Selective Data Replication for Distributed Geographical Data Sets

A thesis submitted in partial fulfilment of the

requirements for the

Degree of Master of Science

October 2008

Xuan Gu
Department of Computer Science & Software Engineering
University of Canterbury, Christchurch, New Zealand
mark.guxuan@gmail.com
Abstract

The main purpose of this research is to incorporate additional higher-level semantics into the existing data replication strategies in such a way that their flexibility and performance can be improved in favour of both data providers and consumers. The resulting approach from this research is referred to as the selective data replication system. With this system, the data that has been updated by a data provider is captured and batched into messages known as update notifications. Once update notifications are received by data consumers, they are used to evaluate so-called update policies, which are specified by data consumers containing details on when data replications need to occur and what data needs to be updated during the replications.

Supervisors:

Former Supervisor: Dr. Richard Pascoe
Christchurch, New Zealand

Supervisor: Dr. Warwick Irwin
Department of Computer Science & Software Engineering
University of Canterbury, Christchurch, New Zealand
warwick.irwin@canterbury.ac.nz

Associate Supervisor: Dr. Neville Churcher
Department of Computer Science & Software Engineering
University of Canterbury, Christchurch, New Zealand
neville.churcher@canterbury.ac.nz
Contents

1 Introduction

2 The Problems

2.1 Background on Data Replication

2.1.1 Data Fragmentation

2.1.2 Replication Strategy

2.2 Research Environment

2.2.1 Multi-Organization Environment

2.2.2 Geographical Information System Environment

2.3 Research Problems

2.3.1 Unawareness

2.3.2 Performance

2.3.3 Heterogeneity

2.4 Case Study

2.4.1 “Post Office” Scenario

2.4.2 “Real Estate” Scenario

2.4.3 “Disaster Control” Scenario

2.5 Research Objectives

3 The Solutions

3.1 Solution to Unawareness

3.1.1 Peer-to-Peer Communication Model

3.1.2 Mediator Communication Model

3.1.3 Research Solution

3.2 Solution to Performance

3.3 Solution to Heterogeneity

3.3.1 Ontologies

3.3.2 Sheth & Larson’s Five-Level Schema Framework

3.3.3 Elements of the Model Driven Architecture

3.3.4 Research Solution

3.4 System Architecture

4 Update Notification

4.1 Who should provide update notifications?

4.2 What does an update notification look like?

4.2.1 Administering Update Notifications

4.2.2 Describing Updates
4.3 How are consumers notified with updates? .................................................. 35
   4.3.1 The “Push” Method ............................................................................ 35
   4.3.2 The “Pull” Method ............................................................................ 36
   4.3.3 Using Both “Push” & “Pull” ............................................................... 37
4.4 How are update notifications managed? ..................................................... 37
   4.4.1 Consumer Side Notification Management ........................................ 38
   4.4.2 Provider Side Notification Management ........................................... 40
4.5 Summary .................................................................................................... 40

5 Update Policy .................................................................................................. 41
   5.1 Update Policy Specification .................................................................... 41
      5.1.1 Ontology-Based Policy Specification .............................................. 42
      5.1.2 Choice of Rule Languages ............................................................... 43
      5.1.3 Update Policy Specification using SWRL ....................................... 44
   5.2 Update Policy Evaluation ........................................................................ 47
   5.3 Summary .................................................................................................... 49

6 Implementation ............................................................................................... 51
   6.1 Data Provider Implementation .................................................................. 52
      6.1.1 Producing the Export PIM Ontology ............................................... 52
      6.1.2 Generating Update Notifications ................................................... 53
      6.1.3 Constructing the Web Service Interface .......................................... 56
   6.2 Data Consumer Implementation ............................................................... 59
      6.2.1 Managing Update Notifications .................................................... 59
      6.2.2 Evaluating Update Policy ................................................................ 61
      6.2.3 Replicating Data ............................................................................. 64

7 Evaluation ...................................................................................................... 65
   7.1 Flexibility ................................................................................................... 65
   7.2 Performance .............................................................................................. 66
      7.2.1 High-Level Task-Breakdown Analysis ............................................ 66
      7.2.2 Test Environment Preparation ....................................................... 67
      7.2.3 Statistics Collection Method ........................................................... 68
      7.2.4 Simulating “Post Office” ................................................................. 68
      7.2.5 Simulating “Real Estate” ................................................................. 70
      7.2.6 Summary ......................................................................................... 72

8 Future Work .................................................................................................... 75
   8.1 Enhancing the Current Implementation .................................................. 75
      8.1.1 Better Support Data & Ontology Translation .................................... 75
      8.1.2 Support Data Integration ................................................................. 75
      8.1.3 Support Multiple Identification Fields ............................................. 76
      8.1.4 Support Multiple Policy Specification Languages ......................... 76
      8.1.5 Automate Export PIM Ontology Generation .................................... 76
      8.1.6 Support Ontology Update Replication ............................................ 76
      8.1.7 Introduce Semantic Web Services ..................................................... 77
   8.2 Migrating onto an Agent Platform ........................................................... 77
      8.2.1 Introduce Software Agents .............................................................. 77
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2.2</td>
<td>Support Collaborative Agent Behaviour</td>
<td>77</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>79</td>
</tr>
<tr>
<td>A</td>
<td>Ontology Translation</td>
<td>87</td>
</tr>
<tr>
<td>A.1</td>
<td>Installation of Required Protége Plugins</td>
<td>87</td>
</tr>
<tr>
<td>A.2</td>
<td>Ontology Translation using Protége User Interface</td>
<td>88</td>
</tr>
<tr>
<td>A.3</td>
<td>Ontology Translation using Protége APIs</td>
<td>90</td>
</tr>
<tr>
<td>A.4</td>
<td>Example Input &amp; Output Files</td>
<td>91</td>
</tr>
<tr>
<td>A.4.1</td>
<td>UML Class Diagram</td>
<td>91</td>
</tr>
<tr>
<td>A.4.2</td>
<td>Input XMI Ontology Document</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>Output OWL Ontology Document</td>
<td>97</td>
</tr>
<tr>
<td>B</td>
<td>Source Code</td>
<td>101</td>
</tr>
<tr>
<td>B.1</td>
<td>Source Code for Ontology Translation using Protége APIs</td>
<td>101</td>
</tr>
<tr>
<td>B.2</td>
<td>Source Code for Update Notification Retrieval</td>
<td>106</td>
</tr>
<tr>
<td>C</td>
<td>SWRL Built-Ins for Geospatial Operations</td>
<td>111</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.1</td>
<td>The four most basic replication strategy categories.</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Multiple views of geographical data sets.</td>
<td>6</td>
</tr>
<tr>
<td>3.1</td>
<td>Peer-to-peer communications between data providers and consumers.</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>A mediator that encapsulates all the communications between organizations.</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>A data consumer communicates with a data provider via the web service interface.</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>The workflow of the selective data replication.</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Sheth and Larson’s five-level schema framework.</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Model transformation using MDA.</td>
<td>21</td>
</tr>
<tr>
<td>3.7</td>
<td>A simple ontology represented in UML.</td>
<td>22</td>
</tr>
<tr>
<td>3.8</td>
<td>Translating UML class diagrams into OWL ontologies.</td>
<td>24</td>
</tr>
<tr>
<td>3.9</td>
<td>A four-level ontology framework by combining Sheth and Larson’s five-level schema framework for FDBS and the OMG’s notions of PIMs and PSMs in the MDA.</td>
<td>25</td>
</tr>
<tr>
<td>3.10</td>
<td>Relationships between various roles of ontologies and organizations.</td>
<td>27</td>
</tr>
<tr>
<td>3.11</td>
<td>The architecture of the selective data replication system.</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Update notification propagation via the “push” method.</td>
<td>35</td>
</tr>
<tr>
<td>4.2</td>
<td>Update notification propagation via the “pull” method.</td>
<td>36</td>
</tr>
<tr>
<td>4.3</td>
<td>The data consumer may require update notifications from a second data provider after receiving an update notification from the first one; therefore, in this case, both the “push” (left) and the “pull” (right) methods are adopted.</td>
<td>37</td>
</tr>
<tr>
<td>4.4</td>
<td>The database table <code>data_updates</code> used to store update notifications.</td>
<td>38</td>
</tr>
<tr>
<td>4.5</td>
<td>Populating the values of <code>property_name</code> and <code>property_value</code> in the <code>data_updates</code> table.</td>
<td>39</td>
</tr>
<tr>
<td>4.6</td>
<td>The update notification ontology represented using a UML class diagram.</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>An example of how SWRL rules are defined in Protégé.</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>An example of how SWRL rules are defined in a UML class diagram.</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>The workflow for update policy specification and evaluation.</td>
<td>48</td>
</tr>
<tr>
<td>6.1</td>
<td>The architecture of the selective data replication system.</td>
<td>51</td>
</tr>
<tr>
<td>6.2</td>
<td>The Export PIM ontology of <code>LambdaCCC.Address</code> represented in UML and XML.</td>
<td>53</td>
</tr>
<tr>
<td>6.3</td>
<td>The ontology translation process for <code>Address</code> in <code>LambdaCCC.Address</code>.</td>
<td>56</td>
</tr>
<tr>
<td>6.4</td>
<td>The UML class diagram of <code>Consumer</code> and <code>ConsumerSubscription</code>.</td>
<td>57</td>
</tr>
<tr>
<td>6.5</td>
<td>The UML sequence diagram of the implementations of <code>subscribe</code> and <code>unsubscribe</code>.</td>
<td>57</td>
</tr>
<tr>
<td>6.6</td>
<td>The UML sequence diagram of the implementation of <code>getNotifications</code>.</td>
<td>58</td>
</tr>
<tr>
<td>6.7</td>
<td>The UML sequence diagram of the implementation of <code>getData</code>.</td>
<td>59</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>6.8</td>
<td>The workflow for update policy specification and evaluation.</td>
<td>61</td>
</tr>
<tr>
<td>7.1</td>
<td>The task-breakdown analysis of the row-level and the selective data replications.</td>
<td>67</td>
</tr>
<tr>
<td>7.2</td>
<td>“Post Office” Scenario: Task-breakdown analysis.</td>
<td>69</td>
</tr>
<tr>
<td>7.3</td>
<td>“Post Office” Scenario: Task-breakdown analysis when “Smart Fallback” is enabled for the selective data replication.</td>
<td>70</td>
</tr>
<tr>
<td>7.4</td>
<td>“Real Estate” Scenario: Performance comparison between selective data replication and Slony-I, with fixed notification size and variable number of desired updates.</td>
<td>71</td>
</tr>
<tr>
<td>7.5</td>
<td>“Real Estate” Scenario: Performance comparison between selective data replication and Slony-I, with variable notification sizes and fixed number of desired updates.</td>
<td>73</td>
</tr>
<tr>
<td>7.6</td>
<td>“Real Estate” Scenario: Performance comparison between selective data replication (with and without “Smart Fallback”) and Slony-I, with variable notification sizes and fixed number of desired updates.</td>
<td>74</td>
</tr>
<tr>
<td>A.1</td>
<td>How to create a new UML project from an existing XMI document in Protégé.</td>
<td>88</td>
</tr>
<tr>
<td>A.2</td>
<td>A sample Address ontology encoded in XMI is loaded in Protégé.</td>
<td>89</td>
</tr>
<tr>
<td>A.3</td>
<td>Enable SWRLTab in Protégé.</td>
<td>89</td>
</tr>
<tr>
<td>A.4</td>
<td>Before and after activating SWRL for an OWL ontology in Protégé.</td>
<td>90</td>
</tr>
<tr>
<td>A.5</td>
<td>A UML Class diagram representing the Address ontology with one update policy rule.</td>
<td>91</td>
</tr>
</tbody>
</table>
## Listings

3.1 The short-cut version of a simple ontology represented in OWL. . . . . . . . . . . . 23
4.1 The XML representation of an update notification with administrative components. 33
4.2 The XML representation of a data update description. . . . . . . . . . . . . . . . 34
4.3 The XML representation of an update notification with all of its components. . . 34
4.4 An example of an update notification generated during the testing. . . . . . . . . . 35
5.1 Snippets of $C_{PostOffice}$’s update policy rule expressed in SWRL’s RDF concrete
    syntax. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.2 $C_{PostOffice}$’s UML model and update policy rule encoded in XMI . . . . . . . . 47
6.1 The SQL statement for creating the local_psm.address table. . . . . . . . . . . . . . 54
6.2 The SQL statement for creating the mirror local_psm.address_update table. . . 54
6.3 The statement for creating the audit_address_update trigger. . . . . . . . . . . . . 55
6.4 Attach the trigger audit_address_update to the local_psm.address table. . . . . . . 55
6.5 Using database views to perform ontology and data translation. . . . . . . . . . . . 56
6.6 Sample code for $C_{PostOffice}$ to retrieve update notifications from $PCCC$. . . . . . 59
6.7 The SQL statement for creating the data_updates table. . . . . . . . . . . . . . . 60
6.8 Custom SWRL built-ins for geospatial operations. . . . . . . . . . . . . . . . . . . . 63
7.1 The sample log file produced during the execution of the selective data replication. 68
A.1 The XML encoded Address ontology with the update policy. . . . . . . . . . . . . . 91
A.2 The OWL encoded Address ontology with the update policy. . . . . . . . . . . . . . 97
B.1 Source Code for Ontology Translation using Protege APIs. . . . . . . . . . . . . . . . 101
B.2 Source Code for Update Notification Retrieval. . . . . . . . . . . . . . . . . . . . . 106
C.1 SWRL Built-Ins for Geospatial Operations. . . . . . . . . . . . . . . . . . . . . . . . . 111
Chapter 1

Introduction

Due to the continuing increase in the quantity of electronic data and the increased sophistication of business requirements, storing all information in one database or on one computing device is sometimes impractical, and may introduce problems in availability and performance. Therefore, to solve these problems, data replication was introduced.

In general, data replication is a process of creating and managing duplicate versions of a database. Each duplicated version could be either an exact copy or a subset of the original database. The original database from which other duplicates are created is called the master database, and each replication is often referred to as a replica or slave database. By creating multiple duplicated versions, each version can be managed differently and even stored on different computers residing in different physical locations, as long as the data shared between them can be synchronized when necessary.

The reason why data replication can address the aforementioned problems is due to two of its most frequently mentioned advantages (Dye 1999, Elmasri & Navathe 2000, Padigela 2008): one, improved availability and the removal of a single point of failure, and two, better throughput, faster response times and reduced communication costs and network bottlenecks to geographically distributed end-users.

Data replication exists in many different domains and is used for various purposes but with different names; for example, RAID (Redundant Array of Independent Disks) technology, mirror sites for web pages and backup mechanisms are all different forms of data replication. However, the data replication that is of interest to this research is database replication, in particular, within geographical information systems (GIS).

Regardless of which variant of a data replication is used, it has always been a challenge to implement a system that requires maintaining data integrity and consistency across multiple replicated data sets. Although there are a few well established patterns and mechanisms available for solving this type of problem, alternatives or derivatives have been constantly studied by many researchers (Schiper & Raynal 1996, Alonso 1997, Keidar & Dolev 1998, Jimenez-Paris et al. 2001) to try to further improve performance (Holliday et al. 1999, Wiesmann & Schiper 2005) and reduce deadlocks (Holliday et al. 2000).

In this research, a new variant of data replication is introduced and is referred to as selective data replication. It is selective because it allows the scheduling of data replications to be based on more flexible and dynamic decisions such as when and how data replications should take place, rather than blindly synchronizing the updated data as is the case in many existing data replication implementations, which are referred to as “traditional data replications” in this research.
Another motivation of this research is to design selective data replication in a way that will enable it to be adapted to a software agent environment, which allows the incorporation of the collaborative and intelligent behaviours, further enhancing the scalability and flexibility of selective data replication. Although such adaptation is beyond the scope of this research, it influences the design and implementation of the selective data replication system in many ways, which are explained when they are encountered.

Before explaining what selective data replication is and how it works, Chapter 2 provides background information on data replication in general, and explains why another data replication system is required by describing the characteristics and the associated problems of this research environment. To better understand why those problems exist, especially in the real world context, a case study with three scenarios is provided to demonstrate how real life users may, or would like to, interact with a data replication system in the envisaged research environment. Throughout the thesis, these three scenarios are frequently referenced through examples and for demonstration purposes. The definition of selective data replication and the research objectives are presented at the end of Chapter 2.

The solutions to the research problems mentioned in Chapter 2 are discussed in Chapter 3. During the discussion of each solution, the approaches currently available are firstly examined, followed by the original contributions of this research. In addition in Chapter 3, the system architecture of the selective data replication process is illustrated to show the data replication workflow and all the functional components within the system and their relationships.

In the subsequent two chapters, the two most important functional components are singled out and discussed in greater detail, namely, the update notification and update policy.

Update notification is an essential component of the selective data replication system, as it is used to notify data updates, and to trigger the subsequent processes in the system. Chapter 4 provides a detailed discussion on how this research processes update notifications, including their creation, propagation and management.

In Chapter 5, the second of the two essential components - update policy - is described. In the simplest terms, an update policy is a list of rules defining conditions on when and how data replications should take place. In this research, most of these conditions are evaluated in terms of the information embedded in update notifications, which means not all updates are necessarily replicated; instead, only the ones that satisfy the update policy are replicated.

Following the breakdown discussions on the two essential functional components, the implementation of the entire selective data replication system is examined in Chapter 6 and evaluated in Chapter 7 to determine whether the research objectives defined in Chapter 2 are achieved.

The future work of this research is listed in Chapter 8 followed by Chapter 9 which concludes the thesis.
Chapter 2

The Problems

The previous chapter briefly introduced selective data replication, aiming at providing a more flexible and dynamic way of scheduling data synchronizations.

To understand the problems addressed by this research, this chapter firstly provides a background on data replication, followed by a description of the research environment, explaining its various characteristics and associated problems, and, more importantly, why these problems are unique to this research environment and why they cannot be easily solved by existing data replication strategies. Following that, a case study is provided with three real life scenarios to further demonstrate the problems that this research encounters and solves. Finally, at the end of this chapter, the research objectives are presented.

2.1 Background on Data Replication

2.1.1 Data Fragmentation

Since database replication is a process of creating and managing duplicate versions of a database, the first step towards data replication is to create replicas, which can be achieved by the process of data fragmentation.

Data fragmentation is a process to fragment a database into multiple logical units called fragments, which can be stored at different replication sites. As long as the database is not partitioned into disjoint fragments, data replication is necessary to maintain the consistency and the integrity for the intersection of these fragments.

There are three ways to fragment a database: horizontal fragmentation, vertical fragmentation and mixed fragmentation (Elmasri & Navathe 2000, Padigela 2008). Horizontal fragmentation specifies a condition on one or more attributes of a database table in order to divide that table horizontally by selecting the relevant data. For example, a department of a company that is located in Christchurch gets the fragment of a Customer table where Customer.city = ‘Christchurch’. Vertical fragmentation divides a table vertically by columns, meaning that each vertical fragment only keeps certain attributes of a table. When doing vertical fragmentation, it is important to include the primary key of the table in every vertical fragment so that the full table can be reconstructed if needed. Mixed fragmentation consists of both horizontal and vertical fragmentation. Although mixed fragmentation is also sometimes referred to as “selective” fragmentation, it is different from the selective data replication introduced in this research in some ways, which will be discussed in Section 3.2.
2.1.2 Replication Strategy

Replication strategy is usually determined by two factors: when and where (Alonso 2006). The “when” factor indicates when updates are propagated - either synchronously or asynchronously. Synchronous, or eager replication, propagates any changes made to one replica immediately to all other replicas inside one transaction, in which way either, all or none of the replicas will be updated, guaranteeing that each replica will contain the most up to date information. However, synchronous replication is slow and deadlock prone (Wiesmann & Schiper 2005). To address these issues, asynchronous, or lazy replication, is designed to update replicas asynchronously, i.e. not within a single transaction. Although this way is efficient, while the propagation takes place the replicas may be inconsistent for typically short periods of time (Gray et al. 1996, Wiesmann et al. 2000).

The “where” factor specifies where updates are allowed to take place. There are two common approaches (Gray et al. 1996, Wiesmann et al. 2000): one is called primary copy - only allowing a single database, i.e. the master database to be updateable and the rest are read only; the other approach is update everywhere - allowing updates to be initiated at any of the databases, including both the master and the slaves. With the former approach, it is relatively easy to implement and manage as all the updates are made to one site and there is no need for inter-site synchronization. The disadvantage, however, is that the load at the primary site could be large. The latter approach is more difficult to implement, but is less restrictive and the load can be evenly distributed across the sites. In the latter approach, the distinction between master and slave databases is redundant since every database is updatable.

Figure 2.1 summarizes the above discussions into the four most basic replication strategy categories: synchronous primary copy, synchronous update everywhere, asynchronous primary copy and asynchronous update everywhere. As will be mentioned later, the selective data replication developed by this research falls into the “asynchronous primary copy” category.

![Figure 2.1: The four most basic replication strategy categories.](image)

2.2 Research Environment

2.2.1 Multi-Organization Environment

Like many environments into which data replications are usually deployed, this research environment is data decentralized and geographically distributed. However, the difference is that, while a traditional data replication scheme is most likely deployed into an environment that is internal to one single organization, this research environment can span multiple organizations.
The single-organization environment is commonly seen in a retail chain company, for example, as the company usually has a central database storing all the sales and customer information and each retail store only keeps a local copy of the data that is relevant to its physical location. In this type of environment, the central database is the master database in the organization, and each replica is a subset of the central database and is usually held by a department or a branch office of the organization at a different physical location.

Instead of having a single organization with a clearly defined master and slave relationship for its internal databases, multiple organizations are involved in this research environment. Some of these organizations provide their data and data updates as services to other organizations, others consume those services, and some do both. Since there is no clear master and slave relationship in this environment, the terms “master”, “replica” and “slave” are no longer appropriate. Instead, in this research an organization that provides others with access to its shared data set is referred to as a **data provider**; an organization that, at some point in time, creates a local subset of all data provided within the environment is referred to as a **data consumer**. One organization can be either a data provider or a data consumer, or both.

In this environment, a shared data set can only be changed by the data provider who owns the data set, and when changes occur, they are asynchronously replicated into the data consumers’ local copies. Although a data consumer can still make changes to its own local copy, the changes will not be replicated back to the provider, which means the replication is unidirectional - only from providers to consumers. Therefore, this is why the selective data replication falls into the “asynchronous primary copy” category as mentioned before.

### 2.2.2 Geographical Information System Environment

The selective data replication system is designed to work within any distributed environment where data sharing and data replications are required, regardless of the computing domain. However, for the purposes of placing this selective data replication system into a real life context, and in order to demonstrate and analyze it with real data, a Geographical Information System (GIS) domain is chosen as the domain of this research.

A GIS is a computer-based system used for creating, managing, analyzing and displaying geographically referenced information, which is also referred to as spatial or geospatial data. GISs are used by many organizations: governments, research institutes and other bodies for tasks such as resource management, development planning, land evaluation, environment observation and cartography. For example, police and fire departments may use GISs to locate landmarks and hazards, to plot destinations and to design emergency routes. During a disaster, government organizations and road workers may use GISs to locate damaged road segments, bridges or traffic signs in order to make informed and rapid decisions on rescuing lives, planning resources and performing recovery activities.

Among many characteristics of a GIS database, one is worth mentioning to avoid confusion later in this thesis. A GIS database is usually constructed using multiple, overlapping, implicitly related but geographically collocated, **feature types**, with each representing a collection of geographical features of the same type. The concept of a feature type is the same as the concept of a database table, but it could also be treated as a mini-database on its own. This is because, usually, each feature type is a standalone entity and has no explicit relationships to other feature types in the database; instead, it relates to other feature types via the longitude and latitude coordinates on Earth. Therefore, most of the time, a feature type is viewed as a single data set, and if a GIS database consists of multiple feature types, it consists of multiple
data sets.

However, under some circumstances, a GIS database needs to be viewed differently. For example, a global positioning system (GPS) is used to navigate from one location to another and each location may be of a different feature type, for example, from an airport to a downtown hotel or from a lake to a nearby golf course. In these cases, all a user sees and cares about is one single data set merged with all the available feature types; therefore, the word “data set” is given a different meaning - a mixture of multiple feature types.

Figure 2.2 is an illustration of these two views: the view that treats a single feature type as a single data set is called the “horizontal view”, and the view that treats multiple feature types as a single data set is called the “vertical view”. The horizontal view is usually used in the manipulation of a single feature type, where isolation from other feature types is required. The vertical view is, however, frequently applied in situations where relationships across multiple feature types are needed, such as the aforementioned GPS example.

\[ \text{Figure 2.2: Multiple views of geographical data sets.} \]

In this thesis, the word “data set”, if used without specific indication, refers to a data set under the vertical view, which means it represents a collection of all the available feature types.

2.3 Research Problems

The difference between a single and multi-organization environment is significant, and it raises the following problems for this research.

