N° D'ORDRE 2009-ISAL-0102 ANNÉE 2009

THÈSE

A DECLARATIVE APPROACH FOR PERVASIVE ENVIRONMENTS: MODEL AND IMPLEMENTATION

UNE APPROCHE DÉCLARATIVE POUR LES ENVIRONNEMENTS PERVASIFS: MODÈLE ET IMPLÉMENTATION

Présentée devant :
L'Institut National des Sciences Appliquées de Lyon
Pour obtenir :
Le grade de docteur
Spécialité :
Informatique
Formation doctorale :
Informatique
École doctorale :
Informatique et Mathématiques
Par :
Yann GRIPAY

SOUTENUE PUBLIQUEMENT LE 10 DÉCEMBRE 2009 DEVANT LE JURY COMPOSÉ DE :

Véronique BENZAKEN, Professeur des Universités, Université de Paris XI ..... Examinatrice
Ahmed K. ELMAGARMID, Professeur, Université de Purdue ..... Rapporteur
Stéphane GRUMBACH, Directeur de Rech., INRIA / Chinese Academy of Sciences Rapporteur
Frédérique LAFOREST, Maître de Conférences HDR, INSA de Lyon .... Co-directrice de thèse
Ioana MANOLESCU, Chargée de Recherche HDR, INRIA Saclay ..... Rapporteur
David MENGA, Ingénieur Recherche, EDF R&D ..... Invité
Jean-Marc PETIT, Professeur des Universités, INSA de Lyon ..... Co-directeur de thèse

À Miao et Zoé

Remerciements

Je remercie activement tous ceux qui m'ont accompagné pendant le déroulement de cette thèse, en particulier les générations passées, présentes et futures de doctorants et doctorantes du LIRIS qui ont contribué à la vie palpitante du laboratoire. Je remercie chaleureusement mes encadrants, Frédérique Laforest et Jean-Marc Petit, pour leur soutien et leurs conseils précieux qui ont permis à ces travaux de devenir ce qu'ils sont actuellement. Je n'oublie pas non plus l'ensemble des membres du LIRIS et du Département Informatique de l'INSA de Lyon qui ont constitué cet environnement accueillant et riche en expériences dans lequel cette thèse s'est déroulée.

Enfin, je remercie le lecteur de ce document pour son intérêt et lui souhaite une lecture agréable de ces quelques 186 pages.

Abstract

Computing environments evolve toward what is called pervasive systems: they tend to be more and more heterogeneous, decentralized and autonomous. On the one hand, personal computers and other handheld devices are largely widespread and take a large part of information systems. On the other hand, available data sources and functionalities may be distributed over large areas through networks that range from a world-wide network like the Internet to local peer-to-peer connections like for sensors. They are dynamic and heterogeneous: distributed databases with frequent updates, data streams from logical or physical sensors, and services providing data from sensors or storage units, transforming data or commanding actuators.

Pervasive environments pose new challenges in order to exploit their full potential, in particular through the management of complex interactions between distributed resources. Their heterogeneous data sources and functionalities are not homogeneously manageable in today's systems. This is a big issue when building pervasive applications. Imperative programming languages (e.g., C++, Java), classical query languages for databases (e.g., SQL), and network protocols (e.g., JMX, UPnP) must be combined in *ad hoc* developments, which is neither convenient nor suitable as a long-term solution.

Declarative approaches offer the advantage of providing a logical view on resources that abstracts physical access issues and enables optimization techniques. SQL queries over relational databases are a typical and well-known illustration of those approaches. Therefore, querying data sources and functionalities in a declarative way is recognized as a major issue in pervasive environments in order to simplify the development of applications.

Currently, extensions of DBMSs (DataBase Management Systems) provide a homogeneous view and query facilities for both relational data and data streams (e.g., DSMSs (Data Stream Management Systems), CEP (Complex Event Processing), ESP (Event Stream Processing)). Services are a common way to represent distributed functionalities in a computing environment, but are not yet fully integrated with DBMSs. Despite a lot of propositions, a clear understanding of the interplays between relational data, data streams and services is still lacking and is the major bottleneck toward the declarative definition of pervasive applications, instead of the current *ad hoc* development of such applications.

In this thesis, we propose a framework that defines a data-centric view of pervasive environments: the standard notion of database is extended to come up with a broader notion, defined as relational pervasive environment, integrating both conventional and non-conventional data sources, namely data, streams and services. It enables the development of applications for pervasive environments using declarative service-oriented

continuous queries combining those data sources.

Within this framework, we propose a data model for pervasive environments, namely the SoCQ data model (standing for Service-oriented Continuous Query), that takes into account their heterogeneity, dynamicity and distribution. We define the *structure* of our data model with the notion of eXtended Dynamic Relation (XD-Relation) representing data sources. We also define an algebraic *language* for our data model with the Service-enabled algebra (Serena algebra), from which a SQL-like language (the Serena SQL) has been devised. This language enables the expression of declarative queries over pervasive environments.

In order to implement this framework, we have designed a Pervasive Environment Management System (PEMS) that supports our data model. A PEMS is a service-enabled dynamic data management system that seamlessly handles network issues like service discovery and remote interactions. It supports the execution of service-oriented one-shot and continuous queries that application developers can easily devise to build pervasive applications. A prototype of PEMS has been implemented, on which experimentations have been conducted.

Résumé

Les environnements informatiques évoluent vers ce qu'on appelle des systèmes pervasifs : ils ont tendance à être de plus en plus hétérogènes, décentralisés et autonomes. D'une part, les ordinateurs personnels et autres terminaux mobiles sont largement répandus et occupent une grande place dans les systèmes d'information. D'autre part, les sources de données et fonctionnalités disponibles peuvent être réparties sur de larges espaces grâce à des réseaux allant du réseau mondial Internet jusqu'aux réseaux locaux pair-à-pair pour les capteurs. Elles sont de plus dynamiques et hétérogènes : bases de données avec des mises à jour fréquentes, flux de données provenant de capteurs logiques ou physiques, et services fournissant des données stockées ou provenant de capteurs, transformant des données ou commandant des actionneurs.

Les environnements pervasifs posent de nouveaux défis pour exploiter leur plein potentiel, en particulier la gestion d'interactions complexes entre ressources réparties. Il est cependant difficile de gérer ces sources de données et fonctionnalités hétérogènes avec les systèmes actuels, ce qui constitue un frein pour le développement d'applications pervasives. Il est ainsi nécessaire de combiner au sein de développements *ad hoc* des langages de programmation impératifs (C++, Java…), des langages de requêtes classiques pour les bases de données (SQL…) et des protocoles réseau (JMX, UPnP…). Ce n'est cependant une solution ni pratique ni adéquate sur le long terme.

Les approches déclaratives offrent l'avantage de fournir une vue logique des ressources qui abstrait les problématiques d'accès physique et permet la mise en œuvre de techniques d'optimisation. Les requêtes SQL sur les bases de données relationnelles en sont une illustration typique et bien connue. C'est pourquoi la définition déclarative de requêtes sur des sources de données et des fonctionnalités est reconnue comme un défi majeur dans le but de simplifier le développement d'applications pervasives.

Actuellement, les extensions des SGBDs (Système de Gestion de Bases de Données) permettent d'avoir une vue homogène et d'effectuer des requêtes sur des bases de données et des flux de données (notamment les SGFDs, Système de Gestion de Flux de Données). La notion de service est un moyen courant de représenter les fonctionnalités réparties d'un système informatique, mais n'est pas encore pleinement intégrée au sein des SGBDs. Malgré de nombreuses propositions, une compréhension claire des interactions entre données, flux de données et services manque toujours, ce qui constitue un frein majeur pour la définition déclarative des applications pervasives, en lieu et place des actuels développements *ad hoc*.

Dans cette thèse, nous proposons un framework définissant une vue orientée données des environnements pervasifs : la notion classique de base de données est étendue pour construire une notion plus large, l'environnement pervasif relationnel, qui intègre

les sources de données à la fois conventionnelles et non-conventionnelles, à savoir données, flux de données et services. Cette notion permet le développement d'applications pervasives de manière déclarative en utilisant des requêtes continues orientées service qui combinent ces sources de données.

Dans ce framework, nous proposons un modèle de données pour les environnement pervasifs, appelé SoCQ (pour Service-oriented Continuous Query), qui prend en compte leur hétérogénéité, dynamicité et répartition. Nous définissons la *structure* de notre modèle de données avec la notion de relation dynamique étendue (eXtended Dynamic Relation, ou XD-Relation) représentant les sources de données. Nous définissons également un *langage* algébrique pour notre modèle de données avec l'algèbre Serena (Service-enabled algebra), à partir de laquelle un langage de type SQL a été défini. Ce langage permet d'exprimer de manière déclarative des requêtes sur les environnements pervasifs.

Afin d'implémenter ce framework, nous avons conçu une architecture de système de gestion d'environnements pervasifs (Pervasive Environment Management System, ou PEMS) qui prend en charge notre modèle de données. Un PEMS est un système de gestion dynamique de données et de services qui gère de manière transparente les problématiques liées au réseau telles que la découverte de services et les interactions à distance. Il supporte l'exécution de requêtes ponctuelles et continues orientées service que les développeurs d'applications peuvent aisément concevoir pour développer des applications pervasives. Un prototype de PEMS a été implémenté, avec lequel des expérimentations ont été réalisées.

Contents

Remerciements i
Abstract iii
Résumé v
Contents vii
List of Tables ix
List of Figures x
List of Definitions xi
List of Examples xii

1 Introduction **1**
1.1 Context 2
1.2 Contributions 6
1.3 Document organization 8

2 Modeling of Pervasive Environments **11**
2.1 Preliminaries 12
2.2 Modeling of distributed functionalities 16
2.3 The relational pervasive environment 24
2.4 The Serena DDL 33
2.5 Summary 37

3 Querying over a Pervasive Environment **39**
3.1 Serena one-shot query algebra 42
3.2 Serena continuous query algebra 51
3.3 Query equivalence 63
3.4 Query optimization 70
3.5 The Serena SQL 75
3.6 Summary 79

4 Toward a Pervasive Environment Management System **81**
4.1 Architecture of a PEMS 82
4.2 Implementation of the SoCQ PEMS 84
4.3 Experimentation 95
4.4 Summary 106

5 Related Work **109**
5.1 Pervasive environments 110
5.2 Related database research 113
5.3 Enabling technologies 120

6 Conclusion **123**
6.1 Summary of contributions 124
6.2 Discussion and perspectives 126
6.3 Final words 129

A Résumé long en français **131**

B Poster de démonstration **159**

Bibliography **163**

List of Tables

1.1 Overview of Notations for the SoCQ Data Model 10

2.1 Example of prototypes and services 19
2.2 Description of X-Relations from the Relational Pervasive Environment for the temperature surveillance scenario 26
2.3 Description of XD-Relations from the Relational Pervasive Environment for the temperature surveillance scenario 27
2.4 Overview of Notations for the Structure of the SoCQ Data Model 34
2.5 Serena DDL syntax for XD-Relations 35
2.6 Example of XD-Relations defined using the Serena DDL 36

3.1 Definition of the Projection, Selection and Renaming operators over X-Relations 44
3.2 Definition of Natural Join operator over X-Relations 45
3.3 Definition of realization operators over X-Relations 46
3.4 Definition of the Service Discovery operator 48
3.5 Examples of one-shot queries expressed in the Serena algebra 50
3.6 Overview of Notations for X-Relations and One-shot Queries 51
3.7 Definition of the selection operator over a finite XD-Relation 53
3.8 Summary of the extension of one-shot query operators over X-Relations to continuous query operators over finite XD-Relations, with s, r, r_1 and r_2 being finite XD-Relations 54
3.9 Definition of the service discovery continuous query operator 55
3.10 Definition of Invocation Binding operator over a finite XD-Relation . . . 57
3.11 Definition of Subscription Binding operator over a finite XD-Relation . . 58
3.12 Definition of Window and Streaming operators over XD-Relations 59
3.13 Examples of continuous queries over XD-Relations expressed in the Serena algebra 62
3.14 Overview of Notations for XD-Relations and Continuous Queries 63
3.15 One-shot queries expressed in the Serena algebra 65
3.16 Rewriting rules with assignment and invocation operators 67
3.17 Serena SQL syntax for continuous and one-shot queries 77
3.18 Generic example of the Serena SQL syntax 78
3.19 Serena SQL syntax for service discovery queries 78

3.20 Example of Serena SQL service discovery query 78

4.1 XD-Relations with Service Discovery Queries for the Temperature Surveillance Scenario 99
4.2 Administration XD-Relations for the Temperature Surveillance Scenario 100
4.3 Main Query for the Temperature Surveillance Scenario 101
4.4 XD-Relation with Service Discovery Query for the RSS Feeds Scenario . 103
4.5 Simple Queries for the RSS Feeds Scenario 104
4.6 XD-Relations and Main Serena Query for the RSS Feeds Scenario 105

List of Figures

4.1 Overview of the PEMS Architecture 83
4.2 Execution of the OSGi Framework Instance 86
4.3 The PEMS Simple GUI (Table Panel and Query Panel) 93
4.4 PEMS GUI developed as an Eclipse RCP application 94
4.5 Interface of the Web Application for the “RSS Feeds” scenario 94
4.6 Virtual Temperature Sensors (roof, corridor, office) 96
4.7 Physical Temperature Sensors (iButton, iButton base, SunSpot) 97
4.8 Illustration of the Execution of the Temperature Surveillance Scenario . . 101

List of Definitions

2.1 Service Availability Predicate 20
2.2 Service Property Function 21
2.3 Prototype Data Function 22
2.4 Extended Relation Schema 27
2.5 Extended Relation 28
2.6 Projection of a Tuple 28
2.7 Extended Dynamic Relation Schema 29
2.8 Extended Dynamic Relation 30
2.9 Relational Pervasive Environment Schema 33
2.10 Relational Pervasive Environment 33
3.1 One-shot Query 49
3.2 Continuous Query 60
3.3 Action Set 64
3.4 Query Equivalence 66

List of Examples

2.1 Infinite Dynamic Relation 15
2.2 Finite Dynamic Relation 15
2.3 Distributed Functionalities as Prototypes and Services 18
2.4 Prototypes and Services 19
2.5 Service Availability 20
2.6 Service Property Function 21
2.7 Invocation and Subscription Prototypes 22
2.8 X-Relations 25
2.9 Relational Pervasive Environment 25
2.10 Extended Relation 29
2.11 Infinite Extended Dynamic Relation 30
2.12 Finite Extended Dynamic Relation 31
2.13 Serena DDL 33
3.1 Serena one-shot service discovery operator 47
3.2 Serena one-shot query 49
3.3 Continuous Query 60
3.4 Action Set 65
3.5 Query Equivalence 66
3.6 Tuple Statistics and Query Cost 72
3.7 Query Optimization 74

1 Introduction

Chapter Outline

1.1 Context **2**
1.1.1 Pervasive environments 2
1.1.2 Requirements of pervasive applications 3
1.2 Contributions **6**
1.2.1 Data model 6
1.2.2 Implementation 8
1.3 Document organization **8**

> "Most important, ubiquitous computers will help overcome the problem of information overload. There is more information available at our fingertips during a walk in the woods than in any computer system, yet people find a walk among trees relaxing and computers frustrating. Machines that fit the human environment instead of forcing humans to enter theirs will make using a computer as refreshing as taking a walk in the woods."
> *Mark Weiser, "The Computer for the 21st Century" [Wei91]*

1.1 Context

Computing environments evolve towards what is called pervasive systems: they tend to be more and more heterogeneous, decentralized and autonomous. On the one hand, personal computers and other handheld devices are largely widespread and take a large part of information systems. On the other hand, available data sources and functionalities may be distributed over large areas through networks that range from a world-wide network like the Internet to local peer-to-peer connections like for sensors. They are dynamic and heterogeneous: distributed databases with frequent updates, data streams from logical or physical sensors, and services providing data from sensors or storage units, transforming data or commanding actuators.

Pervasive environments pose new challenges in order to exploit their full potential, in particular through the management of complex interactions between distributed resources. Their heterogeneous data sources and functionalities are not homogeneously manageable in today's systems. This is a big issue when building pervasive applications. Imperative programming languages (e.g., C++, Java), classical query languages for databases (e.g., SQL), and network protocols (e.g., JMX, UPnP) must be combined in *ad hoc* developments, which is neither convenient nor suitable as a long-term solution.

Declarative approaches offer the advantage of providing a logical view on resources that abstracts physical access issues and enables optimization techniques. SQL queries over relational databases are a typical and well-known illustration of those approaches. Therefore, querying data sources and functionalities in a declarative way is recognized as a major issue in pervasive environments in order to simplify the development of applications.

1.1.1 Pervasive environments

The idea of ubiquitous computing, or pervasive computing, was initiated by Mark Weiser in his famous article "The Computer for the 21st Century" [Wei91] in 1991. His vision of computers fully integrated in the human environment and gracefully providing information and services to users is still an open issue in computer science and computer engineering.

Pervasive computing results from the evolution of the computing paradigm from centralized mainframe computers with "dummy" terminals at an organizational level to more decentralized networks of personal computers at a user level, and toward the multiplication of "smart" small-scale appliances, e.g., hand-held devices like smart phones or PDAs, or embedded devices integrated in the surrounding environment, like autonomous sensors and actuators. In so-called pervasive information systems [KG07a], those smart objects can benefit from wireless and wired networks to remotely access powerful computing and large distributed databases, and to remotely interact with other smart objects, thus creating what could be called the "Internet of Things" [Uni05].

This integration of "computerized artifacts" blurs the distinction between computers and other electronic devices [KG07b], leading to new application models. Service Oriented Architecture (SOA) principles [Erl05], initially dedicated to the design of enterprise information systems, are now derived into Service Oriented Device Architecture (SODA) principles [dDCK+06] that enable a wide integration of functionalities. From a user point of view, applications can be mobile, localized and personalized: new interaction possibilities can make applications go "off the desktop", i.e. applications can run in the background, using the user environment itself as an ubiquitous interface. From a system point of view, sensors and actuators can be distributed in the environment and autonomously gather data and execute actions with no or few human interactions.

1.1.2 Requirements of pervasive applications

Developing applications in such complex computing environments leverages the common issues of distributed applications. It also poses new challenges to handle the dynamicity of such environments. We detail the requirements of pervasive application development from two points of view that we aim at reconciling: middleware-oriented requirements and dataspace-oriented requirements.

1.1.2.1 Middleware-oriented requirements

As presented in [BC07], issues of pervasive environments lead to the need for middlewares that simplify the development of pervasive applications, i.e. (distributed) software layers that manage common issues and offer high-level functionalities in order to help to build and execute such applications. Middlewares for pervasive applications should offer a unified representation of the pervasive environment and an easy access to distributed resources. We summarize those requirements as follows:

1. *Abstraction of devices.* Devices may range from isolated sensors to mainframe computers, including smart phones, PDAs, desktop computers, and may be embedded in the environment, mobile, handheld, or stationary. An abstraction of their

functionalities is required to get a unified representation of the environment resources. Devices can be viewed as distributed entities providing some of the following functionalities:

- *sensor*: reporting one or more environment parameters or events;
- *actuator*: modifying the environment through its actions;
- *computation*: computing some information given some input data;
- *storage*: storing data and answering to queries about it.

Those entities may be reactive, i.e. only reacting to external request; proactive, i.e. initiating interactions with other entities according to a specific behavior, e.g., upon environment changes; or even autonomous, i.e. deciding themselves of their own behavior.

2. *Loosely coupled communications.* Entities from a pervasive environment need to communicate in a distributed dynamic environment setting. Loosely coupled communications are required through an abstraction of the communication layer: interoperable data format, lookup and discovery mechanisms, asynchronous communications, *etc.*

3. *Context management.* The general notion of context can be defined as "any information that can be used to characterize the situation of entities (i.e., whether a person, place, or object) that are considered relevant to the interaction between a user and an application" [DAS01]. Context needs to be captured through acquisition, interpretation and aggregation. Context management is required in order to enable context-aware behaviors within pervasive applications.

4. *Application development support.* Functionalities provided by distributed entities from a pervasive environment may appear or disappear dynamically. In order to make the development of applications easier, applications should be defined using abstract functionalities and dynamically linked to actual implementations at runtime, depending on the available resources. Furthermore, the specification of interactions between dynamically discovered entities should remain simple when building complex applications .

A common trend in middlewares is to represent entities providing functionalities using the notion of service. As devices may be sensors or actuators [ECPS02], services may represent some interactions with the physical environment, like taking a photo from a camera or displaying a picture on a screen. These interactions bridge the gap between the computing environment and the physical environment, that can be both managed by pervasive applications.

In summary, the set of devices in pervasive environments can be abstracted as an environment of distributed services providing sensor, actuator, computation and storage functionalities, where some services may be autonomous. Pervasive applications

can rely on the middleware to easily interact with those services in a loosely coupled way, using context information to adapt their behavior.

However, current middlewares still rely on imperative programming languages for application development, in particular when combining services with databases and data streams to build complex application behaviors.

1.1.2.2 Dataspace-oriented requirements

Data integration has been a long standing theme of research over the past 30 years. Now, the broader notion of dataspace [FHM05] has appeared to provide base functionality over all data sources and applications, regardless of how integrated they are and without having a full control over the underlying data. Dataspaces can be viewed as data-oriented distributed environment containing several autonomous data sources: relational databases, XML databases, data stream providers, publish/subscribe systems, *etc.*

In this setting, a major requirement is to provide a homogeneous view and query facilities over heterogeneous distributed databases. In order to avoid complexity, the definition of queries should remain declarative, e.g., using SQL-like or XQuery-like query languages.

Another major requirement is the integration of dynamic data sources, namely data streams or event streams. This integration has led to the development of the notion of continuous query, i.e. queries that last in time and continuously update their results according to the evolution of their input data sources, as opposed to "one-shot" or "one-time" queries, that are executed only once to produce a static result, like traditional SQL or XQuery queries. Currently, extensions of DBMSs (DataBase Management Systems) provide a homogeneous view and query facilities for both relational data and data streams: DSMSs (Data Stream Management Systems), CEP (Complex Event Processing), ESP (Event Stream Processing). Continuous queries should also remain declarative, although their definition is more complex than the definition of one-shot queries in order to integrate temporal aspects.

Continuous queries can be viewed as simple applications that involve several data sources from a distributed environment and produce data. They can be used to define some parts of more complex pervasive applications in a declarative way, that can then be optimized. They are however not designed to define interactions with devices from the pervasive environment that are more than only data providers.

1.1.2.3 Addressing all requirements

In the one hand, distributed services are a common way to represent devices that provide data sources and functionalities in a pervasive environment, and middlewares can

simplify the development of pervasive applications using those services. On the other hand, data management systems enable to define interactions between distributed data sources through declarative queries, enabling optimization techniques.

We aim at addressing requirements from both points of views, i.e. middleware-oriented and dataspace-oriented requirements: easy management of distributed services (providing data sources and functionalities) and declarative definition of (continuous) interactions. The definition of continuous queries that can combine distributed data sources and distributed services could lead to this goal. However, services are not yet fully integrated with data management systems, although specific external functions can be developed to emulate this integration in current DBMSs.

Despite a lot of propositions in the literature, a clear understanding of the interplays between databases, data streams and services is still lacking. We claim that it is the major bottleneck toward the declarative definition of pervasive applications, instead of the current *ad hoc* development of such applications.

1.2 Contributions

In this thesis, we tackle the requirements of pervasive application development through the leveraging of database principles. We address middleware-oriented requirements and dataspace-oriented requirements by integrating the notion of service within the management of dynamic distributed data sources.

We propose a framework that defines a data-centric view of pervasive environments: the standard notion of database is extended to come up with a broader notion, defined as relational pervasive environment, integrating both conventional and non-conventional data sources, namely data, streams and services. It allows the development of applications for pervasive environments using declarative service-oriented continuous queries combining those data sources.

1.2.1 Data model

Within this framework, we propose a data model for pervasive environments, namely the SoCQ data model (standing for Service-oriented Continuous Query) [GLP07, GLP08, Gri08, GLP09a], that takes into account their heterogeneity, dynamicity and distribution. We define the *structure* of our data model with the notion of eXtended Dynamic Relation (XD-Relation) representing data sources. We also define an algebraic *language* for our data model with the Service-enabled algebra (Serena algebra), from which a SQL-like language (the Serena SQL) has been devised. This language allows to express declarative queries over pervasive environments.

The SoCQ data model is based on the relational data model. Other data models could have been chosen, in particular XML-based data models, e.g., for XML databases,

in the ActiveXML project [Act, AMT06]. XML technologies are also widely used for service-oriented computing, e.g., XML Web Services, the UPnP technology [UPn]. The relational data model has however been chosen for the following reasons:

- it is a simple, robust and powerful model for data management that is well-known in the database community for years;
- most projects of integration of data streams within a data management system are based on it;
- it allows the SoCQ data model to remain relatively simple for the definitions and notations;
- and above all, it allows to keep the "gold standard" SQL language, ensuring a smooth transition from traditional applications to pervasive applications for developers.

The structure of the SoCQ data model is built on a data-oriented representation of distributed functionalities as services and defines the notion of XD-Relation schema that can integrate service descriptions with data schema. XD-Relation schemas are either finite or infinite, have real attributes and virtual attributes, and are associated with binding patterns that link attributes to inputs/outputs of service functionalities. Data, streams and services are homogeneously represented as "extended" data sources by XD-Relations.

The language of the SoCQ data model is the Serena algebra that allows to express queries over a relational pervasive environments, i.e. over a set of XD-Relations. Operators of the relational algebra are redefined and new operators dedicated to virtual attributes and binding patterns are defined, as well as operators handling infinite XD-Relations, i.e. data streams. An additional operator representing service discovery is also defined. Using those operators, we build the notion of query over a relational pervasive environment. We also define query equivalence in this setting, and propose a cost model enabling query optimization techniques.

In order to manage dynamic data sources, the notion of time needs to be integrated into the data model. It however adds complexity. In the SoCQ data model, we consider a discrete time representation as an infinite ordered set of time instants. For the definition of the data model structure, as well as for its language, we first consider the simpler one-time case, with non-dynamic eXtended relations (X-Relations) and one-time queries; and we then consider the more complex continuous case, with XD-Relations and continuous queries.

Consequently, data description languages have been defined for the management of XD-Relations: Serena DDL for creation, Serena DML for data manipulation. A SQL-like syntax (Serena SQL) has been proposed for the expression of queries, which is a direct representation of algebra operators.

1.2.2 Implementation

In order to implement this framework, we have designed a Pervasive Environment Management System (PEMS) [GLP09b, GLLP09] that supports our data model. The general architecture is composed of several modules, with one module that is distributed among all devices of the pervasive environment in order to manage remote interactions. The role and requirements of each module are detailed.

We have developed a prototype of a PEMS, referred to as the SoCQ PEMS prototype, that implements all the required modules. The prototype is based on the OSGi framework and uses the UPnP technology for service discovery and remote interactions. It also implements parsers, using the JavaCC technology, that interprets the different declarative languages used by the PEMS: the Serena DDL, DML, SQL and algebra. It allows to handle distributed services, to manage XD-Relations and to launch one-time and continuous queries.

We have conducted experimentation in order to assess the validity and expressiveness of our data model. Two scenarios have been implemented. The "Temperature Surveillance" scenario involves distributed temperature sensors and sets up a notification system through declarative service-oriented continuous queries. The "RSS Feeds" scenario involves a service that handles RSS feeds from newspaper web sites, and sets up a subscription system that allows users to receive news of interest when they are published.

1.3 Document organization

Following the current Introduction Chapter, the SoCQ data model is thoroughly described and formally defined in Chapter 2 and Chapter 3. In order to help the reader, a summary of the different notations is provided at pertinent places throughout those two chapters. The whole overview is presented at the end of this Chapter in Table 1.1 for future reference.

In Chapter 2, the structure of the data model is presented: some preliminary notions and notations are first defined, in particular the representation of time; the modeling of distributed functionalities as services is then tackled; the fundamental notions of X-Relation and XD-Relation are finally defined, along with the definition of the Serena DDL language.

In Chapter 3, the language of the data model is presented: the Serena one-shot query algebra and the Serena continuous query algebra are formally defined; query equivalence is also defined for the Serena algebra; logical query optimization is tackled, in particular with the description of a dedicated cost model; finally, the Serena SQL is defined.

In Chapter 4, the implementation of the data model is presented: a general architecture for a PEMS is detailed; the implementation of the SoCQ PEMS is then described, with a presentation of the used technologies; finally, the experimentation with the two scenarios are detailed.

Related works are discussed in Chapter 5. In Chapter 6, a summary of the thesis contributions is presented, followed by a final discussion about those contributions and some identified perspectives.

Table 1.1: Overview of Notations for the SoCQ Data Model

	STRUCTURE	LANGUAGE
DATA	• Boolean Domain $\mathcal{B}$ • Constants $\mathcal{D}$ • Attributes $\mathcal{A}$ • Relation Schema R – $schema(R) \subset \mathcal{A}$ • Relation r over R – $r \subset \mathcal{D}^{\lvert schema(R)\rvert}$	• Operators over Relation(s) – Selection $\sigma_F(r)$ – Projection $\pi_Y(r)$ – Renaming $\rho_{A\to B}(r)$ – Natural Join $r_1 \bowtie r_2$
DATA + SERVICES	• Prototypes $\psi \in \Psi$ – Relation Schema $Input_\psi$ – Relation Schema $Output_\psi$ – $active(\psi) \in \mathcal{B}$ • Services Interfaces $\phi \in \Phi$ – $prototypes_\phi \subset \Psi$ – $attributes_\phi \subset \mathcal{A}$ • Services $\omega \in \Omega$ – $id(\omega) \in \mathcal{D}$ – $prototypes(\omega) \subset \Psi$ – $attributes(\omega) \subset \mathcal{A}$ • Interaction with Services – $available(\omega) \in \mathcal{B}$ – $property_\omega(A) \in \mathcal{D}$ – $data_\psi(id(\omega), input) \in \mathcal{D}^{\lvert schema(Output_\psi)\rvert}$ • X-Relation Schema R – Relation Schema R – $realSchema(R)$ – $virtualSchema(R)$ – $BP(R) \subset (\Psi \times \mathcal{A})$ • X-Relation r over R – $r \subset \mathcal{D}^{\lvert realSchema(R)\rvert}$	• Operators over X-Relation(s) – Selection $\sigma_F(r)$ – Projection $\pi_Y(r)$ – Renaming $\rho_{A\to B}(r)$ – Natural Join $r_1 \bowtie r_2$ – Assignment $\alpha_{A\equiv B}(r)$ / $\alpha_{A\equiv c}(r)$ – Binding $\beta_{\langle\psi,S\rangle}(r)$ • Operator for Service Discovery – Service Discovery $\xi_{S,\phi,\phi'}()$
DATA + SERVICES + TIME + STREAMS	• Discrete Time Domain $\tau_i \in \mathcal{T}$ • Interaction with Services – $available(\omega, \tau_i) \in \mathcal{B}$ – $data_\psi(id(\omega), input, \tau_i) \in \mathcal{D}^{\lvert schema(Output_\psi)\rvert}$ • XD-Relation Schema R – X-Relation Schema R – $infinite(R) \in \mathcal{B}$ • XD-Relation r over R – $\tau_{start_r} \in \mathcal{T}$ – $r^*(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R)\rvert}$ – $r^+(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R)\rvert}$ – $r^-(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R)\rvert}$	• Operators over finite XD-Relation(s) – Selection $\sigma_F(r)$ – Projection $\pi_Y(r)$ – Renaming $\rho_{A\to B}(r)$ – Natural Join $r_1 \bowtie r_2$ – Assignment $\alpha_{A\equiv B}(r)$ / $\alpha_{A\equiv c}(r)$ – Binding $\beta_{\langle\psi,S\rangle}(r)$ – Streaming $\mathcal{S}_{[event]}(r)$ • Operator over infinite XD-Relation(s) – Window $\mathcal{W}_{[size]}(r)$ • Operator for Service Discovery – Service Discovery $\xi_{S,\phi,\phi'}()$

2

Modeling of Pervasive Environments

Chapter Outline

2.1 Preliminaries **12**
2.1.1 Time representation 13
2.1.2 Basic notations 13
2.1.3 Dynamic relations 14
2.1.4 Summary of preliminaries 16
2.2 Modeling of distributed functionalities **16**
2.2.1 Modeling distributed functionalities as services 16
2.2.2 Definitions and notations 18
2.3 The relational pervasive environment **24**
2.3.1 Integrating data and services 24
2.3.2 Definitions and notations 26
2.4 The Serena DDL **33**
2.5 Summary **37**

In order to simplify the development of pervasive applications, a first step toward our goal is the modeling of such environments. We choose to build a data-centric view of pervasive environments using database principles, leading to the definition of the structure of a data model. Our data model is based on the key idea of describing functionalities as (parameterized) distributed data sources, in order to seamlessly integrate them with more conventional data sources, namely data relations and data streams. Furthermore, our next step is be to enable the declarative definitions of interactions between data and functionalities, so we will be able to reuse some existing declarative languages from the relational model.

In this chapter, we define the structure of the SoCQ data model: the standard notion of database is extended to come up with the broader notion of *relational pervasive environment*. A relational pervasive environment enables an heterogeneous representation of dynamic data sources and distributed functionalities homogeneously as extended data sources.

In Section 2.1, we first set down our representation of time and the notations for standard relations, leading to the definition of finite and infinite dynamic relations representing respectively data relations and data streams. Those representations and notations are the base for our model that is described in the following sections.

In Section 2.2, we define our model of distributed functionalities provided by heterogeneous devices of a pervasive environment: in order to homogeneously integrate distributed functionalities with data sources, we decouple their declaration as prototypes of methods and their implementation as services.

In Section 2.3, we define our model integrating distributed functionalities within dynamic data sources. At the metadata level, we integrate the declarative part into data schemas with the help of two notions: virtual attributes and binding patterns. The implementation part is integrated at the tuple level with data representing references to services. With this integration, we formally define a relational pervasive environment as a set of extended dynamic relations, or XD-Relations, homogeneously representing relations and data streams extended with services.

In Section 2.4, we define a SQL-like data description language for XD-Relations, namely the Serena DDL, along with a data manipulation language, the Serena DML.

2.1 Preliminaries

We use the standard relational model as a base to express our new data model. We mainly follow the notations given in [LL99]. We first set down basic notations for standard relations. We then detail our representation of time, using a discrete time model similar to [ABB+03]. Combining standard relations with our time representation, we

define some notations for dynamic relations that are used in the next sections to define the data model. Dynamic relations can represent standard relations and data streams.

2.1.1 Time representation

In order to handle continuous aspects of pervasive environments in a formal way, we need to explicitly set up our representation of time. We choose to use a discrete time model: it enables to define in an obvious way the notion of simultaneity of two events. For example, we can consider that a group of tuples is simultaneously inserted in a relation or simultaneously provided by a data source: we do not take into account the insertion/arriving order for a given instant. It enables us to keep a relational-like model that handles sets of tuples. It also enables to define sequences, with respect to time, of sets of simultaneous events at a sufficient granularity for the requirements of pervasive applications.

Like in CQL [ABB+03], we consider a discrete time domain of "time instants". However, we consider this time domain to be infinite: it does not have an earliest time instant as in [ABB+03].

More precisely, we consider a discrete, infinite countable and totally ordered time domain $\mathcal{T}$. We denote by $\tau_{i \in \mathbb{Z}} \in \mathcal{T}$ a *time instant*, or simply *instant*, and $\forall i, j \in \mathbb{Z}, i < j \Rightarrow \tau_i < \tau_j$. We denote by $\tau_{now} \in \mathcal{T}$ the present time instant. The pervasive environment is considered from the "past" instants until τ_{now} included, although it is implicit in the rest of our model.

In the following of our modeling, for the sake of simplicity, we consider the time domain to be the domain of relative integers:

$$\mathcal{T} = \mathbb{Z} = \{-\infty, ..., -5, -4, -3, ..., 15, 16, ..., +\infty\} ,$$

with $\tau_i = i$. For example, we could set the present time instant as $\tau_{now} = \tau_{15} = 15$.

2.1.2 Basic notations

Our data model for pervasive environments is based on five basic elements: time, constants, attributes, prototypes and services. Time enables the modeling of the dynamicity of the environment. Constants and attributes come from the relational model. Prototypes and services are related to our representation of distributed functionalities, and are described in Section 2.2.

Besides the time domain $\mathcal{T}$ previously defined, we define four countable infinite sets that are mutually disjoint:

- $\mathcal{D}$ for constants, i.e. data values;
- $\mathcal{A}$ for attribute names;

- Ψ for prototypes;
- Ω for services.

We also define $\mathcal{B}$ to be the Boolean domain, i.e. $\mathcal{B} = \{true, false\}$.

In this setting, we explicit some simple notations for (standard) *relation schemas* and *relations*. For a given relation schema R, we denote by $schema(R) \subset \mathcal{A}$ the set of attributes in R. A tuple over R is an element of $\mathcal{D}^{|schema(R)|}$, and a relation r over R is a finite set of tuples over R, i.e. $r \subset \mathcal{D}^{|schema(R)|}$.

2.1.3 Dynamic relations

In order to seamlessly handle data relations and data streams in our data model, we propose a common representation as a dynamic relation for both types of data sources, based on the time representation defined above. A dynamic relation is defined over a standard relation schema and represents a set of tuples for each time instant.

We consider that relations represent finite sets, whereas data streams represent infinite append-only sets where tuples are inserted but can not be later deleted. A dynamic relation r over a relation schema R is associated with a start instant $\tau_{start_r} \in \mathcal{T}$ that represents its creation time. Its content can be described by three time-dependent sets of tuples over R for $\tau_i \geq \tau_{start_r}$:

- the instantaneous relation $r^*(\tau_i)$, i.e. the content of r at τ_i;
- the inserted tuple set $r^+(\tau_i)$, i.e. the set of tuples inserted into r at τ_i;
- the deleted tuple set $r^-(\tau_i)$, i.e. the set of tuples deleted from r at τ_i.

The instantaneous relation r^* is a finite set of tuples for relations, and an infinite set of tuples for data streams. The deleted tuple set r^- is always empty for data streams, as we consider that data streams are append-only sets. Furthermore, we consider that all dynamic relations respect the following initial conditions:

- $r^+(\tau_{start_r}) = r^*(\tau_{start_r})$,
- $r^-(\tau_{start_r}) = \emptyset$;

the three sets are then related by the following relation:

$$\forall \tau_i > \tau_{start_r}, r^*(\tau_i) = (r^*(\tau_{i-1}) \cup r^+(\tau_i)) - r^-(\tau_i) \,.$$

Those notations are reused in particular for the formal definition of XD-Relations in Section 2.3. Through those notations, we aim at emphasizing that dynamic relations can represent both relations and data streams, in a similar way to the representation in [ABB+03]. The content of a relation r can be defined only by its instantaneous relation

r^* and the content of a data stream s by its inserted tuple set s^*, the rest being derived from them. It enables functionalities to be viewed as data sources, as it is described in the following sections. We illustrate those notations with two examples: one with a data stream, i.e. an infinite dynamic relation; and one with a relation, i.e. a finite dynamic relation.

Example 2.1 (Infinite Dynamic Relation) *A stream of temperature notifications (e.g., from temperature sensors) can be modelled by an infinite dynamic relation over a relation schema Temperatures with the following attributes:* $schema(Temperatures) = \{location, temperature\}$.

Let temperatures be an infinite dynamic relation over Temperatures with $\tau_{start_{temperatures}} = \tau_{i_0}$. *It starts at* τ_{i_0} *with an empty set of tuples. Two tuples are inserted at* τ_{i_0+1}, *and two others at* τ_{i_0+2}. *No tuple is inserted until* τ_{i_0+6}, *where one more tuple is inserted. The evolution of its content can be represented by its instantaneous relation* $temperatures^*(\tau_i)$ *at different instants, along with its set of inserted tuples* $temperatures^+(\tau_i)$. *Its set of deleted tuples* $temperatures^-(\tau_i)$ *is always empty as temperatures is an infinite dynamic relation.*

$$
\begin{array}{lcl}
temperatures^*(\tau_{i_0}) & = & \emptyset \\
temperatures^+(\tau_{i_0}) & = & temperatures^*(\tau_{i_0}) = \emptyset \\
temperatures^-(\tau_{i \geq i_0}) & = & \emptyset \\
 & & \\
temperatures^*(\tau_{i_0+1}) & = & \{\langle roof, 12.3\rangle, \langle corridor, 25.1\rangle\} \\
temperatures^+(\tau_{i_0+1}) & = & \{\langle roof, 12.3\rangle, \langle corridor, 25.1\rangle\} \\
 & & \\
temperatures^*(\tau_{i_0+2}) & = & \{\langle roof, 12.3\rangle, \langle corridor, 25.1\rangle, \langle roof, 13.1\rangle, \langle office, 23.4\rangle\} \\
temperatures^+(\tau_{i_0+2}) & = & \{\langle roof, 13.1\rangle, \langle office, 23.4\rangle\} \\
 & & \\
temperatures^*(\tau_{i_0+5}) & = & temperatures^*(\tau_{i_0+4}) = temperatures^*(\tau_{i_0+3}) = temperatures^*(\tau_{i_0+2}) \\
temperatures^+(\tau_{i_0+5}) & = & temperatures^+(\tau_{i_0+4}) = temperatures^+(\tau_{i_0+3}) = \emptyset \\
 & & \\
temperatures^*(\tau_{i_0+6}) & = & \{\langle roof, 12.3\rangle, \langle corridor, 25.1\rangle, \langle roof, 13.1\rangle, \langle office, 23.4\rangle, \langle office, 26.2\rangle\} \\
temperatures^+(\tau_{i_0+6}) & = & \{\langle office, 26.2\rangle\}
\end{array}
$$

For example, one tuple from the instantaneous relation at τ_{i_0+1}, *i.e.* $temperatures^*(\tau_{i_0+1})$, *is* $t = \langle roof, 12.3\rangle$.

Example 2.2 (Finite Dynamic Relation) *A list of available contacts (e.g., from an instant messaging software) can be modelled by a finite dynamic relation over a relation schema AvailableContactList with the following attributes:* $schema(AvailableContactList) = \{firstname, location\}$. *Let acl be a finite dynamic relation over AvailableContactList with* $\tau_{start_{acl}} = \tau_{j_0}$.

It starts at τ_{j_0} *with two tuples. One tuple is inserted at* τ_{j_0+2}, *and one tuple is deleted at* τ_{j_0+6}. *The evolution of its content can be represented by its instantaneous relation* $acl^*(\tau_i)$ *at different instants, along with its set of inserted tuples* $acl^+(\tau_i)$ *and its set of deleted tuples* $acl^-(\tau_i)$.

$$
\begin{array}{lcl}
acl^*(\tau_{j_0}) & = & \{\langle Nicolas, Elysee\rangle, \langle Carla, Elysee\rangle\} \\
acl^+(\tau_{j_0}) & = & acl^*(\tau_{j_0}) = \{\langle Nicolas, Elysee\rangle, \langle Carla, Elysee\rangle\} \\
acl^-(\tau_{j_0}) & = & \emptyset \\
\\
acl^*(\tau_{j_0+1}) & = & \{\langle Nicolas, Elysee\rangle, \langle Carla, Elysee\rangle\} \\
acl^+(\tau_{j_0+1}) & = & \emptyset \\
acl^-(\tau_{j_0+1}) & = & \emptyset \\
\\
acl^*(\tau_{j_0+2}) & = & \{\langle Nicolas, Elysee\rangle, \langle Carla, Elysee\rangle, \langle Francois, Matignon\rangle\} \\
acl^+(\tau_{j_0+2}) & = & \langle Francois, Matignon\rangle \\
acl^-(\tau_{j_0+2}) & = & \emptyset \\
\\
acl^*(\tau_{j_0+5}) & = & acl^*(\tau_{j_0+4}) = acl^*(\tau_{j_0+3}) = acl^*(\tau_{j_0+2}) \\
acl^+(\tau_{j_0+5}) & = & acl^+(\tau_{j_0+4}) = acl^+(\tau_{j_0+3}) = \emptyset \\
acl^-(\tau_{j_0+5}) & = & acl^-(\tau_{j_0+4}) = acl^-(\tau_{j_0+3}) = \emptyset \\
\\
acl^*(\tau_{j_0+6}) & = & \{\langle Nicolas, Elysee\rangle, \langle Francois, Matignon\rangle\} \\
acl^+(\tau_{j_0+6}) & = & \emptyset \\
acl^-(\tau_{j_0+6}) & = & \{\langle Carla, Elysee\rangle\} \\
\\
acl^*(\tau_{j_0+7}) & = & \{\langle Nicolas, Elysee\rangle, \langle Francois, Matignon\rangle\} \\
acl^+(\tau_{j_0+7}) & = & \emptyset \\
acl^-(\tau_{j_0+7}) & = & \emptyset
\end{array}
$$

2.1.4 Summary of preliminaries

We have proposed notations for dynamic relations (over a relation schema) enabling the representation data relations that evolve with time and data streams. It is worth noting that tuples are not "timestamped": although tuples are inserted at a given time instant, we do not consider that tuples themselves are linked with this logical instant.

This representation enables the use of set semantics for our data model, that do not take into account an insertion order for tuples inserted at the same discrete instant. Although multiset semantics [GdB94] could handle more precisely those data sources, in particular for data streams as it is proposed in [ABB+03], we use set semantics in order to keep the definition of our data model simple.

2.2 Modeling of distributed functionalities

2.2.1 Modeling distributed functionalities as services

As distributed functionalities from devices may appear or disappear dynamically in pervasive environments, applications should be defined using abstract functionalities: they are then dynamically linked to available implementations at runtime. This is a requirement for the development of pervasive applications. We consider two types

of functionalities: *methods* that can be invoked (e.g., to send an e-mail) and *streams* that can be subscribed to (e.g., the temperature stream from a sensor). The notion of stream enables to integrate devices that provide dynamic data sources into the data model [ECPS02].

We choose to abstract the distributed functionalities of a pervasive environment in a way that decouples the declaration of those functionalities (e.g., sending a message to someone) and their implementations (e.g., an e-mail sender, an instant message sender).

We represent the *declaration* of the *distributed functionalities* as **prototypes**. Input parameters and output parameters are defined by two relation schemas. Prototypes associated with a method are **invocation prototypes**, whereas those providing a stream are **subscription prototypes**, because "invocations" of such prototypes provide a stream of tuples as a result in a similar way to subscriptions to streams of events in publish/subscribe systems (e.g., [DGH+06]). When calling a prototype, input parameters take the form of a relation over the input schema (generally with only one tuple). For an invocation prototype, the call result is a relation over the output schema (0, 1 or several tuples). For a subscription prototype, the call result is an infinite dynamic relation over the output schema, i.e. a set of "inserted" tuples for every instant as long as the subscription is lasting.

A prototype input schema may be empty if the prototype has no input parameter. We however assume that the output schema of every prototype has at least one parameter so that call results provide data tuples that contain data. For example, an invocation prototype that sends a message can provide a single output parameter of type Boolean that indicates if the message has actually been sent. Although it is not a requirement for pervasive applications, we need this assumption to build a data-oriented view of functionalities.

