Towards a Generic Model for Web Services Mashups using the Spreadsheet Paradigm

Dat Dac Hoang

Dissertation submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy

School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia

September 2011

Supervisor: Dr. Hye-Young Paik
Originality Statement

“I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgment is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project’s design and conception or in style, presentation and linguistic expression is acknowledged.”

Signed:

Name: Dat Dac Hoang

Date: .........................
Copyright Statement

“I hereby grant the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all proprietary rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation. I also authorize University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstract International (this is applicable to doctoral theses only). I have either used no substantial portions of copyright material in my thesis or I have obtained permission to use copyright material; where permission has not been granted I have applied/will apply for a partial restriction of the digital copy of my thesis or dissertation.”

Signed: ............................
Date: ...............................
To my parents, my brother,
my wife Hang-Thu, and our son Quang-Minh,
who made all of this possible,
for their endless love, encouragement and patience
Acknowledgements

This dissertation would not have been possible without the supports of many people.

First and foremost, I wish to express my gratitude to my supervisor Dr. Hye-Young (Helen) Paik for her great support and guidance through my PhD candidature. I would like to thank her for accepting me to be one of her research students in the School of Computer Science and Engineering, the University of New South Wales. Working with Helen has been a continuous learning experience and I am grateful to her for invaluable advices and kind encouragements when I was lost. I thank her for trusting me to explore freely on my own when I found my direction. Without her support I would not have achieved what I have done in this dissertation.

I gratefully thank Prof. Boualem Benatallah for his guidance and support. Boualem is an excellent, passionate, world-class scientist. He introduced me to the research field and taught me how to do research.

I would like to acknowledge the Vietnamese Government (through Vietnamese Scholarship Overseas Program), the University of New South Wales, School of Computer Science and Engineering, Prof. Boualem Benatallah and Dr. Hye-Young Paik for providing me with the financial supports. I also would like to thank the School of Computer Science and Engineering for the great research environment and facilities they provide.

I also sincerely thank A/Prof. Fethi Rabhi for his kindly support in providing access to the ADAGE project. The presentations and discussions in the group meetings were really valuable for my research.

Over the years, I enjoyed to be a member of Service Oriented Computing research group and loved the conversations with people regarding my research. I would like to thank all members of the SOC group for the time we spent together, for their friendliness and support. In particular, I thank Dr. Regis Saint-Paul, Dr. Hamid
Motahari, Dr. Hakim Hacid, Dr. Sherif Sakr, Dr. Adnene Guabtni, Dr. Srikumar Venugopal, Dr. Van Munin Chhieng, Dr. Woralak Kongdenfha, Dr. Giusy Di Lorenzo, Dr. Ingo Weber, Seung Ryu, Van Hai Ho, Jae Choi, Hung Vu, Seyed Beheshti, Kwok Chung Yew, Kiet Vo, Vincent Zhao, and Sung Wook Kim. I would also like to thank students who worked with me in their theses: Mei Kee Hung and Jeffrey Dong.

My friends have made my time in Sydney memorable. I would like to thank all of them who keep me sane and make Sydney as my home for years.

My special thanks go to my parents and brother, who taught me to believe that I really could do anything I wanted to do. I am deeply and forever indebted to them for their endless love, support and encouragement. They are the source of my strength and without their countless support this dissertation would have never been started.

Last but not least, I would like to thank my loving wife and son for coping with my hectic schedule and providing me with love, patience and understanding. Without their love this dissertation would probably have never been completed. I would never forget their encouragements during my most difficult times. To you my little boy, Quang-Minh, I would like to say: “When you grow up and are able to read this, please accept my apology for being busy and not giving you the time you deserve to have”.
Abstract

The proliferation of Web services and online resources enabled a self-service application development method, namely “mashup”, that combines together existing data and services to create ad-hoc, idiosyncratic applications. Typically, mashup tools possess user-friendly interfaces with simple user interaction mechanisms that enable individuals rapidly build applications without any in-depth technical skill requirements. To realise the full benefit of mashups, there is a need to support the paradigm through an established platform that has a large user base.

Spreadsheets are popular productivity tools and are proven to be useful in manipulation, analysis, visualisation, and reporting of data. They have been also tagged as the most successful end-user development environment with millions of users.

In this dissertation, we address the issue of providing a generic mashup development framework, through an application of the spreadsheet paradigm. We propose a novel framework, named MashSheet, which allows users to create mashup applications using spreadsheet-like formulas.

The key innovation of MashSheet is a collection of operators that supports coordinating Web services, manipulating and visualising data created by the services. MashSheet applications are incrementally built and data in each intermediary step is only visualised when needed. It makes the mashup applications concise and easy to follow, as well as keeping their computation logic separate from presentation.

Although the issue of reusing already-built applications has been previously studied in mashups and other spreadsheet research areas, its potential has not been fully realised in the state-of-the-art spreadsheet-based mashup tools. To enable users to reuse existing mashup applications in MashSheet, we propose a reuse model which consists of mashup templates and a template repository. The design of mashup templates in MashSheet allows the users to stay in the spreadsheet paradigm they are familiar with when applying the reuse model.

The approach proposed in this dissertation have been implemented in a prototype
and validated via experiments in real-world scenarios. We also conduct studies of
the framework to evaluate its usability and expressiveness. The result shows that
(i) MashSheet is expressive and flexible to support numerous application scenarios;
and (ii) MashSheet reduces the application development time compared to one of
the state-of-the-art spreadsheet-based mashup tools.
Publications Arising from this Dissertation

Prototypes (Demonstration of MashSheet):

- [http://www.cse.unsw.edu.au/~hpaik/mashsheet](http://www.cse.unsw.edu.au/~hpaik/mashsheet)

Published:


Contents

1 Introduction 1

1.1 Key Research Issues . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.1.1 Lack of Generic SB Mashup Programming Model . . . . . . . 4

1.1.2 Lack of Model for Application Reuse in SB Mashups . . . . . 6

1.2 Contribution Overview . . . . . . . . . . . . . . . . . . . . . . . . . . 7

1.2.1 A Systematic Analysis of SB Mashups and the Tools . . . . . 7

Implementing Mashups in the Spreadsheet Paradigm . . . . . . . . 7

A Survey of the State-of-the-art SB Mashup Tools . . . . . . . . . 8

1.2.2 The MashSheet Framework . . . . . . . . . . . . . . . . . . . . 8

Generic Data Model . . . . . . . . . . . . . . . . . . . . . . . 8

Coordinating Web Services . . . . . . . . . . . . . . . . . . . . . . 9

Data Manipulation . . . . . . . . . . . . . . . . . . . . . . . . 9

Data Visualisation . . . . . . . . . . . . . . . . . . . . . . . . 10

1.2.3 Enabling Application Reuse in SB Mashups . . . . . . . . . . 10

Templates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

A Repository of Templates . . . . . . . . . . . . . . . . . . . . . 11

1.3 Dissertation Organisation . . . . . . . . . . . . . . . . . . . . . . . 11

2 Background 13
2.3.1 An Alternative Paradigm for Mashups ........................................ 35
2.3.2 Why Spreadsheets? ................................................................. 36
2.3.3 Motivating Scenario .............................................................. 37
2.3.4 Possible Ways of Enabling Mashups in Spreadsheets ................. 38
   Implicit Mashups ................................................................. 38
   Event-based Mashups ........................................................... 38
   Mashups using Shared Process Data ......................................... 39
   Explicit Mashups ................................................................. 40
   Discussion ......................................................................... 40
2.3.5 A Reference Architecture for SB Mashup Tools ......................... 41
2.4 Chapter Summary ................................................................. 43

3 An Analysis of the State-of-the-art SB Mashup Tools .......................... 44
3.1 Introduction ................................................................. 44
3.2 Related Work ................................................................. 45
3.3 Review Samples ............................................................... 48
3.4 Review Dimensions ........................................................... 50
   3.4.1 Component Model ...................................................... 50
   Component Data Model ...................................................... 50
   Component Access Model ................................................... 52
   Extensibility ................................................................. 52
   3.4.2 Composition Model ..................................................... 53
   Mashup Strategy ............................................................... 53
   Orchestration Style .......................................................... 54
   Data Passing Style ........................................................... 56
   3.4.3 Development Environment ............................................. 57
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>xii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Users</td>
<td>57</td>
</tr>
<tr>
<td>Search-ability</td>
<td>57</td>
</tr>
<tr>
<td>Community Features</td>
<td>58</td>
</tr>
<tr>
<td>Software Engineering Aspects</td>
<td>58</td>
</tr>
<tr>
<td>3.4.4 Runtime Environment</td>
<td>59</td>
</tr>
<tr>
<td>Mashup Engine</td>
<td>59</td>
</tr>
<tr>
<td>Execution Type</td>
<td>59</td>
</tr>
<tr>
<td>Exception and Transaction Handling</td>
<td>60</td>
</tr>
<tr>
<td>3.5 Analysis Results</td>
<td>60</td>
</tr>
<tr>
<td>3.6 Chapter Summary</td>
<td>65</td>
</tr>
</tbody>
</table>