2.3.1 Unawareness

Unlike a traditional data replication environment, where an organization knows exactly what its departments are, and whether and how they participate in the data replication process, in this research, each organization is a standalone entity and has no awareness of the others, i.e. neither a data provider knows who are interested in its shared data nor a data consumer knows who can provide its desired information.

Unawareness is not only a problem itself, but also the cause of the other two problems mentioned in Sections 2.3.2 and 2.3.3.
2.3.2 Performance

In traditional data replication schemes (Elmasri & Navathe 2000, Padigela 2008), the process of creating a replica is called data fragmentation, the purpose of which is to divide a master database into smaller fragments, with each fragment being stored at a replication site with the information relevant to that site. Once a replica is created, its fragmentation configuration is stored by the replication system and used to determine if any updated data belongs to that fragment so that replication decisions can be made. In other words, only the data updates related to a fragment are replicated to that fragment.

During replication, a traditional data replication system commonly uses one or both of two ways to converge data in relational databases: row-level replication and procedural replication (Bobrowski & Smith 1997). With row-level replication, when data records are updated, the replication system automatically captures and stores those updated records, and forwards them to other replicas asynchronously. The disadvantage of this method is that it could potentially load a network with a large quantity of data if there are a lot of updates. Procedural replication, however, instead of replicating the actual updated records, replicates stored procedure calls to converge replicas. In this way, by using the same stored procedure in both the master and the replica databases, the replication system only needs to transmit the stored procedure’s name and parameter values to each replica, resulting in better performance and enforcing the same actions to be applied across databases. For example, if a retail chain store invokes the stored procedure \texttt{UpdateDiscountRateBasedOnAmountThreshold} in its master database to increase the discount rate from 10\% to 20\% for all the products with their original sales prices equal to or greater than $500.00, instead of using row-level replication, which may cause hundreds of data records to be copied to each replica database, procedural replication only needs to propagate three values: the stored procedure name (\texttt{UpdateDiscountRateBasedOnAmountThreshold}) and its two parameters - the amount threshold ($500.00) and the new discount rate (20\%).

In the particular environment being considered in this research, each data provider’s shared data set can be accessed by an unknown number of data consumers, so the provider simply makes all of its updated data available to all the consumers, which means, in the absence of a more sophisticated selection mechanism, each data consumer has to replicate either all or none of the data updates regardless of whether some of these updates are useful or necessary, causing performance overhead and resource wastage.

In addition, due to the heterogeneity that will be mentioned in Section 2.3.3, procedural replication is not an option in this research environment, which makes the performance issue even more critical, especially since GIS data sets will typically be large, and transmitting large parts of these data will be resource intensive and potentially slow.

2.3.3 Heterogeneity

Because of the unawareness between different organizations, the format of the data provided by a data provider may not be compatible with what a consumer would expect, and the consumer cannot pre-negotiate it with the provider. This situation would be worse if the consumer subscribes to multiple data providers, who are sharing similar data sets but with different formats.

Furthermore, the technologies used for data storage and data access could also differ between data providers. For example, one data provider could store its data in a relational database and expose it using WFS (Web Feature Service), another could store its data in an object-oriented database and expose it via web service interfaces, or an organization may not even use
a database at all - simply offering XML data files for download instead. Due to the differences in technology, the procedural replication mentioned in Section 2.3.2 cannot be deployed in this research environment, since the stored procedure defined in a data provider’s database may not exist in a consumer’s database, or the consumer may not even have a database.

Therefore, in such a heterogeneous environment, if a data consumer needs to view and process information from distinct and varied data providers, many types of conversions may be necessary, thereby complicating information sharing.

2.4 Case Study

This section presents a case study with three scenarios, each of which conveys how data providers and data consumers may interact with each other within the envisaged research environment, and provides an example of the problems described in Section 2.3.

In this case study and throughout the thesis, the notation being used is summarized in Table 2.1.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_n$</td>
<td>Data Provider n</td>
</tr>
<tr>
<td>$C_m$</td>
<td>Data Consumer m</td>
</tr>
<tr>
<td>$\Lambda_{n,d}$</td>
<td>The shared data set d provided by $P_n$</td>
</tr>
<tr>
<td>$\lambda_{m,e}$</td>
<td>The local copy of a shared data set e created by $C_m$</td>
</tr>
</tbody>
</table>

The data used in this research is from the Christchurch City Council (CCC), $P_{CCC}$, which is one of the major geographic data providers of various geospatial information about Christchurch, New Zealand. In this case study, $P_{CCC}$’s shared data set contains four spatial feature types: Address, Parcel, Building and Road. The spatial instances of each of these four feature types are stored in the horizontal view data set $\Lambda_{CCC,Address}$, $\Lambda_{CCC,Parcel}$, $\Lambda_{CCC,Building}$ and $\Lambda_{CCC,Road}$, respectively.

Another data provider in this case study is the Ministry of Civil Defense, $P_{CivilDefense}$, who provides a disaster warning data set $\Lambda_{CD,DisasterWarning}$ containing warnings about the latest detected natural disasters across the nation. For various reasons, the real data was not made available to this research, so $\Lambda_{CD,DisasterWarning}$ only contains hypothetical data generated by this research for simulation and testing.

2.4.1 “Post Office” Scenario

Story

The Christchurch Post Office $C_{PostOffice}$ subscribes to $\Lambda_{CCC,Address}$ provided by $P_{CCC}$ and keeps a local copy - $\lambda_{PostOffice,Address}$ for fast and convenient data access. To ensure all mail is accurately delivered to their destinations, $C_{PostOffice}$ considers every single address change issued by $P_{CCC}$ as a significant change, regardless of the nature or the content of the change.

Remarks

This is a very simple scenario demonstrating the most basic environmental setup and the replication logic. The environment of this scenario is a multi-organization environment, involving
2.4. CASE STUDY

only two organizations: one provides the data and the other consumes it. These two organizations may not be initially aware of each other and may use two different database management systems (DBMS). The replication logic is also very simple, the consumer organization simply accepts all the data updates published by the data provider.

Another reason this scenario is included is that, due to its simplicity, it can be implemented using a traditional data replication system, and such implementation can be used as a baseline for evaluating the selective data replication system developed by this research.

2.4.2 “Real Estate” Scenario

Story
A large local real estate company $C_{RealEstate}$ is constantly looking for large sub-divideable residential areas for property development opportunities. In order to get the latest available information on any new or updated parcels within Christchurch, $C_{RealEstate}$ chooses to subscribe to $\Lambda_{CCC,Parcel}$ from $P_{CCC}$. Since $C_{RealEstate}$ is not interested in small sized and non sub-divideable parcels, it defines the following business rules for parcel data replication:

1. the parcel must be of residential type;
2. the parcel must be sub-divideable; and
3. the area of the parcel must be at least 15 hectares.

Remarks
Although this scenario still only involves two organizations, the replication logic becomes more complicated. Instead of accepting all the data updates, the consumer is only interested in, and wishes to replicate, a subset of them. As mentioned before, by using the traditional row-level replication, the consumer has to accept all or none of the updates, regardless of whether some of them are useful and match the consumer’s business requirements, thereby resulting in a potentially negative performance impact. For example, if $P_{CCC}$ updated 500 existing downtown parcels with none having an area greater than or equal to 15 hectares, then by using the traditional row-level replication, $C_{RealEstate}$ would have no choice but to replicate all 500 undesired data updates.

2.4.3 “Disaster Control” Scenario

Story
Both the Christchurch Police $C_{Police}$ and the Christchurch Central Hospital $C_{Hospital}$ have procedures put in place to monitor and keep a local copy of $\Lambda_{CD,DisasterWarning}$ provided by $P_{CivilDefense}$, in order to react to any natural disasters or emergencies in a timely fashion. Whenever a new disaster warning for the Christchurch area is inserted into $\Lambda_{CD,DisasterWarning}$, both $C_{Police}$ and $C_{Hospital}$ will not only download the warning detail from $P_{CivilDefense}$ into their own local copies: $\lambda_{Police,DisasterWarning}$ and $\lambda_{Hospital,DisasterWarning}$ respectively, but will also download the information about buildings and roads that are within the range of 1 to 10 kilometers surrounding the disaster location, depending on the nature and the scale of the disaster. The information on the buildings and roads would come from the subscription to $P_{CCC}$’s $\Lambda_{CCC,Building}$ and $\Lambda_{CCC,Road}$, and is stored locally in $\lambda_{Police,EmergencyWork}$ and
λ_{Hospital,EmergencyWork}, which only contain the geospatial features surrounding the emergency work location. λ_{Police,EmergencyWork} and λ_{Hospital,EmergencyWork} are only synchronized with PCCC whenever there is an emergency, and are left out-of-date for the rest of the time.

Remarks

The complexity of this scenario comes from two aspects. One is that the consumer’s data set λ_{Police,EmergencyWork} (or λ_{Hospital,EmergencyWork}) is not identical to, or a subset of a single provider’s data set. Instead, it contains a mixture of data coming from various data providers or feature types, which complicates the process for integrating the replicated data.

The other aspect of the complexity comes from the involvement of multiple data providers and feature types, like the situation in this scenario where updating one feature type requires updating the others. Under such circumstances, a data consumer sometimes has to ignore data updates completely, but also sometimes has to proactively initiate the update checking process to replicate any previously ignored data updates based on its business requirements.

None of the complexities raised from this scenario can be easily handled by implementing and deploying a traditional row-level replication system.

2.5 Research Objectives

To solve the problems discussed in Section 2.3 and illustrated in Section 2.4, this research develops the selective data replication system by employing the concepts of update notification and ontology-based update policy. In simple terms, an update notification is a message encapsulating information about the updated data, and an update policy is a collection of rules that are specified based on one or more common ontologies to reflect a data consumer’s business requirements about when to perform data replications and what data to replicate. More details on the update notification, ontology, update policy, and how they are used in this research to address the aforementioned problems are discussed in later chapters. It is worth noting that this research is only focused on replicating data updates, rather than data schema updates, which itself is a complicated topic and is beyond the scope of this research.

With the selective data replication system and the concepts of update notification and ontology-based update policy, this research aims at:

offering better flexibility and efficiency to a consumer organization than the traditional row-level data replication, when incorporating data updates from one or more data providers.

The research objective can be divided into four parts, and achieving each becomes a sub-objective of this research; they are:

1. To provide a means by which a data provider organization’s update notification can be generated and propagated to consumer organizations.

2. To provide a means by which a data consumer organization’s update policy can be easily used in conjunction with the common ontologies and evaluated in terms of received update notifications.

3. To implement the selective data replication system.
4. To evaluate the selective data replication system against a traditional row-level replication system in order to determine whether it indeed provides better flexibility and efficiency.

The first sub-objective is discussed in Chapter 4 including discussions on the update notification’s content, propagation methods and management. The second sub-objective, which is about the update policy specification and evaluation, is explained in Chapter 5. The last two sub-objectives: implementation and evaluation of the selective data replication system are addressed in Chapters 6 and 7 respectively.
Chapter 3

The Solutions

The previous chapter provided the background information about this research, including the research environment, problems and objectives. In this chapter, the solutions and methodologies adopted in the selective data replication system, which is developed in this research, to address the problems and meet the objectives, are discussed.

According to Section 2.3, there are three major problems in this research environment: unawareness, performance and heterogeneity; therefore, this chapter includes three sections with each discussing the approaches taken by this research to solve each one of the three problems. During the discussion, the currently available methodologies are firstly assessed to determine their limitations for solving the particular problems. The original contribution of this research, which consists of incorporating both existing methodologies and new ones, is then presented.

At the end of the chapter, the architecture of the selective data replication system is presented to identify the core functional components and their relationships, and to define the research boundaries.

3.1 Solution to Unawareness

The prerequisite of the selective data replication scheme is to establish the relationships between data providers and data consumers, which can be typically achieved in one of the following two ways: using a peer-to-peer or a mediator communication model.

3.1.1 Peer-to-Peer Communication Model

Setting up peer-to-peer communications usually requires data providers and data consumers to go through three steps: provider registration, consumer discovery and subscription, which are illustrated in Figure 3.1 and elaborated in the following subsections.

Registration

Registration is the process for data providers to register themselves so that they can be later discovered by data consumers. This is analogous to making advertisements to attract customers. The common ways for data providers to make such “advertisements” are to use either Internet search engines, such as Google and Yahoo, or directory services, for example, Seekda (Seekda 2008) and XMethods (XMethods 2008).
Relying on Internet search engines is arguably the easiest way, as a data provider needs only to create a web site that best describes the information on its shared data and supported operations; then this web site can be eventually found by data consumers through search engines. The drawback, however, is that the language used to describe the data and operations may not be consistent across data providers, which may cause difficulties for data consumers - specifically data consumers’ applications - to interpret and understand.

To ensure data and services can be consistently represented by data providers and interpreted by data consumers, the web service framework (Lafon 2008) can be adopted to provide such a standardized mechanism. The web service framework facilitates this by defining the WSDL (Web Services Definition Language), which is an XML-based language for describing network services as a set of endpoints, operating on messages containing either document-oriented or procedure-oriented information (Christensen et al. 2001).

Another advantage of adopting the web service is that the defined WSDL can be registered in specialized directory services, such as UDDI (Universal Description, Discovery, and Integration protocol), which enables clients to locate web service providers and services themselves on the Internet by querying the directory services, as explained below.

**Discovery**

Discovery is the process of data consumers searching for, and locating, data providers. As mentioned previously, this can be done using Internet search engines or by querying directory services.

In a distributed environment, if each data provider provides a WSDL file describing the available data and services, during the discovery process, a data consumer can compare that information with its own requirements to see if they match. In reality, it is very likely that more than one data provider’s data and services are of interest to a data consumer, in which case, the consumer needs to decide which one (or more) to subscribe to. This decision can be influenced by many factors, such as data dependencies, associated costs and update frequency.
3.1. SOLUTION TO UNAWARENESS

Subscription

Once a data consumer finds and decides to subscribe to one or more data providers, a subscription process needs to take place between the consumer and all the providers. This process is analogous to subscribing to a magazine, the consumer has to exchange details with each of the providers in order for them to identify each other so they can directly communicate in the future.

3.1.2 Mediator Communication Model

Instead of letting data providers and consumers directly communicate with each other, the mediator communication model introduces a “third-party” - the mediator - that encapsulates all the communications between organizations. Therefore, with this communication model, data providers and consumers are only aware of and able to communicate with the mediator, but not with each other. It is the mediator’s responsibility to forward communications to the correct recipients. Figure 3.2 illustrates the mediator communication model.

![Diagram of mediator communication model]

Figure 3.2: A mediator that encapsulates all the communications between organizations.

Using the mediator communication model for data enquiry and data management is very similar to the Federated Database System (FDBS), which is a type of meta-database management system that transparently integrates multiple autonomous and possibly heterogeneous and geographically decentralized database systems into a single, virtual federated database (Heimbigner & McLeod 1985). Therefore, each user of the FDBS only sees and deals with the federated database, rather than multiple number of smaller databases.

The FDBS and this research share some similarities with regard to data management: each “autonomous, heterogeneous and decentralized” database in FDBS can be viewed as one data provider’s shared data set, and the conjunction of all the providers’ shared data sets forms a super data set that is equivalent to the single federated database in FDBS. However, there are some differences between the FDBS and this research, and they are described in Section 3.3.

3.1.3 Research Solution

In this research, both of the aforementioned communication models are used, but for very different purposes. For data replication that involves the retrieval of the actual data and data updates, the peer-to-peer communication model is adopted as it is easy to set up, and does
not require the complexity of having additional organizations to provide mediation services and host virtual databases as it would with the mediator communication model. The mediator communication model used in this research is for exchanging ontologies, which will be explained later in Section 3.3.

To enable peer-to-peer communication, data providers need to provide at least one interface for data consumers to interact with, such as performing subscriptions and retrieving data updates. In this research, as shown in Figure 3.3, the web service interface is currently the only interface exposed by data providers and used for such purposes. Apart from the advantages mentioned in Section 3.1.1, the web service is also extensible and able to support all the communications between data providers and consumers, especially the ones that are of particular interest to this research, which will be discussed in Section 3.2.

![Figure 3.3: A data consumer communicates with a data provider via the web service interface.](image)

Furthermore, in Figure 3.3, the WSDL document of a data provider’s web service can contain information about the ontology of the shared data set, so that while discovering services, a data consumer can use the embedded ontology information to analyze if the data provided by a particular provider meets its requirements, thereby deciding whether to subscribe to this data provider. The details of the notion of an ontology are discussed in Section 3.3.

### 3.2 Solution to Performance

Section 2.3.2 discussed two common ways to converge data in traditional data replication systems using row-level or procedural replication. Since the procedural replication method is not suitable in this research environment due to the heterogeneity in technology, this research has to adopt an approach that is similar to the row-level replication. However, as illustrated by the case study scenarios in Section 2.4, simply deploying row-level replication in this research environment may result in unnecessary network traffic and a negative performance impact as data consumers may get more than what they require.

To minimize unnecessary network traffic, the selective data replication developed by this research allows data consumers to define a collection of rules that form an *update policy*, to
3.3. SOLUTION TO HETEROGENEITY

determine both the circumstances under which their local copies of the providers’ data sets are to be updated and the actual data to update. In this way, instead of replicating all the data updates from a provider, only the significant data updates, i.e. the updates that are of interest to a data consumer, are replicated. This also distinguishes the word “selective” in the context of selective data replication from that of mixed fragmentation, which was mentioned in Section 2.1.1. In the context of mixed fragmentation, “selective” means selectively choose which part of a database to fragment, whereas in this research, it means selectively choose which data updates to replicate after a provider’s shared data set has been subscribed to by a data consumer and the consumer’s local copy has already been created.

For a data consumer to determine if a data update is a significant one without retrieving the entire updated data, data providers have to firstly provide update notifications, which contain subsets of information about the updated data. Once an update notification is received, the significance of the notification can be determined by evaluating the consumer specified update policy; when deemed significant, a data replication process will be executed. More details on the update notification and update policy will be discussed in Chapters 4 and 5, respectively.

Figure 3.4 summarizes the above discussion into seven sequential steps comprising the workflow of the selective data replication:

1. **Making data changes** The data provider’s application makes one or more changes to the shared data set, initiating the following processes.

2. **Capturing data changes** Database triggers are used to capture data changes. The captured information will be stored temporarily in the data provider’s database. More details on the data change capture mechanism and its implementation are discussed in Chapters 4 and 6.

3. **Generating & propagating update notifications** As mentioned earlier, an update notification firstly needs to be generated and propagated to the subscribed data consumers in order for them to analyze the significance of the updates and to decide whether to carry out data replications. The generation and propagation of update notifications are discussed in Chapter 4.

4. **Evaluating update policy** Once an update notification is received, the data consumer can evaluate the data change information embedded in the update notification in terms of its update policy to determine whether replication is necessary. Policy specification and evaluation are discussed in Chapter 5.

5. **Initiating data replication** If any data changes are identified as significant, the data consumer will initiate the data replication process by requesting the actual updated data.

6. **Retrieving & returning data** Based on the data consumer’s request, the data provider retrieves the data from the shared data set and sends it back to the data consumer.

7. **Persisting the replicated data** The data consumer stores the replicated data in the local copy of the data provider’s shared data set.

3.3 Solution to Heterogeneity

Although the workflow of the selective data replication system is introduced in Section 3.2, it does not discuss a solution to the heterogeneity issue mentioned in Section 2.3.3. Each
time a data consumer receives an update notification or performs a data replication, it has to convert the received information from the format provided by the data provider into a format that can be easily integrated into its own local data set. To transform a heterogenous environment into a homogeneous one, as will be discussed in the following subsections, a *four-level ontology framework*, which was initially described by Gu & Pascoe (2005), is developed as part of this research by integrating elements of three previous approaches: ontologies, Sheth & Larson’s (1990) five-level schema framework for distributed systems and the OMG’s notion of PIM (Platform Independent Model) and PSM (Platform Specific Model) elements of the MDA (Model Driven Architecture) (OMG 2008).

### 3.3.1 Ontologies

An ontology is “an explicit formal specification of a shared set of concepts within a domain and the relationships between these concepts” (Ontology 2008). Ontologies can be used by both people and software applications to encode, share and reason about knowledge not only within
3.3. SOLUTION TO HETEROGENEITY

a domain, but also across domains. In addition, developing an ontology allows the separation of domain knowledge from the operational knowledge, making domain knowledge reusable and easy to analyze (Gruber 1995).

Similar to object-oriented (OO) models that are usually represented in UML (Unified Modelling Language), ontologies also comprise components such as instances, classes, properties, relations, restrictions and rules; however, the difference is that an ontology exploits declarative representation, while OO modelling is more procedural. In OO modelling, “the meaning of class, relations among classes, and methods are procedurally embedded and they are implicit”. In ontologies, on the other hand, “descriptions are made declaratively in most cases to maintain formality and explicitness” (Mizoguchi 1998).

Ontologies are commonly used in a distributed environment to address the heterogeneity issue, by not only providing a set of common vocabularies, but also enriching the description of, and bringing “…additional higher order semantics” (Cole & Hornsby 2005) into the data being shared within such environment, especially when multiple organizations are involved, such as in this research environment. Therefore, using ontologies allows the shared data to be more easily and consistently interpreted by different organizations.

3.3.2 Sheth & Larson’s Five-Level Schema Framework

Sheth & Larson’s (1990) five-level schema framework for distributed systems was initially introduced for the federated database system (FDBS), and was developed based on the concept of data abstraction to enable users to store and retrieve data in multiple non-contiguous and heterogeneous databases with a single query. In the five-level schema framework, as shown in Figure 3.5, the five levels are:

- **local schema**, dependent on the component database management system (DBMS) and expressed in the native data model of that DBMS;

- **component schema**, derived by translating the local schema into a Common Data Model (CDM) of the FDBS in order to support the heterogeneity feature of an FDBS;

- **export schema**, a subset of the component schema as not all data of a component database system (DBS) may be available to the federation and its users;

- **federated schema**, also referred to as a distributed schema, is an integration of multiple export schemas; and

- **external schema**, a subset of the federated schema that is of interest to the individual user.

As mentioned in Section 3.1.2, the FDBS and this research share some similarities, so the concept of the five-level schema framework is incorporated into this research for data and ontology management, which will be discussed in Section 3.3.4. However, the difference is, in this research there is no federated database, and data consumers communicate directly with providers for data enquiries. Although there is no federated database, there is a type of ontology in this research that is analogous to the federated schema. This ontology type will also be discussed in Section 3.3.4.
3.3.3 Elements of the Model Driven Architecture

Sheth and Larson’s five-level schema framework only addresses the heterogeneity issue related to data and data schemas, but not to the technologies and implementations that are used to manage and consume the data that is available within a distributed environment, therefore, the model driven architecture (MDA) is employed to address the technology heterogeneity issue.

MDA has been developed by the Object Management Group (OMG) (OMG 2008) in order to provide an open and vendor-neutral approach to solve the challenge of business and changes in technology. To achieve this, MDA has been introduced with the notions of the Platform Independent Model (PIM) and the Platform Specific Model (PSM).

A PIM, as its name suggests, provides a platform-neutral environment for specifying only a system’s business logic, functionality and behaviour without containing any technological artifacts, such as J2EE, .NET and CORBA. Ideally, a PIM once developed should allow for reuse regardless of the technology used for underlying implementation; the business logic should stay the same. A PSM, on the other hand, is platform-specific and incorporates technological artifacts used to develop the actual software product, or to manage a database. Figure 3.6 is an illustration of a basic PIM to PSM transformation. The transformation specification shown specifies how the PIM to PSM transformation should occur (Miller & Mukerji 2003); for example, it may specify metamodel information for metamodel transformation, or type information for model transformation.

Usually, one PIM is accompanied by one or more PSMs, i.e. multiple implementations of an abstract model; however, in more complicated circumstances, one PIM may not be sufficient to describe an entire system in a generic way; therefore, more PIMs could be defined with one derived from another. In this research, at least one such PIM is expected to be defined for the entire distributed data set that is available to all organizations including both data providers and data consumers. Each organization will elaborate upon the distributed PIM to form another that is also platform independent, but which is restricted to only those components of the PIM that are of interest to the organization. This restricted PIM, for the local copy of the relevant components of the distributed data set, is further elaborated to form a PSM that introduces platform specific semantics reflecting the particular storage techniques being employed. While
3.3. SOLUTION TO HETEROGENEITY

there is likely to be only one PIM describing the entire distributed data set, there will be at least one pair made up of a restricted PIM and a corresponding PSM for each organization.

