette thèse, nous avons implémenté de nombreuses applications couvrant beaucoup de domaines. Ce large spectre d'applications et de domaines montre l'expressivité de notre langage DiaSpec ainsi que du paradigme SCC associé.

8.2 UTILISABILITÉ

Nous avons utilisé l'approche DiaSpec lors de plusieurs cours sur les architectures logicielles. Ce cours incluait des sessions pratiques comprenant quatre groupes d'une vingtaine d'étudiants chacun, aucun d'eux n'ayant eu d'expérience préalable avec l'approche. Leur tâche était de développer une application dont les spécifications étaient données au début sous forme de diagrammes. Nous avons volontairement donné que très peu d'informations sur le *framework* généré pour déterminer jusqu'à quel point celui-ci était capable de guider le développement de lui-même.

Les étudiants ont été répartis en sous-groupes de trois. Tous les sous-groupes ont réalisé une architecture DiaSpec fonctionnelle, la plupart du temps avec très peu d'aide de l'enseignant. Mis à part trois sous-groupes, tous les autres ont produit une implémentation fonctionnelle qui validait les tests unitaires fournis par l'enseignant ainsi que d'autres tests unitaires développés par les étudiants en cours de projet. Quelques sous-groupes sont même allés au-delà de leur tâche en configurant et en exécutant une simulation avec DiaSim. Au total, le développement du projet a été réalisé sur trois séances de deux heures chacune. Cette expérience a montré que les étudiants sont capables d'appréhender DiaSpec dans un temps relativement court.

Les quatre groupes d'étudiants étaient répartis sur trois années. Lors des deux premières années, les étudiants utilisaient une version précédente de DiaSpec qui n'incluait pas les contrats d'interactions. Nous avons alors noté quelques limitations dans le support fourni par le *framework* généré : sans documentation, quelques étudiants ne trouvaient pas quelles méthodes ils devaient appeler ou quels paramètres transmettre (le *framework* généré n'étant pas encore complètement documenté). Dans ces cas, l'enseignant devait expliquer quelles méthodes du *framework* devaient être appelées et comment les appeler correctement. C'est la raison pour laquelle nous avons décidé de transmettre tous les outils requis en paramètre des méthodes à implémenter. De cette façon, les étudiants se laissent guider par la fonction de complétion de leur environnement de développement et il n'est pas nécessaire de leur fournir des explications.

Lors de quelques études informelles au sein du laboratoire, nous avons remarqué que DiaSpec facilite la décomposition des efforts de développement en tâches clairement définies : chaque membre du projet a seulement besoin d'une connaissance locale de la tâche qui lui est assignée. En effet, la compréhension globale du projet et de son architecture est encodée à l'intérieur du *framework* généré et chaque implémentation de composants se

fait en utilisant uniquement les outils fournis en paramètre des méthodes. Nous souhaitons conduire une étude pour évaluer l'impact de ce résultat dans la pratique.

Ces diverses expériences nous ont permis de nous assurer que DiaSpec peut être rapidement pris en main et ceci avec très peu d'explications préalables.

8.3 PRODUCTIVITÉ

Un des bénéfices d'utiliser un DSL est d'améliorer la productivité [39, 62]. Dans la suite, nous montrons comment l'approche DiaSpec réduit le temps de développement, améliore la qualité du code et encourage la réutilisation. Nous pensons que ces trois dimensions donnent un aperçu de l'amélioration en productivité apportée par notre approche.

8.3.1 *Temps de développement*

Le temps de développement initial d'un projet est directement proportionnel à la taille du code écrit manuellement. La génération de code automatique réduit le code à écrire manuellement et donc le temps de développement. Nous avons mesuré la quantité de code généré pour plusieurs applications que nous avons développées. Les résultats calculés pour les applications énoncées précédemment sont présentés Table 4. La première colonne donne le nom de l'application. Les trois suivantes indique respectivement le pourcentage de code écrit en DiaSpec, le pourcentage de code écrit manuellement en Java, et le pourcentage de code généré, dans les trois cas par rapport à tout le code (DiaSpec + Java). La dernière colonne donne le numéro de la page où l'application est présentée. Nos mesures montrent qu'en général, près de 80% du code d'un projet est généré, que l'implémentation représente environ 11% et que le reste correspond aux déclarations DiaSpec.

Il est important de noter que les mesures précédentes n'ont de sens que si le code généré est effectivement utilisé. Sinon, le code généré peut être arbitrairement grand sans que le temps de développement ne soit impacté. Nous avons mesuré la couverture du *framework* lors de plusieurs exécutions d'applications développées avec DiaSpec en utilisant l'outil CodeCover.[4] En moyenne, 70% du *framework* généré est effectivement exécuté. Nous avons étudié les parties du *framework* non exécutées et nous avons découvert que ces parties sont soit de la gestion d'erreurs soit des fonctions inutilisées par la logique applicative, notamment les opérateurs de la découverte d'entités (Section 5.2.3, page 66).

4. http://www.codecover.org/

	archi.	*implém.*	*framew.*	*réf.*
Moniteur de serv. web	12%	13%	75%	p. 29
Cœur	8%	12%	80%	p. 97
Système de sécurité	7%	8%	85%	p. 100
Dialogue	7%	6%	88%	p. 103
Gestion de l'air	7%	12%	81%	p. 105

TABLE 4: Mesures du code généré pour quelques applications

8.3.2 *Qualité du code*

Un programme avec une bonne qualité de code est un programme qui évolue et est maintenu plus facilement. La qualité du code est essentielle car la maintenance d'une application représente plus de 85% du coût total du développement d'une application [33]. Nous avons mesuré la qualité du code écrit manuellement par les développeurs utilisant DiaSpec afin d'évaluer la capacité de notre approche à encourager l'écriture de code de qualité. Nous avons utilisé la plateforme Sonar[5] pour mesurer la qualité du code de trois applications présentées précédemment : le moniteur de serveur web, le système de sécurité et l'application de *newscast*. Cette qualité a été mesurée avec plusieurs critères tels que la duplication de code, la conformité à des règles de codage, la couverture du code et la complexité du code. Par exemple, la complexité du code est mesurée en utilisant la définition de McCabe [76] : cette métrique mesure le nombre de chemins linéairement indépendants dans le code. Le code implémenté pour les trois projets a une complexité moyenne de 3. McCabe note que « du code dont la complexité se situe entre 3 et 7 est très bien structuré ». Une complexité supérieure à 10 indique que le code est de très faible qualité, ce qui entrave la maintenance et donc la productivité. Les autres critères proposés par Sonar présentent des résultats indiquant que le code écrit manuellement par les développeurs est d'une très grande qualité générale. Ces résultats, associés avec le faible pourcentage de code à écrire manuellement, révèlent que le *framework* de programmation généré guide les développeurs vers un code bien structuré et facilement maintenable.

