A Design-Driven Methodology for the Development of Large-Scale Orchestrating Applications

Milan Kabac

Université de Bordeaux

Sciences et Technologies

L'Ecole Doctorale de Mathématiques et Informatique

Une méthodologie dirigée par la conception pour le développement d'applications d'orchestration à grande échelle

THÈSE

pour obtenir le grade de

Docteur de l'Université de Bordeaux
(specialité Informatique)

par

Milan Kabáč

Soutenue le 26 Septembre 2016, devant le jury composé de :

Président :	Denis Barthou,	Professeur à Bordeaux INP
Directeur de thèse :	Charles Consel,	Professeur à Bordeaux INP
Rapporteurs :	Julie A. McCann,	Professeur à l'Imperial College de Londres
	François Taïani,	Professeur à l'Université de Rennes
Examinateurs :	Walid Taha,	Professeur à l'Université d'Halmstad
	Nic Volanschi,	Advanced research position à Inria Bordeaux

University of Bordeaux

Department of Science and Technology

A Design-Driven Methodology for the Development of Large-Scale Orchestrating Applications

Thesis submitted for the degree of

Doctor of Philosophy of the University of Bordeaux

presented by

Milan Kabáč

Defended on 26 September 2016 before the following thesis committee:

Supervisor:	Prof. Charles Consel,	Bordeaux INP
Thesis committee:	Prof. Denis Barthou,	Bordeaux INP
	Prof. Julie A. McCann,	Imperial College London
	Prof. Walid Taha,	Halmstad University
	Prof. François Taïani,	University of Rennes/IRISA/Inria
	Dr. Nic Volanschi,	Inria Bordeaux

Abstract

Our environment is increasingly populated with large amounts of smart objects. Some monitor free parking spaces, others analyze material conditions in buildings or detect unsafe pollution levels in cities. The massive amounts of sensing and actuation devices constitute large-scale infrastructures that span over entire parking lots, campuses of buildings or agricultural fields. Despite being successfully deployed in a number of domains, the development of applications for such infrastructures remains challenging. Considerable knowledge about the hardware/network specificities of the sensor infrastructure is required on the part of the developer. To address this problem, software development methodologies and tools raising the level of abstraction need to be introduced to allow non-expert developers program applications.

This dissertation presents a design-driven methodology for the development of applications orchestrating massive amounts of networked objects. The methodology is based on a domain-specific design language, named DiaSwarm that provides high-level, declarative constructs allowing developers to deal with masses of objects at design time, prior to programming the application. Generative programming is used to produce design-specific programming frameworks to guide and support the development of applications in this domain. The methodology integrates the parallel processing of large-amounts of data collected from masses of sensors. We introduce specific language declarations resulting in the generation of programming frameworks based on the MapReduce programming model. We furthermore investigate how design can be used to make explicit the resources required by applications as well as their usage. To match the application requirements to a target sensor infrastructure, we consider design declarations at different stages of the application lifecycle.

The scalability of this approach is evaluated in an experiment, which shows how the generated programming frameworks relying on the MapReduce programming model are used for the efficient processing of large datasets of sensor readings. We examine the effectiveness of the proposed approach in dealing with key software engineering challenges in this domain by implementing application scenarios provided to us by industrial partners. We solicited professional programmers to evaluate the usability of our approach and present quantitative and qualitative data from the experiment.

Keywords: *software development, domain-specific languages, design, generative programming, sensors, actuators, orchestration*

Résumé

Notre environnement est de plus en plus peuplé de grandes quantités d'objets intelligents. Certains surveillent des places de stationnement disponibles, d'autres analysent les conditions matérielles dans les bâtiments ou détectent des niveaux de pollution dangereux dans les villes. Les quantités massives de capteurs et d'actionneurs constituent des infrastructures de grande envergure qui s'étendent sur des terrains de stationnement entiers, des campus comprenant plusieurs bâtiments ou des champs agricoles. Le développement d'applications pour de telles infrastructures reste difficile, malgré des déploiement réussis dans un certain nombre de domaines. Une connaissance considérable des spécificités matériel / réseau de l'infrastructure de capteurs est requise de la part du développeur. Pour remédier à ce problème, des méthodologies et des outils de développement logiciel permettant de relever le niveau d'abstraction doivent être introduits pour que des développeurs non spécialisés puissent programmer les applications.

Cette thèse présente une méthodologie dirigée par la conception pour le développement d'applications orchestrant des quantités massives d'objets communicants. La méthodologie est basée sur un langage de conception dédié, nommé DiaSwarm qui fournit des constructions déclaratives de haut niveau permettant aux développeurs de traiter des masses d'objets en phase de conception, avant de programmer l'application. La programmation générative est utilisée pour produire des cadres de programmation spécifiques à la conception pour guider et soutenir le développement d'applications dans ce domaine. La méthodologie intègre le traitement parallèle de grandes quantités de données collectées à partir de masses de capteurs. Nous introduisons un langage de déclarations permettant de générer des cadres de programmation basés sur le modèle de programmation MapReduce. En outre, nous étudions comment la conception peut être utilisée pour rendre explicites les ressources requises par les applications ainsi que leur utilisation. Pour faire correspondre les exigences de l'application à une infrastructure de capteurs cible, nous considérons les déclarations de conception à différents stades du cycle de vie des applications.

Le passage à l'échelle de cette approche est évaluée dans une expérience qui montre comment les cadres de programmation générés s'appuyant sur le modèle de programmation MapReduce sont utilisés pour le traitement efficace de grands ensembles de données de relevés des capteurs. Nous examinons l'efficacité de l'approche proposée pour relever les principaux défis du génie logiciel dans ce domaine en mettant en œuvre des scénarios d'application qui nous sont fournis par des partenaires industriels. Nous avons sollicité des programmeurs professionnels pour évaluer l'utilisabilité de notre approche et présenter des données quantitatives et qualitatives de l'expérience.

Mots clefs : *développement logiciel, langages dédiés, conception, programmation générative, capteurs, actionneurs, orchestration*

Résumé étendu

Cette thèse intitulée « Une méthodologie dirigée par la conception pour le développement d'applications d'orchestration à grande échelle » se situe dans le domaine du génie logiciel et répond au défi de la programmation d'applications pour des infrastructures de grande envergure composées de quantités massives de capteurs. La principale contribution de cette thèse est une approche de développement logiciel dédiée au domaine de l'orchestration de masses de capteurs et d'actionneurs. Cette approche est basée sur un langage de conception, qui fournit des constructions déclaratives de haut niveau permettant aux développeurs de gérer des masses d'objets communiquants en phase de conception avant de programmer l'application.

Problématique

Le processus de développement de services pour des infrastructures de masses de capteurs est aujourd'hui complexe du fait de l'utilisation d'approches qui sont souvent de bas niveau et centrées sur le réseau ou le matériel. Les pratiques actuelles sont pilotées par les opérateurs de réseaux ou centrées autour des préoccupations des fabricants de capteurs. Cette approche centrée sur le réseau ou le matériel rend le développement du logiciel complexe, avec pour conséquence une courbe d'apprentissage abrupte pour les programmeurs. Cette situation peut être un obstacle majeur au succès du domaine.

Développer des applications qui orchestrent des masses d'objets implique des défis majeurs en raison de l'échelle à laquelle cette orchestration a lieu. Dans cette thèse, nous abordons ces défis en examinant les phases conceptuelles typiques d'une application d'orchestration dans ce domaine à savoir la découverte de services, la collecte de données, le traitement de données et l'actionnement.

Il est essentiel de comprendre les préoccupations des experts du domaine, ainsi que de résoudre les problèmes communs rencontrés par les développeurs tout au long du processus de développement logiciel pour assurer le succès d'une approche de développement logiciel. Pour atteindre cet objectif, nous avons participé à un projet collaboratif français, appelé Objects World, tout au long de cette thèse. Ce projet vise à construire un écosystème durable des acteurs dans le domaine de l'internet des objets. Cette thèse présente et aborde les principaux défis liés au génie logiciel qui ont été identifiés à la suite de nombreuses interactions avec les sociétés de ce consortium.

Contributions

Pour faire face aux nombreux défis de l'orchestration à grande échelle, cette thèse propose une approche de développement logiciel couvrant tout le cycle de vie d'une application d'orchestration. Cette approche rend explicite l'expertise du domaine nécessaire à l'aide d'un langage de conception dédié afin de guider le développement et le déploiement d'une application d'orchestration à grande échelle. Par ailleurs, pour traiter de grandes quantités de données de capteurs, le langage fournit aux développeurs des déclarations pour exprimer le traitement des données tout en faisant abstraction des complexités liées au traitement parallèle haute performance.

Nos contributions s'articulent autour des thèmes suivants: (1) support de développement logiciel, (2) traitement de données haute performance, (3) infrastructures des réseaux de capteurs et (4) validation de notre approche. Nous illustrons les traits saillants de notre approche à l'aide d'une étude de cas, qui examine le développement d'une application de gestion de stationnement. Le but de cette application est de surveiller l'occupation des parcs de stationnement et de réguler le flux de circulation pour diriger les voitures vers les places de stationnement disponibles.

Nous avons développé un compilateur pour notre langage qui génère du support de programmation de haut niveau à partir de la spécification d'une application. Les déclarations détaillant la conception d'une application sont compilées dans un cadre de programmation spécifique à l'application qui guide la programmation de la logique d'orchestration. Cette stratégie permet au développeur de s'abstraire des spécificités du réseau de capteurs et d'assurer des stratégies de traitement de données appropriées pour atteindre les performances requises. Les cadres de programmation générés s'appuient sur le modèle de programmation MapReduce pour fournir au développeur une approche éprouvée pour le traitement efficace d'ensembles de données volumineux. Cette stratégie permet de faire face à de grands ensembles de données collectés à partir de masses de capteurs.

Pour assurer que l'infrastructure des objets communiquants puisse fournir à l'application les capteurs et les actionneurs requis, nous introduisons la notion de *comportement applicatif*, qui regroupe les caractéristiques d'une application d'orchestration à grande échelle concernant le réseau de capteurs. En tant que tel, les caractéristiques d'une application peuvent être exprimées au début du processus de développement pour fournir un support tout au long du cycle de vie de l'application. Nous illustrons comment les caractéristiques liées au réseau de capteurs peuvent être exprimées via des déclarations de haut niveau et utilisées tout au long du cycle de vie des applications d'orchestration. Nous introduisons des étapes le long du cycle de vie de l'application où les déclarations de *comportement applicatif* peuvent être utilisées pour adapter à la fois l'application et l'infrastructure des objets communicants. Ce processus d'adaptation permet par exemple de vérifier que les capacités en termes de capteurs requises par une application au moment de la conception sont compatibles avec l'infrastructure cible.

Nous avons implémenté notre approche et nous l'avons utilisée pour programmer différentes applications. Pour la phase de développement, notre approche prend la forme d'un plugin pour l'environnement de développement Eclipse [1]. Le plugin est disponible publiquement [2] et fournit aux développeurs notre langage de conception et un générateur de code. Pour le traitement des données, notre compilateur génère actuellement des cadres de programmation ciblant la plate-forme Apache Hadoop [3].

Nous avons évalué le passage à l'échelle de notre approche dans une expérience comportant l'exécution des calculs sur un grand ensemble de données synthétiques. Nous démontrons que notre approche basée sur la conception permet de s'affranchir des détails de l'implémentation tout en exposant les propriétés architecturales utilisées pour générer du code haute performance pour le traitement de grands ensembles de données. Nous avons évalué l'efficacité de notre approche pour résoudre les défis de génie logiciel identifiés en évaluant le support fourni par notre approche pour trois applications fournies par les entreprises du consortium du projet Objects World. Nous montrons que

1. http://eclipse.org
2. http://phoenix.inria.fr/software/diaswarm
3. http://hadoop.apache.org/

notre approche fournit un support efficace pour la programmation d'une large gamme d'applications pour l'internet des objets, et permet aux programmeurs experts de prototyper les applications rapidement. Nous avons évalué la facilité d'utilisation de notre approche en sollicitant des programmeurs professionnels de l'industrie dans une étude d'utilisabilité. Nous fournissons des données quantitatives et qualitatives, y compris les résultats d'un questionnaire d'utilisabilité et des entrevues avec les développeurs afin d'étudier l'utilité perçue d'une approche de développement logiciel dirigée par conception.

Structure du document

Cette thèse est organisée de la manière suivante. Le premier chapitre situe le contexte de ce travail. Ce chapitre présente les défis auxquels sont confrontés les programmeurs tout au long du développement d'applications de haut niveau orchestrant des quantités massives de capteurs. Pour assurer l'adoption d'une approche pour le développement d'applications d'orchestration à grande échelle, nous répertorions les défis de génie logiciel identifiés à la suite de nombreuses interactions avec des partenaires du projet Objects World. Le chapitre énumère les différentes contributions de ce travail et présente la structure de ce document. Le chapitre 2 présente les domaines de recherche de l'informatique ubiquitaire, réseaux de capteurs et de l'internet des objets, afin d'étudier comment sont développés actuellement des systèmes composés d'objets communicants, dotés de capacités de détection et actionnement. Le chapitre examine les principales préoccupations dans chaque domaine en ce qui concerne les défis d'orchestration à grande échelle présentés dans chapitre 1. Dans le chapitre 3, nous présentons une étude de cas qui est utilisée tout au long du document pour illustrer les contributions de cette thèse, qui considère le développement d'un service de ville intelligente à grande échelle pour la gestion des places de stationnement dans les parkings. Notre approche dirigée par la conception, dédiée au développement d'applications d'orchestration à grande échelle est présentée dans le chapitre 4. Nous présentons un langage dédié à la manipulation d'objets communicants à grande échelle et nous montrons comment la spécification de l'application est compilée dans un cadre de programmation spécifique à l'application pour soutenir et guider le processus de développement. Nous étendons notre approche de conception dans le chapitre 5 pour introduire le traitement parallèle de grandes quantités de données collectées à partir de capteurs. Nous démontrons comment les déclarations de conception sont utilisées pour générer des cadres de programmation utilisant le modèle de programmation MapReduce pour un traitement efficace des données. Nous évaluons le passage à l'échelle de notre approche dans une expérience consistant à exécuter des calculs sur un grand ensemble de données. Nous explorons davantage la phase de conception dans le chapitre 6 pour déterminer comment les déclarations peuvent être utilisées pour rendre explicites les ressources requises par les applications ainsi que leur utilisation. Nous introduisons des étapes le long du cycle de vie des applications et discutons comment les déclarations peuvent être exploitées à chaque étape.

L'évaluation de l'approche proposée est présentée dans le chapitre 7 via deux expériences différentes. Dans une première expérience, nous examinons l'efficacité de l'approche dirigée par conception pour faire face aux défis de génie logiciel identifiés. Dans une deuxième expérience, nous mesurons, via une étude d'utilisabilité, le coût de l'apprentissage pour utiliser notre approche. Les conclusions de cette thèse et la discussion des avenues de recherche en cours et à venir sont présentées dans le chapitre 8.

Publications

Conferences

Orchestrating Masses of Sensors: A Design-Driven Development Approach
Proceedings of the 14th ACM SIGPLAN International Conference on Generative Programming: Concepts & Experience (GPCE), 2015, Milan Kabáč and Charles Consel

Designing Parallel Data Processing for Large-Scale Sensor Orchestration
Proceedings of the 13th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC), 2016, Milan Kabáč and Charles Consel, **Best Paper Award**

Leveraging Declarations over the Lifecycle of Large-Scale Sensor Applications
Proceedings of the 13th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC), 2016, Milan Kabáč, Charles Consel, Nic Volanschi

Workshops

An Evaluation of the DiaSuite Toolset by Professional Developers
Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU), 2015, Milan Kabáč, Nic Volanschi, Charles Consel

Journals

Internet of Things: A Challenge for Software Engineering
ERCIM News, Smart Cities, ERCIM, 2014, pp.20-21., Charles Consel and Milan Kabáč

Posters

An Evaluation of the DiaSuite Toolset by Professional Developers
ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH), 2015, Pittsburgh, Pennsylvania, US, Milan Kabáč, Nic Volanschi, Charles Consel

Orchestrating Masses of Sensors: A Design-Driven Development Approach
ACM SIGPLAN conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH), 2015, Pittsburgh, Pennsylvania, US, Milan Kabáč and Charles Consel

Acknowledgements

I would like to thank the people who have supported and helped me so much throughout this period. First and foremost, I would like to express my sincere gratitude to my doctoral supervisor Charles Consel for his guidance and trust during the last four years. His encouragement, high scientific standards and immense knowledge taken together, make him a great mentor. Without his support this dissertation would not have been possible.

My sincere thanks goes to Denis Barthou, Julie McCann, Walid Taha and François Taïani for accepting to participate on my dissertation committee as well as for their insightful comments. I want to express my deep thanks to Nic Volanschi for his enthusiasm, valuable advice and creative ideas, which I greatly appreciate. I also want to thank him for accepting to participate on my dissertation committee. It was a real pleasure working with you and I look forward to future opportunities for collaboration.

In my daily work I have been blessed to work with a friendly and cheerful group of people from the Phoenix team. I would like to thank Damien, Emilie, Julien B. and Quentin E. for introducing me into the lab, their valuable advice and for sharing their knowledge with me. I am especially grateful to Adrien, Lucile, Charles F. and Paul for being there for me when I needed it and for all the interesting discussions and their advice. I am also grateful to Charlotte, Julien D., Ludovic, Maeëlle, Pauline, Quentin B. for providing a fun and stimulating working environment and all the occasional game of foosball.

I want to thank David Sherman for his help with the computer cluster that we used for some of the experimental evaluation in this dissertation. I also want to thank Catherine, Cécile and Chrystel for their help with all the different administrative procedures.

I would like to thank my family for the support they provided me through my entire life. I am especially thankful to my mother who always believed in me and without whose sacrifice this achievement would not be possible. A very special thanks goes to Veronica without whose love, positive energy and encouragement I would not have finished this dissertation. You showed a great deal of understanding and patience when I needed it the most, for which I am infinitely grateful to you. I am deeply grateful to Ivan who taught me the important lessons in life and for all the emotional and material support he provided. I also want to express my gratitude to Roman, Maroš and Tomáš who have been always there for me and who supported me every step of the way.

Many thanks goes to Alan, Cyril, Matthieu, Romain, Sebastien, Thomas C., Thomas F., William and Xavier for all the great football games and for being there whenever I needed a friend. I would like to thank Alan and Matthieu for sharing with me the template for this dissertation, which saved me a lot of time. I am also grateful to Sebastien for providing comments on the draft version of my work.

I am extremely grateful to Anne-Marie who greatly helped me when I first arrived in France and who has been very supportive ever since.

Contents

1

Introduction

Masses of sensors and actuators are increasingly emerging in our daily environments to provide innovative smart services, including parking management [libelium, 2013], traffic monitoring [IBM, 2013], wide-area transportation management [Mizuno and Odake, 2015; Naphade et al., 2011], *etc.* These infrastructures are being increasingly deployed over large-scale spaces, including parking lots in cities and agricultural fields in rural areas. Large-scale sensor infrastructures are now being operated worldwide by companies, enabling economically viable services to be offered. Although existing deployments demonstrate the maturity and practicality of such infrastructures, there are still challenges that need to be addressed to harness the potential benefits of this technology to provide users with innovative and useful services. To achieve this goal, developing software is a crucial activity that enables exploring the scope of potential services, anticipating and responding to users' needs.

Contents

Overview

- Overview of challenges faced by programmers throughout the development of high-level application services orchestrating massive amounts of sensors.
- Overview of software engineering challenges that need to be addressed to ensure the adoption of an approach for the development of large-scale orchestrating applications.

The process of developing services for masses of sensors raises a number of challenges due to the use of approaches that are often low level and network/hardware-centric. Current practices are driven by network operators [1] and centered around the concerns of specific stakeholders: sensor manufacturers. Moreover, research in the domain of sensor networks often ignores realistic application-specific requirements (*e.g.,* expected data traffic, location information and granularity, *etc.*), as discussed by Raman *et al.* [Raman and Chebrolu, 2008], and lacks programming models and methodologies for addressing key domain-specific challenges. This network/hardware-centered approach makes software development low level and feature specific, resulting in a steep learning curve for programmers. This situation can be a major impediment for the success of the domain.

1.1 Large-Scale Orchestration Challenges

Orchestrating applications [2] discover devices (*i.e.,* sensors/actuators), gather and process data from sensors, and possibly trigger actuators. Developing applications that orchestrate masses of objects raises major challenges because of the scale at which this orchestration takes place. In this section, we introduce the main challenges by reviewing the typical conceptual phases of an orchestrating application, namely service discovery, data gathering, data processing, and actuating.

Service discovery

In contrast with standard service discovery that addresses individual objects [Zhu et al., 2005], masses of sensors demand a high-level approach to designating subsets of interest. Specifically, selecting objects of interest among a myriad of objects should be tamed by application-specific abstractions that provide meaningful constructs for grouping sensors. For example, an application may need to manipulate parking spaces at the level of lots or districts. The developer should be able to directly express these application-specific concepts. Beyond expressiveness, when considering masses of sensors, the scalability of a service discovery mechanism is critical to making an orchestrating application usable. In this context, exploiting information about the application behavior is essential to reduce the cost of such activities as service discovery and data gathering, as shown by various works [Liu et al., 2007; Heidemann et al., 2003; Krishnamachari and Heidemann, 2004]. Furthermore, it has been shown that a mismatch between the application behavior and the network routing algorithms can result in poor performance [Krishnamachari and Heidemann, 2004].

1. *e.g.,* SIGFOX, http://www.sigfox.com
2. We use the terms "orchestration" and "orchestrating application" interchangeably.

Data gathering

Models used for delivering data to applications must accommodate masses of sources. For example, applications may require data to be pushed from any number of CO (carbon monoxide) sensors located in underground parking lots, when a given air pollution level is reached. Delivery models have a direct impact on the structure and the logic of an application and thus need to be explicit to avoid mismatches between application requirements and the target sensor network infrastructure (*e.g.,* mismatch between delivery frequency of sensors and the frequency required by an application). Besides, making explicit the delivery models used by an application prior to programming can be valuable information to ensure an optimal routing structure of the underlying sensor network [Liu et al., 2007].

Data processing

When considering tens of thousands of measurements, possibly accumulated over a period of time, processing becomes a challenge. The amount of data to be processed and the requirements of the applications to be developed may entail a variety of implementation strategies, including *parallel processing* [Lee et al., 2012]. For example, as cars rush into a city in the morning, drivers should receive up-to-date information about space availability in parking lots, even if this involves processing massive amounts of data repeatedly. When efficiency is paramount, it is an additional challenge to develop an orchestrating application that exploits properties about the sensors, optimizes the strategies to collect sensor measurements, and crunches large amounts of data.

Actuating

Processing data may result in taking actions by actuating devices. For example, computing the number of available spaces in parking lots allows to periodically update this number on the entrance screen of each lot. In fact, the actuating process is generally driven by the data processing phase, and should consequently leverage the structure of the preceding phase to ease development. Compared to interacting with sensors via high-level subsets, applications may also need to invoke actuators individually to perform context-specific actions, such as displaying a warning at a specific parking level display when an unsafe level of pollution is detected.

1.2 Software Engineering Challenges

To ensure the success of a software engineering approach, it is crucial to understand the concerns of domain experts, as well as to address the common issues faced by developers throughout the software development process. To achieve this goal, we had the opportunity to participate in a French collaborative project, called *Objects World*[3]. This project aims to build a sustainable ecosystem of stakeholders in the domain of Internet of Things (IoT), revolving around a nationwide, low-bandwidth IoT network. In this section, we present key software engineering challenges that have been identified as a result of numerous interactions with the companies of this consortium.

Overcoming heterogeneous APIs

Functionality-rich smart objects become available from different manufacturers every day. They may rely on widely different technologies, but they frequently share similar sensor and actuator capabilities (*e.g.,* temperature measurement, presence detection, heat regulation). Therefore, it becomes increasingly important to build a common vocabulary for describing APIs of smart objects, specific to an area of interest (*e.g.,* agriculture, healthcare, transport, *etc.*), or even some globally standardized ontology for a domain in general, such as IoT [Baccelli and Raggett, 2015] or Machine-to-Machine (M2M) [Verma et al., 2016]. This ontology should expose a hierarchical structure in order to maximize the reuse of the API descriptions and implementations, as is done with object hierarchy.

Supporting rapid software development

Often, highly-valued services to customers correspond to simple application logic. As an example, consider an orchestrating application that turns off heating when a window is open and the exterior temperature is cold. This service corresponds to a logic that orchestrates the heater, window contact sensors, and a weather web service. Even though the logic of this service is simple, developing an orchestrating application requires programming a great deal of boilerplate code for discovering the relevant devices, listening to events of interest originating from physical devices or web-based APIs, synchronizing, serializing, passing events between application components and actuator objects, and so on. The amount of boilerplate code typically exceeds by far the amount of code for the application logic, and may constitute up to 70% or 80% of the code in the implementation [Cassou et al., 2012]. Consequently, application development time is inflated when the boilerplate code is written manually.

3. Further information on the project can be found in Chapter 7

Facilitating testing

Many orchestrating applications are difficult or too costly to test exhaustively because of the nature of the situations required (*e.g.,* triggering fire alarms) or the difficulty to deploy distant objects (*e.g.,* monitoring ocean water quality). Furthermore, orchestrating applications have to be often developed prior to the deployment of physical sensors and actuators, or even before they are manufactured according to their specifications. Routinely, device drivers simulating actual sensors and actuators are developed to test orchestrating applications before their deployment. When such mockup drivers must be developed manually at a large scale, the development effort can become significant, especially when code for the simulation of devices percolates into the application code.

Supporting rapid evolution

The complexity of developing orchestrating applications is further exacerbated by the fact that new sensors and actuators come out at a very fast pace. As a result, innovative applications orchestrating various smart objects have to be continuously adapted to include new objects or to accommodate API changes for new versions of existing objects. The continuous development and maintenance of these applications when using traditional methods requires extensive software development to cope with the highly competitive market of smart objects. This situation can be mitigated by an approach providing suitably abstract entities that would ease and stimulate code reuse. Furthermore, an approach providing maintenance support is crucial for stakeholders specialized in producing software.

1.3 Contributions

To cope with the many dimensions and the various challenges of large-scale orchestration that we introduced above, we propose a software development approach covering the entire lifecycle of an orchestrating application. This approach makes explicit the domain expertise required to guide the development and deployment of a large-scale orchestrating application by means of a domain-specific design language. To deal with large amounts of sensor data the language provides developers with declarations expressing how data processing occurs while abstracting over intricacies related to high-performance parallel processing. Design declarations are processed and compiled into design-specific programming frameworks that support and guide the programming of the orchestration logic. This strategy allows the developer to abstract over characteristics of the sensor network and to ensure appropriate processing strategies to attain the required performance.

Our contributions revolve around the following themes: (1) software development support, (2) high-performance data processing, (3) sensor-network infrastructures and (4) approach validation. In the following, we describe the main contributions of this dissertation.

Domain-specific design language

The main contribution of this dissertation is a design language dedicated to the domain of orchestrating masses of sensors and actuators. The language provides high-level, declarative constructs allowing developers to (1) declare what an application does and to (2) cope with masses of objects at design time, prior to programming the application. The design activity results in support for the development process of the orchestrating application.

Design-specific programming frameworks

We have developed a compiler for our language that generates high-level programming support, customized with respect to a given application design. This programming support takes the form of a programming framework [Fayad and Schmidt, 1997] that provides guidance to developers, while ensuring that programming is driven by design. For example, the compiler generates code that gathers data from sensors with the declared delivery models, allowing the developer to concentrate on what to do once sensor data is gathered.

High-level parallel processing model

Masses of sensors produce large amounts of data [4] that need to be analyzed efficiently to render high-value services to citizens and operators of smart environments. Our compiler generates programming frameworks that have a carefully structured data and control flow to enable data processing to be implemented efficiently. These frameworks rely on the MapReduce programming model [Lämmel, 2008; Dean and Ghemawat, 2008] to provide the developer with a proven approach to efficiently processing large datasets and enable parallel, distributed implementations to be generated. This strategy allows to cope with large datasets collected from masses of sensors.

Application behavior

Because large-scale infrastructures of networked objects are still emerging, their features are neither standardized, nor stable. Thus, it is vital to ensure that a target infrastructure of networked objects can provide the application with required resources (*i.e.,* sensors and actuators) or to determine whether reconfiguration of the sensor infrastructure is needed to

4. For example, a modern offshore oil production platform comprises around 30,000 sensors and may generate up to 2TB of data per day. [Rigzone, 2014]

match application requirements. To this end, we introduce the notion of *application behavior*, which consists of sensor-network characteristics of a large-scale orchestrating application. As such, the characteristics of an application can be expressed at a high level early in the development process to provide support throughout the application lifecycle. We illustrate how sensor-network characteristics can be expressed via high-level declarations and used throughout the lifecycle of orchestrating applications.

Stages of the application lifecycle

We introduce stages along the application lifecycle where the application behavior declarations can be used to adapt both the application and the infrastructure concerns. This adaptation process ranges from checking that the sensing capabilities required by an application at design time are compatible with the target infrastructure, to submitting an application to an admission control procedure at deployment time.

Implementation

We implemented our approach and applied it to a set of examples. For the software development stage, our approach takes the form of a plugin for the Eclipse IDE [5]. The plugin is publicly available [6] and provides developers with our design language and a code generator. For the data processing stage, our compiler currently produces programming frameworks targeting the Apache Hadoop platform [7].

Evaluation

We evaluate our approach through various experiments.

> ***Scalability.*** We evaluate the implementation of our approach with an experiment that runs application computations over a large dataset of synthetic sensor readings. The experiment demonstrates that programming frameworks generated by our approach exhibit scalable behavior with respect to the size of the input dataset.
>
> ***Effectiveness.*** We report on the effectiveness of our approach to resolve identified software engineering challenges by assessing the support provided by our approach for programming three IoT applications specified by companies from the Objects

5. http://eclipse.org/
6. http://phoenix.inria.fr/software/diaswarm
7. http://hadoop.apache.org/

World project consortium. We show that our approach effectively covers a broad range of IoT applications, and enables expert programmers to prototype applications rapidly.