3.3.4 Research Solution

Ontology-Based Update Policy

As mentioned in the research objectives in Chapter 2 Section 2.5, update policy specification in this research is based on ontologies, since an update policy requires very rich and accurate domain knowledge that will be provided by the related ontologies. Much similar work (Uszok et al. 2004, Kagal 2002, Damianou et al. 2001) has been done on defining policies to govern the behaviour of many autonomous processes; however, those policies were written in various languages, including natural language (Michael et al. 2001). Use of multiple policy languages without having a common vocabulary leads to ambiguity and unnecessary complexity, and also impacts upon the correctness and accuracy of system behaviour if policies are ill- or incompletely specified. Therefore, more recent research (Jung et al. 2004, Toninelli et al. 2005, Nejdl et al. 2005, Grimm et al. 2004) defines policies based on ontologies in order to overcome the aforementioned issues. In addition, compared to non-ontology based policy specification, ontology based policy specification could inherit any advantages from ontology definition that would not be available otherwise; for example, Nejdl et al. (2005) borrowed the notions of inheritance and polymorphism from object-oriented modelling, and applied them to policy specification and management.

With regard to the ontology representation, there are a large number of languages capable of representing ontologies, and they can be grouped into nine different categories (Lenzerini et al. 2005): programming languages, conceptual and semantic data models, information systems and software formalisms, frame-based, logic-based, graph-based, XML-related, visual languages and temporal languages. Discussing the suitability of using ontology representation languages from each of these nine categories is certainly beyond the scope of this section; instead, only two of the ontology representation languages receive special focus here: UML (Unified Modelling Language) (OMG 2005b) and OWL (Web Ontology Language) (W3C 2007). The reasons why they are focused on are explained in the following subsections.
Ontology Representation using UML

UML was originally designed for human-to-human communication of models for building software applications in object-oriented programming languages. It has been extended to a variety of different purposes, including design of database schemas, XML document schemas and knowledge models. UML has become one of the most widely used formalisms for information systems design (Lenzerini et al. 2005).

Due to the widespread adoption of UML, this research uses UML as an ontology representation language. For example, Figure 3.7 demonstrates a UML class diagram representing a simple geographic ontology containing only the Parcel and Address feature types.

```
<table>
<thead>
<tr>
<th>Parcel</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ pid: int</td>
<td>+ gid: int</td>
</tr>
<tr>
<td>+ parcelType: String</td>
<td>+ houseLow: String</td>
</tr>
<tr>
<td>+ area: double</td>
<td>+ houseHigh: String</td>
</tr>
<tr>
<td>+ suburb: String</td>
<td>+ roadName: String</td>
</tr>
<tr>
<td>+ city: String</td>
<td>+ roadType: String</td>
</tr>
<tr>
<td>+ country: String</td>
<td>+ address: String</td>
</tr>
<tr>
<td>+ boundingBox: String</td>
<td>+ addressType: String</td>
</tr>
<tr>
<td></td>
<td>+ geometry: String</td>
</tr>
</tbody>
</table>
```

Figure 3.7: A simple ontology represented in UML.

However, since UML is not intended for the use of ontology specification, it has some major disadvantages, such as insufficient support for instances and formal semantics specification (Hart et al. 2006, Lenzerini et al. 2005). Therefore, this research also chooses a second language - OWL, which was designed to work naturally with ontologies, including their specifications and representations.

Ontology Representation using OWL

OWL is a logic-based ontology specification language developed by the World-Wide Web Consortium (W3C) in 2004. The idea of OWL is to annotate web pages with machine-interpretable descriptions, making it easier for machines to automatically process and integrate information available on the Web (W3C 2004). OWL has a number of features that address the weaknesses in UML, making it more suitable for ontology representation (Hart et al. 2006). However, UML is still supported in this research, since it is widely adopted and also has certain features that have no equivalence in OWL (Hart et al. 2006), such as behavioural features (operations, abstract classes, interface classes), part-of relationships (composition, aggregation), and access control (read-only properties, public and private elements).

The OWL language provides three species of sub-languages with increasing power of expressiveness: OWL Lite, OWL DL and OWL Full, which are designed for use under different situations and by different users. In this research, although there are no restrictions on which species of OWL to use, due to the limitations on the currently available ontology processing tool mentioned in Chapter 6, only OWL Lite and OWL DL are supported.

An OWL ontology is usually specified using the RDF/XML syntax (Brickley & Guha 2004); as an example, Listing 3.1 demonstrates a short-cut version of an OWL encoded ontology document representing the same ontology as the one shown in Figure 3.7.
Listing 3.1: The short-cut version of a simple ontology represented in OWL.

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xml:base="http://www.owl-ontologies.com/parcel-address.owl#">
        <owl:Ontology rdf:about="#Parcel"/>
        <rdfs:label>Parcel</rdfs:label>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#pid"/>
                <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger">1</owl:cardinality>
            </owl:Restriction>
            <rdfs:subClassOf>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="#addresses"/>
                    <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger">1</owl:minCardinality>
                </owl:Restriction>
                <rdfs:subClassOf>
                    <owl:Class rdf:ID="Address"/>
                    <rdfs:label>Address</rdfs:label>
                    <rdfs:subClassOf>
                        <owl:Restriction>
                            <owl:onProperty rdf:resource="#gid"/>
                            <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#nonNegativeInteger">1</owl:cardinality>
                        </owl:Restriction>
                    </rdfs:subClassOf>
                    <owl:DatatypeProperty rdf:ID="pid">
                        <dfs:domain rdf:resource="#Parcel"/>
                        <dfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
                    </owl:DatatypeProperty>
                    <owl:DatatypeProperty rdf:ID="gid">
                        <dfs:domain rdf:resource="#Address"/>
                        <dfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
                    </owl:DatatypeProperty>
                    <owl:InverseFunctionalProperty rdf:ID="addresses">
                        <dfs:domain rdf:resource="#Parcel"/>
                        <dfs:range rdf:resource="#Address"/>
                        <owl:inverseOf rdf:resource="#belongingParcel"/>
                    </owl:InverseFunctionalProperty>
                    <owl:ObjectProperty rdf:ID="belongingParcel">
                        <dfs:domain rdf:resource="#Address"/>
                        <dfs:range rdf:resource="#Parcel"/>
                    </owl:ObjectProperty>
                </owl:Class>
            </rdfs:subClassOf>
        </rdfs:subClassOf>
    </rdf:RDF>
</rdf:RDF>
```
Combining UML & OWL

As this research supports both UML and OWL as ontology representation languages, processing and maintaining them separately would complicate the workflow of the selective data replication system. To simplify the workflow, as shown in Figure 3.8, this research firstly translates UML represented ontologies into OWL encoded ontologies, which can then be processed by other selective data replication components.

Figure 3.8: Translating UML class diagrams into OWL ontologies.

As mentioned before, UML and OWL are not completely compatible, i.e. some features exist in one but not the other, so translating UML to OWL may lose some semantics. However, the loss of semantics does not occur in this research due to the four-level ontology framework (see the next section). In the four-level ontology framework, the UML to OWL translation is only necessary for the UML models that are accessible to data consumers, which means these UML models only contain concrete class definitions and public members. Therefore, the lack of support for abstract or interface classes and the lack of access control features in OWL will not be a problem.

Even for features that are supported by both UML and OWL, the mappings from UML to OWL are not always straightforward, especially when mappings can be carried out in different ways or when certain conventions need to be adopted. For example, when translating UML constructs to OWL constructs, consistent naming conventions are usually required to ensure the uniqueness of each translated OWL construct, since “... in OWL a property is defined by default as having range and domain both Thing” meaning that a property name, by default, has global scope; whereas “... in UML the scope of a property is limited to the subclasses of the class on which it is defined” (Hart et al. 2006).

Since this research by no means aims to provide a comprehensive analysis or implementation of the UML to OWL translation, the UML models used in this research are kept as simple and demonstrative as possible. For example, globally unique names are used for classes and properties. Finally, it is worth noting that the UML to OWL translation is already supported by tools, such as Protégé (Stanford Center for Biomedical Informatics Research 2008a) and UML2OWL (Hillairet 2007). They address the above issues in various ways, but they offer no significant advantages to this research as those issues are avoided. As an example, Appendix A
demonstrates a UML to OWL translation using Protégé.

The Four-Level Ontology Framework

To solve the heterogeneity problem in this research, introducing only an ontology is not enough. A two-way translation process is also required to perform translations between the data schema of an organization’s internal database and the ontology. In many real life environments, more than one ontology is usually involved, such that between each pair of ontologies, there needs to be a two-way translation process. The aggregation of those related ontologies and translation processes is then referred to as an ontology framework, such as the four-level ontology framework developed by this research, and a few others (Hepp & Roman 2007, Flahive et al. 2005, Klinc 2007, Motik & Volz 2005).

Figure 3.9: A four-level ontology framework by combining Sheth and Larson’s five-level schema framework for FDBS and the OMG’s notions of PIMs and PSMs in the MDA.

The four-level ontology framework developed by this research, as illustrated in Figure 3.9, is based on three approaches: ontology, five-level schema framework and elements of MDA. Within this framework, an ontology may fulfill one of the following five roles:

Local PSM  In this role, an ontology describes an organization’s domain of interest and reflects the specific platform managing the data. Each organization is likely to have quite different ontologies at this level of abstraction. An ontology in this role is analogous to what Sheth and Larson refer to as a local schema. A data set that conforms to this ontology is referred to as the local data set.

Local PIM  In this role, an ontology also reflects an organization’s domain of interest but in a platform-neutral manner. An ontology in this role is analogous to what Sheth and Larson refer to as a component schema.
Export PIM In this role, an ontology defines those parts of the Local PIM for which each organization is contributing data to form the distributed data set. Such an ontology will be a subset of the Local PIM ontology. A data set that conforms to this ontology is referred to as an export data set.

Distributed PIM In this role, an ontology will define the entire distributed data set available to organizations participating by either providing components of the distributed data set, i.e. data providers, or consuming components of this data set as a local copy, i.e. data consumers, or both. At least one such ontology is expected to be defined in a distributed environment. A data set that conforms to this ontology is referred to as a distributed data set.

Import PIM In this role, an ontology will define the local copy of the distributed data set that is of interest to an organization that consumes the distributed data set.

An ontology fulfilling Export PIM, Distributed PIM and Import PIM corresponds to Sheth and Larson’s notion of an export, federated and external schemas respectively. The roles of Export and Import PIMs are regarded as being at the same level of abstraction and will be defined by an organization acting as a provider or a consumer respectively, while each of the three remaining roles are regarded as being defined at distinct levels of abstraction; therefore, there are four levels of abstraction in this ontology framework.

It is worth noting that a data consumer’s Import PIM ontology only needs to be a subset of the Distributed PIM ontology, and does not need to have any relation to the Export PIM ontologies of the data providers to which this consumer is subscribed, apart from that they are all conforming to the same Distributed PIM ontology. Figure 3.10 illustrates another view of the relationships between various roles of ontologies and organizations. Each circle represents a data set that conforms to an ontology, whose role is labelled inside the circle. Two circles represent the same data set if they are linked by a dashed line, even though their shapes and sizes may be slightly different.

By using this four-level ontology framework, not only the ambiguity of communication between organizations can be removed, but also the heterogeneity resulting from conceptual differences in the way each organization views the shared data, and the heterogeneity resulting from each organization using different implementation specific technologies for managing and processing this data, can be clearly distinguished.

As already indicated in Figure 3.10 in this research, the four-level ontology framework is used by both data providers and consumers. Data supplied by providers are transformed through three distinct ontologies, from the Local PSM to the Local PIM and finally to the Export PIM ontology, before it is consumed by data consumers. The similar but reverse process is also used by data consumers to transform the data received from one or more providers into the form used within the consumers’ local databases.

During the data transformation process, although a data consumer can define the Import PIM ontology purely based on the Distributed PIM ontology, due to the lack of a FDBS for managing the distributed data set in this research environment, the data consumer still has to deal with numerous Export PIM ontologies if it has to subscribe to multiple data providers to get its desired information.

Since the main objective of this research is to try to improve data replication’s flexibility and performance by using ontology-based update policies, rather than to execute actual replications and integrate data changes, the above data transformation process is not focused on in this
3.4 System Architecture

Most of the construction components of the selective data replication system have already been illustrated in Figure 3.4 and briefly discussed in Section 3.2. However, those components were viewed more from the replication workflow perspective rather than from the system architectural perspective. In this section, therefore, the components that make up the selective data replication system are arranged into layers, and the responsibility of each layer is discussed. Figure 3.11 presents the full architecture of the selective data replication system, which is almost identical to the one shown in Figure 3.4 but with clearly defined layers.

The architecture of data-provider side consists of three layers, which, from bottom to top, are the preparation layer, the export layer and the interface layer. The preparation layer contains components and processes for preparing data, data schemas and data updates. They are formatted by the export layer components and subsequently made accessible in the interface.

Figure 3.10: Relationships between various roles of ontologies and organizations.
layer to the subscribed data consumers.

Similarly, the data-consumer architecture also has three layers, which, from bottom to top, are the interface layer, the evaluation layer and the integration layer. The interface layer is mainly responsible for establishing connections and receiving information from data providers, such as data providers’ Export PIM ontologies and update notifications. Information received is passed down to the evaluation layer to be evaluated based on consumer-specified update policies. The evaluation layer also has the responsibility of initiating the data replication process if significant data updates are identified as the result of the update policy evaluation. Finally, the integration layer elements format and integrate the retrieved updated data into the consumers’ local databases.

This layered architecture provides a separation between different groups of functional components and business logic, allowing the implementation of the selective data replication system to be more manageable and extensible. Further advantages of this architecture and how the implementation benefits from it are elaborated in Chapter 6.

In the following chapters, the design and implementation details of the components included
in Figure 3.11 are elaborated. More specifically, Chapters 4 and 5 discuss the design insights with regard to the update notification and update policies respectively. Chapter 6 drills down to the implementation level and talks about the selective data replication system from the database management and coding perspectives.
Chapter 4

Update Notification

An update notification is a message that contains the descriptions of data updates, which are used in the selective data replication system by data consumers to evaluate their update policies, in order to decide if data replications are necessary. Without update notifications, data consumers’ update policies cannot be evaluated, hence selective data replications can never occur.

This chapter is divided into five sections, in each of the first four of which is answered one of the following questions: “Who should provide update notifications?”, “What does an update notification look like?”, “How are consumers notified with updates?” and “How are update notifications managed?”.

4.1 Who should provide update notifications?

In the early chapters, while mentioning the update notification, it was assumed that an update notification is provided by a data provider. This assumption is usually true and has its reasons, which are that the data provider has complete access to its shared data set and knows everything about the changes, so that the provider can use the most effective and efficient way to capture data changes and use them to generate update notifications.

However, as mentioned in Chapter 2, data providers and consumers can belong to different organizations and may not have an awareness of each other; therefore, a data provider has no obligation to provide update notifications to the subscribed consumers. Under such circumstances, a data consumer can only synchronize its data by taking a snapshot of the provider’s shared data set and using it to completely replace its own local copy. Snapshot replication, like the row-level and procedural replication mentioned in Chapter 2, is one of the existing methodologies for converging data. Although snapshot replication guarantees the synchronization of data, it replicates both updated and non-updated data, so unless other replication methods fail, or are incapable of ensuring the data integrity, snapshot replication will not be used.

To make selective data replication useful, this research assumes all the participating data providers do provide update notifications as one of their services to data consumers. Chapter 6 Section 6.1.2 provides the implementation details on capturing data updates and generating update notifications.
CHAPTER 4. UPDATE NOTIFICATION

4.2 What does an update notification look like?

An update notification is a message container that encapsulates the descriptions of data updates, and is sent across a network from a data provider to a data consumer. Without the encapsulated information, the update notification itself does not possess any meaning.

Information contained within each update notification can be grouped into those components concerned with administering the notifications themselves and those concerned with describing the updates. These are discussed in Sections 4.2.1 and 4.2.2, respectively.

In this research, an update notification is represented as an XML-encoded document with the root element `<UpdateNotification>`.

4.2.1 Administering Update Notifications

Administrative components of an update notification can be used by both data providers and consumers for tasks such as identifying the source of notifications, keeping track of notification versions and determining updated feature types. These administrative components are:

Provider ID (`ProviderId`), which is an unique identifier for each data provider within the system, so that the source of notifications can be identified.

Start Time (`StartTime`), which specifies the start date and time used as a constraint for querying the updated data when preparing the update notification, i.e. all the data described in the notification were updated after the value of this component.

End Time (`EndTime`), which specifies the end date and time used as a constraint for querying the updated data when preparing the update notification, i.e. all the data described in the notification were updated prior to the value of this component.

Object Type (`ObjectType`), which is equivalent to the meaning of “Feature Type” in GIS, specifies the type of the updated objects included in this update notification. In this research, each update notification only contains updated feature instances of the same type. If more than one type of feature instance is updated, multiple update notifications need to be created - one for each feature type. The word “object” is used in here instead of “data record” or “row” because, as mentioned in Section 3.3, update notifications are generated based on the Export PIM ontology, which is object-oriented, rather than the relational database schema, which is record or row-oriented.

ID Field (`IdField`), which specifies the identity or key field of the object type. In the current implementation, each object type must be able to be uniquely identified by one identification field. In future, use of multiple identification fields can be supported.

`Is Full` (`IsFull`), which specifies if the update notification contains all the available fields and values of the embedded updated objects. `IsFull` is discussed in detail in Section 4.2.2.

The above administrative components are represented as XML attributes of `<UpdateNotification>` as illustrated in Listing 4.1.

4.2.2 Describing Updates

From the database management perspective, an update could be either a data update or a schema update. A data update may be caused by data insertion, deletion or modification, none
Listing 4.1: The XML representation of an update notification with administrative components.

```xml
<UpdateNotification ProviderId="" StartTime="" EndTime="" ObjectType="" IdField=""
IsFull="">
...
</UpdateNotification>
```

of which alters the metadata objects of a database, such as tables, columns and views. However, a schema update may be caused by the insertion, deletion or modification of one or more of a database’s metadata objects, and may also subsequently result in data updates; for example, deleting an existing table deletes all the data records in that table. As mentioned in Chapter 2, discussing the schema update is beyond the scope of this research, hence update notifications in this research only contain descriptions of data updates - data update descriptions, or simply update descriptions.

Data update description is the core of an update notification, as only the description will be eventually extracted and used for update policy evaluation.

In general, the more descriptive an update description is, the more selective an update policy can be. For example, if the field `sururb` is not included in an update description, having a policy rule such as `sururb = 'Downtown'` is useless since it cannot be evaluated. On the other hand, if an update description contains too much detail, processing the notification itself may become an overhead. The extreme scenario is when all the field values of the updated data are included in a notification, in which case, when the notification is received by a data consumer, the full copy of the data will also be received. If the consumer continues carrying out the policy evaluation and data replication processes, the full copy of the data will be retrieved twice, causing a significant overhead. This situation can happen if a data consumer’s Import PIM ontology is a superset of a provider’s Export PIM ontology, since the full copy of the data to the provider may not be the full copy to the consumer. Even if the consumer is smart enough to stop performing the subsequent processes, the selective data replication system then behaves no differently than a traditional row-level replication.

In order to find the balance in between, an update description needs to be able to be customized by data consumers. For example, if the `Address` feature type has seven fields: `gid`, `houseLow`, `houseHigh`, `roadName`, `roadType`, `address`, and `geometry`, and one consumer only defines policy rules based on the `geometry` field, then `geometry` will be the only field in the data update description transmitted to this consumer. However, there is nothing to prevent another consumer from defining policy rules based on all seven fields; therefore, in this case, the update notification will be indicated as `IsFull = “True”`, which means each embedded update description contains full information on the updated object, and can be integrated straightaway into the consumer’s local data set. In other words, if an update notification is indicated as `IsFull = “True”`, the selective replication will behave the same way as the traditional row-level replication.

However, depending on the size of each field, for example, including six, or even five out of seven fields may also introduce unnecessary performance overhead. Therefore, ideally, whenever such undesired performance overhead occurs, the selective data replication system on the data provider side will indicate the corresponding update notification as `IsFull = “True”`, and include all the field values of the updated objects. This behaviour is referred to as “Smart Fallback”, meaning fallback to the traditional row-level replication. Chapter 7 evaluates and analyzes the performance impact related to the amount of information included in an update notification,
so that data providers may decide when “Smart Fallback” should take place.

To make the content of a data update description flexible and able to be customized, as shown in Listing 4.2, each update description is rooted at the XML element DataUpdateDetail containing a collection of Property elements, each of which has Name and Value attributes specifying the field name and value of the updated object respectively. As can be seen in Listing 4.2, a DataUpdateDetail element also contains three XML attributes; they are:

ID (Id), which specifies the value corresponding to IdField, which is one of the administrative components mentioned in Section 4.2.1.

Update Type (UpdateType), which specifies whether this update was caused by data insertion, deletion or modification. Its value can only be one of the followings: INSERT, DELETE or MODIFY.

Update Time (UpdateTime), which specifies the exact date and time when this object was updated. This time must be between the StartTime and EndTime administrative components.

Listing 4.2: The XML representation of a data update description.

```
<DataUpdateDetail Id="" UpdateType="" UpdateTime="">
  <Property Name="" Value=""/>
  <Property Name="" Value=""/>
  ...
</DataUpdateDetail>
```

The XML representation of an entire update notification skeleton is shown in Listing 4.3, where one or more DataUpdateDetail elements are grouped into DataUpdateDetails element under the UpdateNotification root element. Such an arrangement is to allow for easy extension in the future to include schema updates as well.

Listing 4.3: The XML representation of an update notification with all of its components.

```
<UpdateNotification ProviderId="" StartTime="" EndTime="" ObjectType="" IdField=""
  IsFull="">
  <DataUpdateDetails TotalInsertions="" TotalDeletions="" TotalModifications="">
    <Property Name="" Value="" UpdateType="" UpdateTime="">
      <Property Name="" Value=""/>
      <Property Name="" Value=""/>
      ...
    </DataUpdateDetails>
  </DataUpdateDetails>
</UpdateNotification>
```

The only information in Listing 4.3 that has not been explained is the three attributes of DataUpdateDetails: TotalInsertions, TotalDeletions and TotalModifications. The values of these three attributes are calculated on the fly when an update notification is generated. The purpose of including these attributes is to simply provide some basic statistical information about the embedded data updates and provide data consumers with some general ideas on the nature and the scope of those updates.

Listing 4.4 demonstrates a small example of what a real update notification looks like.
4.3. HOW ARE CONSUMERS NOTIFIED WITH UPDATES?

A local copy is updated when a data consumer becomes aware that this data differs significantly from that which is available from the providers, as they change the export data set to reflect changes to the relevant real world phenomena. In this research, consumer awareness typically occurs in one of two ways: either providers broadcast update notifications to all the subscribed consumers whenever their export data sets have been updated, or consumers periodically check with data providers for updates. These two methods are referred to as “push” and “pull” respectively, and both of them are explained in this section.

4.3.1 The “Push” Method

The “push” method, as illustrated in Figure 4.1, allows data consumers to receive update notifications as soon as they are published by data providers. The advantage of this method is to give data consumers access to the latest updates in a timely fashion.

Figure 4.1: Update notification propagation via the “push” method.
In this research, to avoid causing unnecessary performance overhead mentioned in Section 4.2.2, at the time of subscription, a data consumer can provide a list of feature types to which it wants to subscribe along with the fields that are used in its update policy, so that this information can be stored in the provider’s database, and used to generate “pushed” update notifications to the consumer. If the fields used in the update policy are changed, the data consumer simply needs to update the provider’s copy with the new set of fields.

The disadvantage of the “push” method is that, in some cases, data consumers do not need such timely updates. For example, in the third case study scenario in Section 2.4, \( C_{Police} \) and \( C_{Hospital} \) do not need any updates from \( F_{CCC}'s \) \( \Lambda_{CCC,Building} \) and \( \Lambda_{CCC,Road} \) data sets, until a specific business requirement is met. Therefore, as an alternative, the “pull” method can be used.

4.3.2 The “Pull” Method

Instead of passively waiting for update notifications to be broadcast by data providers, a data consumer can also use the “pull” method to proactively check for updates according to its business requirements for data synchronization. Such business requirements could be either based on time frequency or other specific events, e.g. to update a feature type when another feature type is updated, like the “Disaster Control” scenario described in Section 2.4. Figure 4.2 illustrates the communication when a data consumer “pulls” update notifications from a provider.

![Figure 4.2: Update notification propagation via the “pull” method.](image)

The advantage of the “pull” method is that data consumers have entire control over the initiation of the update notification retrieval process, and, depending on the implementation, can specify additional constraints to further customize the returned update notifications each
4.4. HOW ARE UPDATE NOTIFICATIONS MANAGED?

The disadvantage, however, is that consumers may not get critical updates in a timely fashion.

4.3.3 Using Both “Push” & “Pull”

In this research, both the “push” and “pull” methods are implemented, and they can be used either independently or conjunctively to achieve the maximum flexibility. For example, as shown in Figure 4.3, one data consumer may use both “push” and “pull” to retrieve update notifications from multiple data providers.

![Diagram](image-url)

Figure 4.3: The data consumer may require update notifications from a second data provider after receiving an update notification from the first one; therefore, in this case, both the “push” (left) and the “pull” (right) methods are adopted.

4.4 How are update notifications managed?

Since update notifications are generated by data providers and then propagated to data consumers for utilization, the management of update notifications applies to both data providers and data consumers.
4.4.1 Consumer Side Notification Management

In this research, update notifications are used by data consumers to evaluate their update policies, in order to determine whether some of the data updates embedded in the notifications are significant enough to trigger data replications. Insignificant data updates are simply ignored. However, ignoring an update description should only be temporary, since at a later stage when the update policy is changed, the previously ignored insignificant updates may now become significant. Therefore, there may be a need to store the received update notifications, for example, in the consumer’s local database.

There is another reason why update notifications need to be made persistent. As will be mentioned in Chapter 5, when the selective data replication system is eventually migrated to an agent platform, the tasks of retrieving update notifications and using them for update policy evaluation could be assigned to, and performed by, different software agents; therefore, storing the retrieved update notifications in a storage medium becomes a requirement.

Update Notification Schema

Figure 4.4 illustrates the table data_updates used in this research to store update notifications. All of the table columns displayed in Figure 4.4 are self-explanatory and most of them can be mapped one-to-one to either the administrative or the content components of an update notification described in Section 4.2, except the column is_significant. The is_significant column is a new column and specifies whether the updated data is significant as determined by update policy evaluation, which will be discussed in Section 5.2. The default value of is_significant is false, but as the result of update policy evaluation, the value of this field can be set to true, indicating that this update is significant and needs to be replicated. Once the replication is finished, any records with is_significant = “true” can be removed.

<table>
<thead>
<tr>
<th>provider_id</th>
<th>object_type</th>
<th>id_field</th>
<th>id</th>
<th>property_name</th>
<th>property_value</th>
<th>update_type</th>
<th>update_time</th>
<th>is_significant</th>
</tr>
</thead>
</table>

Figure 4.4: The database table data_updates used to store update notifications.

Removing an is_significant = “false” record from data_updates may compromise data integrity. This is because once such a record is removed, it will be excluded not only from the update policy evaluation process, but also from the data replication process, which means that the updated data described by that record can never be synchronized unless a snapshot replication is performed. However, if some data updates are never going to be of interest to a data consumer, there is no harm in omitting them from the data_updates table.

There are two additional characteristics of the data_updates table that are worth mentioning. The first one is that isFull, which is discussed in Section 4.2.2 does not have a corresponding column in data_updates. This is because when isFull = “True”, the “Smart Fallback” feature will integrate the information embedded in the update notification into the local data set straight-away to ensure the notification does not trigger the policy evaluation and replication processes. In other words, all the update descriptions stored in data_updates have isFull = “False”.

The second characteristic is that, to handle the dynamic content structure of an update description, the columns property_name and property_value simply store the delimiter-separated field names and values included in the update description, respectively. Although this may
not be a preferable approach, it does increase the performance as the number of database IO (Input/Output) operations is reduced, compared to having multiple flat records or using header and detail tables. By using the same example from Listing 4.4 in Figure 4.5 it is demonstrated how the values of property_name and property_value are populated.