8.3.3 *Réutilisabilité*

Notre approche encourage la réutilisation des spécifications ainsi que des implémentations dans plusieurs applications. Une

5. http://www.sonarsource.org/

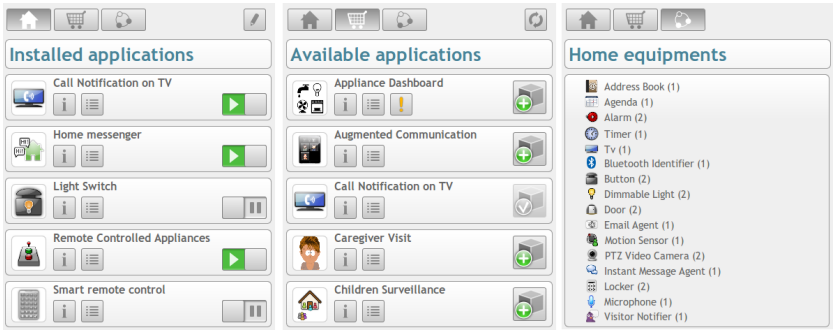

FIGURE 32: Les trois volets de l'application communautaire DiaStore : à gauche les applications actuellement installées, au centre les applications de la communauté qui peuvent être installées et à droite les équipements déployés dans la maison

spécification d'entité et son implémentation peuvent être empaquetées pour être réutilisées plus tard. Pour favoriser encore plus la réutilisation, nous avons développé DiaStore. DiaStore est une application web qui permet de télécharger et déployer facilement des applications DiaSpec, dans l'esprit de l'App Store d'Apple. [6]

DiaStore se compose de trois volets permettant de gérer les applications installées et d'en installer de nouvelles (Figure 32). Le premier volet, à gauche dans la figure, liste les applications installées, permet de les démarrer ou de les arrêter et fournit des informations sur ces applications telles que les entités utilisées. Le deuxième volet, au centre de la figure, permet de télécharger de nouvelles applications : lorsqu'une application utilise une entité qui n'est pas déployée, un point d'exclamation apparaît informant l'utilisateur que certaines fonctionnalités ne seront pas accessibles. Le dernier volet liste les entités actuellement déployées et utilisables par les applications. L'application DiaStore encourage le partage d'applications complètes et d'entités.

Lors des développements d'applications avec DiaSpec, nous avons remarqué que le paradigme SCC encourage la réutilisation d'opérateurs de contextes. En effet, les opérateurs de contexte n'agissent jamais sur l'environnement, ils représentent simplement des données.

Les premières évaluations de notre langage et de ses outils est prometteuse. Comme nous l'avons vu, le spectre d'applications couvertes par le paradigme et le langage DiaSpec est large. L'utilisabilité du *framework* de programmation généré a été validée avec des étudiants qui sont arrivés à développer des applications conséquentes avec très peu d'aide ou de documentation, se reposant simplement sur les capacités de complétion de leur environnement de développement. Enfin, le gain en productivité

6. http://www.apple.com/iphone/features/app-store.html

a été évalué en utilisant plusieurs critères mesurant le temps de développement, la qualité du code et la réutilisabilité.

Ces évaluations sont préliminaires ; nous avons commencé à planifier une évaluation formelle basée sur une méthodologie expérimentale bien définie [115]. En particulier, nous voulons comparer les gains en productivité de notre approche par rapport à d'autres travaux tels que Archface [108] et ArchJava [2].

9 UNE PLATEFORME DE RECHERCHE

Dans ce chapitre, nous montrons que l'approche présentée dans cette thèse peut être un véhicule indispensable à d'autres travaux. Nous listons quelques travaux existants se basant sur notre approche.

9.1 SIMULATEUR POUR L'INFORMATIQUE UBIQUITAIRE

Comme pour tous les domaines, tester une application de l'informatique ubiquitaire est crucial. Cependant, le domaine de l'informatique ubiquitaire a des besoins spécifiques qui empêchent les outils génériques d'y être appliqués [91]. Les outils génériques ne permettent notamment pas de simuler un environnement physique ni des personnes au sein de cet environnement. Pour faciliter le test d'applications de l'informatique ubiquitaire, Bruneau et al. [14] proposent DiaSim, un simulateur dédié à l'informatique ubiquitaire qui s'appuie sur les descriptions écrites en DiaSpec ainsi que sur le *framework* de programmation généré. Grâce au support fourni par le *framework* de programmation, DiaSim est capable de simuler des applications sans nécessiter aucun changement dans le code de l'application. De plus, la couche d'abstraction associée au *framework* de programmation permet à des entités réelles et simulées d'être déployées simultanément pour faciliter le test et le déploiement incrémental des applications.

9.2 CIBLAGE DES UTILISATEURS FINAUX

Des utilisateurs ont montrés de l'intérêt à développer ou personnaliser leurs propres applications, notamment pour les domaines de la domotique et de l'aide à la personne [17, 43]. Cependant, l'approche DiaSpec nécessite des experts de domaine ainsi que des architectes et développeurs expérimentés. Drey et al. [28] proposent Pantagruel [1], un langage visuel associé à une méthodologie permettant aux utilisateurs finaux de développer et de personnaliser des applications SCC, notamment dans le cadre de l'aide à la personne. Pantagruel génère des spécifications DiaSpec

1. http://phoenix.inria.fr/index.php/projects/pantagruel

ainsi que des implémentations s'appuyant sur le *framework* de programmation généré.

9.3 COUVERTURE DES BESOINS NON FONCTIONNELS

Comme nous l'avons vu dans le chapitre présentant le langage DiaSpec, plusieurs approches se basent sur DiaSpec pour faciliter le développement des besoins non fonctionnels dans les applications (page 48). Jakob et al. [59] proposent un cadre général basé sur la programmation par aspects pour étendre DiaSpec avec des déclarations de besoins non fonctionnels. Les auteurs illustrent leur approche avec deux exemples de politique de sécurité pour l'informatique ubiquitaire. Par ailleurs, Mercadal et al. [82] décrivent la gestion des erreurs au niveau de DiaSpec pour pouvoir faciliter ensuite la phase d'implémentation. De leurs côtés, Gatti et al. [49] enrichissent DiaSpec avec des déclarations d'exigences de Qualité de Service et ainsi assurer la traçabilité et la propagation de ces exigences à tout niveau du cycle de développement. En ce qui concerne la tolérance aux fautes, Enard [32] propose d'ajouter des déclarations dédiés dans le langage DiaSpec ainsi que des mécanismes spécifiques, tels que la réplication, dans le *framework* généré.

9.4 ADAPTATION MULTIMÉDIA

Le *streaming*, permet d'accéder à des flux audio et vidéo sans nécessiter un téléchargement complet au préalable. De nombreux sites Internet proposent du contenu multimédia accessible en *streaming*, comme YouTube, Dailymotion ou Deezer. Ces sites proposent parfois plusieurs niveaux de qualités pour un même contenu : un contenu de grande qualité est plus agréable pour l'utilisateur mais nécessite une connexion réseau plus rapide, un processeur plus performant et aussi plus de mémoire. Actuellement, l'utilisateur doit lui-même choisir le niveau de qualité le plus adapté à la capacité de son matériel. L'*adaptation multimédia* permet de décharger l'utilisateur de ce choix : en analysant les capacités de la plateforme de l'utilisateur, un niveau de qualité adéquat est demandé au site de *streaming*. Cette adaptation multimédia peut aussi être réalisée pendant la diffusion du flux si les conditions changent : par exemple, si les paquets réseaux arrivent moins vite à destination, le niveau de qualité pourra être baissé pour diminuer le nombre de paquets requis par seconde.