Usefulness & usability. We evaluate the usability of our approach by soliciting professional programmers from the IoT industry in a usability study. We provide quantitative and qualitative data, including feedback from a usability questionnaire and developer interviews to investigate the perceived usefulness of a design-driven software development approach.

1.4 Outline

The reminder of this dissertation is organised as follows:

- Chapter 2 presents three research domains that are used to investigate how to build systems composed of networked objects, equipped with sensing/actuation capabilities. We discuss the main concerns in each domain with respect to large-scale orchestration challenges presented earlier. We present some existing software development approaches in each domain and examine the support they provide for the development of applications.
- Chapter 3 presents a case study that is used throughout the document to illustrate the contributions of this dissertation. This case study considers the development of a large-scale smart city service for the management of parking spaces in parking lots.
- Chapter 4 introduces our design-driven approach dedicated to the development of large-scale orchestrating applications. We present a domain-specific language dedicated to manipulating objects at a large-scale and demonstrate how application design is compiled into an application-specific programming framework to support and guide the development process.
- In Chapter 5, we extend the design-driven approach to introduce the parallel processing of large amounts of data collected from sensors. We demonstrate how design declarations are used to generate programming frameworks leveraging the MapReduce programming model for efficient processing of sensor data. We evaluate the scalability of our approach in an experiment that runs application computations over a large dataset of sensor readings.
- In Chapter 6, we further explore the design space to determine how design declarations can be used to make explicit the resources required by applications as well as their usage. We introduce stages along the application lifecycle and discuss how declarations can be leveraged at each stage.

- Chapter 7 presents a thorough evaluation of the proposed design-driven approach by means of two different experiments. In a first experiment we examine the effectiveness of the design-driven approach to deal with identified software engineering challenges. In a second experiment, we measure, through a usability study, the cost of learning to use our approach. This study involved professional programmers.
- Chapter 8 details the conclusions of this dissertation and discusses the ongoing and future research avenues.

2

Related Work

Research in different domains is concerned with programming systems composed of physical objects with sensing/actuation capabilities, interconnected through communication networks. In this chapter, we examine the domains of Pervasive Computing, Sensor Networks and the Internet of Things. For each domain, we discuss the main concerns related to programming systems orchestrating physical objects. Furthermore, we examine software development approaches in each domain and discuss the support they provide for the development of large-scale orchestrating applications.

Contents

Overview

- A review of research domains investigating orchestration of networked objects.
- A review of existing approaches dedicated to the development of applications orchestrating networked objects.

This section presents domains where the orchestration of networked objects is a common concern. These domains address different aspects pertaining to the development of orchestrating applications. We discuss software development approaches for each domain and investigate to which extent the support they provide allows to deal with large-scale orchestration challenges introduced in Chapter 1.

2.1 Pervasive & Ubiquitous Computing

Pervasive computing is a domain based on the idea of computers being integrated into the physical world and weaving themselves into the fabric of everyday life until they are indistinguishable from it [Weiser, 1991; Saha and Mukherjee, 2003]. This vision described by Mark Weiser in 1991 has since, to some extent, become a reality. The domain of pervasive computing offers a number of approaches targeting the development of applications orchestrating networked objects. These approaches aim at facilitating the development of software services for smart environments (*e.g.,* offices, buildings) comprising a number of connected heterogeneous devices (*e.g.,* cell phones, sensors, appliances). Applications and software development approaches in this domain commonly revolve around the notion of *context*. In this dissertation, we proceed with following definition of context, taken from the work by Dey *et al.* [Dey, 2001]:

Definition 2.1.1

Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.

Accordingly, a system is *context-aware* if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task [Dey, 2001]. Among the most notable is the work by Dey *et al.* providing foundations for the design and development of context-aware applications [Dey et al., 2001]. In their work, the authors present a conceptual framework to support the development of context-aware applications. They identify basic categories of context and introduce abstractions to encapsulate common context operations. In addition, they discuss the details of the Context Toolkit [1], an implementation of their conceptual framework, which aims at facilitating the development and deployment of context-aware applications. Henricksen *et al.* present an approach and a set of conceptual models to facilitate the development of context-aware applications [Henricksen and Indulska, 2004, 2006]. They define context modelling and programming abstractions to support the development of maintainable and evolvable applications based upon a set of reusable context definitions and processing components.

1. Context Toolkit is available online at http://contexttoolkit.sourceforge.net

The collection, aggregation and dissemination of context has been addressed by Chen *et al.* [Chen and Kotz, 2002]. In their work, the authors present a graph-based abstraction to allow for sharing some of the processing between applications in order to improve flexibility and scalability of context-aware applications. Serral *et al.* [Serral et al., 2010] propose a model-driven approach for the development of context-aware pervasive systems. A domain-specific modeling language called PervML is used to specify the many aspects of a context-aware pervasive system at a high level of abstraction. The language defines a number of conceptual primitives (*e.g.,* service, trigger, interaction, *etc.*) that have to be expressed via UML diagrams and rules in the OCL declarative language [Warmer and Kleppe, 1998]. The approach supports the development phase by translating PervML models into both Java code to provide system services and an OWL [2] specification to infer knowledge from context at runtime. Along this line of work is Olympus [Ranganathan et al., 2005] that provides a programming framework dedicated to the development of pervasive computing systems. The approach introduces the notion of Active Spaces, *i.e.,* physical spaces that comprise sensors, actuators, *etc.* Because it is based on a domain-specific framework, Olympus raises the level of abstraction and facilitates the development of applications. DiaSuite takes these approaches further by introducing a design language dedicated to the Sense/Compute/Control paradigm [Cassou et al., 2011a; Bertran et al., 2012]. A design is used to generate a dedicated programming framework that guides, restricts, and supports the implementation phase. This design-driven approach has been applied to a range of domains involving the orchestration of objects, ranging from Pervasive Computing to Avionics [Enard et al., 2013a,b].

Sehic *et al.* address the problem of programming context-aware applications for large-scale pervasive systems with the Origins programming model [Sehic et al., 2012]. An *origin* is an abstraction of any source of context information. Origins are universal, discoverable, composable, migratable, and replicable components that are associated with type and meta-information. Furthermore, the model supports the creation of processing schemes in context-aware applications via a number of operations, namely filtering, inference, aggregation, and composition.

It must be noted that the above-mentioned list of approaches is not exhaustive. A thorough overview is beyond the scope of this dissertation but can be found in the work of Alegre [Alegre et al., 2016] and Endres [Endres et al., 2005].

2. Web Ontology Language (OWL), http://www.w3.org/2001/sw/wiki/OWL

Summary 2.1

Software development approaches for the domain of pervasive and ubiquitous computing address various recurring issues pertaining to the development process of applications orchestrating networked devices. Different programming models and abstractions are proposed to cope with device heterogeneity and to support context information management. These abstractions facilitate the usual programming tasks, such as context acquisition, modeling, and reasoning. However, the existing approaches have been mostly designed for orchestration of objects in the small (*i.e.,* offices, buildings, homes, *etc.*). Thus, they do not address challenges arising with large-scale infrastructures, such as scalable service discovery or data-intensive processing.

2.2 Sensor Networks

A sensor network can be seen as a system composed of distributed embedded devices comprising modest computational power as well as diverse sensing and possibly actuation capabilities. Compared to conventional distributed systems, sensor networks rely on very small nodes, which are less reliable and mostly battery-powered. In a survey, Sugihara *et al.* explore programming models for sensor networks and present a taxonomy of programming models according to the level of abstraction they provide [Sugihara and Gupta, 2008]. The authors classify approaches to programming sensor networks into low-level and high-level programming models.

Low-level programming models

Low-level programming models focus on abstracting hardware of sensor nodes to ensure their flexible control. A prominent example in this category is the TinyOS operating system [Hill et al., 2000]. Programming support for this platform is provided via nesC, a programming language derived from C, targeting the domain of sensor networks [Gay et al., 2003]. SNACK [Greenstein et al., 2004] builds upon nesC to provide developers with a component composition language and a library to allow for the development of reusable application service libraries and combine them into applications. Some approaches focus on the task of reprogramming sensor nodes. To do so, they use virtual machines to ensure injecting new code into nodes dynamically. In this line of work, Maté [Levis and Culler, 2002] and ASVM [Levis et al., 2005] provide an application-specific virtual machine with a limited number of instructions for a particular application domain. Reprogramming of sensor nodes can be ensured via middleware approaches such as Impala [Liu and Martonosi, 2003] and SensorWare [Boulis et al., 2003].

High-level programming models

High-level programming models take an application-centric view to programming sensor networks. Approaches in this category address the programming of application logic in terms of how nodes in a network collaborate to share, aggregate and process sensed data. As reported by Sugihara *et al.* these approaches can be divided according to the dimension at which collaboration between nodes occurs (*i.e.,* group, network).

Approaches providing group-level abstractions manage sensor nodes as groups defined by their physical distance or some logical properties (*e.g.,* node type). For instance, Welsh *et al.* propose Abstract Regions [Welsh and Mainland, 2004], a family of spatial operators ensuring the communication between sensor nodes within *regions* in different ways, such as topologically or geographically. Apart from physical location, groups can be defined by logical properties, such as the input sensed from the environment, which are more dynamic in nature. For example, EnviroTrack [Abdelzaher et al., 2004] is a middleware layer geared towards the development of applications tracking the physical environment. In this approach, sensors are grouped based on the type of event detected in the environment (*e.g.,* motion). Mottola *et al.* [Mottola and Picco, 2006] propose *logical neighborhoods*, a programming abstraction that defines the notion of proximity according to functionality related characteristics of sensor nodes, including both static and dynamic properties. This programming abstraction is supported by a routing protocol and a language allowing developers to define neighborhoods declaratively.

In contrast to group-level abstractions, network-level abstractions consider the sensor network as one single abstract machine. Network-level abstractions can be further divided into database-oriented and language-oriented approaches, that is, according to how these are designed to support programming of application logic. The typical examples among database-oriented approaches include TinyDB [Madden et al., 2003] and Cougar [Bonnet et al., 2000]. TinyDB is a distributed query processing system for sensor networks addressing when, where and how often data is sampled and delivered. Developers write SQL-like queries, which are optimized, efficiently disseminated into the network and processed by nodes. Similarly, data collection in Cougar is defined via SQL-like queries, which are also leveraged to achieve energy-efficiency. This is done by pushing selection operators to nodes, thus enabling collected data to be reduced locally.

Macroprogramming languages provide an alternative to database-oriented approaches, offering more flexibility and expressiveness to build applications that go beyond data collection. For example, Regiment [Newton et al., 2007] is a functional macroprogramming language for sensor networks allowing programmers to express interest in a group of nodes with some geographic, logical, or topological relationship via *region streams* (*e.g.,* all nodes within k-radio hops of some anchor node). Kairos [Gummadi et al., 2005] is a macroprogramming approach that provides a small set of programming primitives used to (1) read/write variables at nodes, (2) iterate through the one-hop neighbors of a node and to (3) address arbitrary nodes. In Kairos, a dedicated compiler takes a centralized program

to produce a node-specialized version of the compiled program. Some macroprogramming approaches address the problem of naming resources in sensor networks. Along this line of work, Borcea *et al.* [Borcea et al., 2004] propose a programming model allowing resources to be referenced by their physical location and provides access to them via Smart Messages (SM). Similarly, the SpatialViews high-level language [Ni et al., 2005] allows a subset of sensors to be defined as a group and be referenced via properties of interest.

Complementary to the previously presented classification of approaches for programming sensor networks is the work of Mottola and Picco [Mottola and Picco, 2011]. In their survey, authors provide a more in-depth analysis and a taxonomy of programming approaches for wireless sensor networks (WSNs) through a richer set of dimensions. They also identify open research issues related to programming WSNs for which solutions are sorely missing, including tolerance to hardware faults, debugging and testing of applications and the evaluation of effectiveness of programming approaches.

Summary 2.2

Software development approaches for the domain of sensor networks are concerned to a great extent with challenges arising from the resource-constrained nature of the sensor network environment. Typically, energy efficiency is of utmost importance to ensure that sensor nodes remain active over a long period of time considering that these are mostly battery-powered. To do so, programming approaches for sensor networks rely on a number of mechanisms including in-network data aggregation, caching and routing, which are efficient in reducing data transmission, thus lowering power consumption of nodes. Moreover, concerns central to orchestrating sensors in the large, such as organizing sensor nodes into groups of interest or specifying models for collecting data from sensors are addressed by different approaches in this domain. Device heterogeneity as well as strategies for actuating devices are, however, discussed to a lesser extent. Finally, approaches in this domain often necessitate low-level details (*i.e.,* network topology, routing strategies, *etc.*) to be incorporated into application development, which in turn requires expertise in embedded systems and sensor network technology on the part of software developers. In a survey, Mottola and Picco [Mottola and Picco, 2011] also notice that in the domain of wireless sensor networks, developers prefer low-level abstractions to keep every single bit under control. We believe that software development approaches should contribute to raising the level of abstraction and allow non-expert developers to program orchestrating applications.

2.3 Internet of Things

The Internet of Things (IoT) is a novel vision, which in simple terms aims at integrating physical objects (*i.e.,* things) with the virtual world through existing communication networks. Physical objects or *things* are equipped with sensing/actuation capabilities, computational power and are provided with unique identifiers to interact with other things, services and applications over the Internet. The term Internet of Things was coined by Kevin Ashton who used it during his presentation at Procter & Gamble (P&G) in 1999 to promote the idea of linking RFID technology [3] in a supply chain with the then-red-hot topic of the Internet [RFID Journal, 2009]. However, nowadays IoT goes beyond the scope of this initial idea. Indeed, the domain of IoT includes applications providing assistance for elderly or disabled people in their homes [Caroux et al., 2014; Abbate et al., 2012], monitoring parking spaces to optimize the flow of traffic in cities [Worldsensing, 2014; libelium, 2013] and even analyzing soil moisture levels in agriculture [libelium, 2012].

Although IoT is a novel vision with a significant economic potential [Manyika et al., 2015; National Intelligence Council, 2008], at its core, it aggregates efforts undertaken in "traditional" domains including embedded systems [Kortuem et al., 2010], sensor networks [Gluhak et al., 2011], pervasive and ubiquitous computing [Perera et al., 2014], machine-to-machine communication [Wu et al., 2011], cyber-physical systems [Karnouskos, 2011], human-computer interaction [Kranz et al., 2010], *etc.*

The literature on programming applications orchestrating things is less voluminous compared to the domain of sensor networks or pervasive computing and the majority of approaches presented here emerged only recently. Furthermore, they often provide support destined exclusively for specific tasks, such as programming of smart devices, communication management, data analysis, and so forth. The following presentation of approaches in this domain is divided according to where the resulting application logic resides.

We begin by examining approaches where application logic runs inside the network of interconnected smart devices. Sivieri *et al.* propose the ELIoT [Sivieri et al., 2016] platform dedicated to programming smart devices for IoT systems. ELIoT allows programmers to implement functionality running within the local network, while still supporting interactions with Internet-wide services. ELIoT programs are written in a dialect of Erlang that adapts the inter-process communication facilities of Erlang to the specifics of IoT applications, using custom language syntax and semantics. The approach addresses heterogeneity of devices by compiling Erlang code into bytecode, which is interpreted or compiled just-in-time by a virtual machine (VM). Nguyen *et al.* propose a model-driven software development framework for the development of IoT applications called FRASAD [Nguyen et al., 2015]. The approach provides a node-centric, multi-layered software architecture to hide low-level details and to raise the level of abstraction. A rule-based programming model and a domain-specific language are used to describe applications. This approach generates application code from

3. http://www.rfidjournalevents.com/map

initial models through an automatic model transformation process. Riliskis *et al.* propose Ravel [Riliskis et al., 2015], an approach for programming applications across 3-tiers using a distributed Model-View-Controller architecture. In this approach, a sensor network application is programmed as a series of models with views and controllers. The compiler generates static code for each of the tiers including buffering, storage and communication protocols that can be compiled and deployed onto devices at each tier.

Another class of approaches abstracts over specificities of smart devices or the network and provides developers with high-level concepts to build centralized applications. These applications present a central point for gathering data from smart devices. They interact with smart devices via APIs or require specific code to be deployed onto these devices. For example, Patel *et al.* propose a multi-stage, model-driven approach dedicated to the development of IoT applications [Patel and Cassou, 2015; Patel et al., 2013]. This approach provides support at different stages of the development process. At design time, the approach offers a set of customizable modeling languages for specifying an application. The approach is complemented by code generation and task-mapping techniques for deploying of node-level code onto devices. Nastic *et al.* propose PatRICIA [Nastic et al., 2013], a framework for high-level programming and provisioning of IoT applications on cloud platforms. The framework provides high-level programming constructs and operators, which encapsulate domain-specific knowledge and abstract over specificities of low-level device services. These constructs allow programmers to use predefined control and monitor tasks (*e.g.,* controlling physical devices, analyzing sensory data streams) provided in a domain library.

Among other approaches in the domain is the work by Gyrard *et al.* who propose the Machine-to-Machine Measurement (M3) framework [Gyrard et al., 2015] for programming IoT applications. The M3 framework is based on semantic web technologies to explicitly describe the meaning of sensor measurements in a unified way and to ease interpretation of sensor data. The proposed approach builds upon the notion of Semantic Sensor Web introduced by Sheth *et al.* [Sheth et al., 2008], which leverages standardization efforts of the Open Geospatial Consortium (OGC) [4] and Semantic Web Activity of the World Wide Web Consortium (W3C) [5] to provide enhanced descriptions and meaning to sensor data. The M3 framework generates IoT application templates [6] according to the sensors and domains required by the users. To support the development of IoT applications, generated templates comprise M3 domain ontologies, datasets, rules and SPARQL queries [7].

Complementary to the overview above is the literature review on context-aware computing research efforts conducted by Perera *et al.* [Perera et al., 2014]. In their substantial survey, authors analyze, compare and classify most research and commercial solutions (*i.e.,* 50 projects covering a decade) proposed in the field of context-aware computing and investi-

4. http://www.opengeospatial.org
5. http://www.w3.org/2001/sw
6. http://www.sensormeasurement.appspot.com/?p=m3api
7. http://www.w3.org/TR/rdf-sparql-query

gate how techniques in this domain can be applied to solve problems in domains such as the IoT. It is important to note that other contributions in the domain of IoT focus on specific application tasks, such as discovering devices and collecting data [Kolcun et al., 2015; Kolcun and McCann, 2014; Guinard et al., 2010]. Also, middleware [Hachem et al., 2014; Teixeira et al., 2011], architectures [Guinard et al., 2010] and the provisioning of services [Yuriyama and Kushida, 2010; Guinard et al., 2010] for IoT have been widely discussed.

A more detailed insight into open issues, research challenges, enabling technologies, applications, on-going initiatives and standardization activities in the IoT can be found in the work of Al-Fuqaha *et al.* [Al-Fuqaha et al., 2015], Borgia [Borgia, 2014], Stankovic [Stankovic, 2014], Sheng *et al.* [Sheng et al., 2013], Miorandi *et al.* [Miorandi et al., 2012] and Atzori *et al.* [Atzori et al., 2010].

Summary 2.3

Software development approaches for the domain of IoT are presented in the context of small-scale environments, such as homes, offices, administrative buildings, *etc.* These approaches do not make explicit how masses of devices are handled nor do they address issues arising with large-scale smart spaces, such as dealing with huge amounts of data. It seems that this could be a major impediment for the development of applications in this domain, since, as reported by Stankovic [Stankovic, 2014], the amount of raw data that need to be collected and converted into usable knowledge by IoT applications will be enormous. Specific application domains prone to generate large amounts of data within the context of smart cities are also discussed in the work of Hashem *et al.* [Hashem et al., 2016]. We also observe that approaches for programming IoT applications frequently rely on code generation techniques to support programming, however, code examples are often missing or being discussed only briefly. Furthermore, prototypes and tools that would allow to examine these approaches in greater detail are currently unavailable.

3

Case Study

Modern ubiquitous computing systems take the form of wide-area infrastructures, populating a variety of environments with functionality-rich sensors. These infrastructures comprising massive amounts of sensors are increasingly emerging and being deployed over large-scale spaces in smart cities, including campuses of buildings, parking lots, as well as railway lines, agricultural fields and forests in rural areas. These smart environments validate the maturity of large-scale sensor infrastructures for delivering innovative services to citizens. Nowadays, companies worldwide administer such infrastructures to enable economically viable services to be offered.

Contents

Overview

- Presentation of a large-scale smart city service for the management of parking spaces in parking lots.
- Analysis of large-scale orchestration requirements for the parking management system.

The SmartSantander project [Sanchez et al., 2014; libelium, 2013], developed in Spain by stakeholders such as Telefonica I+D [1] and University of Cantabria [2], is typical of the emerging smart environments. This project aims at designing, deploying and validating a citywide infrastructure composed of sensors, actuators, cameras and screens to offer valuable information to citizens. Wireless low-consumption sensors (Waspmotes [3]) have been deployed to monitor parameters such as noise, temperature, luminosity, CO, and parking space presence. In Moscow, SIGFOX [4] deployed a large-scale infrastructure providing the city with the world's largest intelligent parking system, comprising fifteen thousand sensors, which has been operating since November 2013. This infrastructure enables Moscow to reduce traffic in the city center by allowing users to find a parking spot via a mobile app or through electronic street panels. Furthermore, this solution also provides information on most frequent areas to ensure optimal management of urban parking areas [Worldsensing, 2014].

These projects have been an inspiration for our case study described below, which we use to investigate the many aspects of the development process of services orchestrating massive amounts of sensors and actuators, deployed over an entire city. We examine this case study throughout this dissertation as we introduce our contributions and illustrate the salient features of our approach.

3.1 Parking Management Application

This case study examines the development of a parking management application, inspired by existing smart city projects mentioned earlier. The purpose of this application is to monitor the occupancy of parking lots and regulate the flow of traffic to direct cars to available parking spaces. In our scenario, we envision an infrastructure capable of monitoring the availability of parking spaces. Sensors measure magnetic field variations to determine whether a parking space is occupied by a car. They are encapsulated inside a waterproof casing, buried underground, and emit their status at regular intervals. The application gathers values from these sensors and provides drivers with the number of available parking spaces for a given parking lot by displaying this information on a screen at the entrance of the lot. In addition, the application suggests parking lots to drivers entering the city in an attempt to optimize the flow of traffic. In this case, suggestions are being broadcast to drivers via panels located at the entrances to the city. Furthermore, the application processes sensor data acquired over a period of 24 hours to determine the daily average occupancy of a parking lot. Parking managers are kept informed about the occupancy level of a parking lot via messages (*e.g.,* email, text messages).

1. http://www.tid.es
2. http://web.unican.es
3. http://www.libelium.com/products/waspmote
4. http://www.sigfox.com

The application also requires Carbon Monoxide (CO) sensors to detect an unsafe level of pollution in parking lots and alert drivers to this hazard. Pollution alerts are displayed on display panels at the entrance and inside the parking lots. Figure 3.1 presents a graphical representation of the parking management system.

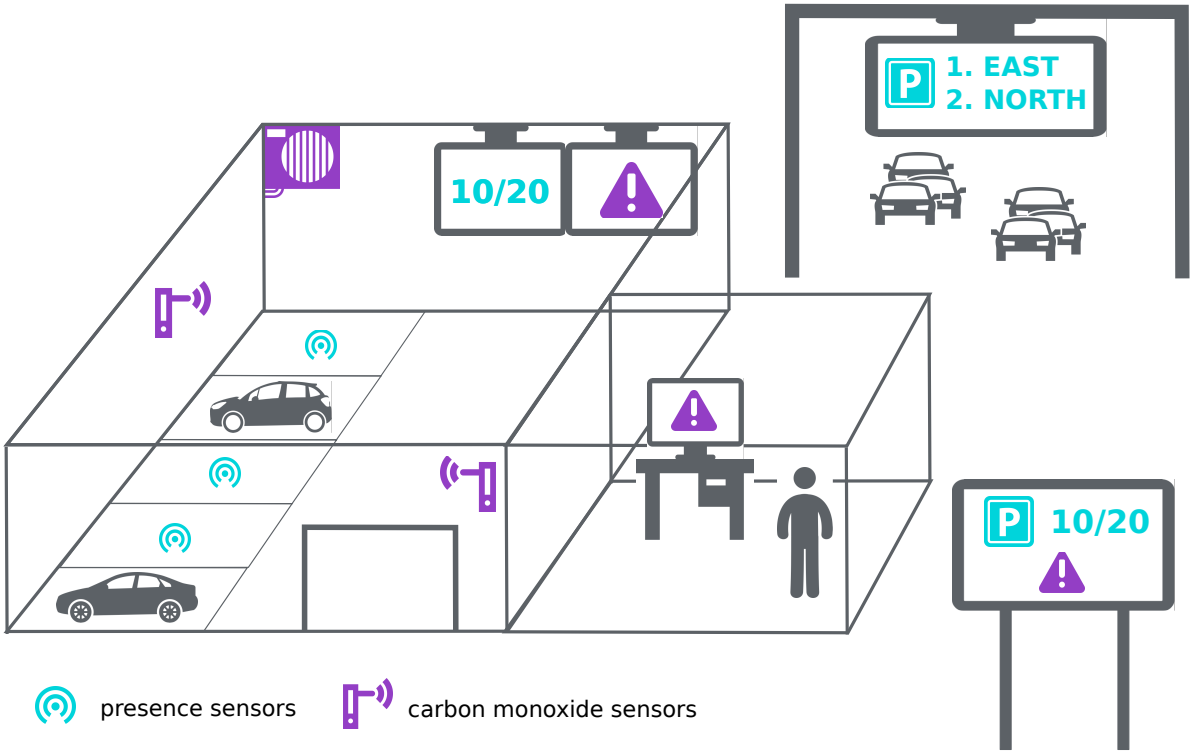

Figure 3.1 – A graphical representation of the parking management system.

3.2 Addressing Large-Scale Orchestration Challenges

This section examines our case study from the perspective of large-scale orchestration and provides an overview of challenges introduced in Chapter 1 for the parking management application. Each challenge consists of a list of application requirements that needs to be addressed during the development process. We consider application requirements progressively from Chapters 4 to 6, as we introduce our approach and the different stages of the development process.

Service discovery requirements

Sensor grouping. Presence sensors and CO sensors need to be grouped by parking levels and parking lots. Indeed, space availability and pollution levels are naturally delivered at both these granularities by a parking management system. Accordingly, the application has to (1) discover high-level objects of interest, such as entire parking levels and parking lots and (2) manipulate sensors via dedicated abstractions.

Data gathering requirements

Presence status delivery. Occupancy of a parking space is published by the associated sensor when the occupancy status changes; this requires the application to implement an event-driven delivery model for gathering data from presence sensors.

CO status delivery. CO sensors deliver their measurements in two ways depending on the needs of the data consumer. For the ventilation of a parking lot, the information is produced every 15 minutes. This time may vary depending on the time it takes to renew the air for a ventilation system and the size of the parking lot. For the purpose of pollution alerts, the application requires the pollution level to be delivered when it reaches a given threshold.

Data processing requirements

Availability computations. The application requires the number of available parking spaces in parking lots to be computed every 10 min. The application also needs to compute the average usage of parking lots from data collected in the last 24 hours.

Actuating requirements

City-entrance display actuation. Parking availability has to be delivered to all displays situated at the level of city entrances. In our case study, we require information displays performing some local processing to disseminate only relevant information such as parking lots near a given city entrance.

Parking-level display actuation. Information displays situated at each level in parking lots need to be invoked to perform a context-specific action, such as display the number of available parking spaces at a given level. We assume that these displays are not capable of local processing and thus need to be actuated with relevant data.

4

A Design-Driven Development Approach

This chapter presents a domain-specific, design-driven software development approach to taming the many dimensions of the orchestration of masses of objects. Generative programming is used to produce design-specific programming frameworks that support and guide the development process, while abstracting over network intricacies. We illustrate our approach using the parking management application presented in Chapter 3 and show how our approach creates synergy between design and programming.

Contents

Contributions

- A software development approach covering all the phases of an orchestrating application.
- A language dedicated to manipulating objects at a large scale.
- A compiler producing programming support customized with respect to a given application design.

4.1 Our Approach

To address the challenges examined in Chapter 1, we propose a software development approach that covers all the phases of an orchestrating application. This approach is based on a domain-specific language (DSL) dedicated to designing orchestrating applications. Design declarations are processed by a compiler to support and guide the programmer using generative programming. This strategy allows (1) to abstract over the characteristics of the sensor network and (2) to ensure that programming is driven by the design. Let us further present the salient features of our approach.

Domain-specific design language

To cope with the many dimensions of the orchestration of masses of sensors, we introduce a design language that is dedicated to this domain, allowing the developer to declare what an application does, prior to programming it. This design language, named DiaSwarm, consists of constructs dedicated to manipulating objects at a large scale. For example, it provides high-level constructs to declare delivery models of sensors at design time. Furthermore, to address the recurring patterns of orchestrating applications, DiaSwarm revolves around the Sense/Compute/Control (SCC) paradigm, promoted by Taylor *et al.* [Taylor et al., 2009].

Design-specific programming frameworks

We have developed a compiler for DiaSwarm that produces programming support customized with respect to a given DiaSwarm design. This programming support takes the form of a programming framework [Fayad and Schmidt, 1997]. For example, the DiaSwarm compiler generates code that gathers data from sensors with declared delivery models, allowing the developer to concentrate on what needs to be done once the data are gathered.

Chapter 4: Outline

In Section 4.2, we introduce DiaSwarm, a design language dedicated to the domain of orchestrating masses of objects. This language provides high-level, declarative constructs that allow a developer to deal with masses of objects at design time, prior to programming the application. The design activity results in support for the development process of the orchestrating application.