```
<UpdateNotification ...>
    <DataUpdateDetails ...>
        <DataUpdateDetail ...>
            <Property Name="address" Value="10A ST JAMES STREET"/>
            <Property Name="geometry" Value="0101000000BC7493D827B14241"/>
        </DataUpdateDetail>
        <DataUpdateDetail ...>
            <Property Name="address" Value="10B ST JAMES STREET"/>
            <Property Name="geometry" Value="0101000000BC7493D827B14241"/>
        </DataUpdateDetail>
    </DataUpdateDetails>
</UpdateNotification>
```

where ":;" is used as a delimiter in this example.

Figure 4.5: Populating the values of property_name and property_value in the data_updates table.

**Update Notification Ontology**

As mentioned earlier in Chapter 3, processes related to the update notification and update policy operate on the ontology level in this research; therefore, an ontology of update notification needs to be created in order for data consumers to specify update policy rules based on the constructs of a notification; for example, a policy rule may be specified against the update type: e.g. to only replicate data insertions and deletions, but not modifications. Figure 4.6 shows the update notification ontology represented using a UML class diagram.

```
+ providerId: String
+ objectType: String
+ idField: String
+ id: Integer
+ updateType: String
+ updateTime: Date
+ isSignificant: Boolean
```

Figure 4.6: The update notification ontology represented using a UML class diagram.

The update notification ontology shown in Figure 4.6 only contains one class - UpdatedObject, whose main purpose is to capture information that is related to the data update operation, such as updateType and updateTime, rather than to the actual updated objects, since the information describing the actual updated objects is already conforming to the Distributed PIM
ontology. The three properties: \texttt{objectType}, \texttt{idField} and \texttt{id} included in \texttt{UpdatedObject} are only for the purpose of cross-referencing the updated object.

### 4.4.2 Provider Side Notification Management

Update notifications are created based on data updates, which are captured and stored in providers’ local databases. From time to time, data providers may feel there is a need to purge previously captured data updates for various reasons, in which cases, if the purged data updates have not been previously retrieved by some data consumers, those consumers will experience data integrity problems for their local copies. For example, if a new record is inserted and the update about this insertion is purged before a subscribed consumer has a chance to know about it, the insertion can never be picked up by this consumer if no further changes are made to this record. Similarly, if a consumer misses a deletion update, the deleted data will stay in the consumer’s local copy indefinitely. If one of the above circumstances happen, the only way to synchronize a consumer’s local copy is to perform a snapshot replication.

To help both data providers and consumers to identify if a snapshot replication is required, each update notification is designed to have two time stamps: \texttt{StartTime} and \texttt{EndTime}. Although these two time stamps have already been introduced in Section 4.2.1, their usages have not yet been explained.

In selective data replication, when a data consumer “pulls” update notifications from a data provider, the provider will record the time when the request is received and then use this time as the \texttt{EndTime} to query the captured data updates. Once the update notification is generated and sent back to the consumer, the \texttt{EndTime} is saved together with the consumer’s subscription information in the provider’s database. When the next time a “pull” request is made from the same consumer, the previously saved \texttt{EndTime} will be treated as the \texttt{StartTime}, which will then be used in combination with a new \texttt{EndTime} to query the captured data updates.

In this way, when the provider tries to purge the previously captured data updates, it can check whether the deleted updates were captured after the \texttt{EndTime}. If at least one deleted data update was captured after the \texttt{EndTime}, the provider can inform this data consumer that a snapshot replication is required.

### 4.5 Summary

This chapter examined update notifications from various perspectives, including their creation, content, propagation and management. Generally, in this research, either periodically or upon data consumers’ requests, data providers will generate XML encoded update notifications based on the captured data changes. During update notification generation, techniques, such as creating dynamic update notification content and “Smart Fallback”, can be adopted to maximize the flexibility and to prevent performance overhead. Once the generated update notifications are received by data consumers using either the “push” or the “pull” method, they will be temporarily stored in the consumers’ local databases for update policy evaluation, which will be discussed in detail in Chapter 5.
Chapter 5

Update Policy

To exploit the selectivity in the selective data replication system, a data consumer can specify one or more rules, each of which contains a condition or conditions determining the circumstances under which the consumer’s local data set is to be updated or synchronized with the currently available data from the relevant data providers. These rules primarily reflect the higher level business requirements and constraints unique to each data consumer. The collection of these rules is referred to as an update policy. Update policy is one of the key ingredients of the selective data replication system, as without it the selective power is lost and the replication behaves no differently than a traditional row-level replication.

This chapter discusses two aspects of update policy: specification and evaluation. For update policy specification, according to the high-level solutions mentioned in Section 3.3, update policies need to be specified based on ontologies in this research; therefore, Section 5.1 discusses how an update policy is specified based on ontologies using one of the rule languages. The evaluation of an update policy is relatively straightforward and its workflow is described and illustrated in Section 5.2.

5.1 Update Policy Specification

In many real world policy examples, a policy is usually constructed by a collection of rules, each of which contains one or more equality comparisons comparing various properties of the context upon which the rule is based, and those equality comparisons are then joined together by boolean operators such as AND, OR and NOT.

The context of a rule is usually an entity or a class representing a collection of objects of the same type; the properties of the context are like the properties or attributes of the class, describing the different characteristics of that type. For example, a casino may have a policy that contains a rule: “You must be 21 years of age or older, and you cannot wear jeans.” The context of the rule is Customer, representing all the casino customers. This rule contains two property comparisons: “Age $\geq 21$” and “Jeans $\notin$ Clothes”, which are connected by the “AND” operator.

The update policy specification in this research follows the same convention as above - each update policy rule needs to have two basic constructs: the context and the comparison(s) using the properties of the context.

In a geographical information environment in which this research resides, the context of an update policy rule is usually a geospatial feature type, such as River and Coastline. In the first
two of the three case study scenarios described in Section 2.4, each of their update policies only contains one rule, and the context of that rule is the geospatial feature type Address and Parcel respectively. Once the context of a rule is identified, the properties of this context can be referenced from within the rule for defining conditions.

In this research, each update policy rule is allowed to have multiple contexts, which is essential when a rule needs to exploit relationships among multiple types. For example, the update policy of the third case study scenario contains three rules, with the last two having two contexts: DisasterWarning and Road, or DisasterWarning and Building, since the rule needs to associate these two contexts by checking if an instance of Road (or Building) is within a certain range surrounding a disaster location, whose value comes from a DisasterWarning instance.

5.1.1 Ontology-Based Policy Specification

An update policy has to be evaluated with data; therefore, when specifying an update policy, it is best to use the same vocabulary as the one used by the data, so that additional mapping or translation processes will not be required during policy evaluation. As mentioned in Section 3.3, the best way to keep the vocabulary the same and to remove unnecessary mapping and translation processes, is to introduce and enforce ontology conformity. To leverage the advantages of ontology, update policies in this research are specified based on ontologies, which means the contexts of a policy rule and the properties of the contexts that are referenced from within the rule have to be based on the constructs of the relevant ontologies.

Policy Specification based on Import PIM Ontology

The purpose of introducing update policies in this research is for data consumers to evaluate the information embedded in update notifications, in order to determine if data replications are necessary. Therefore, as mentioned above, an update policy needs to be specified based on the ontology to which the information included in the update notifications conform.

Although an update notification is generated by a data provider, and the embedded information conforms to the provider’s Export PIM ontology, once a data consumer decides to subscribe to this provider, the ontology constructs of the subscribed feature types will also become part of the consumer’s Import PIM ontology, which means any update notifications generated for those feature types are already in conformity with the consumer’s Import PIM ontology.

When an update policy rule is specified based on a consumer’s Import PIM ontology, the data that is used to evaluate this rule only needs to come from the data_updates table, which contains all the data update descriptions retrieved from providers, as mentioned in Section 4.4.

Policy Specification based on Update Notification Ontology

A data consumer’s Import PIM ontology is one that describes data, not update operations. As mentioned in Section 4.4, this research provides an update notification ontology which encapsulates information related to an update operation, such as UpdateType and UpdateTime. This ontology is necessary, since in some occasions, a data consumer might want to specify its update policy based on, for example, the type of updates or the time when the updates occurred. The update notification ontology is usually used in combination with the Import PIM ontology.

When an update policy rule is specified based on the update notification ontology, the data that is used to evaluate this rule comes from the data_updates table.
5.1.2 Choice of Rule Languages

There are a number of rule languages developed in various application domains for representing rules in different forms and for different purposes. A few commonly seen rule languages are: Frame Logic (F-Logic), Java Expert System Shell (JESS), Rule Markup Language (RuleML), Jena, JBoss, Semantic Web Rule Language (SWRL) and Object Constraint Language (OCL). Although all these rule languages are theoretically capable of representing update policy rules used in this research, only OCL and SWRL are considered the best candidates as they can be naturally integrated into the two ontology representation languages adopted in this research: UML and OWL, which are mentioned in Section 3.3.4.

Object Constraint Language (OCL)

OCL (OMG 2005a) is a language that enables one to describe expressions and constraints on object-oriented models. Ideally, when UML is used as the ontology representation language in this research, OCL should be used for update policy specification since it is part of the UML 2.0 specification (OMG 2005b).

However, upon further investigation and experimentation, it was discovered that OCL has a number of disadvantages, which may complicate the rule specification logic and cause difficulties with the rule translation and evaluation processes later.

One of the major limitations of OCL is that each OCL invariant only allows a single context; therefore, to navigate from one object type to another, it is required either to use the explicitly defined relationships between the two object types, or to traverse through the meta-model objects, such as Class and Package, which involves a large amount of collection operations and type castings, significantly complicating the rule specification and polluting the business logic. In addition, since this research resides in a geographical information environment where spatial features relate to each other via geographical collocation rather than explicit relationships among their feature types, using OCL to define update policies in this research is not a feasible solution.

Another difficulty in using OCL in this research is the lack of support for translating and evaluating OCL invariants. As will be discussed in Section 5.2 regardless of the choice of rule language, update policy rules will be eventually imported into a JESS rule engine for evaluation; therefore, if OCL was used, all the OCL invariants defined in the class diagram would need to be converted into JESS rules. Although a number of rule translators developed by the REWERSE Working Group II (REWERSE Working Group II 2008) can translate OCL invariants into JESS rules using R2ML as the interchange format, their implementations are still incomplete and some key features required by this research are not yet supported. For example, some of the OCL collection operations are not supported since the R2ML does not support collection operators as OCL does (Milanović et al. 2008).

Semantic Web Rule Language (SWRL)

SWRL (Horrocks et al. 2004) is a rule language proposed by the World Wide Web Consortium (W3C) in 2004 for defining rules and formal semantics for OWL ontologies. Since SWRL is designed to work with OWL and supports both the OWL Lite and the OWL DL sub-languages, in this research, when ontologies are encoded in OWL, SWRL is the most suitable rule language for specifying ontology-based update policies.

In addition, the aforementioned problems with OCL do not apply to SWRL; therefore, this research simply uses SWRL as the only policy specification language regardless of which
ontology representation language is adopted.

5.1.3 Update Policy Specification using SWRL

SWRL Syntaxes

There are three types of very distinguishable syntaxes for writing a SWRL rule: the abstract syntax, the human readable syntax and the XML syntax. The abstract syntax of SWRL is a high-level syntax that is very explicit and verbose. The human readable syntax, as its name suggests, is more human readable, but is a relatively informal syntax. (Horrocks et al. 2004) The following uses the first case study scenario described in Section 2.4 to demonstrate what an update policy rule will look like if it is expressed in both the abstract and the human readable syntax of SWRL.

Abstract Syntax:

\[
\text{Implies(Antecedent(UpdatedObject(I-variable(x)))
\text{isSignificant(I-variable(x) I-variable(s))
swrlb:equal(I-variable(s) false))
objectType(I-variable(x) I-variable(t))
swrlb:equal(I-variable(t) "Address"))}
\]

\[
\text{Consequent(isSignificant(I-variable(x), true)))}
\]

Human Readable Syntax:

\[
\text{UpdatedObject(?x) \land isSignificant(?x, ?s) \land swrlb:equal(?s, false) \land objectType(?x, ?t) \land swrlb:equal(?t, "Address") \rightarrow isSignificant(?x, true)}
\]

As can be seen from this simple example, the human readable syntax is much easier to read and to understand - what the rule means is: for an UpdatedObject object, if its isSignificant property equals false and its objectType equals “Address”, then set its isSignificant property to true. It is worth noting that SWRL does not support negated atoms or disjunction (Stanford Center for Biomedical Informatics Research 2008); however, the lack of these features does not limit the expressive power of the language.

The third syntax is the XML syntax, which is further subdivided into the XML concrete syntax and the RDF (Resource Description Framework) concrete syntax (Horrocks et al. 2004). The XML concrete syntax is a combination of the RuleML and the OWL XML presentation syntax, whereas the RDF concrete syntax is based on the OWL RDF/XML exchange syntax. Translations between the XML and the RDF concrete syntaxes can be easily accomplished by using an XSL transformation.

As shown in Listing 5.1 although the XML syntax of SWRL is even more cumbersome to write, it does offer several advantages over the other two (see the second half of Listing A.2 for a complete example of a SWRL-encoded policy). One of its main advantages is that the XML syntax can be more easily processed by software applications and translated into other rule languages, such as RuleML, so that the interoperability between SWRL and other rule languages is simplified. This advantage could be further exploited by this research to allow the selective data replication system to work with update policies that are specified in multiple rule languages. Such enhancement is identified as a future work item in Chapter 8.

Another main advantage of the XML syntax, which is of more concern to this research, is that it can be seamlessly integrated into OWL documents, making update policy rules coexist with the ontology constructs, so that they can be treated as a whole for storage and processing.
5.1. UPDATE POLICY SPECIFICATION

Listing 5.1: Snippets of $C_{PostOffice}$'s update policy rule expressed in SWRL's RDF concrete syntax.

```xml
<swrl:Imp rdf:ID="ReplicateEveryAddressUpdate">
  <swrl:body>
    ...
    <swrl:DatavaluedPropertyAtom>
      <swrl:argument1>
        <swrl:Variable rdf:ID="x"/>
      </swrl:argument1>
      <swrl:propertyPredicate rdf:resource="#isSignificant"/>
      <swrl:argument2 rdf:resource="#s"/>
    </swrl:DatavaluedPropertyAtom>
    ...
    <swrl:BuiltinAtom>
      <swrl:arguments>
        <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">false</rdf:first>
        ...
        <rdf:first rdf:resource="#s"/>
      </swrl:arguments>
    </swrl:BuiltinAtom>
    ...
    <swrl:DatavaluedPropertyAtom>
      <swrl:argument1 rdf:resource="#x"/>
      <swrl:propertyPredicate rdf:resource="#objectType"/>
      <swrl:argument2 rdf:resource="#t"/>
    </swrl:DatavaluedPropertyAtom>
    ...
    <swrl:BuiltInAtom>
      <swrl:arguments>
        <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">true</rdf:first>
        ...
        <rdf:first rdf:resource="#t"/>
      </swrl:arguments>
    </swrl:BuiltInAtom>
    ...
    <swrl:DatavaluedPropertyAtom>
      <swrl:argument2 rdf:resource="http://www.w3.org/2001/XMLSchema#string">Address</rdf:resource>
      <swrl:argument1 rdf:resource="#t"/>
    </swrl:DatavaluedPropertyAtom>
    ...
    <swrl:DatavaluedPropertyAtom>
      <swrl:argument1 rdf:resource="#x"/>
      <swrl:propertyPredicate rdf:resource="#isSignificant"/>
    </swrl:DatavaluedPropertyAtom>
    ...
    <swrl:DatavaluedPropertyAtom>
      <swrl:argument2 rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
    </swrl:DatavaluedPropertyAtom>
  </swrl:body>
  <swrl:head>
    ...
    <swrl:DatavaluedPropertyAtom>
      <swrl:argument2 rdf:datatype="http://www.w3.org/2001/XMLSchema#boolean">true</rdf:datatype>
      <swrl:argument1 rdf:resource="#x"/>
      <swrl:propertyPredicate rdf:resource="#isSignificant"/>
    </swrl:DatavaluedPropertyAtom>
    ...
    <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
  </swrl:head>
</swrl:Imp>
```
There are a number of tools capable of reading, writing and evaluating OWL ontology documents with embedded SWRL rules. In this research, the tool Protégé (Stanford Center for Biomedical Informatics Research 2008a) is used for this purpose. Protégé is an ontology editor and it supports ontologies in a variety of formats including RDF(S), OWL, XML Schema and UML. The usage of Protégé, for various implementation tasks of this research is further explained in Chapter 6.

Both the human readable and the RDF/XML syntaxes of SWRL are used in this research interchangeably for update policy specification, depending on which ontology representation language is used. The following two subsections discuss and demonstrate the ways SWRL rules are specified when ontologies are represented using OWL and UML respectively.

**SWRL with OWL**

As mentioned before, because the XML syntax of SWRL can be naturally integrated into OWL documents, when an ontology is encoded in OWL, SWRL rules are specified in the RDF/XML syntax, like the example shown in Listing 5.1.

Since writing SWRL rules in the RDF/XML syntax is cumbersome, tools such as Protégé can become useful to assist with rule specification. For example, in Protégé, as shown in Figure 5.1, a SWRL rule can be directly entered in the human readable syntax, and will be automatically parsed and stored in the corresponding RDF/XML syntax.

![Figure 5.1: An example of how SWRL rules are defined in Protégé.](image)

**SWRL with UML**

When UML is used as the ontology representation language, each SWRL rule is specified in the UML class diagram as a single stereotyped note `<<UpdatePolicyRule>>`. A note is only allowed to contain one rule, and if there are multiple notes, their embedded rules are disjoined (OR-ed) together to form the update policy.

As mentioned in Section 3.3.4 in this research, UML models are firstly translated into OWL ontologies before being further processed, and the translation usually requires certain conventions, such as the naming conventions for classes and properties, to be adopted. Therefore, when authoring SWRL rules for a UML class diagram, these conventions need to be taken into consideration, since the SWRL rules will be eventually merged into the translated OWL ontology. For simplicity and demonstration purposes, constructs in UML models used in this research have unique names, which can be referenced directly in SWRL rules. In future, as will be discussed in Section 8.1.4 when direct OCL support is enabled, neither the UML-to-OWL translation nor the various conventions will be required.

Continuing with the first case study scenario as the example, Figure 5.2 demonstrates the embedding of an update policy in an UML represented ontology.

When update policy rules are specified in this way, the entire UML class diagram can be exported into an XMI (XML Metadata Interchange) document, which contains all the modelling constructs of the original UML class diagram, including the stereotyped notes with all the SWRL rules.
5.2. UPDATE POLICY EVALUATION

During update policy evaluation, SWRL rules embedded in XMI documents can be identified and extracted automatically by software applications. This process is described in Section 5.2 and its implementation is elaborated in Chapter 6.

5.2 Update Policy Evaluation

The goal of update policy evaluation is to produce a Boolean result for each retrieved data update description, by using consumer-specified update policy rules and the relevant data from the consumer’s database. When at least one “true” is yielded as the result of evaluation, i.e. at...
most one data update is being identified as a significant update, a data replication process will be carried out. Figure 5.3 illustrates the workflow for the update policy evaluation process.

![Diagram](image)

Figure 5.3: The workflow for update policy specification and evaluation.

In this research, update policy evaluation is performed by a rule engine. Before an update policy is evaluated for the first time, all the update policy rules have to be extracted from ontology documents and imported into the rule engine as rules. During the importation, it is necessary to translate the SWRL rules into the language that can be understood by the rule engine. Once all the rules have been successfully imported, the rule engine can be started and left running indefinitely until one or more update policy rules are altered, which requires re-importation of the altered rules and restarting of the rule engine.

The rule engine used in this research is JESS (Java Expert System Shell), which offers an efficient rule processing algorithm and has an extensible development environment (Sandia National Laboratories 2008). In addition, OWL and SWRL to JESS translation has already been supported by the tool Protégé, making it easier for this research.

Update policy evaluation also requires data, which, as mentioned in Section 5.1.1, comes from the data_updates table. Once the data is ready, it can be fed into the JESS rule engine at any time and it will be stored as facts inside the rule engine’s knowledge base. While the rule engine is running, it continues scanning the knowledge base for any facts that satisfy at least one rule; whenever satisfaction of a rule is determined, an actual data replication process will be scheduled to execute.

The implementation of this workflow is elaborated in Section 6.2.2.
5.3 Summary

In this chapter, the second of the two essential components of the selective data replication system - update policy is discussed. By defining an update policy, each data consumer can flexibly replicate only the data updates that are of interest to its business requirements, thereby resulting in a potentially reduced network traffic and improved replication performance.

To maximize the flexibility of update policy, each policy rule can be specified based on either a data consumer’s Import PIM ontology, the update notification ontology introduced in Chapter 4, or both. In this research, the language that is used for update policy rule specification is always SWRL regardless of the adopted ontology representation language; in future, however, rule languages other than SWRL can also be supported. Finally, the task of update policy evaluation is delegated to a rule engine, which determines if a data update is a significant update, thereby deciding if a data replication is necessary.

The implementation of the policy evaluation process is discussed in Chapter 6, followed by Chapter 7 with the evaluation of the selective data replication system, in order to analyze whether using update policies to selectively replicate data updates does indeed result in reduced network traffic and improved replication performance.
Chapter 6

Implementation

In the previous two chapters, the design insights of the two essential components of the selective data replication system: update notification and update policy were discussed in detail. This chapter focuses on the implementation of those two components, together with the rest of the selective data replication system.

The architecture of the selective data replication system, which was initially presented in Section 3.4, is re-displayed here in Figure 6.1. The architecture can be divided into two parts: the data provider part and the data consumer part, and the implementation detail of each part...
is elaborated in Sections 6.1 and 6.2 respectively.

Throughout this chapter, the “Post Office” scenario described in Section 2.4 is employed for the demonstration purpose. In this scenario, \( P_{CCC} \) is the data provider and \( C_{PostOffice} \) the data consumer.

The source code for the implementation of the selective data replication system is partially included in Appendix B.

6.1 Data Provider Implementation

As described in Chapter 3, each data provider is responsible for two essential tasks that are unique to the selective data replication system: producing the Export PIM ontology and generating update notifications, the implementations of which are explained in Sections 6.1.1 and 6.1.2 respectively. Section 6.1.3 describes the construction of the web service interface, which is another important component in the provider part of the architecture for exposing services. In addition, Section 6.1.3 also briefly describes some other tasks for which a provider is responsible, even though they are not unique to the selective data replication system, such as the subscription management and the data replication.

6.1.1 Producing the Export PIM Ontology

Producing an Export PIM ontology is an important process for a data provider, since, as mentioned in Section 3.1, this ontology is accessed and examined by one or more data consumers before they decide whether to subscribe to this provider.

Section 3.3 described two ways to represent ontologies in this research: either using UML class diagrams or OWL.

Producing the Export PIM Ontology in UML

Since the ontologies used in this research do not require any features from UML version 2.0 or above, there is no restriction on the choice of a CASE (Computer Aided Software Engineering) tool, as long as it can create UML class diagrams and export those diagrams to XMI. ArgoUML 0.24 (Tigris.org 2008) was chosen simply because it is free and has been used by the author before. ArgoUML 0.24 supports UML 1.4 and exports to XMI 1.2. As an example, Figure 6.2 demonstrates both the UML class diagram of the Export PIM ontology of \( P_{CCC} \)’s export data set \( \Lambda_{CCC,Address} \), and the corresponding XMI document exported by ArgoUML.

Producing the Export PIM Ontology in OWL

If a data provider wants to encode the Export PIM ontology in OWL, it can either create a new OWL encoded ontology document from scratch, or translate a generated XMI document into OWL. Both of the methods can be easily achieved using the Protégé tool, which has already been introduced in Chapter 5. Instructions on how to use Protégé to translate the encoding of an ontology from XMI to OWL are detailed in Appendix A.

Remarks

As far as a provider is concerned, this is all it needs to do to produce the Export PIM ontology. In the above example, although the provider’s shared data set is very simple as it only contains
6.1. DATA PROVIDER IMPLEMENTATION

one feature type, the aforementioned ontology production process still applies when more feature types are contained.

However, the process of generating the Export PIM ontology based on a provider’s database schema still involves human intervention, since the elements of the schema need to be manually mapped onto either a UML class diagram or OWL constructs. Although this process only needs to occur once unless the schema of the shared data set is changed, it would be better if it can be automated by a software application. This is listed as a potential future improvement in Chapter 8.

As will be demonstrated later in Section 6.1.3, once a data provider’s web service interface is constructed, any data consumers can query that interface for the Export PIM ontology.

6.1.2 Generating Update Notifications

Capturing and Storing Data Updates

Capturing and storing data updates are the first two steps towards generating update notifications. In this research, PostgreSQL (PostgreSQL Global Development Group 2008) is used as a DBMS tool for managing both data providers’ and consumers’ databases, and the PostGIS plugin (Refractions Research 2008) is used to query spatial data and perform spatial operations.

There are multiple ways to capture data updates, and those ways can be implemented at different levels - typically either at the DBMS level or at the application level. In this research, capturing data updates is implemented at the DBMS level by using database triggers since it is the easiest to implement and has the smallest scope of impact. The reason is that database triggers can be freely attached to, or detached from, a database table without requiring changes to a data provider’s existing software application or affecting the existing data in the provider’s database.

When a database trigger is attached to a table, any data insertion, deletion or modification made in that table can be detected by the trigger, and cause the execution of the corresponding operations specified in that trigger.

In this research, for each table that contains data that is going to be shared, a trigger is
written and attached to that table, now referred to as the target table. When a data record is updated in the target table, the trigger will copy the entire updated record to a different so-called mirror table, which not only duplicates all the columns in the target table, but also contains two additional columns storing information related to the update operation itself: the update type and the update time. Therefore, for each target table, two metadata objects need to be created: one is the trigger for capturing data updates, and the other is the mirror table for storing the captured information.

Since a data provider’s application is most likely to issue data updates directly against its local data set, triggers have to be attached to the tables that are in such a data set in order to take effect. In addition, since a data provider’s local data set conforms to the Local PSM ontology, both the trigger and the mirror table also need to conform to the Local PSM ontology. To distinguish between tables that are conforming to different types of ontologies, each table will be created under a schema, which has the same name as the ontology to which the table conforms; for example, local_psm.address means the address table conforms to the Local PSM ontology.

In the first case study scenario, if the SQL (Structured Query Language) statement shown in Listing 6.1 is used by PCCC to create the table for storing the Address features, then the SQL shown in Listing 6.2 can be used to create the corresponding mirror table, which contains two additional columns: update_type and update_time.

Listing 6.1: The SQL statement for creating the local_psm.address table.