As invocations/subscriptions can have an impact on the physical environment, e.g., invoking a prototype that sends a message, we need to consider two categories of prototypes: **active prototypes** and **passive prototypes**. Active prototypes are prototypes having a side effect on the physical environment that can not be neglected. On the opposite, the impact of passive prototypes is non-existent or can be neglected, like reading sensor data through a method or a stream. This is a strong requirement for the management of pervasive environments as pervasive applications involve interactions with human users or the physical environment that need to be taken into account.

We denote the *implementations* of those *distributed functionalities* by methods and streams provided by **services**. Services may represent heterogeneous physical or logical entities from the pervasive environment. As methods and streams correspond to some prototypes, we consider that services **implement** those prototypes. As only prototypes implemented by services are pertinent for our data model, methods and streams provided by services may remain implicit and can be safely hidden. Furthermore, services are identified by a globally unique identifier that we call **service reference**.

In order to provide tractable solutions even with a dynamic environment, the implementation of invocation prototypes are assumed to terminate (no infinite behavior) to avoid termination and recursion problems. This issue is tackled for example in [ABM04]. It does not however apply to subscription prototypes, as they are designed to provide continuous data.

From a system point of view, the notion of method and stream is handled by the "middleware part" of a Pervasive Environment Management System (PEMS). Through service discovery and remote invocation mechanisms [ZMN05], remote services are registered into the PEMS so that a prototype can be invoked on any registered service that implements this prototype by transparently calling the corresponding method/stream provided by this service. The globally unique identifier of services is also managed by those mechanisms. This abstraction of heterogeneous devices is a requirement for an implementation of a PEMS.

In addition to the prototypes they implement, services provide **properties** to describe themselves. Properties are couples of names and values that provide information about the service, e.g., the protocol of a messaging service, the location of a camera. In order to homogeneously integrate service properties into our data model, we assume that names of properties are attributes, and that, for a given service, the values of its properties never change.

Finally, as services may appear and disappear dynamically, we consider the **availability** of services over time. At a given instant, a service is either available or not. Once again from a system point of view, a service is available if it has been discovered by the PEMS. It is also handled by the "middleware part" of the PEMS.

Example 2.3 (Distributed Functionalities as Prototypes and Services) *From the temperature surveillance scenario, 4 prototypes and 5 services that implement them are presented in a pseudo-DDL format, in Table 2.1. Those prototypes correspond to some functionalities from the environment: sending a message to someone, receiving messages from a messaging account, checking if a photo of an area can be taken (indicating the expected delay and photo quality), and taking the photo. The prototype sendMessage is active, whereas the three others are passive. The prototype getMessages is the only subscription prototype (identified by the* STREAMING *keyword, as its "invocation" returns a stream) in this example. The 5 services are identified by their service references, i.e. their identifiers: email, jabber, camera01... Their properties, along with their types and values, are explicitly represented.*

2.2.2 Definitions and notations

Prototypes are defined by two relation schemas, one for the input and the other for the output. Those schemas are supposed to be disjoint and the output relation schema has to be non-empty. Let $\psi \in \Psi$ be a prototype, we denote by:

- $Input_\psi$ the input relation schema of ψ,

Table 2.1: Example of prototypes and services

```
PROTOTYPE sendMessage( address STRING, text STRING ) : (sent BOOLEAN) ACTIVE;

PROTOTYPE getMessages( username STRING, password STRING ) :
    ( address STRING, text STRING ) STREAMING;

PROTOTYPE checkPhoto( area STRING ) : ( quality INTEGER, delay REAL );

PROTOTYPE takePhoto( area STRING, quality INTEGER ) : ( photo BLOB );

SERVICE email    ( protocol STRING = "SMTP" )     IMPLEMENTS sendMessage;
SERVICE jabber   ( protocol STRING = "XMPP" )     IMPLEMENTS sendMessage, getMessages;
SERVICE camera01 ( location STRING = "office" )   IMPLEMENTS checkPhoto, takePhoto;
SERVICE camera02 ( location STRING = "corridor" ) IMPLEMENTS checkPhoto, takePhoto;
SERVICE webcam17 ( location STRING = "office", resolution STRING = "320x240" )
    IMPLEMENTS checkPhoto, takePhoto;
```

- $Output_\psi$ the output relation schema of ψ, with:
  - $schema(Output_\psi) \neq \emptyset$,
  - $schema(Input_\psi) \cap schema(Output_\psi) = \emptyset$;
- $active(\psi) \in \mathcal{B}$ a predicate returning true if ψ is active,
- $streaming(\psi) \in \mathcal{B}$ a predicate returning true if ψ is a subscription prototype (otherwise ψ is an invocation prototype).

Services are defined by the finite set of prototypes they implement and the finite set of attributes (or properties) they provide. A service is associated with a constant that is its service reference. Let $\omega \in \Omega$ be a service, we denote by:

- $prototypes(\omega) \subset \Psi$ the finite set of prototypes implemented by ω,
- $attributes(\omega) \subset \mathcal{A}$ the finite set of attributes provided by ω,
- $id(\omega) \in \mathcal{D}$ the service reference.

In order to simplify some notations, we define a service interface as a set of attributes and a set of prototypes. We denote an interface by $\phi = \langle attributes_\phi, prototypes_\phi \rangle \in \Phi$, with $\Phi = (\mathcal{P}(\mathcal{A}) \times \mathcal{P}(\Psi))$, $attributes_\phi \subset \mathcal{A}$ and $prototypes_\phi \subset \Psi$. It is mainly used for the definition of the service discovery operator in Chapter 3.

Example 2.4 (Prototypes and Services) *For the temperature surveillance scenario, we define four prototypes and five services. The four prototypes are sendMessage, getMessages, checkPhoto and takePhoto. For example, the input and output relation schemas associated with sendMessage are* $Input_{sendMessage}$ *and* $Output_{sendMessage}$*, with:*

- $schema(Input_{sendMessage}) = \{address, text\}$,

- $schema(Output_{sendMessage}) = \{sent\}$.

As sendMessage is an active invocation prototype and getMessages a passive subscription prototype, we have:

- $active(sendMessage) = true$,
- $streaming(sendMessage) = false$,
- $active(getMessages) = false$,
- $streaming(getMessages) = true$.

The five services from Ω are ω_1, ω_2, ω_3, ω_4 and ω_5, corresponding to the following service references: email, jabber, camera01, camera02 and webcam17 (cf. Example 2.3). Each service implements some prototypes and provides some attributes. For ω_1 and ω_5, we note:

- $id(\omega_1) = email$,
- $prototypes(\omega_1) = \{sendMessage\}$,
- $attributes(\omega_1) = \{protocol\}$,
- $id(\omega_5) = webcam17$,
- $prototypes(\omega_5) = \{checkPhoto, takePhoto\}$,
- $attributes(\omega_5) = \{location, resolution\}$.

The availability of services over time can be formally represented by a predicate that returns true if a service is available at a given time instant, false otherwise. It enables the modeling of service appearance and disappearance within our time representation.

Definition 2.1 (Service Availability Predicate) *We denote the availability of a service at a given time instant by the* service availability predicate *that returns true if service ω is available at instant τ_i, false otherwise:*

$$available : \begin{array}{ccc} (\Omega, \mathcal{T}) & \rightarrow & \mathcal{B} \\ (\omega, \tau_i) & \mapsto & b \end{array}$$

Example 2.5 (Service Availability) *We represent the availability of services ω_1 ($id(\omega_1) = email$) and ω_2 ($id(\omega_2) = jabber$) between τ_{k_0} and τ_{k_0+14} by the value of the availability function. For example, ω_2 is initially unavailable at τ_{k_0} and appears at instant τ_{k_0+4}, whereas the service ω_1 is initially available, disappears at instant τ_{k_0+8} and reappears at τ_{k_0+11}*

$$
\begin{array}{l}
available(\omega_1, \tau_{k_0}) = true \\
available(\omega_2, \tau_{k_0}) = false
\end{array}
$$

$$
\begin{array}{l}
available(\omega_1, \tau_{k_0+3}) = available(\omega_1, \tau_{k_0+2}) = available(\omega_1, \tau_{k_0+1}) = true \\
available(\omega_2, \tau_{k_0+3}) = available(\omega_2, \tau_{k_0+2}) = available(\omega_2, \tau_{k_0+1}) = false
\end{array}
$$

$$
\begin{array}{l}
available(\omega_1, \tau_{k_0+4}) = true \\
available(\omega_2, \tau_{k_0+4}) = true
\end{array}
$$

$$
\begin{array}{l}
available(\omega_1, \tau_{k_0+7}) = available(\omega_1, \tau_{k_0+6}) = available(\omega_1, \tau_{k_0+5}) = true \\
available(\omega_2, \tau_{k_0+7}) = available(\omega_2, \tau_{k_0+6}) = available(\omega_2, \tau_{k_0+5}) = true
\end{array}
$$

$$
\begin{array}{l}
available(\omega_1, \tau_{k_0+8}) = false \\
available(\omega_2, \tau_{k_0+8}) = true
\end{array}
$$

$$
\begin{array}{l}
available(\omega_1, \tau_{k_0+10}) = available(\omega_1, \tau_{k_0+9}) = false \\
available(\omega_2, \tau_{k_0+10}) = available(\omega_2, \tau_{k_0+9}) = true
\end{array}
$$

$$
\begin{array}{l}
available(\omega_1, \tau_{k_0+11}) = true \\
available(\omega_2, \tau_{k_0+11}) = true
\end{array}
$$

$$
\begin{array}{l}
available(\omega_1, \tau_{k_0+14}) = available(\omega_1, \tau_{k_0+13}) = available(\omega_1, \tau_{k_0+12}) = true \\
available(\omega_2, \tau_{k_0+14}) = available(\omega_2, \tau_{k_0+13}) = available(\omega_2, \tau_{k_0+12}) = true
\end{array}
$$

Properties provided by services can be formally represented by a function associated with each service. For a given service, it returns a value for each attribute, or property, provided by this service.

Definition 2.2 (Service Property Function) *For each service $\omega \in \Omega$, a* service property function *$property_\omega$ is defined:*

$$
property_\omega : \begin{array}{ccc} attributes(\omega) & \rightarrow & \mathcal{D} \\ A & \mapsto & p \end{array}
$$

This function defines the value p for the attribute A, i.e. the value of property A, provided by service ω.

Example 2.6 (Service Property Function) *We represent the value of properties provided by services ω_1 ($id(\omega_1) = email$), ω_2 ($id(\omega_2) = jabber$) and ω_5 ($id(\omega_5) = webcam17$) through the service property function (values are illustrated in Example 2.3):*

- $property_{\omega_1}(protocol) = \text{"SMTP"}$,
- $property_{\omega_2}(protocol) = \text{"XMPP"}$,
- $property_{\omega_5}(location) = \text{"office"}$,
- $property_{\omega_5}(resolution) = \text{"320x240"}$.

Invocations of invocation prototypes on a service, as well as subscriptions to subscription prototypes on a service, can be formally represented by a function associated with each prototype. For a given prototype, this function takes three parameters: a time instant, a constant, that should be a service reference, and a tuple over the prototype input schema, providing the input parameters. It returns a relation over the prototype output schema, representing the output parameters.

Definition 2.3 (Prototype Data Function) *For each prototype $\psi \in \Psi$, a* prototype data function *$data_\psi$ is defined:*

$$data_\psi : \begin{array}{ccc} (\mathcal{D}, \mathcal{D}^{type(Input_\psi)}, \mathcal{T}) & \rightarrow & \mathcal{P}(\mathcal{D}^{type(Output_\psi)}) \\ (s, t, \tau_i) & \mapsto & r \end{array}$$

For an invocation prototype ($streaming(\psi) = false$), this function represents an invocation of prototype ψ on service ω referenced by $s = id(\omega)$, at instant τ_i. The input parameters are defined by the tuple t over $Input_\psi$. The invocation results are represented by a relation r over $Output_\psi$, i.e. a set of tuples containing the output parameters.

For a subscription prototype ($streaming(\psi) = true$), this function represents the content of a subscription to prototype ψ on service ω referenced by $s = id(\omega)$. The subscription is parametrized by the input parameters defined by the tuple t over $Input_\psi$. The content of this subscription at instant τ_i is represented by relation r over $Output_\psi$, i.e. a set of tuples containing the output parameters.

For both types of prototype, if the constant s is not a valid service reference ($\nexists \omega \in \Omega, s = id(\omega)$) or if the referenced service does not implement the prototype ($\psi \notin prototypes(\omega)$), the result is an empty relation at any instant: $data_\psi(s, t, \tau_i) = \emptyset$. If the referenced service is unavailable at a given instant ($available(\omega, \tau_i) = false$), the result is also an empty relation for this instant.

Example 2.7 (Invocation and Subscription Prototypes) *Concerning messaging functionalities, we define two prototypes. An invocation prototype $sendMessage \in \Psi$, like in Example 2.3, enabling to send messages, and a subscription prototype $getMessages \in \Psi$, enabling to receive a stream of messages. Their definitions are:*

- *Prototype $sendMessage \in \Psi$:*
  - $schema(Input_{sendMessage}) = \{address, text\}$,
  - $schema(Output_{sendMessage}) = \{sent\}$,
  - $streaming(sendMessage) = false$;
- *Prototype $getMessages \in \Psi$:*
  - $schema(Input_{getMessages}) = \{username, password\}$,
  - $schema(Output_{getMessages}) = \{sender, message\}$,
  - $streaming(getMessages) = true$.

For example, we can send a message "Bonjour!" to Carla at instant τ_{13}, and a message "Urgence!" to François at instant τ_{15} and at instant τ_{16}. The first and third invocations return one tuple with "true", the second invocation returns one tuple with "false".

$$t_1 \in \mathcal{D}^2, t_1 = \langle \text{"carla@elysee.fr"}, \text{"Bonjour!"} \rangle$$
$$data_{sendMessage}(email, t_1, \tau_{13}) = \{\langle true \rangle\}$$

$$t_2 \in \mathcal{D}^2, t_2 = \langle \text{"francois@im.gouv.fr"}, \text{"Urgence!"} \rangle$$
$$data_{sendMessage}(jabber, t_2, \tau_{15}) = \{\langle false \rangle\}$$
$$data_{sendMessage}(jabber, t_2, \tau_{16}) = \{\langle true \rangle\}$$

We can also subscribe to the message stream of these two users (with their logins and passwords). We subscribe to Carla's messages from instant τ_{10}, and to François's messages from instant τ_{12}. We keep both subscriptions until instant τ_{17} included. A first message is received by Carla at τ_{13} ("Bonjour!"), another is received by François at τ_{16} ("Urgence!") and third and fourth messages are received by Carla at τ_{17} ("Salut!" and "Buongiorno.").

$$t_3 \in \mathcal{D}^2, t_3 = \langle \text{"carla@elysee.fr"}, \text{"nicolito"} \rangle$$
$$t_4 \in \mathcal{D}^2, t_4 = \langle \text{"francois@im.gouv.fr"}, \text{"MaTiGnoN"} \rangle$$

$$\begin{aligned}
data_{getMessages}(email, t_3, \tau_{10}) &= \emptyset \\
data_{getMessages}(email, t_3, \tau_{11}) &= \emptyset \\
data_{getMessages}(email, t_3, \tau_{12}) &= \emptyset \\
data_{getMessages}(jabber, t_4, \tau_{12}) &= \emptyset \\
data_{getMessages}(email, t_3, \tau_{13}) &= \{\langle \text{"SoCQ System"}, \text{"Bonjour!"} \rangle\} \\
data_{getMessages}(jabber, t_4, \tau_{13}) &= \emptyset \\
data_{getMessages}(email, t_3, \tau_{14}) &= \emptyset \\
data_{getMessages}(jabber, t_4, \tau_{14}) &= \emptyset \\
data_{getMessages}(email, t_3, \tau_{15}) &= \emptyset \\
data_{getMessages}(jabber, t_4, \tau_{15}) &= \emptyset \\
data_{getMessages}(email, t_3, \tau_{16}) &= \emptyset \\
data_{getMessages}(jabber, t_4, \tau_{16}) &= \{\langle \text{"SoCQ System"}, \text{"Urgence!"} \rangle\} \\
data_{getMessages}(email, t_3, \tau_{17}) &= \{\langle \text{"Nicolas"}, \text{"Salut!"} \rangle, \langle \text{"Silvio"}, \text{"Buongiorno."} \rangle\} \\
data_{getMessages}(jabber, t_4, \tau_{17}) &= \emptyset
\end{aligned}$$

2.3 The relational pervasive environment

So far, we have introduced how to model time and distributed functionalities to meet pervasive application requirements. We have now to put everything altogether, along with traditional data sources, within a data-centric representation of a pervasive environment. In this section, we build a homogeneous model, namely the relational pervasive environment, composed of data sources extended with services, namely XD-Relations (standing for eXtended Dynamic Relations).

2.3.1 Integrating data and services

In order to seamlessly integrate data sources and services in a simple and homogeneous model, we propose an integration at two levels: at the metadata level, prototypes are integrated into relation schemas; at the data level, services are integrated into tuples.

At the data level, **service references**, that identify services, are conventional data values (e.g., integer values, or string values like in Example 2.3). Attributes representing service references are not different from other attributes, and tuples can contain those service references in the same way as they contain conventional data values for other attributes.

At the metadata level, we integrate prototypes into relation schemas through two notions: **virtual attributes** and **binding patterns**. In general, virtual attributes represent attributes whose value is not stored, but can be computed from other available data or by an external function, or retrieved from an external source. Binding patterns have been introduced in the context of data integration as a description of limited access patterns to relational datasources [FLMS99]. We adapt those two notions in the context of pervasive environments.

We extend relation schemas with virtual attributes: virtual attributes are defined in the relation schema but do not have a value at the data level, i.e. in tuples. They can be transformed into "real" attributes, i.e. non-virtual attributes, through some query operators of our algebra (defined in Chapter 3). In other words, extending a relation schema with virtual attributes does not influence the tuple representation for its relations: virtual attributes only represent potential attributes that can be used by queries.

Binding patterns are the relationship between service references, virtual attributes and prototypes. A binding pattern is associated with a relation schema and defines which prototype to invoke, or subscribe to, on services in order to retrieve values for one or more virtual attributes. It also specifies the real attribute representing service references. The prototype, with its input and output schemas, indicates which attributes from the relation schema are input parameters and output parameters. Input parameters are real or virtual attributes, whereas output parameters are always virtual attributes.

A binding pattern is said to be **active** if its associated prototype is active, **passive** otherwise. It is a **subscription binding pattern** if its associated prototype is a subscription prototype, an **invocation binding pattern** otherwise. Associating binding patterns with a relation schema does not influence the tuple representation for its relations: binding patterns represent a potential way to provide values for virtual attributes that can be used by queries to interact with services.

Using those two notions, we build the notion of **extended relation**, or **X-Relation**, as a relation over an **extended relation schema**, i.e. a relation schema extended with virtual attributes and associated with binding patterns. We can then build the notion of **extended dynamic relation**, or **XD-Relation**, as a dynamic relation over an **extended dynamic relation schema**. In a similar way to the concept of database, we finally build the notion of **relational pervasive environment** as a set of XD-Relations.

XD-Relations, with virtual attributes and active/passive invocation/subscription binding patterns, turn out to be quite powerful to represent many kinds of heterogeneous data sources and distributed functionalities of a pervasive environment. It is worth noting that a XD-Relation with no virtual attribute (and then no binding pattern) is simply a dynamic relation, representing either a conventional relation or a data stream. Clearly, our representation enables to handle conventional and non-conventional data sources in a homogeneous framework, which is one of the requirements of pervasive applications.

We first present some X-Relations in Example 2.8. X-Relations are however an intermediary structure that represents the instantaneous content of XD-Relations. We then present some XD-Relations from the relational pervasive environment for the temperature surveillance scenario in Example 2.9. For the representation of X-Relations and XD-Relations, we use the Serena DDL that is fully introduced later in Section 2.4.

Example 2.8 (X-Relations) *Using prototypes and services described in Example 2.3, some X-Relations can be build in the relational pervasive environment for the temperature surveillance scenario. They are represented in Table 2.2 using the Serena DDL.*

The X-Relation contact represents an electronic contact list (attributes name and address). It is associated with a binding pattern using the sendMessage prototype: it enables a user to send a message (virtual attribute text) to some contact and to get the send confirmation (virtual attribute sent) using the service identified by the attribute messenger. The real attribute address is also used by the binding pattern.

The X-Relation cameras represents a set of cameras. It describes their location (attribute area of the camera) and is associated with two binding patterns: they enable to get which quality are possible to take a photo and with which delay, and to take a photo with a given quality.

Example 2.9 (Relational Pervasive Environment) *The relational pervasive environment for the temperature surveillance scenario is represented in Table 2.3 using the Serena DDL. X-Relations contacts and cameras from Example 2.8 are now considered as finite XD-Relations without any change in their definition.*

Table 2.2: Description of X-Relations from the Relational Pervasive Environment for the temperature surveillance scenario

```
RELATION contacts (
  name      STRING,
  address   STRING,
  messenger SERVICE,
  text      STRING   VIRTUAL,
  sent      BOOLEAN  VIRTUAL
)
USING BINDING PATTERNS (
  sendMessage[messenger] ( address, text ) : ( sent )
);

RELATION cameras (
  camera  SERVICE,
  area    STRING,
  quality INTEGER  VIRTUAL,
  delay   REAL     VIRTUAL,
  photo   BLOB     VIRTUAL
)
USING BINDING PATTERNS (
  checkPhoto[camera] ( area ) : ( quality, delay ),
  takePhoto[camera] ( area, quality ) : ( photo )
);
```

Another finite XD-Relation messengers represents a set of available messaging services. It is associated with two binding patterns: an invocation binding pattern using the sendMessage prototype, that enables to send a message to a given address (like in the contacts XD-Relation), and a subscription binding pattern using the getMessages prototype, that enables to subscribe to a stream of messages given a username and password, i.e. receiving messages (along with sender addresses) from a messaging account. Invocations and subscriptions are however handled at the query level: the definition of those binding patterns represent only potential interactions with services.

An infinite XD-Relation temperatures represents a stream of temperature notifications along with the location of the measures. This XD-Relation has no virtual attribute (and no binding pattern): it does not define potential interactions with services and is then similar to an infinite dynamic relation.

In the Serena DDL, finite and infinite XD-Relations are respectively declared using the common terms RELATION *and* STREAM*, and subscription binding patterns are identified by the* STREAMING *keyword.*

2.3.2 Definitions and notations

Using previous definitions for relations, prototypes, services and service references, we propose a formal definition of dynamic relations extended with virtual attributes and

Table 2.3: Description of XD-Relations from the Relational Pervasive Environment for the temperature surveillance scenario

```
STREAM temperatures (
  area        STRING,
  temperature REAL
);

RELATION messengers (
  messenger SERVICE,
  protocol  STRING,
  address   STRING   VIRTUAL,
  text      STRING   VIRTUAL,
  sent      BOOLEAN  VIRTUAL,
  username  STRING   VIRTUAL,
  password  STRING   VIRTUAL,
  sender    STRING   VIRTUAL,
  message   STRING   VIRTUAL
)
USING BINDING PATTERNS (
  sendMessage[messenger] ( address, text ) : ( sent ),
  getMessages[messenger] ( username, password ) : ( sender, message ) STREAMING
);
```

binding patterns. We first consider the instantaneous case at a given time instant and define the notion of eXtended Relation, or X-Relation. We then consider the continuous case and build the notion of eXtended Dynamic Relation, or XD-Relation. We finally define the notion of relational pervasive environment.

For the sake of simplicity, we keep the Universal Relation Schema Assumption (URSA) [MUV84] stating that if an attribute appears in several relation schemas, then this attribute represents the same data. In our data model, this assumption concerns attributes from all schemas: prototypes input and output schemas, (dynamic) relation schemas, and extended (dynamic) relation schemas; as well as attributes from service properties.

2.3.2.1 X-Relations

An extended relation is basically defined like a relation. However, its schema is partitioned between a real schema containing real attributes and a virtual schema containing virtual attributes. An extended relation schema is also associated with a finite set of binding patterns: each binding pattern specifies a prototype and an attribute from the real schema representing a service reference. Tuples from an extended relation are defined only over the real schema, i.e. the subset of real attributes, as virtual attributes do not have a value. We then define the notion of extended relation schema, and the notion of extended relation over an extended relation schema.

Definition 2.4 (Extended Relation Schema) *An* extended relation schema *is an extended relation symbol R associated with:*

- $type(R)$ *the number of attributes in R,*
- $att_R : \{1, ..., type(R)\} \mapsto \mathcal{A}$ *a one-to-one (i.e. injective) function mapping numbers to attributes in R,*
- $schema(R)$ *the set of attributes in R, i.e.* $\{att_R(1), ..., att_R(type(R))\} \subseteq \mathcal{A}$,
- $\{realSchema(R), virtualSchema(R)\}$ *a partition of* $schema(R)$ *with:*
  - $realSchema(R)$ *the real schema, i.e. the subset of real attributes,*
  - $virtualSchema(R)$ *the virtual schema, i.e. the subset of virtual attributes;*
- $BP(R) \subset (\Psi \times \mathcal{A})$ *a finite set of binding patterns associated with R, where* $bp = \langle prototype_{bp}, service_{bp} \rangle \in BP(R)$ *with:*
  - $prototype_{bp} \in \Psi$ *the prototype associated with the binding pattern,*
  - $service_{bp} \in realSchema(R)$ *a real attribute from the schema used as a service reference attribute for the binding pattern,*

  constrained by the following restrictions:
  - $schema(Input_{prototype_{bp}}) \subset schema(R)$,
  - $schema(Output_{prototype_{bp}}) \subseteq virtualSchema(R)$.

We denote by $active(bp)$, *with* $bp \in BP(R)$, *a predicate that returns true if the binding pattern* bp *is active, i.e. if its associated prototype is active.*

Definition 2.5 (Extended Relation) *A* tuple over an extended relation schema *R is an element of* $\mathcal{D}^{|realSchema(R)|}$. *An* extended relation over *R (or X-Relation over R) is a finite set of tuples over R.*

Tuples from X-Relations can be projected onto an attribute A_i or, by generalization, onto a group of attributes X. However, as virtual attributes do not have a value, tuples can only be projected onto real attributes: the coordinate corresponding to the i^{th} attribute is the j^{th} coordinate where j is the number of real attributes in the partial schema $\{att_R(1), ..., att_R(i)\}$. We denote this number by $\delta_R(i)$. To keep the generalization, tuples can be projected onto a group of real attributes $X \subseteq realSchema(R)$.

Definition 2.6 (Projection of a Tuple) *The* projection of a tuple t, *from a X-Relation r over an extended relation schema R, onto an attribute* $A_i \in realSchema(R)$ *with* $A_i = att_R(i)$ *in* $schema(R)$, *noted* $t[A_i]$, *is the* j^{th} *coordinate of t, i.e.* $t(j)$, *with* $j = \delta_R(i)$ *the number of real attributes in* $\{att_R(1), ..., att_R(i)\}$, *i.e.* $\delta_R(i) = |\{att_R(1), ..., att_R(i)\} \cap realSchema(R)|$.

The projection of a tuple t onto $X = \{att_R(i_1), ..., att_R(i_n)\} \subseteq realSchema(R)$, *noted* $t[X]$, *is* $t[X] = \langle t(\delta_R(i_1)), ..., t(\delta_R(i_n)) \rangle$.

Example 2.10 (Extended Relation) *The electronic contact list from the temperature surveillance scenario can be modelled by an extended relation schema Contact with the following attributes:*

$$schema(Contact) = \{name, address, text, messenger, sent\},$$
$$realSchema(Contact) = \{name, address, messenger\},$$
$$virtualSchema(Contact) = \{text, sent\}.$$

messenger is a service reference attribute. text and sent are two virtual attributes representing the text to be sent and the result of the sending. It is associated with one binding pattern $\langle sendMessage, messenger\rangle$. Let contacts be a X-Relation over Contact, it can be represented in the following table, where '$\star$' denotes the absence of value for virtual attributes:

name	*address*	*text*	*messenger*	*sent*
Nicolas	*nicolas@elysee.fr*	$\star$	*email*	$\star$
Carla	*carla@elysee.fr*	$\star$	*email*	$\star$
François	*francois@im.gouv.fr*	$\star$	*jabber*	$\star$

This X-Relation contains three tuples, defined as elements of

$$\mathcal{D}^{|realSchema(Contact)|} = \mathcal{D}^3.$$

Let $t \in \mathcal{D}^3$ be the first tuple from the table:

- $t = \langle Nicolas, nicolas@elysee.fr, email\rangle$
- $t[messenger] = t[att_{Contact}(4)] = \langle t(\delta_{Contact}(4))\rangle = \langle t(3)\rangle = \langle email\rangle$
- $t[address, messenger] = t[att_{Contact}(2), att_{Contact}(4)] = \langle t(\delta_{Contact}(2)), t(\delta_{Contact}(4))\rangle = \langle t(2), t(3)\rangle = \langle nicolas@elysee.fr, email\rangle$

2.3.2.2 XD-Relations

Using the previous definitions for X-Relations, we now define the notion of XD-Relation that takes into consideration our representation of time. It is similar to the definition of dynamic relations based on the definition of standard relations.

Definition 2.7 (Extended Dynamic Relation Schema) *An* extended dynamic relation schema *R is an extended relation schema R associated with a predicate $infinite(R)$ returning true if R is an infinite extended dynamic relation (i.e. an extended data stream), false if R is a finite extended dynamic relation (i.e. an extended relation).*

A tuple over an extended dynamic relation schema R is a tuple over the extended relation schema R, i.e. an element of $\mathcal{D}^{|realSchema(R)|}$.

In a similar way to dynamic relations, a XD-Relation r is a sequence of extended relations defined from its specific start instant τ_{start_r} until τ_{now}. For each instant τ_i, a XD-Relation contains an extended relation, namely the instantaneous extended relation, that is either finite or infinite as the XD-Relation schema is itself either finite or infinite. In addition to the instantaneous extended relation, two related sets are defined for each instant τ_i: the set of inserted tuples and the set of deleted tuples. For an infinite XD-Relation, the set of deleted tuples is always empty, as tuples can not be deleted from a data stream.

Definition 2.8 (Extended Dynamic Relation) *An* extended dynamic relation *r over an extended dynamic relation schema R is associated with a start instant $\tau_{start_r} \in \mathcal{T}$ and is defined by three related mappings from $\mathcal{T}$ to sets of tuples over R:*

- *r^* is a mapping from $\mathcal{T}$ to sets of tuples over R for $\tau_i \geq \tau_{start_r}$:*
  - *$r^*(\tau_i)$ is the* instantaneous extended relation *of r,*
  - *$r^*(\tau_{start_r})$ is the* initial instantaneous extended relation *of r,*
  - *if R is finite (respectively infinite), r^* maps to finite sets (respectively infinite sets) of tuples over R ;*
- *r^+ and r^- are mappings from $\mathcal{T}$ to finite sets of tuples over R for $\tau_i \geq \tau_{start_r}$:*
  - *$r^+(\tau_i)$ is the set of tuples over R inserted into r at instant τ_i,*
  - *$r^-(\tau_i)$ is the set of tuples over R deleted from r at instant τ_i,*
  - *with the following initial conditions:*
    - *$r^+(\tau_{start_r}) = r^*(\tau_{start_r})$,*
    - *$r^-(\tau_{start_r}) = \emptyset$;*
- $\forall \tau_i > \tau_{start_r}, r^*(\tau_i) = (r^*(\tau_{i-1}) \cup r^+(\tau_i)) - r^-(\tau_i)$
- $\forall \tau_i > \tau_{start_r}, infinite(R) \Rightarrow r^-(\tau_i) = \emptyset$

Example 2.11 (Infinite Extended Dynamic Relation) *The stream of temperature notifications from Example 2.9 is modelled by an infinite extended dynamic relation schema Temperatures with the following attributes:*

$$\begin{aligned} schema(Temperatures) &= \{area, temperature\}, \\ realSchema(Temperatures) &= schema(Temperatures), \\ virtualSchema(Temperatures) &= \emptyset. \end{aligned}$$

This XD-Relation schema has no virtual attribute and is associated with no binding pattern:

$$BP(Temperatures) = \emptyset.$$

As it is infinite, we have:

$$infinite(Temperatures) = true.$$

Let temperatures be a XD-Relation over Temperatures, with

$$\tau_{start_{temperatures}} = \tau_{i_0}.$$

Tuples from temperatures are defined as elements of:

$$\mathcal{D}^{|realSchema(Temperatures)|} = \mathcal{D}^2.$$

It starts at τ_{i_0} with an empty set of tuples. Two tuples are inserted at τ_{i_0+1}, and two others at τ_{i_0+2}. No tuple is inserted until τ_{i_0+6}, where one more tuple is inserted. The evolution of its content can be represented by its instantaneous X-Relation $temperatures^*(\tau_i)$ at different instants, along with its set of inserted tuples $temperatures^+(\tau_i)$. Its set of deleted tuples $temperatures^-(\tau_i)$ is always empty as temperatures is an infinite XD-Relation.

$$\begin{array}{rcl}
temperatures^*(\tau_{i_0}) & = & \emptyset \\
temperatures^+(\tau_{i_0}) & = & temperatures^*(\tau_{i_0}) = \emptyset \\
temperatures^-(\tau_{i \geq i_0}) & = & \emptyset \text{ as } infinite(Temperatures) = true \\
 & & \\
temperatures^*(\tau_{i_0+1}) & = & \{\langle roof, 12.3\rangle, \langle corridor, 25.1\rangle\} \\
temperatures^+(\tau_{i_0+1}) & = & \{\langle roof, 12.3\rangle, \langle corridor, 25.1\rangle\} \\
 & & \\
temperatures^*(\tau_{i_0+2}) & = & \{\langle roof, 12.3\rangle, \langle corridor, 25.1\rangle, \langle roof, 13.1\rangle, \langle office, 23.4\rangle\} \\
temperatures^+(\tau_{i_0+2}) & = & \{\langle roof, 13.1\rangle, \langle office, 23.4\rangle\} \\
 & & \\
temperatures^*(\tau_{i_0+5}) & = & temperatures^*(\tau_{i_0+4}) = temperatures^*(\tau_{i_0+3}) = temperatures^*(\tau_{i_0+2}) \\
temperatures^+(\tau_{i_0+5}) & = & temperatures^+(\tau_{i_0+4}) = temperatures^+(\tau_{i_0+3}) = \emptyset \\
 & & \\
temperatures^*(\tau_{i_0+6}) & = & \{\langle roof, 12.3\rangle, \langle corridor, 25.1\rangle, \langle roof, 13.1\rangle, \langle office, 23.4\rangle, \langle office, 26.2\rangle\} \\
temperatures^+(\tau_{i_0+6}) & = & \{\langle office, 26.2\rangle\}
\end{array}$$

Tuples from XD-Relations can also be projected. Let $t \in \mathcal{D}^2$ be the first tuple from $temperatures^*(\tau_{i_0+1})$:

- $t = \langle roof, 12.3\rangle$
- $t[temperature] = t[att_{Temperatures}(2)] = \langle t(\delta_{Temperatures}(2))\rangle = \langle t(2)\rangle = \langle 12.3\rangle$

Example 2.12 (Finite Extended Dynamic Relation) *The set of messaging services from Example 2.9 is modelled by an extended dynamic relation schema Messengers with the following attributes:*

$$\begin{array}{rcl}
schema(Messengers) & = & \{messenger, protocol, address, text, sent, \\
 & & \quad username, password, sender, message\}, \\
realSchema(Messengers) & = & \{messenger, protocol\}, \\
virtualSchema(Messengers) & = & \{address, text, sent, username, \\
 & & \quad password, sender, message\}.
\end{array}$$

As it is finite, we have:

$$infinite(Messengers) = false.$$

This XD-Relation schema has two binding patterns:

$$BP(Messengers) = \{bp1, bp2\} ,$$

with:

- $bp1 = \langle sendMessage, messenger \rangle$,
- $streaming(bp1) = streaming(sendMessage) = false$,
- $bp2 = \langle getMessages, messenger \rangle$,
- $streaming(bp2) = streaming(getMessages) = true$.

Let messengers be a XD-Relation over Messengers with:

$$\tau_{start_{messengers}} = \tau_{j_0}.$$

Tuples from messengers are defined as elements of:

$$\mathcal{D}^{|realSchema(Messengers)|} = \mathcal{D}^2.$$

It starts at τ_{j_0} with two tuples. One tuple is inserted at τ_{j_0+2}, and one tuple is deleted at τ_{j_0+6}. The evolution of its content can be represented by its instantaneous X-Relation $messengers^*(\tau_i)$ at different instants, along with its set of inserted tuples $messengers^+(\tau_i)$ and its set of deleted tuples $messengers^-(\tau_i)$.

$$\begin{array}{lcl}
messengers^*(\tau_{j_0}) & = & \{\langle email, \text{"SMTP+POP3"}\rangle, \langle jabber, \text{"XMPP"}\rangle\} \\
messengers^+(\tau_{j_0}) & = & messengers^*(\tau_{j_0}) = \{\langle email, \text{"SMTP+POP3"}\rangle, \langle jabber, \text{"XMPP"}\rangle\} \\
messengers^-(\tau_{j_0}) & = & \emptyset \\
\\
messengers^*(\tau_{j_0+1}) & = & \{\langle email, \text{"SMTP+POP3"}\rangle, \langle jabber, \text{"XMPP"}\rangle\} \\
messengers^+(\tau_{j_0+1}) & = & \emptyset \\
messengers^-(\tau_{j_0+1}) & = & \emptyset \\
\\
messengers^*(\tau_{j_0+2}) & = & \{\langle email, \text{"SMTP+POP3"}\rangle, \langle jabber, \text{"XMPP"}\rangle, \langle sms, \text{"SMS/GSM"}\rangle\} \\
messengers^+(\tau_{j_0+2}) & = & \langle sms, \text{"SMS/GSM"}\rangle \\
messengers^-(\tau_{j_0+2}) & = & \emptyset \\
\\
messengers^*(\tau_{j_0+5}) & = & messengers^*(\tau_{j_0+4}) = messengers^*(\tau_{j_0+3}) = messengers^*(\tau_{j_0+2}) \\
messengers^+(\tau_{j_0+5}) & = & messengers^+(\tau_{j_0+4}) = messengers^+(\tau_{j_0+3}) = \emptyset \\
messengers^-(\tau_{j_0+5}) & = & messengers^-(\tau_{j_0+4}) = messengers^-(\tau_{j_0+3}) = \emptyset \\
\\
messengers^*(\tau_{j_0+6}) & = & \{\langle email, \text{"SMTP+POP3"}\rangle, \langle sms, \text{"SMS/GSM"}\rangle\} \\
messengers^+(\tau_{j_0+6}) & = & \emptyset \\
messengers^-(\tau_{j_0+6}) & = & \{\langle jabber, \text{"XMPP"}\rangle\} \\
\\
messengers^*(\tau_{j_0+7}) & = & \{\langle email, \text{"SMTP+POP3"}\rangle, \langle sms, \text{"SMS/GSM"}\rangle\} \\
messengers^+(\tau_{j_0+7}) & = & \emptyset \\
messengers^-(\tau_{j_0+7}) & = & \emptyset
\end{array}$$

2.3.2.3 Relational pervasive environment

XD-Relations that have no virtual attribute (and no binding pattern) are similar to dynamic relations. In the same way, X-Relations with no virtual attribute are similar to standard relations. Our model enables to homogeneously represent all kinds of data sources: standard relations (as finite dynamic relations), data streams (as infinite dynamic relations), and services integrated within data sources (as finite or infinite extended dynamic relations). We summarize the notations for the structure of our data model in Table 2.4.

XD-Relations are a mean to represent some parts of a pervasive computing system. We define the notion of relational pervasive environment representing a set of XD-Relations, similarly to the notion of database representing a set of relations.

Definition 2.9 (Relational Pervasive Environment Schema) *A* relational pervasive environment schema *P is a finite set $P = \{R_1, ..., R_n\}$, with R_i an extended dynamic relation schema. We denote by $schema(P)$ the set of all attributes associated with the extended dynamic relation schemas in P, i.e. $schema(P) = \bigcup_{R_i \in P} schema(R_i)$.*

Definition 2.10 (Relational Pervasive Environment) *A* relational pervasive environment *over a relational pervasive environment schema $P = \{R_1, ..., R_n\}$ is a set $p = \{r_1, ..., r_n\}$, with $r_i \in p$ an extended dynamic relation over $R_i \in P$.*

2.4 The Serena DDL

In this section, we describe the whole syntax of the Serena DDL for XD-Relations (used for example in Table 2.2 and Table 2.3). In Table 2.5, a BNF grammar is defined. It is similar to the standard SQL DDL syntax.

An XD-Relation is identified by its name and is either finite or infinite (RELATION and STREAM keywords). It defines a list of typed attributes, that may be virtual. It may be associated with binding patterns, specifying the name of a prototype, the name of a service reference attribute (that should be of type SERVICE), and the names of the input and output attributes. A subscription binding pattern is specified by the STREAMING keyword.

Example 2.13 (Serena DDL) *The Serena DDL for the definition of all XD-Relations described in Example 2.8 and Example 2.9 is presented in Table 2.6.*

Table 2.4: Overview of Notations for the Structure of the SoCQ Data Model

	STRUCTURE
DATA	• Boolean Domain $\mathcal{B}$ • Constants $\mathcal{D}$ • Attributes $\mathcal{A}$ • Relation Schema R – $schema(R) \subset \mathcal{A}$ • Relation r over R – $r \subset \mathcal{D}^{\lvert schema(R) \rvert}$
DATA + SERVICES	• Prototypes $\psi \in \Psi$ – Relation Schema $Input_\psi$ – Relation Schema $Output_\psi$ – $active(\psi) \in \mathcal{B}$ • Services Interfaces $\phi \in \Phi$ – $prototypes_\phi \subset \Psi$ – $attributes_\phi \subset \mathcal{A}$ • Services $\omega \in \Omega$ – $id(\omega) \in \mathcal{D}$ – $prototypes(\omega) \subset \Psi$ – $attributes(\omega) \subset \mathcal{A}$ • Interaction with Services – $available(\omega) \in \mathcal{B}$ – $property_\omega(A) \in \mathcal{D}$ – $data_\psi(id(\omega), input) \in \mathcal{D}^{\lvert schema(Output_\psi) \rvert}$ • X-Relation Schema R – Relation Schema R – $realSchema(R)$ – $virtualSchema(R)$ – $BP(R) \subset (\Psi \times \mathcal{A})$ • X-Relation r over R – $r \subset \mathcal{D}^{\lvert realSchema(R) \rvert}$
DATA + SERVICES + TIME + STREAMS	• Discrete Time Domain $\tau_i \in \mathcal{T}$ • Interaction with Services – $available(\omega, \tau_i) \in \mathcal{B}$ – $data_\psi(id(\omega), input, \tau_i) \in \mathcal{D}^{\lvert schema(Output_\psi) \rvert}$ • XD-Relation Schema R – X-Relation Schema R – $infinite(R) \in \mathcal{B}$ • XD-Relation r over R – $\tau_{start_r} \in \mathcal{T}$ – $r^*(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R) \rvert}$ – $r^+(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R) \rvert}$ – $r^-(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R) \rvert}$

Table 2.5: Serena DDL syntax for XD-Relations

```
<XD_RELATION> ::=
  ( RELATION | STREAM ) <xdrelation_name> '('
    <ATTRIBUTE> ( ',' <ATTRIBUTE> )* ]
  ')'
[
  USING BINDING PATTERNS '('
    <BINDING_PATTERN> ( ',' <BINDING_PATTERN> )* ]
  ')'
];

<ATTRIBUTE> ::=
  <attribute_name> <DATA_TYPE> [ VIRTUAL ];

<DATA_TYPE> ::=
  STRING | INTEGER | REAL | SERVICE | BLOB | ... ;

<BINDING_PATTERN> ::=
  <prototype_name> '[' <attribute_name> ']'
    '(' [ <attribute_name> ( ',' <attribute_name> )* ] ')' ':'
    '(' <attribute_name> ( ',' <attribute_name> )* ')' [ STREAMING ];
```

Table 2.6: Example of XD-Relations defined using the Serena DDL

```
RELATION contacts (
  name      STRING,
  address   STRING,
  messenger SERVICE,
  text      STRING   VIRTUAL,
  sent      BOOLEAN  VIRTUAL
)
USING BINDING PATTERNS (
  sendMessage[messenger] ( address, text ) : ( sent )
);

RELATION cameras (
  camera  SERVICE,
  area    STRING,
  quality INTEGER  VIRTUAL,
  delay   REAL     VIRTUAL,
  photo   BLOB     VIRTUAL
)
USING BINDING PATTERNS (
  checkPhoto[camera] ( area ) : ( quality, delay ),
  takePhoto[camera] ( area, quality ) : ( photo )
);

STREAM temperatures (
  area        STRING,
  temperature REAL
);

RELATION messengers (
  messenger SERVICE,
  protocol  STRING,
  address   STRING   VIRTUAL,
  text      STRING   VIRTUAL,
  sent      BOOLEAN  VIRTUAL,
  username  STRING   VIRTUAL,
  password  STRING   VIRTUAL,
  sender    STRING   VIRTUAL,
  message   STRING   VIRTUAL
)
USING BINDING PATTERNS (
  sendMessage[messenger] ( address, text ) : ( sent ),
  getMessages[messenger] ( username, password ) : ( sender, message ) STREAMING
);
```

2.5 Summary

In this chapter, we have introduced the structure of a data model devoted to fit the requirements of pervasive environment. The so-called relational pervasive environment enables the representation of an actual pervasive environment in a data-oriented way: it enables to achieve a high level of abstraction of distributed functionalities required for the pervasive applications through a coherent integration of time, data and services. More precisely, our data model is built on five basic notions: time, constants, attributes, prototypes and services.

From the standard relational model and our representation of time as an infinite ordered domain of discrete time instants, we have proposed notations to homogeneously represent standard relations and data streams as dynamic relations. We have then defined our representation of distributed functionalities as services implementing invocation and subscription prototypes that may be active or passive. We have finally defined the intermediary notion of extended relation, or X-Relation, to integrate prototypes and services within data relations, in order to build the unifying notion of extended dynamic relation, or XD-Relation, enabling the representation of all kinds of data sources. The relational pervasive environment has then been defined, in a similar way to databases, as a set of XD-Relations representing the available data sources and distributed functionalities. A SQL-like DDL, namely the Serena DDL, has also been defined to represent XD-Relations in a declarative way.

Extended relation schemas have been defined using the notion of virtual attributes and binding patterns. Those two notions are integrated at the metadata level, i.e. they do not modify the tuple representation, and represent potential interactions with data and services. Those potential interactions are dynamic and triggered by query operators in declarative queries: it enables a declarative definition of interactions between distributed functionalities, i.e. the declarative definition of pervasive applications. The definition of such queries is the topic of the next chapter.