<p>| 4 The MashSheet Framework | 66 |
| 4.1 Introduction | 66 |
| 4.2 Reference Scenario | 69 |
| 4.3 Overview of MashSheet | 71 |
| 4.3.1 MashSheet’s Graphical User Interface (GUI) | 71 |
| Formula Editor | 71 |
| Component Explorer | 71 |
| Component Repository Interface | 73 |
| Log | 73 |
| 4.3.2 Component Repository | 73 |
| 4.3.3 Mashup Engine | 73 |
| 4.3.4 Wrappers | 74 |
| 4.4 Application Model | 74 |
| 4.4.1 MashSheet Application | 74 |
| 4.4.2 MashSheet Formula | 75 |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Abstract</td>
<td>227</td>
</tr>
<tr>
<td>A.2 Introduction</td>
<td>227</td>
</tr>
<tr>
<td>A.3 MTDL Document Structure</td>
<td>228</td>
</tr>
<tr>
<td>A.3.1 The MTDL Root Element</td>
<td>228</td>
</tr>
<tr>
<td>A.3.2 MTDL Elements</td>
<td>229</td>
</tr>
<tr>
<td>A.3.3 MTDL Schema Definition</td>
<td>230</td>
</tr>
<tr>
<td>B User Study Material</td>
<td>232</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Overview of the MashSheet framework ........................................ 9
2.1 Service-oriented Architecture Reference Model ................................. 16
2.2 Mashup Example: the Housing Maps Application .............................. 23
2.3 A Reference Architecture for Mashups .......................................... 26
2.4 Estimation of Computer Users in America in 2012 ............................ 30
2.5 The Mary’s Scenario .................................................................. 37
2.6 Reference Architecture for Mashup Tools using Spreadsheet Paradigm 42
3.1 The Mashup Tools Landscape ....................................................... 47
3.2 Component Data Model .............................................................. 52
4.1 The Tom’s Scenario .................................................................. 70
4.2 The Architecture of MashSheet ..................................................... 72
4.3 MashSheet Application Model ...................................................... 74
4.4 Example of RL Data and its MOT Representation ............................ 78
4.5 Example of CL Data and its MOT Representation ............................ 78
4.6 The iff() Example .................................................................. 84
4.7 The sync() Example .................................................................. 86
4.8 Visualise RL Data by: (a) Column, (b) Row ................................. 100
4.9 A running example implementing Tom’s scenario. ......................... 105
4.10 Parallel Example with Input-output Dependency ................................ 108
5.1 The Tim’s Scenario .............................................................................. 120
5.2 A MashSheet Template Created from Currency-Conversion Application .... 122
5.3 MashSheet Template Repository’s Data Model .................................... 125
5.4 Template Publishing Process ............................................................... 128
5.5 Template Utilisation Process ............................................................... 129
5.6 A Running Example ............................................................................ 131
6.1 MashSheet’s Architecture .................................................................... 144
6.2 MashSheet GUI on Microsoft Excel ..................................................... 145
6.3 MashSheet Template Repository GUI .................................................. 147
6.4 MOT’s Implementation in MashSheet .................................................. 154
7.1 Implementation of the News Feed Reader Application in MashSheet .... 162
7.2 Implementation of the Housing Direction Application in MashSheet .... 167
7.3 A Case Study in Financial Market Analysis Application Domain ......... 169
7.4 Implementation of the Financial Market Analysis Application in MashSheet:
   Create Template ..................................................................................... 174
7.5 Implementation of the Financial Market Analysis Application in MashSheet:
   Use Template ......................................................................................... 175
7.6 Bioinformatic Application Scenario ..................................................... 176
7.7 AMICO:CALC Approach to Express the Simple Sequence Pattern ....... 182
7.8 AMICO:CALC Approach to Express the Parallel Split Pattern ............. 183
7.9 AMICO:CALC Approach to Express the Exclusive Choice Pattern ....... 184
7.10 AMICO:CALC Approach to Express the Simple Merge Pattern .......... 185
7.11 Completion Time for the First Task (Mean) ........................................ 191
7.12 Completion Time for the First Task (Median) ..................................... 191
7.13 Completion Time for the Second Task (Mean) ........................................... 192
7.14 Completion Time for the Second Task (Median) ...................................... 192
7.15 Completion Time for the Third Task ......................................................... 194
7.16 Completion Time of Three User Groups - Mean ...................................... 195
7.17 Completion Time of Three User Groups - Median .................................... 196
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Layout-based Orchestration in Husky</td>
<td>55</td>
</tr>
<tr>
<td>3.2</td>
<td>Event-based Orchestration in A1</td>
<td>56</td>
</tr>
<tr>
<td>3.3</td>
<td>Evaluation Results</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>MashSheet Operators’ Syntaxes</td>
<td>103</td>
</tr>
<tr>
<td>5.1</td>
<td>MashSheet Template Repository Operators’ Syntaxes</td>
<td>128</td>
</tr>
<tr>
<td>6.1</td>
<td>Products used for MashSheet Implementation in Microsoft Excel</td>
<td>143</td>
</tr>
<tr>
<td>6.2</td>
<td>Database Design for the Template Repository</td>
<td>148</td>
</tr>
<tr>
<td>6.3</td>
<td>Evaluation of Data Importation Approaches in Spreadsheet</td>
<td>151</td>
</tr>
<tr>
<td>6.4</td>
<td>Products recommended for MashSheet Implementation in OpenOffice Calc</td>
<td>157</td>
</tr>
<tr>
<td>6.5</td>
<td>Products recommended for MashSheet Implementation in Gnome Gnumeric</td>
<td>157</td>
</tr>
<tr>
<td>7.1</td>
<td>Patterns Supported in some SB Mashup Tools</td>
<td>186</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Over the last few years, the number of Web-based applications and the amount of information they offer have been increasing at a greater rate than the decade before [213, 137, 196]. Some of the contributing factors to this growth are the advent of Web 2.0 technologies, the Open Data and Open API movements [144] and standardisation efforts, through which many public and private organisations expose their data and applications. This allows many more users to exploit the data in an easy and inexpensive way [130].

In the context of the Web applications, the term “mashup” is used to denote the recently emerging application development method that combines together data, services and presentations of already existing sources to create new situational applications. Mashups – with their programming models, languages, and instruments – are innovative in that they allow users to shift from passive consumers to active contributors of the Web content. Even some sections of social commentators refer to mashup and associated Web 2.0 technologies as a democratising (i.e., power-sharing) tool that enables information sharing to empower the generic population to make salient and informed decisions [105, 187, 210].

Recently, the main trend in the mashup development is to provide an easier solution towards fast accessing, consuming and reproducing of data and applications. Companies, for one, realise that mashup can shorten sales cycles and open up new business models (e.g., pay-per-invocation of mashup application). Mashup users also expect to see tools and applications that blur the boundary between “work” (e.g.,
structured programming activities) and “play” (e.g., an intuitive tool that allows users to try different application scenarios easily), emphasising on their convenience.

There are many mashup tools published through academic research projects (e.g., DERI Pipe [13], CMU’s Marmite [214]) or commercial company experiments (e.g., Yahoo Pipes [80], Microsoft Popfly [14], IBM Damia [88]). Some follow a pipeline-based model where data is filtered through series of “pipes” not dissimilar to UNIX shell scripts. Others follow a workflow-based model where the language features support more sophisticated control flows (e.g., Google Mashup Editors [29], AMICO:CALC [166]).

However, a careful observation of the plethora of mashup tools available today can reveal the challenges that mashups still face as a new application development paradigm:

- Firstly, there is no agreed or widely accepted programming model or standard language for mashups as yet [130].

- Secondly, individual mashup tools are still very domain specific (e.g., purposely built for Map data, scientific data), no single “generic mashup tool” has claimed enough number of users that makes it popular across different data domains.

- Thirdly, many aspects of mashup are still in its infancy in that important issues like privacy, security, governance, administration, version management/reuse are not discussed in detail in the community [136].

Given that mashups are intended to be widely used by the everyday Web users, its enabling technologies should be based on a platform that is widely available and known by the generic population. In fact, we argue that the first and second problems above can be attributed to the fact that there is no common platform, other than perhaps, the Web itself, in mashup tools.

Spreadsheets are a popular platform for building productivity tools. They are useful in easy manipulation, analysis, visualisation, and reporting of data. The popularity of the spreadsheets comes from several factors. For example, (i) The
data models are flexible in presenting and formatting data; and (ii) The formula
languages are simple but effective in using spatial relationships that shield users
from low-level details of programming languages [140]. Spreadsheets are widely used
by many personal computer users (e.g., scientists, students, accountants, executives,
and programmers) and its user base is still increasing. According to Scaffidi et al.
[184], there will be 55 million spreadsheets’ users in America alone in 2012.

We believe this provides a good reason to consider spreadsheets as a platform on
which a generic mashup environment can be designed. In this dissertation, we pro-
pose a generic mashup framework based on the spreadsheet paradigm. In particular,
our framework provides the following:

- An extended spreadsheet data model which is designed to handle simple to
  complex data types. This enables the support of heterogeneous data sources
  existing on the Web,

- An extended spreadsheet formula language and a set of operator classes for
  accessing and coordinating the data sources, as well as manipulating and vi-
  sualising intermediate and resulting data sets,

- Spreadsheet templates for reusing already-built mashup applications.

Throughout the dissertation, we will show that our framework can support both
mashup programming models (i.e., pipeline-based and workflow-based) using the
well-known spreadsheet paradigm and a few added extensions. We will also show
that our framework can be applied to different data domains.

This chapter is organised as follows. In Section 1.1, we outline research problems.
Section 1.2 summarises our contributions. Finally, in Section 1.3, we present the
structure of this dissertation.

1.1 Key Research Issues

There have been many proposals for devising mashup programming models, which
contributed to the plethora of mashup tools published. Most of these tools are based
1.1. Key Research Issues

on the Web (e.g., the tool itself is a Web-based application that operates within a browser environment).

More recently, spreadsheets have gained attention as a potential mashup platform, primarily because of its large user base and data manipulation utilities. As a result, we have seen a few attempts from academia and industry to apply mashup concepts into spreadsheets \[63, 207, 143\]. We will discuss them in detail in later sections, but as we mentioned earlier, this dissertation is particularly focused on this category of mashup tools. We will call them \textit{SB mashups} (as in spreadsheet-based mashups) for easy reference and to distinguish them from the Web-based ones. In the following, we identify the key issues in SB mashups.

1.1.1 Lack of Generic SB Mashup Programming Model

SB mashups are primarily focused on coordinating data integration in a pipeline fashion. Take SalesForce StrikeIron [63] for instance. In this tool, a mashup application is implemented by following the steps similar to these: (i) a cell area is selected to which the output of a Web service is imported (e.g., postcode via a postcode search service), (ii) the service is invoked and the cells are populated with the output values, (iii) another cell area is selected for the next service (e.g., a service that returns the average income of a suburb identified by a postcode), (iv) a cell reference is made to link the first cell area as input to the next service, (v) the next service invoked and the second cell areas are populated with the output values. Pipeline-based models are simple to implement and easy to use. However, it is limited in the types of mashup application scenarios it can create.

Very few mashup tools provide a workflow/orchestration style mashup programming model where more sophisticated control and data flow logic is supported. Even if some claim to support workflow-based model (e.g., [166]), they are generally very awkward to use because they impose certain concepts (e.g., shared read/write space) that are not intuitively known in spreadsheets.

This hinders the generic applicability of SB mashups in various application domain scenarios. We argue that, for SB mashups to be more widely accepted as a common mashup platform, we need to create a generic programming model that can
support both pipeline-based and workflow-based styles, but still can be integrated seamlessly within the spreadsheet environment.

There are some challenges in developing such a mashup framework:

- **To extend conventional spreadsheet data model in order to enable mashups.** Spreadsheets lack of sophisticated type system \[\text{\cite{84}}\]. Conventional spreadsheets usually support simple data types, typically number, string and boolean. Essentially, a complex data type is not supported which means data with special type (e.g., XML data as an output of a Web service invocation) cannot be stored in a cell value. Clearly, there is a gap between the representations of data on the spreadsheets and data on the Web. There is a need to appropriately extend the conventional spreadsheet data model to bridge this gap.

- **To support both data and control flows in spreadsheets.** In fact, the application model of conventional spreadsheets has a limitation in coordinating external data accesses (e.g., invoking multiple Web services in certain order). The evaluation of spreadsheet’s formula is driven by data dependencies between cells. This data flow model allows users to define some mashup operations using natural spreadsheet programming metaphor (e.g., sequential step-by-step invocation of Web services by cell referencing). However, the semantics of some control flow patterns are not inherently supported in spreadsheets and none of the existing SB mashup tools address this issue (e.g., exclusive choice and synchronisation) sufficiently. Consequently, not only supporting the data flow, but also supporting a basic set of control flow patterns is important for SB mashups.