Koumaras et al. [65] ont développé, en langage C, un outil permettant de réaliser automatiquement cette adaptation multimédia. Après plusieurs années de développement, un des auteurs,

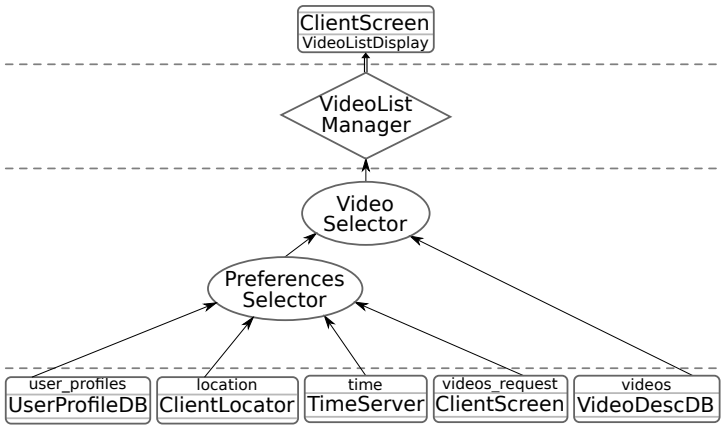

FIGURE 33: Première partie du scénario d'adaptation multimédia

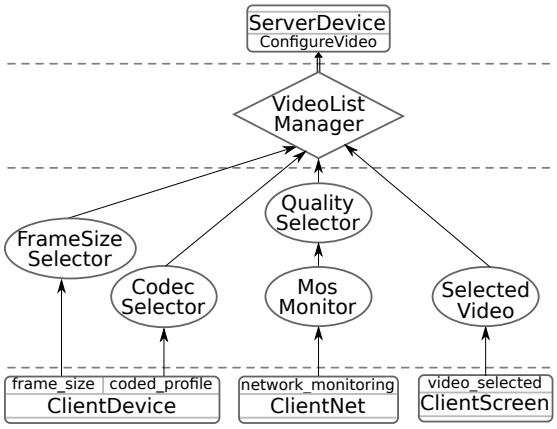

FIGURE 34: Deuxième partie du scénario d'adaptation multimédia

Daniel Négru, a souhaité rendre cet outil plus maintenable en explicitant l'architecture. Il a pour cela utilisé le paradigme SCC et le langage DiaSpec. Le résultat de son travail est représenté dans les Figures 33 et 34. Quand l'implémentation sera terminée, il pourrait être intéressant de comparer l'implémentation d'origine (en langage C) et la nouvelle. Ce projet est aussi intéressant pour comprendre si le paradigme SCC est applicable au domaine de l'adaptation multimédia.

9.5 DÉPLOIEMENT CHEZ LES PARTICULIERS

Dans le cadre du développement de sa nouvelle *box* internet, la Bbox, Bouygues Telecom souhaite apporter la domotique au domicile de tous ses clients. Bouygues Telecom part du constat que beaucoup d'équipements sont déjà reliés à la *box* des particuliers : Internet, téléphones fixes, téléphones mobiles, télévisions, ordinateurs *etc.* Ces équipements fonctionnent actuellement de façon isolée ; le but de Bouygues Telecom est de permettre aux clients d'installer des applications sur leur *box* pour coordonner

ces équipements en fonction de leurs besoins. Voici quelques cas d'utilisation possibles :

- mettre en pause la télévision (fonction de *time shifting*) lorsque le téléphone sonne et y afficher le numéro et le nom de l'appelant tout en récupérant sa photo sur les réseaux sociaux ou sur l'ordinateur ;
- afficher les notifications de nouveau message (téléphonie ou *email*) sur l'écran de télévision ;
- appeler des correspondants avec un téléphone mobile ou un ordinateur *via* la ligne de téléphone fixe ;
- contrôler à distance les équipements de la maison, comme la porte ou les lumières, *via* un logiciel de messagerie instantanée.

Des prototypes de tous ces cas d'utilisation ont été développés avec DiaSpec. Un contrat entre *Bouygues Telecom* et l'équipe projet *Phoenix* vient d'être signé pour étudier la faisabilité d'installer DiaSpec et sa suite d'outils dans le prototype de la future Bbox. L'application web DiaStore présentée précédemment (Section 8.3.3, page 109) prendrait alors tout son sens.

L'approche présentée dans cette thèse doit être à la fois considérée comme une fin en soi et aussi comme un levier ouvrant de nombreux axes de recherche et de développement. En effet, nous avons montré dans ce chapitre que DiaSpec est une plateforme pouvant être utilisée de nombreuses façons différentes : (1) un support pour la recherche sur la couverture des besoins non fonctionnels et la simulation graphique d'applications, (2) une suite d'outils permettant aux développeurs amateurs et professionnels de coordonner les technologies présentes chez eux et aussi de développer de nouveaux outils pour les utilisateurs finaux, et (3) un outil d'aide au développement d'applications.

10 CONCLUSION

Dans cette thèse, nous avons présenté le paradigme SCC et décrit son importance : ce paradigme couvre de nombreux domaines d'applications et son étude peut donc avoir des conséquences notables. Nous avons remarqué que les approches existantes de développement logiciel permettent, mais ne facilitent pas, de développer des applications suivant ce paradigme. La plupart des approches existantes ne permettent notamment pas de décrire une application SCC suivant le paradigme et indépendamment de l'implémentation. Quand une approche fournit des mécanismes haut-niveau pour décrire une application SCC, elle n'utilise pas cette description pour guider et vérifier l'implémentation.

C'est pourquoi nous avons développé une nouvelle approche, basée sur un langage de description d'architectures dédié au paradigme SCC. À partir d'une description d'architecture dans ce langage, un compilateur génère un *framework* de programmation dédié dans un langage généraliste. En s'appuyant sur ce *framework*, des développeurs peuvent implémenter la logique de l'application SCC. Le *framework* de programmation est conçu pour que les contraintes spécifiées dans l'architecture soient nécessairement assurées au niveau de l'implémentation. Ce *framework* de programmation est conçu pour être auto-documenté, ne nécessitant presque aucune période d'apprentissage préalable. Loin d'empêcher un développement itératif, notre approche générative en facilite l'usage en maintenant à tout moment une cohérence forte entre la description architecturale et l'implémentation. Enfin, notre approche permet aux outils de conduire des analyses au niveau de la description de l'architecture, de l'implémentation ainsi que pendant l'exécution.

Les contributions de cette thèse ont été évaluées suivant trois aspects complémentaires : (1) l'*expressivité*, évaluant la portée du paradigme SCC et du langage DiaSpec, (2) l'*utilisabilité*, estimant la facilité d'utilisation des outils, et (3) la *productivité*, mesurant le temps de développement, la qualité du code ainsi que la capacité à réutiliser le code. Enfin, l'approche a été présentée en tant que plateforme de recherche autour de laquelle de nombreux travaux gravitent.

TRAVAUX EN COURS ET FUTURS

Au cours de cette thèse, nous avons régulièrement présenté des perspectives de travail ainsi que des travaux existants autour de ceux de cette thèse. Nous en présentons ici quelques uns en plus et revenons sur les plus importants.

GÉNÉRALISER À D'AUTRES DOMAINES L'approche présentée dans cette thèse, et notamment le langage DiaSpec, est dédiée au paradigme SCC : les concepts du paradigme sont directement encodés à l'intérieur du langage. Nous pensons qu'une approche similaire peut être appliquée à d'autres paradigmes ou domaines. Dans ce sens, nous avons commencé à appliquer une approche similaire au domaine des machines virtuelles (VMs) de langages en collaboration avec Michael Haupt de l'institut de recherche HPI (*Hasso–Platner–Institute*, Potsdam, Allemagne). Haupt et al. [53] proposent une approche architecturale à l'implémentation de VMs qui considère qu'une VM est une composition de services à coordonner ; dans ce cadre, un service peut être le ramasse-miettes (*garbage collector*) ou le gestionnaire de mémoire. Dans ce but, les auteurs utilisent la programmation par aspects pour structurer l'implémentation et faire interagir les services sans que le code de ces services soit mêlé. Nous pensons qu'il est possible de s'abstraire un peu plus de l'implémentation en fournissant un langage dédié et architectural, équivalent de DiaSpec pour les VMs, et en générant un *framework* de programmation à partir d'une description dans ce langage.