3

Querying over a Pervasive Environment

Chapter Outline

3.1 Serena one-shot query algebra **42**
3.1.1 Set operators 42
3.1.2 Relational operators 42
3.1.3 Realization operators 43
3.1.4 Service discovery operator 47
3.1.5 One-shot queries over a relational pervasive environment . . 49
3.2 Serena continuous query algebra **51**
3.2.1 Extension of one-shot query operators 52
3.2.2 Binding operator 53
3.2.3 Stream operators 56
3.2.4 Continuous queries over a relational pervasive environment . 60
3.3 Query equivalence **63**
3.3.1 Action sets 64
3.3.2 Query equivalence 66
3.3.3 Rewriting rules 67
3.4 Query optimization **70**
3.4.1 Metrics of the cost model 70
3.4.2 Estimating the cost of a query 71
3.4.3 Goal of query optimization 74
3.4.4 Rule-based query optimization 74

3.5 The Serena SQL **75**
3.5.1 Queries over XD-Relations 75
3.5.2 Service discovery queries 76
3.6 Summary **79**

In order to achieve our goal of simplifying the development of pervasive applications, the critical next step that remains to be done is the definition of a query language that enables to express both simple and complex interactions within a relational pervasive environment at a declarative level.

In the setting of the relational model, the relational algebra is recognized as a fundamental tool for measuring the expressiveness of a query language, from which the notion of "relational completeness" has been defined [Cod72]. In the setting of pervasive environments, we propose an algebra for relational pervasive environments that defines the “expressiveness” of our query language.

Moreover, this algebra is close to its relational counterpart, from which the definition of a SQL-like language is straightforward. It is an interesting issue to define other query languages, such as logic-based query languages, which are at least as expressive as our proposed algebra. We have nevertheless not explored this non-trivial issue in this thesis.

In this chapter, an algebra for relational pervasive environments, namely the *Serena algebra* (standing for **Ser**vice-**ena**bled algebra), is defined for one-shot and continuous queries over XD-relations: query operators are formally defined, as well as the notion of query itself. The notion of query equivalence is also defined in order to enable query optimization techniques. A SQL-like language, namely the *Serena SQL*, is finally defined.

In Section 3.1, we first define query operators for one-shot queries over relational pervasive environments. Those query operators are defined over the instantaneous X-Relations of involved XD-Relations, considering the instant when the query is launched. Besides standard operators like set operators and relational operators, we define specific operators handling virtual attributes and binding patterns, namely the realization operators: the assignment operator and the binding operator. We also define a service discovery operator enabling the expression of queries over the set of available services. The notion of one-shot query is then formally defined.

In Section 3.2, we define query operators for continuous queries over relational pervasive environments. Among the simple redefinition of one-shot operators over finite XD-Relations, the more complex case of the binding operator is studied. Operators specific to infinite XD-Relations are also defined to homogeneously handle finite and infinite XD-Relations, namely the stream operators: the window operator and the streaming operator. The notion of continuous query is then formally defined.

In Section 3.3, the notion of equivalence between two queries is studied for one-shot and continuous queries, which leads to the definition of rewriting rules for Serena algebra expressions. The notion of active/passive prototypes used by binding patterns plays a key role in this definition.

In Section 3.4, query optimization techniques, based on the query equivalence and rewriting rules, are devised according to a cost model dedicated to relational pervasive environments.

In Section 3.5, the Serena SQL language is defined for one-shot and continuous queries over XD-relations, as well as for the special case of service discovery queries.

3.1 Serena one-shot query algebra

In this section, we define the first half of the Serena algebra, namely one-shot query operators over a relational pervasive environment. We denote by τ_{start_q} the start instant of a one-shot query q, i.e. the instant when the query is launched. Those operators are defined over X-Relations, that are either instantaneous X-Relations of the involved XD-Relations at τ_{start_q} or output X-Relations of another one-shot query operator, as the results of those operators are X-Relations. We first tackle the redefinition of operators from the relational algebra, with set operators and relational operators, then the definition of new operators handling virtual attributes and binding patterns, namely the realization operators. The service discovery operator, dedicated to service discovery, is also defined.

Using the definitions of those operators, the notion of one-shot query over a relational pervasive environment is formally defined. One-shot queries being not pertinent to handle data streams, we then do not consider infinite XD-Relations nor subscription binding patterns for the definition of one-shot queries.

3.1.1 Set operators

The set operators union, intersection and difference can be applied over two X-Relations associated with the same schema. The resulting X-Relation is defined over the same schema. Their definitions over X-Relations remain similar to their definitions over standard relations. Let r_1 and r_2 be two X-Relations over a X-Relation schema R:

- Union: $r_1 \cup r_2 = \{t \mid t \in r_1 \vee t \in r_2\}$
- Intersection: $r_1 \cap r_2 = \{t \mid t \in r_1 \wedge t \in r_2\}$
- Difference: $r_1 - r_2 = \{t \mid t \in r_1 \wedge t \notin r_2\}$

3.1.2 Relational operators

Standard relational operators are defined over one or two standard relations. We extend their definitions over one or two X-Relations. At the data level, their definitions remain very similar. At the metadata level however, the definition of the schema of the result-

ing X-Relations is more specific, because modification or disappearance of some real or virtual attributes can modify or invalidate binding patterns using those attributes.

The *projection* operator (cf. Table 3.1 (a)) reduces the schema of a X-Relation, thus its real and virtual schemas. The resulting relation is associated with binding patterns from the initial X-Relation that remain valid, i.e. binding patterns with their service reference attribute, input and output attributes that are still in the schema of the resulting X-Relation.

The *selection* operator (cf. Table 3.1 (b)) does not modify the schema of the X-Relation. However, selection formulas can only apply on attributes from the real schema, as virtual attributes do not have a value. Apart from this restriction, we keep the standard definition and notation for the logical implication (e.g., in [LL99]), i.e. $t \models F$, with t a tuple and F a selection formula over a relation schema R.

The *renaming* operator (cf. Table 3.1 (c)) replaces an attribute from the schema by another attribute. This operation does not modify the real or virtual status of the renamed attribute. The renaming also impacts on the binding patterns using this attribute: whereas service reference attributes can be simply renamed, binding patterns may be invalidated if their prototypes use the renamed attribute, as prototypes can not be modified.

The *natural join* operator (cf. Table 3.2 (d)) joins two X-Relations, the join attributes being given by the intersection of the two schemas. If a join attribute is a real (respectively virtual) attribute in both operands, it remains real (respectively virtual) in the resulting X-Relation. However, if it is real in one operand and virtual in the other one, it becomes real in the resulting X-Relation (i.e. it is an implicit realization of the virtual attribute, cf. Section 3.1.3). As virtual attributes do not have a value at the data level, only join attributes that are real in both operands imply a join predicate. If all join attributes are virtual in at least one operand, the join is then equivalent, at the data level, to a Cartesian product.

The set of binding patterns for the resulting X-Relation is the union of the sets of binding patterns from both operands, but some may be eliminated if they use output attributes that have become real attributes with the join.

3.1.3 Realization operators

We introduce two new operators, referred to as *realization operators*. They enable to transform virtual attributes into real attributes. Despite the simplicity of their action, they turn out to be key components for seamlessly querying data and services. We call this transformation "realization" of virtual attributes. The reverse transformation is not possible. Realized attributes are given a value by the operators, either directly (assignment operator) or using a binding pattern (binding operator). An implicit realization can also occur in a natural join when a join attribute is real in one operand and virtual

Table 3.1: Definition of the Projection, Selection and Renaming operators over X-Relations

(a)	**Projection**
Input	r a X-Relation over R $Y \subset schema(R)$
Syntax	$s = \pi_Y(r)$
Output	s a X-Relation over S, with: - $schema(S) = Y$ - $realSchema(S) = realSchema(R) \cap Y$ - $virtualSchema(S) = virtualSchema(R) \cap Y$ - $BP(S) = \{bp \mid bp \in BP(R) \wedge service_{bp} \in Y \wedge$ $schema(Input_{prototype_{bp}}) \subset Y \wedge schema(Output_{prototype_{bp}}) \subset Y\}$
Tuples	$s = \{t[Y \cap realSchema(R)] \mid t \in r\}$

(b)	**Selection**
Input	r a X-Relation over R F a selection formula over $realSchema(R)$
Syntax	$s = \sigma_F(r)$
Output	s a X-Relation over R
Tuples	$s = \{t \mid t \in r \wedge t \models F\}$

(c)	**Renaming**
Input	r a X-Relation over R $A \in schema(R)$ $B \in \mathcal{A}, B \notin schema(R)$
Syntax	$s = \rho_{A \to B}(r)$
Output	s a X-Relation over S, with: - $schema(S) = (schema(R) - \{A\}) \cup \{B\}$ - $realSchema(S) = \begin{cases} (realSchema(R) - \{A\}) \cup \{B\} \text{ if } A \in realSchema(R) \\ realSchema(R) \text{ otherwise} \end{cases}$ - $virtualSchema(S) = \begin{cases} (virtualSchema(R) - \{A\}) \cup \{B\} \text{ if } A \in virtualSchema(R) \\ virtualSchema(R) \text{ otherwise} \end{cases}$ - $BP(S) = \{bp' \mid \exists bp \in BP(R), prototype_{bp'} = prototype_{bp} \wedge$ $((A \neq service_{bp} \wedge service_{bp'} = service_{bp}) \vee (A = service_{bp} \wedge service_{bp'} = B)) \wedge$ $schema(Input_{prototype_{bp}}) \subset (schema(R) - \{A\}) \cup \{B\} \wedge$ $schema(Output_{prototype_{bp}}) \subset (schema(R) - \{A\}) \cup \{B\}\ \}$
Tuples	$s = \{t \mid \exists u \in r, t[realSchema(S) - \{B\}] = u[realSchema(R) - \{A\}] \wedge$ $t[\{B\} \cap realSchema(S)] = u[\{A\} \cap realSchema(R)]\}$

Table 3.2: Definition of Natural Join operator over X-Relations

(d)	**Natural Join**
Input	r_1 a X-Relation over R_1 r_2 a X-Relation over R_2
Syntax	$s = r_1 \bowtie r_2$
Output	s a X-Relation over S, with: - $schema(S) = schema(R_1) \cup schema(R_2)$ - $realSchema(S) = realSchema(R_1) \cup realSchema(R_2)$ - $virtualSchema(S) = (virtualSchema(R_1) - realSchema(R_2)) \cup$ $(virtualSchema(R_2) - realSchema(R_1))$ - $BP(S) = \{bp \mid bp \in (BP(R_1) \cup BP(R_2)) \wedge$ $schema(Output_{prototype_{bp}}) \subseteq (virtualSchema(R_1) - realSchema(R_2)) \cup$ $(virtualSchema(R_2) - realSchema(R_1))\ \}$
Tuples	$s = \{t \mid \exists t_1 \in r_1, \exists t_2 \in r_2, t[realSchema(R_1)] = t_1 \wedge t[realSchema(R_2)] = t_2\}$

in the other one: the virtual attribute from the operand becomes a real attribute in the resulting X-Relation.

The *assignment* operator (cf. Table 3.3 (e)) over a X-Relation is the realization operator for individual virtual attributes. It enables to give a value to one virtual attribute. In a similar way to simple selection formulas, this operator enables to assign to a virtual attribute either the value of a real attribute from the schema or a constant value. The virtual attribute thus becomes a real attribute in the resulting X-Relation. The binding patterns for the resulting relation are those of the operand, but some may be eliminated when the assigned attribute is part of their output attributes, as it is now a real attribute.

The *binding* operator (cf. Table 3.3 (f)) over a X-Relation is the realization operator for the output attributes of an invocation binding pattern. Invoking the binding pattern, i.e. invoking the associated prototype on services identified by the service reference attribute, enables to retrieve values for those output attributes. However, this operator can only be applied if all the input attributes of the binding pattern are real attributes in the input X-Relation.

Each tuple from the operand leads to an invocation of the associated prototype $prototype_{bp}$, represented by the function $data_{prototype_{bp}}$ with the query start instant τ_{start_q}, which may result in several tuples for output attributes. Each input tuple is duplicated as many times as the invocation has generated output tuples. For each invocation, the input parameters and the service reference are taken from the input tuple.

Table 3.3: Definition of realization operators over X-Relations

(e)	**Assignment**
Input	r a X-Relation over R $A \in virtualSchema(R)$ $B \in realSchema(R)$ or $a \in \mathcal{D}$
Syntax	$s = \alpha_{A \equiv B}(r)$ or $s = \alpha_{A \equiv a}(r)$
Output	s a X-Relation over S, with: - $schema(S) = schema(R)$ - $realSchema(S) = realSchema(R) \cup \{A\}$ - $virtualSchema(S) = virtualSchema(S) - \{A\}$ - $BP(S) = \{bp \mid bp \in BP(R) \wedge$ $schema(Output_{prototype_{bp}}) \subseteq (virtualSchema(R) - \{A\})\}$
Tuples	$s = \{t \mid \exists u \in r, t[realSchema(S) - \{A\}] = u[realSchema(R)] \wedge t[A] = u[B]\}$ $s = \{t \mid \exists u \in r, t[realSchema(S) - \{A\}] = u[realSchema(R)] \wedge t[A] = a\}$

(f)	**Binding**
Input	r a X-Relation over R $bp \in BP(R), streaming(bp) = false$ $schema(Input_{prototype_{bp}}) \subset realSchema(R)$
Syntax	$s = \beta_{bp}(r)$
Output	s a X-Relation over S, with: - $schema(S) = schema(R)$ - $realSchema(S) = realSchema(R) \cup schema(Output_{prototype_{bp}})$ - $virtualSchema(S) = virtualSchema(R) - schema(Output_{prototype_{bp}})$ - $BP(S) = \{bp' \mid bp' \in BP(R) \wedge$ $schema(Output_{prototype_{bp'}}) \subseteq (virtualSchema(R) - schema(Output_{prototype_{bp}}))\}$
Tuples	$s = \{t \mid \exists u \in r, t[realSchema(S) - schema(Output_{prototype_{bp}})] = u[realSchema(R)] \wedge$ $t[schema(Output_{prototype_{bp}})] \in$ $data_{prototype_{bp}}(u[service_{bp}], u[schema(Input_{prototype_{bp}})], \tau_{start_q})\}$

3.1.4 Service discovery operator

In order to fully integrate into queries distributed functionalities from a dynamic environment, an operator for dynamic service discovery is required. Such an operator enables to consider the relational pervasive environment itself as a datasource providing a set of services: this set of services can be queried in order to extract a subset of services.

The *service discovery* operator (cf. Table 3.4 (g)) represents the selection of a list of available services that match some criteria. It has no X-Relation operand: its unique "operand" is implicitly the set of services Ω.

This operator selects services that implement a given service interface, i.e. that provide some invocation and/or subscription prototypes and some properties (cf. Section 2.2). In fact, two service interfaces are considered for one service discovery operator. The *explicit service interface* is used to build the schema of the resulting X-Relation with the required attributes and binding patterns corresponding to this interface. The *implicit service interface* is used only for the selection of services, along with the explicit service interface, but does not contribute to the X-Relation schema. It enables to select only services providing some additional properties or prototypes that are nevertheless not useful for the rest of a query and can remain safely hidden.

The resulting X-Relation schema has one SERVICE attribute, representing the service reference, and some real attributes, representing the explicitly selected service properties. It is associated with a binding pattern for each explicitly selected prototype and has the needed virtual attributes for their input and output attributes. The resulting X-Relation contains one tuple for each available service that matches the selection criteria from both explicit and implicit service interfaces: the tuple contains the service reference of this service and the values it provides for the selected properties.

Example 3.1 (Serena one-shot service discovery operator) *For example, we want to select all services representing cameras that are available in the environment. We assume that camera services provide (at least) a property location and implement (at least) two prototypes checkPhoto and takePhoto. Those prototypes have already been presented in the previous chapter (Example 2.3 page 18). Using a single service discovery operator with this given explicit interface (and no implicit interface), we can build a X-Relation that contains one tuple for each camera service.*

$$\zeta_{camera,\langle\{location\},\{checkPhoto,takePhoto\}\rangle,\langle\emptyset,\emptyset\rangle}()$$

The X-Relation schema contains two real attributes: one service reference attribute camera, and one real attribute location. It is associated with two binding patterns using the two prototypes checkPhoto and takePhoto, and contains the virtual attributes required for those binding patterns. All binding patterns use the service reference attribute camera. Let R be this resulting X-Relation schema:

Table 3.4: Definition of the Service Discovery operator

(g)	**Service Discovery** (with explicit/implicit service interfaces ϕ and ϕ')
Input	$A \in \mathcal{A}$ $\phi = \langle attributes_\phi, prototypes_\phi \rangle \in \Phi$ $\phi' = \langle attributes_{\phi'}, prototypes_{\phi'} \rangle \in \Phi$
Syntax	$s = \xi_{A,\phi,\phi'}()$
Output	s a X-Relation over S, with: - $schema(S) = \{A\} \cup attributes_\phi \cup$ $\bigcup_{\psi \in prototypes_\phi} (schema(Input_\psi) \cup schema(Output_\psi))$ - $realSchema(S) = \{A\} \cup attributes_\phi$ - $virtualSchema(S) = \bigcup_{\psi \in prototypes_\phi} (schema(Input_\psi) \cup schema(Output_\psi)) -$ $(\{A\} \cup attributes_\phi)$ - $BP(S) = \{bp = \langle prototype_{bp}, service_{bp} \rangle \mid service_{bp} = A \wedge$ $prototype_{bp} \in prototypes_\phi \wedge$ $schema(Output_{prototype_{bp}}) \cap (\{A\} \cup attributes_\phi) = \emptyset \}$
Tuples	$s = \{t \mid \exists \omega \in \Omega, t[A] = id(\omega) \wedge available(\omega, \tau_{start_q}) \wedge$ $(prototypes_\phi \cup prototypes_{\phi'}) \subseteq prototypes(\omega) \wedge$ $(attributes_\phi \cup attributes_{\phi'}) \subseteq attributes(\omega) \wedge$ $(\forall B \in attributes_\phi, t[B] = property_\omega(B)) \}$

$$
\begin{aligned}
schema(R) &= \{camera, location, area, quality, delay, photo\}\,, \\
realSchema(R) &= \{camera, location\}\,, \\
virtualSchema(R) &= \{area, quality, delay, photo\}\,, \\
BP(R) &= \{\langle checkPhoto, camera\rangle, \langle takePhoto, camera\rangle\}.
\end{aligned}
$$

Note that the real attribute location comes from the service property: it represents the actual location of the camera; whereas the virtual attribute area is an input for the binding patterns: it represents the area where the camera should point at to take a photo.

The content of this resulting X-Relation could be:

camera	*location*	*area*	*quality*	*delay*	*photo*
camera01	*office*	$\star$	$\star$	$\star$	$\star$
camera02	*corridor*	$\star$	$\star$	$\star$	$\star$
webcam17	*office*	$\star$	$\star$	$\star$	$\star$

We can use the implicit service interface in order to add another selection criterion without modifying the resulting X-Relation schema and binding patterns. For example, we can add the attribute resolution to the previously empty implicit interface:

$$\xi_{camera,\langle\{location\},\{checkPhoto,takePhoto\}\rangle,\langle\{resolution\},\emptyset\rangle}()$$

The resulting X-Relation then contains one tuple only for each service that additionally provides the property resolution, for example:

camera	*location*	*area*	*quality*	*delay*	*photo*
webcam17	*office*	$\star$	$\star$	$\star$	$\star$

3.1.5 One-shot queries over a relational pervasive environment

The notion of one-shot query over a relational pervasive environment can now be defined as a composition of one-shot query operators over a set of X-Relations. A one-shot query q is associated with its start instant, i.e. the instant when the query is launched, denoted τ_{start_q}. The involved X-Relations are either instantaneous X-Relations at τ_{start_q} of finite XD-Relations, or output X-Relations of other one-shot queries.

Definition 3.1 (One-shot Query) *Let p be a relational pervasive environment. A* one-shot query *q over p is a well-formed expression composed of a finite number of Serena one-shot query algebra operators over X-Relations. It is associated with a start instant, denoted $\tau_{start_q} \in \mathcal{T}$.*

Example 3.2 (Serena one-shot query) *We can express the following one-shot queries over finite XD-Relations contacts and cameras (introduced in Example 2.8 and 2.9 page 25, defined in Example 2.10 page 29):*

Q_1 *send the message "Bonjour!" to all contacts, except "Carla";*
Q_2 *take photos of area "office" with quality superior to "5".*

Query Q_1 is composed of three operators and uses the XD-Relation contacts. A selection operator discards the contact whose name is "Carla". An assignment operator sets the text of the message, "Bonjour!", in the virtual attribute text. Finally, a binding operator invokes the binding pattern associated with the prototype sendMessage, therefore realizing the attribute sent.

$$Q_1 = \beta_{\langle sendMessage,messenger\rangle}(\alpha_{text\equiv"Bonjour!"}(\sigma_{name\neq"Carla"}(contacts)))$$

Query Q_2 is composed of five operators and uses the XD-Relation cameras. A selection operator selects cameras whose "area" is the office. A binding operator invokes the binding pattern associated with the prototype checkPhoto, realizing the two attributes quality and delay. Another selection operator then tests the value of the attribute quality. A second binding operator invokes the binding pattern associated with the prototype takePhoto, realizing the attribute photo. A final projection operator enables the reduction of the resulting X-Relation schema to the single attribute photo.

Table 3.5: Examples of one-shot queries expressed in the Serena algebra

Q_1	$\beta_{\langle sendMessage,messenger\rangle}(\alpha_{text\equiv\text{"Bonjour!"}}(\sigma_{name\neq\text{"Carla"}}(contacts)))$
Q_2	$\pi_{photo}(\beta_{\langle takePhoto,camera\rangle}(\sigma_{quality>5}(\beta_{\langle checkPhoto,camera\rangle}(\sigma_{area=\text{"office"}}($ $cameras$ $)))))$
Q'_2	$\pi_{photo}(\beta_{\langle takePhoto,camera\rangle}(\sigma_{quality>5}(\beta_{\langle checkPhoto,camera\rangle}(\sigma_{area=\text{"office"}}($ $\alpha_{area\equiv location}(\xi_{camera,\langle\{location\},\{checkPhoto,takePhoto\}\rangle,\langle\emptyset,\emptyset\rangle}())$ $)))))$

$$Q_2 = \pi_{photo}(\beta_{\langle takePhoto,camera\rangle}(\sigma_{quality>5}(\beta_{\langle checkPhoto,camera\rangle}(\sigma_{area=\text{"office"}}(cameras))))))$$

We can also define a query Q'_2 similar to query Q_2, but using a service discovery operator instead of the XD-Relation cameras. This operator selects services providing the property location and implementing the prototypes checkPhoto and takePhoto, like in Example 3.1 (page 47). We however need to add an assignment operator in order to assign the value of the real attribute location to the virtual attribute area used by the binding patterns.

$$Q'_2 = \pi_{photo}(\beta_{\langle takePhoto,camera\rangle}(\sigma_{quality>5}(\beta_{\langle checkPhoto,camera\rangle}(\sigma_{area=\text{"office"}}(\alpha_{area\equiv location}(\xi_{camera,\langle\{location\},\{checkPhoto,takePhoto\}\rangle,\langle\emptyset,\emptyset\rangle}())))))$$

We consider their respective start instants $\tau_{start_{Q_1}}$, $\tau_{start_{Q_2}}$ and $\tau_{start_{Q'_2}}$, e.g., $\tau_{start_{Q_1}} = \tau_{start_{Q_2}} = \tau_{start_{Q'_2}} = \tau_{now} = \tau_{15}$. The Serena algebra expression for those queries are summarized in Table 3.5.

One-shot queries over a relational pervasive environment enable the declarative expression of one-time interactions between data and services. Physical access to data sources and functionalities are fully abstracted. Furthermore, the service discovery operator enables the abstraction of the discovery of available resources. We summarize the different notations for X-Relations and one-shot queries in Table 3.6.

Table 3.6: Overview of Notations for X-Relations and One-shot Queries

	STRUCTURE	LANGUAGE
DATA + SERVICES	• Prototypes $\psi \in \Psi$ – Relation Schema $Input_\psi$ – Relation Schema $Output_\psi$ – $active(\psi) \in \mathcal{B}$ • Services Interfaces $\phi \in \Phi$ – $prototypes_\phi \subset \Psi$ – $attributes_\phi \subset \mathcal{A}$ • Services $\omega \in \Omega$ – $id(\omega) \in \mathcal{D}$ – $prototypes(\omega) \subset \Psi$ – $attributes(\omega) \subset \mathcal{A}$ • Interaction with Services – $available(\omega) \in \mathcal{B}$ – $property_\omega(A) \in \mathcal{D}$ – $data_\psi(id(\omega), input) \in \mathcal{D}^{\lvert schema(Output_\psi)\rvert}$ • X-Relation Schema R – Relation Schema R – $realSchema(R)$ – $virtualSchema(R)$ – $BP(R) \subset (\Psi \times \mathcal{A})$ • X-Relation r over R – $r \subset \mathcal{D}^{\lvert realSchema(R)\rvert}$	• Operators over X-Relation(s) – Selection $\sigma_F(r)$ – Projection $\pi_Y(r)$ – Renaming $\rho_{A\to B}(r)$ – Natural Join $r_1 \bowtie r_2$ – Assignment $\alpha_{A\equiv B}(r)$ / $\alpha_{A\equiv c}(r)$ – Binding $\beta_{\langle\psi,S\rangle}(r)$ • Operator for Service Discovery – Service Discovery $\xi_{S,\phi,\phi'}()$

3.2 Serena continuous query algebra

In this section, we define the second half of the Serena algebra, namely continuous query operators over a relational pervasive environment. We still consider the start instant of a continuous query q, i.e. the instant when the query is started, denoted τ_{start_q}. Those operators are defined over finite or infinite XD-Relations, and their results are also finite or infinite XD-Relations. Subscription binding patterns are also taken into consideration.

The start instant of the query plays an important role: it determines the start instant of the resulting XD-relation for each operator, with the associated initial conditions (cf. *Definition 2.8* page 30). It also impacts on the operand XD-Relations: they are considered to start at τ_{start_q}, i.e. initial conditions are valid at this instant (the deleted tuple set is empty and the inserted tuple set equals to the instantaneous tuple set).

We first tackle the extension of one-shot query operators over finite XD-Relations by simply considering their instantaneous X-Relation. However, the binding opera-

tor needs to be slightly modified for two reasons. First, an invocation binding pattern is actually invoked only for newly inserted tuples, and not for every tuple at each time instant. Second, the operator is also defined for subscription binding patterns: it subscribes to the binding pattern prototype for every newly inserted tuple, and unsubscribes from it for every deleted tuple. In other words, at each time instant, this operator is "listening" to subscriptions for each tuple from its operand instantaneous X-Relation.

We then define two additional operators that handle infinite XD-Relations. The window operator computes a finite XD-Relation from an infinite XD-Relation as, for every time instant, the set of tuples inserted during the last n instants. The streaming operator computes an infinite XD-Relation from a finite XD-Relation by inserting, for every time instant, the set of tuples that are inserted/deleted/present at this instant (depending on the event type of the operator). Those two operators do not modify the XD-Relation schema, apart from its finite/infinite status: they transparently handle virtual attributes and binding patterns.

All operators are defined over finite XD-Relations, except the window operator that is defined over an infinite XD-Relation. All operators build a finite XD-Relation as a result, except the binding operator for subscription binding patterns and the streaming operator that build an infinite XD-Relation. Finally, using the definitions of those operators, the notion of continuous query over a relational pervasive environment is formally defined.

3.2.1 Extension of one-shot query operators

The definition of the set operators (union, intersection, difference), the relational operators (selection, projection, renaming, natural join), and the assignment operator over finite XD-Relations is rather straightforward from their definition over X-Relations. For each instant, their instantaneous X-Relation is defined as the result of the one-shot operator over the instantaneous X-Relation(s) of the operand(s).

For example, the selection operator over a finite XD-Relation is defined in Table 3.7. Its definition is similar to the definition of the selection operator over X-Relations. The similarity is highlighted in the "Tuples (bis)" line showing the relation between the two definitions.

All operators over X-Relations are redefined over finite XD-Relation in the same way, using with their respective definitions over X-Relations, except the binding operator that is tackled in Subsection 3.2.2. In particular, the resulting XD-Relation schema R is the resulting X-Relation schema R of the one-shot operator, associated with the predicate $infinite(R) = false$ as the XD-Relation is finite. We summarize in Table 3.8 those definitions at the tuple level using the "Tuples (bis)" style.

The service discovery operator is also redefined in a similar way. However, as the

Table 3.7: Definition of the selection operator over a finite XD-Relation

	Selection
Input	r a XD-Relation over R, $infinite(R) = false$, considering $\tau_{start_r} = \tau_{start_q}$ F a selection formula over $realSchema(R)$
Syntax	$s = \sigma_F(r)$
Output	s a XD-Relation over R, with $\tau_{start_s} = \tau_{start_q}$
Tuples	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = \{t \mid t \in r^*(\tau_i) \wedge t \models F\}$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = \{t \mid t \in r^*(\tau_i) \wedge t \models F\} - \{t \mid t \in r^*(\tau_{i-1}) \wedge t \models F\}$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = \{t \mid t \in r^*(\tau_{i-1}) \wedge t \models F\} - \{t \mid t \in r^*(\tau_i) \wedge t \models F\}$
Tuples (bis)	$\forall \tau_i \geq \tau_{start_q}, s^*(\tau_i) = \sigma_F(r^*(\tau_i))$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = \sigma_F(r^*(\tau_i)) - \sigma_F(r^*(\tau_{i-1}))$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = \sigma_F(r^*(\tau_{i-1})) - \sigma_F(r^*(\tau_i))$
Initial Conditions	$s^+(\tau_{start_s}) = s^*(\tau_{start_s})$ $s^-(\tau_{start_s}) = \emptyset$

service discovery one-shot query operator takes no X-Relation operand, we fully redefine it in Table 3.9. We can remark that the set of services that implement the prototypes and provide the attributes from the two given service interfaces does not evolve with time. This set is denoted $\Omega_{\phi,\phi'}$ in Table 3.9. It is only the availability of services that may change, indicated by the function $available(\omega, \tau_i)$.

3.2.2 Binding operator

For continuous queries, the generic binding operator represents two similar operators: the invocation and subscription binding operators. Both operators are defined over a finite XD-Relation and build respectively a finite XD-Relation and an infinite XD-Relation. Given a binding pattern, the resulting XD-Relation has the same X-Relation schema as with the one-shot query operator. At the data level, both operators use the prototype data function $data_\psi$, but not in the same way.

Given an invocation binding pattern, the invocation binding operator invokes the associated prototype only for newly inserted tuples in the operand XD-Relation, and not for every tuple at each time instant. For example, if a tuple t_1 is inserted at τ_{i_0}, the binding pattern is only invoked at τ_{i_0} for this tuple. Those invocation results are reused at instant $\tau_i > \tau_{i_0}$ as long as tuple t_1 is present in the operand instantaneous X-Relation, i.e. until tuple t_1 is deleted from the operand XD-Relation.

Given a subscription binding pattern, the subscription binding operator subscribes to the associated prototype for every newly inserted tuples in the operand XD-Relation, and unsubscribes from it for every deleted tuples. In other words, at each time instant, this operator is "listening" to subscriptions for each tuple from its operand instantaneous X-Relation. For example, if a tuple t_2 is inserted at τ_{i_0} in the operand XD-Relation,

Table 3.8: Summary of the extension of one-shot query operators over X-Relations to continuous query operators over finite XD-Relations, with s, r, r_1 and r_2 being finite XD-Relations

	Union
Syntax	$s = r_1 \cup r_2$
Tuples (bis)	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = r_1^*(\tau_i) \cup r_2^*(\tau_i)$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = (r_1^*(\tau_i) \cup r_2^*(\tau_i)) - (r_1^*(\tau_{i-1}) \cup r_2^*(\tau_{i-1}))$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = (r_1^*(\tau_{i-1}) \cup r_2^*(\tau_{i-1})) - (r_1^*(\tau_i) \cup r_2^*(\tau_i))$

	Intersection
Syntax	$s = r_1 \cap r_2$
Tuples (bis)	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = r_1^*(\tau_i) \cap r_2^*(\tau_i)$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = (r_1^*(\tau_i) \cap r_2^*(\tau_i)) - (r_1^*(\tau_{i-1}) \cap r_2^*(\tau_{i-1}))$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = (r_1^*(\tau_{i-1}) \cap r_2^*(\tau_{i-1})) - (r_1^*(\tau_i) \cap r_2^*(\tau_i))$

	Difference
Syntax	$s = r_1 - r_2$
Tuples (bis)	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = r_1^*(\tau_i) - r_2^*(\tau_i)$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = (r_1^*(\tau_i) - r_2^*(\tau_i)) - (r_1^*(\tau_{i-1}) - r_2^*(\tau_{i-1}))$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = (r_1^*(\tau_{i-1}) - r_2^*(\tau_{i-1})) - (r_1^*(\tau_i) - r_2^*(\tau_i))$

	Projection
Syntax	$s = \pi_Y(r)$
Tuples (bis)	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = \pi_Y(r^*(\tau_i))$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = \pi_Y(r^*(\tau_i)) - \pi_Y(r^*(\tau_{i-1}))$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = \pi_Y(r^*(\tau_{i-1})) - \pi_Y(r^*(\tau_i))$

	Selection
Syntax	$s = \sigma_F(r)$
Tuples (bis)	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = \sigma_F(r^*(\tau_i))$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = \sigma_F(r^*(\tau_i)) - \sigma_F(r^*(\tau_{i-1}))$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = \sigma_F(r^*(\tau_{i-1})) - \sigma_F(r^*(\tau_i))$

	Renaming
Syntax	$s = \rho_{A \to B}(r)$
Tuples (bis)	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = \rho_{A \to B}(r^*(\tau_i))$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = \rho_{A \to B}(r^*(\tau_i)) - \rho_{A \to B}(r^*(\tau_{i-1}))$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = \rho_{A \to B}(r^*(\tau_{i-1})) - \rho_{A \to B}(r^*(\tau_i))$

	Natural Join
Syntax	$s = r_1 \bowtie r_2$
Tuples (bis)	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = r_1^*(\tau_i) \bowtie r_2^*(\tau_i)$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = (r_1^*(\tau_i) \bowtie r_2^*(\tau_i)) - (r_1^*(\tau_{i-1}) \bowtie r_2^*(\tau_{i-1}))$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = (r_1^*(\tau_{i-1}) \bowtie r_2^*(\tau_{i-1})) - (r_1^*(\tau_i) \bowtie r_2^*(\tau_i))$

	Assignment
Syntax	$s = \alpha_{A \equiv [a\|B]}(r)$
Tuples (bis)	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = \alpha_{A \equiv [a\|B]}(r^*(\tau_i))$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = \alpha_{A \equiv [a\|B]}(r^*(\tau_i)) - \alpha_{A \equiv [a\|B]}(r^*(\tau_{i-1}))$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = \alpha_{A \equiv [a\|B]}(r^*(\tau_{i-1})) - \alpha_{A \equiv [a\|B]}(r^*(\tau_i))$

Table 3.9: Definition of the service discovery continuous query operator

	Service Discovery (with explicit/implicit service interfaces ϕ and ϕ'
Input	$A \in \mathcal{A}$ $\phi = \langle attributes_\phi, prototypes_\phi \rangle \in \Phi$ $\phi' = \langle attributes_{\phi'}, prototypes_{\phi'} \rangle \in \Phi$
Syntax	$s = \xi_{A,\phi,\phi'}()$
Output	s a XD-Relation over S, with: - $\tau_{start_s} = \tau_{start_q}$ - $schema(S) = \{A\} \cup attributes_\phi \cup$ $\bigcup_{\psi \in prototypes_\phi}(schema(Input_\psi) \cup schema(Output_\psi))$ - $realSchema(S) = \{A\} \cup attributes_\phi$ - $virtualSchema(S) = \bigcup_{\psi \in prototypes_\phi}(schema(Input_\psi) \cup schema(Output_\psi)) -$ $(\{A\} \cup attributes_\phi)$ - $BP(S) = \{bp = \langle prototype_{bp}, service_{bp} \rangle \mid service_{bp} = A \wedge$ $prototype_{bp} \in prototypes_\phi \wedge$ $schema(Output_{prototype_{bp}}) \cap (\{A\} \cup attributes_\phi) = \emptyset\}$
Tuples	Let $\Omega_{\phi,\phi'} = \{\omega \in \Omega \mid (prototypes_\phi \cup prototypes_{\phi'}) \subseteq prototypes(\omega) \wedge$ $(attributes_\phi \cup attributes_{\phi'}) \subseteq attributes(\omega)\}$ $\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) =$ $\{t \mid \exists \omega \in \Omega_{\phi,\phi'}, t[A] = id(\omega) \wedge available(\omega, \tau_i) \wedge$ $(\forall B \in attributes_\phi, t[B] = property_\omega(B))\}$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) =$ $\{t \mid \exists \omega \in \Omega_{\phi,\phi'}, t[A] = id(\omega) \wedge available(\omega, \tau_i) \wedge$ $(\forall B \in attributes_\phi, t[B] = property_\omega(B))\}$ - $\{t \mid \exists \omega \in \Omega_{\phi,\phi'}, t[A] = id(\omega) \wedge available(\omega, \tau_{i-1}) \wedge$ $(\forall B \in attributes_\phi, t[B] = property_\omega(B))\}$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) =$ $\{t \mid \exists \omega \in \Omega_{\phi,\phi'}, t[A] = id(\omega) \wedge available(\omega, \tau_{i-1}) \wedge$ $(\forall B \in attributes_\phi, t[B] = property_\omega(B))\}$ - $\{t \mid \exists \omega \in \Omega_{\phi,\phi'}, t[A] = id(\omega) \wedge available(\omega, \tau_i) \wedge$ $(\forall B \in attributes_\phi, t[B] = property_\omega(B))\}$
Initial Conditions	$s^+(\tau_{start_s}) = s^*(\tau_{start_s})$ $s^-(\tau_{start_s}) = \emptyset$

data from the subscription binding pattern are retrieved at each instant $\tau_i \geq \tau_{i_0}$ for this tuple, as long as tuple t_2 is present in the operand instantaneous X-Relation, i.e. until tuple t_2 is deleted from the operand XD-Relation.

The invocation binding operator and the subscription binding operator are both denoted by the generic binding operator β_{bp}. Its type depends on the type of the binding pattern, i.e. if it is an invocation or subscription binding pattern. Their definitions are presented in Table 3.10 and Table 3.11. Initial conditions of the operand XD-Relation, i.e. its start instant is considered to be the query start instant τ_{start_q}, are important for the invocation binding operator, as it relies on the sets of inserted/deleted tuples to select input tuples that need a binding pattern invocation. Furthermore, the definition of the subscription binding operator at the data level relies mainly on the definition of the resulting inserted tuple set, $s^+(\tau_i)$, as the deleted tuple set is always empty (the resulting XD-Relation being infinite) and the instantaneous tuple set is a simple union of all inserted tuple sets (it is an infinite set).

3.2.3 Stream operators

Stream operators are operators that handle infinite XD-Relations. We define two stream operators: the *window* operator and the *streaming* operator. Those two operators do not modify the XD-Relation schema, apart from its finite/infinite status: they transparently handle virtual attributes and binding patterns.

The window operator builds a finite XD-Relation from an infinite XD-Relation by building a union of tuples inserted in the operand between an instant τ_i included and an instant τ_{i-size} excluded, where *size* is the size of the window, i.e. a number of instants. The definition is presented in Table 3.12 (a).

The streaming operator builds an infinite XD-Relation from a finite XD-Relation by inserting tuples that are inserted/deleted/present in the operand for each instant, depending on the event type *event* of the operator: *insertion*, *deletion* or *heartbeat*. The definition is presented in Table 3.12 (b).

At the data level, both definitions rely heavily on the inserted/deleted tuples sets of the operand XD-Relation. Initial conditions are also important for the operand XD-Relation and for the resulting XD-Relation. It is to be noted that the definition of the streaming operator is based on the definition of its inserted tuple set, in a similar way to the subscription binding operator, as the resulting XD-Relation is infinite for those two operators.

The streaming operator could be extended to handle a more complex definition of its event type. For example, a complex event algebra like [DGH+06] could be integrated into the operator definition. It would increase the expressiveness of the algebra when data streams represent event streams. This extension is however not tackled in this thesis.

Table 3.10: Definition of Invocation Binding operator over a finite XD-Relation

	Invocation Binding
Input	r a XD-Relation over R, with $infinite(R) = false$ $bp \in BP(R)$, with $streaming(bp) = false$ $schema(Input_{prototype_{bp}}) \subset realSchema(R)$
Syntax	$s = \beta_{bp}(r)$
Output	s a XD-Relation over S, with: - $\tau_{start_s} = \tau_{start_q}$ - $infinite(S) = false$ - $schema(S) = schema(R)$ - $realSchema(S) = realSchema(R) \cup schema(Output_{prototype_{bp}})$ - $virtualSchema(S) = virtualSchema(R) - schema(Output_{prototype_{bp}})$ - $BP(S) = \{bp' \mid bp' \in BP(R) \wedge$ $schema(Output_{prototype_{bp'}}) \subseteq (virtualSchema(R) - schema(Output_{prototype_{bp}}))\}$
Tuples	$\forall \tau_i > \tau_{start_s}, s^*(\tau_i) = \{t \mid \exists u \in r^*(\tau_i),$ $t[realSchema(S) - schema(Output_{prototype_{bp}})] = u[realSchema(R)] \wedge$ $((u \in (r^*(\tau_{i-1}) - r^-(\tau_i)) \wedge t \in s^*(\tau_{i-1})) \vee (u \notin (r^*(\tau_{i-1}) - r^-(\tau_i)) \wedge$ $t[schema(Output_{prototype_{bp}})] \in$ $data_{prototype_{bp}}(u[service_{bp}], u[schema(Input_{prototype_{bp}})], \tau_i)))\}$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = s^*(\tau_i) - s^*(\tau_{i-1})$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = s^*(\tau_{i-1}) - s^*(\tau_i)$
Initial Conditions	$s^*(\tau_{start_s}) = \{t \mid \exists u \in r^*(\tau_i),$ $t[realSchema(S) - schema(Output_{prototype_{bp}})] = u[realSchema(R)] \wedge$ $t[schema(Output_{prototype_{bp}})] \in$ $data_{prototype_{bp}}(u[service_{bp}], u[schema(Input_{prototype_{bp}})], \tau_i)\}$ $s^+(\tau_{start_s}) = s^*(\tau_{start_s})$ $s^-(\tau_{start_s}) = \emptyset$

Table 3.11: Definition of Subscription Binding operator over a finite XD-Relation

	Subscription Binding
Input	r a XD-Relation over R, with $infinite(R) = false$ $bp \in BP(R)$, with $streaming(bp) = true$ $schema(Input_{prototype_{bp}}) \subset realSchema(R)$
Syntax	$s = \beta_{bp}(r)$
Output	s a XD-Relation over S, with: - $\tau_{start_s} = \tau_{start_q}$ - $infinite(S) = true$ - $schema(S) = schema(R)$ - $realSchema(S) = realSchema(R) \cup schema(Output_{prototype_{bp}})$ - $virtualSchema(S) = virtualSchema(R) - schema(Output_{prototype_{bp}})$ - $BP(S) = \{bp' \mid bp' \in BP(R) \wedge$ $schema(Output_{prototype_{bp'}}) \subseteq (virtualSchema(R) - schema(Output_{prototype_{bp}}))\}$
Tuples	$\forall \tau_i \geq \tau_{start_s}, s^-(\tau_i) = \emptyset$ $\forall \tau_i \geq \tau_{start_s}, s^+(\tau_i) = \{t \mid \exists u \in r^*(\tau_i),$ $t[realSchema(S) - schema(Output_{prototype_{bp}})] = u[realSchema(R)] \wedge$ $t[schema(Output_{prototype_{bp}})] \in$ $data_{prototype_{bp}}(u[service_{bp}], u[schema(Input_{prototype_{bp}})], \tau_i)\}$ $\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = \bigcup_{\tau_j \in \mathcal{T}, \tau_{start_s} \leq \tau_j \leq \tau_i}(s^+(\tau_j))$, with $s^*(\tau_i)$ being an infinite set
Initial Conditions	$s^*(\tau_{start_s}) = s^+(\tau_{start_s})$ $s^-(\tau_{start_s}) = \emptyset$

Table 3.12: Definition of Window and Streaming operators over XD-Relations

(a)	**Window**
Input	r a XD-Relation over R, with $infinite(R) = true$ $size$ a window size, i.e. a number of instants
Syntax	$s = \mathcal{W}_{[size]}(r)$
Output	s a XD-Relation over S, with: - $\tau_{start_s} = \tau_{start_q}$ - $infinite(S) = false$ - $schema(S) = schema(R)$ - $realSchema(S) = realSchema(R)$ - $virtualSchema(S) = virtualSchema(R)$ - $BP(S) = BP(R)$
Tuples	$\forall \tau_i \geq \tau_{start_s}, s^*(\tau_i) = \bigcup_{\tau_{start_s} \leq \tau_j \leq \tau_i \wedge \tau_j > \tau_{i-size}} (r^+(\tau_j))$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = r^+(\tau_i)$ $\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = \begin{cases} r^+(\tau_{i-size}) & \text{if } \tau_{i-size} \geq \tau_{start_s} \\ \emptyset & \text{otherwise} \end{cases}$
Initial Conditions	$s^+(\tau_{start_s}) = s^*(\tau_{start_s})$ $s^-(\tau_{start_s}) = \emptyset$

(b)	**Streaming**
Input	r a XD-Relation over R, with $infinite(R) = false$ $event \in \{insertion, deletion, heartbeat\}$ an event type
Syntax	$s = \mathcal{S}_{[event]}(r)$
Output	s a XD-Relation over S, with: - $\tau_{start_s} = \tau_{start_q}$ - $infinite(S) = true$ - $schema(S) = schema(R)$ - $realSchema(S) = realSchema(R)$ - $virtualSchema(S) = virtualSchema(R)$ - $BP(S) = BP(R)$
Tuples	$\forall \tau_i > \tau_{start_s}, s^-(\tau_i) = \emptyset$ as $infinite(S) = true$ $\forall \tau_i > \tau_{start_s}, s^+(\tau_i) = \begin{cases} r^+(\tau_i) & \text{if } event = insertion \\ r^-(\tau_i) & \text{if } event = deletion \\ r^*(\tau_i) & \text{if } event = heartbeat \end{cases}$ $\forall \tau_i > \tau_{start_s}, s^*(\tau_i) = \bigcup_{\tau_j \in \mathcal{T}, \tau_{start_s} \leq \tau_j \leq \tau_i} (s^+(\tau_j))$
Initial Conditions	$s^+(\tau_{start_s}) = \begin{cases} r^*(\tau_i) & \text{if } event = insertion \\ \emptyset & \text{if } event = deletion \\ r^*(\tau_i) & \text{if } event = heartbeat \end{cases}$ $s^-(\tau_{start_s}) = \emptyset$ $s^*(\tau_{start_s}) = s^+(\tau_{start_s})$

3.2.4 Continuous queries over a relational pervasive environment

The notion of continuous query over a relational pervasive environment can now be defined as a composition of continuous query operators over a set of XD-Relations. A continuous query q is associated with its start instant, i.e. the instant when the query is started, denoted τ_{start_q}. The operand XD-Relations are also considered to be associated with this start instant, i.e. the initial conditions on their instantaneous/inserted/deleted tuple sets apply at this instant. For example, a query q using an operand XD-Relation r that is associated with a start instant $\tau_{start_r} \leq \tau_{start_q}$, considers r with the following initial conditions:

- $r^*(\tau_{start_q})$ unmodified,
- $r^+(\tau_{start_q}) = r^*(\tau_{start_q})$,
- $r^-(\tau_{start_q}) = \emptyset$.