```sql
CREATE TABLE "local_psm"."address" (
  "gid" integer NOT NULL,
  "road_name" varchar(28),
  "mi_id" varchar(6),
  "house_high" varchar(4),
  "sad_id" varchar(9),
  "sufi" varchar(15),
  "road_type" varchar(9),
  "addr_match" varchar(33),
  "house_low" varchar(4)
);
```

Listing 6.2: The SQL statement for creating the mirror local_psm.address_update table.

```sql
CREATE TABLE "local_psm"."address_update" (
  "gid" integer NOT NULL,
  "road_name" varchar(28),
  "mi_id" varchar(6),
  "house_high" varchar(4),
  "sad_id" varchar(9),
  "sufi" varchar(15),
  "road_type" varchar(9),
  "addr_match" varchar(33),
  "house_low" varchar(4),
  "update_type" char(6) NOT NULL,
  "update_time" timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```
Once the mirror table is created, it can be referenced from within the trigger, which is shown in Listing 6.3 and the trigger can be finally attached to the target table address using the SQL statement in Listing 6.4.

Listing 6.3: The statement for creating the audit_address_update trigger.

```sql
create or replace function audit_address_update() returns trigger as '
begin
  if tg_op = 'INSERT' then
    insert into local_psm.address_update (gid, house_low, house_high, road_name,
    road_type, addr_match, mi_id, sad_id, sufi, the_geom, update_type) values (new.
    gid, new.house_low, new.house_high, new.road_name, new.road_type, new.addr_match
    , new.mi_id, new.sad_id, new.sufi, new.the_geom, 'INSERT');
  return new;
end if;
if tg_op = 'DELETE' then
  insert into local_psm.address_update (gid, house_low, house_high, road_name,
  road_type, addr_match, mi_id, sad_id, sufi, the_geom, update_type) values (old.
  gid, old.house_low, old.house_high, old.road_name, old.road_type, old.addr_match
  , old.mi_id, old.sad_id, old.sufi, old.the_geom, 'DELETE');
  return old;
end if;
if tg_op = 'UPDATE' then
  insert into local_psm.address_update (gid, house_low, house_high, road_name,
  road_type, addr_match, mi_id, sad_id, sufi, the_geom, update_type) values (new.
  gid, new.house_low, new.house_high, new.road_name, new.road_type, new.addr_match
  , new.mi_id, new.sad_id, new.sufi, new.the_geom, 'MODIFY');
  return new;
end if;
end
' language plpgsql;
```

Listing 6.4: Attach the trigger audit_address_update to the local_psm.address table.

```sql
create trigger audit_address_update after insert or delete or update on local_psm.
address for each row execute procedure audit_address_update();
```

Transforming Data between Ontologies

Since the captured data updates are stored in a mirror table that only conforms to a provider’s Local PSM ontology, this information cannot be directly used to generate update notifications, as it needs to be firstly transformed into a format that conforms to the provider’s Export PIM ontology. The complexity of this transformation is dependent on the degree of similarity of a provider’s Local PSM ontology and one of the PIM ontologies, such as the Local PIM ontology - the more similar they are, the easier the transformation. This transformation process is known as ontology mapping, which is an active research area (Kalfoglou & Schorlemmer 2003, Ehrig & Sure 2004, Johnson et al. 2007), but it is not a focal point of this research.

Although this research is not focused on the ontology mapping and data translation processes, Figure 6.3 demonstrates a simple ontology translation workflow to translate PCCC’s Address feature type from the Local PSM ontology to the Local PIM, then finally to the Export PIM ontology. During the translation, the number of fields that are exposed to data consumers has been reduced, and some fields have been renamed. In this research, this ontology translation process is implemented using database views, as demonstrated in Listing 6.5.
Creating Update Notifications

The final step is to generate update notifications. As described in the discussion in Section 4.2 about the content of an update notification, once the updated records are captured and translated into a format that conforms to the Export PIM ontology, they can be extracted from the database, and used to create the XML encoded update notification documents.

6.1.3 Constructing the Web Service Interface

If a data consumer wants to access a provider’s Export PIM ontology, update notifications, or the actual data, it has to interact with the provider’s web service interface.

In the current implementation, a provider’s web service interface consists of five methods to expose core services that are related to the selective data replication system. Among these five methods, two, subscribe and unsubscribe, are for managing consumer subscriptions, and the other three, getOntology, getNotifications and getData are for retrieving the provider’s Export PIM ontology, the update notifications and the actual data, respectively.

Before a data consumer can access a provider’s information, it has to call subscribe to register itself with the provider. During the registration, information about the data consumer, such
as its user name and password, may need to be provided. In this research, as the UML class diagram in Figure 6.4 illustrates, a Consumer object is expected by a data provider during the subscription. Optionally, a Consumer can reference a ConsumerSubscription object, which contains additional subscription information including a list of feature types to which the consumer wishes to subscribe and the fields of those feature types that are used in the consumer's update policy. The latter information is to address the performance overhead problem and to support the “Smart Fallback” feature discussed in Section 4.2.2. Once the subscription is performed successfully, all the information provided by the consumer is stored in the provider’s database, and will be used accordingly to generate update notifications.

The unsubscribe method, as its name suggests, unsubscribes a consumer from a provider. The current implementation of this method is to simply remove the consumer’s information from the provider’s database. After the un-subscription, the consumer needs to call subscribe before it can consume the provider’s services again. Figure 6.5 shows the UML sequence diagram illustrating the implementations of the methods, subscribe and unsubscribe.

When a data consumer is subscribed to a provider, the consumer can invoke getOntology, getNotifications or getData to retrieve the corresponding information. In real world applications, security measures are usually put in place to validate a consumer’s identity before any further operation is carried out, but considering the research nature of this project, only the consumer name is needed to establish a genuine connection between the consumer and the provider.

The getOntology method returns a provider’s Export PIM ontology generated using the process described in Section 6.1.1. By default, the generated Export PIM ontology is encoded in XMI.
The `getNotifications` method allows a data consumer to “pull” update notifications from a provider. As illustrated in Figure 6.6, upon a consumer’s request, the provider firstly verifies the consumer’s identity, then tries to generate an update notification by invoking the process described in Section 6.1.2 with the consumer supplied information, which may include both the feature types that are subscribed by the consumer and the fields that are referenced from within the consumer’s update policy. Such information is provided either during the subscription or along with the `getNotifications` request.

Figure 6.6: The UML sequence diagram of the implementation of `getNotifications`.

If a data consumer has not yet subscribed to at least one feature type, no update notifications will be generated. If a consumer subscribes to one or more feature types, but has not provided the fields that are used in the consumer’s update policy, then for each updated feature instance, only the identification field and the corresponding value will be included in the generated update notification.

As mentioned in Chapter 3, although this research does not focus on the actual data retrieval, it still provides a basic implementation of `getData` in order to make the selective data replication system complete. The areas that have been simplified for the actual data retrieval and integration processes are listed in Chapter 8 as future work items.

Once the update policy evaluation process has been completed and one or more data updates have been identified as significant updates, a data consumer can invoke the `getData` method on the provider’s web service interface to retrieve the data it wants to replicate. As part of the invocation of `getData`, the consumer needs to pass in its own identity as well as a list of identifiers, which can uniquely identify the feature instances of the specified feature type. Once the request has been accepted and processed, the data consumer will be returned another XML document containing the full copy of all the requested feature instances. Since this research uses geographical data, the returned XML document is encoded in GML (Geography Markup
6.2 Data Consumer Implementation

The data consumer implementation can be broken down into three sequential tasks: manage received update notifications, evaluate update policy and perform data replication.

6.2.1 Managing Update Notifications

Update notification management consists of two parts: retrieve update notifications from data providers and store the retrieved information.

Retrieving update notifications from a data provider only requires a consumer to invoke the `getNotifications` method on the provider’s web service interface, which is demonstrated in Listing 6.6.

Listing 6.6: Sample code for \texttt{CPostOffice} to retrieve update notifications from \textit{PCCC}.

```java
... CCCDataProviderService service = new CCCDataProviderService(); CCCDataProvider port = service.get CCCDataProviderPort(); Object obj = port.getNotifications(getConsumerInfo()); ...
```
Once `getNotifications` returns, the data update descriptions embedded in the returned update notifications are extracted and then inserted into the consumer’s database in the `data_updates` table. Section 4.4 briefly described the structure of the `data_updates` table and how this table is used to store an update description. Listing 6.7 provides the actual SQL statement used to create the `data_updates` table.

Listing 6.7: The SQL statement for creating the `data_updates` table.

```sql
create table "data_updates"
(
  "provider_id" text not null,
  "object_type" text not null,
  "id_field" text not null,
  "id" integer not null,
  "property_name" text,
  "property_value" text,
  "update_type" char(6) not null,
  "update_time" timestamp not null,
  "is_significant" boolean not null default false
);
```

To populate the `data_updates` table, an XML SAX parser is used to parse the XML encoded update notification documents and to extract the relevant information. The extracted information is transformed into a series of SQL statements, which can then be executed directly by the DBMS to store the information.

A SAX parser is used instead of a DOM one for performance reasons. Generally, the size of a geographical data set is large, and the volume of data changes could be great. A SAX parser processes a document from beginning to end, and does not require the entire document to be preloaded into the computer memory. On the contrary, a DOM parser requires the entire XML document to be loaded into the computer memory first, which may cause a negative performance impact if the size of the document is large.

Another implementation insight that is worth mentioning is that update notification retrieval is implemented in its own thread, so that it automatically and periodically connects to various data providers and “pulls” update notifications from them. Before the thread starts, a configuration file is provided with settings, such as what data providers to retrieve notifications from, and how frequently to retrieve such notifications. Once the thread has been started, no human intervention is required and the retrieval process is entirely controlled by the configuration settings. If the settings within the configuration file are changed, the new settings will be picked up and applied when the next notification retrieval occurs.

For the update notifications that are received using the “push” method, although the notification retrieval process becomes irrelevant, the above methodology still applies to the storage and other management tasks of update notification.

The reason why notification management is implemented in this way is to achieve modularity and automation. In this way, a group of functionalities, such as retrieving and storing update notifications, can be encapsulated inside one module, and each module is independent and can run without human intervention. This design significantly simplifies the process of migrating the selective data replication system onto an agent platform, where each agent is an independent unit and can have its own intelligence. Although such migration is beyond the scope of this research, the implementation is designed in a way to benefit this future development.
6.2.2 Evaluating Update Policy

Based on the policy evaluation mechanism described in Section 5.2, the evaluation workflow, as re-shown in Figure 6.8, is to firstly convert UML or OWL encoded ontologies, together with update policy rules that are specified in SWRL, into JESS templates and JESS rules. Then by running the JESS inference engine, the JESS rules can be evaluated with asserted facts, i.e. the data update descriptions stored in the data_updates table.

To implement this workflow, especially to bridge between OWL/SWRL and JESS, Prot´eg´e with two plugins SWRLTab (Stanford Center for Biomedical Informatics Research 2008e) and SWRLJessTab (Stanford Center for Biomedical Informatics Research 2008c) can be used. SWRLTab is a default plugin of Prot´eg´e, and is a development environment for working with SWRL rules. SWRLTab provides mechanisms to allow interoperation with a variety of rule engines by using additional rule engine plugins; for example, SWRLJessTab is a plugin to SWRLTab, and it enables execution of SWRL rules in the JESS rule engine.

**Importing Ontologies and Update Policies into Prot´eg´e**

Since the update policy evaluation process makes extensive use of the Prot´eg´e tool, ontologies and update policy rules need to be firstly imported into Prot´eg´e before other processes can commence. In addition, since an update policy can be specified based on both a consumer’s Import PIM ontology and the update notification ontology, both ontologies are imported.

Importing an ontology into Prot´eg´e is easy. If the ontology is already encoded in OWL, it can be opened directly in Prot´eg´e. If, however, the XMI encoding is used, the XMI encoded ontology needs to be firstly translated into OWL. Appendix A details this translation.

For update policy specification, Chapter 5 has already discussed two ways, depending on
which ontology representation language is used - either OWL or UML. If an ontology is encoded in OWL, update policy rules will be specified in SWRL and, as noted in Section 5.1.3, they can be created and manipulated in Protégé directly. On the other hand, if an ontology is represented in a UML class diagram, update policy rules will still be specified in SWRL, but each will be contained in a stereotyped note. To import the UML encoded ontology together with the SWRL rules, the UML class diagram needs to be firstly exported to an XMI document, which is then transformed into OWL.

The examples related to the update policy specification using the first case study scenario were already demonstrated in Section 5.1.3. However, the process of transforming an XMI-encoded document into OWL has not yet been covered. This transformation process needed to be custom implemented in this research, since although Protégé or other existing XSL transformation style-sheets know how to transform common UML modelling constructs, such as UML:Class and UML:Attribute, they have no idea how to extract the update policy rules from the <<UpdatePolicyRule>> stereotyped note.

**Parsing OWL Ontologies and SWRL Rules into JESS**

Once OWL-encoded ontologies and SWRL encoded rules are successfully imported into Protégé, parsing them into the corresponding JESS language constructs is easy with the help of the two aforementioned Protégé plugins, SWRLTab and SWRLJessTab. The parsing, as demonstrated in Appendix A, can be performed either by interacting with Protégé’s user interface, or by invoking the corresponding APIs (Application Programming Interfaces) programmatically.

Furthermore, apart from the ontologies and update policy rules, the JESS rule engine also needs to be provided with data, facts in JESS terminology, in order to evaluate the policy. In this research, as mentioned in Section 5.2, the data that is going to be used to evaluate an update policy comes from the data_updates table, which is populated by the update notification retrieval process mentioned in Section 6.2.1.

**Creating custom SWRL built-ins for geospatial operations**

SWRL provides a very powerful extension mechanism that allows user-defined methods, or built-ins, to be used in rules (Stanford Center for Biomedical Informatics Research 2008b). Each built-in is like a predicate that accepts one or more data bound or unbound arguments and produces a Boolean result, so that multiple built-ins can be linked together to achieve sophisticated reasoning logic. There are already a number of built-ins provided by the SWRLTab plugin, including basic mathematical operators, and routines for string and date manipulations. Built-in methods can also be invoked from a rule engine, such as the JESS rule engine.

Since this research is in the GIS domain, it is very likely that rules defined by consumers contain geospatial operations, such as testing for intersection or containment of geometries, or calculating buffer zones. Therefore, in addition to the basic built-ins provided by SWRLTab, a set of built-ins for commonly used geospatial operations is implemented. Listing 6.8 demonstrates one of them - intersects. The full source code including all the custom built-ins is provided in Appendix C.

In Listing 6.8, JTS Topology Suite (Vivid Solutions 2008) is used for performing 2D geospatial operations in Java. JTS conforms to the Simple Feature Specification for SQL published by the Open GIS Consortium (OGC 2005) so that it works seamlessly with PostGIS enabled PostgreSQL databases used in this research.
With these spatial operation built-ins, data consumers can specify corresponding spatial operations in their update policy rules. For example, the following rule replicates all the updated addresses within the range of one kilometer surrounding POINT(23 36):

```
Address(?a) ∧ isSignificant(?a, ?s) ∧ swrlb:equal(?s, false) ∧ geometry(?a, ?g) ∧ swrlgeo:bufferRound(?b, "POINT(23 36)", 1000) ∧ swrlgeo:contains(?b, ?g) → isSignificant(?a, true)
```

When necessary, a data consumer can also develop its own custom built-ins if the data replication logic is too difficult or impossible to be expressed purely in SWRL.

**Implementation Remarks**

Similar to update notification management, update policy evaluation is also implemented as a self-contained functional module that runs automatically and uninterruptedly. When this module is started, it is fed with a data consumer’s Import PIM ontology with predefined policy rules. During execution, the module monitors the `data_updates` table and imports all the data update descriptions from that table into the JESS rule engine’s knowledge base for inference. If one or more data update descriptions satisfy the update policy, the `isSignificant` property of the descriptions will be changed from `false` to `true`, and the changed value will be saved back...
to the `data_updates` table. In this way, any data update descriptions with `isSignificant = true` can be picked up by the data replication process later. The rest of the data update descriptions that do not pass the evaluation still stay in the JESS rule engine’s knowledge base, waiting to be re-evaluated when further changes occur.

Storing the retrieved update notifications in a database table, instead of directly passing them into the JESS rule engine, further decouples the selective data replication system and makes it more flexible and scalable. In an agent environment, these two processes could be performed by two different agents, especially when collaborative behaviour among agents is implemented. For example, if multiple data consumers want to subscribe to the same data set from the same provider, they can instruct their software agents to work collaboratively, to elect one agent to retrieve update notifications and then distribute these to the others (Fox 2007); each consumer still requires its own software agent to perform update policy evaluation. Another common scenario is that the update notification retrieval is usually scheduled at network idle time whereas policy evaluation and data replication can happen at a different time that is preferable to data consumers.

### 6.2.3 Replicating Data

Data replication is the last process in the selective data replication workflow, and it involves replicating the updated data from one or more data providers to a data consumer. As briefly discussed in Section 6.1.3, a data consumer can initiate the data replication process by invoking the `getData` method on the provider’s web service interface, and supplying the relevant information that indicates what data needs to be replicated. In the current implementation, when a data consumer invokes the `getData` method, it also provides the name of the feature type and a list of identifiers, each of which can uniquely identify a feature instance of the specified feature type. The identified feature instances will be formatted into a GML encoded document, which is then sent back to the data consumer. Upon receipt of the GML encoded data document, the consumer can extract the embedded information and integrate it into its own local copy.

Since the actual data replication process is not the focus of this research, its implementation is also simplified, especially for the data translation processes on both the data provider and the data consumer side. Section 6.1.2 mentioned the necessity of the data transformation process for a data provider and the approach taken by this research to realize transformations. However, the approach used in this research - using database views - is not a scalable solution and cannot handle more complicated data transformation scenarios where sophisticated data calculations and manipulations are required. Therefore, as will be mentioned in Chapter 8, a better approach needs to be developed in the future.

The data transformation process on the data consumer side is a similar but reverse process: it transforms information that conforms to the Local PIM ontology, into one that conforms to a consumer’s Local PSM ontology. This transformation process is not implemented in this research, and this research assumes that each data consumer’s local copy is already in conformity with the Import PIM ontology; therefore, no data translation is required when the replicated data is received. As will also be mentioned in Chapter 8, although this assumption is reasonable, since a data consumer should be able to start consuming the replicated information without having to translate it into another format, it would still be better to provide this extra translation process, just in case a data consumer needs to merge the replication information with its own legacy data.
Chapter 7

Evaluation

The previous chapter provides the implementation insights of the selective data replication system, mainly including the update notification generation and the update policy evaluation - the two core components emphasized in Chapter 3.

In this chapter, the implemented selective data replication system is evaluated to determine whether the research objective set in Section 2.5 is achieved. Since the objective is “to offer better flexibility and efficiency...”, the content of this chapter is divided into two sections: the first one is to discuss the flexibility of the selective data replication system, and the second is to analyze the performance.

7.1 Flexibility

In traditional data replication, as already mentioned in Section 2.3.2, when a replica is first created, its fragmentation configuration is stored with the master database, and then subsequently used for data synchronization - any data changes that match the fragmentation configuration are unconditionally replicated, regardless of whether some of the updates are necessary or of interest to that replica. In addition, whenever the replica needs to change its fragmentation configuration, even just changing the value of a field that was used for fragmenting a database, e.g. changing the value of suburb from “Riccarton” to “Downtown”, it needs to coordinate with the master database and download a new snapshot of the data according to the new configuration.

In the selective data replication system, however, the flexibility comes from the ability to specify ontology-based update policies.

The update policy specification is performed by each data consumer according to its unique business requirements; therefore, how a policy is written is entirely up to the data consumer, and all the business logic is encapsulated in the consumer side only - any data provider who provides the data and data updates to this consumer does not need to know anything about it.

Because of such encapsulation, a data consumer can easily change its update policy at any time and in any way, without having to notify data providers. Although, as mentioned in Chapter 4, for performance reasons the fields of feature types that are used in data consumers’ update policies are provided to data providers before retrieving update notifications, only the names of those fields are provided rather than their values and business logic.
7.2 Performance

Performance is another key factor for evaluating the selective data replication system compared to other traditional data replication implementations, especially row-level data replication.

As discussed in Chapter 2, row-level data replication replicates all the data changes from the master database to replicas if the data being updated is also stored in the replica databases; therefore, in many environments, especially the one being considered in this research, row-level replication may not be an efficient solution.