TESTER LES APPLICATIONS Quand une application ne fonctionne pas comme attendue au moment de l'exécution, le testeur doit identifier le composant problématique. Une fois le composant identifié, il faut déterminer quelles entrées envoyées à ce composant fournissent une réponse erronée puis corriger l'implémentation du composant. Actuellement ce processus est long et difficile car le *framework* de programmation généré impose de déployer et exécuter toute l'application pour tester un seul composant. Nous avons développé un prototype, sous la forme d'un *framework* de test, permettant de faciliter le test unitaire de chaque composant. Grâce à ce *framework* de test, un testeur est capable d'indiquer comment un opérateur de contexte doit réagir en fonction de différentes conditions d'activation. Ces scénarios d'utilisation sont décrits indépendamment les uns des autres et aussi indépendamment de l'implémentation. De cette façon, il est possible de décrire des scénarios d'utilisation avant de commencer à implémenter et ainsi de permettre aux développeurs de vérifier leurs implémentations. Ce prototype nécessite cependant

d'utiliser Java et de connaître les technologies JUnit [1] et jMock [2], qui sont deux bibliothèques Java permettant d'implémenter des tests unitaires. Après quelques essais d'utilisation de ce prototype, il semble possible de s'abstraire de ces technologies pour demander moins de connaissances de la part des testeurs et pour leur fournir des mécanismes plus haut-niveau pour décrire des scénarios. Une abstraction possible pourrait prendre la forme d'un langage dédié à la description de scénarios de tests pour les applications SCC.

SUPPORTER LES ASPECTS NON FONCTIONNELS Comme nous l'avons présenté précédemment, plusieurs travaux ont commencé pour faciliter la gestion des aspects non fonctionnels dans le cycle de développement. Jakob et al. [59] étendent DiaSpec avec des annotations génériques pour la description des besoins non fonctionnels et utilisent ces annotations pour générer du support de programmation. Mercadal et al. [82] étendent DiaSpec avec des déclarations de gestion d'erreurs permettant de séparer au niveau de l'implémentation le code de la logique applicative et le code de traitement des erreurs. Enfin, Gatti et al. [49] étendent DiaSpec avec des propriétés de QoS et fournissent du support, tant à l'implémentation qu'à l'exécution, pour assurer que ces propriétés soient vérifiées. D'autres travaux sont en cours, pour par exemple gérer la sûreté de fonctionnement et la sécurité des applications SCC, de la phase de spécification jusqu'au déploiement, en passant par l'implémentation. L'abondance de travaux dans cette direction montre que l'approche décrite dans cette thèse est un bon véhicule de recherche qui permet d'exprimer au plus tôt les besoins non fonctionnels et qui propage ces besoins le long des autres phases du cycle de développement.

ÉVALUER L'APPROCHE EMPIRIQUEMENT Pour pouvoir comprendre l'impact d'un nouvel outil, d'un nouveau langage ou d'une nouvelle méthodologie dans le développement logiciel, il est nécessaire de faire des évaluations empiriques sur les développeurs [52, 115]. Dans ce sens, nous avons mis en place une évaluation empirique que nous avons faite passer à des étudiants de dernière année d'école d'ingénieurs. Nous leur avons aussi demander de remplir un questionnaire (Annexe A). Nous souhaitons poursuivre ce travail avec des développeurs professionnels et étudier les résultats et les commentaires des étudiants.

1. http://www.junit.org/
2. http://www.jmock.org/

Quatrième partie

APPENDICES

A

QUESTIONNAIRE DE L'ÉVALUATION

Les questions suivantes ont été posées à chaque étudiant d'un groupe de dix-huit étudiants en dernière année d'école d'ingénieurs. À chaque question est associée une échelle de cinq valeurs permettant à chaque étudiant de nuancer son choix entre « Vraiment pas d'accord » et « Vraiment d'accord ».

1. L'outil m'a aidé à décrire mon architecture
2. J'ai trouvé cet outil facile à utiliser
3. J'ai trouvé les concepts de l'outil inutilement complexes
4. J'ai trouvé qu'il y a un trop grand nombre de concepts à assimiler pour pouvoir profiter pleinement de l'outil
5. Je pense que j'aurais besoin d'assistance pour être capable de créer d'autres applications avec cet outil
6. Je me suis senti à l'aise en utilisant cet outil
7. Faire une architecture avec cet outil rend l'implémentation plus simple
8. L'outil m'a guidé dans la réalisation de mon architecture
9. L'outil m'a guidé dans la réalisation de mon implémentation
10. J'ai manqué d'explications sur l'utilisation de l'outil
11. J'utiliserais volontiers cet outil si l'occasion s'en présentait
12. Je pense que l'outil est efficace pour structurer une application
13. Je pense que j'aurais des difficultés à ajouter des fonctionnalités à une application développée avec cet outil
14. Je pense que j'aurais des difficultés à corriger une application développée avec cet outil
15. Réutiliser du code existant pour développer une nouvelle application est facilité par l'utilisation de cet outil
16. Je pense que cet outil peut s'appliquer à un grand nombre d'applications différentes
17. Je pense qu'il manque des concepts à cet outil pour pouvoir développer de nombreuses applications différentes
18. Il est facile d'exprimer ce que l'on souhaite concevoir en utilisant cet outil

19. Je pense qu'il est plus difficile d'implémenter une application si l'on doit se conformer à une architecture décrite en utilisant cet outil
20. Je pense que cet outil peut être utilisé pour concevoir de « grosses » applications
21. J'ai trouvé l'exercice clair
22. J'ai bien compris ce qu'il fallait faire
23. Je me sens à l'aise avec Java
24. Je me sens à l'aise avec Eclipse
25. J'ai bien lu tous les documents fournis

Après ces questions, une zone de réponse libre était proposée pour laisser à chaque étudiant la possibilité d'expliquer les réponses aux questions précédentes ou de donner son point de vue. Cette zone était précédée de la mention suivante : « Utilisez l'espace ci-dessous (et le verso de la feuille) pour donner votre sentiment général sur l'outil, sur l'évaluation et sur le questionnaire. Les questions suivantes peuvent vous aider à répondre. Quelles sont les bénéfices/lacunes de l'outil ? Que pensez vous de l'architecture/implémentation que vous avez faites (en terme de maintenabilité, évolutivité, . . .) ? Pour quelles raisons auriez-vous envie (ou n'auriez-vous pas envie) d'utiliser ce logiciel ? Vous a t-il manqué quelque chose pour faire mieux ? Qu'avez-vous trouvé peu clair/pas clair du tout dans cet outil/cette évaluation/ce questionnaire ? »

B

GRAMMAIRE DU LANGAGE DIASPEC

Les pages suivantes présentent la grammaire du langage DiaSpec. La syntaxe choisie est proche de EBNF (*Extended Backus-Naur Form*) et est utilisée par Xtext [1] pour développer des extensions à Eclipse.

```
Document:
        (typeDefs+=TypeDef)*
;
TypeDef:
        IncludeSpec
        | ImportHostType
        | DeviceDef
        | ContextDef
        | ControllerDef
        | ActionDef
        | StructDef
        | EnumDef
;
IncludeSpec:
        'include' importURI=STRING ';'
;
ImportHostType:
        'import' name=ID 'as' package=FULL_HOST_CLASS ';'
;
DeviceDef:
        'abstract'? 'device' name=ID ('extends'
            superDevice=[DeviceDef])?
        '{' (  attributes+=AttributeDef
             | sources+=SourceDef
             | actions+=ActionImpl
            )*
        '}'
;
ContextDef:
        'context' name=ID 'as' dataTypeRef=DataTypeRef
                (contextIndices=Indices)?
        '{' (  sourceRefs+=SourceRef
             | contextRefs+=ContextRef
             | behaviorDefs+=ContextBehaviorDef
            )*
        '}'
;
ControllerDef:
        'controller' name=ID
        '{' (  contextRefs+=ContextRef
             | actionRefs+=ActionRef
             | behaviorDefs+=ControllerBehaviorDef
            )*
```