This restriction applies only at τ_{start_q} for all operand XD-Relations, i.e. for finite XD-Relations and for infinite XD-Relations. For the following instants $\tau_i > \tau_{start_q}$, operand XD-Relations are not considered modified, i.e. their instantaneous/inserted/deleted tuple sets are unmodified. When several continuous queries share the same operand XD-Relation, each query q_i separately considers this XD-Relation with its own start instant $\tau_{start_{q_i}}$.

Definition 3.2 (Continuous Query) *Let p be a relational pervasive environment. A continuous query q over p is a well-formed expression composed of a finite number of Serena continuous query algebra operators. A continuous query is associated with its start instant, i.e. the instant when the query is started, denoted $\tau_{start_q} \in \mathcal{T}$. Its result is a XD-Relation associated with the same start instant τ_{start_q}. All operand XD-Relations are considered to be associated with the same start instant τ_{start_q}, i.e. initial conditions are applied at τ_{start_q}.*

Example 3.3 (Continuous Query) *We consider a relational pervasive environment containing three XD-Relations: contacts and cameras, two finite XD-Relations described in Example 2.9 (page 25) and used for one-shot queries in Example 3.2 (page 49), and temperatures, an infinite XD-relation representing a temperature stream also described in Example 2.9.*

We express the following behaviors by continuous queries Q_3 and Q_4 presented in Table 3.13:

Q_3 *when a temperature exceeds 35.5°C in the corridor, send a message "hot!" to the contacts,*

Q_4 *when a temperature goes down below 12.0°C, take a photo of the area.*

In both queries, the window operator with a period of 1 ($\mathcal{W}_{[1]}$) indicates that we are interested in data tuples from the temperatures stream only at the instant when they are inserted. They are not kept in the intermediary XD-Relation for the following instants.

Query Q_3 filters temperature notifications from temperatures with two selection operators to test the area and the temperature value. It then combines through a natural join those filtered notifications with every contacts from contacts (the natural join is here similar to a Cartesian product as there is no common attribute). A value is then assigned to the virtual attribute text, and the binding pattern associated with the prototype sendMessage is invoked.

$$
\begin{aligned}
Q_3 = \quad & \beta_{\langle sendMessage, messenger\rangle}(\alpha_{text="Hot!"}(contacts \bowtie \sigma_{temperature>35.5}(\\
& \quad \sigma_{area="corridor"}(\mathcal{W}_{[1]}(temperatures)) \\
&))
\end{aligned}
$$

Query Q_4 filters temperature notifications with a selection that tests the temperature value. It then joins those filtered notifications with cameras from cameras on the attribute area (that is real in both operands). Two successive binding patterns are invoked: the prototype checkPhoto realizes attributes quality and delay, and the prototype takePhoto realizes the attribute photo. A final streaming operator builds an infinite XD-Relation from the tuples inserted in its operand, i.e. a stream of tuples that are produced by the last binding operator.

$$
\begin{aligned}
Q_4 = \quad & \mathcal{S}_{[insertion]}(\beta_{\langle takePhoto, camera\rangle}(\beta_{\langle checkPhoto, camera\rangle}(\\
& \quad cameras \bowtie \sigma_{temperature<12.0}(\mathcal{W}_{[1]}(temperatures)) \\
&))
\end{aligned}
$$

Both queries have a similar behavior, but do not have the same type of resulting XD-Relation. The result of Q_3 is a finite XD-Relation, as its last operator is the invocation binding operator. For each instant, the resulting XD-Relation contains only the set of alert messages sent at this instant, because of the 1-instant window. The result of Q_4 is an infinite XD-Relation, as its last operator is the streaming operator $\mathcal{S}_{[insertion]}$. This query builds a continuous stream of photos. Note that this difference does not impact on the invocations of binding patterns, but only on the resulting XD-Relation. For example, we could add a streaming operator $\mathcal{S}_{[insertion]}$ to Q_3 without modifying its behavior.

We also consider a continuous query Q_5 that combines a subscription binding operator with a service discovery operator. The service discovery operator selects all temperature sensors, i.e. services providing a property area and implementing a subscription prototype temperatureNotifications providing one output attribute temperature. This query could replace the infinite XD-Relation temperature in Q_4. Note that in order to use a projection operator, we need to add a 1-instant window operator before and an insertion streaming operator after, as the projection operator is defined only over finite XD-Relations. It would also be the case for other operators over finite XD-Relations.

Table 3.13: Examples of continuous queries over XD-Relations expressed in the Serena algebra

Q_3	$\beta_{\langle sendMessage,messenger\rangle}(\alpha_{text\equiv\text{"Hot!"}}(contacts \bowtie \sigma_{temperature>35.5}($ $\sigma_{area=\text{"corridor"}}(\mathcal{W}_{[1]}(temperatures))$ $)))$
Q_4	$\mathcal{S}_{[insertion]}(\beta_{\langle takePhoto,camera\rangle}(\beta_{\langle checkPhoto,camera\rangle}($ $cameras \bowtie \sigma_{temperature<12.0}(\mathcal{W}_{[1]}(temperatures))$ $)))$
Q_5	$\mathcal{S}_{[insertion]}(\pi_{area,temperature}(\mathcal{W}_{[1]}($ $\beta_{\langle temperatureNotifications,sensor\rangle}(\xi_{sensor,\langle\{area\},\{temperatureNotifications\}\rangle,\langle\emptyset,\emptyset\rangle}()))$ $)))$

$$Q_5 = \mathcal{S}_{[insertion]}(\pi_{area,temperature}(\mathcal{W}_{[1]}(\beta_{\langle temperatureNotifications,sensor\rangle}(\xi_{sensor,\langle\{area\},\{temperatureNotifications\}\rangle,\langle\emptyset,\emptyset\rangle}())))))$$

We consider their respective start instants $\tau_{start_{Q_3}}$, $\tau_{start_{Q_4}}$ and $\tau_{start_{Q_5}}$, e.g., $\tau_{start_{Q_3}} = \tau_{start_{Q_4}} = \tau_{start_{Q_5}} = \tau_{now} = \tau_{15}$. The Serena algebra expression for those queries are summarized in Table 3.13.

Continuous queries over a relational pervasive environment enable the declarative expression of interactions between data, streams and services. In particular, applications that monitor data streams and use service methods to handle some events (e.g., sending notification messages or taking photos) can easily be developed through simple expressions with few operators. Furthermore, the service discovery operator enables to dynamically integrate all available resources (e.g., integrating new temperature sensors) while the continuous query is running. We summarize notations for XD-Relations and continuous queries in Table 3.14.

Table 3.14: Overview of Notations for XD-Relations and Continuous Queries

	STRUCTURE	LANGUAGE
DATA + SERVICES + TIME + STREAMS	• Discrete Time Domain $\tau_i \in \mathcal{T}$ • Interaction with Services – $available(\omega, \tau_i) \in \mathcal{B}$ – $data_\psi(id(\omega), input, \tau_i) \in \mathcal{D}^{\lvert schema(Output_\psi)\rvert}$ • XD-Relation Schema R – X-Relation Schema R – $infinite(R) \in \mathcal{B}$ • XD-Relation r over R – $\tau_{start_r} \in \mathcal{T}$ – $r^*(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R)\rvert}$ – $r^+(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R)\rvert}$ – $r^-(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R)\rvert}$	• Operators over finite XD-Relation(s) – Selection $\sigma_F(r)$ – Projection $\pi_Y(r)$ – Renaming $\rho_{A\to B}(r)$ – Natural Join $r_1 \bowtie r_2$ – Assignment $\alpha_{A\equiv B}(r)$ / $\alpha_{A\equiv c}(r)$ – Binding $\beta_{\langle\psi,S\rangle}(r)$ – Streaming $\mathcal{S}_{[event]}(r)$ • Operator over infinite XD-Relation(s) – Window $\mathcal{W}_{[size]}(r)$ • Operator for Service Discovery – Service Discovery $\xi_{S,\phi,\phi'}()$

3.3 Query equivalence

We now define the crucial notion of equivalence between two queries over a relational pervasive environment, for one-shot queries and continuous queries. The notion of active/passive prototypes used by binding patterns plays a key role in this definition. Query equivalence also leads to the definition of rewriting rules for Serena algebra expressions.

In order to define query equivalence, three issues need to be addressed in the general context of pervasive environments: time-dependence, service determinism and service impacts on the environment. As a pervasive system is a dynamic system, the same service invoked with the same input, but at two different moments may lead to two different results (e.g., a service that takes a photo). Services may be undeterministic, i.e. the invocation order may have an impact on invocation results. Service invocations may also have an impact on the environment that can not be "cancelled", e.g., for a service sending a message by SMS: once received by the person, the message can not be "cancelled".

The first two issues, i.e. time-dependence and service determinism, are handled as follows: we assume that all services are deterministic at a given discrete time instant τ_i, through the definition of the prototype data function. The invocation order has then no impact on invocation results for all service invocations occurring at τ_i. For example, a service that returns the number of times it has been invoked should still return the same value for all invocations occurring at τ_i. These assumptions are classical in the context of distributed service management: even in a "Web environment", [AMZ08] considers deterministic services such that "the answers at time t and $t + \epsilon$, for a small ϵ, [are] equally acceptable".

The third issue, i.e. service impact on the environment, needs to be specifically addressed in our context of relational pervasive environments and is, to the best of our knowledge, completely new. In the following subsections, we first define the notion of action set of a query. We then formally define query equivalence in our context, and finally propose some rewriting rules for Serena algebra expressions.

3.3.1 Action sets

In order to reflect the impact of a query on the environment, we define the notion of *action set* induced by a query against a relational pervasive environment as the set of invocations and subscriptions of *active binding patterns* triggered by this query. For instance, consider query Q_1 in Table 3.5 (page 50): we want to capture the set of messages sent by the execution of this one-shot query. In particular, we want to reflect whether the message "Bonjour!" is sent or not to Carla, depending on the position of the selection operator within Q_1.

Definition 3.3 (Action Set) *An* action *is a 3-tuple $\langle \psi, s, t \rangle$ with $\psi \in \Psi$ an active prototype, $s \in \mathcal{D}$ a service reference and t an input data tuple for ψ, i.e. a tuple over $Input_\psi$.*

The action set *of a query q against a relational pervasive environment p at instant τ_i, denoted by $Actions_q(p, \tau_i)$ (or simply $Actions_q(\tau_i)$ where p is clear from context), is the set of actions induced by binding operators using active binding patterns, i.e. binding patterns with active prototypes.*

A one-shot query q, associated with the start instant τ_{start_q}, induces a single action set at instant τ_{start_q}, i.e. at the instant it is launched:

$$Actions_q(p, \tau_{start_q}) = \{\langle \psi, s, t \rangle \mid \exists \beta_{bp}(q') \in q, active(bp) \wedge \psi = prototype_{bp} \wedge u \in q'(p) \wedge s = u[service_{bp}] \wedge t = u[schema(Input_{prototype_{bp}})]\},$$

where $\beta_{bp}(q') \in q$ denotes the occurrence of the one-shot binding operator β_{bp} in q with subquery q' as its operand, and $q'(p)$ denotes the resulting X-Relation of subquery q' against p.

A continuous query q, associated with the start instant τ_{start_q}, induces an action set for each instant from τ_{start_q}, i.e. from the instant it is started:

$$Actions_q(p, \tau_{start_q}) = \{\langle \psi, s, t \rangle \mid \exists \beta_{bp}(q') \in q, active(bp) \wedge \psi = prototype_{bp} \wedge u \in (q'(p))^*(\tau_{start_q}) \wedge s = u[service_{bp}] \wedge t = u[schema(Input_{prototype_{bp}})]\},$$

$$\forall \tau_i > \tau_{start_q},$$
$$Actions_q(p, \tau_i) = \{\langle \psi, s, t \rangle \mid \beta_{bp}(q'(p)) \in q \wedge active(bp) \wedge \psi = prototype_{bp} \wedge u \in (q'(p))^*(\tau_i) \wedge s = u[service_{bp}] \wedge t = u[schema(Input_{prototype_{bp}})] \wedge (streaming(bp) \vee u \notin ((q'(p))^*(\tau_{i-1}) - (q'(p))^-(\tau_i))\},$$

where $\beta_{bp}(q') \in q$ denotes the occurrence of the invocation or subscription binding operator β_{bp} in q with subquery q' as its operand, and $q'(p)$ denotes the resulting XD-Relation of subquery q' against p.

Table 3.15: One-shot queries expressed in the Serena algebra

Q_1	$\beta_{\langle sendMessage,messenger\rangle}(\alpha_{text\equiv\text{"Bonjour!"}}(\sigma_{name\neq\text{"Carla"}}(contacts)))$
Q_2	$\pi_{photo}(\beta_{\langle takePhoto,camera\rangle}(\sigma_{quality>5}(\beta_{\langle checkPhoto,camera\rangle}(\sigma_{area=\text{"office"}}($ $cameras$ $)))))$
Q'_1	$\sigma_{name\neq\text{"Carla"}}(\beta_{\langle sendMessage,messenger\rangle}(\alpha_{text\equiv\text{"Bonjour!"}}(contacts)))$
Q'_2	$\pi_{photo}(\sigma_{quality>5}(\beta_{\langle takePhoto,camera\rangle}(\beta_{\langle checkPhoto,camera\rangle}(\sigma_{area=\text{"office"}}($ $cameras$ $)))))$

Example 3.4 (Action Set) *We consider the one-shot queries Q_1 and Q_2 described in Example 3.2 (page 49), and two additional queries Q'_1 and Q'_2, respectively similar to Q_1 and Q_2. Those four queries are presented in Table 3.15.*

Considering the content of the XD-Relation contacts described in Example 2.10 at their start instant, e.g., $\tau_{start_{Q_1}} = \tau_{start_{Q'_1}} = \tau_{now}$, the action sets for the two similar queries Q_1 and Q'_1 contains the actions induced by the invocation binding operator $\beta_{\langle sendMessage,messenger\rangle}$ using the active prototype sendMessage.

$$
\begin{aligned}
&Actions_{Q_1}(\tau_{now}) = \{\\
&\quad\langle sendMessage, email, \langle nicolas@elysee.fr, Bonjour!\rangle\rangle,\\
&\quad\langle sendMessage, jabber, \langle francois@im.gouv.fr, Bonjour!\rangle\rangle\},\\
&Actions_{Q'_1}(\tau_{now}) = \{\\
&\quad\langle sendMessage, email, \langle nicolas@elysee.fr, Bonjour!\rangle\rangle,\\
&\quad\langle sendMessage, email, \langle carla@elysee.fr, Bonjour!\rangle\rangle,\\
&\quad\langle sendMessage, jabber, \langle francois@im.gouv.fr, Bonjour!\rangle\rangle\}.
\end{aligned}
$$

For Q_2 and Q'_2, their action sets at their start instant are empty as the invocation binding operators $\beta_{\langle takePhoto,camera\rangle}$ and $\beta_{\langle checkPhoto,camera\rangle}$ use the passive prototypes takePhoto and checkPhoto. Considering $\tau_{start_{Q_2}} = \tau_{start_{Q'_2}} = \tau_{now}$, we have:

$$Actions_{Q_2}(\tau_{now}) = Actions_{Q'_2}(\tau_{now}) = \emptyset\,.$$

In summary, two queries have the same action set if they have the same impact on the environment through active binding pattern invocation/subscription. This notion is required to define the notion of query equivalence in the setting of pervasive environment.

3.3.2 Query equivalence

The previous notion of action set enables the description of the impact of a query defined over a relational pervasive environment schema and evaluated against a relational pervasive environment. Two one-shot queries associated with the same start instant τ_{start} are equivalent if their evaluations at τ_{start} lead to the same resulting X-Relation **and** the same action set at τ_{start}. Equivalent one-shot queries induce the same invocations of active binding patterns, although they may imply different invocations of passive binding patterns.

Two continuous queries associated with the same start instant τ_{start} are equivalent if their evaluations lead to the same resulting XD-Relation and the same action set for $\tau_i \geq \tau_{start}$. Equivalent continuous queries induce the same invocations of, and subscription to, active binding patterns, although they may imply different invocations of, or subscription to, passive binding patterns.

Definition 3.4 (Query Equivalence) *Let P be a relational pervasive environment schema.*

Two one-shot queries q_1 and q_2 over P, associated with the same start instant τ_{start}, are equivalent, *denoted by $q_1 \equiv q_2$,* iff *for any p over P, $q_1(p) = q_2(p)$ and $Actions_{q_1}(p, \tau_{start}) = Actions_{q_2}(p, \tau_{start})$.*

Two continuous queries q_1 and q_2 over P, associated with the same start instant τ_{start}, are equivalent, *denoted by $q_1 \equiv q_2$,* iff *for any p over P, $q_1(p) = q_2(p)$ and for $\tau_i \geq \tau_{start}$, $Actions_{q_1}(p, \tau_i) = Actions_{q_2}(p, \tau_i)$.*

Example 3.5 (Query Equivalence) *One-shot queries Q_1 and Q_1' from Table 3.15 are not equivalent because of their action sets (see Example 3.4), although their resulting X-Relation should be the same. One-shot queries Q_2 and Q_2' (also from Table 3.15) are equivalent, as their action sets are both empty.*

The choice of tagging binding patterns as active or passive is however up to the application developers, and impacts on query equivalence. Whereas *sendMessage* is surely to be defined as active, *takePhoto* may be considered passive, leading to the equivalence of Q_2' into Q_2, or active, leading to the non-equivalence between Q_2' and Q_2. This choice depends on the objectives of the application and on the impacts of executing the services.

Table 3.16: Rewriting rules with assignment and invocation operators

Operator	**Assignment of virtual attribute A (with real attribute B or constant c)**
Projection	$\pi_L(\alpha_{A\equiv B}(r)) \equiv \alpha_{A\equiv B}(\pi_L(r))$ if $A, B \in L$ $\pi_L(\alpha_{A\equiv c}(r)) \equiv \alpha_{A\equiv c}(\pi_L(r))$ if $A \in L$
Selection	$\sigma_F(\alpha_{A\equiv B}(r)) \equiv \alpha_{A\equiv B}(\sigma_F(r))$ if $A \notin F$ $\sigma_F(\alpha_{A\equiv c}(r)) \equiv \alpha_{A\equiv c}(\sigma_F(r))$ if $A \notin F$
Natural Join	$\alpha_{A\equiv B}(r_1 \bowtie r_2) \equiv \alpha_{A\equiv B}(r_1) \bowtie r_2$ if $A, B \in schema(R_1)$ and $A \notin realSchema(R_2)$ $\alpha_{A\equiv c}(r_1 \bowtie r_2) \equiv \alpha_{A\equiv c}(r_1) \bowtie r_2$ if $A \in schema(R_1)$ and $A \notin realSchema(R_2)$
Operator	**Invocation of *passive* binding pattern bp (i.e. where $prototype_{bp}$ is *passive*)**
Projection	$\pi_L(\beta_{bp}(r)) \equiv \beta_{bp}(\pi_L(r))$ if $not(active(bp))$ and $service_{bp} \in L$ and $schema(Input_{prototype_{bp}}) \subset L$ and $schema(Output_{prototype_{bp}}) \subset L$
Selection	$\sigma_F(\beta_{bp}(r)) \equiv \beta_{bp}(\sigma_F(r))$ if $not(active(bp))$ and $schema(Output_{prototype_{bp}}) \cap F = \emptyset$
Natural Join	$\beta_{bp}(r_1 \bowtie r_2) \equiv \beta_{bp}(r_1) \bowtie r_2$ if $not(active(bp))$ and $bp \in BP(R_1)$ and $schema(Input_{prototype_{bp}}) \subset realSchema(R_1)$ and $schema(Output_{prototype_{bp}}) \cap realSchema(R_2) = \emptyset$

3.3.3 Rewriting rules

Based on this query equivalence, *rewriting rules* can be applied to one-shot and continuous queries expressed in the Serena algebra. Without the new operators introduced so far, rewriting rules of the relational algebra are still valid to change the order of standard operators redefined in our context, e.g., pushing down or pulling up a selection around a natural join operator.

Realization operators can be reorganized, but as realization operators modify the real/virtual status of attributes, there are some restrictions at the metadata level for the rewriting rules. For example, with a selection operator, attributes realized by the realization operator should not be part of the selection formula in order to push down the selection operator. As equivalent queries should induce the same action sets, binding operators associated with active binding patterns can not be reorganized: invocations of, or subscriptions to, active binding patterns should remain identical for those queries at any instant for any relational pervasive environment as they impact on the action sets. Binding operators associated with passive binding patterns can however be reorganized. Rewriting rules for the assignment operator and invocation binding operator for passive binding patterns are presented in Table 3.16.

We demonstrate the proof of one rewriting rules for one-shot queries: $\sigma_F(\beta_{bp}(r)) \equiv \beta_{bp}(\sigma_F(r))$, with β_{bp} a one-shot binding operator associated with a passive invocation binding pattern.

Property 3.1 *Let p be a relational pervasive environment. Let $r \in p$ be a finite XD-Relation over R, $bp \in BP(R)$ an invocation binding pattern, F a selection formula over $realSchema(R)$,*

with β_{bp} and σ_F being one-shot operators:

$$\neg active(bp) \Rightarrow \sigma_F(\beta_{bp}(r)) \equiv \beta_{bp}(\sigma_F(r)).$$

Proof of Property 3.1 *We define two one-shot queries q_1 and q_2 over p associated with the same start instant τ_{start}. We denote the passive binding pattern by $bp = \langle \psi, S \rangle$.*

$$q_1 = \sigma_F(\beta_{\langle \psi,S \rangle}(r))$$

$$q_2 = \beta_{\langle \psi,S \rangle}(\sigma_F(r))$$

We first build the tuple set of q_1 from the operator definitions.

$$q_1 = \sigma_F(\beta_{\langle \psi,S \rangle}(r))$$

$\beta_{\langle \psi,S \rangle}(r^*(\tau_{start})) = \{t \mid \exists u \in r^*(\tau_{start}), t[realSchema(R)] = u[realSchema(R)] \wedge t[schema(Output_\psi)] \in data_\psi(u[S], u[schema(Input_\psi)], \tau_{start})\}$

$$q_1 = \{t \mid t \in \beta_{\langle \psi,S \rangle}(r^*(\tau_{start})) \wedge t \models F\}$$

We also build the tuple set of q_2 from the operator definitions.

$$q_2 = \beta_{\langle \psi,S \rangle}(\sigma_F(r))$$

$q_2 = \{t \mid \exists u \in \sigma_F(r^*(\tau_{start})), t[realSchema(R)] = u[realSchema(R)] \wedge t[schema(Output_\psi)] \in data_\psi(u[S], u[schema(Input_\psi)], \tau_{start})\}$
$q_2 = \{t \mid \exists u \in r^*(\tau_{start}), t[realSchema(R)] = u[realSchema(R)] \wedge t[schema(Output_\psi)] \in data_\psi(u[S], u[schema(Input_\psi)], \tau_{start}) \wedge u \models F\}$

As the selection formula F is defined over the real schema of R, we have the following implication:

$$t[realSchema(R)] = u[realSchema(R)] \Rightarrow (u \models F \Leftrightarrow t \models F)$$

Then the tuple set of q_2 can be rewritten:
$q_2 = \{t \mid \exists u \in r^*(\tau_{start}), t[realSchema(R)] = u[realSchema(R)] \wedge t[schema(Output_\psi)] \in data_\psi(u[S], u[schema(Input_\psi)], \tau_{start}) \wedge t \models F\}$
$q_2 = \{t \mid t \in \beta_{\langle \psi,S \rangle}(r^*(\tau_{start})) \wedge t \models F\}$

We have demonstrated that $q_1 = q_2$. As the binding operator is associated with a passive binding pattern, the induced action sets for both queries are empty.

$$\neg active(bp) \Rightarrow Actions_{q_1} = Actions_{q_2} = \emptyset$$

We have then demonstrated that:

$$\neg active(bp) \Rightarrow \sigma_F(\beta_{bp}(r)) \equiv \beta_{bp}(\sigma_F(r)).$$

Operators handling infinite XD-Relations, i.e. window operators, streaming operators and subscription binding operators, represent a particular case. As they modify the finite/infinite status of XD-Relations, they can not be freely reorganized with other operators that are all defined over finite XD-Relations. However, some rewriting rules can still be devised for those operators. For example, window and streaming operators can cancel each other in the case of 1-instant window operator combined with an heartbeat streaming operator, or a n-instant window combined with an insertion streaming operator. We aim at illustrating the "working" of those two specific operators through two related properties of equivalence.

Property 3.2 *Let p be a relational pervasive environment. Let $r \in p$ be a finite XD-Relation and $s \in p$ an infinite XD-Relation. We have:*

$$(1) \quad r \equiv \mathcal{W}_{[1]}(\mathcal{S}_{[heartbeat]}(r)) \,,$$
$$(2) \quad s \equiv \mathcal{S}_{[heartbeat]}(\mathcal{W}_{[1]}(s)) \equiv \mathcal{S}_{[insertion]}(\mathcal{W}_{[n]}(s)).$$

Proof of Property 3.2 *Those rules derive rather directly from the definition of the operators. We have the following relation for a 1-instant window operator:*

$$(\mathcal{W}_{[1]}(s))^*(\tau_i) = \bigcup_{\tau_j \in \mathcal{T}, \tau_{start_s} \leq \tau_j \leq \tau_i \wedge \tau_j > \tau_{i-1}} (s^+(\tau_j)) = s^+(\tau_i).$$

We can then derive that:

$$(\mathcal{W}_{[1]}(\mathcal{S}_{[heartbeat]}(r)))^*(\tau_i) = (\mathcal{S}_{[heartbeat]}(r))^+(\tau_i) = r^*(\tau_i) \,,$$
$$(\mathcal{S}_{[heartbeat]}(\mathcal{W}_{[1]}(s)))^+(\tau_i) = (\mathcal{W}_{[1]}(s))^*(\tau_i) = s^+(\tau_i) \,,$$
$$(\mathcal{S}_{[insertion]}(\mathcal{W}_{[n]}(s)))^+(\tau_i) = (\mathcal{W}_{[n]}(s))^+(\tau_i) = s^+(\tau_i).$$

The instantaneous tuple set for $\mathcal{W}_{[1]}(\mathcal{S}_{[heartbeat]}(r))$ is the same as the instantaneous tuple set of the finite XD-Relation r at each instant. The insertion tuple set for $\mathcal{S}_{[heartbeat]}(\mathcal{W}_{[1]}(s))$ or $\mathcal{S}_{[insertion]}(\mathcal{W}_{[n]}(s))$ is the same as the insertion tuple set of the infinite XD-Relation s at each instant. As no binding operator is involved, the action sets of those expressions are always empty. Those expressions are then respectively equivalent:

$$\mathcal{W}_{[1]}(\mathcal{S}_{[heartbeat]}(r)) \equiv r \,,$$
$$\mathcal{S}_{[heartbeat]}(\mathcal{W}_{[1]}(s)) \equiv \mathcal{S}_{[insertion]}(\mathcal{W}_{[n]}(s)) \equiv s.$$

□

Streaming operators and subscription binding operators should then be reorganized along with their corresponding window operator, as they build infinite XD-Relations that can only be handled by window operators. We can consider "composite" operators, i.e. streaming operators or subscription binding operators composed with a window operator, as operators over finite XD-Relations building finite XD-Relations. Those composite operators can be easily reorganized with other operators in additional rewriting rules.

3.4 Query optimization

In this section, we define a cost model dedicated to relational pervasive environments. Our cost model is kept rather simple. We consider a computation environment where access to local data storage, i.e. for tuples from X-Relations or XD-Relations, is relatively very fast in comparison to access to distributed functionalities, i.e. for interactions with services over the network. We then focus on network I/O costs due to interactions with distributed services more than on local computation resource consumption. We also consider that tuples can be randomly accessed with no penalty, that may not be the case with standard DBMS running over hard disk drives. Our cost model nevertheless enables to take into account the specificity of the relational pervasive environment, with dynamic data sources and distributed functionalities.

The cost of a query against a relational pervasive environment is estimated by four metrics representing service I/O costs and computation/memory consumption. For a continuous query, the cost is estimated for its start instant (the initial cost) as well as for a representative following instant (the running cost). For a one-shot query, the cost is estimated only for its start instant. We then define a comparison order between costs in order to formally define the query optimization goal. We finally devise query optimization techniques, based on query equivalence and rewriting rules defined in the previous section, in order to optimize Serena algebra expressions into equivalent, but hopefully more efficient, expressions.

3.4.1 Metrics of the cost model

In order to represent the execution cost of a query against a relational pervasive environment, we define a cost model based on four metrics: invocation/subscription cost, computation cost, tuple number and tuple size. Those metrics are measured, or estimated, for a given instant τ_i.

The *invocation/subscription cost* represents the cost of interacting with services, e.g., network bandwidth, network latency, service computation. In a query, the only operators that contribute to this cost are the two binding operators, i.e. the invocation binding operator and the subscription binding operator. For the sake of simplicity, we define it to be the number of invocations of invocation prototypes and subscriptions to subscription prototypes for instant τ_i.

The *computation cost* is a measure that reflects the computation cost of query operators, i.e. the computation resource consumption, depending on their complexity. It represents the cost of executing individual operators. For the sake of simplicity, we define it to be the number of tuples that need to be read, stored and/or computed by operators.

Two additional metrics reflect the resource consumption of intermediary X-Relations or XD-Relations in order to manage their tuples. The *tuple number* is the number of tuples read by query operators from their operand(s). It represents the cost of managing internal buffers of tuples. The *tuple size* is a measure combining the number of tuple and their weight, i.e. their size in bytes. We consider the mean weight of tuples over a given XD-Relation schema. It represents the memory cost of internal buffers of tuples.

The cost of a continuous query is defined in two parts: the initial cost for its start instant τ_{start} and the running cost for a representative following instant $\tau_i > \tau_{start}$. The initial cost reflects the initial conditions of the data sources, i.e. their content at the query start instant, whereas the running cost reflects the dynamicity of the data sources, implying the cost of continuously updating the query result. For a one-shot query, the cost is defined only by the initial cost for its start instant τ_{start}, as the query result is not continuously updated.

3.4.2 Estimating the cost of a query

For the purpose of optimization, we need to estimate the cost of a query against a relational pervasive environment before its execution, in order to compare equivalent queries and choose the least expensive. We assume that statistics on base XD-Relations are available, with which we can compute estimations for statistics on intermediary XD-Relations and finally for the query cost metrics. Those statistics are the mean cardinality, the mean dynamicity and the mean tuple weight. We detail those statistics in the next paragraph. The computation of up-to-date statistics in the setting of pervasive environments is however out of the scope of this thesis.

With each operand of a continuous query algebra expression, i.e. base XD-Relations and intermediary XD-Relations, we associate three tuple statistics: the mean cardinality of the XD-Relation, i.e. its mean number of tuples in its instantaneous tuple set; the mean dynamicity of the XD-Relation, i.e. its mean number of tuples in its inserted tuple set; and the mean tuple weight, i.e. the mean size in bytes of a tuple. For infinite XD-Relations, the mean cardinality is set to ∞, as the statistic has no meaning in this case since the instantaneous tuple set is an append-only infinite set. For a one-shot query, the dynamicity is not taken into account for operand X-Relations. We denote tuple statistics of a XD-Relation r as a 3-tuple $\mathcal{T}_r = \langle t_c, t_d, t_w \rangle \in \mathbb{R}^3$, with t_c the tuple cardinality, t_d the tuple dynamicity and t_w the tuple weight.

For each operator of a continuous query algebra expression, we compute its contribution to the cost metrics and the tuple statistics for its output XD-Relation (in a similar way to traditional techniques for databases [GMWU99]). They both depend on the operand tuple statistics as well as on operator specificity, e.g., the selectivity of a selection or a natural join, or the prototype stream rate for a subscription binding.

For a one-shot query, we compute tuple statistics for output X-Relations, considering instantaneous X-Relations from XD-Relations or output X-Relations of other operators.

Contributions of all operators are accumulated into a global cost for the query. We use a simple sum of those contributions for each metrics. A cost is represented by a 4-tuple $\mathcal{C} = \langle c_i, c_c, c_n, c_s \rangle \in \mathbb{R}^4$, with c_i the invocation/subscription cost, c_c the computation cost, c_n the tuple number, and c_s the tuple size. For a continuous query q, we denote $\mathcal{C}_q^*$ the initial cost for its start instant τ_{start_q} and $\mathcal{C}_q^+$ the running cost for a representative following instant $\tau_i > \tau_{start_q}$. For a one-shot query q, we only consider $\mathcal{C}_q^*$ the cost for its start instant τ_{start_q}.

Example 3.6 (Tuple Statistics and Query Cost) *We consider the XD-Relation sensors representing the set of available temperature sensors in the environment, along with their location. Its Serena DDL representation is given below.*

```
RELATION sensors (
  sensor       SERVICE,
  area         STRING,
  temperature  REAL     VIRTUAL
)
USING BINDING PATTERNS (
  getTemperature[sensor] ( ) : ( temperature ),
  temperatureNotifications[sensor] ( ) : ( temperature ) STREAMING
);
```

We further consider that this XD-Relation has a mean cardinality of 120 tuples, i.e. there are about 120 sensors in the environment; and that a new sensor appears (and an old one disappears) about every 40 instants: its mean dynamicity is 0.025 tuples per instants. The tuple weight is determined by the real schema of the XD-Relation: there are two real attributes, one of type SERVICE *and one of type* STRING*. We consider, for example, that the tuple weight is 4 bytes for the* SERVICE *attribute sensor plus a mean of 10 bytes for the* STRING *attribute area. Then, the tuple statistics for the XD-Relation sensors are:*

$$\mathcal{T}_{sensors} = \langle 120.0, 0.025, 14.0 \rangle.$$

We consider the following simple continuous query q that selects sensors located in the office and subscribes to their temperature streams:

$$q = \beta_{\langle temperatureNotifications, sensor \rangle}(\sigma_{area="office"}(sensors)).$$

We calculate the tuple statistics for the intermediary output XD-Relation $s_1 = \sigma_{area="office"}(r)$*. We consider that the selection predicate has a mean selectivity of 0.2, i.e. 20% of sensors are located in the office. A selection operator does not modify the schema, so we consider the same tuple weight.*

$$\mathcal{T}_{s_1} = \langle 0.2 \times 120.0, 0.2 \times 0.025, 14.0 \rangle = \langle 24.0, 0.005, 14.0 \rangle.$$

We can also calculate the tuple statistics for the query output XD-Relation $s_2 = \beta_{\langle temperatureNotifications,sensor\rangle}(s_1)$*, although it does not take part to the query cost computation. We consider that the subscription binding pattern* $temperatureNotifications$ *has a mean selectivity of 0.25, i.e. a tuple is inserted every 4 instants for each subscription. As the virtual attribute* $temperature$ *is realized, we consider an additional weight of 8 bytes per tuple (the size of an attribute of type* REAL*). As the output XD-Relation is infinite, the tuple number is set to* ∞*.*

$$\mathcal{T}_{s_2} = \langle \infty, 0.25 \times 0.005, 8.0 + 14.0 \rangle = \langle \infty, 0.00125, 22.0 \rangle.$$

We now calculate the initial cost of q*,* $\mathcal{C}_q^*$*, which is the cost of its two operators* σ *and* β*. For the selection operator, the invocation/subscription cost is null, as it is not a binding operator. The computation cost is the number of tuples that are read, stored and/or computed: as a selection operator only reads the tuples and evaluates the selection formula on them (it does not need to compute new tuples), we consider this cost to be simply the number of tuples of the operand. The tuple number and size of read tuples can be simply derived from the tuple statistics.*

$$\mathcal{C}_\sigma^* = \langle 0.0, 120.0, 120.0, 120.0 \times 14.0 \rangle = \langle 0.0, 120.0, 120.0, 1680.0 \rangle.$$

For the subscription binding operator, the invocation/subscription cost corresponds to the number of read tuples from s_1*, as each tuple leads to one subscription. The computation cost is the number of read tuples plus the number of generated tuples, whereas the tuple number and size can be simply derived from the tuple statistics of the operand.*

$$\mathcal{C}_\beta^* = \langle 24.0, 24.0 + 0.25 \times 24.0, 24.0, 24.0 \times 14.0 \rangle = \langle 24.0, 30.0, 24.0, 336.0 \rangle.$$

The initial cost of q *is then:*

$$\mathcal{C}_q^* = \mathcal{C}_\sigma^* + \mathcal{C}_\beta^* = \langle 24.0, 150.0, 144.0, 2016.0 \rangle.$$

The running cost of q*,* $\mathcal{C}_q^+$*, can be calculated in a similar way. From the tuple statistics, we consider that 0.025 tuples are inserted and deleted at each instant in the base operand. The selection operator only need to read and evaluate the selection formula on newly inserted tuples.*

$$\mathcal{C}_\sigma^+ = \langle 0.0, 0.025, 0.025, 0.025 \times 14.0 \rangle = \langle 0.0, 0.025, 0.025, 0.35 \rangle.$$

For the subscription binding operator, all tuples correspond to subscriptions, then the invocation/subscription cost is the same as for the initial cost. The computation cost is the number of newly inserted tuples that need to be taken into account plus the number of generated tuples that is the same as for the initial cost (as the mean number of subscriptions does not change). The tuple number and size of read tuples is calculated only for newly inserted tuples.

$$\mathcal{C}_\beta^+ = \langle 24.0, 0.005 + 0.25 \times 24.0, 0.005, 0.005 \times 14.0 \rangle = \langle 24.0, 6.005, 0.005, 0.07 \rangle.$$

The running cost of q *is then:*

$$\mathcal{C}_q^+ = \mathcal{C}_\sigma^+ + \mathcal{C}_\beta^+ = \langle 24.0, 6.03, 0.03, 0.42 \rangle.$$

In summary, the cost of the continuous query q is:

$$\mathcal{C}_q^* = \langle 24.0, 150.0, 144.0, 2016.0 \rangle, \mathcal{C}_q^+ = \langle 24.0, 6.03, 0.03, 0.42 \rangle.$$

3.4.3 Goal of query optimization

The general goal of query optimization is, given a query q, to find the query q' among queries equivalent to q that minimizes a cost. In our setting, we have defined query equivalence and a cost for continuous and one-shot queries over relational pervasive environments. We can now devise optimization techniques for such queries.

The goal of our optimization process is first to minimize the number of interactions with services, and then the more standard optimization goal of reducing the computation and memory resources needed for the execution of a query. For the sake of simplicity, we choose the lexicographical order between two costs that corresponds to the stated goal: the invocation/subscription cost is first compared, then, in case of equality, the computation cost, the tuple number and finally the tuple size. Note that the general goal of reducing the number of tuples in intermediary relations can impact on the four metrics: the invocation/subscription cost for binding operators, and the three other metrics for all operators.

In the case of continuous queries, two costs are computed: the initial cost $\mathcal{C}_q^*$ and the running cost $\mathcal{C}_q^+$. The running cost has a higher priority than the initial cost, as the running cost is "paid" for each instant from the instant when the query is started.

As binding operators with active binding patterns can not be reorganized without losing query equivalence, we only consider queries that do not involve such operators. Those queries can nevertheless be subqueries of larger queries that involve active binding operators. Any queries can then be optimized, fully or by parts.

3.4.4 Rule-based query optimization

We propose a classical rule-based query optimization (see for example [GMWU99]) for one-shot and continuous queries over a relational pervasive environment.

Standard optimization rules from the relational model are still valid for redefined relational operators over X-Relations or XD-Relations, e.g., pushing down selections and projections.

In order to reduce the tuple size of intermediary relations, assignment operators and binding operators may be pushed up in the query tree so that their realized attributes appear later, as virtual attributes do not contribute to tuple weight. This rule may however imply an increase of the number of invocations or subscriptions, depending on the rest of the query.

Example 3.7 (Query Optimization) *One-shot queries presented in Table 3.15 (page 65) are an example of query optimization: query Q_2 is an optimized version for query Q'_2 (whereas Q_1 and Q'_1 are not equivalent).*

As stated previously in Section 3.3, the choice of tagging binding patterns as active or passive is up to the application developers and impacts on query equivalence, and therefore on the optimization opportunities. Considering the invocation prototype takePhoto as passive leads to the optimization of Q'_2 into Q_2, whereas considering it as active leads to the non-equivalence between Q'_2 and Q_2.

3.5 The Serena SQL

In this section, we define a SQL-like language to express service-oriented one-shot and continuous queries over a relational pervasive environment in a declarative way, namely the Serena SQL. The Serena SQL is based on the different operators defined in the Serena algebra. We first define the Serena SQL for queries over XD-Relations, with operators defined over X-Relations for one-shot queries and over XD-Relations for continuous queries. We then define a specific Serena SQL syntax for the service discovery operator.

3.5.1 Queries over XD-Relations

The syntax for one-shot and continuous queries over XD-Relations is presented through its BNF grammar in Table 3.17. A generic example of this syntax is given in Table 3.18.

Serena SQL queries are based on the same model as standard SQL queries. The two mandatory clauses are SELECT and FROM. The FROM clause specifies the operand XD-Relations used by the query. For infinite XD-Relations, a window must be given in this clause. The SELECT clause represents the final projection of attributes, and thus the schema of the output XD-Relation (or X-Relation for one-shot queries). Attributes are tagged as VIRTUAL if they are virtual attributes in the output schema, otherwise they are real attributes.

The optional WHERE clause is also the same as in standard SQL queries. We however limit the global condition to be a conjunction of atomic conditions, i.e. using only a logical AND.

In order to differentiate between one-shot and continuous queries, we introduce a dynamicity clause. It can be either ONCE for one-shot queries, UPDATING for continuous queries resulting in a finite XD-Relation (which is the default dynamicity if not specified), or STREAMING for continuous queries resulting in a infinite XD-Relation. In the last case, the clause represents an additional streaming operator at the head of the query and specifies the event type of the operator with the keyword UPON followed by the name of one of the possible event types: `insertion`, `deletion` or `heartbeat`.

Realization operators are represented by two different optional clauses. The WITH clause represents assignment operators, with a list of assignments between a virtual attribute and either another attribute or a constant. The USING clause represents binding operators, with a list of binding patterns identified by their associated prototype name. Subscription binding patterns must be associated with a window, as they produce infinite XD-Relations, like for the FROM clause.

3.5.2 Service discovery queries

We have chosen to separate the syntax for the service discovery operator from the main syntax of Serena SQL queries over XD-Relations, as this operator can not be smoothly integrated using the standard SQL syntax. We then define a new type of query that is not SELECT - FROM - WHERE anymore, but DISCOVER SERVICES PROVIDING. Like the Serena algebra operator, this syntax does not specify any operand XD-Relation, as the implicit operand is the set of available services discovered in the pervasive environment. The query is defined like a view, as the service discovery query is meant to feed a XD-Relation. The explicit definition of the XD-Relation schema simplifies the expression of the query as the only remaining elements are the service selection criteria. It is to be noted that we consider only continuous service discovery queries, as one-shot queries can also use the resulting XD-Relation through their instantaneous X-Relation.

The syntax for service discovery queries over XD-Relations is presented through its BNF grammar in Table 3.19. An example of this syntax is given in Table 3.20.

The service selection criteria consist in a list of properties (or attributes) and a list of invocation or subscription prototypes that services must provide and implement. Prototypes are identified not only by their name, but by their complete method signature, and subscription prototypes are denoted by the keyword STREAMING, like for subscription binding patterns. Properties are identified by their name and their type, in a similar way to schema attributes. It is to be noted that properties can be mapped explicitly to attributes from the XD-Relation schema if their names are not already the same.

Table 3.17: Serena SQL syntax for continuous and one-shot queries

```
<SERENA_COMMAND> ::=
  [ CREATE <XD_RELATION> AS ] <SERENA_QUERY> ';' | ... ;

<SERENA_QUERY> ::=
  <SELECT_CLAUSE>
  [ <DYNAMICITY_CLAUSE> ]
  <FROM_CLAUSE>
  [ <WITH_CLAUSE> ]
  [ <WHERE_CLAUSE> ]
  [ <USING_CLAUSE> ];

<SELECT_CLAUSE> ::=
  SELECT ( '*' | ( <SELECT_ATTRIBUTE> ( ',' <SELECT_ATTRIBUTE> )* );

<SELECT_ATTRIBUTE> ::=
  ( <ATTRIBUTE_IDENTIFICATION> [ AS <attributealias_name> ] [ VIRTUAL ] ;

<ATTRIBUTE_IDENTIFICATION> ::=
  [ <tablealias_name> '.' ] <attribute_name>;

<DYNAMICITY_CLAUSE> ::=
  ONCE | UPDATING | STREAMING UPON <EVENT>;

<EVENT> ::=
  'insertion' | 'deletion' | 'heartbeat' ;

<FROM_CLAUSE> ::=
  FROM <TABLE_IDENTIFICATION> ( ',' <TABLE_IDENTIFICATION> )* ;

<TABLE_IDENTIFICATION> ::=
  <table_name> [ '[' <window_size> ']' ] [ <tablealias_name> ];

<WITH_CLAUSE> ::=
  WITH <ATTRIBUTE_ASSIGNMENT> ( ',' <ATTRIBUTE_ASSIGNMENT> )* ;

<ATTRIBUTE_ASSIGNMENT> ::=
  <ATTRIBUTE_IDENTIFICATION> ':=' ( <ATTRIBUTE_IDENTIFICATION> | <constant_value> );

<WHERE_CLAUSE> ::=
  WHERE <WHERE_PREDICATE> ( AND <WHERE_PREDICATE> )* ;

<WHERE_PREDICATE> ::=
  <ATTRIBUTE_IDENTIFICATION> <PREDICATE_OP> ( <ATTRIBUTE_IDENTIFICATION> | <constant_value> );

<PREDICATE_OP> ::=
  '=' | '!=' | '<' | '>' | '<=' | '>=' ;

<USING_CLAUSE> ::=
  USING <USING_BINDINGPATTERN> ( ',' <USING_BINDINGPATTERN> )* ;

<USING_BINDINGPATTERN> ::=
  [ <tablealias_name> '.' ] <bindingpattern_name> [ '[' <window_size> ']' ] ;
```

Table 3.18: Generic example of the Serena SQL syntax

```
SELECT r1.att1, r1.att2, s3.att3, s3.att4, r4.att5 VIRTUAL
ONCE / UPDATING / STREAMING UPON insertion/deletion/heartbeat
FROM r1, r2, s3[window], r4
WITH r1.att2 := constant, s3.att4 := r4.att5
WHERE predicate AND predicate AND ...
USING r2.ibp1, r1.ibp2, r4.sbp3[window]
```

Table 3.19: Serena SQL syntax for service discovery queries

```
<SERENA_COMMAND> ::=
  CREATE <XD_RELATION> AS <SERVICE_DISCOVERY_QUERY> ';' | ... ;

<SERVICE_DISCOVERY_QUERY> ::=
  DISCOVER SERVICES [ PROVIDING ( <PROPERTY> | <PROTOTYPE> )+ ];

<PROPERTY> ::=
  PROPERTY <property_name> <DATA_TYPE> [ AS <attribute_name> ];

<PROTOTYPE> ::=
  PROTOTYPE <prototype_name>
    '(' [ <DATA_TYPE> ( ',' <DATA_TYPE> )* ] ')' ':'
    '(' <DATA_TYPE> ( ',' <DATA_TYPE> )* ')' [ STREAMING ];
```

Table 3.20: Example of Serena SQL service discovery query

```
CREATE RELATION messengers (
    messenger SERVICE,
    protocol  STRING,
    address   STRING   VIRTUAL,
    text      STRING   VIRTUAL,
    sent      BOOLEAN  VIRTUAL,
    username  STRING   VIRTUAL,
    password  STRING   VIRTUAL,
    sender    STRING   VIRTUAL,
    message   STRING   VIRTUAL
  )
  USING BINDING PATTERNS (
    sendMessage[messenger] ( address, text ) : ( sent ),
    getMessages[messenger] ( username, password ) : ( sender, message ) STREAMING
  )
AS
  DISCOVER SERVICES PROVIDING
    PROPERTY messaging_protocol STRING AS protocol,
    PROTOTYPE sendMessage ( STRING, STRING ) : ( BOOLEAN ),
    PROTOTYPE getMessages ( STRING, STRING ) : ( STRING, STRING ) STREAMING
;
```

3.6 Summary

In this chapter, we have defined the Serena algebra over XD-Relations (respectively, X-Relations) to express continuous queries (respectively, one-shot queries) over a relational pervasive environment. A specific service discovery operator has also been defined to express the selection of available services from the environment that match a given service interface, i.e. that provide some properties and implement some invocation/subscription prototypes. We have then defined query equivalence for one-shot and continuous queries through the notion of action sets induced by a query against an environment, leading to some rewriting rules for Serena algebra expressions. The notion of active or passive binding pattern plays a key role in those definitions. We have proposed a simple cost model dedicated to relational pervasive environments that lead to basic rule-based query optimization techniques. Finally, a SQL-like query language, the Serena SQL, has been defined in order to declaratively express one-shot and continuous queries over a relational pervasive environment.