Section 4.3 examines how a DiaSwarm design is compiled into a customized programming framework. This programming framework provides high-level support to the developer, while ensuring that programming is driven by the design. We detail how the application logic is programmed against such a framework.

4.2 DiaSwarm

In this section, we introduce the DiaSwarm design language using our case study presented in Chapter 3. Our presentation of DiaSwarm focuses on the aspects pertaining to orchestrating objects in the large. Other aspects are inspired by a design language introduced by Cassou *et al.* [Cassou et al., 2012] named DiaSpec, dedicated to traditional pervasive computing environments (*e.g.,* homes, offices). In Section 4.2.1 we examine how DiaSpec can be used to declare sensors and actuators at design time. DiaSpec provides other specific constructs dedicated to the design of application logic presented in the beginning of Section 4.2.2. Finally, we examine how DiaSwarm extends the DiaSpec language to introduce constructs dedicated to service discovery and data gathering for large-scale infrastructures of objects. Further information on the support generated from DiaSpec declarations targeting the testing, deployment and maintenance of applications is discussed in Chapter 7, Section 7.1.

4.2.1 Device Declarations

An infrastructure relies on numerous objects that allow applications to determine the current state of the environment and to execute actions accordingly. We refer to these elementary building blocks [Kortuem et al., 2010] as *devices*, whether they are hardware (*e.g.,* sensors) or software (*e.g.,* web services). A device declares its ability to sense the state of the environment as a *source*. Also, a device may have an action facet that comprises a set of operations that can alter the current state of the environment. Device properties (*e.g.,* ID, location, *etc.*) allow device instances to be distinguished from each other; they are called *attributes* and have to be defined at deployment time. Finally, device declarations offer inheritance, promoting the reusability of sources, actions and attributes.

Listing 4.1 shows device declarations for the parking management system. Line 1 declares the `PresenceSensor` device, which consists of an attribute (line 2), defining the location of the parking space it is associated with.

```
1 device PresenceSensor {
2   attribute parkingLot as ParkingLotEnum;
3   source presence as Boolean;
4 }
```

Listing 4.1 – Device declaration of a presence sensor device in DiaSwarm.

Contributions presented in this chapter have been published in proceedings of the 14th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE) [Kabáč and Consel, 2015].

This device only declares one source of information (line 3): a boolean value indicating whether a car is present at the parking space associated with a sensor.

In Listing 4.2, two actuators are defined in lines 5 and 9. Each class of actuator defines a location attribute specific to its purpose (*i.e.,* parking lot and city entrance). Both classes of actuators share an operation to display information (`update` – line 2). Likewise, the `Messenger` actuator (line 13) declares an operation (`sendMessage` – line 14) to provide parking managers with information about parking lots.

```
1  device DisplayPanel {
2    update(status as String);
3  }

5  device ParkingEntrancePanel extends DisplayPanel {
6    attribute location as ParkingLotEnum;
7  }

9  device CityEntrancePanel extends DisplayPanel {
10   attribute location as CityEntranceEnum;
11 }

13 device Messenger {
14   sendMessage(message as String);
15 }
```

Listing 4.2 – Device declarations of actuators in DiaSwarm.

For the sake of completeness, declarations of enumerations are displayed in Listing 4.3. In practice, they must be generated automatically, considering the number of parking spaces, parking lots, and display panels involved.

```
1  enumeration ParkingLotEnum {
2    A22, B16, D6,...
3  }

5  enumeration CityEntranceEnum {
6    NORTH_EAST_14Y, SOUTH_EAST_1A,...
7  }
```

Listing 4.3 – Type declarations of enumerations in DiaSwarm.

4.2.2 Application Design

In our target domain, applications can be seen as interacting with an external environment to measure its state via sensors and modify it via actuators. For such a domain, the application logic is naturally expressed with a Sense/Compute/Control (SCC) paradigm, depicted in Figure 4.1. The SCC paradigm, promoted by Taylor *et al.* [Taylor et al., 2009], is general enough for orchestrating objects both in the small and in the large. Consequently, this aspect of DiaSwarm reuses the way DiaSpec declares the design of an application [Cassou et al., 2011a].

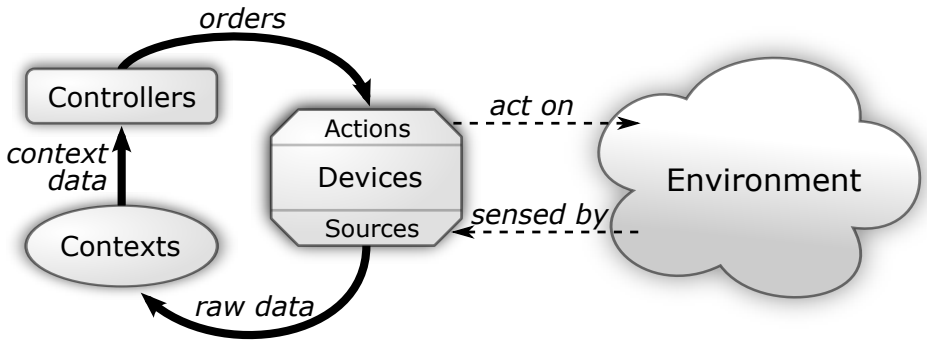

Figure 4.1 – The Sense/Compute/Control paradigm. Illustration adapted from the work by Cassou *et al.* [Cassou et al., 2012]

Specifically, a design consists of (1) declarations of components and devices and (2) descriptions of how they interact with each other, forming an acyclic, directed graph from sensors to actuators. As shown in Figure 4.1, DiaSpec introduces two types of components: *contexts* and *controllers*. We define context components as components that interact with device sources; they receive raw data from the devices, via their sources. They refine (*e.g.*, filter, aggregate) this data into application values, possibly interacting with other context components. When the environment needs to be acted on, a context component declares an interaction with controller components. These components are invoked with refined values and determine what actuators are to be invoked and how.

Figure 4.2 presents a graphical view of the parking management application in the SCC paradigm. The application declares the `PresenceSensor` device, which produces presence values via its `presence` source to the `ParkingAvailability`, `ParkingUsagePattern` and `AverageOccupancy` contexts. The `ParkingAvailability` context computes the number of available parking spaces in parking lots. This information is passed to the `ParkingEntrancePanel` controller; it is in charge of refreshing the number of available spaces. To do so, this controller component invokes the `update` operation of the display panel at the entrance of parking lots.

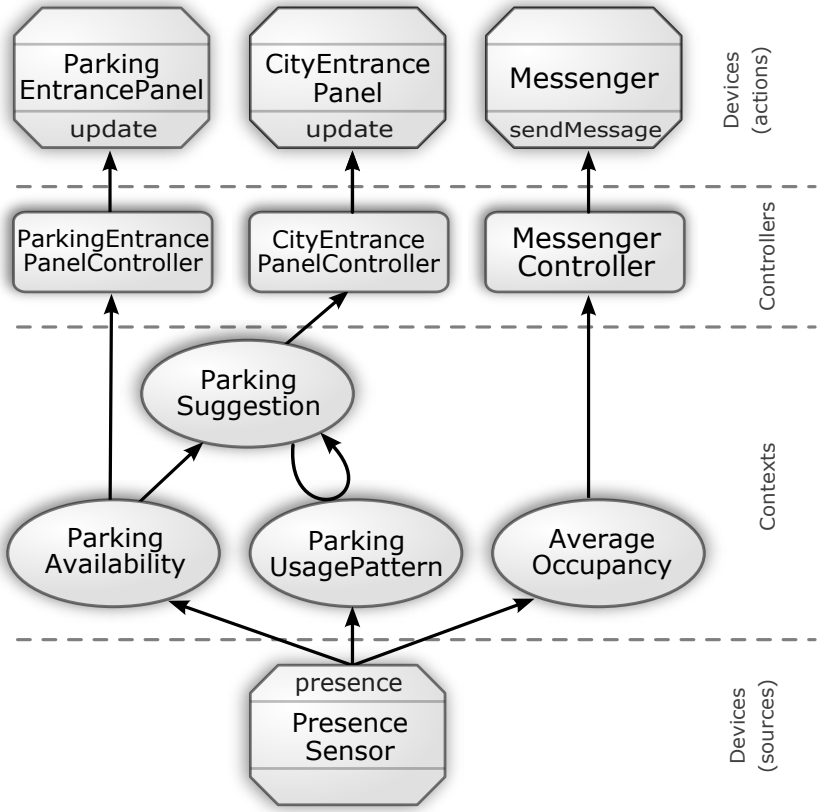

Figure 4.2 – Application design of the parking management application in the SCC paradigm.

The `ParkingSuggestion` context provides a list of suggestions of parking lots, based on the information computed by the `ParkingAvailability` component and the usage statistics of parking lots, accumulated by the `ParkingUsagePattern` component. The list of suggestions is passed to the `CityEntrancePanel` controller that administers display panels located at the entrances of the city. The `AverageOccupancy` context calculates the average occupancy of individual parking lots and passes this information to the `Messenger` controller, which notifies parking managers by sending a message via the `Messenger` device.

As can be seen in Figure 4.2, at a high level, the design of our parking management application does not depend on whether masses of sensors are involved. However, as we examine this application further by presenting the declarations of its constituent components, the need to account for masses of sensors becomes evident, calling for specific constructs. This situation first arises when considering how a context can gather data from a large number of sensors.

Data gathering

We now examine how DiaSwarm extends DiaSpec to address the data gathering challenge. Because of the nature of our domain, context components mostly gather information from a large number of objects. To cope with this dimension, our declarative approach provides three data delivery models, inspired by the domain of wireless sensor networks [Tilak et al., 2002], namely *periodic*, *event-driven* and *query-driven*.

Let us illustrate these three data delivery models with our working example and its DiaSwarm declarations given in Listing 4.4, 4.5, 4.6, 4.7 and 4.8. A context is declared with the keyword `context`, as illustrated in Listing 4.4, line 1 with the declaration of the `ParkingAvailability` context, whose output type is a sequence of values of type `Availability`. Next, line 2 defines how this context component interacts with its input sensor, namely, `PresenceSensor`. Specifically, the data delivery model for this context is defined as `periodic`. Indeed, recall that presence sensors are assumed to send their status periodically. Thus, our declaration specifies that the `ParkingAvailability` context must be activated following a periodic model, every 10 minutes (*i.e.,* `<10 min>` with presence values).

```
1 context ParkingAvailability as Availability[] {
2   when periodic presence from PresenceSensor <10 min>
3   grouped by parkingLot
4   always publish;
5 }
```

Listing 4.4 – Declaration of the `ParkingAvailability` context in DiaSwarm.

However, the application and the myriad of presence sensors are managed independently. This means that values from the presence sensors are gathered at the sensors' pace, and values are pushed to the application at the application's pace, specified by the context declaration. If the context is faster than the sensors, it will be activated with the same values. If it is too slow, it will miss values. This latter case is illustrated by the `ParkingUsagePattern` that collects parking space occupancy every hour (Listing 4.5, line 2), as opposed to every 10 minutes, because usage patterns can be determined from coarser-grained information.

```
1 context ParkingUsagePattern as UsagePattern[] {
2   when periodic presence from PresenceSensor <1 hr>
3   grouped by parkingLot
4   no publish;
5   when required;
6 }
```

Listing 4.5 – Declaration of the `ParkingUsagePattern` context in DiaSwarm.

Finally, the AverageOccupancy context determines the average occupancy of a parking lot by processing sensor data acquired over 24 hours (Listing 4.6, line 3).

```
1 context AverageOccupancy as ParkingOccupancy[] {
2   when periodic presence from PresenceSensor <10 min>
3   grouped by parkingLot every <24 hr>
4   always publish;
5 }
```

Listing 4.6 – Declaration of the AverageOccupancy context in DiaSwarm.

DiaSwarm also offers two other delivery models: *event-driven* and *query-driven*. They are denoted by when provided and when required activation conditions, respectively. Let us present how these models are addressed by DiaSwarm. The ParkingSuggestion context requires data from the ParkingAvailability and ParkingUsagePattern contexts to produce a list of suggestions of parking lots. This list is computed when the ParkingAvailability context outputs a result (see Listing 4.7, line 2). In fact, all components declared as interacting with the ParkingAvailability context will be invoked whenever it produces a value. How the ParkingAvailability context produces values is declared in Listing 4.4, line 4: always publish. This construct specifies that the context must publish an output to subscribed components whenever it is activated (*i.e.,* every 10 minutes). The second input to the ParkingSuggestion context is the ParkingUsagePattern context. The interaction with this context is query-driven, as denoted by the declaration get used in line 3. In fact, the ParkingUsagePattern context never publishes values (see Listing 4.5, line 4). It is assumed that its clients request values from it. This activation condition is expressed by the when required declaration (see Listing 4.5, line 5).

```
1 context ParkingSuggestion as ParkingLotEnum[] {
2   when provided ParkingAvailability
3   get ParkingUsagePattern
4   always publish;
5 }
```

Listing 4.7 – Declaration of the ParkingSuggestion context in DiaSwarm.

For completeness, note that DiaSwarm allows context to conditionally publish values with the maybe publish construct (not used in this example). If a value is not published, the chain of component activations is stopped. Otherwise, the chain of component activations goes one step further towards actuators.

```
1 structure Availability {
2   parkingLot as ParkingLotEnum;
3   count as Integer;
4 }

6 structure UsagePattern {
7   parkingLot as ParkingLotEnum;
8   level as UsagePatternEnum;
9 }

11 structure ParkingOccupancy {
12   parkingLot as ParkingLotEnum;
13   occupancy as Float;
14 }

16 enumeration UsagePatternEnum {
17   HIGH, MODERATE, LOW
18 }
```

Listing 4.8 – Type declarations of the parking management application in DiaSwarm.

Service discovery at design time

Discovering in the large requires high-level constructs that are application-tailored. To achieve this goal, we propose constructs that leverage application-specific design concepts. Specifically, DiaSwarm offers the `grouped by` construct that is parameterized by an attribute. For example, in Listing 4.4, line 3, the `ParkingAvailability` context requires grouping presence statuses in parking spaces by parking lot, enabling availability to be computed for each lot.

Note that in DiaSwarm, service discovery is part of the design phase, contrasting with existing service discovery that are part of the programming phase [Zhu et al., 2005]. This is a key feature to achieve scalability, as discussed later. Furthermore, because our service discovery approach is global (*i.e.,* not specific to individual sensors), it abstracts over sensor failures; this aspect is delegated to an underlying middleware layer.

Data processing

Although high level, the DiaSwarm declarations suggest data processing models. Specifically, an application is reactive and consists of chains of component activations. A chain is executed when its initial activation condition holds (*e.g.,* a sensor publishes), regardless of the delivery model. The execution of a chain ends if one or more actuators are invoked or a component does not publish any value. Additionally, when a component declaration groups values (*e.g.,* `grouped by parkingLot`), it will process a sequence of values, indexed by the

grouping attribute (*i.e.,* `parkingLot`). For example, in the `ParkingAvailability` component, the processing will receive a list of available parking spaces, indexed by parking lot identifiers (*i.e.,* `ParkingLotEnum`). Additionally, this construct allows values to be accumulated over a period of time, as illustrated by the `AverageOccupancy` (Listing 4.6) context. The declaration in line 3 allows presence values, not only to be grouped by `parkingLot`, but also to be accumulated over a 24-hour period (keyword `every`).

Actuating

The declaration of a controller component begins with the `controller` keyword followed by its name. A controller is activated exclusively by the `when provided` condition. For example, the `ParkingEntrancePanel` controller (Listing 4.9, line 1) is activated by the `ParkingAvailability` context (line 2), which causes the `update` action to be triggered on the `ParkingEntrancePanel` device (line 3).

```
1  controller ParkingEntrancePanelController {
2    when provided ParkingAvailability
3    do udpate on ParkingEntrancePanel;
4  }

6  controller CityEntrancePanelController {
7    when provided ParkingSuggestion
8    do update on CityEntrancePanel;
9  }

11 controller MessengerController {
12   when provided AverageOccupancy
13   do sendMessage on Messenger;
14 }
```

Listing 4.9 – Declarations of controller components in DiaSwarm.

4.3 Programming Frameworks

DiaSwarm designs are processed by a compiler that generates customized programming frameworks, currently written in Java. These frameworks provide domain-specific functionalities, including service discovery, data gathering, and component interaction. This approach allows the developer to concentrate on the application logic and abstract over the specificities of the target infrastructure.

To connect the design phase to the programming phase, the DiaSwarm compiler generates an abstract class for each component declaration. The application logic is implemented

by subclassing each abstract class, which in turn requires the abstract methods to be implemented by filling these placeholders with code. This systematic approach provides the developer with a simple interface between design and programming. Furthermore, it allows leveraging integrated development environments, such as Eclipse [1], by assisting the programmer to fill in class templates. In the remainder of this section, we examine the generated programming support for devices, contexts and controllers.

4.3.1 Device Implementation

Listing 4.10 presents a fragment of the abstract class generated from the `PresenceSensor` device declaration. This abstract class provides the developer with ready-to-use getter and setter methods to manipulate device attributes (*e.g.,* `getParkingLot` and `setParkingLot`) and sources (*e.g.,* `setPresence`).

To introduce the `PresenceSensor` device, the developer (1) extends the `AbstractPresenceSensor` class and implements the abstract methods, if any (*e.g.,* actuator operations), and (2) interfaces the device with the generated framework by invoking the `setPresence` method of a device instance, whenever a new measurement is performed. The invocation of this callback method will in turn invoke context components subscribed to this device. The support for devices is examined in greater detail in the work by Cassou *et al.* [Cassou et al., 2012].

```
1  public abstract class AbstractPresenceSensor {
2    private ParkingLotEnum parkingLot;

4    public AbstractPresenceSensor(ParkingLotEnum parkingLot) {
5      setParkingLot(parkingLot);
6    }

8    public ParkingLotEnum getParkingLot() {
9      return parkingLot;
10   }

12   protected void setParkingLot(ParkingLotEnum parkingLot) { ... }
13   protected void setPresence(Boolean presence) { ... }
14 }
```

Listing 4.10 – The abstract class generated from the declaration of the `PresenceSensor` device.

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

4.3.2 Application Logic Implementation

Similarly, the implementation of context and controller components is achieved by subclassing the corresponding generated abstract class. Let us illustrate the implementation of these components with our working example.

Context components

We start by examining the implementation of the `ParkingAvailability` context component, shown in Listing 4.11. The developer extends the generated `AbstractParkingAvailability` class with the `ParkingAvailability` class.

This subclassing requires the developer to implement a callback method (*i.e.,* `onPeriodicPresence`) that receives data gathered from presence sensors, in conformance with the DiaSwarm declaration. Because of the `grouped by` directive, the callback method receives a list of parking spaces indexed by the `parkingLot` attribute. This directive is compiled into a map, which holds entries of the `<ParkingLotEnum, List<Boolean>>` key-value type (line 5), allowing the developer to focus on the data treatment.

This treatment is performed by a `for` loop (line 7) over this map. Each iteration processes the parking spaces of a given parking lot. Each entry holds a list of values, indicating the availability of individual parking spaces in a parking lot.

```
1  public class ParkingAvailability extends AbstractParkingAvailability {

3    @Override
4    protected List<Availability> onPeriodicPresence(
5        Map<ParkingLotEnum, List<Boolean>> presenceByParkingLot) {

6      List<Availability> availabilityList = new ArrayList<Availability>();
7      for(Entry<ParkingLotEnum,List<Boolean>> parkingLot : presenceByParkingLot.entrySet()) {
8        int sum = 0;

10       for (Boolean presence : parkingLot.getValue()) {
11         if (!presence)
12           sum++;
13       }

15       Availability availability = new Availability(parkingLot.getKey(), sum);
16       availabilityList.add(availability);
17     }
18     return availabilityList;
19   }
20 }
```

Listing 4.11 – An implementation of the `ParkingAvailability` context.

In our example, we simply count the number of available parking spaces for each parking lot (line 12). This count is then paired with the parking lot identifier. Our example implementation of `ParkingAvailability` returns a list of counts of available parking spaces, indexed by parking lot identifiers, which matches the type of the component declaration.

As can be noted, our generative approach allows the developer to abstract over how sensed data are gathered. In particular, the `onPeriodicPresence` method can be implemented without knowing the frequency at which sensors emit measurements, and how many sensors are involved.

Controller components

The role of a controller component is to trigger actions on devices to alter the current state of the environment. The controller computes which actions need to be performed using the inputs from context components. Similar to a context, a controller is implemented by subclassing the generated abstract class, as illustrated in Listing 4.12.

The generated abstract class `AbstractParkingEntrancePanelController` ensures that the `ParkingEntrancePanel` controller receives data from the `ParkingAvailability` context in conformance with the design declarations. As a result, the `ParkingEntrancePanel` controller will be notified via the `onParkingAvailability` callback method (line 4) whenever the `ParkingAvailability` context publishes the availability of parking lots.

```
1 public class ParkingEntrancePanelController extends
2                                       AbstractParkingEntrancePanelController {
3   @Override
4   protected void onParkingAvailability(ParkingAvailabilityValue parkingAvailability,
5                                        Discover discover) {

7     for(Availability availability : parkingAvailability.getValue()) {
8       String status = getStatus(availability);

10      discover.parkingEntrancePanels()
11              .whereLocation(availability.getParkingLot())
12              .update(status);
13    }
14  }
15 }
```

Listing 4.12 – An implementation of the `ParkingEntrancePanel` controller.

The `onParkingAvailability` method is implemented by overriding the generated abstract method. The arguments passed to the callback method comprise context data (`parkingAvailability`) and the `discover` object. This object is set by the programming framework according to which actuators (and operations) were declared as interacting with this controller component. As shown in line 11, the `discover` object is used to access the display panel of each parking lot.

To do so, the `discover` object contains a collection of proxies, wrapped inside a composite object, following the composite design pattern [Gamma et al., 1995]. A proxy provides a means to invoke a remote device, without the need to manage distributed systems details. The combination of proxies and the composite design pattern allows the developer to invoke methods on devices in a seamless way. The developer can either invoke a method on all the devices implicitly (line 11) or explicitly, using a loop.

4.3.3 Service Discovery in the Large

Our design-driven approach exposes at a high level which sensors are needed for an application. For example, the design of our parking management system exposes the fact that the application requires all sensors of all the parking lots of the city. Furthermore, the design exposes how often it solicits sensors, via the `periodic` directive. This information is key for the owner of the infrastructure to understand how many resources will be used by an application prior to its deployment.

For example, consider the service discovery, in the interaction contract in Listing 4.4 (line 2 to 4). The context requires sensors to be grouped and discovered per parking lots. When combined with the periodic delivery model (*i.e.,* 10 min.) and the payload message size (*e.g.,* 12 bytes for SIGFOX), these constructs can be used to determine, at design time, the volume of transmitted data with respect to specific parameters (*e.g.,* parking lots) and for different time periods (*e.g.,* 10 min., one day). Also, given that most of the infrastructure resources, such as bandwidth are shared, an admission control process is needed to ensure no degradation of quality of service.

As a byproduct, this assessment of resource usage could be a parameter of the billing model of applications and be beneficial to network operators for network reconfiguration to best fit the traffic and thus avoid congestion [Uthra and Raja, 2012]. These ideas will be further explored in Chapter 6.

Chapter 4: Summary

In this chapter, we have presented DiaSwarm, a design language dedicated to the domain of applications orchestrating masses of sensors. We have introduced domain-specific declarations that express the key aspects of such applications: service discovery, data gathering and actuating. We have illustrated our approach with our case study that exercised the salient features of our language.

We have shown that DiaSwarm declarations can be compiled into programming frameworks customized with respect to a given design. These frameworks guide and support the development of orchestrating applications.

5

Exposing Parallelism through Design

Large-scale orchestrating applications critically rely on the processing of huge amounts of data to analyze situations, inform users, and control devices. To address the challenge, we extended our design-driven approach by integrating parallel processing of large amounts of data. Specifically, we extended DiaSwarm with design declarations used to generate programming frameworks based on the MapReduce programming model. We have developed a prototype of our approach, using Apache Hadoop. We applied it to our case study and obtained significant speedups by parallelizing computations over twelve nodes. In doing so, we demonstrate that our design-driven approach allows to abstract over implementation details, while exposing architectural properties that allow high-performance code to be generated.

Contents

Contributions

- An extension to our approach based on a high-level parallel processing model.
- An implementation of our approach that takes the form of a code generator.
- An experiment that demonstrates the scalable behavior of our approach.

Currently, software development in the domain of large-scale orchestration lacks programming models and methodologies to address key domain-specific challenges. In particular, masses of sensors produce large amounts of data that need to be analyzed efficiently to render high-value services to citizens and operators of smart environments.

When considering tens of thousands of measurements, possibly accumulated over a period of time, processing becomes a critical issue. For example, modern offshore oil production platforms comprise around 30,000 sensors and may generate up to 2TB of data per day [Manyika et al., 2015; Rigzone, 2014].

Such large amounts of data carry valuable information about the state of the environment, users or devices; this information often needs to be discovered by applications in a timely manner to undertake appropriate actions. To do so, applications may rely on *parallel processing* [Lee et al., 2012] and implement specific strategies to deal with large amounts of data efficiently.

For example, as cars rush into a city in the morning, drivers should receive up-to-date information about space availability in parking lots, even if this involves processing massive amounts of data repeatedly. When efficiency is paramount, it is a key challenge to develop an orchestrating application that exploits properties about the sensors, optimizes the strategies to collect sensor measurements, and crunches large amounts of data.

Existing approaches dedicated to big data processing provide limited ways to combine data processing strategies with the application logic. Apache Pig [Apache Software Foundation, 2016b] and Hive [Apache Software Foundation, 2016a] require developers to describe data processing in SQL-like query languages with limited support for user-defined functions. Language libraries, such as FlumeJava [Chambers et al., 2010] allow developers to implement data processing via high-level language abstractions. This approach provides data flow expressions and a set of rich data types to implement data processing. Developers still need to decide when and where data processing occurs, as well as how intermediate computations are combined.

In the case of large-scale orchestration, applications may have to analyze sensor data a number of times using different algorithms, or combine them. These needs put an additional burden on developers since they have to introduce boilerplate code to separate library-specific code from the main application logic, interconnect and coordinate computations, store intermediate results, *etc.*

Contributions presented in this chapter have been published in proceedings of the 13th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC'16) [Kabáč and Consel, 2016].

5.1 Dealing with Large Amounts of Data

To facilitate the processing of large amounts of sensors data, we extend our design-driven approach by integrating parallel processing. The extended approach provides the developer with declarations expressing when and where data processing occurs. The application design then compiles into a programming framework, based on the MapReduce programming model. This framework supports and guides the programming of the orchestration logic, while abstracting over the parallel processing of sensed data.

High-level parallel processing model

Our approach provides the developer with a framework based on the *MapReduce* programming model [Lämmel, 2008; Dean and Ghemawat, 2008]. In doing so, the developer uses a well-proven approach to parallel processing of large datasets. The generated programming frameworks have a carefully structured data and control flow, which enables data processing to be implemented efficiently. We illustrate our approach using our parking management application presented in Chapter 3.

Implementation

We have developed a prototype implementation [1] of our approach, which takes the form of a plugin for the Eclipse IDE [2]. The plugin comprises a code generator, which currently produces programming support for the Apache Hadoop platform [3].

Evaluation

We evaluate our implementation in an experiment that runs application computations over a large dataset of synthetic sensor readings. In doing so, we demonstrate that our design-driven approach allows to abstract over implementation details, while exposing architectural properties used to generate high-performance code for processing large datasets.

1. http://phoenix.inria.fr/software/diaswarm
2. http://eclipse.org/
3. http://hadoop.apache.org/

Chapter 5: Outline

In Section 5.2, we demonstrate how parallel data processing is introduced into the application design to facilitate the processing of large datasets collected from sensor infrastructures. We give details on how design is used to produce programming frameworks that rely on the MapReduce model. In Section 5.3, we evaluate the scalability of our approach by parallelizing computations over a cluster of nodes using the Hadoop framework. Lastly, in Section 5.4 we examine approaches dealing with processing of large datasets, not previously mentioned.

5.2 Exposing Parallelism

The large amount of data collected from sensors calls for efficient processing strategies. We now examine how an application design influences the way data are processed. Based on this study, we propose extensions to DiaSwarm and novel treatments of declarations to generate efficient parallel processing of large-scale datasets. In this chapter, we examine the `ParkingAvailability` and `AverageOccupancy` components highlighted in Figure 5.1 and we address the *availability computations* requirement of our case study (see Chapter 3, Section 3.2). We demonstrate how the processing of large data sets can be introduced into the design to drive programming.

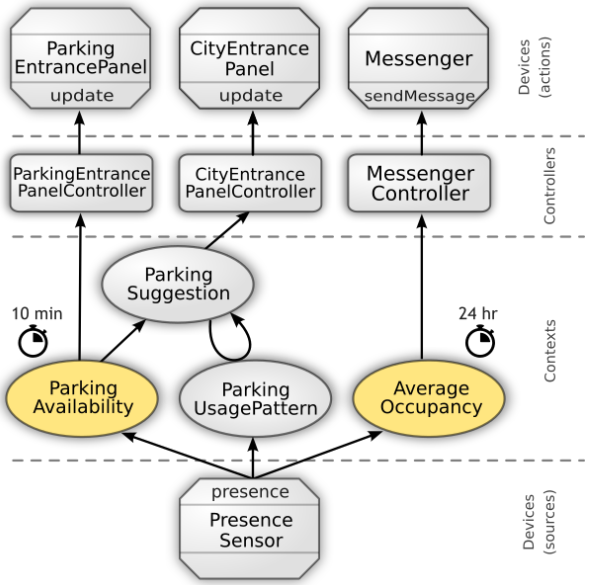

Figure 5.1 – Application design of the parking management application in the SCC paradigm.