1. http://www.eclipse.org/Xtext/

```
43          '}'
44  ;
45  ContextBehaviorDef:
46          'behavior' name=ID?
47          '{'
48                  activationConditions=CtxtBehaviorActivationConditions
49                  dataRequirements=CtxtBehaviorDataRequirements?
50                  behaviorPublication=BehaviorPublication
51          '}'
52  ;
53  CtxtBehaviorActivationConditions:
54          'when' (  (required='required' 'self'?)
55                  | provided=CtxtBehaviorProvided
56                 )
57  ;
58
59  CtxtBehaviorProvided:
60          'provided' sourceList=CtxtBehaviorSourceList
61  ;
62
63  CtxtBehaviorDataRequirements:
64          'get' sourceList=CtxtBehaviorSourceList
65  ;
66
67  CtxtBehaviorSourceList:
68          bsRefs+=CtxtBehaviorSourceRef (','
                bsRefs+=CtxtBehaviorSourceRef)*
69  ;
70
71  CtxtBehaviorSourceRef:
72          BehaviorDeviceSourceRef | BehaviorContextSourceRef
73  ;
74
75  ControllerBehaviorDef:
76          'behavior' name=ID?
77          '{'
78              activationConditions=CtrlBehaviorActivationConditions
79              dataRequirements=CtrlBehaviorDataRequirements?
80              (behaviorActionGroups+=BehaviorActionGroup)+
81          '}'
82  ;
83
84  CtrlBehaviorActivationConditions:
85          'when' 'provided' sourceList=CtrlBehaviorSourceList
86  ;
87
88  CtrlBehaviorDataRequirements:
89          'get' sourceList=CtrlBehaviorSourceList
90  ;
91
92  CtrlBehaviorSourceList:
93          bsRefs+=BehaviorContextSourceRef (','
                bsRefs+=BehaviorContextSourceRef)*
94  ;
95
96  BehaviorActionGroup:
97          'do' BehaviorActionList
98  ;
```

```
99
100 BehaviorPublication:
101         'always publish' | 'no publish' | 'maybe publish'
102 ;
103
104 BehaviorActionList:
105         baRefs+=BehaviorActionRef (',' baRefs+=BehaviorActionRef)*
106 ;
107
108 BehaviorActionRef:
109         actionId=[ActionDef] 'on' deviceId=[DeviceDef]
110 ;
111
112 BehaviorDeviceSourceRef:
113         sourceId=[SourceDef | VarName] 'from' deviceId=[DeviceDef]
114 ;
115
116 BehaviorContextSourceRef:
117         contextId=[ContextDef]
118 ;
119
120 IndexDef:
121         var=VariableDef
122 ;
123
124 SourceDef:
125         'source' sourceName=VarName 'as' dataTypeRef=DataTypeRef
              (sourceIndices=Indices)? ';'
126 ;
127
128 Indices:
129         'indexed by' indices+=IndexDef (',' indices+=IndexDef)*
130 ;
131
132 ActionImpl:
133         'action' name=[ActionDef] ';'
134 ;
135
136 AttributeDef:
137         'attribute' attributeName=VarName 'as'
              dataTypeRef=DataTypeRef ';'
138 ;
139
140 StructDef:
141         'structure' name=ID '{' (structFields+=StructFieldDef
              ';')* '}'
142 ;
143
144 EnumDef:
145         'enumeration' name=ID '{' enumFields+=EnumFieldDef (','
              enumFields+=EnumFieldDef)* '}'
146 ;
147
148 StructFieldDef:
149         var=VariableDef
150 ;
151
152 EnumFieldDef:
```

```
153         enumFieldName=ID
154 ;
155
156 ActionDef:
157         'action' name=ID '{' (orders+=OrderDef)+ '}'
158 ;
159
160 OrderDef:
161          orderName=VarName '(' (params+=ParameterDef (',' params+=ParameterDef)*)? ')' ';'
162 ;
163
164 Keyword:
165         'enumeration' | 'structure' | 'action'
166         | 'device' | 'context' | 'controller'
167         | 'source' | 'on' | 'from' | 'as' | 'extends'
168         | 'attribute' | 'indexed' | 'by'
169         | 'behavior' | 'when' | 'provided' | 'required' | 'this' | 'get'
170         | 'maybe' | 'no' | 'always' | 'publish'
171         | 'do' | 'any' | 'all'
172 ;
173
174 ParameterDef:
175         var=VariableDef
176 ;
177
178 VariableDef:
179          varName=VarName 'as' dataTypeRef=DataTypeRef
180 ;
181
182 SourceRef:
183         'source' sourceId=[SourceDef | VarName] 'from' deviceId=[DeviceDef] ';'
184 ;
185
186 VarName:
187         VAR_ID | Keyword
188 ;
189
190 ContextRef:
191         'context' contextId=[ContextDef] ';'
192 ;
193
194 ActionRef:
195         'action' actionId=[ActionImpl | ActionName] 'on' deviceId=[DeviceDef] ';'
196 ;
197
198 ActionName:
199         ID | Keyword
200 ;
201
202 DataTypeRef:
203         (type=[SourceType] | primitiveTypeRef=PrimitiveTypeRef) (isList=ListTag)?
204 ;
205
```

```
206 SourceType:
207         ImportHostType | StructDef | EnumDef
208 ;
209
210 enum PrimitiveTypeRef:
211         INTEGER='Integer' | BOOLEAN='Boolean' | STRING='String' |
                FLOAT='Float'  | BINARY='Binary'
212 ;
213
214 ListTag:
215         tag='[]'
216 ;
217
218 terminal VAR_ID: 'a'..'z' ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;
219 terminal ID: 'A'..'Z' ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')*;
220 terminal FULL_HOST_CLASS: (('a'..'z') + '.')+ ID;
```

Listing 12: Grammaire du langage DiaSpec

BIBLIOGRAPHIE

[1] ACCORD. État de l'art sur les langages de description d'architecture (ADLs). Technical report, ACCORD, 2002.

[2] Jonathan Aldrich, Craig Chambers, et David Notkin. ArchJava : Connecting software architecture to implementation. Dans *ICSE'02 : Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, Orlando, FL, USA, 2002. doi : 10.1145/581339.581365.

[3] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, et David Notkin. Language support for connector abstractions. Dans *ECOOP'03 : Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 74–102, Darmstadt, Germany, 2003. doi : 10.1007/978-3-540-45070-2_5.

[4] Robert J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1997.

[5] AUTOSAR. AUTomotive Open System ARchitecture, 2010. url : http://www.autosar.org/.

[6] Kent Beck. Manifesto for agile software development, 2001. url : http://agilemanifesto.org.

[7] Ron Ben-Natan. *Corba*. McGraw-Hill, 1995. isbn : 0-07-005427-4.

[8] Gérard Berry. The foundations of Esterel. Dans *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, 2000.

[9] Benjamin Bertran, Charles Consel, Patrice Kadionik, et Bastien Lamer. A SIP-based home automation platform : An experimental study. Dans *ICIN'09 : Proceedings of the 13th International Conference on Intelligence in Next Generation Networks*, pages 1–6, Bordeaux, France, 2009. doi : 10.1109/ICIN.2009.5357075.