- **To enable effective manipulation and visualisation of complex data.** On one hand, data in spreadsheet are represented as simple data types (e.g., string, number) in cells. On the other hand, data used in mashups are often represented as complex types such as XML or HTML documents. Therefore, users would have to “flatten” the complex data into the two-dimensional grid structure of spreadsheets every time any data operation (e.g., sort or filter) or visualisation operation (e.g., map or chart) needs to be applied. This creates unnecessary and inefficient steps in a mashup creation process and ultimately
makes the spreadsheets quite messy (i.e., all intermediate data are visible in the spreadsheet) and error-prone. Providing operators that can manipulate and visualise the complex data “as-is” is also important for the usability of the framework. There is a need to extend the conventional spreadsheet formula language to provide such operators.

1.1.2 Lack of Model for Application Reuse in SB Mashups

Software reuse, which is generally known as a difficult problem, deals with the use of existing software or software knowledge to construct new software. In this sense, mashup is a solution to the generic reuse problem in software.

Of course, we can also discuss the reuse problem in mashups themselves. In the context of mashup, leveraging services’ meta-data and data/control flow building logics that are embedded in already-built mashup applications can significantly improve both the performance and user experience [92] of mashups.

However, the collective effort into SB mashup development has been mainly focused on providing a basic mashup development and enactment solutions (e.g., [166, 68, 207] and [143]). The problem of enabling software reuse in SB mashups has not been addressed sufficiently.

Frakes and Kang [121] identify the general issues known in software reuse as follows:

- Developing a model to describe how to reuse software (e.g., application system reuse, component reuse, object and function reuse),
- Designing a component repository for software reuse,
- Designing a process for systematic software reuse,
- Building an algorithm for component structuring and retrieval.

Amongst the issues above, this dissertation focuses on the first and the second issue in the context of SB mashups so that it may lead to a foundation on which more advanced mashup reuse system can be built.
To provide such an approach, there are some challenges as follows:

- **A model to define a reusable mashup application in spreadsheets**: The spreadsheet paradigm does not impose any constrain on the spatial layout of data so that users can freely organise data in their preferences. Therefore, if users want to reuse an existing mashup application, there is a need to have a mechanism to identify boundary of the application (i.e., input, output) that would clearly describe which parts of the application are intended for reuse.

- **A repository of mashup applications**: It is crucial to provision an approach to build an extensible and shared repository of already-built mashup applications. The relationships between applications, users and requirements (e.g., input and output constraints) must be managed. And the repository should provide users with a low cost (e.g., low learning curve) and rapid accessibility to the applications.

### 1.2 Contribution Overview

Our goal in this dissertation is to address the research problems outlined in the previous section. The main research contributions are the following:

#### 1.2.1 A Systematic Analysis of SB Mashups and the Tools

We believe that the spreadsheet environment has inherent advantages over other mashup paradigms due to its popularity and familiarity with the users. To understand SB mashup tools and related issues, we concentrate on the following tasks:

**Implementing Mashups in the Spreadsheet Paradigm**

We investigate four possible ways of implementing mashups in spreadsheets: implicit mashups, event-based mashups, mashups using shared process data, and explicit mashups [131]. We discuss the benefits and limitations of each approach in terms of providing a mashup tool to users with intuitiveness and comprehensiveness. We use
an example to demonstrate the benefit of enabling mashups in spreadsheet tools. In addition, we propose a reference architecture of SB mashups.

A Survey of the State-of-the-art SB Mashup Tools

We conduct a qualitative analysis of the state-of-the-art SB mashup tools [132]. We collect, install and evaluate fourteen samples according to thirteen evaluation dimensions. The dimensions are classified into four main categories: component model, composition model, development environment, and runtime environment. The analysis result could be useful for the researchers and developers in the area to understand the landscape of SB mashup tools, the strengths and weaknesses of the current approaches, and to develop ideas on creating more advanced SB mashup frameworks.

1.2.2 The MashSheet Framework

We propose our own SB mashup framework, named MashSheet, where spreadsheet users (e.g., office workers, professional accountants, and scientists) are provided with an opportunity to consume a wider range of data through accessing Web services and to produce new data through creating mashup applications [134]. As shown in Figure 1.1, the MashSheet framework is an extension to a conventional spreadsheet program, enabling access, manipulation and integration of data from the Web. It is noted that in MashSheet, the data outside of the framework is accessed through Web services. A Web service in our framework has three types: SOAP-based, RESTful and data feeds such as RSS. We use the term Web services to refer to all three types and by design, our Web service can be extended to support different data sources.

Through the design and implementation of MashSheet, we make the following contributions in the area of SB mashups:

Generic Data Model

To make the framework generic, we first propose a generic data model that supports a wide-range of data types. We name the data type ‘MashSheet Object Type’
1.2. Contribution Overview

Figure 1.1: Overview of the MashSheet framework

(MOT) and it is an XML-based data model. By using MOT, we introduce two components to represent a Web service and Web service output.

Coordinating Web Services

We introduce an extension to the conventional spreadsheet application and its formula language so that users can easily orchestrate Web services. To orchestrate Web services, MashSheet users can either use (i) spatial arrangement of the cell formulas as the service invocation order; (ii) cell referencing mechanism; or (iii) explicit process modeling constructs (e.g., split or merge). The spatial arrangement of cells is considered first and the constructs are considered last.

Data Manipulation

We provide ten data manipulation operators in MashSheet. The innovation of these operators compared to the data mashup operators in other SB mashup tools are:

- The input parameters of the MashSheet data operators are MOT data. In some SB mashup tools (e.g., [166, 63]), Web data must be flattened into the grid structure of cells before users can run a data operator on the data. On the contrary, MashSheet data operators can be run directly on MOT. This means a series of operators can be applied to the data set without having to lay it out on the grid every time, and all intermediary results are stored individually in cells as MOT. It also allows the intermediary results to be reused by other
data and visualisation operators at any stage. This increases the reusability of the intermediary results during the application building steps.

The operators can perform operations either on the data structure (e.g., `merge()` two fields) or on the data value itself (e.g., `filter()` and `sort()`). It is also possible to transform from MOT to simple data type (e.g., `extract()`),

- In conventional spreadsheet or other SB mashup tools (e.g., [166, 63, 207]), the output data of an operation is immediately visualised in cells. Conversely, in MashSheet, the output of a mashup operator is held in a cell for further processing by other data or visualisation operators. This makes the mashup applications concise and easy to follow.

Data Visualisation

We define visualisation operators in MashSheet as components to present service output data with three different visualisation types (e.g., grid, chart, and map). The goal of these operators is to separate the presentation layer from data layer so that users can visualise Web data using different layouts. The separation is important, especially when users want the flexibility and reusability in creating their own display of the Web data. The benefits of having these operators include:

- **Flexibility**: the layout of data is automatically updated when there is a change in the data source,

- **Generic applicability**: we can apply different visualisation operators to produce different layouts of data, and

- **Ease of use**: the operators can be called like spreadsheet functions.

1.2.3 Enabling Application Reuse in SB Mashups

We propose a reference model facilitating the reuse of already-built mashup applications in spreadsheets. In particular, we conceptually define a reusable application and the process involved in reusing it. We have implemented this proposal in MashSheet.
1.3. Dissertation Organisation

Templates

We define the term “template” as a mean to facilitate reuse in SB mashup applications. A template is created by defining meta-data relating to a mashup application (e.g., creators, input, and output) and including all formulas of a MashSheet application. A template is determined as a function without referring to a particular data set. Therefore, it acts as a document to describe the functionality of a MashSheet application without reference to a particular invocation instance (i.e., a template is used to keep the “code” of a MashSheet application to be used with different input data). We propose the MashSheet Template Definition Language (i.e., MTDL), an XML-based language, for specifying MashSheet templates.

A Repository of Templates

We design a repository of reusable mashup applications with five operators. The operators have spreadsheet-like syntaxes and users can easily run the operators to manipulate with already-built applications in the repository.

1.3 Dissertation Organisation

The remainder of this dissertation is organised as follows:

- In Chapter 2 we present some background concepts and terms used in this dissertation. In particular, we review approaches and technologies for Web services, mashups and spreadsheet development. We also identify four possible ways of enabling mashups in spreadsheet environment, highlighting their significances and limitations. Finally, we propose a general architecture of mashup tools using spreadsheet paradigm in this chapter.

- In Chapter 3 we present the state-of-the-art in the problem of devising programming models for mashups. We survey some mashup tools and position the scope of this dissertation in the research field. We also conduct a qualitative analysis on the state-of-the-art SB mashup tools. The analysis result is
used to guide our research in developing a lightweight generic purpose mashup framework using the spreadsheet paradigm.

- In Chapter 4 we describe the MashSheet framework. We first motivate the need of having MashSheet through a reference scenario. We, then, overview the architecture of MashSheet and describe its application model. We present syntaxes and semantics of the MashSheet operators and illustrate each operator through examples. We also have a running example of how to build and execute the MashSheet application in the reference scenario.

- In Chapter 5 we focus on the application reuse issue in SB mashup frameworks. We define a model to describe reusable mashup applications in SB mashup tools. We also design a repository of applications and identify the process of creating and using applications in the repository.

- In Chapter 6 we present the implementation issues of MashSheet in one of the most popular spreadsheet programs - Microsoft Excel. We also discuss the possibility of implementing MashSheet in other spreadsheet tools (e.g., OpenOffice Calc and Gnome Gnumeric).

- In Chapter 7 we present the three-way evaluation of the MashSheet framework: case study, expressiveness review and a user study. In the case study, real-world examples are used to demonstrate and analyse the practical use of MashSheet. In expressiveness review, we evaluate the capability of MashSheet to model the basic control flow patterns in its applications. The user study is conducted to evaluate the usability of the framework.

- Finally, in Chapter 8 we conclude and present the future work of this dissertation.
Chapter 2

Background

In this chapter, we present an introduction to the research fields of Service-oriented Computing and spreadsheet development. We overview some terms, major techniques, research prototypes, software products and standards for mashups and spreadsheet, to help readers gain a better understanding of the work described in this dissertation.

This chapter is organised as follows. In Section 2.1 we first introduce basic concepts and terminologies related to the Service-oriented Computing field of study. In Section 2.2 we overview the spreadsheet research area with some notions, terminologies and definitions. Then, in Section 2.3 we investigate the ways of implementing mashups in the spreadsheet paradigm. Finally, we summarise this chapter in Section 2.4.