5.2.1 MapReduce

Our aim is to put in synergy design and programming by leveraging design declarations to expose parallelism and allow efficient processing strategies to be implemented. An ideal case study is the `grouped by` directive because it partitions a large set of gathered data and exposes a processing strategy that matches the MapReduce programming model. Indeed, this programming model is dedicated to processing large datasets in a massively parallel manner [Lämmel, 2008; Dean and Ghemawat, 2008]. It requires processing to be split into two phases: Map and Reduce. Following our approach, data processing needs to be reflected in the design phase. This is done by extending the `grouped by` directive with an optional clause that specifies what types of values are produced by both the Map and Reduce phases. This is illustrated in Listing 5.1, where the `ParkingAvailability` declaration includes a MapReduce clause that declares the Map phase to produce `Boolean` values and the Reduce phase to produce `Integer` values.

```
1  context ParkingAvailability as Availability[] {
2    when periodic presence from PresenceSensor <10 min>
3    grouped by parkingLot
4    with map as Boolean reduce as Integer
5    always publish;
6  }

8  context AverageOccupancy as ParkingOccupancy[] {
9    when periodic presence from PresenceSensor <10 min>
10   grouped by parkingLot every <24 hr>
11   with map as Presence reduce as Integer
12   always publish;
13 }

15 device PresenceSensor {
16   attribute parkingLot as ParkingLotEnum;
17   source presence as Boolean;
18 }

19 structure Presence {
20   presence as Boolean;
21   time as String;
22 }
```

Listing 5.1 – Excerpt of the parking management application design in DiaSwarm.

The DiaSwarm compiler generates a programming framework that requires the developer to provide an implementation for both the Map and Reduce phases of the data processing. As shown in Listing 5.2, this is done by implementing `map` and `reduce` methods declared in the generated MapReduce interface. In conformance with the MapReduce model, the Map

function is passed a key and a value, which correspond to the parking lot identifier (*i.e.,* the attribute of the `grouped by` directive) and an availability status, provided by the corresponding sensor. The `emitMap` method is invoked to produce each key/value pair result of the Map phase. The framework-generated code groups the results of the Map phase into a list that is then passed to the Reduce phase.

```
1  public class ParkingAvailability extends AbstractParkingAvailability
2                                   implements MapReduce<ParkingLotEnum, Boolean,
3                                                        ParkingLotEnum, Boolean,
4                                                        ParkingLotEnum, Integer> {
5    @Override
6    public void map(ParkingLotEnum parkingLot, Boolean presence,
7                    MapCollector<ParkingLotEnum, Boolean> collector) {
8      if(!presence)
9        collector.emitMap(parkingLot, true);
10   }

12   @Override
13   public void reduce(ParkingLotEnum parkingLot, List<Boolean> values,
14                      ReduceCollector<ParkingLotEnum, Integer> collector) {
15     int sum = 0;
16     for (int i = 0; i < values.size(); i++) {
17       sum++;
18     }

20     collector.emitReduce(parkingLot, sum);
21   }

23   @Override
24   protected List<Availability> onPeriodicPresence(Map<ParkingLotEnum, Integer>
25                                                                presenceByParkingLot) {

26     List<Availability> availabilityList = new ArrayList<Availability>();
27     for(Entry<ParkingLotEnum, Integer> parkingLot : presenceByParkingLot.entrySet()) {
28       Availability availability = new Availability(parkingLot.getKey(),
29                                                    parkingLot.getValue());
30       availabilityList.add(availability);
31     }

33     return availabilityList;
34   }
35 }
```

Listing 5.2 – An implementation of the `ParkingAvailability` context using the generated framework.

This phase sums up the set of values associated with a given intermediate key and, subsequently, emits the availability of a parking lot (`emitReduce`). The data resulting from the MapReduce computation are presented to the developer in the form of a map (line 25). The `onPeriodicPresence` method (line 24 to 34) wraps data resulting from the MapReduce process into the `availabilityList` sequence (line 30), which is returned to subscribed components (*i.e.,* `ParkingEntrancePanelController`, `ParkingSuggestion`).

Although our example involves simple processing, in practice, our design-driven generative approach reduces programming efforts by automatically generating application-specific MapReduce programming frameworks. Furthermore, the generated code keeps the development process straightforward since it prevents specificities of the MapReduce implementation (job scheduling/configuration/execution, distributed file system, APIs, *etc.*) to percolate into the application logic.

5.2.2 Integrating Hadoop

In this section, we show how generative programming is used to produce support for combining an orchestrating application with an actual implementation of MapReduce, namely Hadoop.

Apache Hadoop is an open source implementation of the MapReduce model, which has gained increasing attention over the last years and is currently being used by a number of companies, including IBM, LinkedIn, Facebook and Google [Apache, 2015]. Our compiler generates a MapReduce program that relies on the Hadoop framework. This MapReduce program defines default configuration parameters that enable a job to be executed in Hadoop. Let us illustrate how this is achieved, by examining the code automatically generated for the `ParkingAvailability` context, shown in Figure 5.2.

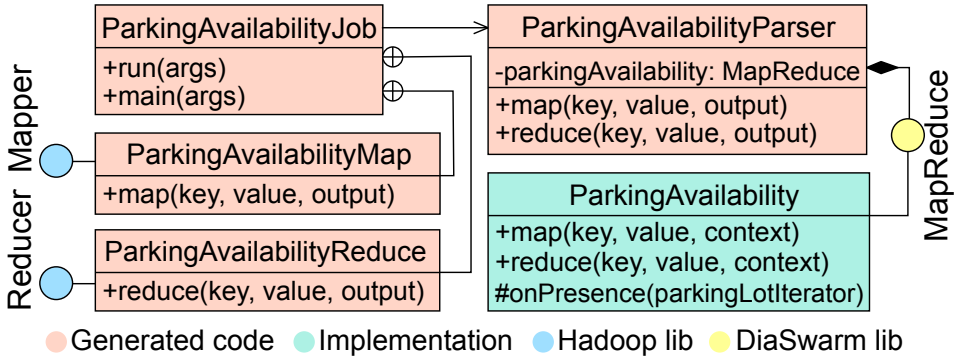

Figure 5.2 – The generated support for integrating Apache Hadoop.

The `ParkingAvailabilityJob` class defines a Hadoop MapReduce program, which comprises the definition of both the `map` and `reduce` methods along with code related to the job

configuration and execution. Both the Map function and the Reduce function are implemented by overriding the `map` and `reduce` methods of the respective `Mapper` and `Reducer` interfaces.

Typically, when using the Hadoop MapReduce library, the definition of the `map` and `reduce` methods resides in the MapReduce program. In this case, however, the implementation of these operations has already been provided by the developer in the `ParkingAvailability` class. The MapReduce program invokes the user-defined `map` and `reduce` methods via the `ParkingAvailabilityParser` class, which keeps an instance of the `ParkingAvailability` context. `ParkingAvailabilityParser` interprets input data of the MapReduce program as corresponding DiaSwarm types and invokes the required `map`/`reduce` method. Consequently, results from the user-defined `map`/`reduce` method are translated to the MapReduce program and submitted via its output collector.

Listing 5.3 shows the `ParkingAvailabilityJob` class, which defines the MapReduce program for the `ParkingAvailability` context. The compiler generates a minimal MapReduce program for every context declared as MapReduce at design time. The type of input data for a generated MapReduce program is defined by the input format, which defaults to `TextInputFormat` (line 26). In our approach, sensor data is stored in the JSON format. In our case study, each presence status delivered to the application is converted to JSON and occupies precisely one line in the resulting dataset. Furthermore, each presence entry is defined by the timestamp of the event, device attributes (*i.e.,* id, parking lot) and the presence source. `TextInputFormat` fits such usage since it splits the input dataset to provide the Map function with one line of text (*i.e.,* one JSON entry) at a time.

In a MapReduce program, any key or value type implements the `Writable` interface, which allows Hadoop to serialize objects for transmission over the network [White, 2012]. To facilitate the development of MapReduce programs, Hadoop already provides `Writable` wrapper classes for the majority of Java primitives (*e.g.,* `boolean` → `BooleanWritable`). In addition, developers may provide custom datatypes by defining classes implementing the `Writable` interface.

At this stage, design declarations are of great importance since they allow the compiler to interpret key and value types of the resulting MapReduce program. For instance, as shown in Listing 5.1, the `ParkingAvailability` context declares the output value type of the Map function as `Boolean` (line 4). As a result, the compiler matches the `Boolean` data type with the corresponding `BooleanWritable` wrapper class (Listing 5.3, line 8). Moreover, an enumeration is interpreted as a string and matched with the `Text` wrapper class (Listing 5.3, line 8). Finally, design declarations using complex data types result in the generation of a custom wrapper class, which implements the `Writable` interface and reflects the entire structure of the datatype.

The execution of a MapReduce program depends upon the data delivery model underlying the interaction between sensors (devices) and the application logic (contexts). In our case study, the ParkingAvailability context declares that data must be gathered from presence sensors in a 10-minute time window according to a periodic delivery model (Listing 5.1, line 2). Data processing takes place when the time window elapses; that is, every 10 minutes, for our case study. At runtime, this job is executed with respect to the gathered sensed data and produces a result. The orchestrating application recovers the result, which is passed to the context via its callback method (*e.g.,* onPeriodicPresence for ParkingAvailability).

```
1  public class ParkingAvailabilityJob extends Configured implements Tool {

3    public static class ParkingAvailabilityMap extends MapReduceBase
4                                              implements Mapper<LongWritable, Text,
5                                                                 Text, BooleanWritable> {
6      @Override
7      public void map(LongWritable key, Text value,
8                      OutputCollector<Text, BooleanWritable> output, Reporter reporter) {
9        jobLauncher.doMap(key, value, output);
10     }
11   }

13   public static class ParkingAvailabilityReduce extends MapReduceBase
14                                                 implements Reducer<Text, BooleanWritable,
15                                                                    Text, IntWritable> {
16     @Override
17     public void reduce(Text key, Iterator<BooleanWritable> values,
18                        OutputCollector<Text, IntWritable> output, Reporter reporter) {
19       jobLauncher.doReduce(key, values, output);
20     }
21   }

23   @Override
24   public int run(String[] args) {
25     JobConf conf = new JobConf(getConf(), ParkingAvailabilityJob.class);
26     conf.setInputFormat(TextInputFormat.class);
27
28     // Remaining configuration
29   }
30 }
```

Listing 5.3 – An example of the generated Hadoop MapReduce program for the ParkingAvailability context.

5.2.3 Alternative Data Processing Methods

Nowadays, the field of Big Data is attracting much attention from research and industry. The tool-development efforts devoted to dealing with rapidly emerging sources of big data result in an abundance of open-source projects. [4]

Apache Hadoop is a widely-used tool to deal with large-scale datasets because it provides a reliable and scalable solution, maintained by a large community of developers. Hadoop is a batch-processing tool, typically used to analyze log files of large-scale systems, collected over a long period of time. The order of magnitude of these datasets may range from hundreds of gigabytes to terabytes and, possibly petabytes.

Apache Spark [5] is an alternative large-scale, data processing tool, which is gaining popularity due to its promise to outperform Hadoop by 10 times [Zaharia et al., 2010]. Spark is an in-memory, data processing framework, which builds upon fault tolerant abstractions, manipulated using a rich set of operators, called Resilient Distributed Datasets (RDDs) [Zaharia et al., 2012].

In contrast with batch-processing tools, Apache Storm [6] primarily targets the processing of unbounded streams of data. Storm is an example of a Complex Event Processing (CEP) [Cugola and Margara, 2012] system, where data flow through a network of transformation entities. An application topology forms a directed acyclic graph, where stream sources (spouts) flow data to sinks (bolts); it implements a single transformation on the provided stream.

In the context of large-scale orchestration, the power of batch-processing tools can be leveraged to analyze long-term datasets for trends in the usage of the city's infrastructure (*e.g.,* parking lots) and to identify structural degradation (*e.g.,* buildings, bridges). Stream processing tools, on the other hand, are best-suited to deal with high-frequency sensor readings, which typically involve tracking applications (*e.g.,* vehicle position, parking place availability). In the future, we intend to extend the parallel data-processing compiler to integrate both Spark and Storm, allowing developers to choose the right tool for their project.

4. http://projects.apache.org/projects.html?category#big-data
5. http://spark.apache.org
6. http://storm.apache.org

5.3 Experimental Evaluation

To assess our approach, we conducted a series of tests to examine the overall behavior of the generated programming frameworks based on the MapReduce model for processing large amounts of sensor data. To do so, we developed a prototype of the parking management system, with Hadoop as the target platform, and analyzed the scalability of our approach using various datasets. In addition, we evaluated the design of the application and observed how specific design choices may impact the overall performance of an orchestrating application.

5.3.1 Experimental Setup

The experimentation focuses on the average parking occupancy feature of our case study. The `AverageOccupancy` context processes sensor data acquired over a 24-hour period, calculates the average occupancy of a parking lot, and notifies the parking manager via a `Messenger` device.

Machines

The experiment was carried out on a cluster of 12 nodes running within a private Eucalyptus [7] cloud. Each node in the cloud corresponds to a m2.xlarge [8] type virtual machine instance with 2 CPUs, 2GB of RAM and 10GB of disk space. Every instance ran the DataStax Enterprise 4.6.1 [9] image, which is a big data platform leveraging tools such as Apache Hadoop and Apache Spark.

Datasets

We generated synthetic datasets to simulate a city's sensor infrastructure for the parking management system. Each dataset contains sensor data, indicating parking space occupancy, which is emitted every 10 minutes over 24 hours (*i.e.,* 144 measurements per sensor). We generated datasets for different sensor infrastructures, ranging from 10 000 to 200 000 sensors per dataset, thus testing the MapReduce program with datasets including up to 28 800 000 input records.

7. http://www.eucalyptus.com
8. http://docs.hpcloud.com/eucalyptus/4.2.2/#user-guide/vm_types.html
9. http://www.datastax.com

5.3.2 Experimental Results

Scalability

Figure 5.3 shows the performance of our parking management program. We compare its execution time with respect to 3 cluster setups – one, six and twelve nodes – and an increasing input dataset size.

As can be expected, the execution time of the one-node setup increases the fastest, compared to the six and twelve node setups. The six and twelve node setups perform at par for the smallest dataset sizes (from 10 000 to 50 000 sensors) because their computing power is under-used. As the size of the datasets increases, the performance of these two setups gradually separate, showing better performance for the twelve-node setup.

These preliminary results show that our compiler generates MapReduce implementations that attain expected scalability. Furthermore, these results demonstrate that declarations at the design level can benefit performance by driving compilation strategies, such as parallelization in our case study. This is achieved by introducing high-level insights (MapReduce constructs) in DiaSwarm.

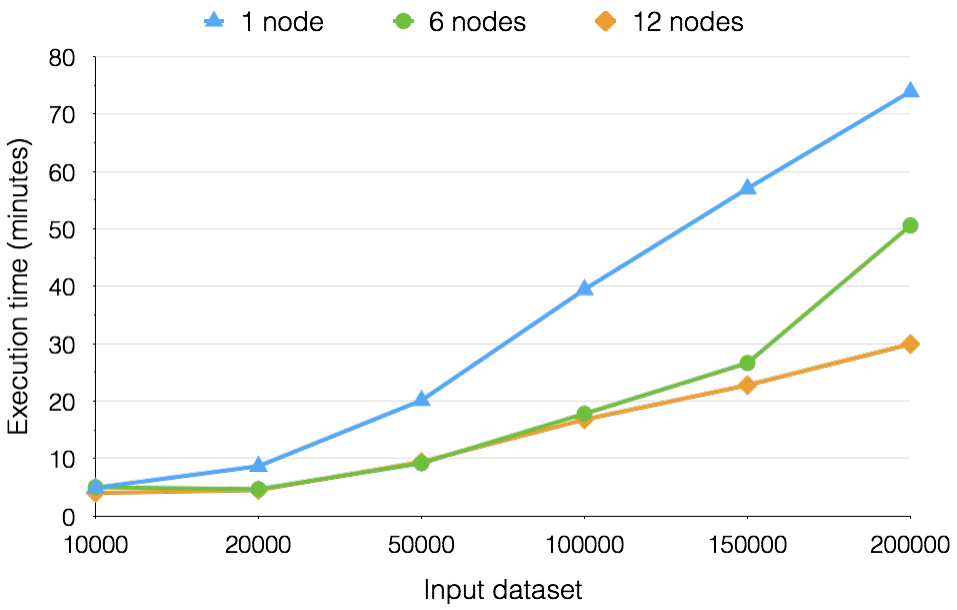

Figure 5.3 – Performance comparison between different cluster setups.

Optimization through design

Beyond significantly improving the execution time of an orchestrating application, Hadoop opens up further optimization opportunities at the design level. For instance, in our case study, the `AverageOccupancy` context processes a dataset of presence values to produce the average occupancy of each parking lot for the last 24 hours. A closer look at the application design reveals that the computation provided by the `AverageOccupancy` context could be achieved by leveraging the computation of the `ParkingAvailability` context. The computed availability of parking spaces could thus be provided to the `AverageOccupancy` context at regular intervals, defined by the data delivery contract (*i.e.,* <10 min>) of the `ParkingAvailability` context. As a result, the `AverageOccupancy` context would use the provided data to calculate an average over the period of 24 hours.

The suggested design adjustments are depicted in Listing 5.4. As can be noticed, the design of the application remains straightforward. More importantly, this design prevents sensor readings from being processed multiple times: the `AverageOccupancy` context factorizes the computations performed by the `ParkingAvailability` context. This caching strategy reduces the total time and resources the application requires for data processing. In fact, as shown in Listing 5.4, the computation performed by the `AverageOccupancy` context no longer involves processing of a large dataset on a cluster (hence the MapReduce clause is omitted).

This is a major optimization that has a direct impact on application upkeep costs, since nowadays companies delegate processing of large datasets to cloud computing platforms (*e.g.,* Amazon Web Services [10]) with a time-of-use pricing model.

```
1 context ParkingAvailability as Availability[] {
2   when periodic presence from PresenceSensor <10 min>
3   grouped by parkingLot
4   with map as Boolean reduce as Integer
5   always publish;
6 }

8 context AverageOccupancy as ParkingOccupancy[] {
9   when provided ParkingAvailability // replaces line 9, Fig. 5.1
10  grouped every <24 hr> //replaces line 10, Fig.5.1
11  always publish;
12 }
```

LISTING 5.4 – The `ParkingAvailability` context factorizing the computation performed by `AverageOccupancy`.

10. http://aws.amazon.com

5.4 Related Work

In this section, we review approaches from the domain of wireless sensor networks that address the parallel processing of data arising from sensors. Furthermore, we highlight the differences between our approach and large-scale data processing support.

Wireless sensor networks

Gupta *et al.* propose sMapReduce [Gupta et al., 2011], a programming pattern inspired by the MapReduce programming model for mapping application behavior onto a sensor network and enabling complex data aggregation. sMapReduce divides the network-level user program into sMap and Reduce functions; this strategy respectively associates a behavior to sensor nodes and executes data aggregation over the network. Compared to our approach, sMapReduce remains lower-level since it provides network-level programming abstractions and introduces the network topology in computations.

Often, programming applications for WSNs is done at a low level, requiring the developer to have extensive knowledge about the underlying layers (network, hardware, OS). Mottola and Picco [Mottola and Picco, 2011] surveyed a number of programming approaches for WSNs aimed to facilitate the programming of layers underlying applications; these approaches target sensor nodes, communication operations, routing strategies, *etc.* These works are complementary to ours in that they provide high-level abstractions that can be used by our compiler to target frameworks for WSNs. However, they do not provide support dedicated to dealing with large datasets produced from massive-scale sensor infrastructures.

Large-scale data processing

Apache Pig [Apache Software Foundation, 2016b] and Apache Hive [Apache Software Foundation, 2016a] are widely used as high-level platforms for analyzing large-scale datasets. These platforms provide SQL-like declarative query languages (*i.e.,* PigLatin & HiveQL) to express data analysis programs. These tools are well-suited for offline data analysis, but require some effort for running scripts from application code (*e.g.,* setting up a connection with a JDBC server).

Sawzall [Pike et al., 2005] used by Google is a high-level scripting language for automating analyses on large data sets on top of the MapReduce execution model. Sawzall is not publicly available but is reported to improve the programming significantly, compared to the C++ programming of MapReduce.

High-level language libraries, such as FlumeJava [Chambers et al., 2010], provide high-level abstractions dedicated to parallel processing; they provide support for user-defined functions, compared to SQL-like approaches.

Compared to the above-mentioned supports, our approach integrates, at the design level, two domain-specific fundamental dimensions: large-scale orchestration of sensors and

large-scale data processing. The integrated nature of our approach allows developers to easily combine results from various computations. The design-driven nature of our approach is supported by high-level declarations, exposing such domain-specific information as service discovery and data delivery. Declarations are analyzed to determine data and control flow information, which in turn, is used to generate efficient, parallel-data processing frameworks.

Chapter 5: Summary

We have proposed a design-driven approach to developing orchestrating applications for masses of sensors that integrates parallel processing of large amounts of sensed data. Our new approach provides the developer with design declarations expressing when and where data processing occurs. A compiler takes an application design as input and produces a programming framework based on the MapReduce programming model. The generated framework supports and guides the programming of the orchestration logic, while abstracting over the parallel processing of sensed data.

We have demonstrated that our approach creates synergy between design and programming, allowing seamless introduction of high-performance computing strategies, as illustrated by the MapReduce programming model. We illustrated our approach with a case study of a parking management system. This case study was used to conduct an experiment on Apache Hadoop, demonstrating how our design-driven approach can be leveraged to parallelize the processing of large datasets and obtain significant speedups.

In the future, we intend to support the processing of unbounded streams of data, typical of sensors. Our declarative approach will allow us to design orchestrating applications that mix the processing of both large datasets and unbounded data streams, allowing us to abstract away these aspects.

6

Leveraging Declarations over the Application Lifecycle

The development of applications orchestrating massive amounts of sensors is difficult due to the inherent resource-constrained nature of sensor network infrastructures. Therefore, it is important to determine the requirements of an orchestrating application on the sensor network early on in the development process and ideally preserve such information throughout the application lifecycle to ensure the required quality of service. Towards this goal, we generalize the DiaSwarm language to further explore the design space and to identify how design can be used to make explicit the resources required by applications as well as their usage. Design takes the form of declarations, which are considered at different stages of the application lifecycle.

Contents

Contributions

- Sensor-network characteristics of a large-scale orchestrating application.
- Stages along the application lifecycle, where the behavior declarations of an application can be used to adapt both the application and the infrastructure concerns.

Just like mobile application development, large-scale sensor infrastructures need to provide programmers with methodologies and tools to support the development of services. This work is essential to allow innovative services to be developed, leading to the adoption of such infrastructures. For example, the Android platform comes with a software framework that manages the lifecycle of applications, provides and controls access to device resources, and allows data sharing between applications. In addition, developers need to make explicit, via a manifest file, a number of application properties (*i.e.,* application components, permissions, *etc.*). This manifest provides a high-level view of an application, which is leveraged by the Android device owner, as well as the operator of the Android infrastructure (*i.e.,* Google), to reduce the risk of abusing resources (*e.g.,* privacy, battery, network) when running this application.

Similarly, for large-scale sensor infrastructures, an application would need to expose, prior to runtime, how it may use resources. Specifically,

- *What are the required sensors/actuators?*
- *How does it gather/receive data from sensors?*
- *When are sensor readings delivered?*

Such issues need to be examined at various stages of an application lifecycle, from design to runtime, and may raise such infrastructure concerns as determining whether the sensor infrastructure can provide the application with the required resources (*i.e.,* admission control) or whether the infrastructure needs to be configured to factorize sensor readings (*i.e.,* caching mechanism). Domain expertise is needed to examine infrastructure concerns along the lifecycle of the application. Currently, this expertise is implicit, making the few existing experts a bottleneck for producing new services that take into account the infrastructure concerns. As a result, not only are sensor-network characteristics of an application missing to support its development, but they are also missing to match the infrastructure to the application needs.

6.1 Addressing Infrastructure Concerns

We propose an approach that makes explicit the domain expertise required to guide the development and deployment of a large-scale orchestrating application. To do so, based on the literature and practical insights, we introduce the notion of *application behavior*, which characterizes how an application behaves to orchestrate sensors at a large scale along three dimensions: service discovery – what sensors are required, data delivery – how and when

Contributions presented in this chapter have been published in proceedings of the 13th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC'16) [Kabáč et al., 2016].

data are delivered to the application, and actuating process – what actuators are issued orders by the application. These behavior dimensions are then instantiated across the stages of the application lifecycle and examined with respect to sensor infrastructure concerns. This process ranges from checking that the sensing capabilities required by an application at design time are compatible with the target infrastructure, to submitting an application to an admission control procedure at deployment time. Our contributions can be summarized as follows.

Application behavior

We identify the sensor-network characteristics of a large-scale orchestrating application. As such, the characteristics of an application can be expressed at a high level and early in the development process, providing support throughout the application lifecycle. To do so, we present a simple declaration language that allows to express the key sensor-network behavior of an orchestrating application. We show how declarations make explicit essential information about the application.

Application lifecycle

We introduce stages, along the application lifecycle, where the behavior declarations of an application can be used to adapt both the application and the infrastructure concerns. We illustrate each stage with our case study presented in Chapter 3, namely, a citywide parking management system. This case study illustrates all the aspects of our proposed approach and demonstrates how it addresses the sensor-network behavior of an application.

We build on practical experience gained from (1) working with operators of sensor-network infrastructures [Kabáč et al., 2015] and (2) previous implementations of case studies leveraging these infrastructures [Kabáč and Consel, 2015, 2016]. As such, our approach provides a design framework for researchers working on methodologies and tools supporting the development of applications for large-scale sensor infrastructures.

Chapter 6: Outline

Section 6.2 decomposes our notion of application behavior into a set of key sensor-network dimensions, drawn from the literature on sensor networks. These dimensions give a design framework for a declaration language dedicated to the sensor-network behavior of applications. This language is introduced in Section 6.3, as well as the main stages of an application lifecycle. Section 6.4 discusses how our approach can leverage existing approaches addressing infrastructure concerns. Related works are covered in Section 6.5.

6.2 Application Behavior Dimensions

A sensor network is an environment constrained in many ways (bandwidth, energy, computational power, *etc.*). When it serves a resource-intensive application or resource-competing applications, their usage profile needs to be determined to ensure quality of service. Towards addressing this issue, we introduce the notion of *application behavior* dedicated to sensor-network dimensions. In this section, we build on the literature on sensor networks and decompose the notion of application behavior into three key dimensions: service discovery, data delivery and actuating.

6.2.1 Service Discovery

Service discovery for large-scale orchestrating applications poses unique challenges due to the resource-constrained nature of sensor networks. Service discovery defines the sensor nodes of interest, the *sources*, and the applications consuming sensor data, the *sinks*. The more accurately sources are selected by applications, the less communication occurs between sources and sinks. This strategy is of utmost importance in the context of a bandwidth-poor environment, comprising masses of sensors.

Meshkova *et al.* [Meshkova et al., 2008] notice that sensor nodes with limited computational resources are not suited for computational and memory hungry service discovery protocols. Furthermore, most well-known protocols, such as UPnP or SLP, are too large to be processed by a sensor network. A promising approach to service discovery, proposed by Heidemann [Heidemann et al., 2001], is to organize sensor-network communications with respect to the application *attributes*, rather than with respect to the network topology.

Estrin *et al.* [Estrin et al., 1999] notice that this application-specific approach is aligned with common application scenarios in the domain. That is, it relies on data generated from the sensor network infrastructure, rather than from individual sensors. Concretely, applications are more likely to ask: *"Where are the nodes whose temperature recently exceeded 30 degrees?"* than *"What is the temperature at sensor #27?"*.

Heidemann *et al.* [Heidemann et al., 2001] also demonstrate the benefits of an attribute-based naming approach, when driven by application-specific requirements. In particular, they show that this approach significantly reduces network traffic.

Based on these works, we conclude that the service discovery dimension should be made explicit and contribute to the requirements of an application, early in the development process. This service discovery dimension should consist of attributes (*e.g.,* sensor types and locations), exposing information to optimize the target sensor network (*e.g.,* network traffic).

6.2.2 Data Delivery

Beyond the service discovery dimension, an application behavior is also characterized with respect to how sensor data are delivered. Tilak *et al.* [Tilak et al., 2002] propose a classification that introduces fundamental delivery models: continuous, event-driven, observer-initiated and hybrid. Importantly, this classification is defined with respect to the observer's interest (*e.g.,* the application).

The data delivery model of an application needs to be explicit to avoid mismatches between application requirements and the target sensor network infrastructure. For example, an application may require data to be delivered at a certain frequency. However, this requirement may not be fulfilled because the target infrastructure does not provide enough bandwidth to transmit the data. Similarly, an application may need to access sensors in a query-driven fashion, which may not be supported by the target infrastructure.

It is important to notice that the data delivery models required by applications also suggest a network structure and specific algorithms that best match the applications' needs, especially in the context of a resource-poor environment. For instance, in her Ph.D. thesis, Heinzelman showed that clustering is most efficient for static networks where data is continuously transmitted [Heinzelman, 2000].

The number of sources and sinks used by an application is also valuable information for choosing an appropriate communication strategy [Heidemann et al., 2003]. Liu *et al.* [Liu et al., 2007] introduce a communication pattern for large-scale sensor networks that adapts the communication strategy with respect to the relative frequency between application queries and detected events from sensors. Furthermore, because dissemination protocols for large-scale sensor networks are data-centric, they can exploit application semantics to improve performance [Ye et al., 2004]. Communication costs can be reduced by introducing some computation inside the network; this is referred to as in-network data aggregation and processing [Heidemann et al., 2001]. For example, information on how data is to be presented to an application (*e.g.,* parking spaces via parking lots) can be used by the sensor network infrastructure to perform in-network data aggregation per node cluster (*i.e.,* parking lots). Estrin *et al.* carry out this idea by using intermediate nodes to perform application-specific data aggregation and caching [Estrin et al., 1999]. *Localized algorithms* are introduced in the context of distributed computations to allow sensors to only communicate with neighbor sensor nodes. As the number of nodes increases, localized clustering can contribute to more scalable behavior, since communication between nodes is kept within a neighborhood.