[10] Benjamin Bertran, Charles Consel, Wilfried Jouve, Hongyu Guan, et Patrice Kadionik. SIP as a universal communication bus : A methodology and an experimental study. Dans *ICC'10 : Proceedings of the 9th International Conference on Communications*, pages 1–5, Cape Town, South Africa, 2010. doi : 10.1109/ICC.2010.5502591.

[11] Jean Bézivin et Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. Dans *ASE'01 : Proceedings of 16th International Conference on Automated Software Engineering*, pages 273–282, San Diego, CA, USA, 2001. doi : 10.1109/ASE.2001.989813.

[12] Grady Booch, James Rumbaugh, et Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, second edition, 2005.

[13] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical report, Artificial Intelligence Laboratory, Cambridge, MA, USA, 1985.

[14] Julien Bruneau, Wilfried Jouve, et Charles Consel. DiaSim, a parameterized simulator for pervasive computing applications. Dans *Mobiquitous'09 : Proceedings of the 6th International Conference on Mobile and Ubiquitous Systems : Computing, Networking and Services*, pages 1–10, Toronto, Ontario, Canada, 2009. doi : 10.4108/ICST.MOBIQUITOUS2009.6851.

[15] Damien Cassou, Julien Bruneau, et Charles Consel. A tool suite to prototype pervasive computing applications. Dans *PerCom'10 : Proceedings of the 8th International Conference on Pervasive Computing and Communications*, pages 820–822, Mannheim, Germany, 2010. doi : 10.1109/PERCOMW.2010.5470550. Demonstration.

[16] Guanling Chen et David Kotz. Context aggregation and dissemination in ubiquitous computing systems. Dans *WMCSA'02 : Proceedings of the 4th Workshop on Mobile Computing Systems and Applications*, pages 105–114, Washington, DC, USA, 2002. doi : 10.1109/MCSA.2002.1017490.

[17] Jeannette S Chin, Vic Callaghan, et Graham Clarke. An end-user programming paradigm for pervasive computing applications. Dans *ICPS'06 : Proceedings of the 3rd International Conference on Pervasive Services*, pages 325–328, Washington, DC, USA, 2006. doi : 10.1109/PERSER.2006.1652254.

[18] Paul Clements et Mary Shaw. "the golden age of software architecture" revisited. *IEEE Software*, 26(4) :70–72, 2009. doi : 10.1109/MS.2009.83.

[19] R.P.G. Collinson. *Introduction to Avionics Systems*. Springer, second edition, 2003. isbn : 978-1402072789.

[20] Pascal Costanza, Robert Hirschfeld, et Wolfgang De Meuter. Efficient layer activation for switching context-dependent behavior. Dans *JMLC'06 : Proceedings of the Joint Modular Languages Conference*, pages 84–103, Oxford, UK, 2006. doi : 10.1007/11860990_7.

[21] Eric M. Dashofy, Nenad Medvidovic, et Richard N. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. Dans *ICSE'99 : Proceedings of the 21st International Conference on Software engineering*, pages 3–12, Los Angeles, CA, USA, 1999. doi : 10.1145/302405.302407.

[22] Luca de Alfaro et Thomas A. Henzinger. Interface automata. Dans *ESEC'01 : Proceedings of the 8th European Software Engineering Conference*, pages 109–120, Vienna, Austria, 2001. doi : 10.1145/503209.503226.

[23] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, et Wolfgang De Meuter. Ambient-oriented programming in ambienttalk. Dans *ECOOP'06 : Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 230–254, Nantes, France, 2006. doi : 10.1007/11785477_16.

[24] Anind K. Dey, Gregory D. Abowd, et Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16(2) :97–166, 2001. doi : 10.1207/S15327051HCI16234_02.

[25] Theo D'Hondt, Kris De Volder, Kim Mens, et Roel Wuyts. Co-evolution of object-oriented software design and implementation. Dans *SACT'00 : Proceedings of the 1st International Symposium on Software Architectures and Component Technology*, pages 207–224, Enschede, The Netherlands, 2000.

[26] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, et Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2) :223–259, 2006. doi : 10.1145/1186778.1186782.

[27] Troy Bryan Downing. *Java RMI : Remote Method Invocation*. IDG Books Worldwide, Inc., 1998. isbn : 978-0764580437.

[28] Zoé Drey, Julien Mercadal, et Charles Consel. A taxonomy-driven approach to visually prototyping pervasive computing applications. Dans *DSL WC'09 : Proceedings of the 1st Working Conference on Domain-Specific Languages*, pages 78–99, Oxford, UK, 2009. doi : 10.1007/978-3-642-03034-5_5.

[29] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Gaurav Sukhatme, et

Brad Petrus. Architecture-driven self-adaptation and self-management in robotics systems. Dans *SEAMS'09 : Proceedings of the 4th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 142–151, Los Alamitos, CA, USA, 2009. doi : 10.1109/SEAMS.2009.5069083.

[30] Torbjörn Ekman et Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9) :455–475, 2007. doi : 10.5381/jot.2007.6.9.a23.

[31] Torbjörn Ekman et Görel Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1-3) :14–26, 2007. doi : 10.1016/j.scico.2007.02.003.

[32] Quentin Enard. Projection de politiques de tolérance aux fautes dans les architectures logicielles. Master's thesis, University of Bordeaux, 2009.

[33] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3) :17–23, 2000. doi : 10.1109/6294.846201.

[34] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, et Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2) :114–131, 2003. doi : 10.1145/857076.857078.

[35] Martin Fowler. UML mode, 2003. url : http://www.martinfowler.com/bliki/UmlMode.html.

[36] Martin Fowler. UML as programming language, 2003. url : http://www.martinfowler.com/bliki/UmlAsProgrammingLanguage.html.

[37] Martin Fowler. UML as sketch, 2003. url : http://www.martinfowler.com/bliki/UmlAsSketch.html.

[38] Martin Fowler. Fluent interface, 2005. url : http://www.martinfowler.com/bliki/FluentInterface.html.

[39] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010. isbn : 0321712943.

[40] Martin Fowler et Jim Highsmith. The Agile manifesto. *Software Development Magazine*, 9(8) :29–30, 2001.

[41] Piero Fraternali. Tools and approaches for developing data-intensive web applications : a survey. *ACM Computing Surveys*, 31(3) :227–263, 1999. doi : 10.1145/331499.331502.

[42] Steve Freeman et Nat Pryce. Evolving an embedded domain-specific language in Java. Dans *OOPSLA'06 : Companion to the 21st Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 855–865, Portland, OR, USA, 2006. doi : 10.1145/1176617.1176735.

[43] Krzysztof Gajos, Harold Fox, et Howard Shrobe. End user empowerment in human centered pervasive computing. Dans *Pervasive'02 : Proceedings of the 1st International Conference on Pervasive Computing*, pages 134–140, Zurich, Switzerland, 2002.

[44] Erich Gamma, Richard Helm, Ralph Johnson, et John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995. isbn : 978-0201633610.

[45] Alan G. Ganek et Thomas A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1) :5–18, 2003. doi : 10.1147/sj.421.0005.

[46] Benoît Garbinato et Philippe Rupp. From ad hoc networks to ad hoc applications. Dans *ConTEL'03 : Proceedings of the 7th International Conference on Telecommunications*, pages 145–149, Zagreb, Croatia, 2003. doi : 10.1109/CONTEL.2003.176903.

[47] David Garlan. An introduction to the Aesop system. Technical report, Carnegie Mellon University, 1995.