The declarative definition of pervasive applications is now possible, along with optimization opportunities using dedicated cost models. In the next chapter, we present the implementation of our framework, leading to the realisation of a complete Pervasive Environment Management System, along with scenarios using real and simulated data sources in order to test the expressiveness and performance of our model.

4

Toward a Pervasive Environment Management System

Chapter Outline

4.1 Architecture of a PEMS **82**
4.2 Implementation of the SoCQ PEMS **84**
4.2.1 The OSGi framework 84
4.2.2 Common SoCQ libraries 85
4.2.3 The Environment Resource Manager 87
4.2.4 The Extended Table Manager 89
4.2.5 The Query Processor 90
4.2.6 Interacting with the SoCQ PEMS 92
4.3 Experimentation **95**
4.3.1 Scenario: “Temperature Surveillance” 95
4.3.2 Scenario: “RSS Feeds” 102
4.3.3 Conclusion of experimentation 106
4.4 Summary **106**

In order to validate our approach and conduct experiments, we have designed and developed a prototype of a Pervasive Environment Management System (PEMS). The role of a PEMS is to manage a relational pervasive environment, with its dynamic data sources and set of services, and to execute continuous queries over this environment.

Our prototype enables users to interact with a relational pervasive environment through the PEMS without worrying about low-level technical considerations like programming languages or network protocols. Our prototype compiles and executes the Serena DDL, DML and SQL (for one-shot and continuous queries). Note that queries can also be expressed directly by Serena algebra expressions.

In Section 4.1, we propose a general architecture for a PEMS that is modular and distributed. The role of each module is defined and their interactions are described.

In Section 4.2, we describe the implementation of our PEMS prototype. The whole architecture has been developed in Java [Java] using the OSGi framework [OSG], including UPnP technologies [UPn] for network issues. Each module of the architecture corresponds to an OSGi bundle. Existing OSGi bundles for UPnP implement protocols that allow service discovery and remote interactions. Parsers have also been developed for the Serena DDL, DML, SQL and algebra using the JavaCC technology [Javb].

In Section 4.3, we present two experimentation scenarios: the "Temperature Surveillance" scenario and the "RSS Feed" scenario. Both scenarios involve all types of data sources and queries. For each scenario, we first detail the required services and their implementation. We then describe the involved XD-Relations and queries, and report on the experimentation results.

4.1 Architecture of a PEMS

A PEMS is responsible of three main roles in order to manage a relational pervasive environment. Those roles are:

1. the management of the distributed functionalities of the pervasive environment, i.e. the distributed services: service discovery and remote interaction techniques are needed to enable the integration of those functionalities into the PEMS;
2. the management of the dynamic data sources: database-like management is needed to maintain a catalog of XD-Relations and to handle their dynamic content;
3. the execution of queries over the relational pervasive environment, i.e. one-shot and continuous queries over XD-relations and service discovery queries: an implementation of the query operators, as well as query optimization techniques, are needed for this role.

In order to fulfill those three main roles, we propose a modular architecture composed of three core modules on the "PEMS core" device: the Environment Resource Manager, the Extended Table Manager, and the Query Processor; and several distributed modules on so-called "PEMS peer" devices, the Local Environment Resource Managers. A PEMS GUI (Graphical User Interface) module is also present in order to simplify user interactions from a remote "PEMS client" device. The deployment of the different modules and their interactions are illustrated in Figure 4.1.

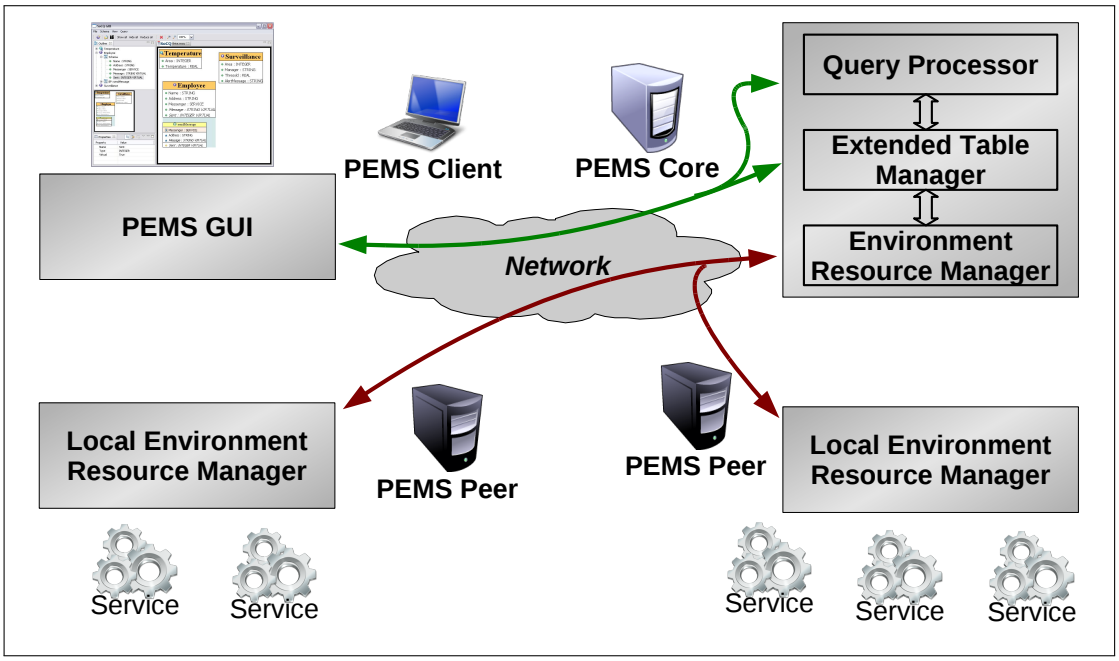


Figure 4.1: Overview of the PEMS Architecture

The Environment Resource Manager handles network issues for service discovery and remote interactions like property retrievals, method invocations and stream subscriptions. The core Environment Resource Manager discovers and communicates with the Local Environment Resource Managers that are distributed in the network. The Local Environment Resource Managers are the local representatives of the PEMS on the distributed devices of the pervasive environment: services only need to register to their Local Environment Resource Manager to be transparently available through the core Environment Resource Manager.

The Extended Table Manager builds the representation of the pervasive environment as a relational pervasive environment. It manages the catalog of XD-Relations and allows to manage their data and to use their binding patterns, relying on the core Environment Resource Manager for actual interactions with services. It also manages the list of available services discovered by the core Environment Resource Manager and produces notifications of service availability. Furthermore, it allows to subscribe to the

content of XD-Relations in order to retrieve their dynamic content in real-time through data notifications, for finite and infinite XD-Relations.

The Query Processor allows to execute one-shot queries over XD-Relations, to register continuous queries over XD-Relations and to execute them in a real-time fashion. It subscribes to the content of required XD-relations and uses required binding pattern invocations and subscriptions through the Extended Table Manager. The Query Processor also handles service discovery queries using notifications of service availability produced by the Extended Table Manager. For continuous queries, including service discovery queries, the query processor feeds the query results into the resulting XD-Relations managed by the Extended Table Manager.

In summary, the three core modules represent three layers: the Environment Resource Manager, along with the Local Environment Resource Manager, is a kind of pervasive environment middleware; the Extended Table Manager, relying on the Environment Resource Manager, builds the relational pervasive environment representation; and the Query Processor, relying on the Extended Table Manager, allows to execute queries over the relational pervasive environment. Each module is a layer that fulfills one of the required roles.

4.2 Implementation of the SoCQ PEMS

In this section, we describe the implementation of our PEMS prototype. The whole architecture has been developed in Java [Java] using the OSGi framework [OSG], including UPnP technologies [UPn] for network issues. Each module of the architecture corresponds to an OSGi bundle. Other OSGi bundles have been developed for the common libraries.

Bundle	*all PEMS bundles (excluding services)*
Packages	31
Classes / Interfaces	161 / 16
Methods	874
Total Lines Of Code	12254

4.2.1 The OSGi framework

OSGi is the dynamic module system for Java [OSG]: "The OSGi technology is a set of specifications that define a dynamic component system for Java. These specifications enable a development model where applications are (dynamically) composed of many different (reusable) components. The OSGi specifications enable components to hide their implementations from other components while communicating through services, which are objects that are specifically shared between components. This surprisingly

simple model has far reaching effects for almost any aspect of the software development process."

Components are implemented as *bundles*: a *bundle* is basically a JAR file (Java ARchive) whose *manifest* declares one of the bundle classes as the *Activator* that controls the lifecycle of this component, i.e. how it is started and stopped. A bundle can share some Java objects by registering them as services, and can search services registered by other bundles to interact with them. Registered services are associated with a name, typically the name of the Java interface they implement, in order to identify and use them easily.

We have chosen to use the OSGi technology for several reasons:

1. as claimed by the OSGi Alliance [OSG], "the OSGi specifications provide a mature and comprehensive component model with a very effective (and small) API": it is a very good candidate to implement a modular architecture for pervasive environments;
2. an OSGi framework can run on various devices with different capabilities, from small embedded devices to application servers, which is a requirement for applications in pervasive environments;
3. the principle of registered services that can be searched by other bundles matches conveniently the SoCQ model for services;
4. the existence of a large number of already developed bundles that can be reused, in particular for network communications (using network protocols JMX [JMX], UPnP [UPn], *etc.*).

OSGi bundles are deployed on an instance of the standardized *OSGi framework*, the "bundle runtime environment" [OSG]. Several implementations exist: we have chosen the open source implementation Apache Felix [Fel], also used by the open source application server GlassFish v3 [Gla]. An illustration of the execution of an OSGi framework is shown in Figure 4.2.

4.2.2 Common SoCQ libraries

In order to develop the whole architecture, we have first developed two common libraries used by the PEMS modules, the "Base Model" and the "Extended Model".

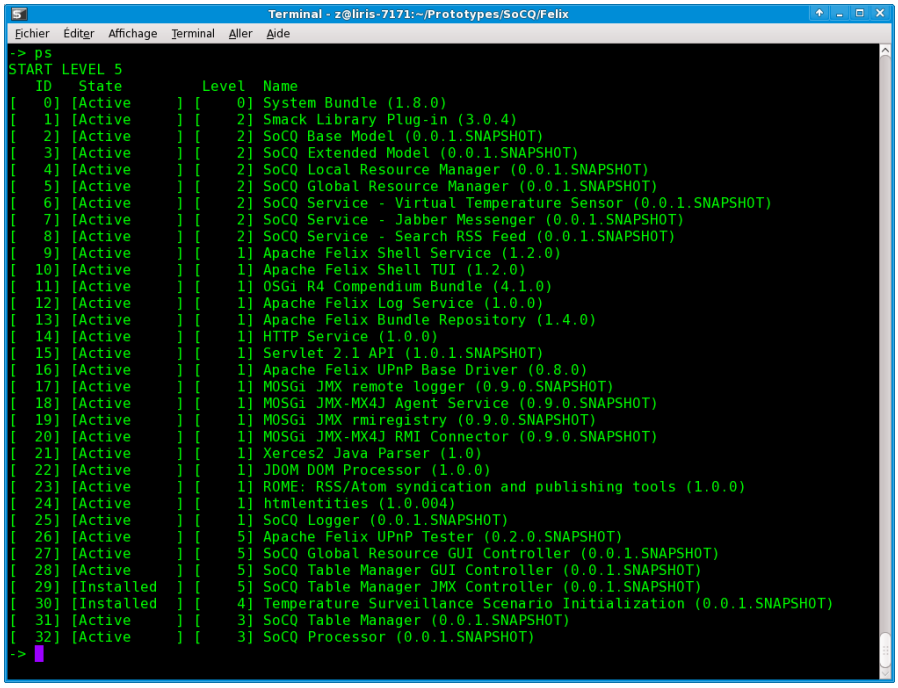


Figure 4.2: Execution of the OSGi Framework Instance (Apache Felix): the OSGi framework instance can provide a command-line interface. The `ps` command lists all installed bundles with their status (`Installed`, `Active`, `Resolved`). Here, all bundles required for the execution of the SoCQ PEMS are installed and started (for the core PEMS and the PEMS peer).

4.2.2.1 The Base Model library

Bundle	`fr.liris.cnrs.socq.model.base`
Packages	3
Classes / Interfaces	15 / 2
Methods	68
Total Lines Of Code	1504

The "Base Model" library implements the basic notion for the description of a relational pervasive environment. It contains classes and interfaces in the one hand for basic data description (data types, relation schema, tuples...), and in the other hand for service description and implementation (prototypes, properties, services).

This library is the only required bundle to develop new SoCQ services. A bundle providing SoCQ services simply contains classes that implement the `SoCQService` interface and registers instances of such classes as OSGi services in the OSGi framework. SoCQ services are then handled by a Local Environment Resource Manager.

4.2.2.2 The Extended Model library

Bundle	`fr.cnrs.liris.socq.model.extended`
Packages	8
Classes / Interfaces	51 / 2
Methods	298
Total Lines Of Code	3805

The "Extended Model" library implements the remaining notions for the description of a relational pervasive environment according to the SoCQ data model. It contains classes and interfaces for data source description, i.e. XD-Relation schemas and their binding patterns, and for query description in Serena algebra and Serena SQL.

It also contains parsers for the different languages that can be compiled by the SoCQ PEMS: Serena DDL, DML, SQL and algebra. Those parsers have been developed using the JavaCC technology [Javb], that enables the compilation of dedicated grammar files to Java classes implementing parsers. They can read character strings and build syntax tree according to those grammars. Syntax tree visitors have also been developed to build the internal representation of the language statements, i.e. Java objects representing Serena DDL, DML, SQL or algebra statements.

4.2.3 The Environment Resource Manager

The Environment Resource Manager is a kind of middleware for "traditional" pervasive environments. Its role is to discover distributed services and make them available for the PEMS core. Network issues for remote service discovery and interactions are handled by the UPnP technology.

4.2.3.1 The UPnP technology

The UPnP technology [UPn], now the "International Standard for Device Interoperability for IP-based Network Devices" (ISO/IEC 29341) as of December 2008, is a set of protocols based on Internet technologies (IP, TCP, UDP, HTTP...) that allows to discover, describe and control distributed devices on a network. *UPnP Devices* can be composed of several *UPnP Services*. *UPnP Services* can provide several *UPnP Actions*, similar to object methods, and notify subscribers of "evented" *State Variable* changes. UPnP enables a dynamic discovery of available devices, through the Simple Service Discovery Protocol (SSDP) that uses multicast advertising. A software that uses distributed UPnP Devices is called a *UPnP Control Point*.

We have chosen the UPnP technology because it fits the requirements for a protocol of pervasive environment:

1. services can be dynamically discovered;

2. a control point can remotely interact with services by invoking actions, i.e. invoking methods;

3. a control point can receive notifications in real-time through event subscription, that can be used as data stream subscription.

Furthermore, there exist OSGi bundles for UPnP (e.g., distributed on the Apache Felix download page [Fel]) that allow service discovery and remote interactions in a simple way: proxies for discovered UPnP Devices on the network are registered as local services into the OSGi framework. Nevertheless, as the service model for UPnP does not match exactly with the service model for the SoCQ PEMS, in particular for the data streams, we have built a specialized UPnP device type that maps the two models, without modifying the UPnP protocols. This specialized device type is handled by the Local Environment Resource Managers and the core Environment Resource Manager.

4.2.3.2 The Local Environment Resource Managers

Bundle	`fr.cnrs.liris.socq.resourcemanager.localmanager`
Packages	5
Classes / Interfaces	21 / 0
Methods	118
Total Lines Of Code	1368

The Local Environment Resource Managers search for locally registered SoCQ services, map them to specialized UPnP devices, and register those UPnP devices into the OSGi framework. The main part of this bundle is the dozen of classes needed to implement a UPnP device that represents and maps to a SoCQ service implementation. Service description, property retrieval and method invocation are mapped directly to dedicated UPnP Actions. Subscriptions to data streams are mapped to two UPnP Actions (subscribe/unsubscribe) and data notifications for the subscribed data streams are transmitted via a shared "evented" UPnP State Variable by multiplexing notifications.

The network issues are however not handled by this module, but only by the dedicated OSGi UPnP bundles. A "PEMS peer" that provides SoCQ services should then contain at least the "Base Model" bundle, the "Local Environment Resource Manager" bundle and the OSGi UPnP bundles.

4.2.3.3 The core Environment Resource Manager

Bundle	`fr.cnrs.liris.socq.resourcemanager.globalmanager`
Packages	2
Classes / Interfaces	11 / 2
Methods	54
Total Lines Of Code	799

The core Environment Resource Manager searches for locally registered UPnP devices representing SoCQ services, that are in fact proxies to distributed UPnP devices dynamically discovered by the OSGi UPnP bundles. It maps those UPnP devices back to simple SoCQ service proxies, in particular with the demultiplexing of data notifications for data stream subscriptions. It maintains a directory of available SoCQ services in the pervasive environment and can notify of service changes, i.e. when a SoCQ service appears or disappears. It also ensures that each SoCQ service has a unique identifier that is used as the service reference. Currently, the service reference is an integer number generated by the core Environment Resource Manager to ensure its uniqueness.

The "Environment Resource Manager" bundle is only dependent on the "Base Model" bundle and the OSGi UPnP bundles. It does not depend on the "Local Environment Resource Manager" bundle. Furthermore, if another software implements the specialized UPnP device type, the corresponding SoCQ services can be seamlessly integrated into the PEMS as the UPnP protocols are platform- and language-independent.

4.2.4 The Extended Table Manager

Bundle	`fr.cnrs.liris.socq.tablemanager`
Packages	6
Classes / Interfaces	28 / 6
Methods	144
Total Lines Of Code	1305

The Extended Table Manager handles XD-Relations and make an abstraction of the core Environment Resource Manager by managing XD-Relation binding patterns and by maintaining its own SoCQ service directory. This abstraction allows upper layers, namely the Query Processor, to depend on the "Extended Table Manager" bundle without directly depending on the "Environment Resource Manager" bundle. With regard to pervasive application development, this distinction is similar to the physical/logical independence in DBMSs.

The Extended Table Manager manages the catalog of XD-Relations. It allows to create and delete XD-Relations. An XD-Relation is a named XD-Relation schema associated with a content, i.e. a set of tuples. It is called internally an *extended table*, in analogy to tables in DBMS that implement data relations. For the SoCQ PEMS prototype, we have chosen to keep all data in memory without writing them to disk, as we focus on continuous query execution with dynamic data sources and not on reliable long-term data storage like in DBMS.

Tuples can be added into and removed from an extended table. The Extended Table Manager also allows to subscribe to the content of an extended table in order to retrieve its dynamic content in real-time through data notifications, i.e. tuple insertion and dele-

tion. For finite XD-Relations, i.e. extended data relation, inserted tuples are stored until they are deleted, and can be accessed at any moment. For infinite XD-Relations, i.e. extended data streams, inserted tuples are not stored and are only accessible through data notifications to active subscriptions at the moment of the insertion.

The Extended Table Manager maintains a SoCQ service directory and can produce service availability notifications by relying on the core Environment Resource Manager. In the same way, it also manages service interactions required by the use of invocation and subscription binding patterns of extended tables.

In order to handle concurrency problems with dynamic access to extended tables or even to the catalog, we have developed an asynchronous access system to the Extended Table Manager. The Extended Table Manager runs as an infinite loop in a single thread and gives an exclusive access to its internal state to one entity at a time. External commands, namely commands from a user controlled interface (e.g., a GUI) such as query commands or catalog access commands, are asynchronously posted to a command queue. The Extended Table Manager gives the exclusive access to the Query Processor during a certain time to make its required processing, potentially handling query commands. It then processes the remaining commands requiring its internal state. Such a model for execution ensures that there are no concurrent modifications of the internal state.

4.2.5 The Query Processor

Bundle	`fr.cnrs.liris.socq.socqprocessor`
Packages	7
Classes / Interfaces	35 / 4
Methods	192
Total Lines Of Code	3473

The Query Processor allows to execute one-shot and continuous queries over a relational pervasive environment, i.e. queries involving XD-Relations and discovered services managed by the Extended Table Manager. Queries are launched through query commands to the Extended Table Manager that forwards them to the Query Processor. They are managed differently depending on their type: continuous queries over XD-Relations, one-shot queries over XD-Relations, service discovery queries, and DDL/DML statements.

Continuous queries over XD-Relations are first parsed and transformed into a logical algebra form. They are then checked and optimized against the XD-Relation catalog managed by the Extended Table Manager, and finally transformed into a physical algebra form that is executable. Each logical algebra operator over XD-Relations, except the renaming operator that is purely logical, corresponds to an implemented physical operator: selection, projection, join, assignment, invocation/subscription binding,

window and streaming. We have however not yet implemented set operators (union, intersection, difference).

Physical operators are based on a tuple queue model: they consume one (or two for the join operator) input tuple queue representing their operand extended table, i.e. their operand XD-Relation implementation, and produce resulting tuples into an output tuple queue. Two additional physical operators are nevertheless needed to homogeneously handle those queries: a *source* operator that "reads" an existing XD-Relation, i.e. that produces an output tuple queue by subscribing to a source extended table from the catalog, and a *sink* operator that "writes" an existing XD-relation, i.e. that consumes an input tuple queue and alter accordingly the content of a target extended table from the catalog.

Queries in their executable form are a list of chained operators that can be executed independently in a real-time fashion, as they rely only on their input/output tuple queues. They are registered into a query list and executed continuously as long as they are not deleted by a query control command.

One-shot queries over XD-Relations are handled in a similar way to XD-relations. They are however executed only once against the current content of involved XD-Relations and not registered into a query list. They are deleted as soon as they have completed their processing. The same implementations of physical operators as for continuous queries are used, except that the source operator does not subscribe to source extended tables, but only retrieve their current content.

Service discovery queries are handled separately as they only involve the specific service discovery algebra operator and do not involve input XD-Relations. They are nevertheless first parsed and transformed into an internal logical representation composed of two elements: an output XD-Relation and a list of service attributes and prototypes to select discovered services from the environment. Their physical implementation handled service notifications forwarded by the Query Processor: they select discovered services that match their required interface (attributes and prototypes), build tuples representing those services by retrieving the service reference and values for the service attributes, and update their output extended table according to service appearance and disappearance. As they are continuous queries, they are registered into a query list, but separately from continuous queries over XD-Relations, and continuously executed by forwarding them service notifications as long as they are not deleted by a query control command.

DDL and DML statements are handled immediately, like one-shot queries: they are parsed into an internal logical representation and directly executed against the XD-Relation catalog managed by the Extended Table Manager. The execution of DDL statements alters the catalog by creating new XD-Relations or removing existing XD-Relations, i.e. creating or removing extended tables. The execution of DML statements results in tuple insertions into, or deletions from, some given existing XD-Relations, i.e. into/from existing extended tables.

4.2.6 Interacting with the SoCQ PEMS

The SoCQ PEMS prototype is modular and allows external modules to interact with it. More precisely, the Extended Table Manager is registered into the OSGi framework as an OSGi service. Other modules can search for this OSGi service and interact with it through an interface that allows to send asynchronous external commands: commands to retrieve the XD-Relation catalog or to subscribe to XD-Relation contents, and query commands (expressed in Serena SQL, DDL, DML) that are in fact handled by the Query Processor.

This modularity allows to build several interfaces to the PEMS prototype, including remote interfaces by adding dedicated communication modules to the OSGi framework. We have built a lightweight GUI (Graphical User Interface) using the Java SWING library in order to monitor and control the prototype on the same machine. This GUI is a small additional OSGi bundle that is not mandatory for the execution of the prototype. It is presented in Figure 4.3.

We have also built two additional remote interfaces:

1. A standalone desktop GUI application that presents a UML-like view of the relational pervasive environment. This application has been built as an Eclipse RCP application, i.e. using the Eclipse Platform as a container. It communicates remotely with the SoCQ PEMS prototype through the JMX protocol: a dedicated OSGi bundle that implements a JMX interface has been developed and should be installed on the same OSGi framework instance as the PEMS prototype, as well as OSGi bundles dedicated to JMX management (e.g., those distributed with Apache Felix). This application is shown in Figure 4.4.

2. A web application that allows a user to interacts with the "RSS Feeds" scenario (described in the next section). This web application should be hosted in the same OSGi framework instance as the PEMS prototype, along with an OSGi HTTP Service that implements a web server. It uses the Java servlet technology to make links between the user browser software and the PEMS prototype, through HTTP transfers of documents (e.g., HTML pages, XML documents). More complex web applications can also be hosted in an application server based on OSGi like GlassFish v3, that should then also host the PEMS prototype: this technique allows to use the large panel of technologies available on application servers, like the Google Web Toolkit for AJAX application development. This application is shown in Figure 4.5.

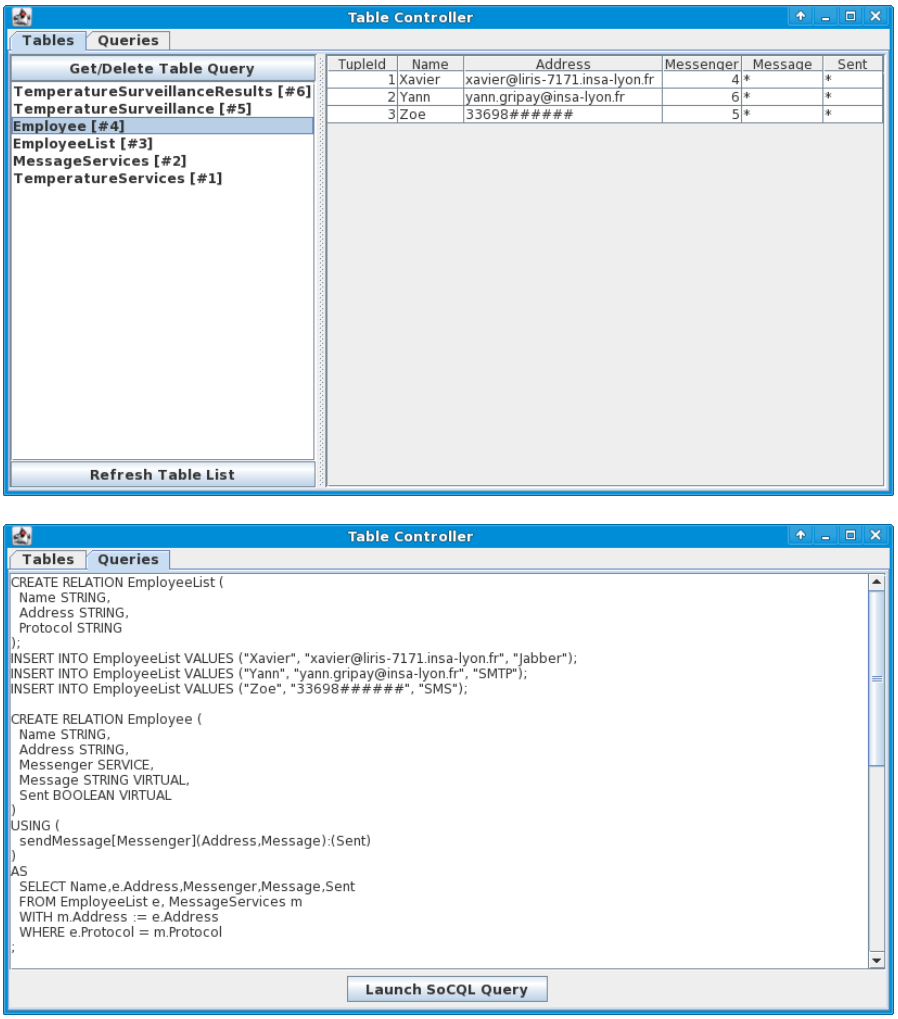


Figure 4.3: The PEMS Simple GUI (Table Panel and Query Panel)

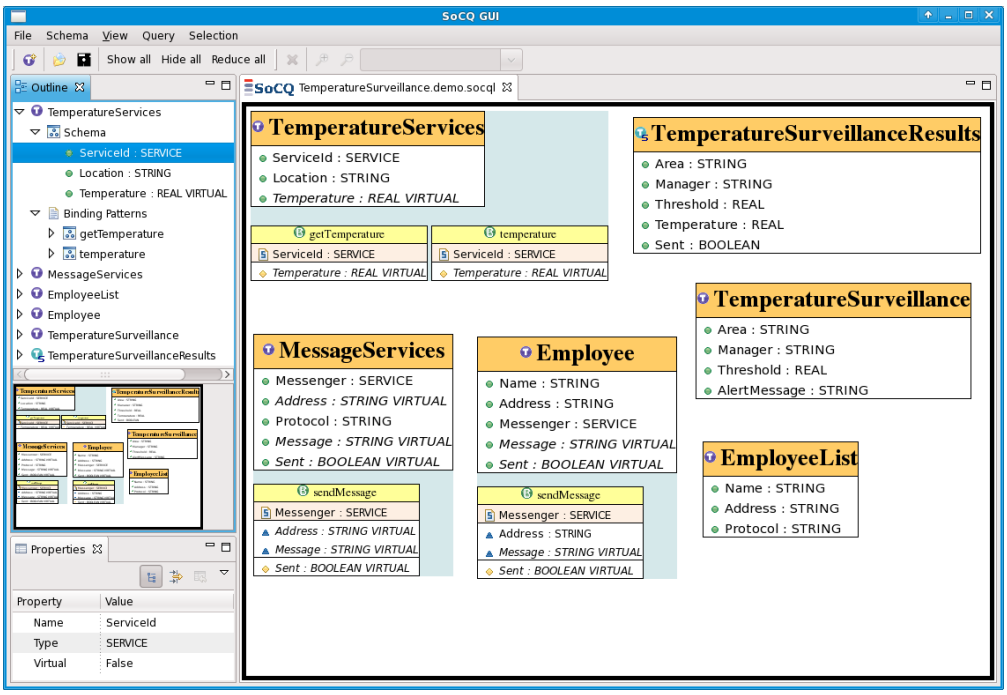


Figure 4.4: PEMS GUI developed as an Eclipse RCP application

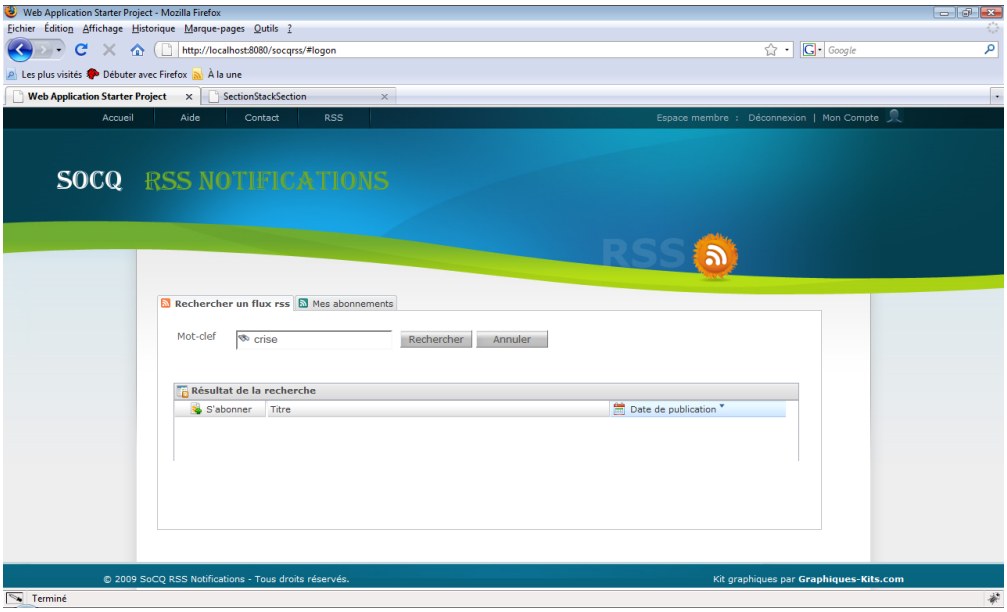


Figure 4.5: Interface of the Web Application for the “RSS Feeds” scenario

4.3 Experimentation

The SoCQ PEMS prototype implements all the required functionalities of a PEMS. In order to assess the validity and the usefulness of our approach, and to test our implementation, we have conducted some experiments. We have devised two scenarios that involve different kinds of data sources and services. The "Temperature Surveillance" scenario is the complete scenario detailed as the running example throughout this document and involves the monitoring of temperature sensors. The "RSS Feeds" scenario involve a more complex service that reads RSS feeds (Really Simple Syndication, or sometimes Rich Site Summary) from different sites, and allows users to subscribe to some topics. In the rest of this section, we detail the implementation of each of those two scenarios in term of SoCQ services, XD-Relations and SoCQ queries; and we report on the results of the execution of those scenarios.

4.3.1 Scenario: "Temperature Surveillance"

With the "Temperature Surveillance" scenario, we simulate the monitoring of temperature in a building where temperature sensors are distributed in the different areas of the building. The main goal is to send alert messages when the temperature exceeds some threshold in a given area. Basic functionalities such as retrieving the current temperature in an area or sending a message to someone can also be used separately in a simple way. In addition to the temperature sensors, various messaging services are present in order to send the alerts, as well as administration data for system configuration.

4.3.1.1 SoCQ services

Temperature sensors

Temperature sensors are represented in the relational pervasive environment as services that provide/implement:

- `Location STRING`, a property representing their location;
- `getTemperature():(temperature REAL)`, an invocation prototype returning the current temperature value;
- `temperature():(temperature REAL) STREAMING`, a subscription prototype providing a stream of temperature values.

We have implemented those temperature services with three kinds of devices. We have developed virtual sensors in order to control the temperature sent by the corresponding services: a small GUI with an horizontal slider allows to precisely set the virtual sensor to a given temperature value. Several virtual sensors can be launched at the same time in order to simulate many sensors in the environment, as illustrated in Figure 4.6.

Bundle	`fr.cnrs.liris.socq.service.virtualtemperaturesensor`
Packages	1
Classes / Interfaces	5 / 1
Methods	27
Total Lines Of Code	269

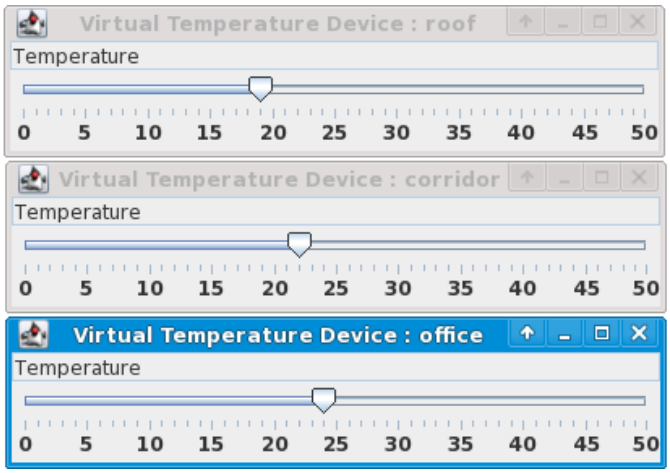


Figure 4.6: Virtual Temperature Sensors (roof, corridor, office)

We have also developed services that handle physical devices: one service for a *Thermochron iButton DS1921*, that is wired to a computer via a USB or serial port, using the communication library provided by the device constructor (1-Wire API for Java SDK); and one service for a *Sun SPOT*, a small wireless device embedding several sensors including a temperature sensor, using the library from the constructor (SunSPOT SDK). Both are presented in Figure 4.7.

Bundle	`fr.cnrs.liris.socq.service.realtemperaturesensor`
Packages	3
Classes / Interfaces	9 / 1
Methods	54
Total Lines Of Code	689

Figure 4.7: Physical Temperature Sensors (iButton, iButton base, SunSpot)

Messaging services

Messaging services are represented in the relational pervasive environment as services that provide/implement:

- `Protocol STRING`, a property representing their underlying protocol, i.e. their type of messaging service;
- `sendMessage(address STRING,text STRING):(sent BOOLEAN)`, an invocation prototype that allows to send a message to a given address.

We have implemented three different messaging services that act as a gateway to messaging servers. We have developed a service that allows to send e-mail messages through a given SMTP server, using the `javax.mail` API, and another service that send SMS messages through a commercial service (*Clickatell Bulk SMS Gateway*), using HTTP communications. We have also developed a service that interacts with a Jabber/XMPP instant messaging server (*Openfire* server from *Jive Software*), using a library dedicated to the XMPP protocol (*Smack* Java library, also from *Jive Software*). For this last service, we have additionally implemented a subscription prototype (`messages():(sender STRING, message STRING) STREAMING`) that provides the stream of text messages received by a specific instant messaging user.

Bundle	`fr.liris.cnrs.socq.service.messenger`
Packages	1
Classes / Interfaces	6 / 0
Methods	33
Total Lines Of Code	452

In order to interact with those messaging services, we have used some XMPP-compatible instant messaging clients (*Psi*, *Pidgin*,*iChat*), an e-mail client (*Mozilla Thunderbird*), and a smart phone for SMS.

4.3.1.2 XD-Relations and SoCQ queries

The relational pervasive environment for the Temperature Surveillance Scenario is composed of several XD-Relations. We first build two XD-Relations to discover on the one hand temperature services (`TemperatureServices`), and on the other hand messaging services (`MessageServices`). The DDL of those XD-Relations, along with their corresponding service discovery queries in Serena SQL, are shown in Table 4.1.

We build the list of "employees" in two steps: we first define a simple relation `EmployeeList` containing their name, address and the type of the address, i.e. the corresponding messaging protocol; we then define a XD-Relation `Employee` with an invocation binding pattern to send messages by combining the two XD-Relations `EmployeeList` and `MessageServices` in a query, in order to associate an available corresponding service to each address. The last XD-Relation `TemperatureSurveillance` allows to "configure" the surveillance system: its content specifies which employees must be alerted for a given area when the given threshold is exceeded. The DDL of those XD-Relations, as well as DML statements to insert data, are shown in Table 4.2.

Given those XD-Relations, we have devised a continuous query that combines temperature streams from sensors, the employee list and configuration data in order to send alert messages when needed. The result is a stream of alerts that have been sent. This query, expressed in the Serena SQL, is shown in Table 4.3.

While this continuous query is executed, the following behavior is set: when temperature sensors (physical or simulated) are heated over the threshold specified in `TemperatureSurveillance`, alert messages start to be sent to the "managers" of the associated area, by mail, instant message or SMS. This behavior is illustrated in Figure 4.8. Temperature sensors are dynamically discovered and integrated in the temperature stream without the need to stop the continuous query execution. Users can also be added to or removed from the XD-Relation `EmployeeList`, and the XD-Relation `TemperatureSurveillance` can be updated on-the-fly, in order to dynamically configure the system behavior.

Table 4.1: XD-Relations with Service Discovery Queries for the Temperature Surveillance Scenario

```
CREATE RELATION TemperatureServices (
  ServiceId SERVICE,
  Location STRING,
  Temperature REAL VIRTUAL
)
USING (
  getTemperature[ServiceId]():(Temperature),
  temperature[ServiceId]():(Temperature) STREAMING
)
AS
  DISCOVER SERVICES PROVIDING
    PROPERTY Location STRING,
    METHOD getTemperature ( ) : ( REAL ),
    STREAM temperature ( ) : ( REAL )
;

CREATE RELATION MessageServices (
  Messenger SERVICE,
  Address STRING VIRTUAL,
  Protocol STRING,
  Message STRING VIRTUAL,
  Sent BOOLEAN VIRTUAL
)
USING (
  sendMessage[Messenger](Address,Message):(Sent)
)
AS
  DISCOVER SERVICES PROVIDING
    PROPERTY Protocol STRING,
    METHOD sendMessage ( STRING, STRING ) : ( BOOLEAN )
;
```

Table 4.2: Administration XD-Relations for the Temperature Surveillance Scenario

```
CREATE RELATION EmployeeList (
  Name STRING,
  Address STRING,
  Protocol STRING
);
INSERT INTO EmployeeList VALUES ("Xavier", "xavier@liris-7171.insa-lyon.fr", "Jabber");
INSERT INTO EmployeeList VALUES ("Yann", "yann.gripay@insa-lyon.fr", "SMTP");
INSERT INTO EmployeeList VALUES ("Zoe", "33698######", "SMS");

CREATE RELATION Employee (
  Name STRING,
  Address STRING,
  Messenger SERVICE,
  Message STRING VIRTUAL,
  Sent BOOLEAN VIRTUAL
)
USING (
  sendMessage[Messenger](Address,Message):(Sent)
)
AS
  SELECT Name,e.Address,Messenger,Message,Sent
  FROM EmployeeList e, MessageServices m
  WITH m.Address := e.Address
  WHERE e.Protocol = m.Protocol
;

CREATE RELATION TemperatureSurveillance (
  Area STRING,
  Manager STRING,
  Threshold REAL,
  AlertMessage STRING
);
INSERT INTO TemperatureSurveillance VALUES ("roof", "Xavier", 45.0, "Alert: roof on fire!");
INSERT INTO TemperatureSurveillance VALUES ("corridor", "Yann", 35.0, "Do not run in the corridor!");
INSERT INTO TemperatureSurveillance VALUES ("office", "Zoe", 32.0, "Too hot in the office...");
```

Table 4.3: Main Query for the Temperature Surveillance Scenario

```
CREATE STREAM TemperatureSurveillanceResults (
  Area STRING,
  Manager STRING,
  Threshold REAL,
  Temperature REAL,
  Sent BOOLEAN
)
AS
  SELECT s.Area,Manager,Threshold,Temperature,Sent
  STREAMING UPON insertion
  FROM TemperatureServices t, TemperatureSurveillance s, Employee
  WITH Message := AlertMessage
  WHERE t.Location = s.Area
    AND Temperature > Threshold
    AND Manager = Name
  USING temperature [1], sendMessage
;
```

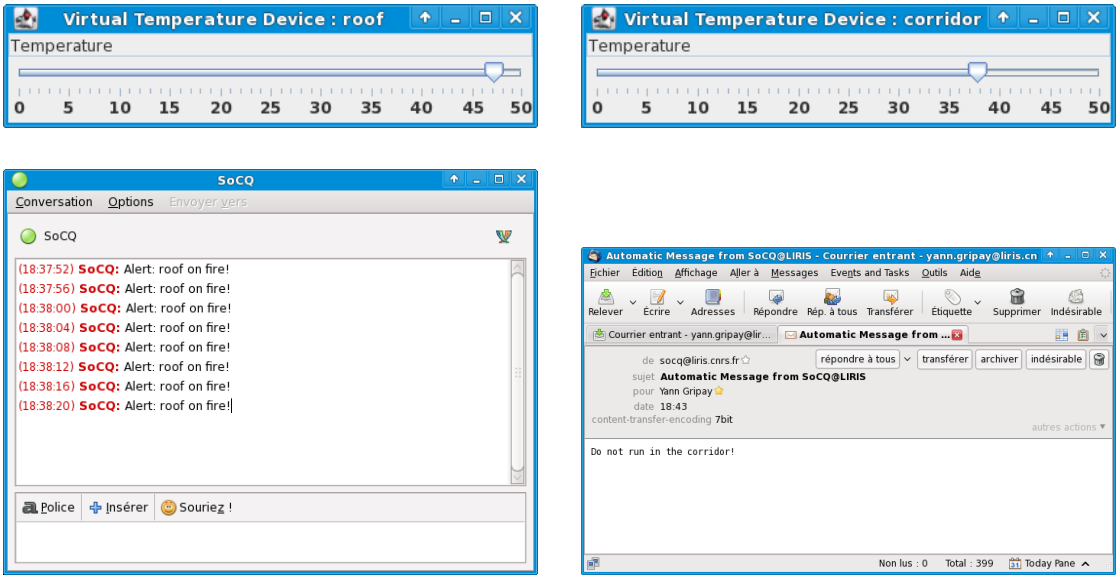


Figure 4.8: Illustration of the Execution of the Temperature Surveillance Scenario: when the temperature of the virtual sensor in the location "roof" is over the threshold (here, the temperature value is 47°C and the threshold value is set to 45°C in the XD-Relation `Surveillance`), an alert message is sent. It is received by Xavier in its instant messaging client (here, Pidgin). In the same way, when the temperature is over 35°C in the corridor, Yann receives an alert e-mail (using Mozilla Thunderbird).

4.3.2 Scenario: "RSS Feeds"

We have also experimented a scenario with RSS feeds published by several national and international information websites (e.g., french newspapers "Le Monde" and "Le Figaro", "CNN Europe"). It is based on a wrapper service that transforms RSS feeds into data streams: RSS feeds are periodically checked in order to insert tuples in the data stream when new items appear in the RSS feed. Using this service, we have devised a Serena query that maintain a table with all RSS items containing a given keyword (e.g., "Obama") published in the last one hour. We have also devised Serena queries for a more complex system of subscriptions allowing a user to receive notifications of RSS items of interest.