2.1 Service-oriented Computing

Service-oriented computing (SOC) is an umbrella term used to refer to all aspects of the new service-oriented paradigm in distributed computing. It promotes the concept of services as the level of abstraction in application integrations. Composing application components as services is a way to build networks of flexible, dynamic and agile applications, achieving large-scale application integrations in a rapid and low-cost manner. This paradigm is expected to be dominant in the next generation of application developments [189]. In this section, we introduce the key concepts,
abstractions and technologies of SOC that are relevant to our topic: Service-oriented Architecture, service composition, Web service, and mashup.

2.1.1 Service-oriented Architecture

According to Alonso et al. [87], the improvements in computer hardware and networks have led to the evolution of information systems. There are four architectures of information systems representing for this evolutionary process: 1-tier, 2-tier, 3-tier, and n-tier architecture. Single tier (or 1-tier) architecture is used when mainframes were used as the main computer architecture. Client/server (or 2-tier) architecture developed when local area network appeared. Three-tier architecture introduces a new layer, namely middleware, between the client and server which promotes a higher form of information system. Multi tier (or n-tier) architecture is a result of applying three-tier model in its full generality.

Service oriented architecture (SOA) is one more step in this evolutionary process and can be considered as a response to the problems that cannot be easily solved with three-tier and multi-tier architecture (e.g., the requirements of loosely coupled, standards-based and protocol-independent distributed computing).

Papazoglou [173] defined SOA as follows:

“Service-oriented architecture is a logical way of designing a software system to provide services to either end-user applications or to other services distributed in a network, via published and discoverable interfaces.”

This definition emphasises on three main characteristics of SOA: software is designed as services, their interfaces are described and published, and they are consumed over a network.

Singh and Huhns [189] listed other characteristics of SOA as follows:

- **Loose-coupling.** Services maintain a relationship that minimises dependencies and only require that they retain an awareness of each other;

- **Abstraction.** Beyond what is described in the service contract, services hide logic from the outside world;
2.1. Service-oriented Computing

- **Flexible.** The services are bound to each other late in the process and can be changed dynamically;

- **Reusability.** Logic is divided into services with the intention of promoting reuse;

- **Long life-time.** Services must exist long enough to be discovered, to be relied upon and to engender trust in their behavior; and

- **Granularity.** The participants in an SOA should be understood at a coarse granularity in order to reduce dependencies and communications

According to SOA Reference model defined by OASIS\(^1\), there are three main roles in SOA\(^93\) as depicted in Figure 2.1.1:

- **Service provider**, who provides software as services to satisfy specific needs;

- **Service consumer**, who seeks for a service that fits its needs and requirements; and

- **Service broker**, who provides a searchable repository or service registry for service descriptions, where service providers can register their services and service consumers can find those services and binding information to invoke the services

SOA can be implemented by using a wide range of technologies\(^174\). SOAP\(^2\), RPC\(^3\), REST\(^4\), DCOM\(^5\), and CORBA\(^6\), just to name a few.

2.1.2 Service Composition

In the SOC paradigm, developers can solve complicated problems by combining available services and orchestrating them to best suit their problem requirements.

---

\(^1\)Organisation for the Advancement of Structured Information Standards

\(^2\)Simple Object Access Protocol

\(^3\)Remote Procedure Call

\(^4\)Representational State Transfer

\(^5\)Distributed Component Object Model

\(^6\)Common Object Request Broker Architecture
This capability, namely service composition, is considered as one of the top SOC’s native capabilities. Undoubtedly, service composition accelerates application development, service reuse, and complicated service consummation [160].

According to Yang et al. [215], service composition could be static (i.e., service components interact with each other in a pre-negotiated manner) or dynamic (i.e., they discover each other and negotiate on the fly). Basically, it defines the following two aspects:

- What is the order in which the services are invoked? What are the conditions under which a certain service may or may not be invoked? (i.e., control flow),
- How and what does the service exchange data with another? (i.e., data flow),

Control flow composition concentrates on behavioral aggregation of services. It defines the logical dependencies between independent services by specifying an invocation order. Data flow composition is the favourite model of resource oriented architecture [171] and concentrates mainly on data aggregation. It defines the rules for transferring data items between different invocations in data-centric model [86]. On one hand, control flow composition includes process oriented patterns [199] such as sequential, parallel, or alternative control constructs. Data flow composition, on the other hand, deals primarily with matching, filtering, aggregating data from
different sources.

We can also consider two different approaches to service composition depending on the target users of the composition tool: traditional *developer-centric* and *consumer-centric* [151]. In the developer-centric composition, applications can be built through the composition of distributed services by using a service composition “programming” language, such as WS-BPEL⁷ [75], BPML⁸ [7], WSFL⁹ [148], BPSS¹⁰ [8], or WSCI¹¹ [204]. The common characteristic of these languages is that they are mainly designed for professional developers. Although these languages are powerful, the skill barrier is too high for everyday Web users to adopt this service composition approach. The developers need to master SOA technologies (e.g., XML [201], WSDL [206], SOAP [202], UDDI [203], WS-BPEL [75]) and tools (e.g., integrated development environment, runtime environment) in order to create a composite service application.

One of the impacts of Web 2.0 and the whole idea of “users as active Web content creators”, is the rise of consumer-centric composition approach, or sometimes called “lightweight service compositions”. In this approach, the focus is more on encouraging and supporting the consumers of the services to express their creativity to develop applications to fit their needs. Hence, the approach comes with many ideas to support low-level technical users (e.g., highly graphical programming environment) to easily build composite services. Mashup is the term that we use to refer to this lightweight approach.

### 2.1.3 Web Service

As a defining concept, a Web service is the crux of SOA. There exists several definitions for a Web service:

The W3C¹² [68] defined a Web service as follows [205]:

---

⁷Web Services Business Process Execution Language  
⁸Business Process Modeling Language  
⁹Web Services Flow Language  
¹⁰Business Process Specification Schema  
¹¹Web Service Choreography Interface  
¹²The World Wide Web Consortium
“Web services provide standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. Web services are characterised by their great interoperability and extensibility, as well as their machine-processable descriptions thanks to the use of XML. They can be combined in a loosely coupled way in order to achieve complex operations. Programs providing simple services can interact with each other in order to deliver sophisticated added-value services.”

This definition highlights all the important characteristics of Web services: platform-independent, self-contained, self-describing, and composable modular units that can be published, located, and invoked over the Web.

In [173], Papazoglou defined a Web service as follows:

“A Web service is a self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or application.”

This definition emphasises on the aspect of using Web services as building blocks in SOA. Users or applications consume software as Web services to complete their tasks.

In this dissertation, we broadly consider it as a software built according to well-known standards and is accessible to other applications over the Web [87], without contradicting the academic definitions. In particular, depending on the standards being applied to building a service, we consider the following three types of Web services.

**SOAP-based Web Services**

Many Web services are based on an architecture having three elements: the service requester, the service provider and the service registry [87]. This architecture provides necessary infrastructure to implement Web services: a way to communicate (SOAP [202]), a way to describe services (WSDL [206]), and a name and directory
server (UDDI [203]). We call a Web service that is developed and operated by using this architecture as a “SOAP-based Web service”.

SOAP-based Web services communicate to each others by using SOAP messages. In the following, we show an example of a simple request-response SOAP service for stock quote using the HTTP protocol [191]. The SOAP request message is as follows:

```
GET /StockPrice HTTP/1.1
Host: example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
               xmlns:s="http://www.example.org/stock-service">
  <env:Body>
    <s:GetStockQuote>
      <s:TickerSymbol>IBM</s:TickerSymbol>
    </s:GetStockQuote>
  </env:Body>
</env:Envelope>
```

The SOAP response message is as follows:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
               xmlns:s="http://www.example.org/stock-service">
  <env:Body>
    <s:GetStockQuoteResponse>
      <s:StockPrice>45.25</s:StockPrice>
    </s:GetStockQuoteResponse>
  </env:Body>
</env:Envelope>
```

**RESTful Web Services**

Representational State Transfer (REST) is an architectural style for building distributed hypermedia systems. It was first described by Fielding in his PhD thesis [117]. Fielding explained the meaning of REST as follows:

“Representational State Transfer is intended to evoke an image of how a well-designed Web application behaves: a network of Web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use.”
The most basic concept of RESTful Web service is a resource since services are used to manage resources and its states. A RESTful Web service accesses resources through HTTP protocol. A resource is defined by its identity, state, and behaviour. It can also have many representations, although we mostly consider XML-based representations in service composition scenarios. A resource is identified by a URI$^{13}$ and is manipulated by using four basic operations: Create, Read, Update, and Delete. In many situations, the resources in a RESTful Web service can be considered as “data exposed over the Web with uniformly structured access interfaces”.

Costello listed the characteristics of a RESTful Web service as: client-server, stateless, cached, uniform interface, and layered system $^{107}$. A RESTful Web service consists of three artefacts: a URI (i.e., to identify the service), data type supported (e.g., JSON$^{14}$ $^{39}$, XML, YAML$^{15}$ $^{67}$), and a set of operations (e.g., POST, GET, PUT and DELETE) $^{53}$. While RESTful services might use XML in their responses (e.g., as one way of organising structured data), REST requests rarely use XML. For example, the stock quote service (as exemplified in the SOAP-based Web services) can be written as a RESTful Web service $^{191}$. The URI for this service is as follows:

http://example.org/StockPrice?TickerSymbol=IBM

The REST request message is as follows:

GET /StockPrice/IBM HTTP/1.1
Host: example.org
Accept: text/xml
Accept-Charset: utf-8

The REST response message is as follows:

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<s:Quote xmlns:s="http://example.org/stock-service">
  <s:TickerSymbol>IBM</s:TickerSymbol>
  <s:StockPrice>45.25</s:StockPrice>
</s:Quote>

$^{13}$Uniform Resource Identifiers
$^{14}$JavaScript Object Notation
$^{15}$Yet Another Markup Language
RSS/Atom Data Feeds

Content syndication is a concept denotes the mechanism to structure and publish Web resource content in an agreed format that independent of any visual layout [145]. According to Ogrinz [167], Really Simple Syndication (RSS) and Atom are content syndication formats for publishing Web contents. An application aggregates multiple feeds so that users can view the content of various Web pages in an unique environment is called “Feed reader”. Both RSS and Atom are specified by using XML. For example, an extract of a Google News feeds is as follows:

```xml
<rss version="2.0">
  <channel>
    <title>Top Stories - Google News</title>
    <link>http://news.google.com/news?pz=1&ned=au&amp;hl=en</link>
    <language>en</language>
    ...<item>
      <title>PM's Abbott solution falters as Julia Gillard rises in Newspoll—The Australian</title>
      <link>http://news.google.com/news/url?sa=t&amp;fd...</link>
      <description><table>...</table></description>
    </item>
  </channel>
</rss>
```

- `<rss version="2.0">` is the declaration tag;
- `<channel>` is used to define channel elements:
  - `<title>` is the name of the channel;
  - `<link>` is an HTML link to the channel Web site;
  - `<language>` is the encoding language of the channel
  - `<description>` is a short description of the channel;

To read the feed in a human-readable way, users use a RSS feeds reader and paste the URL of the RSS channel.

### 2.1.4 Mashup

The term “mashup” originates from musical domain where artists mix several music tracks of usually different genres into one record [144]. Like music mashup, mashup
in service composition domain denotes the reuse and mixing of existing resources on the Web, facilitating rapid, on-demand application development.

Definition

There is no formal, widely-agreed definition of mashup in the literature. Koschmider et al. defined mashup as “a Web-based application that is created by combining data, presentation or functionality from online third party resources [144]”. Although, we argue that mashup is a framework technology that enables mashup applications, hence not a Web-based application by itself, Koschmider’s definition emphasises an important characteristic of mashup that the content to be integrated is not only data but also functionality (i.e., programming logic) and layout of components (i.e., presentations).

Example

A classical example of mashup available in literature is the “Housing Map” application. The application considers a scenario where users want to combine property listings from Craigslist [12] with a map from Google [28] to assist themselves in house searching. Figure 2.2 shows an implementation of the Housing Map application on the Web.\footnote{The figure is created by running the application in \[32\].}

This application exemplifies the definition of mashup where data, functionalities and layouts of two Web resources (i.e., RSS feed of Craigslist and Map API of Google) are combined to create a new application.

Types

Koschmider et al. [144] classified mashup applications based on four questions as follows:

- What to mashup?
- Where to mashup?
- How to mashup?
- When to mashup?
Firstly, mashup applications can be classified by the content of the integration: *data*, *process*, and *presentation*. Data mashup allows users to retrieve data from one or several data sources, to process and mix the data, and to publish the result either as a feed or another data source. Process mashup allows users to automate processes by coordinating services. Presentation mashup allows the user to change the user interface of Web sites by removing elements or adding additional components. Schmelzer highlights the differences between data and process mashup as follows. On one hand, data mashup is a class of ad-hoc, situational applications that meet the short-term needs of individuals. It is unlikely to provide a long term value to users and use the design concept of “to use it is to create it”. Process mashup, on the other hand, supports flexible business processes with a higher level of repeatability (i.e., it provides organisations with higher level of automation).
Secondly, depending on the location where the integration take place, there are two types of mashup applications: client side and server-side [170]. Client-side mashup applications combine services on the client (e.g., a Web browser). Server-side mashup applications compose several services on a server. In this case, the server plays a role as a proxy between a mashup application on the client and resources in other Web sites that take part in the mashup application. Client-side mashup applications can be also further classified by the environments where the mashup take place (e.g., browser-based and desktop-based (e.g., spreadsheets)).

Thirdly, depending on the modality of the integrating resources, there are extraction and flow mashup applications. In extraction mashup, the application acts as a wrapper to collect and merge data resources. In flow mashup, data are transformed and combined to form the application [144].

Fourthly, mashup applications can be categorised into consumer and enterprise. Consumer mashup applications are designed for public use, expose APIs and feeds in a public Web site. Enterprise mashup applications merge resources and applications of several systems in an enterprise environment [167].

Characteristics

The following are common characteristics of mashup applications:

- **User-centric.** Users are not expected to have good technical knowledge or programming skills in order to create and run mashup applications. Also, mashup applications are normally created with rich user interfaces (e.g., map or chart) and simple user interaction styles (e.g., point and click, drag and drop).

- **Situational.** Mashup applications often provide solutions for situational contexts. Indeed, ad hoc and idiosyncratic applications are created for satisfying the need of individuals or narrow groups of users.

- **Lightweight.** Mashup application is SaaS\(^\text{17}\) [98] in the sense that it requires no installation or update in order to run [129]. It often utilises techniques such
as copy-and-paste snippets of Javascript [69], use feeds and XML to connect various components together, or embed external API (e.g., Google Map).

- **Rapid development.** Mashup applications have short time to market: usually measured in hours or days rather than weeks or months as in traditional application development [101].

- **Reuse.** The idea of reuse is an important and integral part of mashup initiation. Mashup applications benefit from the “building on the shoulders of giants” philosophy that enables rapid development. Also, the code of mashup applications are often opened to share amongst other users.

- **Leveraging the user community.** The user community is essential for mashup application development. It allows users to share applications to other members, to collaborate for developing applications, to exchange idea, and to discuss and comment on various issues.

**Reference Architecture**

There are some research works proposing architectures for mashups [87]. In this section, we present the architecture introduced in [219], as it encapsulates many concepts that are necessary in a generic mashup framework. According to Yu et al. [219], a mashup framework is characterised by the following four elements: component model, composition model, development environment and runtime environment. The component model determines building blocks for mashup applications (e.g., resource type and interface). The composition model identifies how components are integrated (e.g., orchestration style and data passing style). The development environment is the mashup development tool where users write “code” for mashup applications. The runtime environment is the place where the executions of mashup applications take place and the execution results are visible to the users. Figure 2.3 illustrates the mashup reference architecture.
Alternative Architecture

There are, however, some limitations of the reference architecture proposed in [219]. For example, the development environment and runtime environment are not necessarily separated. However, in practice, the development environment and runtime environment of most mashup tools are integrated. This employs the idea that a user does not always need to develop something before using the run time environment, and the distinct separation of development and deployment increases the learning curve for the novice mashup users. Moreover, the combined environment and the result of the coding in-progress (i.e., immediate feedback) allow the users to develop an application gradually and incrementally.
Tools

There are many tools available on mashups and each of them takes on different approaches to building mashup applications. We reserve the introduction and survey of mashup tools to Chapter 3 of this dissertation.

2.1.5 Workflow management systems

Workflow management systems (WFM) are rule-based management software that coordinate and monitor executions of multiple tasks to perform complex organizational functions [124]. A workflow is a process of tasks, usually expressed in terms of activities and their dependencies. There exist several types of workflow (e.g., business workflow, scientific workflow). Business workflows are used to automate human or computer agents tasks within an organisation [218]. Scientific workflows are utilised by scientists to solve scientific problems [135]. Many of the modern scientific workflow tools are also considered for potential mashup environments.

The models of workflow management systems mostly fall into one of two classes: control flows or data flows [111]. The two classes are similar in that they specify the interaction between individual activities within the group that comprise the workflow, but they differ in their methods of implementing that interaction. In control flow workflows, the connectivities between tasks represent a transfer of control from the preceding task to succeeding one. This includes control structures such as sequence, conditional, and iteration. Data flow workflows are designed to support data-driven applications. The connectivities between tasks represent the flow of data transferring between them [194].

In the following we introduce some scientific workflow management systems:

- **Kepler**\(^\text{18}\) [154] is a workflow tool allowing scientists to design and execute scientific workflows. The predecessor of Kepler is Ptolemy II system\(^\text{19}\) which is developed for composing concurrent systems. Kepler contains a number of predefined components (called actors) for different purposes, such as query

\(^{18}\)http://kepler-project.org/

\(^{19}\)http://ptolemy.eecs.berkeley.edu/
and transform data, execute application and web service, or send and receive email. Users select actors and put them into the design canvas to create a workflow. The connection between actors are defined in the pipeline fashion (i.e., output port of one actor links with the input port of another actor) [154].

- **Taverna** [168], a part of myGrid\(^{20}\) project, is a scientific workflow management system. Users develop Taverna workflows by using a workbench environment. There are two Taverna workbenches: stand-alone and add-on plug-in which is integrated with the “myExperiment research environment”\(^{21}\). In the myExperiment environment users can create, run, save and share their workflows with other users. Taverna’s workflows are modeled by using an XML-like language named SCUFL (Simple Conceptual Unified Flow Language) [169]. Execution of Taverna’s workflows is relied on the FreeFluo\(^{24}\) enactment engine. A Taverna workflow comprises of three components: processor, link and coordination constraints. A processor component transforms data from input ports into a set of data in output ports. The interaction between processors is designed in two ways: link and coordination constraints. The link component defines the data flow between processors in pipeline fashion. The coordination constraints define the control flow between processors [168].

- **Triana** [102] is a workflow environment for data analysis developed by Cardiff University\(^{22}\). Similar to Kepler and Taverna, Triana identifies workflow components (called units) by their input ports, output ports and names. A unit is written in a Java class and interactions between units (called cables) are defined in both data flow and control flow styles. Triana also allows users to define their own components by using provided wrapping code [102].

## 2.2 Spreadsheet

In this section we briefly introduce key concepts, abstractions, characteristics of spreadsheet. Spreadsheets are applications that contain a collection of cells organ-

\(^{20}\)http://www.mygrid.org.uk

\(^{21}\)http://www.myexperiment.org

\(^{22}\)http://www.cardiff.ac.uk/
ised in a tabular grid. Each cell is identified by its coordinates and contains either empty or atomic type value. Spreadsheets make it easy to display information, and people can insert formulas to work with the data in the cells.

At the beginning, spreadsheets have been designed as a multi-purpose productivity tool (e.g., solving calculation problem, or aiding personal decision making). Now, spreadsheets are also an application development platform on which many small-scale desktop applications are built. For example, many business data management applications such as customer orders, inventory controls or accounting are developed as spreadsheet applications. Kandel et al. use spreadsheets to manage photo images and their annotations. Spreadsheets are often touted as the most successful End-user Development environments.

Spreadsheets come in a variety of implementations and bear varied sets of features. However, in this section we introduce the traditional model of spreadsheets representing the vast majority of commercial spreadsheet products.