[48] David Garlan, Robert T. Monroe, et David Wile. ACME : An architecture description interchange language. Dans *CASCON'97 : Proceedings of the 7th Conference of the Centre for Advanced Studies on Collaborative Research*, pages 169–183, Toronto, Ontario, Canada, 1997. doi : 10.1145/782010.782017.

[49] Stéphanie Gatti, Emilie Balland, et Charles Consel. A stepwise approach for integrating QoS throughout software development. Dans *FASE'11 : Proceedings of the 14th European Conference on Fundamental Approaches to Software Engineering*, Saarbrücken, Germany, 2011. to appear.

[50] Antonio Goncalves. *Beginning Java EE 6 with GlassFish 3*. Apress, second edition, 2010. isbn : 978-1430228899.

[51] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, et David Wetherall. System support for pervasive applications. *ACM Transactions on Computer Systems*, 22(4) :421–486, 2004. doi : 10.1145/1035582.1035584.

[52] Stefan Hanenberg. Faith, hope, and love : an essay on software science's neglect of human factors. Dans *OOPSLA'10 : Proceedings of the 25th International Conference on Object Oriented Programming Systems Languages and Applications*, pages 933–946, Reno/Tahoe, NV, USA, 2010. doi : 10.1145/1869459.1869536.

[53] Michael Haupt, Bram Adams, Stijn Timbermont, Celina Gibbs, Yvonne Coady, et Robert Hirschfeld. Disentangling virtual machine architecture. *IET Software*, 3(3) :201–218, 2009. doi : 10.1049/iet-sen.2007.0121.

[54] Karen Henricksen, Jadwiga Indulska, et Andry Rakotonirainy. Modeling context information in pervasive computing systems. Dans *Pervasive'02 : Proceedings of the 1st International Conference on Pervasive Computing*, pages 167–180, London, UK, 2002. doi : 10.1007/3-540-45866-2_14.

[55] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, 2003. isbn : 0321228626.

[56] Michael Huth et Mark Ryan. *Logic in Computer Science : Modelling and Reasoning about Systems*. Cambridge, second edition, 2004. isbn : 0-521-54310-X.

[57] IBM. An architectural blueprint for autonomic computing. Technical Report June, IBM, 2006.

[58] Shahram Izadi, Pedro Coutinho, Tom Rodden, et Gareth Smith. The FUSE platform : Supporting ubiquitous collaboration within diverse mobile environments. *Automated Software Engineering*, 9 :167–186, 2002. doi : 10.1023/A:1014534414062.

[59] Henner Jakob, Nicolas Loriant, et Charles Consel. An aspect-oriented approach to securing distributed systems. Dans *ICPS'09 : Proceedings of the 6th International Conference on Pervasive Services*, pages 21–30, London, UK, 2009. doi : 10.1145/1568199.1568204.

[60] Wilfried Jouve, Nicolas Palix, Charles Consel, et Patrice Kadionik. A SIP-based programming framework for advanced telephony applications. Dans *IPTComm'08 : Proceedings of the 2nd Conference on Principles, Systems and Applications of IP Telecommunications*, pages 1–20, Heidelberg, Germany, 2008. doi : 10.1007/978-3-540-89054-6_1.

[61] Jevgeni Kabanov et Rein Raudjärv. Embedded typesafe domain specific languages for Java. Dans *PPPJ'08 : Proceedings of the 6th International Symposium on Principles and Practice*

of Programming in Java, pages 189–197, Modena, Italy, 2008. doi : 10.1145/1411732.1411758.

[62] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, et Lisa Walton. A software engineering experiment in software component generation. Dans *ICSE'96 : Proceedings of the 18th International Conference on Software engineering*, pages 542–552, Berlin, Germany, 1996.

[63] Claude Y. Knaus. Essential programming paradigm. Dans *OOPSLA'08 : Companion to the 23rd Conference on Object-Oriented Programming Systems Languages And Applications*, pages 823–826, Nashville, TN, USA, 2008. doi : 10.1145/1449814.1449873.

[64] Alan Knight et Naci Dai. Objects and the web. *IEEE Software*, 19(2) :51 –59, 2002. doi : 10.1109/52.991332.

[65] Harilaos Koumaras, Daniel Negrou, Fidel Liberal, Javier Arauz, et Anastasios Kourtis. ADAMANTIUM project : Enhancing IMS with a PQoS-aware multimedia content management system. Dans *AQTR'08 : Proceedings of the 16th International Conference on Automation, Quality and Testing, Robotics*, pages 358–363, Cluj-Napoca, Romania, 2008. doi : 10.1109/AQTR.2008.4588768.

[66] Glenn E. Krasner et Stephen T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3) : 26–49, 1988.

[67] Caroline Lu. *Robustesse du logiciel embarqué multicouche par une approche rflexive : application à l'automobile*. PhD thesis, University of Toulouse, 2009.

[68] Caroline Lu, Jean-Charles Fabre, et Marc-Olivier Killijian. An approach for improving fault-tolerance in automotive modular embedded software. Dans *RTNS'09 : Proceedings of the 17th International Conference on Real-Time and Network Systems*, pages 132–147, Paris, France, 2009.

[69] David C. Luckham. Rapide : A language and toolset for simulation of distributed systems by partial ordering of events. Dans *DIMACS Partial Order Methods Workshop IV*. 1996.

[70] David C. Luckham. *The Power of Events : An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001. isbn : 0201727897.

[71] David C. Luckham et James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9) :717–734, 1995. doi : 10.1109/32.464548.

[72] Nancy A. Lynch et Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Dans *PODC'87 : Proceedings of the 6th Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, BC, Canada, 1987. doi : 10.1145/41840.41852.

[73] Jeff Magee. Behavioral analysis of software architectures using LTSA. Dans *ICSE'99 : Proceedings of the 21st International Conference on Software Engineering*, pages 634–637, Los Angeles, CA, USA, 1999. doi : 10.1145/302405.302726.

[74] Jeff Magee, Naranker Dulay, Susan Eisenbach, et Jeffrey Kramer. Specifying distributed software architectures. Dans *ESEC'95 : Proceedings of the 5th European Software Engineering Conference*, pages 137–153, Sitges, Spain, 1995. doi : 10.1007/3-540-60406-5_12.

[75] Louis Mandel et Marc Pouzet. ReactiveML, a reactive extension to ML. Dans *PPDP'05 : Proceedings of the 7th International conference on Principles and Practice of Declarative Programming*, pages 82–93, Lisbon, Portugal, 2005. doi : 10.1145/1069774.1069782.

[76] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4) :308–320, 1976. doi : 10.1109/TSE.1976.233837.

[77] MDA. OMG model driven architecture, 2000. http ://www.omg.org/mda/.

[78] Nenad Medvidovic et Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, 2000. doi : 10.1109/32.825767.

[79] Nenad Medvidovic, David S. Rosenblum, et Richard N. Taylor. A language and environment for architecture-based software development and evolution. Dans *ICSE'99 : Proceedings of the 21st International Conference on Software engineering*, pages 44–53, Los Angeles, CA, USA, 1999. doi : 10.1145/302405.302410.

[80] Nenad Medvidovic, Eric M. Dashofy, et Richard N. Taylor. Moving architectural description from under the technology lamppost. *Information and Software Technology*, 49(1) : 12–31, 2007. doi : 10.1016/j.infsof.2006.08.006.

[81] Julien Mercadal, Nicolas Palix, Charles Consel, et Julia Lawall. Pantaxou : A domain-specific language for developing safe coordination services. Dans *GPCE'08 : Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, pages 149–160, Nashville, TN, USA, 2008. doi : 10.1145/1449913.1449936.