The selective data replication developed in this research is aimed at improving efficiency by allowing each data consumer to specify an update policy to evaluate and filter data changes published by data providers, thereby replicating less information.

However, instead of evaluating update policies in terms of the actual updated data, which requires all the changes to be firstly transmitted from a data provider to a data consumer, update policies are evaluated in terms of update notifications, which only contain a subset of information describing the updated data. The problem is that once the evaluation is finished, the actual data that has been updated, which is now of interest to the data consumer, still has to be transmitted; therefore, depending on the size of an update notification and the time taken to evaluate an update policy, the selective data replication system may not always offer better efficiency.

This section firstly provides a high-level performance analysis, by breaking down the selective data replication system into tasks, and then analyzing the performance issues related to each task. Following that, a statistical empirical analysis is presented.

7.2.1 High-Level Task-Breakdown Analysis

Figure 7.1 presents a high-level task-breakdown analysis of both row-level and selective data replication. In Figure 7.1 each replication methodology is broken down into a number of tasks, and the width of each task represents the amount of time required to execute this task. Since this is a high-level estimative analysis to discover how the time spent on each task contributes to the performance of a replication system, the exact execution time of each task does not matter to this analysis; instead, the comparison matters. It is also worth noting that the order shown for these tasks is unrelated to the order of their executions - it is simply arranged in this way for easier comparison.

In the top part of Figure 7.1 where the number of the desired data changes is relatively small, selective data replication performs better since the total time spent on the update notification retrieval and policy evaluation is still less than the time spent on replicating the undesired data changes.

In the middle part of Figure 7.1 if both the total number of the data changes and the size of the update notifications stay the same, this means the time spent on the policy evaluation is also the same. But when the number of the desired data changes starts to grow, the performance advantage offered by the selective data replication will decline. Eventually, the selective data replication will perform worse than the traditional row-level data replication, due to the overhead introduced by the update notification retrieval and policy evaluation processes.

Finally, at the bottom of Figure 7.1 even if the number of the desired data changes stays relatively small compared to the total number of the data changes, selective data replication may still perform worse due to the size of the update notifications. According to the discussion in Section 4.2, the more fields of a feature type that are used in the update policy specification,
7.2. PERFORMANCE

the richer the notification’s content, and the larger the notification’s size. Once the size of an update notification becomes too large, it also impacts on the performance.

The following subsections present simulations of the first two case study scenarios described in Section 2.4 and analyze the statistics collected from these simulations to provide more concrete evidence of the above high-level performance analysis. The reason why the third scenario is excluded is that it is basically a combination of the first two.

7.2.2 Test Environment Preparation

To effectively evaluate and compare the performance between the selective and a row-level data replication system, both of the replication systems need to be set up first.

In this research, Slony-I (Slony Development Group 2008) is used as the traditional row-level data replication system, which supports the “master to multiple slaves” asynchronous replication environment. Slony-I is chosen because it is an open source replication system and works naturally with PostgreSQL databases.

As mentioned before, the first two case study scenarios from Section 2.5 are used as the context environments for simulating the performance evaluation tests. Each scenario may consist of multiple tests, each of which is configured with different settings reflecting different real world phenomena. Finally, to ensure a fair comparison, in each test both the databases and the SQL scripts that are used for generating data updates are identical in each data replication system.
7.2.3 Statistics Collection Method

In the selective data replication system developed by this research, various time checkpoints are programmed into both the data provider and the data consumer side applications to record the start and end time for executing each task, such as generating update notifications and evaluating update policy. During the execution, when each time checkpoint is hit, the actual time at that point will be logged into a log file. At the end of the execution, by processing the log file, the duration of each task can be calculated, and the sum of all the durations makes up the total time spent on the selective data replication implementation. Listing 7.1 demonstrates a block of the log file produced by the selective data replication. The numbers shown in Listing 7.1 indicate the number of milliseconds since January 1, 1970, 00:00:00 GMT.

Listing 7.1: The sample log file produced during the execution of the selective data replication.

```
RetrieveUpdateNotifications:START:1222251937257
RetrieveUpdateNotifications:END:1222251937694
... SaveUpdateNotifications:START:1222252275996
SaveUpdateNotifications:END:1222252277089
... EvaluateUpdatePolicy:START:1222252373390
EvaluateUpdatePolicy:END:1222252373566
... ReplicateData:START:1222252434684
ReplicateData:END:1222252440141
... 
```

A similar approach is used to collect timing information about the Slony-I system. Although the Slony-I system does output some time stamps during its execution, this research injects additional time checkpoints into the Slony-I source code.

In the following subsections, each test is repetitively carried out three times, and only the average is used for the performance analysis.

7.2.4 Simulating “Post Office”

The “Post Office” scenario is an extreme case for selective data replication, as the data consumer \( C_{PostOffice} \) is interested in every single data change published by the provider \( P_{CCC} \).

To simulate this scenario, an SQL script is generated to update 5000 address records, including insertions, deletions and modifications, in \( P_{CCC} ’s A_{CCC,Address} \) within a single transaction.

In this scenario, since \( C_{PostOffice} \) is interested in all the data updates, it only needs to verify the existence of an address record in the received update notification in order to trigger the replication of that record. \( C_{PostOffice} \) can adopt one of the following two ways to achieve this.

Use Update Notification & Policy

In this way, \( C_{PostOffice} \) can specify the following update policy to identify any addresses that need to be replicated.

\[
UpdatedObject(?x) \land \text{isSignificant}(?x, ?s) \land \text{swrlb:equal}(?s, \text{false}) \\
\Rightarrow \text{isSignificant}(?x, \text{true})
\]
Since the above update policy is only specified based on the update notification ontology, as explained in Section 4.2, the update notification for $C_{PostOffice}$ does not have to include any additional fields and field values apart from idField and id, which are already part of the update notification ontology, and included in every update notification by default.

Using the task-breakdown analysis discussed in Section 7.2.1, Figure 7.2 illustrates that using the selective data replication in this way is going to introduce a significant overhead. This analysis is confirmed by the performance statistics collected from the execution of both replication systems, which is shown in Table 7.1: the selective data replication system performs more than one and half times slower than Slony-I.

![Figure 7.2: “Post Office” Scenario: Task-breakdown analysis.](image)

<table>
<thead>
<tr>
<th>Selective Data Replication</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slony-I</td>
<td>17483</td>
</tr>
<tr>
<td>Selective Data Replication</td>
<td>28043</td>
</tr>
</tbody>
</table>

Use “Smart Fallback”

The “Smart Fallback” feature of the selective data replication system was introduced in Chapter 4 while discussing the content of an update notification. The purpose of this feature is to prevent the selective data replication from causing performance overhead introduced by the update notification retrieval and the policy evaluation processes, like the situation demonstrated in the previous subsection.

As discussed in Section 4.2, to enable “Smart Fallback”, $C_{PostOffice}$ can supply all the fields of the Address feature type during the subscription or before the notification retrieval. Figure 7.3 presents a refined task-breakdown analysis of the “Post Office” scenario when “Smart Fallback” is enabled.

After rerunning the test with “Smart Fallback” enabled, the average time taken by the selective data replication system to replicate 5000 address updates is shortened from 28043 to 19506 milliseconds. The selective data replication is still marginally slower than Slony-I; the reasons for this could be related to the differences between the adopted programming languages, deployment environment, runtime environment and so on.
7.2.5 Simulating “Real Estate”

In this scenario, instead of accepting all the data updates published by the data provider $P_{CCC}$, the data consumer $C_{RealEstate}$ only replicates the updated parcels that match the following business requirements:

1. the updated parcel must be of residential type;
2. the updated parcel must be sub-dividable; and
3. the area of the updated parcel must be at least 15 hectares.

These business requirements can be translated into an update policy rule as below:

\[
\text{UpdatedObject}(?x) \land \text{id}(?x, ?i) \land \text{isSignificant}(?x, ?s) \land \text{swrlb:e}\text{qual}(?s, \text{false}) \land \text{objectType}(?x, ?ot) \land \text{swrlb:e}\text{qual}(?ot, "Parcel") \land \text{Parcel}(?p) \land \text{parcelId}(?p, ?pi) \land \text{swrlb:e}\text{qual}(?pi, ?i) \land \text{parcelType}(?p, ?pt) \land \text{swrlb:e}\text{qual}(?pt, "Residential") \land \text{isSubDivisible}(?p, ?d) \land \text{swrlb:e}\text{qual}(?d, \text{true}) \land \text{geometry}(?p, ?g) \land \text{swrlgeo:a}\text{rea}(?g, ?a) \land \text{swrlb:g}\text{reaterThanOrEqual}(?a, 15) \rightarrow \text{isSignificant}(?x, \text{true})
\]

To verify the various performance influencing factors discussed in Section 7.2.1, a number of tests are carried out using this scenario. These tests can be grouped into two categories: the first one keeps the update notification size consistent and adjusts the number of data updates that are of interest to $C_{RealEstate}$. Conversely, the second category of tests keeps the number of the desired data updates the same but adjusts the size of the update notifications. All tests from both categories have the same number of total data updates - 5000.

### Adjusting the number of the desired data updates

Tests in this category are used to discover and measure the relationship between the number of data updates of interest to a data consumer and the amount of performance gain or loss by using the selective data replication.

There are seven tests in this category, each containing the following number of parcel changes that match $C_{RealEstate}$’s update policy: 500 (10%), 1250 (25%), 2000 (40%), 2500 (50%), 3000 (60%), 3750 (75%) and 4500 (90%). As mentioned before, each test is repeated three times, and the average is recorded in Table 7.2 and graphed in Figure 7.4.
Table 7.2: “Real Estate” Scenario: Performance statistics collected from Slony-I and the selective data replication when only adjusting the number of the desired data updates.

<table>
<thead>
<tr>
<th>Percentage of the desired data updates</th>
<th>Selective Replication (ms)</th>
<th>Slony-I (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>14340</td>
<td>20018</td>
</tr>
<tr>
<td>25%</td>
<td>16256</td>
<td></td>
</tr>
<tr>
<td>40%</td>
<td>19023</td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td>19766</td>
<td></td>
</tr>
<tr>
<td>60%</td>
<td>21069</td>
<td></td>
</tr>
<tr>
<td>75%</td>
<td>23778</td>
<td></td>
</tr>
<tr>
<td>90%</td>
<td>26066</td>
<td></td>
</tr>
</tbody>
</table>

"Real Estate" Scenario: Performance Comparison (Fixed notification size & Variable number of desired updates)

Figure 7.4: “Real Estate” Scenario: Performance comparison between selective data replication and Slony-I, with fixed notification size and variable number of desired updates.

Figure 7.4 clearly shows that the performance advantage of the selective data replication only exists when the percentage of the consumer’s desired number of data updates is around 50% or lower, of the total number of updates: the lower the percentage, the greater the performance gains.

When the percentage of the desired number of data updates is above 50%, i.e. when the selective data replication performs worse than the traditional row-level replication, the “Smart Fallback” feature is unable to help improve the performance, since it is a feature that is enabled or disabled by a data provider based on the amount of information included in an update notification. In these tests, the amount of information included in the update notifications is always fixed - four fields for each updated Parcel instance: id, parcelType, isSubDividable and geometry. In addition, the data provider has no way to tell how many data updates the consumer desires it to replicate, as this is the result of the update policy evaluation process performed by the data consumer.
Adjusting the size of the update notification

The purpose of the tests in this category is to help the selective data replication system, especially the part on the data provider side, to decide when the “Smart Fallback” feature should be automatically enabled to avoid potential performance overhead, which, as mentioned before, could be the result of including too much information of a feature type in an update notification.

To find out when the performance overhead starts to emerge, each test in this category includes a different number of fields inside an update notification. In this scenario, the Parcel feature type contains six fields in addition to the identification field, so six tests are carried out.

To ensure the fairness of these tests, all the six fields in the database have been either trimmed down or padded to the same number of bytes and populated with the same text value; therefore, if an update notification contains four out of seven fields, it means the size of the update notification is 57.14% of the size of the actual updated data. In addition, all the tests assume only 10% (500) of the total data updates are of interest to \( C_{\text{RealEstate}} \), and the “Smart Fallback” feature is disabled during these tests. The test results are recorded in Table 7.3 and graphed in Figure 7.5.

Table 7.3: “Real Estate” Scenario: Performance statistics collected from Slony-I and the selective data replication when only adjusting the size of the update notification.

<table>
<thead>
<tr>
<th>Percentage of the size of an update notification</th>
<th>Selective Replication (ms)</th>
<th>Slony-I (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>28.57%</td>
<td>13150</td>
<td>21204</td>
</tr>
<tr>
<td>42.86%</td>
<td>14566</td>
<td></td>
</tr>
<tr>
<td>57.14%</td>
<td>17268</td>
<td></td>
</tr>
<tr>
<td>71.43%</td>
<td>19623</td>
<td></td>
</tr>
<tr>
<td>85.71%</td>
<td>22789</td>
<td></td>
</tr>
<tr>
<td>100.00%</td>
<td>25814</td>
<td></td>
</tr>
</tbody>
</table>

As shown in Figure 7.5, the performance drops below that of Slony-I when the size of the update notification is about 78% or more of the size of the total updated information. This means if the “Smart Fallback” feature is enabled, when six or all seven fields are requested to be included in an update notification by the data consumer, the provider should automatically include all the updated data into the update notification and mark the update notification as IsFull = “True”, so that the consumer does not have to carry out the update policy evaluation and data replication processes. Table 7.4 records the performance statistics collected from the selective data replication when “Smart Fallback” is enabled, and Figure 7.6 redraws the graph using the test results recorded in both Table 7.3 and 7.4.

Table 7.4: “Real Estate” Scenario: Performance statistics collected from the selective data replication with “Smart Fallback” when only adjusting the size of the update notification.

<table>
<thead>
<tr>
<th>Percentage of the size of an update notification</th>
<th>Selective Replication with “Smart Fallback” (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>85.71%</td>
<td>21863</td>
</tr>
<tr>
<td>100.00%</td>
<td>21824</td>
</tr>
</tbody>
</table>

7.2.6 Summary

The performance of the selective data replication is evaluated from two aspects: one is the number of consumer-desired data updates, and the other is the size of an update notification. These two aspects are the two changing factors in the selective data replication. Although Section 7.2.5 evaluated the performance impact of each of the two factors individually, by
changing one and fixing the other, in this section, instead, the relationship between those two factors and how the selective data replication system performs when both of the factors are changing are discussed. As the result, a mathematical formula is developed.

To establish a mathematical formula, a few variables are defined as follows: for a particular feature type provided by a data provider, \( n \) is the number of instances that have been updated, and each instance is of an average size \( m \). A data consumer is subscribed to this feature type and specifies its update policy based on a number of fields of this feature type. To generate an update notification for this consumer, the data provider needs to create a notification with its size equal to \( x\% \) of all the updated information. Finally, as the update policy evaluation outcome, the data consumer is only interested in replicating \( y\% \) of all the data updates.

Firstly, the total size of the updated information is:

\[
n \cdot m \tag{7.1}
\]

This is also equal to the total amount of information that needs to be replicated when a traditional row-level replication is used.

For the selective data replication, the size of the update notification that needs to be transmitted from the data provider to the consumer is:

\[
n \cdot (m \cdot x\%) \tag{7.2}
\]
"Real Estate" Scenario: Performance Comparison
(Variable notification sizes & Fixed number of desired updates)

Figure 7.6: “Real Estate” Scenario: Performance comparison between selective data replication (with and without “Smart Fallback”) and Slony-I, with variable notification sizes and fixed number of desired updates.

After the update policy evaluation, the data consumer needs to replicate \( y\% \) of the total data updates, so the amount of information that needs to be transmitted is:

\[
(n \cdot m) \cdot y\%
\]  (7.3)

Therefore, at the end of the data replication process, the total amount of information transmitted from the provider to the consumer is the sum of the two values from (7.2) and (7.3):

\[
n \cdot (m \cdot x\%) + (n \cdot m) \cdot y\%
\]

which can be re-arranged to:

\[
n \cdot m \cdot (x\% + y\%)
\]  (7.4)

By comparing the expressions from (7.1) and (7.4), it is easy to conclude that, if \((x\% + y\%) < 100\%\),

then selective data replication will be faster than traditional row-level replication. Although the time spent on some processes, such as the update policy evaluation, are not taken into account in the above equations, those processes are performed either on a data provider or a data consumer’s local machine, which usually requires significantly less time compared to the time spent on retrieving data over the Internet.
Chapter 8

Future Work

Because the focal point of this research is on the ontology-based update policy specification and evaluation, the implementation of the selective data replication system has been simplified in certain areas; therefore, in this chapter are listed some future improvements that can be made to complete and enhance those areas.

In addition, as mentioned a few times in the thesis, beyond the objectives of this research, the ultimate goal is to migrate the selective data replication system onto an agent platform, in order to introduce more dynamic and intelligent behaviour to further improve the data replication scheme’s flexibility and performance. This chapter also includes brief discussions about the future work items that are required to achieve such a migration.

8.1 Enhancing the Current Implementation

8.1.1 Better Support Data & Ontology Translation

As mentioned in Section 6.1.2, database views are used in this research to perform the data and ontology translation process, in order for data providers to prepare their data and update notifications. However, using database views is insufficient in many situations where complicated data calculations and manipulations are required. In future, a more sophisticated process needs to be developed to streamline such translation process. This is a must-have feature and needs to be implemented in the near future.

8.1.2 Support Data Integration

As mentioned in Section 6.2.3 once the updated data is received from a data provider at the end of the data replication process, the data consumer needs to integrate the data into its own local copy. The current implementation assumes that the consumer’s local copy is already in conformity with its Local PIM ontology, so that the received data can be directly inserted into the local copy without having to be translated from the Local PIM ontology to the Local PSM ontology. Although this assumption is probably preferable since it avoids an additional translation process, data consumers, who may have already set up a traditional data replication and now want to migrate to the selective data replication system, may still require this extra data translation process, since their local copies may not conform to the Local PIM ontology.

Therefore, in future, similar to the data translation process for data providers, a data translation process needs to be developed for data consumers as well. In addition, such a process
also needs to be able to translate and integrate the data received from multiple data providers.

This future improvement is just “nice-to-have”, since without it, it still does not stop organizations who have not deployed a data replication system before from adopting selective data replication.

### 8.1.3 Support Multiple Identification Fields

As mentioned in Section 4.2.1, the updated objects described in an update notifications can be identified using one field only. In some situations, it would be necessary to allow multiple identification fields to be specified for each updated object.

Supporting multiple key fields requires modifying and extending the current XML schema of the update notification document, and it also involves code changes to populate and extract the information on those extended key fields. However, the changes required are minimal and easy to make.

Supporting multiple key fields is a must-have feature, and due to its simplicity, it can be implemented in the near future.

### 8.1.4 Support Multiple Policy Specification Languages

As mentioned in Section 5.1, this research uses SWRL as the only update policy rule specification language, which fits into the system well if OWL is used for ontology representation. However, if UML is used to represent ontologies, using SWRL is a little awkward.

Ideally, OCL should be adopted as the policy rule specification language when UML is used, even though it has several disadvantages, as mentioned in Section 5.1. In future, if the OCL processing tools become better and more sophisticated, supporting OCL would be a desirable feature. In addition, rule languages other than SWRL and OCL could also be supported if necessary.

### 8.1.5 Automate Export PIM Ontology Generation

Chapter 6 mentioned that the provider’s Export PIM ontology is created manually, which is prone to human errors. Furthermore, if data schema updates are handled within this selective data replication system, manually ensuring the consistency between the data schema and the Export PIM ontology would be a maintenance overhead.

As one of the nice-to-have features, the generation of the Export PIM ontology could be done automatically by examining the underlying database schema and additional business logic.

### 8.1.6 Support Ontology Update Replication

As mentioned in Chapter 2, this research is only concerned with capturing, propagating and analyzing data updates. Handling ontology updates, however, is a much more complicated topic, and requires a very careful approach in order to allow any ontological changes to be successfully propagated from a data provider’s database to a consumer’s local copy, and integrated without causing data corruption or loss.

Due to the complexity of this task, handling ontology updates is a long term nice-to-have feature. At the moment, if a data provider’s ontology is changed, it is much easier and safer for all of its subscribed data consumers to perform a snapshot replication.
8.1.7 Introduce Semantic Web Services

According to Section 3.1, this research uses the web service framework to provide consistent representations of data and data operations in a heterogenous environment. Nevertheless, the current web service technologies can only specify the syntactical aspects rather than the semantic aspects of the data; for example, WSDL cannot specify semantic meanings of the data nor semantic constraints on the data (Semantic Web Services 2008).

Semantic web services, however, can describe the various aspects of a web service using explicit, machine-understandable semantics, enabling the automatic location, combination and use of web services (Lara et al. 2005, Roman et al. 2006). Therefore, with the introduction of semantic web services, data consumers should be able to more easily and automatically find and subscribe to their favourite data providers.

8.2 Migrating onto an Agent Platform

8.2.1 Introduce Software Agents

Introducing software agents and adapting the selective data replication system into an agent environment would allow both the scalability and flexibility of the selective data replication system to be significantly improved. The reason is that within an agent environment, one data consumer can deploy multiple software agents, each of which can be assigned one or more tasks, such as discovering data providers, maintaining subscriptions, retrieving data and data updates and evaluating the update policies. When those agents are deployed, they can be configured differently and run independently without further human intervention. If more tasks are required, more agents can be deployed, making the system very extensible.

Introducing software agents is a highly desirable extension, and the current implementation, as mentioned in Chapter 6, has already been designed in a way that allows easier transition to the agent environment.

8.2.2 Support Collaborative Agent Behaviour

While software agents deployed by each data consumer may act independently of each other, the ultimate expectation is that they will collaborate where common goals can be found within their update policies. Such collaboration among agents is intended to facilitate the process by selectively combining many similar update policies in order to generate a derived update policy containing common goals within those update policies that have been found automatically by software agents.

Supporting the collaborative agent behaviour is a complicated task. Although Fox (2007) has made some progress, there are still many areas requiring further improvements that have been identified in his paper. This feature would make a nice-to-have extension.
Chapter 9

Conclusion

This research developed an enhanced data replication mechanism, namely selective data replication. This mechanism addresses problems associated with data replication within a data-decentralized and geographically distributed environment, where multiple organizations may exchange their data and data updates. In this research environment, an organization can be either a data provider, which provides others with its data set, or a data consumer, which consumes the shared data sets, or both.

The three problems that this research tackles by using the selective data replication are: unawareness, performance and heterogeneity. The unawareness problem is caused by the distributed environment and the involvement of multiple organizations, as organizations do not form any kind of awareness until they have gone through a typical registration-discovery-subscription process, which is implemented using the web service framework and the peer-to-peer communication model in this research. Although such a process is not contributed to by this research, it has to be built into the selective data replication system to provide the underlying communication infrastructure, in order for different organizations to interact with each other.

Performance and heterogeneity are two problems inherited from unawareness. Without knowing the needs of the other organizations, each organization simply uses or develops its own technology and standards for information sharing, thereby resulting in potential performance overheads related to the transmission of undesired information, and the additional processes and efforts required to migrate technologies and to interpret data. The solutions that are contributed to by this research are based on the concepts of update notification and update policy, and the four-level ontology framework.

During the selective data replication, to minimize the amount of information transmitted from a data provider to a consumer, especially when some of the information is not desired by the consumer, an update notification, which only contains a subset of information about updates, becomes useful. The notification not only notifies the consumer with updates, but also allows the consumer to evaluate whether the information embedded is significant to that consumer. How a data consumer evaluates the significance of an update notification is achieved by specifying and evaluating an update policy, which contains a collection of rules specified by the data consumer to reflect its unique data synchronization requirements. Once the significance of an update notification is determined, a data replication process can then be initiated to only replicate the significant data.

Update notification and update policy are the core contributions of this research, and should be treated as a pair of concepts, since missing one will compromise the benefits offered by the selective data replication system.
Another contribution of this research is the four-level ontology framework, which is developed to remove the ambiguities and unnecessary complexities raised at various stages of the selective data replication. This framework is used by both data providers and consumers in this research: a data provider uses it to prepare its shared data set and to generate update notifications; a data consumer uses it to import shared data sets and to specify the update policy.

Selective data replication was compared with a traditional row-level replication system to evaluate its flexibility and performance. The selective data replication system is more flexible, as, by manipulating the update policy, a consumer can easily change its data synchronization logic regarding when to replicate as well as what to replicate, without having to notify data providers. With respect to performance, although the selective data replication does not always guarantee a more efficient data replication, under some circumstances it can offer a significant amount of performance improvement. Based on a mathematical formula derived here, the performance advantage starts to emerge when the percentage of a data consumer’s desired number of data updates to the total number of updates, plus the percentage of an update notification’s size to the total size of the updated data, is less than 100%. Therefore, the research objectives were attained.