To summarize, researchers have shown that application-specific information is critical to optimize sensor networks at various levels. This application-specific information mainly consists of knowing what sensors are used by an application and how sensor data are to be delivered. That is, service discovery and data delivery.

6.2.3 Actuating

When orchestrating devices in the large, the nature of actuators introduces some differences in the way an application uses them, compared to sensors. In our experience, we identify two approaches to issuing orders to actuators. The first approach is symmetrical to sensors: it consists of *multicasting* an order to a group of actuators. In our case study, such operation is used to inform car drivers of parking availability at the periphery of the city. The second approach to issuing orders to actuators corresponds to what is done when orchestrating devices in the small; it consists of invoking a specific actuator (*e.g.,* an information display at a given parking level) to perform a context-specific action (*e.g.,* display the number of available parking spaces of a given level).

Exposing how an application invokes actuators provides the sensor network infrastructure with valuable information. For example, individually invoking actuators is likely to be a key outcome of an application. Therefore, the application should probably not be launched, if individual actuators are not reachable, or at least, human intervention should be required to resolve the problem. In contrast, when an application invokes actuators using multicast, it should have some impact, even though some actuators may not be reachable when launching it.

6.3 Stages of Application Lifecycle

To support orchestration in the large, we introduce declarations expressing the sensor-network dimensions of an application presented in Section 6.2. To address these dimensions systematically, they are matched against each stage of the application lifecycle, from design to runtime. This approach has the following methodological and programming benefits.

- Sensor-network dimensions of an application are expressed early, and gradually matched against the sensor network infrastructure all along the stages of the application lifecycle.
- This gradual mapping raises a need for adaptation layers, when the sensor-network dimensions of an application cannot be reconciled with the target sensor network infrastructure (*e.g.,* missing sensors, unavailable delivery model).

As can be noticed, our staged approach keeps the application development independent from infrastructure details (*e.g.,* network protocol, bandwidth, data link features). In doing so, we strive to be as agnostic to the network as possible. We present the different stages of the application lifecycle and instantiate each stage with our case study.

We do not provide a formal definition of our declaration language dedicated to sensor-network dimensions of an application. Because it is simple, this language is presented informally, throughout the next section, with fragments from our case study. The information denoted by these fragments, declared at design time, is leveraged across the remaining sections devoted to the later stages.

6.3.1 Design Stage

The declared sensor-network dimensions provide the blueprint for later stages in the application lifecycle. In particular, they are valuable documentation to be used by the stakeholders of the sensor network infrastructure. Let us illustrate our approach with our case study of parking management, following the behavior dimensions introduced previously.

Discovery of sensors and actuators

Orchestration in the large requires us to categorize the sensors and actuators of interest. To do so, the designer needs to leverage a physical-space partitioning that is well-suited for their target application. This partitioning is likely to have been already introduced by the infrastructure owner and used to register sensors and actuators as they get installed.

A partitioning expressed in terms of declarations is given in Listing 6.1; it fulfills the *sensor grouping* requirement of our case study (see Chapter 3, Section 3.2). In these declarations, parking spaces are regrouped into parking levels, which are regrouped into parking lots. This partitioning of spaces can then be used by an application to discover sensors grouped by parking lots, matching the granularity most common to car drivers. Last, devices are registered as being in a city entrance, if they are located in the corresponding region.

```
1 parking_level INCLUDES parking_space

3 parking_lot INCLUDES parking_level

5 city INCLUDES city_entrance
```

Listing 6.1 – Extracts from the city partitioning of spaces.

Listing 6.2 uses these partitioning declarations to define application-specific discovery for sensors and actuators. For each parking lot, the application discovers presence sensors (lines 4 to 8) and CO sensors (lines 9 to 13) that are populating each parking level. Information displays are discovered within the parking lot for user information (lines 14 to 18) and at the periphery of the city to guide car drivers (lines 19 to 23).

```
1  APPLICATION parking_management
2  IMPORT city_partitioning

4  DISCOVER presence_sensors = {
5    service = sensor
6    source = presence :: boolean
7    group = parking_lot
8  }

9  DISCOVER co_sensors = {
10   service = sensor
11   source = co_level :: float
12   group = parking_lot
13 }

14 DISCOVER plot_info_displays = {
15   service = actuator
16   action = display
17   group = parking_lot
18 }

19 DISCOVER city_info_displays = {
20   service = actuator
21   action = display
22   group = city_entrance
23 }
```

Listing 6.2 – Application-specific service discovery for the parking management application.

Data delivery

The designer needs to define how data are delivered to the application. Listing 6.3 presents delivery model declarations for sensors, which build upon service discovery rules introduced previously. Event-driven delivery model is chosen for presence sensors because it allows to react only when the status of parking spaces changes (line 1). Given that this status is a boolean value, events are the most natural delivery model. This fulfills the *presence status delivery* requirement of our case study (see Chapter 3, Section 3.2).

To fulfill the *CO status delivery* requirement (see Chapter 3, Section 3.2), CO sensors are assigned the periodic data delivery model (line 3) and the event-based model (line 4). The periodic model is used by the ventilating system to renew the parking air, whereas the event-based model is used to warn users of an air pollution event when the CO level has reached a given threshold.

```
1 DELIVERY presence_sensors AS event_driven
2
3 DELIVERY co_sensors AS periodic::15::min
4                     AND event_driven
```

Listing 6.3 – Application-specific data delivery for the parking management application.

Actuating

Listing 6.4 presents declarations for actuating information displays using service discovery rules introduced earlier. Information displays are actuated with respect to their level within the parking lot since the application tailors the information to this granularity (lines 1 to 2). However, at the city periphery, information displays are actuated more globally because they receive recommendations for drivers entering the city and looking for an available parking space at their destination (lines 4 to 5).

In our case study, we envision information displays performing some local processing to select relevant information (*e.g.,* parking lots nearby a city entrance). For this purpose, we use the `MULTICAST` directive that triggers a partition of information displays (*e.g.,* at city entrances) to disseminate information on parking lots. A `FOREACH` directive is also available to allow for fine-grained actuation of ventilation systems based on the CO level measured at each parking level (not shown here).

Despite being high-level, our declarations provide a precise conceptual framework for the programming stage.

```
1 TRIGGER display ON plot_info_displays
2   MULTICAST parking_level
3
4 TRIGGER display ON city_info_displays
5   MULTICAST city_entrance
```

Listing 6.4 – Application-specific device actuation for the parking management application.

6.3.2 Programming Stage

This stage consists of developing the application, addressing all of its sensor-network dimensions including programming. In doing so, data structures accommodating sensor readings are implemented. Furthermore, the processing of sensor readings is programmed with respect to data delivery models, chosen at design time (Listing 6.3). Finally, note that the approach to processing data and producing results typically follows what has been defined at the design stage. For example, the reporting on parking space availability follows the granularity of the service discovery (*i.e.,* parking levels) (Listing 6.2). Furthermore, the processing of presence sensor readings follows an event-based model. Also, design declarations result in software architectural patterns. For example, a callback mechanism is typically implemented to serve an event-driven delivery model such as the one declared for presence sensors (Listing 6.3, line 1).

6.3.3 Deployment Stage

At deployment time, the key features of the target sensor network infrastructure are revealed. These features should thus be matched against the sensor-network requirements of the application to identify whether adaptations are needed. Let us illustrate this stage in the context of our parking management application. For service discovery, we need to verify that the target sensor network infrastructure provides the required sensors and actuators declared in Listing 6.2. Assuming that CO sensors are not present, the deployment process should fail, unless regulation does not require CO sensors in which case an adaptation layer could be invoked to produce fake values. Such a situation would likely demand human intervention.

Similarly, if bandwidth limitations in the sensor infrastructure or energy constraints on a sensor type prevents the periodic delivery requirement to be fulfilled (*e.g.,* Listing 6.3, line 3), a human intervention may be needed to make a decision. Such a situation would occur, for example, if the periodicity of CO measurements needed to be adapted because it likely needs to adhere to strict regulations.

The partitioning of sensors is another key feature of the target infrastructure. Adaptation code may be required to map the infrastructure partitioning into the application partitioning; this can be done when the application partitioning is more coarse than the infrastructure partitioning. For example, if the application required presence sensors to be grouped per parking lot (Listing 6.2, line 7) and the infrastructure only grouped them per level, a layer could gather the sensors for all levels to deliver the appropriate information to the application.

For data delivery, the models of the target infrastructure are matched against the ones required by the application. Some adaptation strategies can be used to account for mismatches. For example, an event delivery model can be simulated with periodic delivery, combined with an event condition. In our case study, if an application alerts parking users

when the air pollution has reached a given threshold, it might select an event-based delivery model to achieve this goal. An adaptation layer consists of monitoring the CO periodic measurements and trigger an event when a given threshold is reached.

In practice, depending on the physical architecture of the sensor network infrastructure, adaptation code may be placed close to the sensors (*e.g.,* at a base station) or run as an additional layer of the application if the infrastructure cannot be extended.

6.3.4 Launch Stage

When launched, the application may not have access to its required resources, if they are already serving other applications, or if the infrastructure has been temporarily or permanently reconfigured. This stage is distinct from the next stage, namely runtime because it does not address changes that may impact the application while it is running. The aim at launch stage is to adapt for changes that occurred between the time the application was deployed and its launching. For example, the sensor network infrastructure may have reserved some bandwidth to serve a given periodicity for CO sensors and be unable to fulfill this requirement because of a temporary failure of network nodes.

6.3.5 Runtime Stage

At runtime, the operating conditions of the application can change arbitrarily. For example, resources may be put offline for some technical reasons. In our case study, a base station failure may disconnect a number of sensors and actuators, sensors may fail, bandwidth may degrade, *etc.* These changing conditions can violate the quality of service requirements of the application and compromise its purpose. In our case study, if the failure of CO sensors does not offer the expected coverage of the parking lot levels, the threshold for a pollution level may not be reached because of the resulting inaccurate measurements. A crude approach could consist of terminating an application when its sensor-network requirements are violated. Human intervention could then be required to analyze the situation and resume operation, if possible.

A more advanced solution would consist of introducing a runtime mechanism that monitors the cardinality of the sensor partitions defined by the application. This mechanism should allow the application to adapt at runtime when operating conditions degrade. Similarly, the application needs a mechanism to react to data delivery models that are violated at runtime. Interfacing these events with a programming language can be done via the exception mechanism by introducing exceptional events dedicated to runtime errors, as described by Mercadal *et al* [Mercadal et al., 2010], or violation of quality of service contracts, as presented by Gatti *et al.* [Gatti et al., 2011].

	Design stage	**Programming stage**	**Deployment stage**	**Launch stage**	**Runtime stage**
Goals → ——————— **Dimensions ↓**	High-level specification	Supporting & guiding programming	Admission control	App/Network adaptations	Resolving QoS violations
Service discovery	Declaring sensors/actuators and their partitioning	Implementing data structures	Matching sensors/actuators	Validating resource allocation	Coping with infrastructure failures
Data delivery	Declaring data delivery models	Implementing data processing models	Matching data delivery	Validating & accommodating data delivery	
Actuating	Declaring actuation strategies	Implementing actuation strategies	–	Validating & accommodating actuation	Coordinating concurrent actuation

Table 6.1 – Overview of mapping sensor-network dimensions throughout the application lifecycle.

Summary 6.3

Table 6.1 summarizes how declarations of sensor-network dimensions are leveraged throughout the application lifecycle. As can be noticed, the range of actions based on declarations is very large: from guiding programming to admission control at deployment time, to coordinating concurrent activations of actuators at runtime. The table illustrates the framework laid out by our approach, providing a spectrum of opportunities that goes beyond our case study.

6.4 Discussion

We first explain how our work can leverage existing approaches on sensor networks. We then consider our design-driven development approach presented in Chapter 4 and discuss how it could be extended with the present work.

6.4.1 Leveraging Approaches from Sensor/Actuator Networks

This section shows how our sensor-network declarations could be further exploited for adapting the infrastructure to the application needs. This could be done by leveraging various existing approaches to application-specific optimizations in sensors/actuators networks. The discussion is organized around the previously discussed set of the key infrastructure concerns drawn from the literature on sensor networks.

Admission control

Madria *et al.* [Madria et al., 2014] propose the Sensor Cloud paradigm as a computing environment spread in a wide geographical area, unifying multiple WSNs, and available to one or multiple applications. This can be viewed as an extension to the notion of Cloud computing, adding virtualized sensing and actuating abstractions. The Missouri S&T Sensor Cloud is a concrete realization of this paradigm. It provides applications with sensing as a service, taking such parameters as the region of interest, the frequency and latency of sensed data. An admission control module (called provision management) examines the service requests to decide whether they can be fulfilled. When physical sensors are virtualized to several applications, this module computes the sampling durations and frequencies for satisfying all the requests. This configuration is recomputed when new applications join, or existing applications leave the system.

Because our approach exposes application needs, such as the required sampling frequencies of sensors, it could be beneficial for automating service (re)negotiation between the application and the sensor Cloud, at different stages.

Network configuration

Heideman *et al.* [Heidemann et al., 2003] show how WSN application performance can be improved by up to 60% and network traffic cut by half, by matching data dissemination algorithms to the application requirements. To this purpose, they offer a network API, allowing the application developer to choose between several data diffusion algorithms, such as push-based or pull-based. Additionally, they provide experimental data to determine which algorithm is best for which application communication patterns. For instance, a pull-based algorithm (namely two-phase pull diffusion) performs poorly when there are many sensors

potentially sending data to many sinks but the sensor data are actually sent rarely. In this situation, some push-based diffusion algorithms can significantly improve performance.

Clearly, by making sensor-network dimensions of applications explicit, our approach allows to automatically select the appropriate dissemination algorithm. The stage at which this selection should occur depends on when relevant information is known. In a single-application setting, where both sensor discovery and periodicity are known statically, the network can be configured at design time. For another example, if sensor periodicity is static but the number of actual sensors is known later, the choice can be done at deployment time or at launch time. In multi-application settings, the selection of a dissemination algorithm must at least be examined at each application launching.

Liu *et al.* [Liu et al., 2007] introduce a family of data dissemination/gathering algorithms in $n \times n$ grid WSNs, spanning the whole space between pure push – when sensors send data to applications, to pure pull — when applications send queries to sensors. Their algorithm family is based on a variable diffusion structure, similar to a comb. They show how the optimal balance can be expressed as a function of the grid size n and the relative frequencies of application queries and sensor events. As both frequencies are made explicit in our approach, the right push *vs* pull configuration in such a network could be computed automatically, as soon as the grid size n is known. For static networks, the configuration can be done at design time. When sensors are discovered at a later stage, such as deployment or launch time, the choice has to be performed at the corresponding stage.

Delicato *et al.* [Delicato et al., 2005] propose an architecture based on web services, in which sensors and applications declare the services they provide, respectively need, using standard SOAP configuration messages. These service descriptions include the sensor and data type, the geographical location, the acquisition interval (data rate), and the acquisition duration. A threshold may also be specified for non-periodic sensing. These sensors and application characteristics are used during network configuration for setting up the data dissemination protocol to minimize the energy consumption of the sensors used for delivering their data to client applications. When using our approach, the application needs are made explicit at design time, and can be thus used to configure the network at deployment time or to reconfigure it at launch time, to accommodate an incoming application.

Event filtering and processing

TiNA [Sharaf et al., 2003] is a scheme for minimizing sensor power consumption by exploiting the temporal coherency tolerance of applications. This approach goes beyond in-network data aggregation in that it does not send sensor readings at all, if this fits the QoD (quality of data) needs of applications. Thus, applications express their sensing needs using annotated SQL-style queries. These queries mention the type of data, its possible aggregation, and periodicity, and are annotated with temporal coherency tolerance. TiNA uses these annotations to suppress sensor readings (hence to save power) while still providing high quality approximated answers to application queries.

This approach is complementary to ours in that applications are able to declare their sensing needs as early as possible. These declarations can be directly used to leverage sophisticated underlying optimizations such as those provided by TiNA.

6.4.2 A Domain-Specific Language Approach

In Chapter 4, we introduced the DiaSwarm domain-specific language dedicated to the design of applications orchestrating sensors. Design declarations are processed to support and guide the programmer using generative programming. This strategy abstracts over the characteristics of the sensor network and allows the developer to declare what an application does, prior to programming it.

The work presented in this chapter was inspired by DiaSwarm but it is more general in that it is not dedicated to supporting the programming stage and covers more aspects with its declaration language (*e.g.,* actuators). Furthermore, we go beyond DiaSwarm in that we examine how sensor-network dimensions can be leveraged throughout the lifecycle of an application. In doing so, we bridge the gap between the application needs and the sensor-network concerns: we describe the details of their interactions and leverage the literature in sensor networks. Using these results, DiaSwarm could be further developed by extending its compiler to generate code that would address the stages and infrastructure concerns presented in our work. For example, adaptation layers could be generated automatically, either added to the application code or loaded in the sensor-network infrastructure.

6.5 Related Work

To identify the key sensor-network dimensions of an application, we already mentioned a range of works (Section 6.2). In this section, we review works pertaining to other aspects related to sensor networks, not discussed earlier. Specifically, we first examine the generality of sensor networks with respect to applications. Then, we review middleware support for large-scale sensor applications. Last, we examine how adaptive algorithms allow an infrastructure to dynamically adapt to an application behavior.

From WSN to SSN

Traditionally, Wireless Sensor Networks (WSNs) have been designed for a single application. As a result, they could be highly optimized in an application-specific manner at design time. This approach is well suited for small-scale networks but does not scale for networks of thousands of nodes. Indeed, at this scale, the cost of network deployment and maintenance becomes more important and return on investment is a key issue. Consequently, in recent years, research has been focusing on building *Shared Sensor Networks* (SSN), allowing several applications to run concurrently on the same sensor network infrastructure. Despite

their benefits, SSNs are still in an early stage, compared to WSNs. A very recent survey of SSNs [Farias et al., 2016] identifies several specificities of SSNs, raising new challenges that must be addressed. Such challenges include dealing with the heterogeneity of the infrastructure, dealing with resource contentions, and optimizing sensor data flows for several applications at the same time. Heterogeneity imposes a looser coupling between the applications and the infrastructure. Dealing with resource contention requires exposing application needs early to ensure that a new application does not interrupt currently running ones. Optimizing sensor data flows also requires exposing early the sensor-network behaviors of the application to enable cross-application optimizations. While SSNs bring concrete solutions to these challenges, they do not (yet) offer tools for streamlining the development of SSN applications.

Our work proposes an approach towards resolving this development problem by exposing the sensor-network dimensions of an application early, at design time. These dimensions can then be exploited at different stages. For example, at deployment time, our declarations address such challenges as resource contention.

Middleware

Several works propose middleware solutions for supporting large-scale sensing and actuating applications. A general approach to this problem domain is the SwarmOS vision. In a white paper [Lee et al., 2014], the authors promote a middleware platform for developing applications at the frontier between a large-scale WSN and the Cloud. The goal of SwarmOS is to offer high-level services, intermediating between the WSN/Cloud resources and applications (called "swarmlets"). Such services include access control and virtualization, data summarization and aggregation, discovery, *etc.* While such middleware services could greatly simplify the development of sensing/actuating applications, they do not offer a systematic method for developing such applications. Additionally, SwarmOS specifically targets applications with dynamic component graphs, allowing continuous graph reconfiguration. Our approach favors applications with a static internal structure, exposing as many of their characteristics at early stages. Extending application behavior with dynamic aspects will be studied in future work.

Other middleware solutions simplify the development of applications on SSNs by increasing the abstraction level. However, they do not address infrastructure optimizations. For example, the LooCI middleware [Hughes et al., 2009] implements an easy-to-use component composition model, based on event publishing and subscription. This middleware allows for run-time re-configuration and introspection. Such middleware solutions mostly match application needs against sensors capabilities at runtime; they do not attempt to anticipate (mis)matches at earlier phases in application lifecycle to improve their reliability.

Adaptive algorithms

Another body of work concerns adaptive algorithms for sensor networks. This is a highly dynamic approach based on observing application needs and the associated communication patterns, and adapting the network infrastructure to optimize performance. The Adaptive Multi-Criteria Routing (AMCR) algorithm [Eltarras and Eltoweissy, 2010] is a key instance of this approach. Its authors argue that most of the WSN routing protocols are designed for a specific application class. AMCR is a generic routing protocol able to adapt to traffic patterns recognized by observing running applications. More specialized routing algorithms, such as the comb-needle routing structure [Liu et al., 2007], also integrate adaptive behavior based on monitoring an application at runtime and estimating the frequencies of sensor events and application queries. The above-mentioned adaptive approaches are more ambitious than ours, in that their aim is to automatically optimize the sensor infrastructure, without making the application needs explicit. However, networks supplying such an advanced self-tuning would need to be extensively tested to ensure the robustness of the adaptive optimizations.

Our approach is more predictable in that it exposes sensor-network dimensions of applications such that their execution follows statically defined parameters. Nevertheless, we could re-use the work on adaptive algorithms by leveraging our sensor-network application declarations to perform their optimizations statically, instead of dynamically. As a result, our static approach would incur no runtime overhead.

Chapter 6: Summary

In this chapter, we introduced an approach to developing large-scale orchestrating applications that revolves around a declaration language covering the key sensor-network dimensions. This declaration language has been described through a case study of a parking management system. We have shown that our sensor-network declarations can be leveraged across the main stages of an application lifecycle to match the application requirements to a target infrastructure. We have discussed a set of key infrastructure concerns identified in the literature on sensor networks and we have shown how our approach could take these works further by leveraging information from our declarations.

7

An Evaluation of our Approach

The success of a software development approach dedicated to a domain critically relies on its ability to overcome prevailing software development obstacles. Also, the usability and the usefulness of the approach have to be studied to ensure its acceptance among the developers community. Towards this end, we carried out two experiments in which we applied our design-driven development approach to the domain of Internet of Things. In a first experiment, we put our approach to a test by developing four orchestrating applications specified by different sensor manufacturers. We report on the effectiveness of our approach in solving the identified software engineering challenges. In a second experiment, we assess the learning cost to use our approach by involving four professional programmers in a usability study. This second experiment brings preliminary evidence that the proposed design-driven approach can be used effectively by professional developers after only half a day of training.

Contents

Contributions

- Quantitative and qualitative data on the approach effectiveness, usage and usability.
- Lessons learned from the experimental evaluation of a design-driven approach.

This chapter is a first step for assessing the potential for transferring our design-driven approach to the industrial practice of the domain of Internet of Things (IoT). The evaluation has been carried out in the context of a French collaborative project, called *Objects World*, involving five sensor manufacturing companies and four research labs [Objects World Consortium, 2012]. Objects World aims at building a sustainable ecosystem of IoT stakeholders based upon a nationwide, low-power ultra-narrow band radio network, called SIGFOX [1]. This network already supports products and services that demonstrate its market potentials. It has been deployed in a number of countries (*e.g.,* France, United Kingdom, Spain, Netherlands) and is supported by major players such as Samsung, which integrated the SIGFOX network protocol into its ARTIK IoT platform [2]. [M2M World News, 2015]

When the Objects World project was being set up, the companies involved knew from their experience how much software development is a bottleneck to realize the full potential of IoT. In fact, this domain is just emerging from an industrial viewpoint. While extensive work has been devoted to the infrastructure of IoT, such as traffic management and device constraints (*e.g.,* battery life), little effort has been dedicated to the process of developing applications and services [Links, Cees, 2015]. This situation prompted the Objects World consortium to invite a research group specializing in software engineering to participate to the project and bring tools to improve the development process of IoT applications. Hence, our participation to the project. Numerous interactions with the consortium companies revealed key challenges to be addressed in this domain. We presented these challenges in the beginning of this dissertation in Chapter 1.

Given these challenges, we decided to evaluate to what extent a design-driven development approach coupled with an automated software engineering toolset, named DiaSuite, could overcome these challenges. To do so, we solicited researchers working on this technology and professional programmers.

The evaluation was carried out in two stages. In a first experiment, we examined the effectiveness of our approach in dealing with software engineering challenges introduced in Chapter 1, Section 1.2. To do so, we implemented four applications specified by different sensor manufacturers and addressed these challenges for each application. In a second experiment, we evaluated the learning cost and usability of our approach by means of a usability study, which involved professional programmers from sensor manufacturers. The contributions in this chapter can be summarized as follows.

Effectiveness

Using four case studies, we show that our design-driven development approach effectively covers a broad range of orchestrating applications and provides support to deal with software

1. http://www.sigfox.com
2. http://www.artik.io

engineering challenges presented in Chapter 1. We examine each case study and describe how application design allows us to cope with these challenges.

Usefulness & usability

By means of a usability study, we show that the approach can be efficiently transferred to developers in the IoT industry with only half a day of training. Further quantitative and qualitative data are provided, including a usability questionnaire and developer interviews to investigate the perceived utility of the tools.

Lessons learned

We sketch the steps we are currently taking to consolidate these preliminary results into a more complete evaluation of our approach, including its impact on programmers' productivity and application quality.

Chapter 7: Outline

Section 7.1 provides details on the development toolset used throughout both experiments. Section 7.2 gives further details on the Objects World project and presents the first experiment that evaluates the effectiveness of our approach via four industrial case studies. In Section 7.3, we evaluate our approach in the second experiment through a usability study involving professional programmers. Related work on development tools for IoT as well as experiments for tool support evaluation is given in Section 7.4. The lessons learned from both experiments are documented in Section 7.5.

7.1 Used Tool Support

In contrast with Chapter 5, the evaluation presented here focuses exclusively on software engineering challenges. We conducted two experiments of which neither examined large-scale orchestration challenges. Instead, our evaluation is aimed at assessing the benefit of a design-driven approach to drive the development of orchestrating applications in general.

Both experiments were carried out using the DiaSuite toolset, dedicated to orchestrating applications for traditional pervasive computing environments [Bertran et al., 2012]. This toolset provides developers with the DiaSpec domain-specific language dedicated to designing applications orchestrating sensors and actuators [Cassou et al., 2012]. As shown in Figure 7.1 design declarations are used by a compiler to generate application-specific support covering the complete application lifecycle.

The taxonomy layer of the DiaSpec language provides a flexible and reusable catalog of entities for a given application domain, abstracting over underlying technologies and implementation details. For each entity, either hardware or software, its data sources and actions are declared, as well as attributes such as a name, a unique identifier or a location. The application design layer of the DiaSpec language allows to declare the application components and the possible flows of data connecting them together.

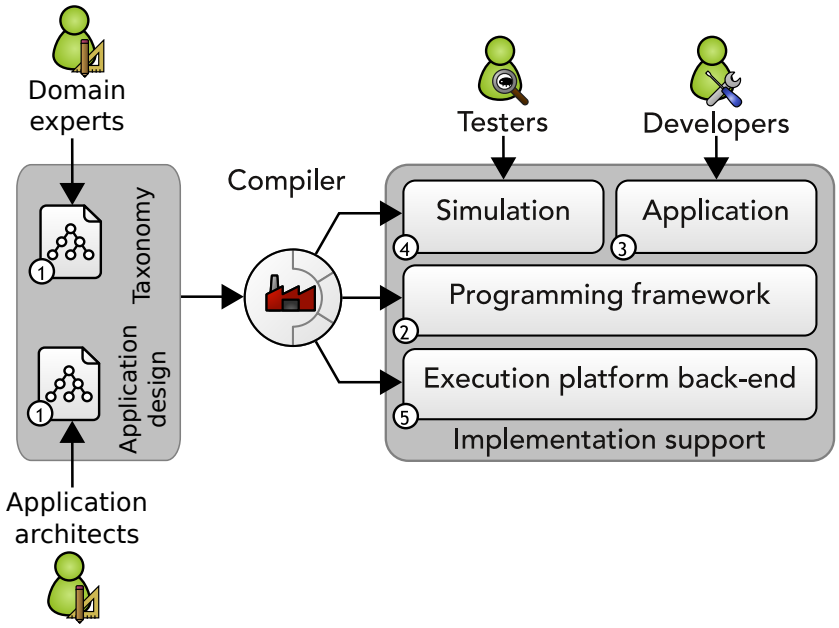

Figure 7.1 – DiaSuite tool support for the SCC application lifecycle.

The DiaSuite toolset has been used for developing many successful proof-of-concept applications in several domains: telecommunications [Bertran et al., 2009], building automation [Bruneau et al., 2009], avionics [Enard et al., 2013a], software monitoring [Cassou et al., 2011a], robotics [Cassou et al., 2011b], and assisted living [Caroux et al., 2014]. The variety of domains successfully targeted by DiaSuite demonstrates the broad applicability of a design-driven methodology in various kinds of applications involving sensors and actuators. However, most of these applications were developed in a research setting. Although some specifications were fueled by industrial needs, they did not directly correspond to external specifications.

In the following we give details on support generated to facilitate testing, deployment, and maintenance, not previously discussed.

Testing support

DiaSpec declarations are used to produce simulation support for every entity, in the form of a concrete class implementing a "mock" entity, containing methods to simulate each possible activity of a device, such as pushing data from a sensor or making some data available for subsequent pulls. This support allows for thorough testing of orchestrating applications prior to their deployment on real infrastructures composed of sensors and actuators. In addition, mock devices can be gradually substituted with real devices without the need to introduce any changes in the application code. Similarly, support is generated for testing entity implementations (also called drivers) before any application is developed on top of them.

Deployment support

The compiler produces a runtime layer dedicated to a given deployment platform: it can be local (*e.g.,* OSGi) or distributed (*e.g.,* WebServices). This support allows application developers to abstract over how communications between components are implemented. It also allows applications to be deployed in an already running platform, reusing available entities. Thus, the implementation of entities is typically deployed independently of applications: deploying an application simply consists of deploying its context and controller components, reusing the already installed entities, possibly shared with other applications.