[82] Julien Mercadal, Quentin Enard, Charles Consel, et Nicolas Loriant. A domain-specific approach to architecturing error handling in pervasive computing. Dans *OOPSLA'10 : Proceedings of the 25th International Conference on Object Oriented Programming Systems Languages and Applications*, pages 47–61, Reno/Tahoe, NV, USA, 2010. doi : 10.1145/1869459.1869465.

[83] OMG. The common object request broker : Architecture and specification. Technical Report 2.0, Object Management Group, 1995.

[84] OSGi. Osgi alliance, 2000. url : http://www.osgi.org.

[85] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, et Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. Dans *EuroSys'08 : Proceedings of the 3rd European Conference on Computer Systems*, pages 247–260, Glasgow, UK, 2008. doi : 10.1145/1352592.1352618.

[86] Ruben Picek et Vjeran Strahonja. Model Driven Development - future or failure of software development ? Dans *IIS'07 : Proceedings of the 18th International Conference on Information and Intelligent Systems*, pages 407–413, Varazdin, Croatia, 2007.

[87] Anand Ranganathan et Roy H. Campbell. Advertising in a pervasive computing environment. Dans *WMC'02 : Proceedings of the 2nd International workshop on Mobile commerce*, pages 10–14, Atlanta, GA, USA, 2002. doi : 10.1145/570705.570708.

[88] Anand Ranganathan, Shiva Chetan, Jalal Al-Muhtadi, Roy H. Campbell, et M. Dennis Mickunas. Olympus : A high-level programming model for pervasive computing environments. Dans *PerCom'05 : Proceedings of the 3rd International Conference on Pervasive Computing and Communications*, pages 7–16, Kauai, HI, USA, 2005. doi : 10.1109/PERCOM.2005.26.

[89] Lukas Renggli. *Dynamic Language Embedding With Homogeneous Tool Support*. Phd thesis, University of Bern, 2010.

[90] Lukas Renggli, Tudor Gîrba, et Oscar Nierstrasz. Embedding languages without breaking tools. Dans *ECOOP'10 : Proceedings of the 24th European Conference on Object-Oriented Programming*, pages 380–404, Maribor, Slovenia, 2010. doi : 10.1007/978-3-642-14107-2_19.

[91] Vinny Reynolds, Vinny Cahill, et Aline Senart. Requirements for an ubiquitous computing simulation and emulation environment. Dans *InterSense'06 : Proceedings of the 1st International Conference on Integrated Internet Ad hoc and Sensor Networks*, pages 1–9, Nice, France, 2006. doi : 10.1145/1142680.1142682.

[92] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, et Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4) :74–83, 2002. doi : 10.1109/MPRV.2002.1158281.

[93] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, et Eve Schooler. SIP : Session Initiation Protocol. Technical report, RFC 3261, 2002.

[94] David Schmidt. Software architecture, an informal introduction, 2008. url : http://www.cis.ksu.edu/santos/schmidt/EJCP/talk08.pdf.

[95] Douglas C. Schmidt. Guest editor's introduction : Model-driven engineering. *Computer*, 39(2) :25–31, 2006. doi : 10.1109/MC.2006.58.

[96] Douglas C. Schmidt et Frank Buschmann. Patterns, frameworks, and middleware : their synergistic relationships. Dans *ICSE'03 : Proceedings of the 25th International Conference on Software Engineering*, pages 694–704, Portland, OR, USA, 2003. doi : 10.1109/ICSE.2003.1201256.

[97] João Costa Seco et Luís Caires. A basic model of typed components. Dans *ECOOP'00 : Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, London, UK, 2000. doi : 10.1007/3-540-45102-1_6.

[98] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, et Jean-Bernard Stefani. Reconfigurable SCA applications with the FraSCAti platform. Dans *SCC'09 : Proceedings of the 6th International Conference on Service Computing*, pages 268–275, Bangalore, India, 2009. doi : 10.1109/SCC.2009.27.

[99] Estefanía Serral, Pedro Valderas, et Vicente Pelechano. Towards the model driven development of context-aware pervasive systems. *Pervasive and Mobile Computing*, 6(2) : 254–280, 2010. doi : 10.1016/j.pmcj.2009.07.006.

[100] Mary Shaw et Paul Clements. The golden age of software architecture : A comprehensive survey. *IEEE Software*, 23 (2) :31–39, 2006. doi : 10.1109/MS.2006.58.

[101] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, et Gregory Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4) :314–335, 1995. doi : 10.1109/32.385970.

[102] Bruno Siciliano et Oussama Khatib, editors. *Springer Handbook of Robotics*. Springer, 2008. isbn : 978-3-540-23957-4.

[103] Fausto Spoto. Nullness analysis in boolean form. Dans *SEFM'08 : Proceedings of the Sixth International Conference on Software Engineering and Formal Methods*, pages 21–30, Washington, DC, USA, 2008. doi : 10.1109/SEFM.2008.8.

[104] Vugranam C. Sreedhar. Mixin'up components. Dans *ICSE'02 : Proceedings of the 24th International Conference on Software Engineering*, pages 198–207, Orlando, FL, USA, 2002. doi : 10.1145/581339.581366.

[105] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr. E. James Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy, et Deborah L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22(6) :390–406, 1996. doi : 10.1109/32.508313.

[106] Richard N. Taylor, Nenad Medvidovic, et Eric M. Dashofy. *Software Architecture : Foundations, Theory, and Practice*. Wiley, 2009. isbn : 978-0470167748.

[107] Dave Thomas. MDA : Revenge of the modelers or UML utopia ? *IEEE Software*, 21(3) :15–17, 2004. doi : 10.1109/MS.2004.1293067.

[108] Naoyasu Ubayashi, Jun Nomura, et Tetsuo Tamai. Archface : A contract place where architectural design and code meet together. Dans *ICSE'10 : Proceedings of the 32nd International Conference on Software Engineering*, pages 75–84, Cape Town, South Africa, 2010. doi : 10.1145/1806799.1806815.

[109] Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, et Wolfgang De Meuter. Ambienttalk : Object-oriented event-driven programming in mobile ad hoc networks. Dans *SCCC'07 : Proceedings of the 26th International Conference of the Chilean Computer Science Society*, pages 3–12, Iquique, Chile, 2007. doi : 10.1109/SCCC.2007.12.

[110] Peter Van Roy. Self management and the future of software design. Dans *FACS'06 : Proceedings of the 3rd International Workshop on Formal Aspects of Component Software*, pages 201–217, Prague, Czech Republic, 2006. doi : 10.1016/j.entcs.2006.12.043.

[111] Eric Van Wyk, Lijesh Krishnan, Derek Bodin, et August Schwerdfeger. Attribute grammar-based language extensions for java. Dans *ECOOP'07 : Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 575–599, Berlin, Germany, 2007. doi : 10.1007/978-3-540-73589-2_27.

[112] John Vlissides. Generation gap. *C++ Report*, 1996.

[113] Guillaume Waignier, Sriplakich Prawee, Anne-Françoise Le Meur, et Laurence Duchien. A model-based framework for statically and dynamically checking component interactions. Dans *MODELS'08 : Proceedings of the 11th International Conference on Model-Driven Engineering Languages and Systems*, pages 371–385, Toulouse, France, 2008. doi : 10.1007/978-3-540-87875-9_27.

[114] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3) :66–75, 1991.

[115] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, et Anders Wesslén. *Experimentation in Software Engineering*. Kluwer Academic Publishers, 2000. isbn : 978-0792386827.