In the future, there are a number of areas that can be improved with regard to the current implementation of the selective data replication system. Those areas have been identified, and include both smaller scale functional enhancements and larger scale system extensions and platform migration. However, with experience gained in this research, the future work items should be relatively easy to proceed with.
Bibliography


URL: http://en.wikipedia.org/wiki/Ontology_language


URL: http://www.omg.org/docs/ad/06-05-01.pdf


Hillairet, G. (2007), ‘ATL Use Case - ODM Implementation (Bridging UML and OWL)’.

URL: http://www.eclipse.org/m2m/atl/usecases/ODMImplementation/


URL: http://www.w3.org/Submission/SWRL


Kagal, L. (2002), Rei : A policy language for the me-centric project, Techreport, HP Labs.


Miller, J. & Mukerji, J. (2003), MDA Guide Version 1.0.1, 1.0.1 edn, OMG.


   URL: http://www.opengis.org/standards/sfs


   URL: http://www.omg.org/mda

   URL: http://en.wikipedia.org/wiki/Ontology_(computer_science)

   URL: http://www.mahipalreddy.com/dbdesign/dbarticle1.htm


PostgreSQL Global Development Group (2008), ‘PostgreSQL: The world’s most advanced open source database’.
   URL: http://www.postgresql.org/

   URL: http://postgis.refractions.net

   URL: http://oxygen.informatik.tu-cottbus.de/reverse-i1/?q=R2ML

   URL: http://www3.interscience.wiley.com/cgi-bin/summary/112653345/SUMMARY

   URL: http://www.jessrules.com/jess/index.shtml


Seekda (2008), ‘Web Service Search Engine @ seekda.com’.
   URL: http://seekda.com/

   URL: http://en.wikipedia.org/wiki/Semantic_Web_Services


   URL: http://www.slony.info
Stanford Center for Biomedical Informatics Research (2008a), ‘Protégé’.

URL: http://protege.stanford.edu

Stanford Center for Biomedical Informatics Research (2008b), ‘SWRLBuiltInFAQ’.

URL: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLBuiltInFAQ

Stanford Center for Biomedical Informatics Research (2008c), ‘SWRLJessTab’.

URL: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLJessTab

Stanford Center for Biomedical Informatics Research (2008d), ‘SWRLLanguageFAQ’.

URL: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLLanguageFAQ

Stanford Center for Biomedical Informatics Research (2008e), ‘SWRLTab’.

URL: http://protege.cim3.net/cgi-bin/wiki.pl?SWRLTab

Tigris.org (2008), ‘argouml.tigris.org’.

URL: http://argouml.tigris.org/


URL: http://www.vividsolutions.com/jts/jtshome.htm

W3C (2004), ‘OWL Web Ontology Language Overview’.

URL: http://www.w3.org/TR/owl-features

W3C (2007), ‘Web Ontology Language (OWL)’.

URL: http://www.w3.org/2004/OWL


XMethods (2008), ‘Welcome to XMethods’.

URL: http://www.xmethods.net/ve2/index.po
Appendix A

Ontology Translation

This appendix provides a detailed step-by-step instruction on how to translate the encoding of an ontology from XMI to OWL by using the tool called Protégé\(^1\). The steps described in this appendix are only suitable for Protégé v3.4 beta, and may not be the same for earlier or future releases of Protégé.

The translation from XMI to OWL can be carried out in two ways in Protégé, either by interacting with Protégé’s user interface or by programming against its API. Both of the methods are described in this appendix and the intermediary results are also listed.

A.1 Installation of Required Protégé Plugins

Regardless of which method is used to carry out the desired translation, either manually through the UI or programmatically through the API, the following Protégé plugins are required.

UMLBackend

The default installation package of Protégé requires a separate plugin called UMLBackend\(^2\) to be installed before XMI representations of ontologies expressed using UML can be processed using Protégé. With this plugin, a Protégé knowledge base can then read from and write to a UML model.

To install UMLBackend, simply decompress the contents of the downloaded file into the plugins directory under the Protégé’s installation directory. After doing that, upon restart of Protégé, the entry UML 1.4 Class Diagrams should appear in the Create New Project dialog, and a new menu item UML 1.4 should appear under the File - Export to Format menu.

There is one limitation of UMLBackend, which is that it supports UML 1.4 but not 2.x at the moment, so reading or importing an XMI document generated by more recent UML 2.x compliant CASE tools, such as Rational Rose, Enterprise Architect, MagicDraw and Poseidon for UML, will fail. To overcome this, an XSL transformation style sheet can be created to either translate UML 2.x to 1.4 or translate UML 2.x directly to OWL. Other tools such as TopBraid Composer\(^3\) and doU2O\(^4\) may also become helpful.

---

\(^1\)Protégé: available from http://protege.stanford.edu
\(^2\)UMLBackend: available from http://protege.cim3.net/cgi-bin/wiki.pl?UMLBackend
\(^3\)TopBraid Composer: available from http://www.topquadrant.com/topbraid/composer/index.html
\(^4\)doU2O: available from http://projects.semwebcentral.org/docman/view.php/50/8/DUET_Guides_V0.5.2.doc
SWRLTab

SWRLTab is a development environment for working with SWRL rules in Protégé and it provides mechanisms to allow interoperation with a variety of rule engines, such as Jess. SWRLTab only becomes necessary when SWRL rules are involved in an ontology, which could be the case for data consumers in this research as they are embedding SWRL rules in their UML class diagrams, and these rules are imported into OWL-encoded ontologies during the ontology translation.

SWRLTab comes with the default installation package of Protégé so it does not need to be downloaded and installed separately.

A.2 Ontology Translation using Protégé User Interface

This section presents a step-by-step guide to perform the ontology translation process using Protégé’s user interface. As mentioned before, the following guide is based on Protégé v3.4 beta and may not be the same for future releases of Protégé.

1. run Protégé v3.4 beta;
2. choose File, New Project...;
3. in the popped out Create New Project dialog (Figure A.1(a)), tick Create from Existing Sources and choose UML 1.4 Class Diagrams from the list below, then click Next;
4. in the following screen (Figure A.1(b)), navigate to and select the XMI file that needs to be imported, then click Finish;
   
   ![Create New Project - Step 1](image1.png) ![Create New Project - Step 2](image2.png)

   (a) New Project - Step 1  (b) New Project - Step 2

Figure A.1: How to create a new UML project from an existing XMI document in Protégé

5. after the XMI file is successfully imported into Protégé, all the classes included in the original UML class diagram should appear under Class Hierarchy on the left, and the properties of a class should be displayed in Template Slots on the bottom right when the class is highlighted. Figure A.2 provides an example of an XMI encoded ontology displayed in Protégé.

![Example XMI encoded ontology](image3.png)

Figure A.2: Example of an XMI encoded ontology displayed in Protégé
6. choose File, Export to Format, OWL;

7. in the popped out OWL File to Export dialog, specify a file name and location, then click OK.

After following the above steps, an ontology should now be successfully translated into OWL. However, if there are SWRL rules embedded in the XMI encoded ontology in the way described in Chapter 5, those rules cannot be automatically imported into the OWL-encoded ontology. To achieve this, the following steps have to be performed.

1. open the translated OWL encoded ontology in Protégé. It is very likely that the SWRL Rules tab is not visible.

2. choose Project, Configure..., tick SWRLTab from the list in Tab Widgets (Figure A.3), then click OK:

![Figure A.3: Enable SWRLTab in Protégé](image)

3. go to the just appeared SWRL Rules tab and click Activate SWRL... to import SWRL related ontologies (Figure A.4). After activating SWRL, there are a number of SWRL
related namespaces, classes, properties and individuals added under each corresponding tab in Protégé.

(a) Before activating SWRL

(b) After activating SWRL

Figure A.4: Before and after activating SWRL for an OWL ontology in Protégé

4. open up the original XMI document in a text editor, copy any predefined SWRL rules in it and paste them into the SWRL Rules tab one by one;

5. save the ontology.

A.3 Ontology Translation using Protégé APIs

Protégé provides extensive APIs for programmers to directly work with various parts of Protégé without running the Protégé application. In this research, the following three APIs are used to automate the ontology translation process described in Section A.2.

• UMLBackend API: is used for importing XMI encoded UML class diagrams representing the ontology.

• Protégé Core API: is used to access basic Protégé functionalities and knowledge bases.

• Protégé OWL API: extends the Core API to provide access to OWL ontologies and SWRL related functionalities.

In addition, since Protégé is an open source application, its source code can also become quite useful when a specific functionality is not exposed via an API. For example, the functionality of activating SWRL and importing SWRL related ontologies is not directly exposed through the OWL API, so the source code had to be extracted and reused in this research. Appendix B shows the Java source code for ontology translation.
A.4 Example Input & Output Files

A.4.1 UML Class Diagram

Figure A.5: A UML Class diagram representing the Address ontology with one update policy rule.

A.4.2 Input XMI Ontology Document

Listing A.1: The XMI encoded Address ontology with the update policy.

```xml
<?xml version='1.0' encoding='UTF-8'?>
<XMI xmi.version='1.2' xmlns:UML='org.omg.xmi.namespace.UML'
timestamp='Sun Oct 26 17:06:07 NZDT 2008'>
  <XMI.header>
    <!--XMI.exporter>ArgoUML (using Netbeans XMI Writer version 1.0)</XMI.exporter>
    <XMI.exporterVersion>0.24(5) revised on $Date: 2006-11-06 19:55:22 +0100 (Mon, 06 Nov 2006)$</XMI.exporterVersion>
  </XMI.header>
  <XMI.metamodel xmi.name="UML" xmi.version="1.4"/>
  <UML:Model xmi.id='-64--88-1-103-69166c94:119b27331ca:-8000:000000000000077B'
    name='untitledModel' isSpecification='false' isRoot='false' isLeaf='false'
isAbstract='false'>
    <UML:Namespace ownedElement>
      <UML:Class xmi.id='-64--88-1-103-69166c94:119b27331ca:-8000:000000000000077C'
        name='UpdatedObject' visibility='public' isSpecification='false' isRoot='false'
isLeaf='false' isAbstract='false' isActive='false'>
        <UML:Classifier.feature>
          <UML:Attribute xmi.id='-64--88-1-103-69166c94:119b27331ca:-8000:000000000000077F'
            name='providerId' visibility='public' isSpecification='false'
ownerScope='instance'
changeability='changeable' targetScope='instance'>
            <UML:StructuralFeature.multiplicity>
              <UML:Multiplicity xmi.id='-64--88-1-103--6bada2e0:11d374e925c:-8000:0000000000000818'>
```
<UML:MultiplicityRange xmi.id='-64--88-1-103--6bad2e0:11d374e925c:-8000:0000000000000823' lower='1' upper='1'/>
</UML:Multiplicity>
</UML:StructuralFeature.multiplicity>
<UML:StructuralFeature.type>
<UML:DataType xmi.id.ref='-64--88-1-103-69166c94:119b27331ca:-8000:000000000000077E'/>
</UML:StructuralFeature.type>
</UML:Attribute>
<UML:Attribute xmi.id='127-0-0-1-78fc7d6c:11a192253d3:-8000:00000000000008FC' name='updateType' visibility='public' isSpecification='false' ownerScope='instance' changeability='changeable' targetScope='instance'>
<UML:StructuralFeature.multiplicity>
<UML:Multiplicity xmi.id='-64--88-1-103-69166c94:119b27331ca:-8000:000000000000092C'/>
</UML:Multiplicity>
</UML:StructuralFeature.multiplicity>
<UML:Expression xmi.id='-64--88-1-103-69166c94:119b27331ca:-8000:0000000000000906' language='/' body=''/>
</UML:Attribute.initialValue>
</UML:Class xmi.id.ref='-64--88-1-103-69166c94:119b27331ca:-8000:0000000000000788'/>
</UML:StructuralFeature.type>
</UML:Attribute>
<UML:Attribute xmi.id='127-0-0-1-78fc7d6c:11a192253d3:-8000:000000000000091B' name='updateTime' visibility='public' isSpecification='false' ownerScope='instance' changeability='changeable' targetScope='instance'>
<UML:StructuralFeature.multiplicity>
<UML:Multiplicity xmi.id='-64--88-1-103--6bad2e0:11d374e925c:-8000:0000000000000827' lower='1' upper='1'/>
</UML:Multiplicity>
</UML:StructuralFeature.multiplicity>
</UML:Attribute>
<UML:Attribute xmi.id='127-0-0-1-78fc7d6c:11a192253d3:-8000:0000000000000920' name='isSignificant' visibility='public' isSpecification='false' ownerScope='instance' changeability='changeable' targetScope='instance'>
<UML:StructuralFeature.multiplicity>
<UML:Multiplicity xmi.id='-64--88-1-103--6bad2e0:11d374e925c:-8000:0000000000000827' lower='1' upper='1'/>
</UML:Multiplicity>
</UML:StructuralFeature.multiplicity>
</UML:Attribute>
A.4. EXAMPLE INPUT & OUTPUT FILES

```xml
<uml:Multiplicity range="1" upper="1"/>
<uml:Multiplicity range="1" upper="1"/>
<uml:Multiplicity range="1" upper="1"/>
<uml:Multiplicity range="1" upper="1"/>
```

```xml
<uml:StructuralFeature multiplicity="1" type="UML:Class xmi:idref='-64--88-1-103-69166c94:119b27331ca:-8000:000000000000077E'"/>
<uml:Attribute xmi:id='-64--88-1-103-69166c94:119b27331ca:-8000:000000000000079E' name='houseLow' visibility='public' isSpecification='false' ownerScope='instance' changeability='changeable' targetScope='instance'>
<uml:Multiplicity range="1" upper="1"/>
</uml:Attribute>
<uml:StructuralFeature multiplicity="1" type="UML:Class xmi:idref='-64--88-1-103-69166c94:119b27331ca:-8000:00000000000007A2'"/>
<uml:Attribute xmi:id='-64--88-1-103-69166c94:119b27331ca:-8000:00000000000007A1' name='houseHigh' visibility='public' isSpecification='false' ownerScope='instance' changeability='changeable' targetScope='instance'>
<uml:Multiplicity range="1" upper="1"/>
</uml:Attribute>
<uml:StructuralFeature multiplicity="1" type="UML:Class xmi:idref='-64--88-1-103-69166c94:119b27331ca:-8000:0000000000000788'"/>
<uml:Attribute xmi:id='-64--88-1-103-69166c94:119b27331ca:-8000:0000000000000788' name='roadName' visibility='public' isSpecification='false' ownerScope='instance' changeability='changeable' targetScope='instance'>
<uml:Multiplicity range="1" upper="1"/>
</uml:Attribute>
```

A.4.3 Output OWL Ontology Document

Listing A.2: The OWL encoded Address ontology with the update policy.

```xml
<?xml version="1.0"?>
<rdf:RDF
    xmlns:temporal=http://swrl.stanford.edu/ontologies/built-ins/3.3/temporal.owl#
    xmlns:swrla=http://swrl.stanford.edu/ontologies/3.3/swrla.owl#
    xmlns:swrlxml=http://swrl.stanford.edu/ontologies/built-ins/3.4/swrlxml.owl#
    xmlns:swrl=http://www.w3.org/2003/11/swrl#
    xmlns:swrlx=http://swrl.stanford.edu/ontologies/built-ins/3.3/swrlx.owl#
    xmlns:owl=http://www.w3.org/2002/07/owl#
    xmlns:xsd=http://www.w3.org/2001/XMLSchema#
    xmlns:sqwrl=http://sqwrl.stanford.edu/ontologies/built-ins/3.4/sqwrl.owl#
    xmlns:tbox=http://swrl.stanford.edu/ontologies/built-ins/3.3/tbox.owl#>
    <owl:Ontology rdf:about=""/>
</rdf:RDF>
```
</owl:Ontology>
<owl:Class rdf:ID="Address"/>
<owl:Class rdf:ID="UpdatedObject"/>
<owl:Class rdf:ID="Date"/>
<owl:DatatypeProperty rdf:ID="updateTime">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#DateTime"/>
  <rdfs:domain rdf:resource="#UpdatedObject"/>
</owl:DatatypeProperty>
<owl:FunctionalProperty rdf:ID="gid">
  <rdfs:domain rdf:resource="#Address"/>
  <rdfs:range rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="isSignificant">
  <rdfs:domain rdf:resource="#UpdatedObject"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="roadType">
  <rdfs:domain rdf:resource="#Address"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="houseHigh">
  <rdfs:domain rdf:resource="#Address"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="geometry">
  <rdfs:domain rdf:resource="#Address"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="objectType">
  <rdfs:domain rdf:resource="#UpdatedObject"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="address">
  <rdfs:domain rdf:resource="#UpdatedObject"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="id">
  <rdfs:domain rdf:resource="#UpdatedObject"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="providerId">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:FunctionalProperty>
<owl:FunctionalProperty rdf:ID="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
A.4. EXAMPLE INPUT & OUTPUT FILES

```
<owl:FunctionalProperty rdf:ID="updateType">
  <rdfs:domain rdf:resource="#UpdatedObject"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:ID="idField">
  <rdfs:domain rdf:resource="#UpdatedObject"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:ID="houseLow">
  <rdfs:domain rdf:resource="#Address"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:ID="roadName">
  <rdfs:domain rdf:resource="#Address"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<owl:FunctionalProperty rdf:ID="addressType">
  <rdfs:domain rdf:resource="#Address"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
</owl:FunctionalProperty>

<swrl:Impl rdf:ID="ReplicateEveryAddressUpdate">
  <swrl:head>
    <swrl:AtomList>
      <rdf:first>
        <swrl:DatavaluedPropertyAtom>
          <swrl:argument1 rdf:ID="#s"/>
          <swrl:argument2 rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
        </swrl:DatavaluedPropertyAtom>
      </rdf:first>
    </swrl:AtomList>
  </swrl:head>
  <swrl:body>
    <swrl:AtomList>
      <rdf:first>
        <swrl:ClassAtom>
          <swrl:argument1 rdf:resource="#x"/>
        </swrl:ClassAtom>
      </rdf:first>
    </swrl:AtomList>
  </swrl:body>
```

<swrl:Variable rdf:ID="t"/>
<swrl:argument2/>
<swrl:propertyPredicate rdf:resource="#objectType"/>
<swrl:argument1 rdf:resource="#x"/>
<swrl:DatavaluedPropertyAtom>
  <rdf:first>
    <swrl:AtomList>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
      <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
Appendix B

Source Code

Please contact the author for the full source code of the selective data replication system.

B.1 Source Code for Ontology Translation using Protégé APIs

Listing B.1: Source Code for Ontology Translation using Protégé APIs.

```java
import com.hp.hpl.jena.util.FileUtils;
import edu.stanford.smi.protege.model.KnowledgeBase;
import edu.stanford.smi.protege.model.Project;
import edu.stanford.smi.protege.owl.ProtégéOWL;
import edu.stanford.smi.protegex.owl.jena.JenaKnowledgeBaseFactory;
import edu.stanford.smi.protegex.owl.jena.JenaOWLModel;
import edu.stanford.smi.protegex.owl.model.OWLModel;
import edu.stanford.smi.protegex.owl.model.OWLOntology;
import edu.stanford.smi.protegex.owl.model.RDFProperty;
import edu.stanford.smi.protegex.owl.model.util.ImportHelper;
import edu.stanford.smi.protegex.owl.storage.ProtégéSaver;
import edu.stanford.smi.protegex.owl.swrl.model.SWRLFactory;
import edu.stanford.smi.protegex.owl.swrl.model.SWRLImp;
import edu.stanford.smi.protegex.owl.swrl.model.SWRLNames;
import edu.stanford.smi.protegex.owl.swrl.parser.SWRLParseException;
import edu.stanford.smi.protegex.storage.uml.UMLImport;
import edu.stanford.smi.protegex.storage.uml.UMLOntology;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.xml.XMLConstants;
import javax.xml.namespace.NamespaceContext;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

/** *
 */
```
public class OntologyLoader {

    private static final String RULE_STEREOTYPE_NAME = "UpdatePolicyRule";

    public static OWLModel loadOwlModelAndRulesFromXmi(String xmiFilePath) {
        JenaOWLModel owlModel = null;

        // Create a collection for containing processing errors.
        Collection errors = new ArrayList();

        try {
            KnowledgeBase umlKb = loadUmlModelFromFile(xmiFilePath, errors);
            owlModel = (JenaOWLModel) getOwlModelFromUml(umlKb, errors);

            // Extract SWRL rules from the UML 1.4 ontology.
            Hashtable<String, String> namedRules = OntologyLoader.loadRulesFromXmiFile(xmiFilePath);

            // Activate SWRL related constructs for the OWL ontology.
            activateSwrl(owlModel);
            SWRLFactory swrlFactory = new SWRLFactory(owlModel);

            // Import SWRL rules.
            for (String ruleName : namedRules.keySet()) {
                String ruleBody = namedRules.get(ruleName);
                SWRLImp imp = swrlFactory.createImp(ruleName, ruleBody);
            }
        }
        catch (SWRLParseException ex) {
            Logger.getLogger(OntologyLoader.class.getName()).log(Level.SEVERE, null, ex);
        }
        catch (Exception ex) {
            Logger.getLogger(OntologyLoader.class.getName()).log(Level.SEVERE, null, ex);
        }

        return owlModel;
    }

    public static void convertUmlToOwl(String xmiFilePath, String owlFilePath) {
        // Create a collection for containing processing errors.
        Collection errors = new ArrayList();

        KnowledgeBase umlKb = loadUmlModelFromFile(xmiFilePath, errors);
        JenaOWLModel owlModel = (JenaOWLModel) getOwlModelFromUml(umlKb, errors);

        // Save to file.
        owlModel.save(new File(owlFilePath).toURI(), FileUtilities.langXMLAbbrev, errors);
    }

    public static KnowledgeBase loadUmlModelFromFile(String xmiFilePath) {
        return loadUmlModelFromFile(xmiFilePath, new ArrayList());
    }

    public static KnowledgeBase loadUmlModelFromFile(String xmiFilePath, Collection errors) {
B.1. SOURCE CODE FOR ONTOLOGY TRANSLATION USING PROTEGE
APIS

```java
{
   // Create a UML knowledgebase.
   UMLKnowledgeBaseFactory umlKbFactory = new UMLKnowledgeBaseFactory();
   KnowledgeBase umlKb = umlKbFactory.createKnowledgeBase(errors);

   // Import the given XMI file into the knowledgebase.
   UMLImport.importFile(umlKb, errors, xmiFilePath);
   return umlKb;
}

public static JenaOWLModel loadOwlModelFromFile(String owlFilePath) throws Exception {
    // Import the OWL ontology from the given file path.
    return ProtegeOWL.createJenaOWLModelFromURI(new File(owlFilePath).toURI().toString());
}

public static OWLModel getOwlModelFromUml(KnowledgeBase umlKb) {
    return getOwlModelFromUml(umlKb, new ArrayList());
}

public static OWLModel getOwlModelFromUml(KnowledgeBase umlKb, Collection errors) {
    // Create a Jena OWL knowledgebase as the target knowledgebase.
    JenaKnowledgeBaseFactory factory = new JenaKnowledgeBaseFactory();
    Project newProject = Project.createNewProject(factory, errors);
    JenaOWLModel owlModel = (JenaOWLModel) newProject.getKnowledgeBase();

    // Convert the UML 1.4 ontology encoded in XMI into OWL.
    new ProtegeSaver(umlKb, owlModel).run();
    return owlModel;
}

public static Hashtable<String, String> loadRulesFromXmiFile(String xmiFilePath) {
    Hashtable<String, String> namedRules = new Hashtable<String, String>();
    try {
        InputSource inputSrc = new InputSource(new FileInputStream(xmiFilePath));
        XPath xpath = XPathFactory.newInstance().newXPath().newXPath().setNamespaceContext(new NamespaceContext() {
            public String getNamespaceURI(String prefix) {
                if (prefix.equals("UML"))
                    return "org.omg.xmi.namespace.UML";
                return XMLConstants.NULL_NS_URI;
            }

            public String getPrefix(String namespaceURI) {
                return null;
            }

            public Iterator getPrefixes(String namespaceURI) {
                return null;
            }
        });
    }
    try {
        InputSource inputSrc = new InputSource(new FileInputStream(xmiFilePath));
        XPath xpath = XPathFactory.newInstance().newXPath().newXPath().setNamespaceContext(new NamespaceContext() {
            public String getNamespaceURI(String prefix) {
                if (prefix.equals("UML"))
                    return "org.omg.xmi.namespace.UML";
                return XMLConstants.NULL_NS_URI;
            }

            public String getPrefix(String namespaceURI) {
                return null;
            }

            public Iterator getPrefixes(String namespaceURI) {
                return null;
            }
        });
    }
```
String xpathStr = "//UML:Comment[descendant::UML:Stereotype[@xmi.idref=" + "//UML:Stereotype[@name='" + RULE_STEREOTYPE_NAME + "+" + 
(and, UML:Stereotype.baseClass, 'Comment')/@xmi.id]]";