### 2.2.1 Users

According to Scaffidi et al., there will be 90 million people using computers at work in America in 2012. Among the 90 million people, they anticipate that there will be 55 million spreadsheet or database users (potentially they will do some level of programming), 13 million who describe themselves as programmers, and 3 millions professional programmers. Figure illustrates this estimation and shows that spreadsheet is a popular productivity tool now and probably still will be in the foreseeable future.

### 2.2.2 Programs

VisiCalc was the first spreadsheet program that appeared in 1979. Many different spreadsheet programs were then introduced, including SuperCalc, Corel Quattro Pro, PlanMaker, Quantrix, Lotus 1-2-3, and Apple Numbers. By now the dominating spreadsheet program is Microsoft Excel. Open-source spreadsheet programs include Gnumeric, OpenOffice Calc, and
2.2. Spreadsheet

Several online spreadsheet programs exist, such as Google spreadsheet [26], EditGrid [18], wikiCalc [77], iRows [37], and NumSum [49]. Research prototype spreadsheets are abundant, such as Forms/3 [96], Query spreadsheet [136], Object-oriented spreadsheet [175, 103], Visualisation spreadsheet [138, 101], Haxcel [149], Functional spreadsheet [109, 110], PhotoSpread [141], and Spreadsheet as DBMS\textsuperscript{23} [211, 197, 212].

2.2.3 Model

The traditional spreadsheet model is based on a grid, also called a worksheet, where cells are identified by their coordinates, denoted using letters for the columns and numbers for the rows. For example, D5 is the cell located in the fourth column of the fifth row. Each cell contains either a single atomic value or no value at all, in which case it is said to be empty. The cell value is obtained either from a direct user input (e.g. user may set A1 = 4) or from a formula expression (e.g. user may set A2 = SQRT(D5) * 2). Formulas are expressions that combine:

\textsuperscript{23}Database Management System
2.2. Spreadsheet

- Functions (e.g. function $\text{SQRT}(x)$ returns the square root of $x$),
- Constant expressions (e.g. 2 in this example), and
- Cell references denoted as cell coordinates. A reference can be relative (e.g. D5 in this example), or absolute (e.g., $\$D\$5$) or a mixture (e.g., $\$D5$ which is column-absolute but row relative).

Two cells are said to have dependency when there is a cell reference between cells’ formulas. Cell dependencies in spreadsheet define two aspects: evaluation order and data flow. The evaluation order determines the order in which formulas in cells are evaluated and the condition under which a certain formula may or may not be evaluated. Data flow defines how the data are passed between cells’ formulas.

When the content (i.e., value or formula) of a cell in spreadsheet is changed, all cells having dependency with that cell must be recalculated. The order of the recalculation is determined by a dependency graph among cells. For example, if cell A2 depends on cell D5 then the evaluation of D5 should be completed before the evaluation of A2 is completed.

There exists two views in a spreadsheet application: formula view and evaluation view. The formula view represents development environment where users can edit the content of a cell (e.g., enter formula in cell, or edit cell value). The evaluation view represents execution environment where all cells’ formulas are evaluated to values.

2.2.4 Characteristics

Spreadsheets provide simple, intuitive and easy to use environments that are suitable for end-users. The following are characteristics of the spreadsheet paradigm:

- **Dynamically typed.** Data in spreadsheet are dynamically typed [186]. This means that the type of data in cells are not required to determine in advance. For example, if cell D1 contains the formula $=\text{IF}(A1 > 0; 42; D1)$ then the value of D1 could be a number (i.e., 42) or the circular reference to its value when the value in cell A1 > 0.
• **Mixture of interaction styles.** As Nardi [162] noted, spreadsheets mix a textual language with charts and graphs, offering a mixture of command line and graphical interaction styles;

• **Resilient and forgiving.** An error in a cell only affects referring cells and does not cause the program to crash [142];

• **Lazy evaluation.** Spreadsheet programs are “lazy” in the sense that an intermediate expression is evaluated only when there is a demand for it, and its value is then cached so that subsequent demands will use that value [165];

• **Incremental development.** A spreadsheet program intermingles program codes with the data it is applied to and supports incremental development. Intermingling program codes with data allows users to examine cells formula, to understand how it works and to build a new program based on existing code. Incremental development means that users can write program step by step, enter each step in a cell and visualise the output at each step [142]; and

• **Immediate feedback.** Spreadsheets exhibit an interesting auto-updating feature: as soon as the value of a cell in the spreadsheet changes, such a modification triggers the automatic update of the referred cells. As a consequence, in a chain of dependent cells, the update of a cell propagates to all cells in the chain [142].

### 2.2.5 Limitations

Apart from the advantages, spreadsheets also have the following limitations:

• **Limited data type.** Spreadsheets lack sophisticated type system [84]. Conventional spreadsheets usually support simple data types, typically number, text and Boolean. Essentially, complex data type is not supported so that data with special type (e.g., visualisation) cannot be stored in a cell value. In addition, it is not possible to store more than one value in a spreadsheet cell.

• **Abstraction.** Spreadsheets expose limitation in abstraction capabilities hampering the expressiveness from procedural abstraction, data abstraction and
exception handling [96].

- **Language.** The conventional spreadsheet language is restricted. For example, it does not allow recursion. The repetition is supported in a limited way (e.g., through replication of the same formula across rows and columns).

- **Error-proneness.** Some studies have shown that existing spreadsheets contain errors at a high rate [172, 176, 85]. Rajalingham have et al. classified spreadsheet errors into different categories [179]. For example, system-generated, user-generated, quantitative, accidental, developer-committed, omission, alteration, and duplication, just to name a few.

## 2.2.6 Extensibility

Several approaches have been made to extend the spreadsheet model in various directions. In this subsection, we review three ways to extend the Microsoft Excel spreadsheet:

- **Visual Basic for Application.** Developers can extend the Microsoft Excel spreadsheet program by using an implementation of Visual Basic language on Microsoft Office products named Visual Basic for Application (VBA). By using VBA, developers can create user defined functions to modify user interface, automate process, or access functionality of the application. The advantages of VBA are as follows:
  
  - It is a natural scripting language of Microsoft Office products; and
  - It is easy to use, and it provides full application level access to Microsoft Office’s programs

The limitations of VBA are as follows:

- It has poor syntax and functionality (i.e., not a first order language, only a subset of Visual Basic language); and
- It is not integrated in .Net framework
2.2. Spreadsheet

- **COM\(^{24}\)/ Automation add-ins** \[^{19}\]. Developers can also use COM and Automation add-ins to extend the functionality of the Microsoft Office program. They are used to automate Excel in response to user events (e.g., click a button) and system events (e.g., open a worksheet, enter a formula in a cell). The main advantages of COM/ Automation add-in are as follows:
  - They are compatible with .Net framework; and
  - They provide application level extensibility to Microsoft Office products

The main limitation is that they do not support some latest features of the program (e.g., TaskPane in Microsoft Excel 2003).

- **Visual Studio Tools for Office (VSTO) Add-in** \[^{94}\]. VSTO provides means for developers to extend the Microsoft Office 2003 and later versions products. It allows to isolate the presentation elements of a spreadsheet from the data it contains by storing them as separate, well structured XML documents. The main advantage is that it is fully compatible with .Net framework and it is considered as the main extension method for future Office products. The main limitation is that it mainly provides document level accessibility. This means that the extension will only be loaded and executed when a particular Office document is opened. The extension will also be unloaded when the corresponding document is closed. This is opposed to application level add-in, where the add-in will be loaded whenever a new Office application instance is initiated.

According to Carter and Lippert \[^{99}\], VSTO is the best candidate for extending Microsoft Excel spreadsheet because of the following reasons:

- It is the most evolving extensibility model and has a clear road map from Microsoft; and

- It has maximal functionality and performance in the .Net framework

\[^{24}\]Component Object Model
2.2.7 Table-Oriented Programming

Table-Oriented Programming (TOP) is a programming paradigm dealing with tables, lists, indexing, sorting, searching, filtering of data [65]. Spreadsheet model is a special case of table-oriented programming model.

There are various situations where TOP can be served as a design philosophy:

- Mashup tool. TOP can be applied in designing mashup tools. For example, Mashroom [207] is a programming environment using nested table model and spreadsheet-like programming for mashups. It defines a set of mashup operators facilitating a spreadsheet-like programming environment for end-users. Using these operators users can import service as a table, visually drag and drop a column onto other for merging, invoke and link another service directly on a range of cells.

- Data query tool. TOP can also be use in designing graphical query languages based on the nested table such as QSBByE [118], GXQL [178], Liu et al. [150], or Witkowski et al. [211].

- Relational DBMS engine. TOP concept can also be applied to the design of a relational database engine (e.g., [197]).

- Data management tool. Google Fusion Table [27] is an application allowing users to upload data table (e.g., spreadsheet file), join (or “fuse”) the data in some column with other tables and view data in spreadsheet-like views.

2.3 Towards a SB Mashup Framework

2.3.1 An Alternative Paradigm for Mashups

Typically, mashups use Web browsers as environments for rapid, on-the-fly integration of data, functionalities and services. The reasons for using the browser are apparent: intuitive, drag & drop manipulations, and easy to use for end-users.
However, the development and execution of browser-based mashup applications may depend on the availability of additional browser’s plug-ins or extensions. This sometime causes inconvenience to the users. In addition, in some scenarios, using browser for mashup is not the best choice. For example, in the scenario that requires heavy computation and relationship between data, spreadsheet could be a better choice. Spreadsheets are widely used programs and users can turn hours of previously tedious works into automatic processes by writing some formulas.

It is too early to judge which model is the best for mashup. To our best knowledge, the spreadsheet paradigm for mashup has its own advantages over browser-based model in some scenarios and vice versa. Moreover, thanks to the large user base of spreadsheets, it is worth providing a mashup environment which is designed in the spreadsheet paradigm. So it would be better to have both models in parallel for supplement instead of two competing models. With a SB mashup tool, spreadsheet users are given an opportunity to access more information and techniques that may make their jobs more efficient.

### 2.3.2 Why Spreadsheets?

In the following, we explain why the spreadsheet paradigm is suitable for mashup:

- **User-centric.** Spreadsheet users are not expected to have high level of technical knowledge in order to use the application. Users mainly use simple user interactions to manipulate with spreadsheet (e.g., write formulas, copy, paste, drag, and drop).

- **Lightweight.** The spreadsheet paradigm provides a lightweight environment for users to develop their applications. It offers a mixture of development and runtime environments, facilitating immediate feedback to the users [142].

- **Rapid development.** Spreadsheets provide a programming environment that is very familiar with non-programmer knowledge workers. Spreadsheets make these users think that spreadsheet work is not necessarily a programming activity (i.e., the user thinks that they are not “programming”, just “write formulas” or “build a template/model”) [163].
- **Leverage community.** Spreadsheets are widely used applications with millions of users [184]. The popularity of spreadsheets can help increase the level of acceptance of any mashup tool built in a familiar environment.

## 2.3.3 Motivating Scenario

Let us illustrate the benefits of mashup integration with spreadsheet through a simple scenario\(^{25}\): Mary, a student, is learning Spanish. To study a word, she uses text-to-speech service (TTS), translation service (TSL) and spell-checking service (SPL). Instead of invoking these services separately and manually link the results, she opens a spreadsheet program, inputs an English word, writes formulas to link these services so that the completed formulas will now automatically suggest spelling corrections, pronounce the word in English, translate the word to Spanish and then finally pronounce the word in Spanish. Obviously, the ability to quickly compose such services in spreadsheet enables her to create new applications which suit her needs and save her time, while interacting with a tool she is already familiar with.