Maintenance support

There is no specific code generated for supporting the maintenance of an orchestrating application. Rather, maintenance is supported by automatically regenerating all the pieces of code mentioned above, whenever the taxonomy or the application design are modified. The separation of generated code (which takes the form of abstract classes) from developer-supplied code (separate sub-classes) enables a smooth update of the generated code. Typically, following a design change, the developer-supplied code has to be adapted to conform to the regenerated code API. During this maintenance process, developers are guided by the Java builtin type checker that points at the code locations needing changes. Most of the time, the needed changes are available as suggested editing actions (*e.g.,* adding a method parameter or changing a type) thanks to the integration of the approach with the Eclipse IDE in the DiaSuite toolset.

7.2 Evaluating Approach Effectiveness in Dealing with Soft. Eng. Challenges

To assess the benefits of design to support the development of orchestrating applications and to determine the potential of transferring our approach to an industry setting, we implemented four applications, as specified by different partners of the Objects World project. Before describing these applications, let us further introduce the Objects World project.

7.2.1 The Objects World Project

The leading partner of the Objects World project [Objects World Consortium, 2012] is an emerging wireless network operator dedicated to low-bandwidth IoT applications, called SIGFOX. Five object manufacturers cooperate within the Objects World project to make their products compatible with the low-bandwidth network. Their products include alarm systems, energy monitoring, fall detectors, odor detectors, taste sensors, vibration sensors for engine monitoring, tire pressure sensors, and connected door locks. Four research labs bring to this consortium complementary know-how on electronic chip design and integration, nano-technology, chemistry applied to sensor design, as well as object orchestration. Based on these technologies, one of the outcomes of the project is to demonstrate a few prototype IoT applications in different domains, so as to validate the potential of the resulting eco-system of actors.

The following subsections describe four applications developed by two researchers in our group who are trained experts in using the DiaSuite toolset. For each case study, we asses the efficacy of our design-driven methodology to address identified software engineering challenges. This assessment ranks from "not at all effective" (no star) to "very effective" (3 stars) with the exception of the rapid software development challenge examined in the discussion section for all the case studies. All the results are shown in Table 7.1, together with some quantitative aspects, such as the size of both the manually written code and the automatically generated code.

	Code size		**API**	**Testing**	**Evolution**
	man.	**gen.**	**reuse**	**sup.**	**sup.**
DoorLocks	186	1580	***	***	***
PalletTracker	326	2642	*	***	***
HomeAlarm	221	2858	***	***	**
HeatingMonitor	300	2272	***	*	**

Table 7.1 – Efficacy of DiaSuite in solving the development challenges, when used by experts on four applications.

7.2.2 Connected Door Locks

The DoorLocks application was specified by Axible Technologies [3], a provider of digital products and services for gate access control. The goal of this application is to remotely monitor instances of electronic door locks connected to the Internet through an onboard SIGFOX transmitter. The features specified were: real-time event logging, abnormal event alerts (stuck or forced door), and low battery state notifications.

We implemented all the features by integrating the smart door locks, an email web service, a logging device, and mobile phones for SMS notifications in an orchestrating application. Figure 7.2 shows the architecture of the application. Door locks provide two sensing capabilities: the `locked` facet detecting when a door has been locked or unlocked, and the `state` facet detecting state changes, such as a door becoming stuck or forced and the battery level. The `Format` context component is triggered by any event coming from a door lock. The two kinds of sensors do not provide the same information, but the `Format` context builds a uniform `Event` structure. `Event` structures are received by the `Logger` controller component, which forwards them to the `Logger` device. Thus, the `Format` context together with the `Logger` controller completely implement the feature of real-time event logging.

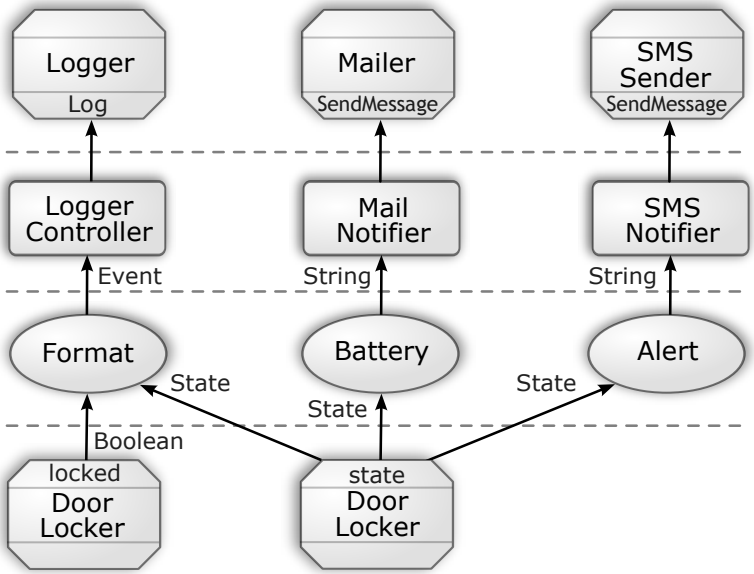

Figure 7.2 – The graphical view of the DoorLocks application.

A separate data flow, going through the `Battery` context and the `MailNotifier` controller implements the low battery notification feature: the `Battery` context parses the events coming from the state sensor of the door locker, selects those concerning a low battery state, and

3. http://www.axible-connects-for-you.com

forwards them as a concise string to the `MailNotifier` controller. This latter controller reformats these compact messages in more explicit textual form, producing a complete e-mail message for a pre-defined email address — the service ensuring general maintenance of the door locks. Finally, a third data flow going through the `Alert` context and the `SmsNotifier` controller implements the critical conditions notification feature: whenever a stuck or forced door is detected, an SMS is sent to a predefined cellphone number, for urgent intervention.

In the DoorLocks application, the described challenges are instantiated and handled as follows.

Overcoming heterogeneous object APIs

The API of an email web service is already included in the standard device taxonomy, under the `Mailer` entity, and used by many other applications. Similarly, a standard `SmsSender` entity is reused for sending SMS messages to a cellphone, abstracting from the underlying platform (various webservices for SMS sending). A new object called `DoorLocker` has been added to the taxonomy, (see Listing 7.1), specializing the standard entity `PhysicalDevice`. Thereby, the `DoorLocker` entity reuses attributes such as the `location` and the `user` of the `PhysicalDevice` entity. Moreover, generic `Lock` and `Unlock` actions have been added to the taxonomy that could serve other kinds of locks, such as electronic padlocks. (***)

Facilitating testing

The automated generation of a fake `DoorLocker` entity was useful for testing the application without using real customer data, and also for simulating events such as a forced door, without changing a single line of application code. (***)

Supporting rapid evolution

We evaluated the maintenance support in the case of the DoorLocks application by executing the maintenance scenario suggested below.

The DoorLocks application handles sensitive information such as status information related to a forced door, which designates a location that is temporarily vulnerable until a repair action is performed on site. In order to completely ensure the privacy of forced door information, the DoorLocker API should be redefined by segregating non-critical battery state from critical state, such as a forced door. We segregated in the DoorLocker API, non-critical battery state from critical state (*e.g.,* a forced door) by creating two distinct facets, respectively called `batstate` and `state` in the taxonomy. Then, we modified the application architecture to segregate the two data flows as shown in Figure 7.3. The Eclipse plugin regenerated all the application-specific framework as soon as these modifications were saved. As a result, the Eclipse IDE pinpointed three locations in the code that needed changes, and

```
device DoorLocker extends PhysicalDevice {
        action Lock;
        action UnLock;
        source locked as Boolean;
        source state as State;
}

device PhysicalDevice extends Device {
        attribute location as String;
        attribute user as String;
}

device Device {
        attribute id as String;
        source isAlive as Boolean;
}

action UnLock {
        unlock();
}

action Lock {
        lock();
}

structure State {
        state as String;
        timestamp as String;
        batteryLevel as String;
}
```

Listing 7.1 – Device declarations used in DoorLocks.

the modified application could be completed and tested successfully in less than one hour of work. (***)

7.2.3 Pallet Tracking

The specification for the pallet tracking application was provided by LDL Technology [4], a company specializing in wireless embedded electronic systems for Tire Pressure Monitoring Systems (TPMS), Vehicle Immobilization (IMMO), Keyless Entry and Start system (KESS) and Body control units. The goal of this application is to track the state of metal pallets, used to transport merchandise to warehouses and retail stores. Often, pallets are not returned

4. http://www.ldl-technology.com

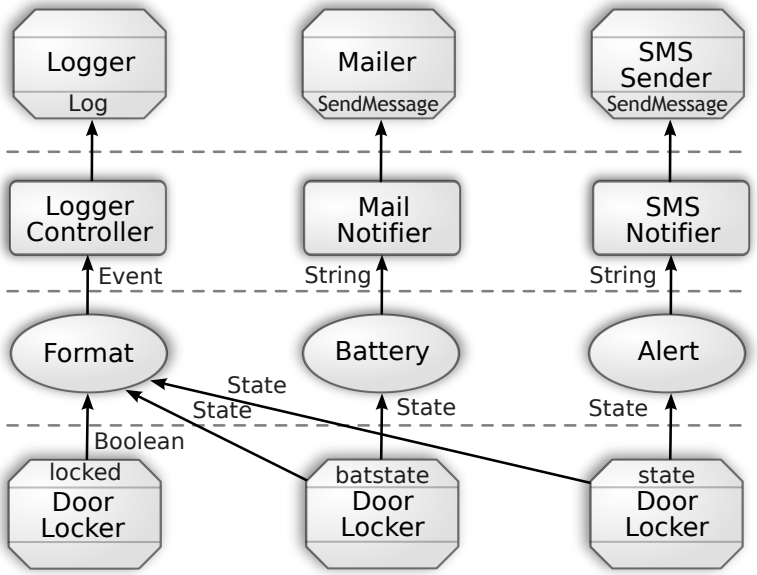

Figure 7.3 – The architecture of the DoorLocksPlus application.

or get lost, which forces companies to re-invest into these assets. To ensure that pallets are returned, each pallet is equipped with a sensor and an integrated SIGFOX transmitter, sending data to the application periodically (*e.g.*, every 10 min.). Collected data are used to identify the location of a pallet and its temperature. The position of a pallet is determined by triangulation using GPS coordinates of SIGFOX base stations that are receiving data from the device. Data originating from the sensor infrastructure are stored in a database and used by client applications to explore the history of received messages per pallet or warehouse. Also, an inventory feature allows client apps to keep track of the number of available pallets per warehouse. Finally, the user may define alerts via the client app in order to identify unusual situations, such as a pallet status not transmitted for a long period of time.

The design of the PalletTracker application is depicted in Figure 7.4. The features of the pallet tracking device are ensured by the `PalletTracker` and `TemperatureSensor` devices. The `PalletTracker` device defines the position facet, indicating the position of a pallet using GPS coordinates. In this scenario, however, the position of a pallet corresponds to the position of the base station, which received the message. The accurate position of a pallet is computed in the `Position` context via triangulation using data collected from multiple base stations. The temperature of pallets is processed separately in the `Temperature` context. Finally, the `State` context gets triggered upon publishing the state of the `PalletTracker` device via the state facet and retrieves the computed device position and temperature from the corresponding contexts.

The remaining application components are dedicated to the detection of unusual situa-

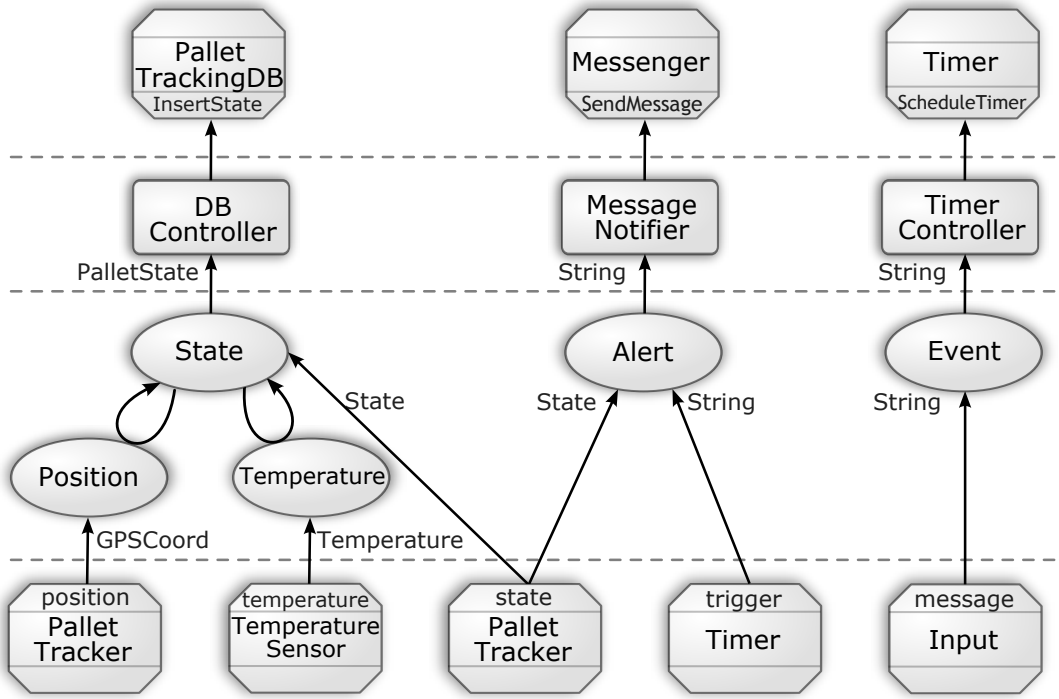

Figure 7.4 – The architecture of the PalletTracker application.

tions. The `Input` device is an abstraction for the client application used for sending messages indicating events of interest that need to be detected. The message is processed in the `Event` context, which recovers the name and verification period (*e.g.,* one hour) of the event. This result is passed to the `Timer` controller, which schedules a new `Timer` according to the given period. Upon expiration of this period, the `Timer` device triggers the `Alert` context, which is in charge of collecting timestamps, indicating the last state published by a `PalletTracker`. In case no device state has been received during the verification period (e.g. during the last hour), the context returns a message that is to be sent to the client application via the `Messenger` device.

Overcoming heterogeneous object APIs

The application models the pallet tracking device using different abstractions from the taxonomy. It reuses the default declaration of the `TemperatureSensor` device to recover the temperature of pallets. A `PalletTracker` device has been added to the taxonomy to detect the state and position of pallets. The decomposition of a complex device into multiple simple devices promotes API reusability. Interfaces for the `Timer`, `Messenger` and `Input` devices have been reused. The `PalletTrackingDB` abstraction had to be added to the taxonomy. The device is not likely to be reused by other applications since it exposes only ad hoc constructs

for the given application. Introducing a database as a reusable entity is not easily amenable to a DiaSpec declaration, thus an ad hoc abstraction has been chosen instead. This aspect is discussed in more detail in the conclusion of this chapter. When a composite device (*e.g.*, multisensor) is frequently reused, it can be advantageous to declare such a device via multiple inheritance to leverage the existing infrastructure. Currently, our design language does not support multiple inheritance. (*)

Facilitating testing

The automated generation of fake `PalletTracker` and `TemperatureSensor` entities was useful for testing the application without using real customer data, and also for simulating events such as a lost pallet, without changing a single line of application code. (***)

Supporting rapid evolution

We assessed the maintenance support by integrating, in a second step, the application feature that sends an alert message to the user upon detecting that a pallet status has not been received for a long period of time. The integration of this feature was carried out by defining two separate control flows without any modifications to the previously defined application code. The new application feature has been completed in less than one hour. (***)

7.2.4 Home Alarm System

The application specification was provided by Telecom Design [5], a company specializing for more than a decade in the development of innovative solutions for the domain of IoT/Machine-to-Machine. Their solutions typically involve sensors and more complex smart objects, such as a home alarm system. The goal of the application is to detect dangerous situations at home and consequently notify the owner, as well as authorities and organizations in charge of home safety and security (security agencies, insurance companies, police forces, *etc.*). A home alarm system comprises a smoke sensor to detect fire outbreaks and a motion sensor for intrusion detection. A keypad allows to arm or disarm the entire system. Upon the detection of a dangerous situation, the home alarm system sends data to the application, which notifies the owner and the authorities in charge via messages.

The design of the HomeAlarm application is depicted in Figure 7.5. The detection of an intrusion is ensured via the `IntrusionDetector`, which publishes the device state information to the `Intrusion` context. This context returns a message that is to be sent to the owner and the authorities via the `SMSSender` and `Messenger`, respectively. Likewise, the `Fire` context returns a message to both actuators upon the detection of smoke in a home. Furthermore, a contact sensor has been installed on the entrance door to detect when the door is

5. http://www.telecom-design.com

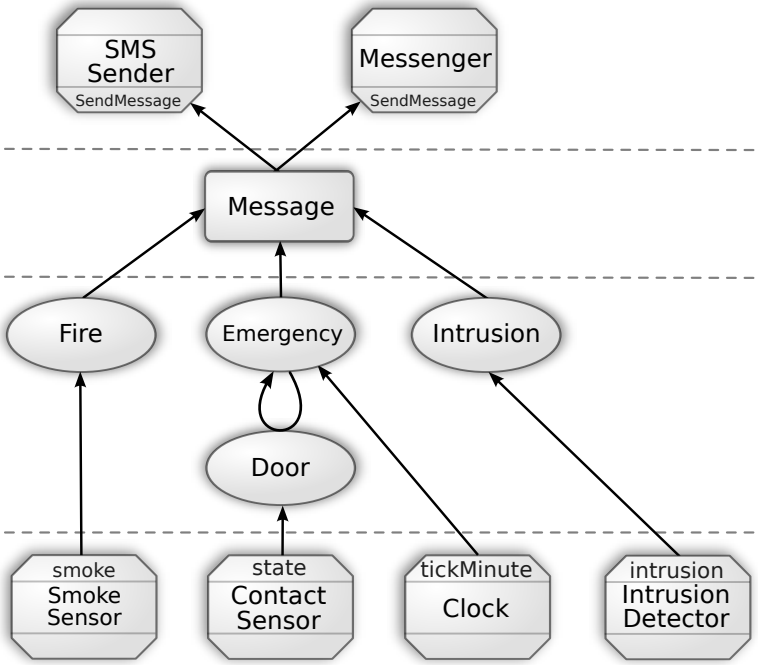

Figure 7.5 – The architecture of the HomeAlarm application.

left open for an unusual long time. This verification is performed by the `Emergency` context, which is triggered every minute via the `Clock` device. The `Emergency` context recovers the list of doors that have been left open from the `Door` context.

Overcoming heterogeneous object APIs

The `SmokeSensor` and the `ContactSensor` reuse the `Sensor` device, which provides attributes allowing a device to be identified by its user and location. The `IntrusionDetector` device reuses the `PhysicalDevice` for the same purpose. The `Clock` device is typically reused in applications verifying periodically the state of the environment or for sending regularly notifications to the user. Finally, the `SMSSender` and `Messenger` devices are reused in most of orchestrating applications since their interfaces allow to abstract over a multitude of display devices (*e.g.,* mobile phones, tablets, dedicated dashboards, *etc.*). (***)

Facilitating testing

As in the previous cases, the generated support for testing both sensors and the `IntrusionDetector` device allowed the application to be tested without the need to deploy real sensors. Furthermore, a fake `Clock` device has been generated allowing the

application to be tested without the need to wait until the clock time expires (*e.g.,* one second, minute, hour). No modifications had to be provided to the application code. (***)

Supporting rapid evolution

The evolution support has been assessed by implementing only in a second version of the application the feature detecting an entrance door that has been left open for an unusually long period of time. The application evolution required the definition of two supplementary contexts as well as the usage of a `Clock` device. The `Message` controller has been extended with an additional interaction contract, which required the implementation of a newly generated abstract method in the existing class. (**)

7.2.5 Heating Monitoring

To complement the previous three applications specified by object manufacturers, the HeatingMonitor application was specified by our group as a use case involving more complex application logic. By integrating a novel heating energy meter produced by Telecom Design [6] with other data sources and services, the HeatingMonitor application monitors and analyses the efficiency of heating systems online in order to detect an inefficient building environment and to propose improvements to the owner. Also, to enable more in-depth, offline analysis of the selected heating systems, the application records relevant events in a log.

The architecture of the HeatingMonitor application is given in Figure 7.6. The logging feature is implemented by a simple data flow, going from the heating meter, through the `Format` context and the `Logger` controller, to the `Logger` actuator; this latter component records relevant heating events on stable storage for later analyses. The online monitoring of heating systems efficiency is implemented by the `Efficiency` context. Whenever the `HeatingMeter` entity pushes a consumption value (either periodically or when the heating consumption changes significantly), the `Efficiency` context queries the target temperature from the `Thermostat` entity and the outside temperature from a weather web service, and computes a current efficiency factor for the heating system. The `AverageEfficiency` context computes the average efficiency value for the heating system of each customer. The `ReferenceEfficiency` context computes a global reference efficiency value by averaging all the instant efficiency values of all customers. Finally, the `Analysis` context, triggered by a `Clock` entity, periodically (e.g., every month) compares the efficiency value of various customers heating systems with the reference efficiency value, to detect inefficient ones. If such a system is found, the name of the corresponding customer is pushed to the `MailNotifier` controller. This latter component prepares an email for suggesting a detailed audit for that customer, and sends the mail using an email web service to the appropriate consulting ser-

6. http://www.telecom-design.com

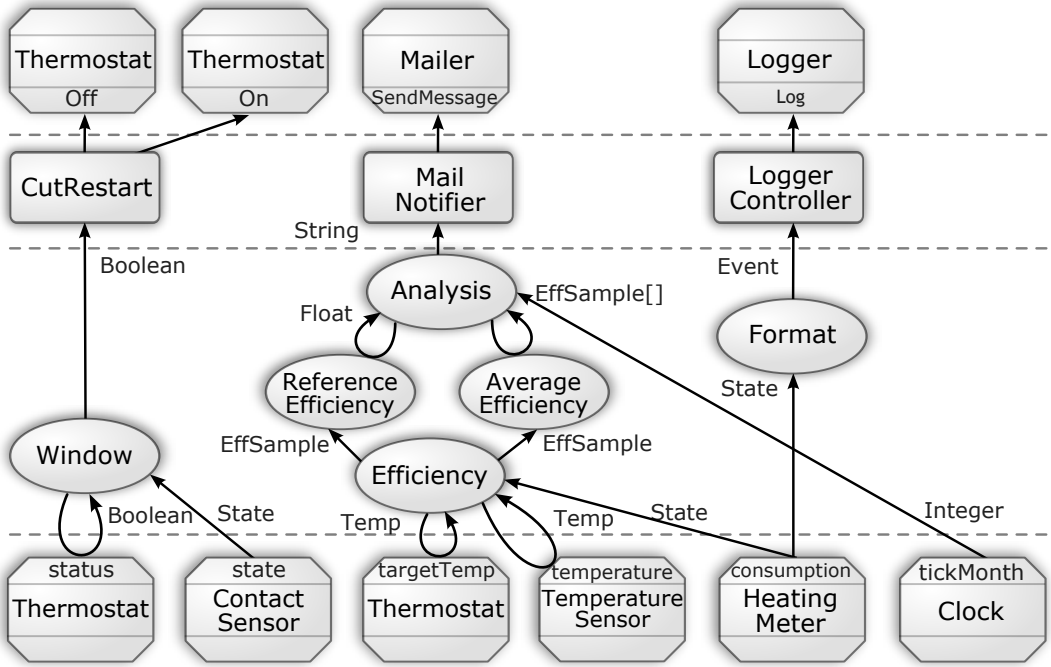

Figure 7.6 – The architecture of the HeatingMonitor application.

vice. A third data flow going through the `Window` context has been implemented separately in the maintenance scenario described at the end of this section. In the `HeatingMonitor` application, the identified software engineering challenges are instantiated and handled as follows.

Overcoming heterogeneous object APIs

The standard smart object taxonomy contains a generic `Sensor` entity extending `PhysicalDevice`, and thereby inheriting attributes, such as its user and location. The `Sensor` models any object containing sensing capabilities but no actuator capability. The `HeatingMeter` entity has been defined by reusing the `Sensor` entity extended with a particular data source giving the current heating consumption. The heating consumption is encapsulated in a generic `State` structure used by many other devices. This structure contains additional data beyond the sensed value, namely a timestamp and the battery level. Thus, the definition of `HeatingMeter` reuses several standard APIs in the taxonomy. (***)

Testing support

The automatically generated mockup device drivers for the `HeatingMeter`, `Thermostat`, and `Temperature` web service were useful for testing the application before potentially massive future deployments. However, even using these simulated devices resulted in quite complex testing scenarios. Indeed, the application architecture graph (in Figure 7.6) is deeper than in the other applications and only its sensor inputs can be controlled. For instance, before the `Analysis` context is able to detect a particularly inefficient heating system, many heating measures in several customer sites must be simulated to accumulate a representative reference efficiency and an efficiency value standing out. It would be simpler if one could directly inject the input of any context module, but this is not currently possible in our approach. Consequently, we had to trace the values of various intermediate contexts using log messages before coming to the relevant test scenarios of this application. (*)

Supporting rapid evolution

The maintenance scenario selected for the HeatingMonitor application consisted in adding a new feature, independent of the implementation of the other features. This feature aimed at optimizing energy consumption by pausing the heating system while any window is opened within its scope. This function is implemented by the `Window` context being triggered by a contact sensor fixed on a window. The context then checks whether the thermostat in the corresponding area is switched on; if so, it passes a pause command to the `CutRestart` controller which switches off the thermostat, and records this thermostat as being paused. The thermostat is switched back on only when all the related windows are closed. In order to implement this new feature, the standard `Appliance` entity — a common ancestor to many devices including `Thermostat` — had to be extended by adding a new sensor giving the current on/off status. This change thus causes the compiler to regenerate the code of all inheriting entities, but this has no impact on the application code already written, so no manual adaptation was necessary. Arguably, this would be the case in any object-oriented implementation of this application, because adding a new field to an ancestor class does not change the users of its subclasses. (**)

7.2.6 Discussion

We now discuss how our approach contributes to the rapid software development of orchestrating applications in the IoT domain in terms of code support generated for the examined case studies. We then present two additional challenges, not previously examined that have been intensively discussed in IoT research and that should be considered in future experiments.

Supporting rapid software development

As can be seen in Table 7.1, the majority of the application code in the presented case studies was generated from design declarations, which considerably reduces development time and thus supports the rapid software development of orchestrating applications. Depending on the application, the proportion of automatically generated code varies between 88% and 92% percent. This contributes to demonstrating the "efficiency" of the approach (beyond its effectiveness).

Ensuring security and privacy

IoT applications interact with the physical world. As such, they commonly gather sensitive information: the state of a critical equipment, the vital sign of a user, or the occupancy of a home. Our partners routinely detect and process sensitive information: forced doors, absence from a building for a long time, location of valuable containers, *etc.* Managing such information raises numerous issues in security and privacy. Many aspects related to security are being addressed by network protocols and authentication layers [Seitz et al., 2013]. However, guarding against information leakage should be also ensured at the application level. Indeed, from a developer perspective, security and privacy concerns may obfuscate the application code. Furthermore, from the perspective of end-users downloading an application from an online store, how can they know about what sensitive information flows to an unsecure sink? DiaSpec diagrams partially meet this challenge [Van Der Walt, 2015].

Going beyond silo-based applications

Several studies [Links, Cees, 2015; Baccelli and Raggett, 2015; Miori and Russo, 2015; Raggett, 2015; Girolami et al., 2015; Vallati et al., 2015] point out that IoT is in an emerging state where silo-style applications are the norm: these applications provide a limited number of services implemented over isolated objects. A partner, manufacturing sensors, gave us his account of the situation and an example: *"Today, we're mostly in a model with one application, one server, one product. But we already had a customer who bought two different products from us and wanted to control both of them using a single interface; we had to say no ..."*. The tremendous potential of IoT will map into service functionalities when applications horizontally integrate sensors and actuators from various silos. To build such applications, it is not only necessary to reuse object APIs (see the first challenge above), but also to reuse the objects that are already deployed and used by other running applications. To do so, objects and applications must be deployable onto a uniform platform that standardizes runtime communications between them, service initialization and discovery.

All the applications presented in this case study reuse smart objects to a certain level. For instance, the HeatingMonitor application intensively reuses data providers, such as the Thermostat of the existing heating system and the TemperatureSensor giving the outside

temperature. A weather web service gives the temperature of each customer's city. However, other temperature sensors closer to the customer location (*e.g.,* mounted on a customer's building) could be reused, if available, increasing the accuracy of the computed efficiency. Finally, the maintenance act of the HeatingMonitor application, for instance, shares contact sensors that may typically be used for security applications such as alerting when a window is left open in an empty building.

Summary 7.2

The experience of implementing and maintaining the four use cases above, summarized in Table 7.1 shows that our design-driven approach effectively solves the software bottleneck challenges in most situations, with a few notable exceptions. However, in all these examples, the DiaSuite toolset was used by experts. This does not at all disqualify the evidence; it just means that the effectiveness of the tools has been probably pushed to its maximum potential. Therefore, it is legitimate to question the effectiveness of the same tools when used by developers that are new to this technology: how much training do they need to start using the toolset? After the training, can they successfully develop a typical orchestrating application? Do they perceive the advantages of the design-driven approach in solving some of the challenges? To give a preliminary answer to these questions, we performed a usability study detailed in the next section, complemented by questionnaires and some interviews.

7.3 Evaluating Approach Usability

In this section, we present the second evaluation of our approach, which was carried out by means of a usability study, questionnaires and interviews. We give details about the different stages of the user study and the results we obtained. Application code, the used questionnaires and the training material are available on the accompanying Objects World website [Objects World Consortium, 2012].

7.3.1 Usability Study Definition

The main goal of the usability study is to evaluate the cost of learning our design-driven development approach for programmers with various levels of experience. Experimental results are used to validate an upper bound of the learning cost needed for transferring our approach to professional programmers. The study was carried out on a group of four participants with different levels of experience in software development. Participants were

Contributions presented in this section have been published in proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU). [Kabáč et al., 2015]

assigned a software engineering task, which involved the design, implementation and testing of an orchestrating application. Each participant attended a training session prior to the experiment to acquire basic skills in rapid prototyping of orchestrating applications using our approach. The perceived usability of the approach was assessed using questionnaires. Finally, an interview was conducted with each participant to acquire further information on the programming experience.