XPathExpression expr = xpath.compile(xpathStr);
NodeList commentNodeList = (NodeList) expr.evaluate(inputSrc, XPathConstants.NODESET);

for (int i = 0; i < commentNodeList.getLength(); i++)
{
    Node commentNode = commentNodeList.item(i);
    String name = commentNode.getAttributes().getNamedItem("name").getNodeValue();
    String body = commentNode.getAttributes().getNamedItem("body").getNodeValue();

    // Some special characters are required by Protege to interpret SWRL rules.
    body = body.replace("\", \").replace("->", ");

    if (name != null && !name.equals(""))
    {
        if (body != null && !body.equals(""))
        {
            namedRules.put(name, body);
        }
    }
}

catch (IOException ex)
{
    Logger.getLogger(OntologyLoader.class).log(Level.SEVERE, null, ex);
}

catch (XPathExpressionException ex)
{
    Logger.getLogger(OntologyLoader.class).log(Level.SEVERE, null, ex);
}

return namedRules;

public static void activateSswrl(OWLModel owlModel) throws Exception
{
    // Check to see if SWRL has already been imported.
    if (isSwrlImported(owlModel))
    {
        return;
    }

    // Enable SWRL related namespace.
    owlModel.getNamespaceManager().setPrefix(new URI(SWRLNames.SWRL_NS), SWRLNames.SWRL_PREFIX);
    owlModel.getNamespaceManager().setPrefix(new URI(SWRLNames.SWRLB_NS), SWRLNames.SWRLB_PREFIX);
    owlModel.getNamespaceManager().setPrefix(new URI(SWRLNames.SWRLA_NS), SWRLNames.SWRLA_PREFIX);
    owlModel.getNamespaceManager().setPrefix(new URI(SWRLNames.SWRLX_NS), SWRLNames.SWRLX_PREFIX);
    owlModel.getNamespaceManager().setPrefix(new URI(SWRLNames.SWRLM_NS), SWRLNames.SWRLM_PREFIX);
    owlModel.getNamespaceManager().setPrefix(new URI(SWRLNames.SWRLTBOX_NS), SWRLNames.SWRLTBOX_PREFIX);
    owlModel.getNamespaceManager().setPrefix(new URI(SWRLNames.SWRLABOX_NS), SWRLNames.SWRLABOX_PREFIX);
    owlModel.getNamespaceManager().setPrefix(new URI(SWRLNames.SWRLTEMPORAL_NS), SWRLNames.SWRLTEMPORAL_PREFIX);
    owlModel.getNamespaceManager().setPrefix(new URI(SWRLNames.SWRLXML_NS), SWRLNames.SWRLXML_PREFIX);
}
B.1. SOURCE CODE FOR ONTOLOGY TRANSLATION USING PROTÉGÉ APIS

```java
owlModel.getNamespaceManager().setParameter(new URI(SWRLNames.SQWRLAMESPACE),
   SWRLNames.SQWRLPREFIX);
owlModel.getNamespaceManager().setParameter(new URI("http://swrl.canterbury.ac.nz/
cosc/ontologies/built-ins/geo/swrlgeo.owl"), "swrlgeo");

ImportHelper importHelper = new ImportHelper((JenaOWLModel) owlModel);
String curDir = OntologyLoader.class.getResource(".*").toString();
if (!curDir.endsWith("/")) curDir += "/
;
   // Import SWRL related ontologies.
   // importHelper.addImport(new URI(SWRLNames.SWRLIMPORT));
   // importHelper.addImport(new URI(SWRLNames.SWRLXIMPORT));
   // importHelper.addImport(new URI(SWRLNames.SWRLMIMPORT));
   // importHelper.addImport(new URI(SWRLNames.SWRLTBOXIMPORT));
   // importHelper.addImport(new URI(SWRLNames.SWRLTEMPORALIMPORT));
   // importHelper.addImport(new URI(SWRLNames.SWRLXMIMPORT));
   // importHelper.addImport(new URI("http://swrl.canterbury.ac.nz/cosc/ontologies/
   // built-ins/geo/swrlgeo.owl"));
   // Use already downloaded ontologies, so that the Internet connection is not
   // required.
   importHelper.addImport(new URI(curDir + "swrla.owl"));
   importHelper.addImport(new URI(curDir + "swrlx.owl"));
   importHelper.addImport(new URI(curDir + "swrlm.owl"));
   importHelper.addImport(new URI(curDir + "tbox.owl"));
   importHelper.addImport(new URI(curDir + "abox.owl"));
   importHelper.addImport(new URI(curDir + "temporal.owl"));
   importHelper.addImport(new URI(curDir + "swrlxml.owl"));
   importHelper.addImport(new URI(curDir + "swrlgeo.owl"));
   importHelper.importOntologies();
   // Make ":TO" and ":FROM" visible for dynamic expansion.
   RDFProperty to = owlModel.getOWLObjectProperty(edu.stanford.smi.protege.model.
      Model.Slot.TO);
   to.setVisible(true);
   RDFProperty from = owlModel.getOWLObjectProperty(edu.stanford.smi.protege.model.
      Model.Slot.FROM);
   from.setVisible(true);
}

private static boolean isSwrlImported(OWLModel owlModel) {
   boolean swrlFound = false;
   boolean swrlbFound = false;
   Iterator iter = owlModel.getOWLOntologies().iterator();
   while (iter.hasNext() && !(swrlbFound && swrlFound)) {
      OWLOntology ont = (OWLOntology) iter.next();
      if (ont.getNamespace().equals(SWRLNames.SWRLNAMESPACE))
         swrlFound = true;
      if (ont.getNamespace().equals(SWRLNames.SWRLBNAMESPACE))
         swrlbFound = true;
   }
   return swrlFound && swrlbFound;
}
```
### B.2 Source Code for Update Notification Retrieval


```java
import java.io.*;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.Calendar;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.postgresql.*;
import org.w3c.dom.*;
import com.megginson.sax.DataWriter;
import exceptions.*;
import helpers.*;
import model.notification.*;
import org.xml.sax.helpers.AttributesImpl;

/**
 * @author Xuan Gu
 */
public class NotificationManager {

    public static byte[] getNotifications(Connection conn, int consumerId, String[] featureTypes)
            throws NotificationGenerationException, InvalidFeatureTypeException, SQLException {
        ArrayList<UpdateNotification> notifications = new ArrayList<UpdateNotification>();
        Timestamp endTimestamp = new Timestamp(Calendar.getInstance().getTime().getTime());

        if (featureTypes == null || featureTypes.length == 0)
            featureTypes = SubscriptionManager.getConsumerSubscribedFeatureTypes(conn, consumerId); toArray(featureTypes);
        else {
            ArrayList<String> allFeatureTypes = DatabaseHelper.getSubscribableFeatureTypes(conn);
            ArrayList<String> featureTypeList = new ArrayList<String>(featureTypes.length);
            for (String ft : featureTypes)
                featureTypeList.add(ft);
            if (!allFeatureTypes.containsAll(featureTypeList))
                throw new InvalidFeatureTypeException();
        }

        for (String ftName : featureTypes) {
            Timestamp startTimestamp = SubscriptionManager.getLastUpdateTime(conn, consumerId, ftName);
            UpdateNotification notification = NotificationManager.getNotificationForFeatureType(conn, consumerId, ftName, startTimestamp, endTimestamp);
            notifications.add(notification);
        }

        return notifications.toArray(new byte[0][]);
    }
```

if (notification != null)
    notifications.add(notification);
}
SubscriptionManager.setLastUpdateTime(conn, consumerId, featureTypes, endTimestamp);
return NotificationManager.serializeNotifications(notifications);
}

public static byte[] getNotifications(Connection conn, int consumerId, String[] featureTypes, Timestamp startTimestamp, Timestamp endTimestamp)
    throws NotificationGenerationException, InvalidFeatureTypeException, SQLException {
    ArrayList<UpdateNotification> notifications = new ArrayList<UpdateNotification>();
    if (featureTypes == null || featureTypes.length == 0)
        featureTypes = SubscriptionManager.getConsumerSubscribedFeatureTypes(conn, consumerId).toArray(featureTypes);
    else {
        ArrayList<String> allFeatureTypes = DatabaseHelper.getSubscribableFeatureTypes(conn);
        ArrayList<String> featureTypeList = new ArrayList<String>(featureTypes.length);
        for (String ft : featureTypes)
            featureTypeList.add(ft);
        if (!allFeatureTypes.containsAll(featureTypeList))
            throw new InvalidFeatureTypeException();
    }
    for (String fName : featureTypes)
        {  
            UpdateNotification notification = NotificationManager.getNotificationForFeatureType(conn, consumerId, fName, startTimestamp, endTimestamp);
            if (notification != null)
                notifications.add(notification);
        }
    return NotificationManager.serializeNotifications(notifications);
}
private static UpdateNotification getNotificationForFeatureType(Connection conn, int consumerId, String featureType, Timestamp startTimestamp, Timestamp endTimestamp)
    throws SQLException {
    Statement s = conn.createStatement();
    try {
        ArrayList<String> interestedFields = SubscriptionManager.getConsumerInterestedFields(conn, consumerId, featureType);
        if (interestedFields == null || interestedFields.size() == 0)
            interestedFields = DatabaseHelper.getRelationNames(conn, featureType);
        if (interestedFields == null || interestedFields.size() == 0)
            return null;
        String fields = "";
        for (String field : interestedFields)
            fields += "\"" + field + "\"", "
        fields = fields.substring(0, fields.length() - 1);
APPENDIX B. SOURCE CODE

```java
String sql = "select " + DatabaseHelper.DATA_ID_FIELD + "," + changeType + "changeTime" + fields + "from " + featureType + DatabaseHelper.CHANGE_TABLE_SURFFIX + " where " + changeTime + ">" + startTimestamp + " and " + changeTime + "<" + endTimestamp + ";

ResultSet r = s.executeQuery(sql);

UpdateNotification notification = new UpdateNotification(ServiceHelper.PROVIDER_ID, featureType, startTimestamp, endTimestamp);
DataUpdateDetails updateDetails = notification.getDataUpdateDetails();

while (r.next()) {
    String id = r.getString(1);
    String changeTypeStr = r.getString(2).toUpperCase();
    DataUpdateDetail.DataUpdateType changeType;
    if (changeTypeStr.equals("INSERT"))
        changeType = DataUpdateDetail.DataUpdateType.INSERT;
    else if (changeTypeStr.equals("DELETE"))
        changeType = DataUpdateDetail.DataUpdateType.DELETE;
    else
        changeType = DataUpdateDetail.DataUpdateType.UPDATE;
    Timestamp changeTime = r.getTimestamp(3);
    DataUpdateDetail detail = new DataUpdateDetail(DatabaseHelper.DATA_ID_FIELD, id, changeType, changeTime);
    for (int i = 0; i < interestedFields.size(); i++)
        { String fieldName = interestedFields.get(i);
            String fieldValue = r.getString(i + 4);
            detail.addInterestedFieldValue(fieldName, fieldValue);
        }
    updateDetails.add(detail);
}

if (updateDetails.size() == 0)
    return null;
else
    return notification;
}

finally
{
    s.close();
}

private static byte[] serializeNotifications(ArrayList<UpdateNotification> notifications) throws NotificationGenerationException
{
    try
    {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        DataWriter writer = new DataWriter(new BufferedWriter(new OutputStreamWriter(outputStream)));
        AttributesImpl atts = new AttributesImpl();
        writer.startDocument();
        writer.startElement("UpdateNotifications";
```
B.2. SOURCE CODE FOR UPDATE NOTIFICATION RETRIEVAL

```java
for (UpdateNotification notification : notifications) {
   atts.clear();
   atts.addAttribute("", "ProviderId", "ProviderId", "CDATA", notification.getProviderId());
   atts.addAttribute("", "StartTime", "StartTime", "CDATA", DatabaseHelper.toSqlDateString(notification.getStartTime()));
   atts.addAttribute("", "EndTime", "EndTime", "CDATA", DatabaseHelper.toSqlDateString(notification.getEndTime()));
   atts.addAttribute("", "FeatureType", "FeatureType", "CDATA", notification.getFeatureType());
   atts.addAttribute("", "IdField", "IdField", "CDATA", notification.getFeatureTypeIdField());
   atts.addAttribute("", "IsFull", "IsFull", "CDATA", notification.getIsFull() + "");
   writer.startElement("", "UpdateNotification", "UpdateNotification", atts);

    DataUpdateDetails details = notification.getDataUpdateDetails();
   atts.clear();
   atts.addAttribute("", "TotalInsertions", "TotalInsertions", "CDATA", details.getTotalInsertions() + "");
   atts.addAttribute("", "TotalDeletions", "TotalDeletions", "CDATA", details.getTotalDeletions() + "");
   atts.addAttribute("", "TotalUpdates", "TotalUpdates", "CDATA", details.getTotalUpdates() + "");
   writer.startElement("", "DataUpdateDetails", "DataUpdateDetails", atts);

    for (DataUpdateDetail detail : details.getAll()) {
       atts.clear();
       atts.addAttribute("", "Id", "Id", "CDATA", detail.getId());
       atts.addAttribute("", "UpdateType", "UpdateType", "CDATA", detail.getChangeType().name().toUpperCase());
       atts.addAttribute("", "UpdateTime", "UpdateTime", "CDATA", DatabaseHelper.toSqlDateString(detail.getChangeTime()));
       writer.startElement("", "DataUpdateDetail", "DataUpdateDetail", atts);

        for (String key : detail.getInterestedFields()) {
           atts.clear();
           atts.addAttribute("", "Name", "Name", "CDATA", key);
           atts.addAttribute("", "Value", "Value", "CDATA", detail.getInterestedFieldValue(key));
           writer.startElement("", "Property", "Property", atts);
           writer.endElement("Property");
        }
        writer.endElement("DataUpdateDetail");
    }
    writer.endElement("DataUpdateDetails");
    writer.endElement("UpdateNotification");
}
```

return outputStream.toByteArray();

catch (Exception e)
```java
{ e.printStackTrace();
    throw new NotificationGenerationException(e);
}
```
Appendix C

SWRL Built-Ins for Geospatial Operations

Listing C.1: SWRL Built-Ins for Geospatial Operations.

```java
package edu.stanford.smi.protegex.owl.swrl.bridge.builtin.swrlgeo;
import com.vividsolutions.jts.geom.Geometry;
import com.vividsolutions.jts.io.∗;
import edu.stanford.smi.protegex.owl.swrl.bridge.∗;
import edu.stanford.smi.protegex.owl.swrl.bridge.builtin.∗;
import edu.stanford.smi.protegex.owl.swrl.bridge.exceptions.∗;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

public class SWRLBuiltInLibraryImpl extends AbstractSWRLBuiltInLibrary {
    private static String SWRLGEOLibraryName = "SWRLGeoSpatialOpBuiltIns";
    private ArgumentFactory argumentFactory;

    public SWRLBuiltInLibraryImpl() {
        super(SWRLGEOLibraryName);
        argumentFactory = ArgumentFactory.getFactory();
    }

    @Override
    public void reset() throws BuiltInException {
    }

    public boolean equals(List<BuiltInArgument> arguments) throws BuiltInException {
        try {
            SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(2, arguments.size());
            String strA = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
            String strB = SWRLBuiltInUtil.getArgumentAsString(1, arguments);

            WKTReader reader = new WKTReader();
            Geometry geoA = reader.read(strA);
            Geometry geoB = reader.read(strB);

            return geoA.equals(geoB);
        }
    }
}
```
public boolean disjoint(List< BuiltInArgument > arguments) throws BuiltInException {
    try {
        SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(2, arguments.size());
        String strA = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
        String strB = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
        WKTReader reader = new WKTReader();
        Geometry geoA = reader.read(strA);
        Geometry geoB = reader.read(strB);
        return geoA.disjoint(geoB);
    } catch (ParseException ex) {
        Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
        throw new BuiltInException("Error parsing String into Geometry: \u201c" + ex.getMessage());
    }
}

public boolean intersects(List< BuiltInArgument > arguments) throws BuiltInException {
    try {
        SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(2, arguments.size());
        String strA = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
        String strB = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
        WKTReader reader = new WKTReader();
        Geometry geoA = reader.read(strA);
        Geometry geoB = reader.read(strB);
        return geoA.intersects(geoB);
    } catch (ParseException ex) {
        Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
        throw new BuiltInException("Error parsing String into Geometry: \u201c" + ex.getMessage());
    }
}

public boolean touches(List< BuiltInArgument > arguments) throws BuiltInException {
    try {
        SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(2, arguments.size());
        String strA = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
        String strB = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
        WKTReader reader = new WKTReader();
    } catch (ParseException ex) {
        Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
        throw new BuiltInException("Error parsing String into Geometry: \u201c" + ex.getMessage());
    }
}
Geometry geoA = reader.read(strA);
Geometry geoB = reader.read(strB);
return geoA.touches(geoB);
}
catch (ParseException ex)
{
Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
throw new BuiltInException("Error parsing String into Geometry: "+ ex.getMessage());
}

public boolean crosses(List<BuiltInArgument> arguments) throws BuiltInException
{
try
{
SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(2, arguments.size());
String strA = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
String strB = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
WKTReader reader = new WKTReader();
Geometry geoA = reader.read(strA);
Geometry geoB = reader.read(strB);
return geoA.crosses(geoB);
}
catch (ParseException ex)
{
Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
throw new BuiltInException("Error parsing String into Geometry: "+ ex.getMessage());
}

public boolean within(List<BuiltInArgument> arguments) throws BuiltInException
{
try
{
SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(2, arguments.size());
String strA = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
String strB = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
WKTReader reader = new WKTReader();
Geometry geoA = reader.read(strA);
Geometry geoB = reader.read(strB);
return geoA.within(geoB);
}
catch (ParseException ex)
{
Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
throw new BuiltInException("Error parsing String into Geometry: "+ ex.getMessage());
}

public boolean contains(List<BuiltInArgument> arguments) throws BuiltInException
{
try
{
SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(2, arguments.size());
}
String strA = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
String strB = SWRLBuiltInUtil.getArgumentAsString(1, arguments);

WKTReader reader = new WKTReader();
Geometry geoA = reader.read(strA);
Geometry geoB = reader.read(strB);

return geoA.contains(geoB);
}

} catch (ParseException ex) {
    Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
    throw new BuiltInException("Error parsing String into Geometry: \n" + ex.getMessage());
}

}

public boolean overlaps(List<BuiltInArgument> arguments) throws BuiltInException {
    try {
        SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(2, arguments.size());
        String strA = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
        String strB = SWRLBuiltInUtil.getArgumentAsString(1, arguments);

        WKTReader reader = new WKTReader();
        Geometry geoA = reader.read(strA);
        Geometry geoB = reader.read(strB);

        return geoA.overlaps(geoB);
    }
    catch (ParseException ex) {
        Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
        throw new BuiltInException("Error parsing String into Geometry: \n" + ex.getMessage());
    }
}

public boolean intersection(List<BuiltInArgument> arguments) throws BuiltInException {
    try {
        SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(3, arguments.size());
        SWRLBuiltInUtil.checkForUnboundNonFirstArguments(arguments);

        String strA = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
        String strB = SWRLBuiltInUtil.getArgumentAsString(2, arguments);

        WKTReader reader = new WKTReader();
        Geometry geoA = reader.read(strA);
        Geometry geoB = reader.read(strB);

        Geometry geoResult = geoA.intersection(geoB);
        String strResult = geoResult.toText();

        if (SWRLBuiltInUtil.isUnboundArgument(0, arguments)) {
            arguments.set(0, argumentFactory.createDatatypeValueArgument(strResult))
        }
        return true;
    }
else {
    String str0 = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
    return strResult.equals(str0);
}
}

public boolean union(List< BuiltInArgument> arguments) throws BuiltInException {
    try {
        SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(3, arguments.size());
        SWRLBuiltInUtil.checkForUnboundNonFirstArguments(arguments);
        String strA = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
        String strB = SWRLBuiltInUtil.getArgumentAsString(2, arguments);
        WKTReader reader = new WKTReader();
        Geometry geoA = reader.read(strA);
        Geometry geoB = reader.read(strB);
        Geometry geoResult = geoA.union(geoB);
        String strResult = geoResult.toText();
        if (SWRLBuiltInUtil.isUnboundArgument(0, arguments)) {
            arguments.set(0, argumentFactory.createDatatypeValueArgument(strResult));
            return true;
        } else {
            String str0 = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
            return strResult.equals(str0);
        }
    }
    catch (ParseException ex) {
        Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
        throw new BuiltInException("Error parsing String into Geometry: "+ex.getMessage());
    }
}

public boolean difference(List< BuiltInArgument> arguments) throws BuiltInException {
    try {
        SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(3, arguments.size());
        SWRLBuiltInUtil.checkForUnboundNonFirstArguments(arguments);
        String strA = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
        String strB = SWRLBuiltInUtil.getArgumentAsString(2, arguments);
        WKTReader reader = new WKTReader();
        Geometry geoA = reader.read(strA);
Geometry geoB = reader.read(strB);
Geometry geoResult = geoA.difference(geoB);
String strResult = geoResult.toText();

if (SWRLBuiltInUtil.isUnboundArgument(0, arguments))
{
    arguments.set(0, argumentFactory.createDatatypeValueArgument(strResult));
    return true;
}
else
{
    String str0 = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
    return strResult.equals(str0);
}
}

public boolean symDifference(List<BuiltInArgument> arguments) throws BuiltInException
{
    try
    {
        SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(3, arguments.size());
        String strA = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
        String strB = SWRLBuiltInUtil.getArgumentAsString(2, arguments);
        WKTReader reader = new WKTReader();
        Geometry geoA = reader.read(strA);
        Geometry geoB = reader.read(strB);
        Geometry geoResult = geoA.symDifference(geoB);
        String strResult = geoResult.toText();
        if (SWRLBuiltInUtil.isUnboundArgument(0, arguments))
        {
            arguments.set(0, argumentFactory.createDatatypeValueArgument(strResult));
            return true;
        }
        else
        {
            String str0 = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
            return strResult.equals(str0);
        }
    }
    catch (ParseException ex)
    {
        Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
        throw new BuiltInException("Error parsing String into Geometry: "+ ex.getMessage());
    }
}

public boolean bufferRound(List<BuiltInArgument> arguments) throws BuiltInException
{
{  
try  
  
SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(3, arguments.size());
SWRLBuiltInUtil.checkForUnboundNonFirstArguments(arguments);

String str = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
double distance = SWRLBuiltInUtil.getArgumentAsADouble(2, arguments);

WKTRender reader = new WKTRender();

Geometry geo = reader.read(str);
BufferOp bufOp = new BufferOp(geo);
bufOp.setEndCapStyle(BufferOp.CAP_ROUND);

Geometry geoResult = bufOp.getResultGeometry(distance);
String strResult = geoResult.toText();

if (SWRLBuiltInUtil.isUnboundArgument(0, arguments))
  
  arguments.set(0, argumentFactory.createDatatypeValueArgument(strResult))
  
  return true;
else
  
  String str0 = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
  
  return strResult.equals(str0);
}
}

public boolean bufferButt(List<BuiltInArgument> arguments) throws BuiltInException
{
try
  
SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(3, arguments.size());
SWRLBuiltInUtil.checkForUnboundNonFirstArguments(arguments);

String str = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
double distance = SWRLBuiltInUtil.getArgumentAsADouble(2, arguments);

WKTRender reader = new WKTRender();

Geometry geo = reader.read(str);
BufferOp bufOp = new BufferOp(geo);
bufOp.setEndCapStyle(BufferOp.CAP_BUTT);

Geometry geoResult = bufOp.getResultGeometry(distance);
String strResult = geoResult.toText();

if (SWRLBuiltInUtil.isUnboundArgument(0, arguments))
  
  arguments.set(0, argumentFactory.createDatatypeValueArgument(strResult))
  
  return true;
else
  
  String str0 = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
public boolean bufferSquare(List< BuiltInArgument> arguments) throws BuiltInException {
    try {
        SWRLBuiltInUtil.checkNumberOfArgumentsEqualTo(3, arguments.size());
        SWRLBuiltInUtil.checkForUnboundNonFirstArguments(arguments);

        String str = SWRLBuiltInUtil.getArgumentAsString(1, arguments);
        double distance = SWRLBuiltInUtil.getArgumentAsADouble(2, arguments);

        WKTReader reader = new WKTReader();
        Geometry geo = reader.read(str);
        BufferOp bufOp = new BufferOp(geo);
        bufOp.setEndCapStyle(BufferOp.CAP_SQUARE);

        Geometry geoResult = bufOp.getResultGeometry(distance);
        String strResult = geoResult.toText();

        if (SWRLBuiltInUtil.isUnboundArgument(0, arguments)) {
            arguments.set(0, argumentFactory.createDatatypeValueArgument(strResult));
            return true;
        } else {
            String str0 = SWRLBuiltInUtil.getArgumentAsString(0, arguments);
            return strResult.equals(str0);
        }
    }
    catch (ParseException ex) {
        Logger.getLogger(SWRLBuiltInLibraryImpl.class.getName()).log(Level.SEVERE, null, ex);
        throw new BuiltInException("Error parsing String into Geometry: "+ ex.getMessage());
    }
}