4.3.2.1 SoCQ services

The RSS service implements the following prototypes:

- `getAllRssFeeds() : (Url STRING)`, an invocation prototype returning a list of all RSS feed URLs managed by the service;
- `searchRssFeeds(Keyword STRING): (Url STRING)`, an invocation prototype returning a list of RSS feed URLs for RSS feeds that contain at least one item that matches the given keyword;
- `getRssFeedDescription(Url STRING):(FeedTitle STRING, FeedDescription STRING, FeedLink STRING, FeedDate STRING)`, an invocation prototype returning the RSS feed description for a given RSS feed URL;
- `subscribeRssFeed(Url STRING):(ArticleTitle STRING, ArticleDescription STRING, ArticleLink STRING, ArticleDate STRING) STREAMING`, a subscription prototype providing a stream of RSS items from the given RSS feed URL;
- `subscribeFilteredRssFeed(Url STRING, Keyword STRING):(ArticleTitle STRING, ArticleDescription STRING, ArticleLink STRING, ArticleDate STRING) STREAMING`, a subscription prototype providing a stream of RSS items from the given RSS feed URL that matches the given keyword.

This service stores a list of RSS feed URLs read from a configuration file. It uses the ROME open source Java library to parse RSS feed documents (in several format, e.g., RSS 1.0, RSS 2.0, Atom 1.0), in order to extract the RSS feed description and the list of RSS items (or articles). It manages data streams for the subscription prototypes by periodically checking the required RSS feeds for updates. Furthermore, it offers the possibility to specify a keyword to filter the list of RSS feed URLs or to filter RSS items from a given RSS feed.

Bundle	`fr.cnrs.liris.socq.service.rssfeed`
Packages	4
Classes / Interfaces	8 / 0
Methods	59
Total Lines Of Code	907

4.3.2.2 XD-Relations and SoCQ queries

In order to use the RSS service within queries, we first build a XD-Relation, associated with a service discovery query. This XD-Relation gives access to RSS feeds managed by the service and to their RSS items. Queries can use or not use the `Keyword` attribute: the XD-Relation binding patterns allow both functionalities. For our experimentation, this XD-Relation contains only one tuple, as we developed a single RSS service. The DDL of this XD-Relation is shown in Table 4.4.

Table 4.4: XD-Relation with Service Discovery Query for the RSS Feeds Scenario

```
CREATE RELATION RssFeeds (
  ServiceId SERVICE,
  Keyword STRING VIRTUAL,
  Url STRING VIRTUAL,
  FeedTitle STRING VIRTUAL,
  FeedDescription STRING VIRTUAL,
  FeedLink STRING VIRTUAL,
  FeedDate STRING VIRTUAL,
  ArticleTitle STRING VIRTUAL,
  ArticleDescription STRING VIRTUAL,
  ArticleLink STRING VIRTUAL,
  ArticleDate STRING VIRTUAL
)
USING (
  getAllRssFeeds[ServiceId]():(Url),
  searchRssFeeds[ServiceId](Keyword):(Url),
  getRssFeedDescription[ServiceId](Url):(FeedTitle,FeedDescription,FeedLink,FeedDate),
  subscribeRssFeed[ServiceId](Url):
      (ArticleTitle,ArticleDescription,ArticleLink,ArticleDate) STREAMING,
  subscribeFilteredRssFeed[ServiceId](Url,Keyword):
      (ArticleTitle,ArticleDescription,ArticleLink,ArticleDate) STREAMING
)
AS
  DISCOVER SERVICES PROVIDING
    METHOD getAllRssFeeds ( ) : (STRING),
    METHOD searchRssFeeds (STRING) : (STRING),
    METHOD getRssFeedDescription (STRING) : (STRING, STRING, STRING, STRING),
    STREAM subscribeRssFeed (STRING) : (STRING, STRING, STRING, STRING),
    STREAM subscribeFilteredRssFeed (STRING, STRING) : (STRING, STRING, STRING, STRING)
;
```

Using this XD-Relation `RssFeeds`, we have devised two simple Serena continuous queries that provide the last RSS items from all available RSS feeds containing the keyword "Obama", with a one-hour window (the window size is 3600 seconds). The first query does not use the `Keyword` attribute, but uses a regular expression operator between string values to filter RSS items: the predicate `ArticleTitle > ".*Obama.*"` returns true if the attribute value matches the Java-style regular expression, here if it contains the word "Obama". The second query uses the `Keyword` attribute, along with the binding patterns `searchRssFeeds` and

subscribeFilteredRssFeed. Those queries, expressed in the Serena SQL, are shown in Table 4.5.

While those continuous queries are executed, the resulting tables are continuously updated, when news of interest appeared and when one-hour-old news expired.

Table 4.5: Simple Queries for the RSS Feeds Scenario

```
CREATE RELATION RssItemsForObama1 (
  ArticleTitle STRING,
  ArticleDescription STRING,
  ArticleLink STRING,
  ArticleDate STRING
)
AS
  SELECT ArticleTitle, ArticleDescription, ArticleLink, ArticleDate
  FROM RssFeeds
  WHERE ArticleTitle > ".*Obama.*"
  USING getAllRssFeeds, subscribeRssFeed[3600]
;

CREATE RELATION RssItemsForObama2 (
  ArticleTitle STRING,
  ArticleDescription STRING,
  ArticleLink STRING,
  ArticleDate STRING
)
AS
  SELECT ArticleTitle, ArticleDescription, ArticleLink, ArticleDate
  FROM RssFeeds
  WITH Keyword := "Obama"
  USING searchRssFeeds, subscribeFilteredRssFeed[3600]
;
```

We have also devised a more complex Serena query that combines the XD-Relation RssFeeds with the XD-Relation Employee (from the Temperature Surveillance scenario) in order to send notifications to users when news of interest appear. In a similar way to the XD-Relation TemperatureSurveillance, we build an additional XD-Relation RssSubscriptions that allows to "configure" the notification system, i.e. to specify which user is interested in which RSS feed with which filter keyword. The Serena continuous query then combines three XD-Relations in order to set the required behavior: a notification is sent to a user as soon as an item of interest appears on the specified RSS feeds, using the subscription binding pattern subscribeFilteredRssFeed. The XD-Relation RssSubscriptions and the query are shown in Table 4.6.

Table 4.6: XD-Relations and Main Serena Query for the RSS Feeds Scenario

```
CREATE RELATION RssSubscriptions (
  UserName STRING,
  RssFeedName STRING,
  Keyword STRING
);
INSERT INTO RssSubscriptions VALUES ("Xavier", "Le Figaro",  "Sarkozy");
INSERT INTO RssSubscriptions VALUES ("Yann",   "Le Monde",   "Obama");
INSERT INTO RssSubscriptions VALUES ("Zoe",    "CNN Europe", "Merkel");

CREATE STREAM RssSubscriptionNotifications (
  ArticleTitle STRING,
  UserName STRING,
  Sent BOOLEAN
)
AS
  SELECT ArticleTitle, UserName, Sent
  STREAMING UPON insertion
  FROM RssFeeds rf, RssSubscriptions rsb, Employee e
  WITH rf.Keyword := rsb.Keyword, Message := ArticleTitle
  WHERE e.Name = rsb.UserName
    AND rf.FeedTitle = rsb.RssFeedName
  USING getAllRssFeeds, getRssFeedDescription, subscribeFilteredRssFeed[1], sendMessage
;
```

4.3.3 Conclusion of experimentation

We have successfully tested the two scenarios detailed in this section, the Temperature Surveillance scenario and the RSS Feeds scenario. They show that the SoCQ data model can handle different kinds of services implementing invocation and subscription prototypes, i.e. providing remote methods and data streams: messaging services, temperature sensor services, RSS feed services, *etc.* The implementation of those services is relatively simple and can be seamlessly deployed and integrated into the PEMS with no effort thanks to the OSGi framework, even for distributed services.

All those services can be easily combined with traditional data and data streams in order to build simple and complex behaviors of pervasive applications in a declarative way, through Serena SQL queries. Moreover, service discovery queries manage the dynamic integration of distributed services that can appear or disappear at any moment.

Clearly, developing pervasive applications with a PEMS is simpler than with a *ad hoc* development. For the two scenarios, Serena SQL queries are relatively short (less than ten lines of SQL-like syntax). The declaration of XD-Relations, required to describe the pervasive environment, easily combines SQL-like table creation syntax with binding patterns. In comparison, developing such applications with an imperative language like Java would require specific code to dynamically discover services, to handle service invocations and the more complex service subscriptions, to read data from data sources, *etc.* Many frameworks, in particular OSGi, may simplify some of this work (e.g., service discovery), but nevertheless require complex code to handle dynamic and asynchronous behaviors defined by SoCQ queries. Furthermore, maintaining a set of declarative SoCQ queries with less than ten lines each is far simpler than maintaining a piece of complex code.

4.4 Summary

In this chapter, we have presented the design and implementation of a Pervasive Environment Management System (PEMS). A PEMS manages a relational pervasive environment, i.e. XD-Relations and distributed services, and can execute one-shot and continuous queries over it. We have proposed a PEMS architecture that is modular and distributed. We have detailed the implementation of our PEMS prototype using the Java-based OSGi framework, along with the UPnP technology for distributed service management, and also the JavaCC technology to build parsers for the Serena DDL, DML, SQL and algebra. Our PEMS prototype can be launched in a stand-alone way with a simple GUI, or can be integrated in an OSGi-based application server to be used as a middleware for complex web applications. We have then presented our experimentation through two scenarios: the monitoring of distributed temperature sensors, and the management of user subscriptions to RSS feeds from different websites.

Our implementation of the PEMS, along with the experimentation scenarios, shows the feasibility of our approach, as well as its adaptation to different kinds of data sources and services. The design of a PEMS indeed leads to the declarative definition of pervasive applications that may combine data, data streams and services.

A short-term perspective that remains to be done is to assess the scalability and the robustness of our PEMS in some specific application domains, e.g., intelligent homes. In the context of pervasive environments, this is nevertheless not a trivial issue since, to the best of our knowledge, no benchmark can be used for that purpose.

5
Related Work

Chapter Outline

5.1 Pervasive environments **110**
5.1.1 Overview 110
5.1.2 High-level projects of pervasive environments 110
5.1.3 Addressing pervasive system requirements 111
5.1.4 Modeling pervasive environments 112
5.2 Related database research **113**
5.2.1 Data streams 113
5.2.2 Data and service integration 116
5.3 Enabling technologies **120**

5.1 Pervasive environments

5.1.1 Overview

The idea of ubiquitous computing, or pervasive computing, was initiated by Mark Weiser in his famous article "The Computer for the 21st Century" [Wei91] in 1991. Despite the great technology advances that have happened since, his vision remains today a challenge with numerous open issues for computer science and computer engineering in order to build "pervasive information systems" [KG07a]. Pervasive computing is "a paradigm for the 21st century" [SM03] where devices are not used to run local applications, but are a way for users to access a global "application-data space" in order to perform their tasks. The computing environment can then become an "information-enhanced physical space" [SM03], a "Data Space" [FHM05] or a "Programmable Pervasive Space" [HMEZ+05].

In order to build pervasive computing environments, "connecting the physical world" [ECPS02] to a ubiquitous network is required, through the deployment of sensors and actuators embedded in mobile devices, in common everyday objects, or even integrated in buildings and other infrastructures. It leads to the issue of discovering available resources [ZMN05] in a dynamic distributed environment.

5.1.2 High-level projects of pervasive environments

Many high-level research projects on pervasive environments have been conducted by research teams throughout the world. Besides their technological propositions, the examples they provide focus mainly on intelligent workspaces or intelligent homes.

In the Oxygen project [MIT, GFS02, KTD+03], new devices for the end-user have been defined, called H21s, that include computing and communication facilities using a dedicated network technology, as well as multimodal user interfaces. They propose a goal-oriented programming technique [SPP+03] to develop applications that adapt to the ever changing pervasive environment. Applications are decomposed in two levels: the goal level that abstracts the end-user task, and the software component level containing actual code realization that can be assembled dynamically by the goal planning mechanism. In a similar way, the Aura project [Car, GSSS02] aims at providing users with an invisible halo of computing and information services that persists regardless of location, in order to allow them to interact continuously with their "aura" that follows them everywhere and at every time. It focuses on the notion of task, i.e. the set of applications and data a user is currently using, that should be instantiated in every environment this user goes into, using any available appliance.

The EasyLiving project [Mic, BMK+00] also works on intelligent environments (in-home or in-office). It focuses on the context sensing and modeling as well as the interaction with the end-user using computer vision and visual user interaction, adaptation of

user interfaces... Application adaptation is based on an abstraction of users' tasks and on the discovery and composition of available services in the environment. The context modeling is enhanced by a precise geometric model of the environment, that allows to specify situations involving the relative locations of physical and logical entities, e.g., an object entering a certain area, or being close to a certain software component. The Sentient project [ATT] also focuses on location-based interactions: devices, called "bats" because they embed an ultrasound transmitter, enable a precise localization of the different physical entities of the environment. A central controller manages the world model that stores the correspondence between bats and their owners, and can launch predefined actions when some situations are detected.

The Portolano project [Unib] focuses on sensors management and networking. They propose an infrastructure based on mobile agents that interact with applications and users. Data-centric routing automatically migrates data among applications on the user's behalf. Data become "smart" and serve as an interaction mechanism within the environment. The Endeavour project [Unia] aims at providing a planet-scale, self-organizing, and adaptive "Information Utility". Their main objective is to arbitrarily and automatically distribute data among "Information Devices", where data are seen as software components that can advertise themselves, provide their own adaptable user interface and their own negotiation process for their integration in applications. The OneWorld project [GDL+04] also provides a data-oriented "system support for pervasive applications" in order to simplify data and application sharing among distributed devices, including user interface devices.

In the context of pervasive systems, the SoCQ data model enables the unified representation of heterogeneous data sources and services. Some parts of pervasive applications, like users' tasks, event detection and notification, can be defined through declarative SoCQ queries that involve and combine dynamically discovered services, data streams, and databases. We do not however focus on location-dependent functionalities, although a geometric model layer could be added into the PEMS, nor on data sharing among devices. The SoCQ PEMS can nevertheless be used as a middleware to simplify the development of those applications.

5.1.3 Addressing pervasive system requirements

One current trend for the management of heterogeneous devices in a pervasive environment is the leveraging of the Service Oriented Architecture (SOA) paradigm [Erl05]. SOA was mainly an enterprise-level effort to deploy business functionalities as services with well-defined interfaces in order to simplify the development and the maintenance of applications. Among others, Web Service technologies have been developed to implement SOA designs. SOA concepts have however been reused in different contexts, in particular for the design of distributed applications. In the more complex context of pervasive environments, SOA can be adapted to handle device-level issues. The SODA

(Service-Oriented Device Architecture [dDCK+06] or Service-Oriented Device & Delivery Architecture [SOD]) approach is to represent devices with distributed services. It enables the integration of device functionalities into existing SOA systems, or simply to manage devices from pervasive environments with SOA techniques.

Developing applications for pervasive environments remains a challenge today. A common approach for the actual development and execution of such applications is the use of middleware. In [dCYG08], a general software infrastructure model is proposed in order to detail with precision ten issues that a middleware should address: heterogeneity, scalability, dependability/security, privacy/trust, spontaneous interoperation, mobility, context awareness, context management, transparent user interaction, and invisibility. They however state that, although past high-level projects aimed at accomplishing several of those aspects, middlewares now tend to address only some specific issues.

In order to evaluate pervasive systems, a proposition of benchmark has been done in [RAMB+05]. It is a human-centered benchmark that nevertheless aims at providing objective indicators. Four general task sets are proposed to compare different systems: presentation tasks, notification/trigger-based tasks, user collaboration tasks, information finding tasks. Metrics are grouped in three domains: programmability, usability and security. It however mainly focus on programmability: programming effort needed to create new applications or support new devices, programming support for mobility or composition, context sensitivity, automation of actions. Security is evaluated by the expressiveness of policy, unobtrusiveness of mechanisms, and user control over private information. Usability reflects the effort (head turns, keystrokes...) and satisfaction of users to accomplish a given task with the system. This proposition of benchmark is interesting as it points out high-level characteristics that are (or should be) common among pervasive systems and enables a certain level of comparison.

With the SoCQ PEMS, we do not address all issues pointed out by [dCYG08] or [RAMB+05]. The SoCQ PEMS focuses on a declarative approach for the development of application involving heterogeneous devices and data sources. It is based on a SODA approach: functionalities of devices are represented by distributed services. Our approach focuses on the programmability level. We do not address security concerns, nor usability for the end-user.

5.1.4 Modeling pervasive environments

Modeling pervasive environments may be done at different levels. From a system view, architectures supporting pervasive computing are often modeled with layers (e.g., in [HMEZ+05]): from the physical layer with sensors and actuators, to the application layer that defines some application logic, with intermediate layers like context management layers.

Many works are dedicated to the modeling of context and its impact on the development of context-aware applications. A famous example is the "Context Toolkit" [DAS01] where three layers of components are required to capture the context: *widgets* that acquire low-level information from sensors, *interpreters* abstracting this information and *aggregators* gathering information by entity. A common representation for the context is also a requirement to enable interoperability. Whereas simple forms of context can be expressed using key-value pairs (e.g., `[name="carla", location="elysee"]`), more elaborate context models need graph-model representation like RDF (Resource Description Framework) or the more general concept of ontology (e.g., the Context Ontology Language (CoOL) [SLp03]). Ontologies enable independent components to reason about the same concepts with a shared ontology or to agree about concepts with ontology alignments.

However, a formal modeling of pervasive environments as a whole, i.e. entities and behaviors, has not been often addressed. In [RC08], such a modeling is proposed using ambients: an ambient is a "bounded place where computation happens" (e.g., a web page, a laptop) that can be nested. Ambient calculus is used to describe entities (devices, services, users), operations that can be performed and events that can occur. Ambient logic is used to describe the properties of the pervasive environment. Furthermore, they propose a method to prove that some properties are verified (or not) in a given environment, like application mobility or automatic service discovery. It however does not handle time aspects.

In the SoCQ framework, pervasive environment are modeled using a data model. The structure of the data model enables the representation of entities as distributed services and data sources, whereas the language enables the description of the required behaviors using declarative queries. Context representation is not tackled directly: we consider that context data are provided by sensor services, and higher-level context information can be computed by queries. We have focused on the definition of an algebra and on query equivalence in this setting. Logic-based languages could also be defined, in order to compare the expressiveness of our data model with other models using ambient calculus and logic.

5.2 Related database research

5.2.1 Data streams

Data stream processing

In the last years, many projects have been launched on data streams and continuous query processing. One of the first project, namely NiagaraCQ [CDTW00], introduces some definitions of continuous queries over XML data streams. Queries, expressed using XML-QL, can be defined as triggers and produce event notifications in real-time,

or be timer-based and produce periodically their result at a given time interval. Incremental evaluation is used to reduce the required amount of computation: queries are evaluated only for changed data. Furthermore, queries are grouped by similarity in order to share a maximum of computation between queries. Interestingly, they introduce the notion of "action" to handle query results. Although an action may be any user-defined function, it is used in the examples to specify how to send notification messages to users, e.g., with a "MailTo" action.

Other data stream processing systems tend to use a SQL-like language to express continuous queries. The TelegraphCQ system [CCD+03] proposes adaptive continuous query processing over streaming data and historical data. Data and queries are treated symmetrically: "new queries [are] applied to old data and new data [are] applied to old queries". Adaptive group query optimization is realized by a dynamic routing of tuples among (commutative) query operators depending on operator load. Cougar [YG03, BGS01] and TinyDB [GM04] handle continuous queries over sensor networks with a focus on the optimization of energy consumption for sensors using in-network query processing. Each sensor having some query processing capabilities, distributed execution plans can use progressive data aggregation to reduce the amount of raw data transmitted through the network.

STREAM [ABB+03] defines a homogeneous framework for continuous queries over relations and data streams. They claim that simply reusing a relational query language with streams instead of relations is not sufficient for queries that are not simple. Formal abstract semantics are defined, using a "discrete, ordered time domain": time is explicitly represented. Relations are time-varying bags (multisets) of tuples, and streams are unbounded bags of timestamped tuples. Three categories of operators are clearly identified: relation-to-relation (standard operators: selection, projection, join...), relation-to-stream (insert/delete/relation stream), and stream-to-relation (windows). SQL is little extended to build CQL (Continuous Query Language) that enables the declarative definition of continuous queries. They further focus on computation sharing to optimize the parallel execution of several continuous query.

A different approach is tackled with Aurora/Borealis [HXCZ07, AAB+05, CBB+03]: a "Distributed Stream Processing System" (DSPS) enables the definition of dataflow graphs of operators in a "box & arrows" fashion. It makes distributed query processing easier: boxes are computation units that can be distributed, and arrows represent tuple flows between boxes. Adaptive query optimization is made by a load-balancing mechanism. Query optimization is however limited with this procedural approach, compared to declarative approaches that use SQL-like languages, as query rewriting is not possible.

Data streams can also represent streams of events. Complex Event Processing (CEP) or Event Stream Processing (ESP) techniques allow to express specialized continuous queries that detect complex events from input event streams. For example, Cayuga [DGP+07, DGH+06] is a stateful publish/subscribe system for complex event

monitoring where events are defined by SQL-like continuous queries over data streams. It is based on formally-defined algebra operators that are translated to Non-Finite Automatons for physical processing.

In this thesis, we use a representation of dynamic data sources that is close to STREAM [ABB+03], with an homogeneous representation for relations and data streams, and a discrete time domain. It enables clear semantics with regard to time aspects. We however go further and define a whole data model with its structure and language. We nevertheless do not focus on the same issues: the SoCQ data model enables the representation of, and queries over, a pervasive environment with dynamic data sources and functionalities, whereas STREAM focuses on optimizing continuous query execution over data streams. Furthermore, the integration of functionalities requires the redefinition of one-shot queries, whereas in STREAM, one-shot queries would simply be standard SQL queries over instantaneous relations. With regard to event management, an event algebra like in Cayuga [DGH+06] might be included in the streaming operator of the Serena algebra in order to enhance its expressiveness.

Continuous query processing issues

For continuous query processing, unbounded tuple streams potentially require unbounded memory space in order to be joined, as every tuple should be stored to be compared with every tuple from the other stream. Tuple sets should then be bounded: a window defines a bounded subset of tuples from a stream (it is the only stream-to-relation operator in the STREAM framework [ABB+03]), based on time or on the number of tuples. Sliding windows [ABB+03, DR04] have a fixed size and continuously move forward (e.g., the last 100 tuples, tuples within the last 5 minutes). Hopping windows [YG03] have a fixed size and move by hop, defining a range of interval (e.g., 5-minute window every 5 minutes). In [CCD+03], windows can be defined in a flexible way: the window upper and lower bound are defined separately (fixed, sliding or hopping), allowing various type of windows. [ABB+03] also defines a partitioned window as the union of windows over a partitioned stream based on attribute values (e.g., the last 5 tuples for every different ID). With windows, join operators handle bounded sets of tuples and traditional techniques can be applied. Although the output is intuitively thought as a stream, join operators are seen in [ABB+03] as relation-to-relation operators: the output is a time-varying relation.

A similar problem exists for aggregation operators over streams: they need to have a complete view of all tuples (e.g., the COUNT operator in SQL), which is impossible for unbounded streams. A mechanism of punctuations [DR04], indicating the end of a group of related tuples, can be used to enable the aggregation operator to output its resulting aggregated tuples, thus creating an aggregated stream. In [ABB+03], aggregation operators are seen as relation-to-relation operators: input streams should be windowed, and the output is a time-varying relation.

In the Serena algebra, we take a similar approach to the STREAM framework [ABB+03]

for windows. We however only define time-based sliding windows. Aggregation operators are not yet integrated in the algebra, although it would enhance the expressiveness of queries.

Data-oriented pervasive applications

Continuous queries can be used to define in a declarative way some parts of pervasive applications that involve only data processing. In [FJK+05, JAF+06], the progressive cleaning process for data retrieved from numerous physical sensors is defined by a pipeline of continuous queries declaratively defined in SQL. A complex event processing using state-machine operators producing data streams is also proposed.

In [AHS06, AHS07], the Global Sensor Network is a middleware for sensor networks that provides continuous query processing facilities over distributed data streams. It enables to specify continuous queries as virtual sensors whose processing is specified declaratively in SQL, with a subquery for preprocessing each input stream. Virtual sensors hide implementation details of data sources and homogeneously represent data streams provided by physical sensors and data streams that are produced by continuous queries over other virtual sensors.

With SoCQ queries, pervasive applications like cleaning sensor data or queries over virtual sensors are clearly handled, as it involves continuous queries over data streams provided by services.

5.2.2 Data and service integration

Dataspaces

Data integration has been a long standing theme of research over the past 30 years. Now, the broader notion of *dataspace* has appeared in order to homogeneously handle a large number of heterogeneous data sources. In [FHM05], they define an agenda toward this high-level goal. Existing data management solutions range from highly controlled and semantically integrated solutions with DBMSs, to the opposite with web search. A DataSpace Support Platform will enable different query interfaces, including queries over streaming data, and, using a discovery mechanism, it will be able to dynamically locate and integrate data sources. Due to heterogeneous administrative control, a best effort strategy is required. For example, to answer a query when some data sources are unavailable, the data accessible at the time of the query have to be used to propose the best possible results.

Data sources can also be associated with physical location information to build a dataspace that is coupled with the physical world [IN02]. Location-dependent and location-independent information is constantly produced by a great number of devices and users. Information is stored where it is produced and queries are “beamed” in the physical space to retrieve required information.

Dataspaces remain high-level goals for future information systems. The SoCQ data model also aims at integrating distributed data sources from a dynamic environment and providing one-shot and continuous query facilities. Although it does not focus on data source integration at the semantic level, it can become part of the query level of a dataspace. Adding physical location information to data sources is also an interesting issue in order to build location-aware behaviors in pervasive environments.

Binding patterns

In the setting of data integration, the notion of *binding patterns* appears to be quite interesting since they allow to model a restricted access pattern to a relational data source as a specification of "which attributes of a relation must be given values when accessing a set of tuples" [FLMS99]. In [FLMS99], a relation with binding patterns can represent an external data source with limited access patterns. It makes explicit the "mismatch between the logical and physical views of the data". Whereas traditional query optimization assume a complete scan of a relation is always possible, they extend optimization techniques to take into account that only binding patterns can be used to retrieve tuples. Query execution (sub)plans are annotated with input attributes (or "bound" attributes, in opposition to "free" attributes) that are required for their execution because of the involved binding patterns, and query plan equivalence is redefined accordingly.

Binding patterns can be combined with the notion of virtual table. In general, a virtual table represents a relational table whose content is dynamically generated at query execution time, typically by a call to an external function or wrapper. In [GW00], a virtual table with a binding pattern can represent the interface to an infinite data source, e.g., a web site search engine providing a list of URLs corresponding to some given keywords. It enables the integration of web services within traditional databases: queries can combine data from regular relations with invocations of services. In their setting, query optimization focuses on an asynchronous invocation system ("asynchronous iteration") that aims at handling invocation latency through parallelization by decoupling the invocations of services and the retrieving of results. In [SMWM06], virtual tables with binding patterns can represent more general data services, e.g., web services providing data sets. They aims at "providing DBMS-like capabilities when data sources are web services". SQL-like queries are optimized into pipelined execution plans of web service calls where the output data produced by a service are the input data of the next service. Pipelines are in fact directed acyclic graphs due to precedence constraints between services. In the setting of large data sets that need to flow into the pipeline, they propose a dedicated cost metric, namely the "bottleneck cost metric", that represents the throughput of the slowest web service.

In a similar way to binding patterns, the ActiveXML language [Act] allows to define XML documents containing extensional data, i.e. data that are present in the document, and intensional data, representing service calls that provide data when needed. Intensional data is close to the notion of virtual tables and binding patterns, transposed to the XML technology. ActiveXML is also a "framework for distributed XML data manage-

ment" [AMT06]: it defines an algebra to model operations over ActiveXML documents distributed among peers, and query optimization techniques are used to minimize the need for materializing data on peers.

Binding patterns are a key notion for the integration of external data sources at the logical level. In the SoCQ data model, we extend this notion to integrate external functionalities within relation schema. We however do not use the notion of virtual table, but work at a finer grain. Whereas virtual tables represent external data sources, we represent those data sources as services at the tuple level: several data sources are represented by individual tuples in the same table, or XD-Relation, whose schema contains real attributes and virtual attributes. The SoCQ data model enables very simple queries to involve many data sources. Furthermore, binding patterns are extended to also represent data streams provided by services in addition to invocation of service methods. Query optimization techniques involving binding patterns proposed in the literature are nevertheless interesting to be integrated into the SoCQ data model.

External functions

The SQL standard itself supports some forms of access to external functionalities through User-Defined Functions (UDF). UDFs can be scalar functions (returning a single value) or table functions (returning a relation). UDFs are defined in SQL or in another programming language (e.g., C, Java), enabling to access any external resources. Table functions are a way to implement the notion of virtual tables, however limited to having only one binding pattern determined by the function input parameters. UDFs are also tagged as deterministic or non-deterministic: query rewriting may not change the number of invocations for non-deterministic UDFs.

In [BGS01], Abstract Data Types (ADT) are used to build an object-oriented view of external functionalities. Access to device functionalities (or "signal-processing functions") is modeled as ADT scalar functions. A dedicated relational operator, the so-called virtual join associated with a virtual scan technique, plays a role of buffer between repeated asynchronous calls to those functions and continuous query execution. It enables to use SQL queries with little modifications, but one of the drawback of this approach is the difficulty to precisely handle time (e.g., the ordering of sensor data, time windows). In [GM04], all sensors from a sensor network are supposed to provide the same measures (e.g., light, temperature). They are represented by an infinite virtual table `sensors`, with an attribute for the sensor identifier and one attribute by measured value, that conceptually contains all measures for every possible time instant. SQL queries over this virtual table enable to implicitly retrieve (aggregated) measured values from sensors.

In the SoCQ data model, external functionalities are represented by methods and data stream provided by services. They are however not represented as external functions, but by binding patterns at the metadata level. Contrary to previous approaches, those functionalities are not implicit like with a global virtual table, assuming that every

services has the same interface, but nevertheless do not require a specific ADT for each kind of devices. Services are first-class citizens of the data model, and are seamlessly integrated into XD-Relations. At the query level, interactions with services are explicitly defined by binding operators. It furthermore enables the expression of dynamic service discovery in queries.

Optimization with external functions

Optimization of queries involving expensive functions or methods leads to the redefinition of cost models to integrate the estimated cost of computation. This issue has been studied for standard databases [CS93, CS99, Hel98, HS93], and more recently also for continuous query processing [DF06]. In [Hel98], the issue of expensive methods in object-relational database management systems (ORDBMS) is addressed, where new data types can be defined and associated with new methods. In particular for Geographic Information Systems (GIS), objects can be large and their methods can be very expensive. They propose a "predicate migration" framework that takes into account a balance between the selectivity and the cost per tuple of methods used in selection and join predicates when optimizing a query plan. In [CS99], they consider the similar case of expensive user-defined predicates (i.e. UDFs that are used as predicates) in relational DBMSs. They propose an approach that is guaranteed to produce an optimal plan and address some limitations of previous works. In the context of continuous queries over data streams, where execution time issues meet real-time considerations, a "trade-off between work and accuracy" [DF06] may be required for expensive functions that need to be repeatedly called. In [DF06], they propose Variable Accuracy Operators (VAOs) that enable the choice of a sufficient precision level of function results with regard to a given query. In [DF05], they propose the CASPER system (CAching System for PrEdicate Result ranges), a caching mechanism that aggregates previously computed function results by ranges of input values. Cached values can be reused to avoid unnecessary computation of expensive functions. Finally, as summarized in [DF06], continuous query optimization can use three techniques to handle expensive functions: predicate re-ordering (like for one-shot queries), caching, and precision reduction.

In Aorta [XL05], expensive functions are considered in the setting of pervasive computing. SQL-like continuous queries can implicitly interact with devices through external function calls, called "actions". An action is tied to one physical implementation that takes at least one input parameter that identifies the device (e.g., an IP address for a network camera, a phone number for a smart phone). Their "action-oriented query processing" introduces a selection among available devices that offer the same action based on their current state, in order to choose the optimal way of evaluating a function. For example, the query engine chooses the network camera that requires the minimum movement to focus on a given location and take a photo. They propose a cost-model dedicated to actions to enable dynamic query optimization: an action is decomposed in several atomic operations with an associated cost, and a global cost can be evaluated given the device current state and the target goal. Furthermore, a group optimization

allows to optimally distribute simultaneous actions among the possible candidate devices. However, the relationship between functions and devices are not explicit and the optimization criteria cannot be declaratively defined in queries.

In the SoCQ data model, binding patterns are similar to expensive external functions. Although virtual attributes and binding patterns pose additional constraints in the query operator ordering, cost models and optimization techniques proposed in the context of (object-)relational DBMSs could be adapted. In our cost model, the distributed service invocation/subscription cost is however considered separately from the centralized query execution cost, whereas in the settings of DBMSs, the cost of expensive functions is integrated into a single query execution cost value. In the setting of pervasive environment, the proposition of the Aorta system [XL05] has inspired the beginning of this thesis. In particular, we have adapted the "Temperature Surveillance" scenario from the scenario used in this work: it originally involves network cameras and movement sensors in order to send alert messages by SMS. The SoCQ data model enables the representation of such scenarios, but is also suitable for scenarios involving many kinds of services and data sources that can be seamlessly integrated into a PEMS. Integrating a cost model describing the service cost per invocation as proposed in [XL05] would lead to a more detailed cost model for SoCQ queries, but this level of precision has not been tackled yet.

5.3 Enabling technologies

Object Request Brokers

CORBA [OMG] (Common Object Request Broker Architecture) is an open architecture and infrastructure that enables applications to interoperate over network links. It can be defined as an object bus: applications can access local or remote objects without worrying about underlying network issues (including serialization issues). A lookup allows to search objects by name and get object references. Objects are defined using the platform-independent IDL (Interface Description Language) that can be used to generate stub and/or skeleton in many programming languages.

Some systems tackle the same issues, but are more platform- or language-dependent, like Microsoft DCOM (Distributed Component Object Model) or Java RMI (Remote Method Invocation) along with the Jini technology (now Apache River).

Messaging protocols

In contrast with Object Request Brokers, most of more recent systems focus on a messaging protocol between services to achieve interoperability. Those protocols are more data-oriented. The open standard XML (eXtended Markup Language) is often used as the message format for such protocols, like for the simple yet efficient XML-RPC (XML - Remote Procedure Call) or its more complex but powerful evolution SOAP (Simple

Object Access Protocol) for Web Services. REST (REpresentational State Transfer) relies on the HTTP API to transfer messages, but does not define a message format: it is rather an architecture style using the well-established HTTP protocol to simplify communications. For those protocols, service discovery needs to be done by external registries, like UDDI (Universal Description, Discovery and Integration) for Web Services.

UPnP [UPn] (Universal Plug and Play) is also based on messaging protocols (using SOAP) and includes automatic discovery mechanisms (SSDP), using network multicast facilities. UPnP devices represent physical or logical devices that host several UPnP services providing methods and publishing events. The UPnP technology is supported by a great numbers of device manufacturers, the so-called UPnP Alliance, and is now widely available (for example, it is handled by recent Microsoft Windows OSs), although not very widely used yet. A recent trend, in particular with DPWS [SOD, SOA] (Devices Profile for Web Services), aims at a convergence between the UPnP technology and Web Service standards. With the recently standardized UPnP ("International Standard for Device Interoperability for IP-based Network Devices" (ISO/IEC 29341)) in December 2008, a step toward this convergence was done. Besides, DPWS was also accepted as an international open standard by the OASIS consortium in July 2009 (more precisely, the name of the standard is "Web Services Discovery and Web Services Devices Profile" (WS-DD), and includes "Web Services Dynamic Discovery" (WS-Discovery), "SOAP Over User Datagram Protocol" (UDP), and "Devices Profile for Web Services" (DPWS)).

Frameworks

OSGi [OSG], and the older JMX [JMX] (Java Management eXtension), are two Java frameworks that can host some (potentially active) java objects as services and enable a local and remote access to them. In particular, OSGi seamlessly enables the use of various network protocols like RMI (by reusing JMX), Web Services, or even UPnP and DPWS if dynamic discovery is needed. It is a set of specifications that defines a modular architecture for applications. It has been implemented in several projects (e.g., the open source implementation Apache Felix [Fel]). Bundles, i.e. OSGi modules, can be deployed on any specification-compliant implementations. Initially designed for home gateway devices, OSGi implementations can be executed on many devices with different capabilities, from small embedded devices to big servers. Note that a current trend is to redesign Java-based application servers using OSGi specifications to benefit from the well-defined modular architecture. For example, the open source application server GlassFish [Gla] uses, from its recent version 3, the OSGi implementation Apache Felix [Fel].

Nowadays, these relatively recent technologies have become mature, some of them being supported by the industry (in particular for application servers). Focused on interoperability issues in a heterogeneous setting, they can be re-used in the context of

pervasive environment management systems. For the development of the SoCQ PEMS prototype, we have chosen the OSGi framework along with the UPnP technology that, on the one hand, simplify the development of the PEMS itself, and on the other hand, enable an easy development and integration of distributed services providing methods and data streams.

6
Conclusion

Chapter Outline

6.1 Summary of contributions **124**
6.1.1 Modeling pervasive environments as relational pervasive environments 124
6.1.2 Developing pervasive applications through declarative queries 125
6.1.3 Building a pervasive environment management system 126
6.2 Discussion and perspectives **126**
6.2.1 Foundations of the data model 126
6.2.2 Extending database design principles to pervasive environments 127
6.2.3 Optimization, Evaluation & Benchmark 128
6.2.4 Peer-to-peer PEMS 128
6.3 Final words **129**

Pervasive environments leverage existing issues for distributed systems and pose new challenges that need to be addressed in order to benefit from their full potential: complex, but manageable, interactions between heterogeneous devices that provide dynamic data sources and functionalities. Two axes of studies have been developed in the literature: on the one hand, studies about middlewares and service-oriented architectures, and on the other hand studies about data management with dataspaces or data streams. In this thesis, we propose to address pervasive environment issues by combining those two approaches in a single data model.

We first summarize our contributions in Section 6.1. We then discuss the choices of our approach and remaining open issues in Section 6.2: this discussion lead us to tackle some perspectives for future works.

6.1 Summary of contributions

The main goal of this thesis is to simplify the development of pervasive applications, i.e. applications involving distributed functionalities and dynamic data sources. We have proposed a declarative approach through a data model dedicated for pervasive environments, namely the SoCQ data model (SoCQ standing for **S**ervice-**o**riented **C**ontinuous **Q**uery). The standard notion of database is extended to define the notion of relational pervasive environment integrating heterogeneous and dynamic data sources: databases, data streams and functionalities provided by distributed devices. Pervasive applications can be declaratively expressed as one-shot and continuous queries. We have also implemented the data model in a Pervasive Environment Management System (PEMS) to demonstrate the feasibility and the expressiveness of our approach.

6.1.1 Modeling pervasive environments as relational pervasive environments

The SoCQ data model is based on three fundamental notions: time, data, and functionality. Time is represented by a discrete time domain of time instants. Data come from the relational model including constants and attributes. Functionality prototypes and services are part of our proposition to represent distributed functionalities of pervasive environments. Prototypes represent a data-oriented declaration of functionalities, whereas services represent implementation of those functionalities. Prototypes are either invocation or subscription prototypes, corresponding to invocation of service methods or to the subscription to service data streams; and are either active or passive, depending whether their impact on the environment is significant or not.

By combining time, data and functionalities, we have defined the notion of XD-Relation that can represent standard relations, data streams and distributed functionalities. XD-Relations are either finite or infinite: infinite XD-relations represent infinite append-only relations like data streams, whereas finite XD-Relations represent finite re-

lations (that can evolve with time as usual). Functionality prototypes are integrated at the metadata level into relation schema through the key notions of *virtual attributes* and *binding patterns*. Service references are integrated at the data level into tuples, allowing a great dynamicity in service management at runtime, which is a requirement for applications in pervasive environments. Binding patterns represent the link between the data-oriented representation of the pervasive environment and actual invocations of, or subscriptions to, distributed device functionalities, where virtual attributes are placeholders for unbound input and output parameters.

6.1.2 Developing pervasive applications through declarative queries

The development of pervasive applications consists of building interactions between data sources and functionalities. Even with existing tools like middlewares, high-level programming skills are still required in order to build such applications. Through declarative queries over relational pervasive environments, we have simplified their development such that basic SQL knowledge is required to deploy sophisticated applications. Two types of queries are possible: one-shot queries (like standard SQL queries), for one-time interactions, and continuous queries (like queries over data streams), for interactions that last in time.

We have defined a query language for our data model, namely the Serena algebra (standing for **Ser**vice-**ena**bled algebra). The Serena algebra for one-shot queries over X-Relations is composed of extensions of set and relational operators: union, intersection, difference, projection, selection, renaming, natural join; of realization operators that handle virtual attributes and binding patterns: assignment and binding; and of a service discovery operator. The Serena algebra for continuous queries is composed of extensions of one-shot operators over finite XD-Relations; and of operators dedicated to streams: window and streaming. Queries are associated with their start instant that represents the instant they are launched.

We have also defined the crucial notion of query equivalence for one-shot and continuous queries. Besides the equality at the data level, two queries are equivalent if they lead to the same action set: the action set represents all invocations of, and subscriptions to, active prototypes, without considering passive prototypes. Some rewriting rules for algebra expression have been proposed. Query equivalence enables logical query optimization: a simple cost model has been proposed, along with some hints for rule-based query optimization.

Finally, we have defined a SQL-like query language to simplify the expression of Serena queries, namely the Serena SQL. It enables the definition of one-shot and continuous queries over XD-relations, as well as service discovery queries.

6.1.3 Building a pervasive environment management system

We have proposed and implemented an architecture that supports the SoCQ data model. It can be built on existing distributed middleware architecture: we have used the Universal Plug and Play (UPnP) technology for service discovery and remote interaction issues. We have implemented a service-enabled dynamic data management system, namely a Pervasive Environment Management System (PEMS), using the Java OSGi framework. This SoCQ PEMS manages XD-Relations and enables the execution of one-shot and continuous queries. Using parsers, it interprets the Serena DDL, DML and SQL. We also have implemented easy to use GUIs that simplify the handling of the PEMS for application developers.

We have experimented two scenarios with the SoCQ PEMS: the "Temperature Surveillance" scenario and the "RSS feeds" scenario. Those scenarios show that the SoCQ PEMS, built on the SoCQ data model, can handle and combine different types of data sources and functionalities: conventional data, temperature sensors, messaging services, RSS feed services, *etc.* Queries from the scenarios also illustrate the expressiveness of the query language.

6.2 Discussion and perspectives

We now discuss some of the choices we have made for our approach. We also point out some remaining open issues and perspectives for future works.

6.2.1 Foundations of the data model

The SoCQ data model has been defined using set semantics for X-Relations and XD-Relations, although using the multiset semantics would enable a slightly more seamless integration of continuous aspects, in particular for data streams. Integrating multiset semantics in the SoCQ data model, in a similar way to STREAM [ABB+03], is a short-term objective.

In the SoCQ data model, the proposed data-oriented representation for services is expressive enough for a large number of different services. It may however not be adapted to all cases, in particular with regard to the widely used XML Web Services and the associated technologies (denoted `WS-*` in the literature) that cover many issues like security, transactions, *etc.* The expressiveness of such technologies are nevertheless more complex to handle in a declarative data model. We can also remark that the notion of data streams provided by services is not yet fully integrated in those technologies. A detailed comparison of the approach taken by Web Service technologies and our proposed data model is still lacking and would point out some potential gap between Web Services and the requirements of pervasive environments. It may also lead to integrate a wider variety of services in the SoCQ data model.

SoCQ services can provide methods and parameterized data streams (infinite dynamic relations). We could also consider services that provide parameterized relations (finite dynamic relations). The notion of subscription binding pattern could be extended to, on the one hand, "streaming" subscription binding patterns for infinite dynamic relations, and on the other hand, "updating" subscription binding patterns for finite dynamic relations.

Queries expressed in the Serena algebra enable the definition of complex and dynamic interactions between distributed devices and data sources. The expressiveness of those queries may however not be sufficient to detect complex events, with regard to existing dedicated complex event processing techniques [DGP+07]. Precise event correlations over time can not be easily expressed with only extensions of relational operators over data streams. A combination of data stream processing, complex event detection and interactions with services opens interesting issues to be addressed.

We furthermore aim at integrating an aggregation operator that we think would greatly enhance the expressiveness of the Serena algebra (and therefore the Serena SQL) and enable more subtle and complex scenarios. For example, considering context management [DAS01], context needs to be captured through acquisition, interpretation and aggregation. Acquisition from sensors can be handled at the service level and interpretation can be expressed with SoCQ queries. Context aggregation could also be handled by SoCQ queries, but with an additional aggregation operator similar to the GROUP BY clause in SQL.

In the setting of pervasive environments, devising logical formalisms and logic-based query languages is a difficult issue that opens interesting perspectives for the declarative definition of applications (e.g., the Netquest project [BBG+09], ambient calculus and logic [RC08]). We aim at comparing the expressiveness of such models with the SoCQ data model. We also aim at exploring the definition of logical query languages within the SoCQ data model itself.

6.2.2 Extending database design principles to pervasive environments

With the SoCQ data model, we have extended database principles to meet the requirements of pervasive environments. The strength of the SoCQ data model for a given application comes from the expressiveness of the designed XD-Relations, with their virtual attributes and binding patterns, so that declarative queries over those XD-Relations can build suitable interactions.

Devising design principles in this context is a quite important issue. An interesting idea to address this issue is to extend *database design principles* to pervasive environments. Existing models like the entity/relationship model could be leveraged to handle dynamic relations, virtual attributes and binding patterns. Constraints need to be redefined in this context. This issue could lead to the definition of a methodology dedicated to the development of pervasive applications.

6.2.3 Optimization, Evaluation & Benchmark

In the SoCQ data model, query equivalence has been defined using a key distinction between active and passive prototypes and their impact on the environment. A dedicated cost model has been proposed, but actual optimization techniques have not yet been fully defined for service-oriented continuous queries. This is a natural continuation of our work.

One related perspective is the evaluation of SoCQ query execution. Issues such as QoS (Quality of Service), service replaceability, interaction latency, or missing answers are interesting to be tackled in the context of pervasive environments. In particular, the definition and execution of pervasive applications through SoCQ queries can be compared to other application development techniques (conventional *ad hoc* development, service composition, *etc.*).

We furthermore aim at developing an actual benchmark dedicated to pervasive environments. It would allow to measure the performance of pervasive applications, including SoCQ queries, with objective indicators. This benchmark is part of a French National Research Agency (ANR) project called OPTIMACS, started in December 2008 in collaboration with the HADAS team of the LIG laboratory (Grenoble, France) and the ROI team of the LAMIH laboratory (Valenciennes, France).