7.3.2 Methodology

Participants

Our study involved four professional software developers with a background in the development of IoT applications and currently working for sensor manufacturing companies. The participants had different level of experience in software development, ranging from 0 to 7 years as depicted in Table 7.2. The study required the participants to have basic experience in Java programming and the usage of the Eclipse IDE. Three participants matched these criteria. We also recruited one participant who did not match any of the above stated criteria to examine whether a developer with no prior experience in Java programming/Eclipse usage finds the development support provided by our approach useful and easy to use. It is important to note that for this participant, only data collected from questionnaires and the interview have been taken into account in the evaluation process of our approach.

Training session

The training session giving the participants basic skills on rapid prototyping of orchestrating applications with our approach was dedicated to the development of an orchestrating application revolving around an HVAC (Heating, Ventilation, and Air Conditioning) system. The application development was guided by a DiaSuite expert, member of our group, and addressed all the development phases from design to testing. The training session lasted approximately four hours.

Object of the study

The study was dedicated to the development of the DoorLocks application presented in Figure 7.2. The DoorLocks application was specified by the Axible Technologies company and previously presented in Section 7.2. We recall that the goal of this application is to remotely monitor instances of electronic door locks, connected to the Internet through an onboard SIGFOX transmitter. The features specified were: real-time event logging, abnormal event alerts (stuck or forced door), and low battery state notifications. These features can be implemented by integrating in an orchestrating application the smart door locks, an email web service, a logging device, and mobile phones for SMS notifications.

Following the training session, the software engineering task evaluated in the study comprised the design, implementation and testing of this application. All the participants had precisely four hours to complete the task. Participants had access to the training material, the online help of the DiaSuite toolset and Eclipse, but no internet access was provided. At first, participants were asked to establish the design of the application. We enforced a limit of 30 minutes for this phase, with the possibility for us to provide a correct design after this period, in case the produced design was too complex or incorrect. This limit was imposed to avoid compromising the measurements of the subsequent phases.

Then, all the functionalities had to be implemented and tested one by one. The implementation of a specific functionality had to be validated via unit tests, which were given to the participants prior to the task assignment. An application functionality was considered complete upon the successful execution of all the corresponding tests. Participants who provided a valid application design were allowed to redefine their design if needed during the study.

Questionnaires

Participants filled out a first questionnaire assessing their experience in software development, Java programming and the usage of the Eclipse IDE. A standard System Usability Scale (SUS) [Brooke, 1996] questionnaire and a custom questionnaire evaluating our approach were handed out to the participants at the end of the study.

Interviews

The interviews with the participants were conducted to collect further information on the the programming experience with our design-driven development approach and to discuss the software development process and support currently used by these companies. We also discussed the possibility of using our approach for programming future, real-scale, IoT applications.

7.3.3 Experimental Results

The experimental results of the development task assigned to the participants are given in Table 7.2. The first two columns give the participants' experience in Java programming and the use of Eclipse in years. The next four columns give the time it took for each participant to complete the design phase, respectively the coding of each of the three application features, in minutes. The last two columns give the size of the code of the completed application, and the percentage of the manually written part. As the participant Dev0 has been included only for the questionnaire and interview, his development times were not measured (he was assisted to complete them, to sample functionalities of our approach). The other participants are generically called Dev1 to Dev3, in order of increasing experience in Java/Eclipse. The

Table 7.2 – Experimental results for the DoorLocks development task.

Developer	Java exp. (years)	Eclipse exp. (years)	Design (min)	Dev. feat1 (min)	Dev. feat2 (min)	Dev. feat3 (min)	Code size (LOC)	Man. part (%)
Dev0	0	0	N/A	N/A	N/A	N/A	N/A	N/A
Dev1	1	1	30	92	30	-	1671	10.4
Dev2	3	3	30*	55	15	5	2141	10.7
Dev3	7	7	27	87	18	9	1724	10.3
Expert	3	3	15	32	14	6	1760	10.2

last participant is an expert in our research group, used as a baseline for the development times.

From Table 7.2, we can see that the design phase lasted between 15 and 30 minutes for all the participants. To achieve our 30-minute time limit for this phase, we had to fix the design for only one participant, namely Dev2; his design was 80% correct and the corresponding time in Table 7.2 is marked by an asterisk.

Globally, the first feature of the DoorLocks application took most of the time to develop. This is due to the fact that participants changed the application design and returned back to the implementation more than once when working on the first feature. Also some helper classes had to be created only once for all the features. One participant did not finish the last feature on time. 89% of the the final application code was generated by the compiler for all the participants.

Overall, data in Table 7.2 constitutes a preliminary validation of our hypothesis that the initial cost of learning our approach, before starting to code real-world applications, is roughly half a day. Indeed, after such a training, two developers out of three completed the design phase, and the third one completed it at 80%; this represents an average completion rate of 93%. As for the implementation and testing phase, 89% (*i.e.,* 8 out of 9) of the application features were correctly developed and tested. Moreover, the approach appears easy to learn for developers with various levels of experience in Java and Eclipse, ranging from 1 to 7 years.

Questionnaires

The pre-experiment questionnaire revealed that three out of four participants used to define their software architecture in an informal way on a piece of paper and were skeptical about the utility of tools dedicated to software architecture and code generation. Only one participant had a positive opinion on the usefulness of such tools and used to define his software architecture via UML diagrams. Two participants had a prior experience with code generation tools, such as Rational Rose [IBM, 2016], AndroMDA [AndroMDA, 2016] and

ArgoUML [ArgoUML, 2016]. The post-experiment SUS questionnaire revealed that experienced developers found DiaSuite toolset more usable than developers with little or no experience in Java/Eclipse programming. Interestingly, Dev2 and Dev3's SUS questionnaires revealed an identical usability score of 67,5 points. Dev0 and Dev1 scores were 62,5 and 57,5 respectively. The overall SUS score of 63,75 indicates a usability level in the middle between OK and GOOD, which makes our approach a candidate for continued improvement to reach high acceptability [Bangor et al., 2008]. The post-experiment questionnaire investigated the perceived level of complexity related to design, implementation and testing using our approach, as well as its usefulness for coping with the challenges in development of orchestrating applications presented in Chapter 1, Section 1.2. The questionnaire revealed a unanimity on the application design being easily defined using our approach. Also, three out of four participants found application testing easy. Two participants (Dev3 and Dev0) considered the implementation phase easy, while the remaining ones had a neutral opinion. The usefulness of our approach for coping with heterogeneous APIs of smart objects has been confirmed by all participants. Three out of four participants confirmed that our approach is useful for rapid prototyping of orchestrating applications, thanks to the generated code support, which also greatly facilitates application testing. Half of the participants agreed that our approach can be useful for rapid evolution of applications, although the experiment did not involve this task. Interestingly, the participant having a positive opinion about software architecture tools prior to the experiment, considered our approach useful for the design of orchestrating applications. In contrast, the rest of the participants kept a neutral opinion. The questionnaire also revealed that the generated code was considered useful by three participants. This is a positive outcome of the experiment because only one participant found code generation tools useful prior to being exposed to our approach. Finally, participants expressed a unanimous appreciation of the integration of our approach with Eclipse via the DiaSuite toolset.

Interviews

In the following, we present key comments of our participants on various aspects of the proposed design-driven approach. We asked the participants about their assessment of the development process. One participant stated: *"Once the design is established, a big part of the work is already done"*. Another participant compared the design phase using DiaSpec with UML: *"DiaSpec appears to be simpler since it is more abstract. UML is much closer to the programming language"*. When asking participants about how the proposed design-driven approach could be used in their companies, one participant stated: *"It could be useful to cross multiple data sources when increasing the number of products. We could provide a solution by rapidly combining the different products we have, even with products from different manufacturers in an heterogeneous environment"*. We also asked participants what they think about the transfer of this approach to the industry. A participant stated: *"It's difficult to move to a new technology when you already master a more "traditional" one"*. Another participant pro-

posed to improve the approach as follows: *"The integration of the DiaSuite toolset with the Java EE platform and the Spring framework [Pivotal Software, 2016] would be a great asset".*

7.3.4 Threats to Validity

In this section we review the threats to the validity of our usability study results and the measures we took for avoiding them as much as possible.

Construct validity

These threats concern the adequacy of the experimental setup for measuring the desired outcome. The training required participants to develop a minimal application, called HVAC, that regulates the temperature of a room. As a result, one concern is whether this application resembles the one they have to develop autonomously. If they were similar, the participants could program the DoorLocks application by imitation based on the HVAC example, possibly without really understanding our approach. To address this threat, we intentionally introduced many differences in the DoorLocks application, including greater size (three times more components and three times more features) and several important differences in the interaction contracts, addressing typical pitfalls of a superficial assimilation of the approach. This definitely prevented participants from programming by imitation.

As a downside, these subtleties increased the risk for the participants to get lost during the design phase, and potentially end up with a design more complex than necessary. As the implementation is partially generated from the design, this increased the risk of not finishing the implementation on time. Thus, verifying the design ability could have compromised verifying the implementation ability, while both abilities are complementary indicators of a correct assimilation of our approach. To address this second construct threat, we introduced the time bound of 30 minutes for the application design.

Internal validity

These threats concern factors other than the control variables, which may influence the output variables. In our case, one potential perturbation could come from discomfort with the assigned workstation, as compared to the participant's usual computing environment, because of a different operating system, for instance. To avoid this threat, we asked participants to come with their own laptop computer, and installed the DiaSuite toolset on their machines.

External validity

These threats are factors that may invalidate the generalization of our result. The intended scope for our result is the industrial prototyping and development of orchestrating appli-

cations. This is why we selected the participants among professional developers within companies addressing the smart objects market. The first issue is that professional developers are a scarce resource for research experiments. This is why we could only include 4 participants, among which one participated just in the questionnaire and interview parts. The generalization of our result is thus limited by the representativeness of this very limited population sample. To mitigate this threat, we recruited the participants with various levels of experience, ranging from 0 to 7 years of experience in Java/Eclipse development project. This makes our result more likely to be generalized, but still constitutes only preliminary data, to be confirmed by a larger experiment.

Another external threat concerns the size of the software engineering task under test. Four hours of development is not negligible, but is small in comparison with a real-world development. Nevertheless, application prototyping is a very common task for sensor and smart object manufacturers, as confirmed by our interviews.

Finally, usability studies are frequently externally threatened by the fact that they occur in a closed world, where participants are isolated from the interruptions they handle every day in the real world. To alleviate this threat, we only isolated them during the training, but allowed some amount of interruptions during the development exercise. For example, one participant handled a phone call during 15 minutes, and another participant was preempted for a (real) work meeting in another room during 25 minutes.

7.4 Related Work

Languages and tools for sensor/actuator applications

Several other approaches provide languages and tools for developing applications orchestrating sensors and actuators, including IoT applications. For instance, several dedicated languages for describing smart object APIs and features have been defined, such as DomoML for the home automation domain [Miori and Russo, 2015] and the Puzzle building blocks [Danado and Paternò, 2015]. Some approaches provide graphical tools for composing applications out of modules, either by developers in Reactive Blocks [Kraemer and Herrmann, 2015], Dioptase [Billet and Issarny, 2015], and Compose [Raggett, 2015], or by end users in Puzzle [Danado and Paternò, 2015]. Boilerplate code for integrating reactive application modules is generated for instance in Reactive Blocks from UML component descriptions [Kraemer and Herrmann, 2015]. Support for testing IoT applications on simulated devices is provided by the Cooja network simulator [Eriksson et al., 2009; Osterlind et al., 2006].

While these approaches present several similarities with our design-driven development approach, none of them generates, starting from a device taxonomy and a specific application design, a customized programming framework that both guides and constraints the developer during all the application lifecycle, namely design, implementation, testing, deployment, and maintenance. We refer the reader to an existing paper [Bertran et al., 2012]

for a more detailed comparison between our approach and related approaches. On the other hand, we are not aware about empirical studies on these approaches involving professional developers, aimed at assessing learning cost and usability.

Experiments on using tools supporting software development

Vogel-Heusen [Vogel-Heuser, 2014] presents a usability study and seven usability experiments for evaluating the use of UML to increase the efficiency and quality in designing and maintaining software for manufacturing systems. Both generic UML versions such as UML 2.0 and domain-specific dialects such as SysML-AT are considered. A number of interesting aspects about experiment design are discussed. Maeder and Egyed [Mäder and Egyed, 2015] present a controlled experiment to measure the benefits of a source code navigation tool able to trace requirements, when used in software comprehension and software maintenance tasks. Hanenberg *et al.* perform controlled experiments to measure the benefits of using Aspect Oriented Programming (AOP) tools for developing crosscutting concerns. A more general overview of controlled experiments in software engineering can be found in a survey by Sjøberg *et al.* [Sjøberg et al., 2005]. According to their classification, our software engineering experiment has the following features: the task category is Create, with subcategories Design and Coding; the number of subjects falls in the Small category, professionals only; the task duration falls in the Large category; globally, the experiment size is then considered as Medium.

Finally, let us contrast this work with other works evaluating the usability of IDEs for developers, for instance those using various visual languages [Rouly et al., 2014]. First, DiaSpec is a textual language, although the application architecture can be visualized as a graph (*e.g.,* Figure 7.2). More importantly, our study investigates the usability of our approach, not of its IDE in particular. Indeed, our approach is integrated most naturally within the Eclipse IDE via the DiaSuite toolset, and it does not aim at changing the usability of Eclipse.

7.5 Lessons Learned

This section discusses some important lessons we learned during our experiments.

Learning cost

The usability study constitutes promising preliminary evidence that the DiaSuite toolset can be transferred to professional developers in the IoT domain. Indeed, with a minimal learning cost of half a day, developers are able to rapidly develop a typical prototype application.

Need for further experiments

A larger study is needed to confirm these findings on a statistically representative population, and to cover all the development phases addressed by our approach, including the deployment and maintenance phases.

Simple contexts help with the implementation, but may complicate testing

By comparing the different solutions of the developers to the assigned development task, we found that cutting application functionalities into several specialized contexts greatly simplifies the subsequent implementation phase. It also allows developers to anticipate many design decisions and uncover architecture defects earlier in the development process, before coding has even started. At the other end of the spectrum is an application architecture with a unique context, connected to all the data sources, and one controller connected to all actuator actions. This degenerated design leads to complex code, where all application features are intertwined. This architecture style postpones the identification of defects, increasing the cost of fixes. A much lighter version of this design error appeared in the solution of developer Dev2, who defined a single controller for two devices which mixed two completely independent application features. The manual and training should therefore be improved by adding some explicit design counter-examples to be avoided, clearly mentioning their drawbacks.

However, we found a limit to the benefits of modularization: when the application architecture graph becomes deeper, the test cases are more difficult to produce. Here, we identified a lack of support in our approach for simulating the output of context modules. This support should be added in a future version of the tools, to further encourage modular application structure.

Need of support for GUI development

Applications in the IoT frequently involve a graphical user interface (GUI) that is used for instance for displaying logs or alerts on a map. While our approach quite effectively helps in developing the application logic, there is no support for developing the GUI. In our four applications, we could reuse a common GUI, but some support for describing and partially generating a GUI customized for each application would have been helpful.

Lack of support for database interaction

Viewing databases as sensor and actuator entities is possible, but not always natural. More precisely, when a database is only used for output, such as for logging events, it is natural to declare it as an actuator entity (*e.g.,* in the PalletTracking application). Conversely, when a database is only used for input, such as querying a contact database, it is natural to declare it

as a source entity. However, when an application needs to interact with a database in several input and output phases, this leads to the declaration of multiple sources and actions for the same database, which seems contorted. In practice, one can implement such database interactions as side effects in context components, but this goes against the declarative nature of the DiaSpec language; some specific support for database interaction should be very helpful.

8

Conclusion

This chapter details the conclusions of this dissertation. We begin with a discussion on the different contributions we have presented throughout this dissertation. Then, we review the evaluation of the proposed design-driven approach by addressing the software engineering challenges of orchestrating masses of sensors, as identified in Chapter 1. Finally we discuss the limitations of our methodology and the avenues for future work.

Contents

Overview

- Concluding remarks on the proposed design-driven development approach.
- Current limitations of our approach and future work.

8.1 Discussion

In this dissertation, we have presented a design-driven methodology dedicated to the development of large-scale orchestrating applications. This methodology relies on the DiaSwarm domain-specific design language, which provides developers with constructs addressing the typical conceptual phases in the domain of large-scale orchestration, namely, service discovery, data gathering and data processing. Application design is used to produce design-specific code support that takes the form of a programming framework. Generated programming frameworks support and guide the implementation of orchestrating applications, while abstracting over specificities of the target network infrastructure.

Our approach relies on the MapReduce programming model to deal with large amounts of data collected from sensors. Design declarations describe when and where data processing occurs and are used to generate code support leveraging the Apache Hadoop platform. This strategy allows to introduce high-performance computing into the programming framework while abstracting over parallel processing of collected data. We evaluated the scalability of our approach by parallelizing computations over a cluster of nodes using the prototype implementation of our approach targeting the Hadoop platform.

We furthermore explored the design space and identified sensor-network characteristics of orchestrating applications carrying essential information to support the development process and to match application needs to the target sensor infrastructure. We have introduced stages along the application lifecycle and demonstrated how design declarations can be used to adapt both the application and the sensor infrastructure. We discussed how our work can be used to leverage existing approaches in the domain of sensor networks.

We carried out a thorough evaluation of our approach by means of two different experiments, in which we assessed the benefit of the design phase to drive the development of orchestrating applications. In a first experiment, we evaluated the effectiveness of our approach to deal with key software engineering challenges by implementing and examining four industrial case studies. In a second experiment, we carried out a usability study to evaluate the cost of learning our design-driven development approach by professional programmers with various levels of experience. The preliminary findings suggest that our approach can be used effectively by professional developers after only half a day of training.

We now review our approach in more detail with respect to key challenges identified in Chapter 1. We addressed the large scale orchestration challenges introduced in Chapter 1, Section 1.1 by providing developers with language constructs targeting the typical conceptual phases of large-scale orchestration.

The service discovery challenge has been addressed with high-level constructs allowing developers to group sensors into objects of interest according to domain-specific concepts. Service discovery results in the generation of code support that groups data according to design declarations and exposes collected data to developers via application-tailored data structures.

The challenge of gathering data from masses of sensors has been addressed with design support for models used by sensors to deliver data to applications. Generated programming frameworks gather data from sensors according to delivery models declared at design time.

The processing of large amounts of data gathered from sensors is ensured by the MapReduce programming model, which is made explicit at design time to generate code support for the parallel processing of sensed data. This support currently relies on the Apache Hadoop platform.

Finally, the generated programming support also ensures device actuation and allows devices to be actuated individually or globally according to their characteristics (*e.g.,* location).

Evaluation of design-driven software engineering

We have addressed the effectiveness of our design-driven development approach in dealing with the key software engineering challenges identified in Chapter 1, Section 1.2.

At design time, our approach provides developers with a classification comprising reusable descriptions of devices specific to a domain (*e.g.,* healthcare). This device taxonomy provides a generic interface for a given class of devices, which exposes sensing and actuation capabilities of a device as well as its characteristics (*e.g.,* id, location, *etc.*). These design declarations are leveraged by the compiler to provide the development phase with generic device interfaces that separate heterogeneous device implementations from the main application logic.

We have demonstrated that application design is of great importance to support the rapid development of orchestrating applications. The preliminary results obtained from the evaluation of our approach indicate that the vast majority of the resulting application code can be generated from design declarations. However, application case studies presented in our evaluation required only simple application logic. Thus further experiments entailing more complex application logic are needed to confirm these preliminary results.

The testing of orchestrating applications is greatly simplified by providing developers with mock entities generated from the device taxonomy. This support allows the application logic of orchestrating applications to be tested prior to an actual deployment. Furthermore, mock entities can be progressively substituted with physical devices and thus tested in hybrid environments.

We have introduced new features and modifications to case studies examined throughout the evaluation of our approach to estimate the effort related to application maintenance. Here too, design played an important role since some modifications only had an impact on the generated code and did not necessitate changes of the previously defined application logic. Furthermore, the integration of our approach with the Eclipse IDE has revealed to be useful in suggesting editing actions.

8.2 Ongoing and Future Work

Dealing with unbounded streams of data

In this dissertation, one of the challenges to achieve scalability consists of the efficient processing of large amounts of data collected from a sensor infrastructure. Another body of work in this direction is concerned with the processing of unbounded streams of data and has recently received much attention from the research community [Shahrivari, 2014; Madsen et al., 2013; Akidau et al., 2015; Boykin et al., 2014; Hirzel et al., 2013]. Indeed, sensors infrastructures producing high-frequency data streams create an additional bottleneck on the development of orchestrating applications. In future work, we plan to investigate how the processing of high-frequency data streams can be introduced into our design-driven development approach. This work will build upon the DiaSwarm language and introduce design constructs to allow the generation of code support dedicated to processing high-frequency data streams.

Reusable application components

Currently, our approach provides developers with a taxonomy of devices, which can be furthermore extended to create new devices comprising previously declared sensing and actuation capabilities. The reuse of devices is an important aspect of our approach since it allows devices to be shared among different applications. In the same spirit, the reuse of application components is an interesting subject to be addressed in future research. This line of work would allow the sharing or reuse of application logic among orchestrating applications. Often, applications reusing sensors and actuators implement the same algorithms for data filtering and aggregation. For example, a number of applications may rely on information such as the average temperature or the current pollution level in an area to achieve different objectives. The pallet tracking application examined in Chapter 7, Section 7.2 implements a context component that computes the location of devices via GPS triangulation. This component could be used by different applications tracking the position of pallets or other devices.

Addressing limitations of the SCC paradigm

The SCC (Sense/Compute/Control) paradigm is well suited for the design of orchestrating applications and provides a natural way for describing the sensing and actuation capabilities of devices. However, the paradigm is not well adapted for describing graphical user interfaces (GUIs) and databases. Indeed, it appears to be difficult to introduce a generic reusable GUI that could be easily shared among applications. Furthermore, databases rely on a specific data model, which currently has to be reflected in the design to make the development of applications using them more straightforward. Currently, to introduce user interfaces

and databases into the application design we typically rely on adhoc abstractions. However, nowadays, the integration of GUIs and databases with orchestrating applications is essential in domains such as the Internet of Things. We should address this issue in future work by providing extensions to the paradigm as well as dedicated constructs to our design language.

Simulation support for large-scale sensor infrastructures

This dissertation proposed a design-driven development approach covering the entire application lifecycle of orchestrating applications from design to runtime. Currently, the testing of orchestrating applications in our approach relies on mock entities used to replace physical sensors and actuators. The functionalities of an orchestrating application are tested via unitary tests, using few mock entities to test application data flow and control flow. A dedicated simulation support allowing applications to be tested against a real-size infrastructure of sensors and actuators would greatly improve application testing. Orchestrating applications could be thus tested under real traffic prior to their deployment on a target sensor network infrastructure. This line of work should leverage our generative programming approach to produce simulation support from DiaSwarm declarations. Furthermore, it should draw on previous work by Bruneau [Bruneau and Consel, 2012] to provide support where orchestrating applications can execute in a simulated environment without requiring any change to their code, thanks to our design-driven approach that abstracts over the underlying layers.

A

DiaSwarm grammar

```
DomainModel =
              (IncludeSpec
              | ActionDef
              | AbstractElement)*;

AbstractElement =
              StructDef
              | EnumDef
              | DeviceDef
              | ContextDef
              | ControllerDef;

IncludeSpec = 'include' STRING;

StructDef = 'structure' ID '{' (StructFieldDef ';')+ '}';

StructFieldDef = ID as DataTypeRef;

EnumDef = 'enumeration' ID '{' ID (',' ID)* '}';

VariableDef = ID as DataTypeRef;

DataTypeRef =
              (StructDef
              | EnumDef
              | PrimitiveTypeRef) (ListTag)?;

ListTag = '[]';
```

```
PrimitiveTypeRef =
                'Integer'
              | 'Boolean'
              | 'String'
              | 'Float'
              | 'Binary';

DeviceDef = 'device' ID ('extends' DeviceDef)?
'{' (AttributeDef | SourceDef | ActionImpl)* '}';

AttributeDef = 'attribute' ID 'as' DataTypeRef ';' ;

SourceDef = 'source' ID 'as' DataTypeRef
('indexed' 'by' VariableDef (',' VariableDef)*)? ';' ;

ActionImpl = 'action' ActionDef ';' ;

ActionDef = 'action' ID '{' (OrderDef)+ '}';

OrderDef = ID '('(VariableDef(',' VariableDef)*)?')' ';' ;

ContextDef = 'context' ID 'as' DataTypeRef
('indexed' 'by' VariableDef (',' VariableDef)*)?
       '{' (InteractionContractDef)* '}';

InteractionContractDef =
       'when'
         ('required' (PullInteractionContract)?
       | ('provided' PushInteractionContract BehaviorPublication)
       | ('periodic' PushInteractionContract BehaviorPublication)) ';';

PullInteractionContract = ('get'
              (SourceRef | ContextDef))?
              (',' (SourceRef | ContextDef))*;

PushInteractionContract = DataRequirement
              (PullInteractionContract)?;

DataRequirement = DataSource
    ('grouped' 'by' AttributeDef
    ('every' Periodicity)?)?
    ('with'
     (('map' 'as' (DataTypeRef | '<'DataTypeRef ','
                                 DataTypeRef '>'))?
    &
     ('reduce' 'as' DataTypeRef)?))?;

DataSource = SourceRef | ContextDef (Periodicity)?;
```

```
SourceRef = SourceDef 'from' DeviceDef;

Periodicity = '<' INT TimeUnit '>';

BehaviorPublication = 'always' 'publish'
                    | 'no' 'publish'
                    | 'maybe' 'publish';

TimeUnit = 'hr' | 'min' | 's';

ControllerDef = 'controller' ID
              '{' (ControllerBehaviorDef)* '}' ;

ControllerBehaviorDef = 'when' 'provided' ContextDef
              ('get' ContextDef (',' ContextDef)*)?
              'do' ActionRef (',' ActionRef)* ';' ;

ActionRef = ActionDef 'on' DeviceDef;

ID = '^'?('a'..'z'|'A'..'Z'|'_')
         ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

INT = ('0'..'9')+;

STRING =
  '"' ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */
    | !('\\'|'"') )* '"' |
  "'" ( '\\' . /* 'b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\' */
    | !('\\'|"'") )* "'";
```

List of Figures

List of Tables

List of Listings

Bibliography

Abbate, S., Avvenuti, M., Bonatesta, F., Cola, G., Corsini, P., and Vecchio, A. (2012). A smartphone-based fall detection system. *Pervasive and Mobile Computing*, 8(6):883 – 899. Special Issue on Pervasive Healthcare. (see page 17)

Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., Luo, L., Son, S., Stankovic, J., Stoleru, R., and Wood, A. (2004). EnviroTrack: towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 582–589. (see page 15)

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., and Whittle, S. (2015). The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing. *Proc. VLDB Endow.*, 8(12):1792–1803. (see page 106)

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., and Ayyash, M. (2015). Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376. (see page 19)

Alegre, U., Augusto, J. C., and Clark, T. (2016). Engineering context-aware systems and applications: A survey. *Journal of Systems and Software*, 117:55–83. (see page 13)

AndroMDA (2016). *Generate components quickly with AndroMDA*. URL http://www.andromda.org. Accessed March 2016. (see page 95)

Apache (2015). *PoweredBy - Hadoop Wiki*. URL http://wiki.apache.org/hadoop/PoweredBy. Accessed March 2016. (see page 47)

Apache Software Foundation (2016a). *Apache Hive*. URL https://hive.apache.org. Accessed March 2016. (see pages 42 and 54)

Apache Software Foundation (2016b). *Apache Pig*. URL https://pig.apache.org. Accessed March 2016. (see pages 42 and 54)

ArgoUML (2016). *Welcome to ArgoUML*. URL http://argouml.tigris.org. Accessed March 2016. (see page 96)

Atzori, L., Iera, A., and Morabito, G. (2010). The Internet of Things: A Survey. *Comput. Netw.*, 54(15):2787–2805. (see page 19)

Baccelli, E. and Raggett, D. (2015). The Promise of the Internet of Things and the Web of Things - Introduction to the Special Theme. *ERCIM News*, 101:8–10. (see pages 4 and 91)

Bangor, A., Kortum, P. T., and Miller, J. T. (2008). An Empirical Evaluation of the System Usability Scale. *International Journal of Human-Computer Interaction*, 24(6):574–594. (see page 96)

Bertran, B., Bruneau, J., Cassou, D., Loriant, N., Balland, E., and Consel, C. (2012). DiaSuite: a Tool Suite To Develop Sense/Compute/Control Applications. *Science of Computer Programming*. (see pages 13, 77, and 98)

Bertran, B., Consel, C., Kadionik, P., and Lamer, B. (2009). A SIP-Based Home Automation Platform: An Experimental Study. In *13th International Conference on Intelligence in Next Generation Networks*, pages 1–6, Bordeaux, France. IEEE. (see page 78)

Billet, B. and Issarny, V. (2015). Dioptase: Data Streaming Middleware for the Internet of Things. *ERCIM News*, 101:23–24. (see page 98)

Bonnet, P., Gehrke, J., and Seshadri, P. (2000). Querying the physical world. *IEEE Personal Communications*, 7(5):10–15. (see page 15)

Borcea, C., Intanagonwiwat, C., Kang, P., Kremer, U., and Iftode, L. (2004). Spatial programming using smart messages: design and implementation. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 690–699. (see page 16)

Borgia, E. (2014). The internet of things vision: Key features, applications and open issues. *Computer Communications*, 54:1 – 31. (see page 19)