Figure 2.5 illustrates this scenario with two different views of the spreadsheet: formula view and evaluation view.

\(^{25}\)This scenario is inspired by the scenario provided in [166]
2.3.4 Possible Ways of Enabling Mashups in Spreadsheets

In this subsection, we investigate possible ways of enabling mashup using the spreadsheet environment as well as discussing the benefits and drawbacks of each approach.

Implicit Mashups

The idea of implicit mashup model comes from the fact that mashup applications are made by using natural data flow created by cell dependencies in spreadsheets. For example, a very simple mashup application can be implemented by linking one service’s output into another service’s input field (e.g., in the Mary’s scenario in Section 2.3.3, the translation service takes the output of the spelling service as one of its input parameters). StrikeIron SOA Express for Excel is a typical example of this approach. Carefully analysing the data flows created by cell dependencies, we can show that spreadsheet paradigm naturally supports some basic workflow patterns, namely: sequence, parallel split and simple merge. This approach is easy to learn as it conforms to the conventional spreadsheet programming metaphors (i.e., cell referencing, cell formula). However, the main limitations of this approach are as follows:

- The mashup approach mainly relies on services that operate in a request-response manner (i.e., it expects a service to always return output). It does not consider other types of services which do not operate in such a manner (e.g., one-way only interaction service); and
- The execution order of services in a mashup application relies on data flow created by cell referencing. There is no explicit control-flow construct specified in the application thus it is less suitable for a scenario which requires complex execution logic.

Event-based Mashups

In event-based mashup, the flow of service executions is determined by user actions (e.g., button clicks, key presses) or interactions with other services (i.e., the comple-
2.3. Towards a SB Mashup Framework

...tion of an action in one service may cause an action in other services). Autonomic Task Manager for Administrators (also known as A1) [142] is a typical example of this approach. Each cell in A1 spreadsheet is an arbitrary Java object. Each object is associated with a listener which is designed to manage events. The listener explicitly triggers the execution of a code block based on a condition. For example, the following code snippet shows how a listener construct $\text{on}(\cdot)$ is used to trigger cell $B2$ to increase its value by 1 when a button in cell $B1$ is pressed:

$$
B1 := \text{Button}(); \\
B2 := 0; \\
B3 : \text{on}(B1) \ B2 = B2 + 1;
$$

By using events, the users can express all basic workflow patterns and some complicated patterns (e.g., iteration). However, the event-based mashup model requires the users to have programming knowledge (e.g., the notion of events, expressions, assigning commands) and the conventional spreadsheet program itself needs to be extended to handle more complicated event conditions such as aggregating events.

**Mashups using Shared Process Data**

This mashup model utilises not only the data required/produced by services (i.e., input and output) but also the meta-data generated during the execution of services (e.g., service A can have status: “initiated”, “processing” or “completed”. The mashup application simply tests for the existence of these data in order to decide if a service B following service A can be started. In this approach, a shared data repository is maintained by the mashup framework. Every piece of data (e.g., the input/output of a service, or the execution status of a service) is read from/written to the repository so that it can be shared amongst the services. AMICO:CALC [166] is a typical example of this approach. AMICO:CALC uses variables (which encapsulate services’ data structure) to abstract the heterogeneity of services. Users are provided with various predefined functions (e.g., $\text{Amico\_Read}$, $\text{Amico\_Write}$, etc) to get data from and to post data to services. In [166], authors demonstrated that AMICO:CALC supports the basic workflow patterns [199]. The shared data repository allows the users to express composition logic using a combination of
low-level operations such as *Amico_Read* and *Amico_Write*. However the main limitation of this approach is that it requires an external repository to store the shared data. Users need to be aware of the presence of these data in order to build mashup applications, thus increasing the learning curve (i.e., not a natural spreadsheet paradigm).

**Explicit Mashups**

The Explicit mashup model allows users to explicitly define the execution order of Web services in a mashup application by using predefined flow-control constructs (e.g., sequence, if-then-else, for-loop). Husky [192] is a typical example of this approach. The composition logic in Husky is defined by transforming the spatial organisation of services’ events into their ordering in time. Time progresses from left to right and from top to bottom in cell blocks. A set of adjacent cells makes a sequence of events, while empty cells disjoin the workspace into temporally independent event sequences. By introducing the notion of time, Husky explicitly introduces some operators such as iteration, if-then control, etc. By using predefined constructs, this approach gives the users more benefits of generality and extensibility of the language. However, it may break the philosophy of spreadsheet programming and increase the learning curve.

**Discussion**

Through the investigation, we identified that although the implicit mashup model has limitations, it has the lowest learning curve [217] which is the most important feature of a mashup development environment. We will rely on this approach to create a SB mashup framework with two guiding principles: usage simplicity and generic applicability. In order to achieve such a framework, there are several challenges that need to be tackled as follows:

- *Implicit control flow created by cell referencing is insufficient to define all composition possibilities*. Although cell referencing mechanism makes composition easy to use for end-users, using it is insufficient to define all the control flow
possibilities. Consider a scenario where there is no data dependency between services, or where a user wants the execution order of the mashup application to be different from the implicit control flow created by cell dependencies, we need some manual efforts to define the control flow. For example, in our motivating scenario, Mary cannot swap the execution order of the TTS service for English word in cell B3 with the TTS service for Spanish word in cell B5 since they are already have constrains with the TSL service in cell B4 and the SPL service in cell B2;

- **Easy definition of composition logic.** Spreadsheets typically offer users with manipulations such as writing formulas, cell referencing, copy/paste/move-ing cell contents or formula, etc. Defining mashup logic is normally done by writing code (e.g., Google Mashup Editor [29], WMSL [182], Swashup [157]), or visually linking service components (e.g., Yahoo Pipes[80], Microsoft Popfly [44], and Marmite [214]) which are neither intuitive to spreadsheet users nor maintain the spreadsheet programming metaphor. We argue that it is important to provide a mashup development method which spreadsheet users are already familiar with as well as to maintain the simplicity and declarative nature of the spreadsheet paradigm. The mashup development tool should use spreadsheet-provided manipulations; and

- **Generic applicability.** The mashup framework should be generic to cater for a wide range of mashup applications in multiple situations. It should be able to coordinate Web services, manipulate and visualise data created by the services.

### 2.3.5 A Reference Architecture for SB Mashup Tools

In this subsection, we describe a reference architecture of SB mashup tools. Through the architecture, the readers should be able to imagine how a mashup application is built in the spreadsheet environment. As depicted in Figure 2.6, the architecture comprises of the following four elements:

- **Spreadsheet Interface (1).** This element is a spreadsheet interface which includes a grid of cells and a formula editor. The spreadsheet interface plays
2.3. Towards a SB Mashup Framework

Figure 2.6: Reference Architecture for Mashup Tools using Spreadsheet Paradigm

- **Repository (2).** This element is a repository of all mashup components available in the tool. Mashup components are representations of external resources (e.g., Web service, file, database or application) to be used in the tool. For example, in Mary’s mashup application (Section 2.3.3), SPL is a component representing a SOAP-based service that suggests correct spelling of words. It is created based on the WSDL document of the spelling service.

- **Mashup Engine (3).** Mashup engine is responsible for evaluating the formula (i.e., composition logic) and “wire” mashup components together. It operates in a centrally-mediated fashion and plays a role as a server to manage the execution flow among components. Mashup engine is also responsible for maintaining the formula evaluation context (i.e., the mashup result) and facilitates the reaction to cell modification by triggering the re-evaluation of dependent formula (i.e., upon a service invocation returns, the corresponding references need to be updated with the returning value). For example, in the Mary’s mashup application, whenever she changes the value in cell B1...
(e.g., change “Orage” to “Lemoon”\textsuperscript{26}), the formulas in cells B2 – B6 will be re-evaluated by the engine to create new results.

- **Wrappers (4).** This element facilitates interoperability among resources which have different data formats (e.g., HTML, XML, and RSS) or use different access protocols (e.g., SOAP and REST). For example, we need different wrappers to create components for correctly serving Mary’s scenario, such as SOAP service wrapper for the SPL service, REST service wrapper for the LOG service or an application specific wrapper for the TTS service.

### 2.4 Chapter Summary

In this chapter, we have presented an introduction to Service-oriented Computing and spreadsheet development research areas with some terms, techniques, products and standards. We have also discussed the possibility of integrating mashups with the spreadsheet paradigm. The incentive of using spreadsheet as an alternative mashup paradigm has been discussed with a motivation scenario. We have surveyed four possible ways of enabling mashup in spreadsheets and have sketched a reference model for SB mashup development framework. In the next chapters, we describe the main contributions of this dissertation.

\textsuperscript{26}Note that both words are in incorrect spelling format
Chapter 3

An Analysis of the State-of-the-art SB Mashup Tools

In this chapter, we present an analysis on the state-of-the-art SB mashup tools and position the SB mashups among other mashup research.

This chapter is organised as follows. We, first, introduce the goal and motivation of the analysis in Section 3.1. We, then, review the related work on classifying mashup tools in Section 3.2. Samples (i.e., mashup tools) and dimensions for the analysis are presented in Section 3.3 and Section 3.4, respectively. The analysis results are summed up in Section 3.5. Finally, we summarise this chapter in Section 3.6.

3.1 Introduction

Recently, we have witnessed a sharp rise of mashup tools and applications on the Web. In 2006 alone, there were hundreds of individual mashup tools released. Yahoo Pipes [80], Google Mashup Editor [29], Microsoft Popfly [44], and Intel Mash Maker [36], just to name a few. Many of the early tools, such as Google Mashup Editor and Popfly have since been discontinued, but there has been continuing development of novel research prototypes.

Different mashup tools solve different problems, so users need to choose an ap-
3.2. Related Work

Appropriate mashup tool for their task. For example, a data mashup tool allows users to retrieve data from one or several data sources, to process and mix data and to publish the result either as a feed or as another data source. A process mashup tool allows users to automate processes by orchestrating services, forms, and other resources in a workflow. A mashup tool that focuses on Web page customisation is used for altering Web pages by removing elements or adding additional widgets.

Spreadsheets are gaining attention as potential mashup environments in both academia and industry communities. However, to date, there is no thorough analysis of existing SB mashup tools, nor a study of possible ways to incorporate the mashup and spreadsheets. Understanding existing SB mashup tools and other related issues will pave the way for research directions to effectively address some of the research problems. In the sections below, we will present a qualitative analysis on a set of mashup tools using the spreadsheet paradigm.

3.2 Related Work