6.2.4 Peer-to-peer PEMS

As presented through the proposed PEMS architecture, the execution model for queries is centralized in the PEMS Core modules, although it involves interactions with distributed services. Distributed query processing techniques would however be more suitable in the context of pervasive environments, where some individual devices may have limited computation resources but are numerous in the environment. Devising a peer-to-peer architecture for PEMS is a more long-term perspective, with an execution model adapted for distributed computation, a dedicated cost model and adaptive query optimization like load balancing.

6.3 Final words

To conclude, working on this thesis has been a great adventure. From the "trendy" topic of mobile devices and pervasive/ubiquitous computing, to the depth and severity of the foundations of the relational model, a large panel of issues needed to be tackled. The theoretical part of this work has led to the definition of a data model with its structure and language, and the practical part is materialized by a working prototype that actually executes one-shot and continuous queries over a relational pervasive environment. From this point, many interesting research perspectives with their own new challenges can be followed, and the SoCQ framework can now be considered for the design and implementation of real-world applications.

A

Résumé long en français

Une approche déclarative pour les environnements pervasifs: modèle et implémentation

Yann Gripay*

* *Université de Lyon, CNRS*
INSA-Lyon, LIRIS, UMR5205
7 avenue Jean Capelle, F-69621, Villeurbanne, France
yann.gripay@liris.cnrs.fr

RÉSUMÉ. *Interroger des sources de données non-conventionnelles est reconnu comme une problématique majeure dans les nouveaux environnements comme ceux de l'informatique pervasive. Un point clé est la possibilité d'interroger des données, des flux de données et des services de manière déclarative afin de faciliter le développement d'applications pervasives. Dans cet article, nous définissons une vue orientée données des environnements pervasifs: la notion classique de base de données est étendue pour construire une notion plus large, l'environnement pervasif relationnel. Nous définissons également une algèbre, appelée algèbre Séréna, qui permet l'expression de requêtes continues ou ponctuelles traitant de manière homogène les données, flux de données et services. Un prototype de Système de Gestion d'Environnement Pervasif a été implémenté et des expérimentations ont été réalisées afin de valider notre approche.*

ABSTRACT. *Querying non-conventional data is recognized as a major issue in new environments and applications such as those occurring in pervasive computing. A key issue is the ability to query data, streams and services in a declarative way in order to make the development of pervasive applications easier. In this article, we define a data-centric view of pervasive environments: the classical notion of database is extended to come up with a broader notion, the relational pervasive environment. We then define the so-called Serena algebra that enables the expression of one-shot or continuous queries homogeneously handling data, streams and services. A prototype of Pervasive Environment Management System has been implemented and experiments have been conducted in order to validate our approach.*

MOTS-CLÉS : *bases de données, flux de données, services, requêtes continues, algèbre, optimisation, environnements pervasifs*

KEYWORDS: *databases, data streams, services, continuous queries, algebra, optimization, pervasive environments*

1. Introduction

1.1. *Contexte*

Les environnements informatiques évoluent vers ce qu'on appelle des systèmes pervasifs : ils ont tendance à être de plus en plus hétérogènes, décentralisés et autonomes. D'une part, les ordinateurs personnels et autres terminaux mobiles sont largement répandus et occupent une grande place dans les systèmes d'information. D'autre part, les sources de données et fonctionnalités disponibles peuvent être réparties sur de larges espaces grâce à des réseaux allant du réseau mondial Internet jusqu'aux réseaux locaux pair-à-pair pour les capteurs. Elles sont de plus dynamiques et hétérogènes : bases de données avec des mises à jour fréquentes, flux de données provenant de capteurs logiques ou physiques, et services fournissant des données stockées ou provenant de capteurs, transformant des données ou commandant des actionneurs.

Les environnements pervasifs posent de nouveaux défis pour exploiter leur plein potentiel, en particulier la gestion d'interactions complexes entre ressources réparties. Il est cependant difficile de gérer ces sources de données et fonctionnalités hétérogènes avec les systèmes actuels, ce qui constitue un frein pour le développement d'applications pervasives. Il est ainsi nécessaire de combiner au sein de développements *ad hoc* des langages de programmation impératifs (C++, Java...), des langages de requêtes classiques pour les bases de données (SQL...) et des protocoles réseau (JMX, UPnP...). Ce n'est cependant une solution ni pratique ni adéquate sur le long terme.

Les approches déclaratives offrent l'avantage de fournir une vue logique des ressources qui abstrait les problématiques d'accès physique et permet la mise en œuvre de techniques d'optimisation. Les requêtes SQL sur les bases de données relationnelles en sont une illustration typique et bien connue. C'est pourquoi la définition déclarative de requêtes sur des sources de données et des fonctionnalités est reconnue comme un défi majeur dans le but de simplifier le développement d'applications pervasives.

Actuellement, les extensions des SGBDs (Système de Gestion de Bases de Données) permettent d'avoir une vue homogène et d'effectuer des requêtes sur des bases de données et des flux de données (notamment les SGFDs, Système de Gestion de Flux de Données). La notion de service est un moyen courant de représenter les fonctionnalités réparties d'un système informatique, mais n'est pas encore pleinement intégrée au sein des SGBDs. Malgré de nombreuses propositions, une compréhension claire des interactions entre données, flux de données et services manque toujours, ce qui constitue un frein majeur pour la définition déclarative des applications pervasives, en lieu et place des actuels développements *ad hoc*.

1.2. *Contribution*

Dans cet article, nous proposons un framework définissant une vue orientée données des environnements pervasifs : la notion classique de base de données est étendue pour construire une notion plus large, l'environnement pervasif relationnel, qui

intègre les sources de données à la fois conventionnelles et non-conventionnelles, à savoir données, flux de données et services. Cette notion permet le développement d'applications pervasives de manière déclarative en utilisant des requêtes continues orientées service qui combinent ces sources de données.

Dans ce framework, nous proposons un modèle de données pour les environnement pervasifs, appelé SoCQ (pour Service-oriented Continuous Query), qui prend en compte leur hétérogénéité, dynamicité et répartition. Nous définissons la *structure* de notre modèle de données avec la notion de relation dynamique étendue (*eXtended Dynamic Relation*, ou XD-Relation) représentant les sources de données. Nous définissons également un *langage* algébrique pour notre modèle de données avec l'algèbre Séréna (*Service-enabled algebra*), à partir de laquelle un langage de type SQL a été défini. Ce langage permet d'exprimer de manière déclarative des requêtes sur les environnements pervasifs.

Afin d'implémenter ce framework, nous avons conçu une architecture de système de gestion d'environnements pervasifs (Pervasive Environment Management System, ou PEMS) qui prend en charge notre modèle de données. Un PEMS est un système de gestion dynamique de données et de services qui gère de manière transparente les problématiques liées au réseau telles que la découverte de services et les interactions à distance. Il supporte l'exécution de requêtes ponctuelles et continues orientées service que les développeurs d'applications peuvent aisément concevoir pour développer des applications pervasives. Un prototype de PEMS a été implémenté, avec lequel des expérimentations ont été réalisées.

1.3. *Scénario*

Pour les exemples de cet article, nous considérons le scénario suivant. Un bâtiment "intelligent" comporte différentes zones (pièces, toit, *etc.*) qui contiennent des capteurs de températures et des caméras permettant de prendre des photos. L'objectif est de surveiller la température des zones en fonction de seuils d'alerte stockés dans une base de données. En cas d'alerte, un message spécifique accompagné d'une photo est envoyé au responsable de la zone (par e-mail, SMS ou messagerie instantanée). L'intégration de nouveaux capteurs de température doit être prise en compte dynamiquement, de même que la désactivation de certains capteurs.

1.4. *Organisation de l'article*

La suite de cet article est organisée de la manière suivante. La section 2 porte sur la structure du modèle de données SoCQ, en particulier sur la notion clé de XD-Relation. La section 3 porte sur le langage de ce modèle de données à travers la définition de l'algèbre Séréna. La section 4 présente l'implémentation du PEMS, ainsi que les expérimentations qui ont été réalisées avec le prototype. Enfin, la section 5 conclut et discute des perspectives.

2. Modélisation des environnements pervasifs

Afin de simplifier le développement d'applications dans les environnements pervasifs, un première étape est la modélisation de ces environnements. Nous avons choisi de construire une vue orientée données de ces environnements en utilisant les principes des bases de données, ce qui aboutit à la définition de la structure d'un modèle de données. Notre modèle de données repose sur l'idée clé de décrire les fonctionnalités de l'environnement (messageries, capteurs de température, *etc.*) comme des sources de données réparties (et paramétrables), afin de les intégrer sans peine avec les sources de données plus conventionnelles que sont les données relationnelles et les flux de données. Cela permet l'étape suivante, sujet de la section 3, qui est d'exprimer déclarativement des interactions entre les données et les fonctionnalités, c'est-à-dire des parties d'applications pervasives, par des langages de requêtes proche de ceux du modèle relationnel.

Dans cette section, nous posons tout d'abord nos notations pour la représentation du temps, des données et des flux de données. Nous proposons ensuite une modélisation des fonctionnalités réparties de l'environnement, avant de définir la notion d'*environnement pervasif relationnel* étendant la notion classique de base de données.

2.1. *Préliminaires*

Les notations de notre modèle de données se basent sur celles du modèle relationnel, en particulier sur les notations de (Levene *et al.*, 1999). La représentation du temps s'inspire quant à elle des notations de (Arasu *et al.*, 2003).

Afin de prendre en compte les aspects continus des environnements pervasifs, c'est-à-dire leur évolution au cours du temps, il est nécessaire de rendre explicite la notion de temps. Nous utilisons une représentation discrète du temps. Nous considérons un domaine $\mathcal{T}$ discret, infini dénombrable et totalement ordonné d'instants $\tau_{i\in\mathbb{Z}} \in \mathcal{T}$, avec $i < j \Rightarrow \tau_i < \tau_j$. Nous considérons également l'instant présent τ_{now}.

Notre modèle de données est construit sur cinq notions de base, représentées par cinq ensembles infinis dénombrables mutuellement exclusifs : le temps $\mathcal{T}$ (défini précédemment) ; les constantes $\mathcal{D}$ et les attributs $\mathcal{A}$ (venant du modèle relationnel) ; les prototypes Ψ et les services Ω (venant de notre modélisation des fonctionnalités). Nous définissons également le domaine des booléens $\mathcal{B} = \langle true, false \rangle \subset \mathcal{D}$.

Nous réutilisons les notations classiques pour les schémas de relations et les relations. Pour un schéma R, $schema(R) \subset \mathcal{A}$ est l'ensemble de ses attributs. Un tuple sur R est un élément de $\mathcal{D}^{|schema(R)|}$ et une relation r sur R est un ensemble fini de tuples sur R.

Afin de représenter de manière homogène les relations classiques et les flux de données, nous proposons la notion de relation dynamique. Une relation dynamique r sur R est un ensemble de tuples qui évolue avec le temps. Plus précisément, trois

ensembles de tuples sont définis à chaque instant τ_i : la relation instantanée $r^*(\tau_i)$, l'ensemble fini des tuples ajoutés $r^+(\tau_i)$, et l'ensemble fini des tuples supprimés $r^-(\tau_i)$. Nous considérons d'une part les relations finies : $r^*(\tau_i)$ est alors un ensemble fini ; et d'autre part les relations infinies, c'est-à-dire les flux de données : $r^*(\tau_i)$ est alors un ensemble infini, et l'ensemble des tuples supprimés $r^-(\tau_i)$ est toujours vide. Cette propriété du schéma de relation dynamique R est dénotée par le prédicat $infinite(R) \in \mathcal{B}$. De plus, une relation dynamique r est associée à son instant de début τ_{start_r}, avec les conditions initiales suivantes : $r^+(\tau_{start_r}) = r^*(\tau_{start_r})$ et $r^-(\tau_{start_r}) = \emptyset$.

2.2. *Modélisation des fonctionnalités réparties*

La modélisation des fonctionnalités réparties d'un environnement pervasif doit notamment faire face à deux aspects dynamiques. D'une part, certaines fonctionnalités peuvent apparaître et disparaître au cours du temps. D'autre part, deux types de fonctionnalités coexistent : les méthodes qui peuvent être invoquées (par exemple, envoyer un message), et les flux de données auxquels il est possible de s'abonner (par exemple, un flux de température venant d'un capteur). De plus, il est important de considérer que certains fonctionnalités qui ont un "effet de bord" non négligeable sur l'environnement, même si elles sont vues comme des sources de données. Par exemple, l'envoi d'un message a un impact non négligeable, alors que la lecture d'une valeur sur un capteur de température n'en a pas.

Afin de prendre en compte ces aspects, nous avons choisi d'abstraire ces fonctionnalités d'une manière qui découple leur déclaration (par exemple, l'envoi d'un message à une adresse) et leur implémentation (par exemple, un envoi par messagerie instantanée). La déclaration de ces fonctionnalités prend la forme d'un **prototype de fonctionnalité** $\psi \in \Psi$ associé à deux schémas de relations représentant les paramètres d'entrée et de sortie, $Input_\psi$ et $Output_\psi$. Un prototype est soit un **prototype d'invocation**, représentant une méthode, soit un **prototype d'abonnement**, représentant un flux de données. Cette propriété est dénotée par le prédicat $streaming(\psi) \in \mathcal{B}$. L'appel d'un prototype nécessite un tuple sur le schéma $Input_\psi$ et produit un ou plusieurs tuples sur le schéma $Output_\psi$. Dans le cas d'une invocation, les tuples sont produits en une seule fois. Dans le cas d'un abonnement, des tuples peuvent être produits à chaque instant τ_i tant que dure cet abonnement. Les prototypes de fonctionnalités (invocations et abonnements) sont dits **actifs** si leur impact est significatif, ou **passifs** dans le cas contraire. Cette propriété est dénotée par le prédicat $active(\psi) \in Booleans$.

L'implémentation des fonctionnalités réparties est représentée par les méthodes et flux de données fournis par des **services**. Un service $\omega \in \Omega$ représente une entité logique ou physique de l'environnement pervasif qui **implémente des prototypes** de fonctionnalités, c'est-à-dire qui fournit des méthodes et des flux de données correspondant à des prototypes d'invocation et d'abonnement. Chaque service possède un identifiant global unique $id(\omega) \in \mathcal{D}$, appelé **référence de service**. L'ensemble des fonctionnalités qu'il implémente est dénoté par $prototypes(\omega) \subset \Psi$.

L'implémentation d'un prototype ψ se traduit par une fonction $data_\psi(s, t, \tau_i) \subset \mathcal{D}^{|schema(Output_\psi)|}$, avec la référence de service $s = id(\omega)$ et le tuple de paramètre d'entrée $t \in \mathcal{D}^{|schema(Input_\psi)|}$. Pour un prototype d'invocation, cette fonction représente le résultat d'une invocation à un instant τ_i. Pour un prototype d'abonnement, elle représente le contenu du flux à chaque instant τ_i tant que dure l'abonnement.

Un service se décrit à travers des **propriétés**, c'est-à-dire des couples attributs/valeurs. L'ensemble de ces attributs est dénoté par $attributes(\omega) \in \mathcal{A}$. La valeur d'un attribut $A \in attributes(\omega)$ décrivant ce service est représentée par la fonction $property_\omega(A) \in \mathcal{D}$.

De plus, un service est, à chaque instant, soit disponible, soit indisponible. Cette propriété est dénotée par le prédicat dépendant du temps $available(\omega, \tau_i) \in \mathcal{B}$.

Exemple 1 (Services et prototypes de fonctionnalités) *Considérons un environnement comprenant trois prototypes de fonctionnalités : envoi de message, récupération d'une valeur de température et abonnement à un flux de température ; et quatre services implémentant certaines de ces fonctionnalités : deux services de messagerie par e-mail (protocole SMTP) et par messagerie instantanée (protocole XMPP), et deux capteurs de température (ayant chacun une propriété indiquant leur localisation). Le tableau 1 présente ces services en utilisant un pseudo-DDL, et le tableau 2 utilise les notations introduites précédemment. On peut noter que la notion de découverte dynamique des fonctionnalités disponibles dans l'environnement est traduite dans notre modèle par le prédicat* $available(\omega, \tau_i) \in \mathcal{B}$*. Pour un service* ω *donné, sa découverte à l'instant* τ_j *signifie qu'il n'était pas disponible à l'instant précédent, et l'est devenu à cet instant :* $available(\omega, \tau_{j-1}) = false, available(\omega, \tau_j) = true$*. Le service est disponible tant que* $available(\omega, \tau_i)$ *reste vrai.*

Tableau 1. *Services et prototypes de fonctionnalités (pseudo-DDL)*

```
PROTOTYPE sendMessage( address STRING, message STRING ) : (sent BOOLEAN) ACTIVE;
PROTOTYPE getTemperature( ) : ( temperature REAL );
PROTOTYPE temperatureNotifications( ) : ( temperature REAL ) STREAMING;

SERVICE email    ( protocol STRING = "SMTP" )     IMPLEMENTS sendMessage;
SERVICE jabber   ( protocol STRING = "XMPP" )     IMPLEMENTS sendMessage;
SERVICE sensor01 ( location STRING = "office" )   IMPLEMENTS getTemperature, temperatureNotifications;
SERVICE sensor02 ( location STRING = "corridor" ) IMPLEMENTS getTemperature, temperatureNotifications;
```

2.3. *L'environnement pervasif relationnel*

Nous avons pour le moment introduit une modélisation du temps et des fonctionnalités réparties présentes dans un environnement pervasif telles qu'elles sont utilisées par les applications pervasives (découverte dynamique, invocations de méthodes, abonnements à des flux de données). Nous proposons maintenant une intégration de ces fonctionnalités avec les sources de données traditionnelles pour construire une

Tableau 2. *Services et prototypes de fonctionnalités (notation formelle)*

Prototypes : $sendMessage, getTemperature, temperatureNotifications \in \Psi$

- $schema(Input_{sendMessage}) = \{address, message\}$
- $schema(Output_{sendMessage}) = \{sent\}$
- $schema(Input_{getTemperature}) = schema(Input_{temperatureNotifications}) = \emptyset$
- $schema(Output_{getTemperature}) = schema(Output_{temperatureNotifications}) = \{temperature\}$
- $streaming(sendMessage) = false, active(sendMessage) = true$
- $streaming(getTemperature) = false, active(getTemperature) = false$
- $streaming(temperatureNotifications) = true, active(temperatureNotifications) = false$

Services : $\omega_1, \omega_2, \omega_3, \omega_4 \in \Omega$

- $id(\omega_1) = email, id(\omega_2) = jabber, id(\omega_3) = sensor01, id(\omega_4) = sensor02$
- $prototypes(\omega_1) = prototypes(\omega_2) = \{sendMessage\}$
- $attributes(\omega_1) = attributes(\omega_2) = \{protocol\}$
- $property_{\omega_1}(protocol) =$ "SMTP", $property_{\omega_2}(protocol) =$ "XMPP"
- $prototypes(\omega_3) = prototypes(\omega_4) = \{getTemperature, temperatureNotifications\}$
- $attributes(\omega_3) = attributes(\omega_4) = \{location\}$
- $property_{\omega_3}(location) =$ "office", $property_{\omega_4}(location) =$ "corridor"

vue orientée données de l'environnement pervasif. Cette vue permet de représenter et manipuler de manière homogène les données, flux de données et fonctionnalités. Nous présentons tout d'abord l'intégration des données et des fonctionnalités pour construire la notion de relation étendue, ou X-Relation (*eXtended Relation*). Nous construisons ensuite la notion de relation dynamique étendue, ou XD-Relation (*eXtended Dynamic Relation*), prenant pleinement en compte les aspects continus.

2.3.1. *Les X-Relations*

Nous proposons une intégration entre les données et les fonctionnalités à deux niveaux : au niveau méta-données, les prototypes sont intégrés dans les schémas de relation ; au niveau données, les services sont intégrés dans les tuples de données simplement par leur **référence de service** (pour un service $\omega \in \Omega$, sa référence est $id(\omega) \in \mathcal{D}$).

Au niveau méta-données, nous intégrons les prototypes de fonctionnalités dans les schémas à travers deux notions : les **attributs virtuels** et les ***binding patterns***. En général, les attributs virtuels représentent des attributs dont la valeur n'est pas stockée, mais peut être calculée à partir d'autres données disponibles, par une fonction externe, ou récupérée d'une source externe. Les *binding patterns* ont été introduits dans le contexte de l'intégration de données comme une description des *patterns* d'accès restreint à des sources de données relationnelles (Florescu *et al.*, 1999). Nous adaptons ces deux notions dans le contexte des environnements pervasifs.

Nous étendons les schémas de relation avec des attributs virtuels : ceux-ci sont définis dans le schéma, mais n'ont pas de valeur au niveau données, c'est-à-dire dans les

tuples. Les attributs virtuels peuvent être transformés en attribut “réels”, c’est-à-dire non virtuel, par certains opérateurs de requête (cf. section 3). Il est à noter qu’étendre un schéma avec des attributs virtuels ne modifie pas la représentation des données des relations définies sur ce schéma : les attributs virtuels représentent seulement des attributs potentiels qui peuvent être utilisés par les requêtes.

Les *binding patterns* sont le lien entre les références de service, les attributs virtuels et les prototypes de fonctionnalité. Un *binding pattern* est associé à un schéma étendu et définit quel prototype (d’invocation ou d’abonnement) utiliser sur les services pour récupérer des valeurs pour un ou plusieurs attributs virtuels. Il spécifie également quel attribut réel représente les références de service. Les schémas d’entrée et de sortie du prototype indique quels sont les attributs de la relation à utiliser comme paramètres d’entrée et de sortie. Les paramètres d’entrée peuvent être des attributs réels ou virtuels, alors que les paramètres de sortie doivent être des attributs virtuels. Un *binding pattern* prend les caractéristiques du prototype qu’il utilise : il est soit un *binding pattern* d’invocation, soit un *binding pattern* d’abonnement, et est soit actif, soit passif. Associer un *binding pattern* à un schéma étendu de relation n’influence pas la représentation des données des relations définies sur ce schéma : les *binding patterns* représentent un moyen potentiel de fournir des valeurs à des attributs virtuels qui peut être utilisé par les requêtes pour interagir avec les services.

À l’aide de ces deux notion, nous définissons maintenant la notion de relation étendue, ou X-Relation, sur un schéma de relation étendu, c’est-à-dire étendu avec des attributs virtuels et associé avec des binding patterns. À noter qu’un tuple n’est défini que sur la partie réelle du schéma. La projection de ces tuples sur un sous-ensemble d’attributs réels doit ainsi prendre en compte le “décalage” dû aux attributs virtuels du schéma.

Définition 1 (Schéma de relation étendue) *Un* schéma de relation étendue *est un symbole de relation étendue R associé à :*

– $type(R)$ le nombre d’attributs dans R,

– $att_R : \{1, ..., type(R)\} \mapsto \mathcal{A}$ un fonction injective associant à des nombres entiers des attributs dans R,

– $schema(R)$ l’ensemble des attributs dans R, c’est-à-dire $\{att_R(1), ..., att_R(type(R))\} \subseteq \mathcal{A}$,

– $\{realSchema(R), virtualSchema(R)\}$ une partition de $schema(R)$ avec :

- $realSchema(R)$ le schéma réel, c’est-à-dire le sous-ensemble des attributs réels,

- $virtualSchema(R)$ le schéma virtuel, c’est-à-dire le sous-ensemble des attributs virtuels ;

– $BP(R) \subset (\Psi \times \mathcal{A})$ un ensemble fini de binding patterns associés à R, où $bp = \langle prototype_{bp}, service_{bp} \rangle \in BP(R)$ avec :

- $prototype_{bp} \in \Psi$ le prototype associé au binding pattern,

- $service_{bp} \in realSchema(R)$ un attribut réel du schema représentant les références de service,
avec les contraintes suivantes :

- $schema(Input_{prototype_{bp}}) \subset schema(R)$,
- $schema(Output_{prototype_{bp}}) \subseteq virtualSchema(R)$.

Nous dénotons par $active(bp)$, avec $bp \in BP(R)$, un prédicat indiquant si le binding pattern bp est actif, c'est-à-dire si le prototype associé est actif.

Définition 2 (Relation étendue) *Un* tuple sur un schéma de relation étendue R *est un élément de $\mathcal{D}^{|realSchema(R)|}$. Une* relation étendue sur R *(ou X-Relation sur R) est un ensemble fini de tuples sur R.*

Définition 3 (Projection d'un tuple) *La* projection d'un tuple t*, d'une X-Relation r sur un schéma de relation étendu R, sur un attribut $A_i \in realSchema(R)$ avec $A_i = att_R(i)$ dans $schema(R)$, est dénotée $t[A_i]$. Cette projection est la j^{eme} coordonnée de t, c'est-à-dire $t(j)$, avec $j = \delta_R(i)$ le nombre d'attributs réels dans $\{att_R(1), ..., att_R(i)\}$, c'est-à-dire $\delta_R(i) = |\{att_R(1), ..., att_R(i)\} \cap realSchema(R)|$.*

La projection d'un tuple t sur $X = \{att_R(i_1), ..., att_R(i_n)\} \subseteq realSchema(R)$, dénotée $t[X]$, est $t[X] = \langle t(\delta_R(i_1)), ..., t(\delta_R(i_n))\rangle$.

Exemple 2 (Relation étendue) *Une liste de contacts électroniques peut être modélisée par le schéma de relation étendue $Contact$:*

$$\begin{array}{lcl} schema(Contact) & = & \{name, address, message, messenger, sent\}, \\ realSchema(Contact) & = & \{name, address, messenger\}, \\ virtualSchema(Contact) & = & \{message, sent\}, \\ BP(Contact) & = & \{\langle sendMessage, messenger\rangle\}. \end{array}$$

$messenger$ est un attribut représentant des références de service. $message$ and $sent$ sont deux attributs virtuels représentant le message à envoyé et le résultat de l'envoi. Ce schéma est associé à un binding pattern $\langle sendMessage, messenger\rangle$ utilisant le prototype d'invocation $sendMessage$. Soit $contacts$ une X-Relation sur $Contact$, elle peut être représentée dans le tableau suivant, où '$*$' dénote l'absence de valeur pour les attributs virtuels :

$name$	$address$	$message$	$messenger$	$sent$
Nicolas	*nicolas@elysee.fr*	$*$	*email*	$*$
Carla	*carla@elysee.fr*	$*$	*email*	$*$
François	*francois@im.gouv.fr*	$*$	*jabber*	$*$

2.3.2. *Les XD-Relations*

À partir de ces définitions concernant les X-Relations, nous définissons maintenant la notion de relation dynamique étendue, ou XD-Relation, prenant en compte à la fois les aspects temporels et l'intégration des données et des fonctionnalités. Cette notion étend notamment la notion préliminaire de relation dynamique représentant

de manière homogène les données et flux de données : une XD-Relation peut ainsi toujours représenter une source de données conventionnelle.

Définition 4 (Schéma de relation dynamique étendue) *Un* schéma de relation dynamique étendue *est un schéma de relation étendue R associé à un prédicat $infinite(R) \in \mathcal{B}$ qui est vrai si R est un schéma de relation dynamique étendue infinie, c'est-à-dire représentant un flux de données étendu, et faux si R est un schéma de relation dynamique étendue finie.*

Un tuple sur un schéma de relation dynamique étendue R est un tuple sur le schéma de relation étendue R, c'est-à-dire un élément de $\mathcal{D}^{|realSchema(R)|}$.

Définition 5 (Relation dynamique étendue) *Une* relation dynamique étendue r *sur un schéma de relation dynamique étendue R est associé à un instant de début $\tau_{start_r} \in \mathcal{T}$ et est définie par trois ensembles de tuples sur R pour chaque instant $\tau_i \geq \tau_{start_r}$: la X-Relation instantanée $r^*(\tau_i)$, c'est-à-dire le contenu de la XD-Relation à l'instant τ_i, l'ensemble fini des tuples ajoutés $r^+(\tau_i)$, et l'ensemble fini des tuples supprimés $r^-(\tau_i)$. Si r est une XD-Relation finie, la relation instantanée $r^*(\tau_i)$ est un ensemble fini. Si r est une XD-Relation infinie, la relation instantanée $r^*(\tau_i)$ est un ensemble infini et l'ensemble des tuples supprimés $r^-(\tau_i)$ est toujours vide. Pour l'instant de début τ_{start_r}, les conditions initiales suivantes sont toujours valides : $r^+(\tau_{start_r}) = r^*(\tau_{start_r})$ et $r^-(\tau_{start_r}) = \emptyset$.*

2.3.3. *L'environnement pervasif relationnel*

Les XD-Relations, avec leurs attributs virtuels et leurs *binding patterns* actifs/passifs, d'invocation ou d'abonnement, se révèlent être une notion suffisamment puissante pour représenter une grande variété de sources de données hétérogènes et de fonctionnalités réparties d'un environnement pervasif. Notre représentation permet de gérer ces sources de données conventionnelles et non-conventionnelles dans un cadre homogène, ce qui est un besoin fort pour le développement des applications pervasives.

Nous concluons la définition de la structure de notre modèle de données par la définition de la notion d'environnement pervasif relationnel, représentant un ensemble de XD-Relations, de manière similaire à la notion de base de données. Par souci de simplicité, nous conservons l'hypothèse de nommage de la relation universelle (*Universal Relation Schema Assumption* (URSA) (Maier *et al.*, 1984)) indiquant que si un attribut apparaît dans plusieurs schémas de relation, alors cet attribut représente les mêmes données. Dans notre modèle, cette hypothèse concerne les attributs de tous les schémas : schémas d'entrée et de sortie des prototypes, schémas de relation (dynamique), schémas de relation étendue (dynamique) ; de même que les attributs représentant les propriétés des services.

Définition 6 (Schéma d'environnement pervasif relationnel) *Un* schéma d'environnement pervasif relationnel P *est un ensemble fini $P = \{R_1, ..., R_n\}$, avec R_i un*

schéma de relation dynamique étendue. Nous dénotons par $schema(P)$ *l'ensemble des attributs associés aux schémas de relation dynamique étendue de* P*, c'est-à-dire* $schema(P) = \bigcup_{R_i \in P} schema(R_i)$.

Définition 7 (Environnement pervasif relationnel) *Un* environnement pervasif relationnel *sur un schéma d'environnement pervasif relationnel* $P = \{R_1, ..., R_n\}$ *est un ensemble* $p = \{r_1, ..., r_n\}$*, avec* $r_i \in p$ *une relation dynamique étendue sur* $R_i \in P$.

Avant de poursuivre la présentation de notre modèle de données par la définition de son langage (section 3), l'ensemble des notations de sa structure est résumé dans le tableau 3.

2.4. *Syntaxe DDL*

Nous avons défini une syntaxe de DDL similaire à celle du SQL. Une XD-Relation est identifiée par son nom et est soit finie (mot-clé RELATION), soit infinie (mot-clé STREAM). Elle comprend une liste d'attributs typés (STRING, INTEGER, REAL, BINARY, SERVICE) qui peuvent être virtuels (mot-clé VIRTUAL). Les *binding patterns* sont décrits par le nom du prototype, le nom de l'attribut représentant les références de service (de type SERVICE), et les noms des attributs d'entrée/sortie. Un *binding pattern* d'abonnement est spécifié par le mot-clé STREAMING.

Exemple 3 *Pour le scénario de surveillance de température, les XD-Relations suivantes sont utilisées : le carnet d'adresses électroniques* `contacts`*, une liste de caméras* `cameras`*, une liste de capteurs de température* `sensors`*, et un exemple de flux de température* `temperatures`*. Ces XD-Relations sont décrites en DDL dans le tableau 4.*

Tableau 3. *Résumé des notations de la structure du modèle de données SoCQ*

	STRUCTURE
DONNÉES	– Domaine des Booléens $\mathcal{B}$ – Constantes $\mathcal{D}$ – Attributs $\mathcal{A}$ – Schéma de relation R - $schema(R) \subset \mathcal{A}$ – Relation r sur R - $r \subset \mathcal{D}^{\lvert schema(R) \rvert}$
DONNÉES + SERVICES	– Prototypes $\psi \in \Psi$ - Schéma de relation $Input_\psi$ - Schéma de relation $Output_\psi$ - $active(\psi) \in \mathcal{B}$ – Services $\omega \in \Omega$ - $id(\omega) \in \mathcal{D}$ - $prototypes(\omega) \subset \Psi$ - $attributes(\omega) \subset \mathcal{A}$ – Interaction avec les services - $available(\omega) \in \mathcal{B}$ - $property_\omega(A) \in \mathcal{D}$ - $data_\psi(id(\omega), input) \in \mathcal{D}^{\lvert schema(Output_\psi) \rvert}$ – Schéma de X-Relation R - Schéma de relation R - $realSchema(R)$ - $virtualSchema(R)$ - $BP(R) \subset (\Psi \times \mathcal{A})$ – X-Relation r sur R - $r \subset \mathcal{D}^{\lvert realSchema(R) \rvert}$
DONNÉES + SERVICES + TEMPS + FLUX	– Domaine du temps discret $\tau_i \in \mathcal{T}$ – Interaction avec les services - $available(\omega, \tau_i) \in \mathcal{B}$ - $data_\psi(id(\omega), input, \tau_i) \in \mathcal{D}^{\lvert schema(Output_\psi) \rvert}$ – Schéma de XD-Relation R - Schéma de X-Relation R - $infinite(R) \in \mathcal{B}$ – XD-Relation r sur R - $\tau_{start_r} \in \mathcal{T}$ - $r^*(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R) \rvert}$ - $r^+(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R) \rvert}$ - $r^-(\tau_i) \subset \mathcal{D}^{\lvert realSchema(R) \rvert}$

Tableau 4. *XD-Relations du scénario de surveillance de température (DDL)*

```
RELATION contacts (
  name      STRING,
  address   STRING,
  messenger SERVICE,
  message   STRING   VIRTUAL,
  sent      BOOLEAN  VIRTUAL
)
USING BINDING PATTERNS (
  sendMessage[messenger] ( address, message ) : ( sent )
);

RELATION cameras (
  camera    SERVICE,
  location  STRING,
  photo     BINARY   VIRTUAL
)
USING BINDING PATTERNS (
  takePhoto[camera] ( ) : ( photo )
);

RELATION sensors (
  sensor      SERVICE,
  location    STRING,
  temperature REAL     VIRTUAL
)
USING BINDING PATTERNS (
  getTemperature[sensor] ( ) : ( temperature ),
  temperatureNotifications[sensor] ( ) : ( temperature ) STREAMING
);

STREAM temperatures (
  location    STRING,
  temperature REAL
);
```

3. Requêtage des environnements pervasifs

Afin d'atteindre notre but de simplifier le développement d'applications pervasives, l'étape critique suivante est la définition d'un langage de requête qui permet d'exprimer à la fois les interactions simples et les interactions complexes au sein d'un environnement pervasif relationnel à un niveau déclaratif. Dans le cadre du modèle relationnel, l'algèbre relationnelle est reconnue comme un outil fondamental pour évaluer l'expressivité d'un langage de requête (d'où la notion de "complétude relationnelle" (Codd, 1972)). Dans le cadre des environnements pervasifs, nous proposons une algèbre pour les environnements pervasifs relationnels, qui nous permet de définir l'"expressivité" de notre langage de requête. De plus, cette algèbre étant proche de l'algèbre relationnelle, la définition d'un langage similaire au SQL est simple.

Dans cette section, nous définissons l'algèbre Séréna (*Service-enabled algebra*) sur les environnements pervasifs relationnels. Nous définissons tout d'abord l'algèbre pour les requêtes ponctuelles (*one-shot*), c'est-à-dire les requêtes non continues, puis

l'algèbre pour les requêtes continues. La notion d'équivalence de requêtes est également abordée afin de permettre des techniques d'optimisation.

3.1. *Algèbre de requête ponctuelle*

Pour une requête q, nous considérons son instant de début τ_{start_q}, c'est-à-dire l'instant auquel cette requête est lancée. L'algèbre de requête ponctuelle est composée d'opérateurs définis sur une ou deux X-Relations, qui peuvent être les X-Relations instantanées à τ_{start_q} de XD-Relations de l'environnement pervasif relationnel, ou des X-Relations produites par d'autres opérateurs ponctuels, car les résultats produits par ces opérateurs sont des X-Relations. Une partie des opérateurs ponctuels sont des redéfinitions d'opérateurs de l'algèbre relationnelle, et d'autres sont spécifiquement dédiés à la manipulation des attributs virtuels, des binding patterns et des services. À noter que les requêtes ponctuelles ne sont pas pertinentes pour manipuler des flux de données : nous ne considérons donc pas pour le moment les XD-Relations infinies et les *binding patterns* d'abonnement.

3.1.1. *Opérateurs ensemblistes*

Les opérateurs d'union, d'intersection et de différence peuvent s'appliquer sur deux X-Relations associées au même schéma. La X-Relation produite est définie sur ce schéma commun. La définition de ces opérateurs reste similaire à leur définition dans l'algèbre relationnelle. Soit r_1 et r_2 deux X-Relations sur un schéma de X-Relation R :

– Union : $r_1 \cup r_2 = \{t \mid t \in r_1 \vee t \in r_2\}$
– Intersection : $r_1 \cap r_2 = \{t \mid t \in r_1 \wedge t \in r_2\}$
– Différence : $r_1 - r_2 = \{t \mid t \in r_1 \wedge t \notin r_2\}$

3.1.2. *Opérateurs relationnels*

Nous étendons les opérateurs relationnels standards pour les définir sur une ou deux X-Relations. Au niveau données, leur définition reste similaire, excepté le fait que seuls les attributs réels ont une valeur. Au niveau méta-données, la définition du schéma de la X-Relation produite doit prendre en compte les *binding patterns* des opérandes, en particulier le fait que le renommage ou la disparition de certains attributs peut en modifier ou en invalider certains.

L'opérateur de **projection** ($\pi_Y(r)$, avec $Y \subset schema(R)$) réduit le schéma d'une X-Relation, et donc ses schémas réel et virtuel. La X-Relation produite est associée avec les *binding patterns* qui restent valides, c'est-à-dire ceux dont l'attribut représentant des références de services et les attributs d'entrée/sortie sont inclus dans le schéma réduit.

L'opérateur de **sélection** ($\sigma_F(r)$, avec F formule de sélection sur $realSchema(R)$) ne modifie pas le schéma de la X-Relation. Il sélectionne les

tuples satisfaisant la formule de sélection. Cette formule de sélection ne peut cependant porter que sur les attributs réels de la X-Relation puisque les attributs virtuels n'ont pas de valeur.

L'opérateur de **renommage** ($\rho_{A \to B}(r)$, avec $A, B \in \mathcal{A}, A \in schema(R), B \notin schema(R)$) remplace un attribut du schéma par un autre attribut, sans modifier son statut réel ou virtuel. Les *binding patterns* sont également impactés : l'attribut représentant des références de service peut être simplement renommé, mais un *binding pattern* est invalidé si l'attribut renommé est dans les attributs d'entrée/sortie de son prototype (qui lui ne peut être modifié).

L'opérateur de **jointure naturelle** ($r_1 \bowtie r_2$) effectue une jointure de deux X-Relations, les attributs de jointure étant les attributs communs des schémas des deux opérandes. Si un attribut de jointure est réel (respectivement virtuel) dans les deux opérandes, il reste réel (respectivement virtuel) dans la X-Relation produite. Cependant, si il est réel dans une opérande et virtual dans l'autre, alors il devient réel dans la X-Relation résultante (ce qui est une réalisation implicite, cf. les opérateurs de réalisation). Seuls les attributs de jointure réels dans les deux opérandes impliquent un prédicat de jointure, puisque les attributs virtuels n'ont pas valeur. Par exemple, si tous les attributs de jointure sont virtuels, la jointure est équivalente, au niveau données, à un produit cartésien. L'ensemble des *binding patterns* de la X-Relation produite est l'union des ensembles de *binding patterns* des deux opérandes, excepté ceux dont les attributs de sortie initialement virtuels sont devenus réels du fait de la jointure.

3.1.3. *Opérateurs de réalisation*

Nous introduisons deux nouveaux opérateurs, appelés **opérateurs de réalisation**, permettant de transformer des attributs virtuels en attributs réels (d'où le nom de ces opérateurs). En dépit de leur simplicité, ces opérateurs sont une composante clé pour effectuer des requêtes de manière homogène sur des données et des services. La "réalisation" des attributs virtuels consiste à leur donner une valeur, soit directement (opérateur d'affectation), soit en utilisant un *binding pattern* (opérateur de *binding*). Une réalisation implicite peut également avoir lieu au sein d'une jointure naturelle quand un attribut est réel dans un opérande et virtual dans l'autre, l'attribut virtuel devenant un attribut réel dans la X-Relation résultante.

L'opérateur d'**affectation** ($\alpha_{A \equiv c}(r)$ ou $\alpha_{A \equiv B}(r)$, avec $A \in virtualSchema(R), B \in realSchema(R), c \in \mathcal{D}$) permet de réaliser individuellement un attribut virtuel en lui affectant une valeur : soit la valeur d'une constante, soit la valeur d'un autre attribut réel du schéma. L'attribut devient ainsi réel dans la X-Relation produite. Les *binding patterns* sont les mêmes que ceux de l'opérande, excepté ceux qui ont l'attribut réalisé dans leurs attributs de sortie.

L'opérateur de **binding** ($\beta_{\langle \psi, S \rangle}(r)$ avec $\langle \psi, S \rangle \in BP(R)$) permet de réaliser les attributs de sortie d'un *binding pattern*, à condition que tous les attributs d'entrée soient réels dans l'opérande (c'est-à-dire $schema(Input_\psi) \subseteq realSchema(R)$). Pour chaque tuple t de l'opérande, les valeurs des attributs de sortie sont récupérées

en invoquant le prototype ψ du *binding pattern* sur le service identifié par l'attribut S. Les paramètres d'entrée de cet invocation sont extraits du tuple t, et les paramètres de sortie sont extraits du résultat de cet invocation, représenté par la fonction $data_\psi(t[S], t[schema(Input_\psi)], \tau_{start_q})$. Les *binding patterns* de la X-Relation produite sont les mêmes que ceux de l'opérande, excepté ceux qui ont un des attributs réalisés dans leurs attributs de sortie.

3.1.4. *Opérateur de découverte de services*

Afin d'intégrer pleinement au sein des requêtes les fonctionnalités réparties d'un environnement dynamique, un opérateur représentant la découverte dynamique de services est nécessaire. Un tel opérateur permet de manipuler l'ensemble des services disponibles de l'environnement pour en extraire un sous-ensemble de services respectant certains critères.

L'opérateur de **découverte de services** ($\xi_{\langle S, \mathcal{A}_\phi, \Psi_\phi \rangle}()$, avec $S \in \mathcal{A}, \mathcal{A}_\phi \subset \mathcal{A}, \Psi_\phi \subset \Psi$) ne prend aucune X-Relation comme opérande. Il produit néanmoins une nouvelle X-Relation qui représente l'ensemble des services $\omega \in \Omega$ disponibles ($available(\omega, \tau_{start_q}) = true$), qui implémentent un ensemble de prototypes d'invocation et d'abonnement ($implements(\omega) \subseteq \Psi_\phi$) et qui sont décrits par un ensemble d'attributs, ou propriétés ($attributes(\omega) \subseteq \mathcal{A}_\phi$). Le schéma de la X-Relation contient un attribut réel représentant des services, des attributs réels correspondants aux attributs requis décrivant les services, et des attributs virtuels correspondant aux attributs d'entrée/sortie des prototypes requis. Les *binding patterns* associés au schéma sont l'ensemble des prototypes requis associés à l'attribut représentant des services. Soit R le schéma de cet X-Relation, on a :

$$
\begin{array}{lcl}
schema(R) & = & \{S\} \cup \mathcal{A}_\phi \cup \bigcup_{\psi \in \Psi_\phi} (schema(Input_\psi) \cup schema(Output_\psi)) \,, \\
realSchema(R) & = & \{S\} \cup \mathcal{A}_\phi \,, \\
virtualSchema(R) & = & \bigcup_{\psi \in \Psi_\phi} (schema(Input_\psi) \cup schema(Output_\psi)), \\
BP(R) & = & \bigcup_{\psi \in \Psi_\phi} (\langle \psi, S \rangle).
\end{array}
$$

Pour chaque service ω respectant ces critères, un tuple t est présent dans la X-Relation, comportant l'identifiant du service ($t[S] = id(\omega)$) et la valeur des attributs décrivant ce service ($\forall A \in \mathcal{A}_\phi, t[A] = property_\omega(A)$).

Exemple 4 (Opérateur de découverte de services) *L'expression algébrique suivante, composée d'un simple opérateur de découverte de services, permet de construire une X-Relation représentant tous les capteurs de température de l'environnement :*

$$\xi_{\langle sensor, \{location\}, \{getTemperature, temperatureNotifications\} \rangle}()$$

Le schéma R de cet X-Relation est :

$$
\begin{array}{lcl}
schema(R) & = & \{sensor, location, temperature\}\,, \\
realSchema(R) & = & \{sensor, location\}\,, \\
virtualSchema(R) & = & \{temperature\}, \\
BP(R) & = & \{\langle getTemperature, sensor\rangle, \langle temperatureNotifications, temperature\rangle\}.
\end{array}
$$

En considérant les services présenté à l'exemple 1 (et en supposant qu'ils soient disponibles à l'instant τ_{start_q}), cette X-Relation contiendrait deux tuples pour les services ω_3 et ω_4 qui implémentent les prototypes $getTemperature$ et $temperatureNotifications$ et qui sont décrits par l'attribut $location$:

$sensor$	$location$	$temperature$
sensor01	*office*	*
sensor02	*corridor*	*

3.1.5. *Requête ponctuelle sur un environnement pervasif relationnel*

Suite à la définition des opérateurs algébriques sur des X-Relations, nous pouvons définir la notion de requête ponctuelle sur un environnement pervasif relationnel.

Définition 8 (Requête ponctuelle) *Une* requête ponctuelle q *sur un environnement pervasif relationnel p est une expression bien formée composée d'un nombre fini d'opérateurs de l'algèbre de requête ponctuelle sur des X-Relations de p. Elle est associée à un instant de début, dénoté $\tau_{start_q} \in \mathcal{T}$.*

Exemple 5 (Requêtes ponctuelles) *Nous pouvons exprimer les requêtes ponctuelles suivantes sur les X-Relations de l'environnement pervasif relationnel décrit dans l'exemple 3 :*

Q_1 envoyer le message "Bonjour !" à tous les contacts, sauf à Carla ;

Q_2 obtenir la température de tous les capteurs localisés dans la zone "office".