Boulis, A., Han, C.-C., and Srivastava, M. B. (2003). Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 187–200, New York, NY, USA. ACM. (see page 14)

Boykin, O., Ritchie, S., O'Connell, I., and Lin, J. (2014). Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. *Proc. VLDB Endow.*, 7(13):1441–1451. (see page 106)

Brooke, J. (1996). SUS: a "quick and dirty" usability scale. In *Usability Evaluation in Industry*. Taylor and Francis, London. (see page 94)

Bruneau, J. and Consel, C. (2012). DiaSim: A Simulator for Pervasive Computing Applications. *Software: Practice and Experience*. (see page 107)

Bruneau, J., Jouve, W., and Consel, C. (2009). DiaSim: A Parameterized Simulator for Pervasive Computing Applications. In *6th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous'09)*, Toronto, Canada. IEEE. (see page 78)

Caroux, L., Consel, C., Dupuy, L., and Sauzéon, H. (2014). Verification of Daily Activities of Older Adults: A Simple, Non-Intrusive, Low-Cost Approach. In *ASSETS - The 16th International ACM SIGACCESS Conference on Computers and Accessibility*, pages 43–50, Rochester, NY, United States. (see pages 17 and 78)

Cassou, D., Balland, E., Consel, C., and Lawall, J. (2011a). Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications. In ICSE '11. (see pages 13, 30, and 78)

Cassou, D., Bruneau, J., Consel, C., and Balland, E. (2012). Towards a Tool-based Development Methodology for Pervasive Computing Applications. *IEEE TSE: Transactions on Software Engineering*, 38(6):1445–1463. (see pages 4, 28, 30, 36, 77, and 111)

Cassou, D., Stinckwich, S., and Koch, P. (2011b). Using the DiaSpec design language and compiler to develop robotics systems. In *2nd International Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob-11)*. IEEE Computer Society. (see page 78)

Chambers, C., Raniwala, A., Perry, F., Adams, S., Henry, R. R., Bradshaw, R., and Weizenbaum, N. (2010). FlumeJava: Easy, Efficient Data-parallel Pipelines. PLDI '10, pages 363–375. ACM. (see pages 42 and 54)

Chen, G. and Kotz, D. (2002). Context aggregation and dissemination in ubiquitous computing systems. In *Proceedings Fourth IEEE Workshop on Mobile Computing Systems and Applications, 2002*, pages 105–114. (see page 13)

Cugola, G. and Margara, A. (2012). Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.*, 44(3):15:1–15:62. (see page 50)

Danado, J. and Paternò, F. (2015). A Mobile End-User Development Environment for IoT Applications Exploiting the Puzzle Metaphor. *ERCIM News*, 101:26–27. (see page 98)

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113. (see pages 6, 43, and 45)

Delicato, F. C., Pires, P. F., Pirmez, L., and Carmo, L. F. R. d. C. (2005). A Service Approach for Architecting Application Independent Wireless Sensor Networks. *Cluster Computing*, 8(2):211–221. (see page 70)

Dey, A. K. (2001). Understanding and Using Context. *Personal Ubiquitous Comput.*, 5(1):4–7. (see page 12)

Dey, A. K., Abowd, G. D., and Salber, D. (2001). A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Hum.-Comput. Interact.*, 16(2):97–166. (see page 12)

Eltarras, R. and Eltoweissy, M. (2010). Adaptive Multi-Criteria Routing for Shared Sensor-Actuator Networks. In *Proceedings of the Global Communications Conference. GLOBECOM 2010, 6-10 December 2010, Miami, Florida, USA*, pages 1–6. IEEE. (see page 73)

Enard, Q., Gatti, S., Bruneau, J., Moon, Y.-J., Balland, E., and Consel, C. (2013a). Design-driven Development of Dependable Applications: A Case Study in Avionics. In *PECCS*. (see pages 13 and 78)

Enard, Q., Stoicescu, M., Balland, E., Consel, C., Duchien, L., Fabre, J.-C., and Roy, M. (2013b). Design-Driven Development Methodology for Resilient Computing. In *CBSE'13*. (see page 13)

Endres, C., Butz, A., and MacWilliams, A. (2005). A Survey of Software Infrastructures and Frameworks for Ubiquitous Computing. *Mobile Information Systems*, 1(1):41–80. (see page 13)

Eriksson, J., Österlind, F., Finne, N., Tsiftes, N., Dunkels, A., Voigt, T., Sauter, R., and Marrón, P. J. (2009). COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 27:1–27:7, ICST, Brussels, Belgium, Belgium. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). (see page 98)

Estrin, D., Govindan, R., Heidemann, J., and Kumar, S. (1999). Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '99, pages 263–270, New York, NY, USA. ACM. (see pages 60 and 61)

Farias, C. M. D., Li, W., Delicato, F. C., Pirmez, L., Zomaya, A. Y., Pires, P. F., and Souza, J. N. D. (2016). A Systematic Review of Shared Sensor Networks. *ACM Comput. Surv.*, 48(4):51:1–51:50. (see page 72)

Fayad, M. and Schmidt, D. C. (1997). Object-oriented application frameworks. *Commun. ACM*, 40(10):32–38. (see pages 6 and 27)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc. (see page 39)

Gatti, S., Balland, E., and Consel, C. (2011). A Step-wise Approach for Integrating QoS throughout Software Development. In *FASE'11: Proceedings of the 14th European Conference on Fundamental Approaches to Software Engineering*, volume 6603 of *Lecture Notes in Computer Science*, pages 217–231, Sarrebruck, Germany. Springer. (see page 67)

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. (2003). The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA. ACM. (see page 14)

Girolami, M., Furfari, F., and Chessa, S. (2015). COMPOSE: An Open Source Cloud-Based Scalable IoT Services Platform. *ERCIM News*, 101:31–32. (see page 91)

Gluhak, A., Krco, S., Nati, M., Pfisterer, D., Mitton, N., and Razafindralambo, T. (2011). A survey on facilities for experimental internet of things research. *IEEE Communications Magazine*, 49(11):58–67. (see page 17)

Greenstein, B., Kohler, E., and Estrin, D. (2004). A Sensor Network Application Construction Kit (SNACK). In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, SenSys '04, pages 69–80, New York, NY, USA. ACM. (see page 14)

Guinard, D., Trifa, V., Karnouskos, S., Spiess, P., and Savio, D. (2010). Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services. *IEEE Transactions on Services Computing*, 3(3):223–235. (see page 19)

Gummadi, R., Gnawali, O., and Govindan, R. (2005). Macro-programming Wireless Sensor Networks Using Kairos. In *Proceedings of the First IEEE International Conference on Distributed Computing in Sensor Systems*, DCOSS'05, pages 126–140, Berlin, Heidelberg. Springer-Verlag. (see page 15)

Gupta, V., Tovar, E., Pinho, L. M., Kim, J., Lakshmanan, K., and Rajkumar, R. (2011). sMapReduce: A Programming Pattern for Wireless Sensor Networks. In SESENA '11. (see page 54)

Gyrard, A., Datta, S. K., Bonnet, C., and Boudaoud, K. (2015). Cross-Domain Internet of Things Application Development: M3 Framework and Evaluation. In *3rd International Conference on Future Internet of Things and Cloud (FiCloud), 2015*, pages 9–16. (see page 18)

Hachem, S., Pathak, A., and Issarny, V. (2014). Service-Oriented Middleware for Large-Scale Mobile Participatory Sensing. *Pervasive and Mobile Computing*, 10:66–82. (see page 19)

Hashem, I. A. T., Chang, V., Anuar, N. B., Adewole, K., Yaqoob, I., Gani, A., Ahmed, E., and Chiroma, H. (2016). The role of big data in smart city. *International Journal of Information Management*, 36(5):748 – 758. (see page 19)

Heidemann, J., Silva, F., and Estrin, D. (2003). Matching Data Dissemination Algorithms to Application Requirements. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 218–229, New York, NY, USA. ACM. (see pages 2, 61, and 69)

Heidemann, J., Silva, F., Intanagonwiwat, C., Govindan, R., Estrin, D., and Ganesan, D. (2001). Building Efficient Wireless Sensor Networks with Low-level Naming. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 146–159, New York, NY, USA. ACM. (see pages 60 and 61)

Heinzelman, W. B. (2000). *Application-specific Protocol Architectures for Wireless Networks*. PhD thesis, Cambridge, MA, USA. (see page 61)

Henricksen, K. and Indulska, J. (2004). A software engineering framework for context-aware pervasive computing. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications, 2004. PerCom 2004*, pages 77–86. (see page 12)

Henricksen, K. and Indulska, J. (2006). Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing*, 2(1):37–64. (see page 12)

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. (2000). System Architecture Directions for Networked Sensors. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 93–104, New York, NY, USA. ACM. (see page 14)

Hirzel, M., Andrade, H., Gedik, B., Jacques-Silva, G., Khandekar, R., Kumar, V., Mendell, M., Nasgaard, H., Schneider, S., Soulé, R., and Wu, K.-L. (2013). IBM Streams Processing Language: Analyzing Big Data in Motion. *IBM J. Res. Dev.*, 57(3-4):1:7–1:7. (see page 106)

Hughes, D., Thoelen, K., Horré, W., Matthys, N., Cid, J. D., Michiels, S., Huygens, C., and Joosen, W. (2009). LooCI: A Loosely-coupled Component Infrastructure for Networked Embedded Systems. In *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '09, pages 195–203, New York, NY, USA. ACM. (see page 72)

IBM (2013). *Dutch City Region of Eindhoven Works with IBM and NXP to Improve Traffic Flow and Road Safety*. URL http://www-03.ibm.com/press/us/en/pressrelease/40212.wss. Accessed March 2016. (see page 1)

IBM (2016). *Rational Rose family*. URL http://www-03.ibm.com/software/products/en/ratirosefami. Accessed March 2016. (see page 95)

Kabáč, M. and Consel, C. (2015). Orchestrating Masses of Sensors: A Design-Driven Development Approach. In *14th International Conference on Generative Programming: Concepts & Experience (GPCE'15)*, Pittsburgh, Pennsylvania, United States. (see pages 28 and 59)

Kabáč, M. and Consel, C. (2016). Designing Parallel Data Processing for Large-Scale Sensor Orchestration. In *13th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC 2016)*, Toulouse, France. (see pages 42 and 59)

Kabáč, M., Consel, C., and Volanschi, N. (2016). Leveraging Declarations over the Lifecycle of Large-Scale Sensor Applications. In *13th IEEE International Conference on Ubiquitous Intelligence and Computing (UIC 2016)*, Toulouse, France. (see page 58)

Kabáč, M., Volanschi, N., and Consel, C. (2015). An Evaluation of the DiaSuite Toolset by Professional Developers: Learning Cost and Usability. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2015, pages 9–16, New York, NY, USA. ACM. (see pages 59 and 92)

Karnouskos, S. (2011). Cyber-Physical Systems in the SmartGrid. In *9th IEEE International Conference on Industrial Informatics*, pages 20–23. (see page 17)

Kolcun, R., Boyle, D., and McCann, J. A. (2015). Optimal processing node discovery algorithm for distributed computing in IoT. In *5th International Conference on the Internet of Things (IOT), 2015*, pages 72–79. (see page 19)

Kolcun, R. and McCann, J. A. (2014). Dragon: Data discovery and collection architecture for distributed IoT. In *International Conference on the Internet of Things (IOT), 2014*, pages 91–96. (see page 19)

Kortuem, G., Kawsar, F., Sundramoorthy, V., and Fitton, D. (2010). Smart Objects As Building Blocks for the Internet of Things. *IEEE Internet Computing*, 14(1):44–51. (see pages 17 and 28)

Kraemer, F. A. and Herrmann, P. (2015). Creating Internet of Things Applications from Building Blocks. *ERCIM News*, 101:19–20. (see page 98)

Kranz, M., Holleis, P., and Schmidt, A. (2010). Embedded Interaction: Interacting with the Internet of Things. *IEEE Internet Computing*, 14(2):46–53. (see page 17)

Krishnamachari, B. and Heidemann, J. (2004). Application-specific modelling of information routing in wireless sensor networks. In *IEEE Performance, Computing, and Communications*. (see page 2)

Lämmel, R. (2008). Google's MapReduce programming model - Revisited. *Science of Computer Programming*, 70(1):1–30. (see pages 6, 43, and 45)

Lee, E. A., Rabaey, J., Blaauw, D., Fu, K., Guestrin, C., Hartmann, B., Jafari, R., Jones, D., Kubiatowicz, J., Kumar, V., Mangharam, R., Murray, R., Pappas, G., Pister, K., Rowe, A., Sangiovanni-Vincentelli, A., Seshia, S. A., Rosing, T. S., Taskar, B., Wawrzynek, J., and Wessel, D. (2014). The Swarm at the Edge of the Cloud. *Design & Test, IEEE*, 31(3):1–13. (see page 72)

Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y. D., and Moon, B. (2012). Parallel Data Processing with MapReduce: A Survey. *SIGMOD Rec.*, 40(4):11–20. (see pages 3 and 42)

Levis, P. and Culler, D. (2002). MatÉ: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 85–95, New York, NY, USA. ACM. (see page 14)

Levis, P., Gay, D., and Culler, D. (2005). Active Sensor Networks. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 343–356, Berkeley, CA, USA. USENIX Association. (see page 14)

libelium (2012). *Smart Agriculture project in Galicia to monitor vineyards with Waspmote*. URL http://www.libelium.com/smart_agriculture_vineyard_sensors_waspmote. Accessed June 2016. (see page 17)

libelium (2013). *Smart City project in Santander to monitor Parking Free Slots*. URL http://www.libelium.com/smart_santander_parking_smart_city. Accessed March 2016. (see pages 1, 17, and 22)

Links, Cees (2015). *The Internet of Things will Change our World. ERCIM News*, 101:3. (see pages 76 and 91)

Liu, T. and Martonosi, M. (2003). Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 107–118, New York, NY, USA. ACM. (see page 14)

Liu, X., Huang, Q., and Zhang, Y. (2007). Balancing Push and Pull for Efficient Information Discovery in Large-Scale Sensor Networks. *Mobile Computing, IEEE Transactions on*, 6(3):241–251. (see pages 2, 3, 61, 70, and 73)

M2M World News (2015). *Samsung Adds SIGFOX Internet of Things Protocol to Its New Samsung ARTIK Platform and Invests in IoT Pioneer*. URL http://m2mworldnews.com/2015/06/15/77001-samsung-adds-sigfox-internet-of-things-

protocol-to-its-new-samsung-artik-platform-and-invests-in-iot-pioneer. Accessed June 2016. (see page 76)

Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W. (2003). The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 491–502, New York, NY, USA. ACM. (see page 15)

Mäder, P. and Egyed, A. (2015). Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, 20(2):413–441. (see page 99)

Madria, S., Kumar, V., and Dalvi, R. (2014). Sensor Cloud: A Cloud of Virtual Sensors. *IEEE Software*, 31(2):70–77. (see page 69)

Madsen, K. G. S., Su, L., and Zhou, Y. (2013). Grand Challenge: MapReduce-style Processing of Fast Sensor Data. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 313–318, New York, NY, USA. ACM. (see page 106)

Manyika, J., Chui, M., Bisson, P., Woetzel, J., Dobbs, R., Bughin, J., and Aharon, D. (2015). The Internet of Things: Mapping the value beyond the hype. URL http://www.mckinsey.com/business-functions/business-technology/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world. Accessed June 2016. (see pages 17 and 42)

Mercadal, J., Enard, Q., Consel, C., and Loriant, N. (2010). A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing. In *OOPSLA: Conference on Object Oriented Programming Systems Languages and Applications*, Reno, United States. (see page 67)

Meshkova, E., Riihijärvi, J., Petrova, M., and Mähönen, P. (2008). A Survey on Resource Discovery Mechanisms, Peer-to-peer and Service Discovery Frameworks. *Comput. Netw.*, 52(11):2097–2128. (see page 60)

Miorandi, D., Sicari, S., Pellegrini, F. D., and Chlamtac, I. (2012). Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516. (see page 19)

Miori, V. and Russo, D. (2015). Home Automation Devices Belong to the IoT World. *ERCIM News*, 101:22–23. (see pages 91 and 98)

Mizuno, Y. and Odake, N. (2015). Current status of smart systems and case studies of privacy protection platform for smart city in Japan. In *Management of Engineering and Technology (PICMET), 2015 Portland International Conference on*, pages 612–624. (see page 1)

Mottola, L. and Picco, G. P. (2006). Logical Neighborhoods: A Programming Abstraction for Wireless Sensor Networks. In *Proceedings on Distributed Computing in Sensor Systems, Second IEEE International Conference, DCOSS 2006, San Francisco, CA, USA, June 18-20, 2006*, pages 150–168. (see page 15)

Mottola, L. and Picco, G. P. (2011). Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art. *ACM Comput. Surv.*, 43(3):19:1–19:51. (see pages 16 and 54)

Naphade, M., Banavar, G., Harrison, C., Paraszczak, J., and Morris, R. (2011). Smarter Cities and Their Innovation Challenges. *Computer*, 44(6):32–39. (see page 1)

Nastic, S., Sehic, S., Vögler, M., Truong, H. L., and Dustdar, S. (2013). PatRICIA – A Novel Programming Model for IoT Applications on Cloud Platforms. In *IEEE 6th International Conference on Service-Oriented Computing and Applications, 2013*, pages 53–60. (see page 18)

National Intelligence Council (2008). *Disruptive Civil Technologies – Six Technologies with Potential Impacts on US Interests*. URL https://fas.org/irp/nic/disruptive.pdf. Conference Report CR 2008-07. (see page 17)

Newton, R., Morrisett, G., and Welsh, M. (2007). The Regiment Macroprogramming System. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 489–498, New York, NY, USA. ACM. (see page 15)

Nguyen, X. T., Tran, H. T., Baraki, H., and Geihs, K. (2015). FRASAD: A framework for model-driven IoT Application Development. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, pages 387–392. (see page 17)

Ni, Y., Kremer, U., Stere, A., and Iftode, L. (2005). Programming Ad-hoc Networks of Mobile and Resource-constrained Devices. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2005*, PLDI '05, pages 249–260, New York, NY, USA. ACM. (see page 16)

Objects World Consortium (2012). *Objects World project*. URL http://phoenix.inria.fr/research-projects/objects-world. Accessed March 2016. (see pages 76, 80, and 92)

Osterlind, F., Dunkels, A., Eriksson, J., Finne, N., and Voigt, T. (2006). Cross-Level Sensor Network Simulation with COOJA. In *Proceedings of the the 31st IEEE Conference on Local Computer Networks, 2006*, pages 641–648. (see page 98)

Patel, P. and Cassou, D. (2015). Enabling high-level application development for the Internet of Things. *Journal of Systems and Software*, 103:62 – 84. (see page 18)

Patel, P., Pathak, A., Cassou, D., and Issarny, V. (2013). Enabling High-Level Application Development in the Internet of Things. In *S-CUBE'13*, Lucca, Italy. (see page 18)

Perera, C., Zaslavsky, A., Christen, P., and Georgakopoulos, D. (2014). Context Aware Computing for The Internet of Things: A Survey. *IEEE Communications Surveys Tutorials*, 16(1):414–454. (see pages 17 and 18)

Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. (2005). Interpreting the Data: Parallel Analysis with Sawzall. *Sci. Program.*, 13(4):277–298. (see page 54)

Pivotal Software (2016). *Spring Framework*. URL http://projects.spring.io/spring-framework. Accessed March 2016. (see page 97)

Raggett, D. (2015). COMPOSE: An Open Source Cloud-Based Scalable IoT Services Platform. *ERCIM News*, 101:30–31. (see pages 91 and 98)

Raman, B. and Chebrolu, K. (2008). Censor Networks: A Critique of "Sensor Networks" from a Systems Perspective. *SIGCOMM Comput. Commun. Rev.*, 38(3):75–78. (see page 2)

Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R. H., and Mickunas, M. D. (2005). Olympus: A High-Level Programming Model for Pervasive Computing Environments. In PERCOM '05. (see page 13)

RFID Journal (2009). *That 'Internet of Things' Thing*. URL http://www.rfidjournal.com/articles/view?4986. Accessed June 2016. (see page 17)

Rigzone (2014). *Internet of Things Technologies Could Transform Oil, Gas Industry*. URL http://www.rigzone.com/news/oil_gas/a/134738/Internet_of_Things_Technologies_Could_Transform_Oil_Gas_Industry/?all=HG2. Accessed June 2016. (see pages 6 and 42)

Riliskis, L., Hong, J., and Levis, P. (2015). Ravel: Programming IoT Applications As Distributed Models, Views, and Controllers. In *Proceedings of the 2015 International Workshop on Internet of Things Towards Applications*, IoT-App '15, pages 1–6, New York, NY, USA. ACM. (see page 18)

Rouly, J. M., Orbeck, J. D., and Syriani, E. (2014). Usability and Suitability Survey of Features in Visual Ides for Non-Programmers. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '14, pages 31–42, New York, NY, USA. ACM. (see page 99)

Saha, D. and Mukherjee, A. (2003). Pervasive computing: a paradigm for the 21st century. *Computer*, 36(3):25–31. (see page 12)

Sanchez, L., Muñoz, L., Galache, J. A., Sotres, P., Santana, J. R., Gutierrez, V., Ramdhany, R., Gluhak, A., Krco, S., Theodoridis, E., and Pfisterer, D. (2014). SmartSantander: IoT experimentation over a smart city testbed. *Computer Networks*, 61:217 – 238. Special issue on Future Internet Testbeds – Part I. (see page 22)

Sehic, S., Li, F., Nastic, S., and Dustdar, S. (2012). A programming model for context-aware applications in large-scale pervasive systems. In *IEEE 8th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob), 2012*, pages 142–149. (see page 13)

Seitz, L., Selander, G., and Gehrmann, C. (2013). Authorization framework for the Internet-of-Things. In *14th International Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE Computer Society. (see page 91)

Serral, E., Valderas, P., and Pelechano, V. (2010). Towards the Model Driven Development of context-aware pervasive systems. *Pervasive Mob. Comput.*, 6(2):254–280. (see page 13)

Shahrivari, S. (2014). Beyond Batch Processing: Towards Real-Time and Streaming Big Data. *CoRR*, abs/1403.3375. (see page 106)

Sharaf, M. A., Beaver, J., Labrinidis, A., and Chrysanthis, P. K. (2003). TiNA: A Scheme for Temporal Coherency-aware In-network Aggregation. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDe '03, pages 69–76, New York, NY, USA. ACM. (see page 70)

Sheng, Z., Yang, S., Yu, Y., Vasilakos, A. V., Mccann, J. A., and Leung, K. K. (2013). A survey on the ietf protocol suite for the internet of things: standards, challenges, and opportunities. *IEEE Wireless Communications*, 20(6):91–98. (see page 19)

Sheth, A., Henson, C., and Sahoo, S. S. (2008). Semantic Sensor Web. *IEEE Internet Computing*, 12(4):78–83. (see page 18)

Sivieri, A., Mottola, L., and Cugola, G. (2016). Building Internet of Things software with ELIoT. *Computer Communications*, pages –. (see page 17)

Sjøberg, D. I. K., Hannay, J. E., Hansen, O., By Kampenes, V., Karahasanovic, A., Liborg, N.-K., and C. Rekdal, A. (2005). A Survey of Controlled Experiments in Software Engineering. *IEEE Trans. Softw. Eng.*, 31(9):733–753. (see page 99)

Stankovic, J. A. (2014). Research Directions for the Internet of Things. *IEEE Internet of Things Journal*, 1(1):3–9. (see page 19)

Sugihara, R. and Gupta, R. K. (2008). Programming models for sensor networks: A survey. *ACM Transactions on Sensor Networks (TOSN)*, 4(2):8. (see page 14)

Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing. (see pages 27 and 30)

Teixeira, T., Hachem, S., Issarny, V., and Georgantas, N. (2011). Service Oriented Middleware for the Internet of Things: A Perspective. In *Proceedings of the 4th European Conference on Towards a Service-based Internet*, ServiceWave'11, pages 220–229, Berlin, Heidelberg. Springer-Verlag. (see page 19)

Tilak, S., Abu-Ghazaleh, N. B., and Heinzelman, W. (2002). A Taxonomy of Wireless Microsensor Network Models. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(2):28–36. (see pages 32 and 61)

Uthra, R. A. and Raja, S. V. K. (2012). QoS Routing in Wireless Sensor Networks - a survey. *ACM Comput. Surv.*, 45(1):9:1–9:12. (see page 39)

Vallati, C., Mingozzi, E., and Tanganelli, G. (2015). BETaaS: Building the Future Platform for Development and Execution of Machine-to-Machine Applications. *ERCIM News*, 101:36–37. (see page 91)

Van Der Walt, P. (2015). *A language-independent methodology for compiling declarations into open platform frameworks*. Phd thesis, Université de Bordeaux. (see page 91)

Verma, P. K., Verma, R., Prakash, A., Agrawal, A., Naik, K., Tripathi, R., Alsabaan, M., Khalifa, T., Abdelkader, T., and Abogharaf, A. (2016). Machine-to-Machine (M2m) communications: A survey. *Journal of Network and Computer Applications*, 66:83–105. (see page 4)

Vogel-Heuser, B. (2014). Usability Experiments to Evaluate UML/SysML-Based Model Driven Software Engineering Notations for Logic Control in Manufacturing Automation. *Journal of Software Engineering and Applications*, 7:943–973. (see page 99)

Warmer, J. and Kleppe, A. (1998). *The Object Constraint Language: Precise Modeling With Uml (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional. (see page 13)

Weiser, M. (1991). The Computer for the 21st Century. *Scientific American*, 265(3):94–104. (see page 12)

Welsh, M. and Mainland, G. (2004). Programming Sensor Networks Using Abstract Regions. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 3–3, Berkeley, CA, USA. USENIX Association. (see page 15)

White, T. (2012). *Hadoop: The Definitive Guide*. O'Reilly Media, Inc. (see page 48)

Worldsensing (2014). *Worldsensing and SIGFOX announce the world's largest Intelligent Parking deployment with Micronet, the SIGFOX Network Operator for Russia*. URL http://www.worldsensing.com/news-press/press-release-worldsensing-and-sigfox-

announce-the-worlds-largest-intelligent-parking-deployment-with-micronet-the-sigfox-network-operator-for-russia.html. Accessed March 2016. (see pages 17 and 22)

Wu, G., Talwar, S., Johnsson, K., Himayat, N., and Johnson, K. D. (2011). M2M: From mobile to embedded internet. *IEEE Communications Magazine*, 49(4):36–43. (see page 17)

Ye, F., Luo, H., Lu, S., and Zhang, L. (2004). Wireless sensor networks. chapter Dissemination Protocols for Large Sensor Networks, pages 109–128. Kluwer Academic Publishers, Norwell, MA, USA. (see page 61)

Yuriyama, M. and Kushida, T. (2010). Sensor-Cloud Infrastructure - Physical Sensor Management with Virtualized Sensors on Cloud Computing. In *13th International Conference on Network-Based Information Systems (NBiS), 2010*, pages 1–8. (see page 19)

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. (2012). Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In NSDI'12. (see page 50)

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster Computing with Working Sets. In HotCloud'10. (see page 50)

Zhu, F., Mutka, M. W., and Ni, L. M. (2005). Service Discovery in Pervasive Computing Environments. *IEEE Pervasive Computing*, 4(4):81–90. (see pages 2 and 34)

Résumé

Notre environnement est de plus en plus peuplé de grandes quantités d'objets intelligents. Certains surveillent des places de stationnement disponibles, d'autres analysent les conditions matérielles dans les bâtiments ou détectent des niveaux de pollution dangereux dans les villes. Les quantités massives de capteurs et d'actionneurs constituent des infrastructures de grande envergure qui s'étendent sur des terrains de stationnement entiers, des campus comprenant plusieurs bâtiments ou des champs agricoles. Le développement d'applications pour de telles infrastructures reste difficile, malgré des déploiement réussis dans un certain nombre de domaines. Une connaissance considérable des spécificités matériel / réseau de l'infrastructure de capteurs est requise de la part du développeur. Pour remédier à ce problème, des méthodologies et des outils de développement logiciel permettant de relever le niveau d'abstraction doivent être introduits pour que des développeurs non spécialisés puissent programmer les applications.

Cette thèse présente une méthodologie dirigée par la conception pour le développement d'applications orchestrant des quantités massives d'objets communicants. La méthodologie est basée sur un langage de conception dédié, nommé DiaSwarm qui fournit des constructions déclaratives de haut niveau permettant aux développeurs de traiter des masses d'objets en phase de conception, avant de programmer l'application. La programmation générative est utilisée pour produire des cadres de programmation spécifiques à la conception pour guider et soutenir le développement d'applications dans ce domaine. La méthodologie intègre le traitement parallèle de grandes quantités de données collectées à partir de masses de capteurs. Nous introduisons un langage de déclarations permettant de générer des cadres de programmation basés sur le modèle de programmation MapReduce. En outre, nous étudions comment la conception peut être utilisée pour rendre explicites les ressources requises par les applications ainsi que leur utilisation. Pour faire correspondre les exigences de l'application à une infrastructure de capteurs cible, nous considérons les déclarations de conception à différents stades du cycle de vie des applications.

Le passage à l'échelle de cette approche est évaluée dans une expérience qui montre comment les cadres de programmation générés s'appuyant sur le modèle de programmation MapReduce sont utilisés pour le traitement efficace de grands ensembles de données de relevés des capteurs. Nous examinons l'efficacité de l'approche proposée pour relever les principaux défis du génie logiciel dans ce domaine en mettant en œuvre des scénarios d'application qui nous sont fournis par des partenaires industriels. Nous avons sollicité des programmeurs professionnels pour évaluer l'utilisabilité de notre approche et présenter des données quantitatives et qualitatives de l'expérience.

Mots clefs : *développement logiciel, langages dédiés, conception, programmation générative, capteurs, actionneurs, orchestration*

Inria Bordeaux - Sud-Ouest — 200 Avenue de la Vieille Tour — 33405 Talence — France