La requête Q_1 est composée de trois opérateurs et implique la XD-Relation $contacts$. Un opérateur de sélection élimine les contacts dont le nom est "Carla". Un opérateur d'affectation spécifie le texte du message, "Bonjour !", dans l'attribut virtuel $message$. Enfin, un opérateur de binding *invoque le* binding pattern *associé au prototype $sendMessage$, réalisant ainsi l'attribut $sent$.*

$$Q_1 = \quad \beta_{\langle sendMessage, messenger\rangle}(\alpha_{message \equiv \text{"Bonjour !"}}(\sigma_{name \neq \text{"Carla"}}(contacts)))$$

La requête Q_2 est composée de quatre opérateurs. Elle n'implique aucune XD-Relation, car elle utilise un opérateur de découverte de services pour produire une X-Relation contenant tous les capteurs de température. Un opérateur de sélection conserve uniquement les capteurs présent dans la zone "office". Un opérateur de binding *permet de réaliser l'attribut virtuel $temperature$ grâce au* binding pattern

associé au prototype $getTemperature$. Enfin, un opérateur de projection est utilisé pour conserver uniquement les attributs $sensor$ et $temperature$.

$$Q_2 = \pi_{sensor,temperature}(\beta_{\langle getTemperature,sensor\rangle}(\sigma_{location="office"}(\xi_{\langle sensor,\{location\},\{getTemperature,temperatureNotifications\}\rangle}()))$$

Les requêtes ponctuelles sur un environnement pervasif relationnel permet l'expression déclarative d'interactions ponctuelles entre données et services. L'accès physique aux sources de données et aux fonctionnalités est complètement abstrait. De plus, l'opérateur de découverte de services permet l'abstraction de la découverte des ressources disponibles.

3.2. *Algèbre de requête continue*

À la suite de la définition de l'algèbre pour les requêtes ponctuelles, nous définissons maintenant l'algèbre pour les requêtes continues. Les requêtes continues sont définies sur des XD-Relations finies et infinies, et produisent des XD-Relations finies ou infinies. Les *binding patterns* d'abonnement sont également pris en compte.

Une requête continue q est, comme une requête ponctuelle, associée à un instant de début τ_{start_q}. Cet instant de début détermine l'instant de début de la XD-Relation produite, mais impacte également sur les XD-Relations opérandes. En effet, la requête considère ses opérandes comme débutant à τ_{start_q}, c'est-à-dire que les conditions initiales s'appliquent à cet instant (l'ensemble des tuples supprimés est vide, et l'ensemble des tuples ajoutés correspond à la relation instantanée). À noter que ces conditions initiales à τ_{start_q} sont seulement applicable pour cette requête q, et n'affectent en aucun cas d'autres requêtes q_i, ayant chacune leur propre instant de début $\tau_{start_{q_i}}$, qui impliqueraient les mêmes XD-Relations comme opérandes.

Les opérateurs continus comprennent l'ensemble des opérateurs ponctuels simplement étendus sur des XD-Relations finies. L'opérateur de *binding* doit en revanche être plus précisément redéfini. Deux nouveaux opérateurs sont également introduits pour manipuler les XD-Relations infinies.

3.2.1. *Extension des opérateurs ponctuels*

La définition des opérateurs ensemblistes (union, intersection, différence), des opérateurs relationnels (sélection, projection, renommage, jointure naturelle) et de l'opérateur d'affectation sur des XD-Relations finies est simple à partir de leur définition sur des X-Relations. Ces opérateurs produisent des XD-Relations finies avec le même schéma que pour les X-Relations : pour chaque instant τ_i, la X-Relation instantanée produite est le résultat de l'opérateur ponctuel appliqué sur les X-Relations instantanées des opérandes. Par exemple, pour l'opérateur continu de sélection $s = \sigma_F(r)$:

$\forall \tau_i \geq \tau_{start_q}, s^*(\tau_i) = \sigma_F(r^*(\tau_i))$
$\forall \tau_i > \tau_{start_q}, s^+(\tau_i) = \sigma_F(r^*(\tau_i)) - \sigma_F(r^*(\tau_{i-1}))$
$\forall \tau_i > \tau_{start_q}, s^-(\tau_i) = \sigma_F(r^*(\tau_{i-1})) - \sigma_F(r^*(\tau_i))$

L'opérateur de découverte de services n'ayant pas d'opérande, son résultat continu est simplement l'application à chaque instant τ_i de l'opérateur ponctuel. À noter que le seul paramètre pouvant influer sur le résultat est le prédicat $available(\omega, \tau_i)$ indiquant si le service ω est disponible à l'instant τ_i.

3.2.2. *Opérateur de* binding

L'opérateur continu de binding $\beta_{\langle \psi, S \rangle}$ représente en fait deux opérateurs : l'opérateur de *binding* d'invocation et l'opérateur de *binding* d'abonnement. Sa nature dépend de la nature du *binding pattern* utilisé. Dans les deux cas, l'opérateur continu produit une XD-Relation avec le même schéma que l'opérateur ponctuel de *binding*, c'est-à-dire que les attributs de sortie du prototype ψ du *binding pattern* sont réalisés.

L'opérateur de *binding* d'invocation produit une XD-Relation finie. L'invocation du prototype associé se fait pour tout nouveau tuple ajouté dans l'opérande, c'est-à-dire que chaque tuple n'implique qu'une seule invocation à l'instant où il est ajouté. L'opérateur de *binding* d'abonnement produit une XD-Relation infinie. L'abonnement au prototype associé est fait pour tout nouveau tuple ajouté dans l'opérande, comme pour une invocation, mais l'abonnement dure tant que ce tuple n'est pas supprimé de l'opérande. L'opérateur est ainsi abonné à autant de prototype de flux de données que de tuples dans son opérande, et agrège ces flux : chaque donnée arrivant sur un des flux à l'instant τ_i ajoute un tuple à cet instant τ_i dans la XD-Relation infinie produite.

3.2.3. *Opérateur de flux*

Les opérateurs de flux permettent de gérer les XD-Relations infinies. Ils ne modifient pas le schéma de la XD-Relation, mis à part son caractère fini ou infini : les attributs virtuels et les *binding patterns* sont gérés de manière transparente.

L'opérateur de **fenêtrage** ($\mathcal{W}_{[size]}(r)$) produit une XD-Relation finie à partir d'une XD-Relation infinie en conservant à l'instant τ_i tous les tuples ajoutés à son opérande entre l'instant τ_i et $\tau_i i - size$, où $size$ représente la taille de la fenêtre temporelle.

L'opérateur de *streaming* ($\mathcal{S}_{[event]}$) produit une XD-Relation infinie à partir d'une XD-Relation finie en ajoutant à l'instant τ_i l'ensemble des tuples qui sont ajoutés/supprimés/présents dans l'opérande à cet instant, en fonction du type de l'évènement $event$: $insertion$, $deletion$ ou $heartbeat$.

3.2.4. *Requête continue sur un environnement pervasif relationnel*

De manière similaire aux requêtes ponctuelles, à partir de la définition des opérateurs algébriques sur des XD-Relations, nous pouvons définir la notion de requête continue sur un environnement pervasif relationnel.

Définition 9 *Une* requête continue q *sur un environnement pervasif relationnel* p *est une expression bien formée composée d'un nombre fini d'opérateurs de l'algèbre de requête continue sur des XD-Relations de* p*. Elle est associée à un instant de début, dénoté* $\tau_{start_q} \in \mathcal{T}$*, et produit une XD-Relation associée à ce même instant de début. Les XD-Relations opérandes sont considérées, pour cette requête* q*, comme associée au même instant de début, c'est-à-dire que les conditions initiales s'appliquent à* τ_{start_q}*.*

Exemple 6 (Requêtes continues) *Nous pouvons exprimer la requête continue suivante sur les XD-Relations de l'environnement pervasif relationnel décrit dans l'exemple 3 :*

Q_3 *quand la température dépasse 35.5°C dans la zone "office", envoyer le message "Chaud !" à tous les contacts.*

Cette requête Q_3 *utilise le flux de données* $temperatures$ *avec une fenêtre temporelle de taille 1 (c'est-à-dire, contenant seulement les tuples ajoutés à l'instant présent). Les notifications de température sont filtrées avec deux opérateurs de sélection successifs sur la zone et la température. Ces notifications filtrées sont ensuite combinées avec tous les contacts par une jointure naturelle (équivalent à un produit cartésien dans ce cas). Une valeur est affectée à l'attribut virtuel* $message$*, puis le* binding pattern *associé au prototype* $sendMessage$ *est invoqué.*

$$
\begin{aligned}
Q_3 = \quad & \beta_{\langle sendMessage, messenger\rangle}(\alpha_{message \equiv \text{"Chaud !"}}(contacts \bowtie \sigma_{temperature > 35.5}(\\
& \quad \sigma_{location = \text{"office"}}(\mathcal{W}_{[1]}(temperatures)) \\
&)))
\end{aligned}
$$

Les requêtes continues sur un environnement pervasif relationnel permettent d'exprimer déclarativement des interactions entre des données, des flux de données et des services. En particulier, les applications qui font du *monitoring* de flux de données et utilisent des méthodes de service pour gérer certains évènements (par exemple, envoyer des messages de notifications ou prendre des photos) peuvent facilement se développer à travers de simples expressions composées de peu d'opérateurs. De plus, l'opérateur de découverte de service permet d'intégrer dynamiquement toutes les ressources disponibles (par exemple, de nouveaux capteurs de température) pendant l'exécution de la requête continue.

3.3. *Équivalence de requêtes*

La notion d'équivalence de deux requêtes est cruciale pour permettre d'une part de définir des règles de réécriture de requêtes, d'autre part de concevoir des techniques d'optimisation, notamment à partir de ces règles de réécriture. Définir cette équivalence dans le contexte dynamique des environnements pervasifs nécessite de répondre à trois points : la dépendance au temps, le déterminisme des services et l'impact des services sur l'environnement.

Pour répondre aux deux premiers points, nous supposons que tous les services sont déterministes à un instant τ_i donné. Ainsi, deux appels au même service effectué au même instant τ_i avec les mêmes paramètres d'entrée retourneront le même résultat, et ce quel que soit l'"ordre" des deux appels. Ces suppositions sont classiques dans le contexte de la gestion de services répartie (par exemple, dans (Abiteboul *et al.*, 2008)).

Le troisième point nécessite un traitement particulier, en particulier dans notre contexte des environnements pervasifs relationnels. Nous voulons faire ressortir l'impact des requêtes utilisant des binding patterns sur l'environnement. Par exemple, un message SMS ne peut pas être "annulé" une fois envoyé, alors que lire une température sur un capteur n'a pas d'impact significatif.

Nous proposons la notion d'**ensemble d'actions** d'une requête q sur un environnement pervasif relationnel p, dénoté $Actions_q(p, \tau_i)$, pour représenter l'impact de cette requête sur l'environnement. Une action est un triplet $\langle \psi, s, t \rangle$, avec un prototype actif ψ, une référence de service s et un tuple t sur $Input_\psi$. L'ensemble d'actions de q à l'instant τ_i comprend l'ensemble des invocations de prototypes actifs réalisées à τ_i, ainsi que l'ensemble des abonnements à des prototypes actifs en cours à τ_i. Pour une requête ponctuelle q, l'ensemble d'actions est défini pour τ_{start_q}. Pour une requête continue q, il est défini pour tout $\tau_i \geq \tau_{start_q}$.

L'**équivalence de deux requêtes** peut maintenant être défini grâce à cette notion d'ensemble d'actions. Soit P un schéma d'environnement pervasif. Deux requêtes ponctuelles q_1 et q_2 sur P, associée au même instant de début τ_{start}, sont équivalente si et seulement si pour tout environnement pervasif relationnel p sur P, on a $q_1(p) = q_2(p)$ **et** $Actions_{q_1}(p, \tau_{start}) = Actions_{q_2}(p, \tau_{start})$. De manière similaire, deux requêtes continues q_1 et q_2 sur P, associée au même instant de début τ_{start}, sont équivalente si et seulement si pour tout environnement pervasif relationnel p sur P, on a $q_1(p) = q_2(p)$ **et** $\forall \tau_i \geq \tau_{start}, Actions_{q_1}(p, \tau_i) = Actions_{q_2}(p, \tau_i)$.

À noter que seuls les *binding patterns* actifs contribuent à l'ensemble d'actions d'une requête. Deux requêtes sont donc équivalentes si, en plus de produire la même X-Relation/XD-Relation, elles impliquent les mêmes invocations et abonnements de *binding patterns* actifs, bien qu'elles puissent impliquer différentes invocations et abonnements de *binding patterns* passifs.

4. Vers un système de gestion d'environnements pervasifs

Afin de valider notre approche et réaliser des expérimentations, nous avons conçu et développé un prototype de système de gestion d'environnement pervasif, ou PEMS (*Pervasive Environment Management System*). Le rôle d'un PEMS est de gérer un environnement pervasif relationnel, avec ses sources de données dynamiques et son ensemble de services, et d'exécuter des requêtes ponctuelles et continues sur cet environnement. Le scénario de surveillance de température a été expérimenté sur notre prototype de PEMS.

4.1. *Implémentation du PEMS*

4.1.1. *Architecture du PEMS*

Nous avons conçu une architecture modulaire répartie pour répondre aux besoins du PEMS. Trois modules principaux sont déployés sur un *PEMS Core* pour la gestion globale de l'environnement pervasif relationnel. Des modules répartis sur des *PEMS Peers* prennent en charge localement les fonctionnalités de ces équipements. Une interface graphique (*PEMS GUI*) est également présente afin de simplifier les interactions avec les utilisateurs depuis un client distant (*PEMS Client*).

Le gestionnaire des ressources de l'environnement (*Environment Resource Manager*) est responsable des problématiques réseau concernant la découverte de services et les interactions distantes avec ces services. Il communique avec les gestionnaires locaux de ressources (*Local Environment Resource Manager*) déployés sur divers équipements répartis dans le réseau, afin que les fonctionnalités locales soient accessibles de manière transparente à travers le gestionnaire principale.

Le gestionnaire de tables étendues (*Extended Table Manager*) construit la représentation de l'environnement pervasif en environnement pervasif relationnel. Il gère le catalogue de XD-Relations et permet la récupération et la manipulation de leurs données et de leurs *binding patterns*. Il construit également une vue plus abstraite de l'ensemble des services de l'environnement. Il utilise le gestionnaire des ressources de l'environnement comme intermédiaire pour les interactions avec les services.

Le moteur de requête (*Query Processor*) permet d'exécuter de requêtes ponctuelles sur des XD-Relations, d'enregistrer des requêtes continues sur des XD-Relations et de les exécuter en temps réel. Il utilise le gestionnaire de tables étendues pour accéder au contenu des XD-Relations et utiliser leurs *binding patterns* d'invocation et d'abonnement. Il permet également l'exécution de requêtes continues de découverte de services.

Ces trois modules principaux, avec les modules répartis, remplissent les fonctions nécessaires à la gestion d'un environnement pervasif relationnel.

4.1.2. *Prototype*

L'architecture du PEMS est implémentée dans un prototype, appelé simplement le PEMS. Ce prototype a été développé en Java en utilisant le *framework* OSGi et le protocole UPnP pour les aspects réseau. Le *framework* OSGi facilite le développement d'applications modulaires et dynamiques. Chaque module est implémenté par un *bundle* OSGi qui peut être déployé, démarré et arrêté indépendamment et sans effort. La technologie UPnP (*Universal Plug and Play*) est adaptée aux environnement dynamiques et répartis en permettant de découvrir dynamiquement des services distants, et d'interagir avec ces services par des appels de méthodes ou des abonnements à des évènements. De plus, il existe une implémentation de UPnP pour OSGi qui permet d'intégrer complètement la notion de service UPnP avec la notion de service OSGi au sein du *framework*.

Notre prototype utilise également la technologie JavaCC pour implémenter des analyseurs syntaxiques (*parsers*). Il peut ainsi compiler et exécuter un DDL et un DML pour gérer les XD-Relations, et un langage de type SQL pour les requêtes ponctuelles et continues. Il peut également comprendre directement des expressions algébriques pour les requêtes.

Bundles *(sauf interfaces graphiques et services)*	6
Packages	31
Classes / Interfaces	161 / 16
Méthodes	874
Total de lignes de code	12254

4.2. *Expérimentations*

En utilisant le prototype de PEMS que nous avons développé, nous avons mis en place le scénario de surveillance de température décrit au long de cet article. Pour cela, nous avons d'une part développé les services nécessaires, d'autres part mis en place les XD-Relations de l'environnement pervasif relationnel. Nous avons ensuite pu testé l'exécution de différentes requêtes ponctuelles et continues sur cet environnement.

Pour le scénario de surveillance de température, trois types de services ont été développés : les capteurs de température, les services de messagerie et les services de caméras. Les services gérant des capteurs de température implémentent une propriété `location`, une méthode `getTemperature` et un flux `temperatureNotifications`. Les capteurs de température sont soit simulés pour permettre de fixer dynamiquement la température par une interface graphique, soit réels avec des *Thermochron iButton DS1921* et des *Sun SPOTs*. Les services de messagerie implémentent une propriété `protocol` et une méthode `sendMessage`, et permettent d'envoyer des messages par e-mail, SMS ou messagerie instantanée. Les services de caméras implémentent une propriété `location` et une méthode `takePhoto` permettant de prendre une photo à partir d'une webcam.

Les XD-Relations mises en place correspondent à l'environnement présenté dans l'exemple 3 : `contacts`, `sensors`, `cameras` ; avec une XD-Relation finie supplémentaire `TemperatureSurveillance` permettant de "configurer" la surveillance, en indiquant qui surveille quelle zone selon quel seuil d'alerte. À noter que le flux `temperatures` n'est lui pas utile, car il peut être dynamiquement généré à partir de la XD-Relation `sensors` en utilisant le *binding pattern* `temperatureNotifications`.

CONTACTS

`name`	`address`	`message`	`messenger`	`sent`
Xavier	xavier@liris-7171.insa-lyon.fr	*	*jabber*	*
Yann	yann.gripay@insa-lyon.fr	*	*email*	*
Zoé	33698######	*	*sms*	*

CAMERAS

camera	location	photo
webcam01	roof	*
webcam02	corridor	*
webcam03	office	*

SENSORS

sensor	location	temperature
sensor01	roof	*
sensor02	corridor	*
sensor03	office	*

TEMPERATURESURVEILLANCE

area	manager	threshold	alertMessage
roof	Xavier	45.0	Alert : roof on fire !
corridor	Yann	35.0	Do not run in the corridor !
office	Zoé	32.0	Too hot in the office...

Sur cet environnement, différentes requêtes peuvent être exprimées en “SQL”. Nous exprimons par exemple les requêtes ponctuelles Q_1 et Q_2 évoquées dans l’exemple 5.

```
// Q1
SELECT *
ONCE FROM contacts
WITH message := "Bonjour!"
WHERE name != "Zoé"
USING sendMessage;
```

```
// Q2
SELECT sensor, temperature
ONCE FROM sensors
WHERE location = "office"
USING getTemperature;
```

L’exécution de ces requêtes, outre la production des X-Relations attendus en sortie, entraînent bien l’appel des services impliqués. Pour Q_1, Xavier et Yann reçoivent le message “Bonjour !” respectivement par messagerie instantanée et par e-mail, et Zoé ne reçoit pas ce message. Pour Q_2, la température est bien récupérée, produisant (par exemple) la X-Relation suivante :

sensor	temperature
sensor03	23.5

Pour réaliser le scénario complet, c’est-à-dire la surveillance continue de température avec envoi de message d’alertes, une seule requête continue est nécessaire. Une XD-Relation infinie `TemperatureSurveillanceResults` est créée pour représenter la XD-Relation produite par la requête (XD-Relation infinie, car le requête utilise un opérateur de *streaming*). En combinant les XD-Relations `sensors`, `TemperatureSurveillance` et `contacts`, cette requête permet de s’abonner aux flux de température de capteurs, de vérifier si un seuil est dépassé, et la cas échéant d’envoyer le message d’alerte aux managers concernés.

```
CREATE STREAM TemperatureSurveillanceResults (
  area STRING,
  manager STRING,
  threshold REAL,
  temperature REAL,
  sent BOOLEAN
)
AS
  SELECT t.area,t.manager,t.threshold,s.temperature,c.sent
  STREAMING UPON insertion
  FROM sensors s, TemperatureSurveillance t, contacts c
  WITH c.message := t.alertMessage
  WHERE s.location = t.area
    AND s.temperature > t.threshold
    AND t.manager = c.name
  USING s.temperature [1], c.sendMessage
;
```

Lors de l'exécution de cette requête, il est possible de changer la température des capteurs simulés afin de valider le comportement de la requête. Par exemple, lorsque l'on fixe la température d'un capteur de la zone "office" à 34.0°C, le message d'alerte "Too hot in the office..." est effectivement envoyé à Zoé par SMS.

5. Conclusion

Les systèmes pervasifs posent de nouveaux défis à résoudre afin de bénéficier pleinement de leur potentiel : des interactions complexes, mais néanmoins maîtrisables, entre équipements hétérogènes fournissant des sources de données et fonctionnalités dynamiques. Les développements actuels basés sur un mélange *ad hoc* de langages de programmation impératifs, de langages de requêtes classiques et de protocoles réseau ne sont une solution ni pratique, ni adéquate sur le long terme. Les approches déclaratives offrent l'avantage de fournir une vue logique des ressources qui abstrait les problématiques d'accès physique et permet la mise en œuvre de techniques d'optimisation. C'est pourquoi la définition déclarative de requêtes sur des sources de données et des fonctionnalités est reconnue comme un défi majeur dans le but de simplifier le développement d'applications pervasives. Cette approche nécessite néanmoins une compréhension claire des interactions entre données, flux de données et services.

Dans cet article, nous avons proposé un framework définissant une vue orientée données des environnements pervasifs, l'environnement pervasif relationnel, qui intègre les sources de données à la fois conventionnelles et non-conventionnelles. L'environnement pervasif relationnel permet le développement d'applications pervasives de manière déclarative en utilisant des requêtes continues orientées service qui combinent ces sources de données.

Nous avons ainsi proposé un modèle de données pour les environnement pervasifs, appelé SoCQ (pour Service-oriented Continuous Query), qui prend en compte leur hétérogénéité, dynamicité et répartition. Nous avons défini la *structure* de notre modèle de données avec la notion de relation dynamique étendue (eXtended Dynamic Relation, ou XD-Relation) représentant les sources de données. Nous avons également défini un *langage* algébrique pour notre modèle de données avec l'algèbre Séréna, à

partir de laquelle un langage de type SQL a été défini. Ce langage permet d'exprimer de manière déclarative des requêtes sur les environnements pervasifs.

Enfin, nous avons implémenté ce framework au sein d'un prototype de système de gestion d'environnements pervasifs (Pervasive Environment Management System, ou PEMS) qui prend en charge notre modèle de données. Un PEMS est un système de gestion dynamique de données et de services qui gère de manière transparente les problématiques liées au réseau telles que la découverte de services et les interactions à distance. Il supporte l'exécution de requêtes ponctuelles et continues orientées service que les développeurs d'applications peuvent aisément concevoir pour développer des applications pervasives.

6. Bibliographie

Abiteboul S., Manolescu I., Zoupanos S., « OptimAX : Optimizing Distributed ActiveXML Applications », *ICWE2008, 8th International Conference on Web Engineering*, 2008.

Arasu A., Babcock B., Babu S., Datar M., Ito K., Motwani R., Nishizawa I., Srivastava U., Thomas D., Varma R., Widom J., « STREAM : The Stanford Stream Data Manager », *IEEE Data Engineering Bulletin*, vol. 26, n° 1, p. 19-26, 2003.

Codd E. F., « Relational completeness of data base sublanguages », *Database Systems*, Prentice-Hall, p. 65-98, 1972.

Florescu D., Levy A., Manolescu I., Suciu D., « Query Optimization in the Presence of Limited Access Patterns », *SIGMOD'99 : Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, p. 311-322, 1999.

Gripay Y., « Service-oriented Continuous Queries for Pervasive Systems », *EDBT 2008 PhD Workshop (unoffical proceedings)*, 2008.

Gripay Y., La L.-Q., Laforest F., Petit J.-M., « SoCQ : un système de gestion de données et de services en environnement pervasif », *BDA'09, 25èmes journées Bases de Données Avancées*, 2009a.

Gripay Y., Laforest F., Petit J.-M., « Towards Action-Oriented Continuous Queries in Pervasive Systems », *BDA'07, 23èmes journées Bases de Données Avancées*, p. 1-20, 2007.

Gripay Y., Laforest F., Petit J.-M., « Vers une algèbre relationnelle étendue aux services », *BDA'08, 24èmes journées Bases de Données Avancées*, p. 1-20, 2008.

Gripay Y., Laforest F., Petit J.-M., « SoCQ : A Framework for Pervasive Environments », *IS-PAN 2009, 10th International Symposium on Pervasive Systems, Algorithms and Networks*, 2009b.

Gripay Y., Laforest F., Petit J.-M., « SoCQ : a Pervasive Environment Management System », *UbiMob'09, 5èmes journées Francophones Mobilité et Ubiquité*, 2009c.

Levene M., Loizou G., *A Guided Tour of Relational Databases and Beyond*, Springer-Verlag, 1999.

Maier D., Ullman J. D., Vardi M. Y., « On the foundations of the universal relation model », *ACM Transactions on Database Systems (TODS)*, vol. 9, n° 2, p. 283-308, 1984.

B

Poster de démonstration

SoCQ: un système de gestion de données et de services en environnement pervasif

Le modèle de données SoCQ

Yann Gripay, Le-Quyen La, Frédérique Laforest, Jean-Marc Petit
Laboratoire d'InfoRmatique en Image et Systèmes d'information
UMR5205 CNRS / Université de Lyon / INSA de Lyon
INSA de Lyon - Bâtiment Blaise Pascal, 7 avenue Jean Capelle - 69621 Villeurbanne Cedex, France
`http://liris.cnrs.fr` – prenom.nom@liris.cnrs.fr

Objectifs

- Simplification du développement d'applications en environnement pervasif
- Approche déclarative : extension des principes des bases de données
- Modèle de données **SoCQ** : ***S**ervice-**o**riented **C**ontinuous **Q**uery*

Représentation des fonctionnalités

- **Prototype** : représentation abstraite d'une fonctionnalité de l'environnement pervasif sous forme de signature de fonction
  - **Méthodes** : envoyer un message, prendre une photo, ouvrir une porte automatique...
  - **Flux de données** : mesures de capteurs en continu (température, luminosité, accélération), flux RSS...
- **Service** : entité distante, liée à des dispositifs physiques et/ou logiques, implémentant un ou plusieurs prototypes de fonctionnalité

Services et prototypes

- Prototypes de fonctionnalité :
  - Méthode `sendMessage( address,text ) : ( sent )`
  - Méthode `takePhoto( ) : ( photo )`
  - Flux `temperatures( ) : ( temperature )`

Intégration données / services

- **XD-Relation** : relation dynamique étendue
  - **Attributs réels** (avec données) ou **virtuels** (sans données)
  - **Binding patterns**
    - prototype de fonctionnalité (méthode ou flux de données)
    - attribut référençant des services
    - lien entre attributs et entrées/sorties du prototype
  - Relation dynamique **finie** (avec mise à jour) ou **infinie** (flux)
- **Environnement pervasif relationnel**
  - Ensemble de XD-Relations
  - Extension des bases de données classiques

XD-Relations

```
RELATION employees (
  name       STRING,
  address    STRING,
  messenger  SERVICE,
  text       STRING   VIRTUAL,
  sent       BOOLEAN  VIRTUAL
)
USING BINDING PATTERNS (
  sendMessage[messenger] ( address, text ) : ( sent )
);
```

name	address	messenger	*text*	*sent*
Nicolas	nicolas@elysee.fr	*email1*	*	*
Carla	carla@elysee.fr	*email1*	*	*
François	francois@im.gouv.fr	*jabber3*	*	*

```
RELATION temperatureSensors (
  sensor       SERVICE,
  area         STRING,
  temperature  REAL     VIRTUAL
)
USING BINDING PATTERNS (
  getTemperature[sensor] ( ) : ( temperature ),
  temperatures[sensor] ( ) : ( temperature ) STREAMING
);
```

sensor	area	*temperature*
sensor1	office	*
sensor3	roof	*
sensor17	corridor	*

Requêtes déclaratives

- **Algèbre Séréna** sur les XD-Relations
  - Ensemblistes : union $\cup$, intersection $\cap$, différence $-$
  - Relationnels : sélection σ_C, projection π_X, jointure naturelle $\bowtie$
  - Dynamiques : fenêtrage $\mathcal{W}_{[size]}$, *streaming* $\mathcal{S}_{[event]}$
  - De réalisation : **affectation** d'une valeur à un attribut virtuel $\alpha_{A \equiv B/c}$, utilisation d'un **binding pattern** $\beta_{\langle\phi,\mathcal{S}\rangle}$
  - De découverte (dynamique) de services $\xi_{A,\psi,\psi'}$
- **Serena SQL**
  - Requêtes ponctuelles ou continues sur des XD-Relations
  - Requêtes de découverte de services

Requête SoCQ

```
SELECT surveillance.area, surveillance.manager, employees.sent
FROM   temperatureNotifications [1], employees, surveillance
WITH   employees.text := surveillance.alertMessage
WHERE  surveillance.manager = employees.name
  AND  surveillance.area = temperatureNotifications.area
  AND  surveillance.threshold < temperatureNotifications.temperature
USING  employees.sendMessage
```

$$\pi_{area,manager,sent}\Big(\beta_{\langle sendMessage,messenger\rangle}\Big(\alpha_{text \equiv alertMessage}\Big(employees \bowtie \sigma_{threshold<temperature}\big(surveillance \bowtie \mathcal{W}_{[1]}(temperatureNotifications)\big)\Big)\Big)\Big)$$

Intérêts du modèle de données SoCQ

- Approche déclarative face à la complexité des environnements pervasifs
- Développement simplifié d'applications "pervasives" avec le Serena SQL
- Gestion dynamique : découverte de services, appels de méthodes, abonnements aux flux de données...

SoCQ: un système de gestion de données et de services en environnement pervasif

Démonstration du Système de Gestion d'Environnement Pervasif (PEMS)

Yann Gripay, Le-Quyen La, Frédérique Laforest, Jean-Marc Petit
Laboratoire d'InfoRmatique en Image et Systèmes d'information
UMR5205 CNRS / Université de Lyon / INSA de Lyon
INSA de Lyon - Bâtiment Blaise Pascal, 7 avenue Jean Capelle - 69621 Villeurbanne Cedex, France
`http://liris.cnrs.fr` – prenom.nom@liris.cnrs.fr

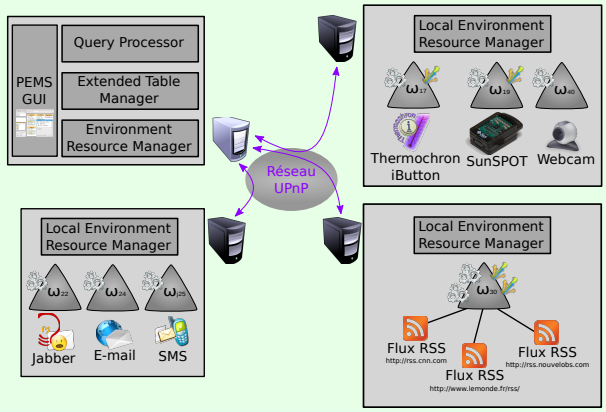
Démonstration

- Mise en œuvre de l'environnement pervasif relationnel
- Développement d'applications pervasives sous forme de requêtes SoCQ
- Test de scénarios impliquant différents services et sources de données

Implémentation du prototype

- Architecture modulaire du PEMS
  - *Environment Resource Manager* réparti pour la découverte et l'interaction à distance avec les services (méthodes et flux)
  - *Extended Table Manager* pour gérer les XD-Relations
  - *Query Processor* pour l'exécution des requêtes SoCQ
  - *PEMS GUI* pour contrôler le PEMS
- Services pour dispositifs logiques et physiques
  - Messageries : e-mail, Jabber, SMS
  - Capteurs de température : Thermochron, SunSpot
  - Caméras prenant des photos (webcams)
  - Flux RSS provenant de divers sites d'information
- Technologies utilisées
  - Framework Java/OSGi
  - Protocole UPnP

PEMS et services



Scénario de surveillance de température

- XD-Relations
  - `TemperatureServices` – capteurs de température
  - `Employees` – liste des personnes
    - *Construite à partir de* `MessageServices` *et* `EmployeeList`
  - `TemperatureSurveillance` – "configuration" du système
- Requête principale
  - Surveillance des flux de température dans différentes zones
  - Envoi de messages d'alerte aux managers d'une zone quand le seuil est dépassé dans cette zone

Scénario : découverte de services

- Découverte dynamique des capteurs de température

```
CREATE RELATION TemperatureServices (
  Sensor       SERVICE,
  Location     STRING,
  Temperature  REAL     VIRTUAL
)
USING (
  getTemperature [ Sensor ] ( ) : ( Temperature ),
  temperatures [ Sensor ] ( ) : ( Temperature ) STREAMING
)
AS
  DISCOVER SERVICES PROVIDING
    PROPERTY Location STRING,
    METHOD getTemperature ( ) : ( REAL ),
    STREAM temperatures ( ) : ( REAL ) ;
```

Sensor	Location	*Temperature*
sensor1	office	*
sensor2	roof	*
sensor3	corridor	*
sensor17	roof	*
sensor19	office	*

Scénario : XD-Relations simples

```
RELATION Employees (
  Name       STRING,
  Address    STRING,
  Messenger  SERVICE,
  Message    STRING   VIRTUAL,
  Sent       BOOLEAN  VIRTUAL
)
USING (
  sendMessage[Messenger](Address,Message):(Sent)
);
```

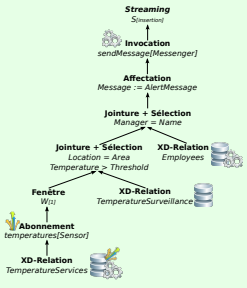
Name	Address	Messenger	*Message*	*Sent*
Xavier	xavier@localhost	*jabber22*	*	*
Yann	yann.gripay@insa-lyon.fr	*email24*	*	*
Zoe	33698######	*sms25*	*	*

```
RELATION TemperatureSurveillance (
  Area           STRING,
  Manager        STRING,
  Threshold      REAL,
  AlertMessage   STRING
);
```

Area	Manager	Threshold	AlertMessage
roof	Xavier	49.0	Alert : roof on fire !
corridor	Yann	35.0	Do not run in the corridor !
office	Zoe	32.0	Too hot in the office. . .

Scénario : requête principale

```
SELECT s.Area, Manager, Threshold,
       Temperature, Sent
STREAMING UPON insertion
FROM   TemperatureServices t,
       TemperatureSurveillance s,
       Employees
WITH   Message := AlertMessage
WHERE  t.Location = s.Area
  AND  Temperature > Threshold
  AND  Manager = Name
USING  temperatures [1], sendMessage
```



Intérêts du PEMS

- Gestion dynamique de données, de flux de données et de services
- Modélisation pratique de l'environnement sous forme de XD-Relations
- Développement simplifié d'applications sous forme de requêtes déclaratives en Serena SQL

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR 2005, Proceedings of Second Biennial Conference on Innovative Data Systems Research*, 2005.

[ABB+03] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[ABM04] Serge Abiteboul, Omar Benjelloun, and Tova Milo. Positive Active XML. In *PODS '04: Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 35–45, New York, NY, USA, 2004. ACM.

[Act] ActiveXML. `http://www.activexml.net/`.

[AHS06] Karl Aberer, Manfred Hauswirth, and Ali Salehi. A middleware for fast and flexible sensor network deployment. In *VLDB 2006, Proceedings of the 32th International Conference on Very Large Data Bases*, 2006.

[AHS07] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *MDM 2007, Proceedings of the 8th International Conference on Mobile Data Management*, 2007.

[AMT06] Serge Abiteboul, Ioana Manolescu, and Emanuel Taropa. A Framework for Distributed XML Data Management. In *EDBT 2006, 10th International Conference on Extending Database Technology*, pages 1049–1058, 2006.

[AMZ08] Serge Abiteboul, Ioana Manolescu, and Spyros Zoupanos. OptimAX: Optimizing Distributed ActiveXML Applications. In *ICWE2008, 8th International Conference on Web Engineering*, 2008.

[ATT] ATT Laboratories, Cambridge. Sentient Computing Project. http://www.cl.cam.ac.uk/research/dtg/attarchive/spirit/.

[BBG+09] Michel Bauderon, Christophe Bobineau, Stephane Grumbach, Antoine Henry, Xin Qi, Wenwu Qu, Kun Suo, Fang Wang, and Zhilin Wu. Netquest: An Abstract Model for Pervasive Applications. In *Pervasive 2009, Proceedings of the 7th International Conference on Pervasive Computing*, 2009.

[BC07] Gregory Biegel and Vinny Cahill. *Requirements for middleware for pervasive information systems*, pages 86–102. Volume 10 of Kourouthanassis and Giaglis [KG07a], 2007.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In *MDM 2001, Proceedings of the Second International Conference on Mobile Data Management*, pages 3–14, 2001.

[BMK+00] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven Shafer. EasyLiving: Technologies for intelligent environments. In *HUC 2000, Proceedings of the Second International Symposium on Handheld and Ubiquitous Computing*, pages 12–29, 2000.

[Car] Carnegie Mellon University. Project Aura, Distraction-free Ubiquitous Computing. http://www.cs.cmu.edu/~aura/.

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003, Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshankar Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR 2003, Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 379–390, 2000.

[Cod72] Edgar F. Codd. Relational completeness of data base sublanguages. In *Database Systems*, pages 65–98. Prentice-Hall, 1972.

[CS93] Surajit Chaudhuri and Kyuseok Shim. Query Optimization in the Presence of Foreign Functions. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 529–542, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[CS99] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, 1999.

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

[dCYG08] Cristiano André da Costa, Adenauer Corrêa Yamin, and Cláudio Fernando Resin Geyer. Toward a General Software Infrastructure for Ubiquitous Computing. *IEEE Pervasive Computing*, 7(1):64–73, 2008.

[dDCK+06] Scott de Deugd, Randy Carroll, Kevin E. Kelly, Bill Millett, and Jeffrey Ricker. SODA: Service Oriented Device Architecture. *IEEE Pervasive Computing*, 5(3):94–96, 2006.

[DF05] Matthew Denny and Michael J. Franklin. Predicate result range caching for continuous queries. In *SIGMOD'05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 646–657, New York, NY, USA, 2005. ACM.

[DF06] Matthew Denny and Michael J. Franklin. Operators for Expensive Functions in Continuous Queries. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 147, Washington, DC, USA, 2006. IEEE Computer Society.

[DGH+06] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards Expressive Publish/Subscribe Systems. In *EDBT 2006, 10th International Conference on Extending Database Technology*, pages 627–644, 2006.

[DGP+07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A General Purpose Event Monitoring System. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*, pages 412–422, 2007.

[DR04] Luping Ding and Elke A. Rundensteiner. Evaluating Window Joins over Punctuated Streams. In *CIKM'04, Proceedings of the 13th ACM international conference on Information and Knowledge Management*, pages 98–107, 2004.

[ECPS02] Deborah Estrin, David Culler, Kris Pister, and Gaurav Sukhatme. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.

[Fel] Apache Felix Project. `http://felix.apache.org/`.

[FHM05] Michael Franklin, Alon Halevy, and David Maier. From Databases to Dataspaces: a new Abstraction for Information Management. *SIGMOD Rec.*, 34(4):27–33, 2005.

[FJK+05] Michael J. Franklin, Shawn R. Jeffery, Sailesh Krishnamurthy, Fredrick Reiss, Shariq Rizvi, Eugene Wu, Owen Cooper, Anil Edakkunni, and Wei Hong. Design Considerations for High Fan-In Systems: The HiFi Approach. In *CIDR 2005, Proceedings of Second Biennial Conference on Innovative Data Systems Research*, 2005.

[FLMS99] Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query Optimization in the Presence of Limited Access Patterns. In *SIGMOD'99: Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 311–322, 1999.

[GdB94] Paul W. P. J. Grefen and Rolf A. de By. A multi-set extended relational algebra - a formal approach to a practical issue. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 80–88, Washington, DC, USA, 1994. IEEE Computer Society.

[GDL+04] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System Support for Pervasive Applications. *ACM Transactions on Computer Systems*, 22(4):421–486, November 2004.

[GFS02] Krzysztof Gajos, Harold Fox, and Howard Shrobe. End user empowerment in human centered pervasive computing. In *Pervasive 2002*, Zurich, Switzerland, 2002.

[Gla] GlassFish - Open Source Application Server. `http://glassfish.dev.java.net/`.

[GLLP09] Yann Gripay, Le-Quyen La, Frédérique Laforest, and Jean-Marc Petit. SoCQ: un système de gestion de données et de services en environnement pervasif. In *BDA'09, 25èmes journées Bases de Données Avancées*, 2009.

[GLP07] Yann Gripay, Frédérique Laforest, and Jean-Marc Petit. Towards Action-Oriented Continuous Queries in Pervasive Systems. In *BDA'07, 23èmes journées Bases de Données Avancées*, pages 1–20, 2007.

[GLP08] Yann Gripay, Frédérique Laforest, and Jean-Marc Petit. Vers une algèbre relationnelle étendue aux services. In *BDA'08, 24èmes journées Bases de Données Avancées*, pages 1–20, 2008.

[GLP09a] Yann Gripay, Frédérique Laforest, and Jean-Marc Petit. SoCQ: A Framework for Pervasive Environments. In *ISPAN 2009, 10th International Symposium on Pervasive Systems, Algorithms and Networks*, 2009.

[GLP09b] Yann Gripay, Frédérique Laforest, and Jean-Marc Petit. SoCQ: a Pervasive Environment Management System. In *UbiMob'09, 5èmes journées Francophones Mobilité et Ubiquité*, 2009.

[GM04] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *IEEE Pervasive Computing*, 3(1):46–55, Jan.-March 2004.

[GMWU99] Hector Garcia-Molina, Jennifer Widom, and Jeffrey D. Ullman. *Database System Implementation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[Gri08] Yann Gripay. Service-oriented Continuous Queries for Pervasive Systems. In *EDBT 2008 PhD Workshop (unoffical proceedings)*, 2008.

[GSSS02] David Garlan, Daniel P. Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Toward Distraction-Free Pervasive Computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

[GW00] Roy Goldman and Jennifer Widom. WSQ/DSQ: A Practical Approach for Combined Querying of Databases and the Web. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 285–296, 2000.

[Hel98] Joseph M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 23(2):113–157, 1998.

[HMEZ+05] Sumi Helal, William Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura, and Erwin Jansen. The gator tech smart house: A programmable pervasive space. *Computer*, 38(3):50–60, 2005.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD'93, Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 267–276, 1993.

[HXCZ07] Jeong-Hyon Hwang, Ying Xing, Ugur Cetintemel, and Stan Zdonik. A Cooperative, Self-Configuring High-Availability Solution for Stream Processing. In *ICDE'07, Proceedings of the 23rd International Conference on Data Engineering*, 2007.

[IN02] Tomasz Imielinski and Badri Nath. Wireless graffiti: data, data everywhere. In *VLDB'2002: Proceedings of the 28th international conference on Very Large Data Bases*, pages 9–19, 2002.

[JAF+06] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. Declarative Support for Sensor Data Cleaning. In *Pervasive*, pages 83–100, 2006.

[Java] Java: Java.sun.com - The Source for Java Developers. `http://java.sun.com/`.

[Javb] JavaCC: Java Compiler Compiler - The Java Parser Generator. `http://javacc.dev.java.net/`.

[JMX] JMX: Java Management Extensions (JMX) Technology. `http://java.sun.com/javase/technologies/`.

[KG07a] Panos E. Kourouthanassis and George M. Giaglis, editors. *Pervasive Information Systems*, volume 10 of *Advances in Management Information Systems*. M.E. Sharpe, Armonk, NY, 2007.

[KG07b] Panos E. Kourouthanassis and George M. Giaglis. *Toward pervasiveness*, pages 3–25. Volume 10 of *Advances in Management Information Systems* [KG07a], 2007.

[KTD+03] Kimberle Koile, Konrad Tollmar, David Demirdjian, Howard Shrobe, and Trevor Darrell. Activity zones for context-aware computing. In *Proceedings of UbiComp 2003*, pages 90–106. Springer-Verlag, 2003.

[LL99] Mark Levene and George Loizou. *A Guided Tour of Relational Databases and Beyond*. Springer-Verlag, 1999.

[Mic] Microsoft Research. The EasyLiving Project. `http://research.microsoft.com/easyliving/`.

[MIT] MIT. Oxygen Project, Pervasive, Human-centered Computing. `http://oxygen.csail.mit.edu/`.

[MUV84] David Maier, Jeffrey D. Ullman, and Moshe Y. Vardi. On the foundations of the universal relation model. *ACM Transactions on Database Systems (TODS)*, 9(2):283–308, 1984.

[OMG] OMG. CORBA. `http://www.corba.org/`.

[OSG] OSGi: The Dynamic Module System for Java. `http://www.osgi.org/`.

[RAMB+05] Anand Ranganathan, Jalal Al-Muhtadi, Jacob Biehl, Brian Ziebart, Roy H. Campbell, and Brian Bailey. Towards a pervasive computing benchmark. In *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 194–198, Washington, DC, USA, 2005. IEEE Computer Society.

[RC08] Anand Ranganathan and Roy H. Campbell. Provably correct pervasive computing environments. In *PerCom'08, Proceedings of IEEE International Conference on Pervasive Computing and Communications*, pages 160–169, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[SLp03] Thomas Strang and Claudia Linnhoff-popien. Service interoperability on context level in ubiquitous computing environments. In *SSGRR2003w, Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[SM03] Debashis Saha and Amitava Mukherjee. Pervasive Computing: A Paradigm for the 21st Century. *Computer*, 36(3):25–31, 2003.

[SMWM06] Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Query Optimization over Web Services. In *VLDB 2006, Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 355–366, 2006.

[SOA] SOA4D: Service-Oriented Architecture for Devices - Welcome to the SOA4D Forge. `http://www.soa4d.org/`.

[SOD] DPWS: Devices Profile for Web Services - Welcome to the ITEA SODA project. `http://www.soda-itea.org/`.

[SPP+03] U. Saif, H. Pham, J. M. Paluska, J. Waterman, C. Terman, and S. Ward. A case for goal-oriented programming semantics. In *UbiSys'03: Workshop on System Support for Ubiquitous Computing, 5th International Conference on Ubiquitous Computing (UbiComp 2003)*, 2003.

[Unia] University of California, Berkeley. The Endeavour Expedition: Charting the Fluid Information Utility. `http://endeavour.cs.berkeley.edu/`.

[Unib] University of Washington. Portolano: An Expedition into Invisible Computing. `http://portolano.cs.washington.edu/`.

[Uni05] International Telecommunication Union. *The Internet of Things*. ITU Internet Reports. International Telecommunication Union, 2005.

[UPn] UPnP: Universal Plug and Play - Welcome to UPnP Forum. `http://www.upnp.org/`.

[Wei91] Mark Weiser. The Computer for the 21st Century. *Scientific American*, 265(3):94–104, September 1991.

[XL05] Wenwei Xue and Qiong Luo. Action-Oriented Query Processing for Pervasive Computing. In *CIDR 2005, Proceedings of the Second Biennial Conference on Innovative Data Systems Research*, 2005.

[YG03] Yong Yao and Johannes Gehrke. Query Processing in Sensor Networks. In *CIDR 2003, Proceedings of the First Biennial Conference on Innovative Data Systems Research*, 2003.

[ZMN05] Fen Zhu, Matt W. Mutka, and Lionel M. Ni. Service Discovery in Pervasive Computing Environments. *IEEE Pervasive Computing*, 4(4):81–90, 2005.