Approaches for classifying mashup tools are diverse. They range from the types of mashup applications a tool creates, the target users of the tools, to the mashup language paradigms that a tool adopts. According to Maximilien et al. [158], there are three types of mashup applications: data mashup, process mashup and presentation mashup (i.e., Web page customisation). Many surveys on mashup tools listed below follow these classifications.

In [125], Grammel and Storey reviewed six mashup tools from the End-user Development perspective. They utilised six dimensions for the review: level of abstraction, learning support, community support, search-ability, UI design, and software engineering techniques. The six mashup tools were grouped by types of mashup applications they generated. Di Lorenzo et al. [112] analysed seven popular mashup tools from the view point of data integration. The dimensions for the analysis cover various aspects of data level concerns: data format and access, data model, data mapping, data flow operators, data refresh, output format, extensibility, and sharing. Daniel et al. [108] discussed the application integration issues in mashups with re-
3.2. Related Work

spect to the interface integration aspect. Four dimensions for the analysis include models and languages for specifying components; models and languages for specifying component composition; communication styles; and discovery and binding mechanisms. Wong and Hong [213] conducted a qualitative survey using seven criteria: search, visualisation, real-time, widget, personalised, folksonomy, and in-situ use. These criteria are arbitrary collected, covering mashup issues in three layers of the integration: data, process and presentation. Mashup tools are judged by analysing the applications that they generate based on these criteria.

By utilising a different approach, Fischer et al. [119] outlined the mashup tools landscape in relation to the development paradigm and the target users. According to the development paradigm, they classified mashup tools into six categories: integrated development environment, scripting language, spreadsheet, wiring paradigm, programming by demonstration, and automatic creation. Users of mashup tools can be divided into three classes: developer, power user and casual user.

Hoyer and Fischer [136] provided a market overview of more than thirty enterprise mashup tools. Their criteria for evaluation include three groups of information: general information (e.g., vendor of a tool), functionality (e.g., catalogue and composition style), and usability (e.g., user-friendliness of a tool).

Yu et al. [219] characterised mashup tools from the view of data and application integration systems. They looked at the objects of integration (i.e., components), how such objects are glued together (i.e., the composition logic) and the software instrument (i.e., development environment and runtime environment).

These surveys on mashup tools in general helped us to form our own evaluation and analysis criteria for the SB mashup tools.

In the following, we present an analysis on the state-of-the-art SB mashup tools. Our analysis shares the same viewpoint with Yu et al. [219] in the sense that we consider mashup tools as data and application integration systems. However we define different dimensions that are adequately applied to the spreadsheet paradigm. To the best of our knowledge, our work is the first analysis of mashup tools that are based on the spreadsheet paradigm.

The landscape of mashup tools is broad, reflecting the rapidly evolving tech-
nologies in the SOC area. In Figure 3.1 we sketch the landscape of all types of mashups tools and highlight where SB mashup tools are positioned. We utilise two dimensions for the landscape: **paradigms used by mashup tools** and **types of mashup applications**.

![Figure 3.1: The Mashup Tools Landscape](image_url)

1. Eclipse IDE [17],
2. Bungee Connect Platform [6],
3. IBM Mashup Center [34],
4. Google Mashup Editor [29],
5. RSSBus [59],
6. Web Mashup Scripting Language [182],
7. Swashup [157],
8. Autonomic Task Manager for Administrators [142],
9. Human-centered Service Composition Workspace and Methodology [192],
10. AMICO:CALC [166],
11. Mashroom [207],
12. SpreadMash [143],
13. StrikelIron [63],
14. Extensio Extender [22],
15. On-line spreadsheets (Google docs & spreadsheets [26], EditGrid [18]),
16. Mixup [220],
17. JackBe Presto Wires [38],
18. Kapow [40],
19. Serena Mashup Composer [61],
20. WebRB [147],
21. Microsoft Popfly [44],
22. Yahoo Pipes [80],
23. CMU’s Marmite [214],
24. DERI Pipes [14],
25. Potluck [137],
26. Dapper [13],
27. Karma [190],
28. Intel MashMaker [39],
29. Matchup [126], and
30. Microsoft Visual Studio [45].
3.3 Review Samples

In this analysis, we review the following tools: spreadsheet connectors for mashup engines, on-line spreadsheets, StrikeIron [63], Extensio Extender [22], Al [142], AMICO:CALC [166], Husky [192], SpreadMash [143], and Mashroom [207].

1. **Spreadsheet connectors for mashup engines.** JackBe Presto [38], IBM Mashup Center [34], and Kapow [40] are mashup engines which provide separate spreadsheet connectors (e.g., [20, 35], and [82]). These spreadsheet connectors allow Microsoft Excel users to directly consume already-built mashup application (outside spreadsheets) in the spreadsheet environment. However, they are not fully-functional SB mashup tools in their own right.

2. **On-line spreadsheets.** Recently, online spreadsheets such as Google Docs & Spreadsheet [26], EditGrid [18], Zoho [81], and wikiCalc [77] have gained popularity. Within these spreadsheets, the emphasis of service extensions is on accessing external resources (e.g., Web services and spreadsheet data) from spreadsheet environment, not on using spreadsheet as a mashup development tool. The spreadsheet itself is a productivity tool which is now equipped with functions to access external resources and place data in cells. For example:

   - Zoho introduces several APIs that enable developers to access spreadsheet data through REST and XML-RPC interfaces;
   - EditGrid enables access to spreadsheets through RESTful and SOAP-based APIs (e.g., Alexa [2], Google search [31], Xignite currency exchange rate [23], and Xignite stock quote price [79]); and
   - Google Docs & Spreadsheets can make use of predefined functions (e.g., GoogleFinance, GoogleLookup, and GoogleTournament) for getting data from various financial services

3. **StrikeIron SOA Express for Excel [63] and Extensio Extender for Microsoft Excel [22]¹** are commercial Microsoft Excel plug-ins. The basic idea of these tools is that they allow users to fetch data from the various data sources (e.g.,

¹We will call StrikeIron and Extensio Extender for short
Web services) by using XML/HTTP and put the data into Microsoft Excel worksheets. The data “live” in cells, and are integrated directly by users by “hooking” the output value of one service with the input parameter of another service. StrikeIron only allows access to SOAP-based services while Extensio Extender enables access to a wider range of resources (e.g., database).

4. **AMICO:CALC** \[166\] is an OpenOffice Calc extension that let users to configure, to connect and to compose services through a spreadsheet interface. The execution of a mashup application is based on a middleware named Adaptable Multi-Interface COmmunicator (i.e., AMICO). By using a combination of Amico\_Read and Amico\_Write functions, the users can model basic control flow patterns \[199\] in their mashup applications.

5. **A1**, also known as Autonomic Task Manager for Administrators \[142\], is a research prototype from IBM facilitating a programming environment for system administrators. By using A1, users not only use a spreadsheet-like environment with a task-specific language to access remote and heterogeneous systems but also gather and integrate status data and orchestrate control of different systems. A1 extends the conventional spreadsheets with the following features:

   - Allowing cells to contain arbitrary Java objects;
   - Extending cell formulas to include calls to methods of cell objects; and
   - Allowing cells to contain procedural code blocks whose execution is triggered by events in the sheet.

6. **Husky** \[192\] is a service composition tool. It extends the spreadsheet paradigm enabling users to express composition logic through a spatial arrangement of component services within spreadsheet cells. Users are also provided predefined operators (e.g., Clipboard, Queue, TokenCenter, BrokerCenter) to be used in their composite applications.

7. **Mashroom** \[207\] is a mashup tool that relies on nested table algebra and spreadsheet programming metaphor to simplify the mashup creation process. It mainly focuses on data mashups and the data visualisation in Mashroom is based on nested table.
8. *SpreadMash* [143] is a research prototype that provides a high-level language and tool for interactive data browsing and analysis for data services. The striking feature of SpreadMash is a repository of application building blocks called data widgets that characterise various data importation and presentation patterns in spreadsheets. Multiple widgets can be composed by means of links to build composite worksheets.

### 3.4 Review Dimensions

Inspired by the mashup architecture in [219], we believe that a SB mashup framework needs the definitions of the following four basic elements: *component model*, *composition model*, *development environment* and *runtime environment*. In this section, we introduce dimensions in these four elements.

#### 3.4.1 Component Model

This element determines the nature of components by means of the following dimensions:

**Component Data Model**

A data model, as defined in [198], contains a notation for describing data and a set of operations used to manipulate that data. Conventional spreadsheets have limited atomic data types, such as number, string or datetime. Some spreadsheets extend the conventional spreadsheet data model to accommodate heterogeneous resources (e.g., Web services, databases, and files) so that a cell may contain complex data types such as object or XML document. We identify two data models which are currently used in the SB mashup tools: grid-based and object-based.

In the grid-based model, data are presented in a range of cells and each cell contains an atomic data value (i.e., number, string, and datetime). By using this data model, complex data need to be transformed into two-dimensional structure of the model. For example, in AMICO:CALC, a service invocation may return
an XML document with complex structure (e.g., nested). In order to display the output, AMICO:CALC uses service adapters to linearise the complex structure to a set of variables, which can be easily mapped into spreadsheets cells and formulas. This transformation at the component data model level means that users do not have to deal with complex hierarchical structures.

In the object-based model, since the spreadsheet cell is extended to contain complex data types, a complex XML document can be mapped into a single cell. For example, A1 extends the conventional spreadsheets by allowing cells to contain arbitrary Java objects and formulas’ calls to methods in cell objects. The following code snippet illustrates object-based data model used in A1:

\[
\begin{align*}
\text{C1: } & \quad 149.171.225.4 \\
\text{C2: } & \quad \text{JMX(“http://example.net/IPtoCountry?wsdl”)}
\text{C3: } & \quad =\text{C2.IP2Country(C1)}
\end{align*}
\]

Cell C1 contains a string value representing for an IP address. In cell C2, we use JMX (Java Management Extensions) binding to a SOAP service, namely IPToCountry. This Web service has two operations: IP2Country() and Country2IP() which return a country name of a given IP address and vice versa. In cell C3, we invoke the IP2Country() operation of this Web service. The result is returned as an XML document and stored in cell C3.

To illustrate the differences between two component data models, let us consider an example given in Figure 3.2 which shows an RSS feed of the Google news service\(^2\). The grid-based model is presented in Figure 3.2(a). The news data, presented as an XML document, are mapped into the range of cells B1 : D6. Figure 3.2(b) shows how the data can be presented by using the object-based model. The news data are mapped into cell B1. Each element of the news could be accessed by using the “dot” notation in the mapping definition. For example, the value of the “language” element can be achieved by using the following formula:

\[
=\text{B1.language}
\]

\(^{2}\text{http://news.google.com}\)
Components are entities that allow users to interact with external resources inside a mashup tool. Resources can have heterogeneous data formats and use different access protocols. Leveraging on this dimension, we investigate different data formats and access protocols that are supported by the mashup tools. Common data formats of component services include HTML, XML, RSS/Atom, JSON, spreadsheet, and text file. The access protocols could be HTTP, TCP, UDP, SMTP, REST, SOAP, RPC or user-specific application protocols.

Extensibility

Extensibility is the ability of a mashup tool supporting users to define their own functionalities (e.g., creating components or defining new mashup operators). Normally, expert users use a general purpose language (e.g., Java or C#) to achieve this extension. For example, A1 allows users to extend the functionality of the tool to best suit their needs by developing custom plug-in components. These plug-in components provide new interaction capabilities and functionalities and could be developed by creating a new Java class extending the A1Component class. The following code snippet illustrates an example of a user-defined component named SampleComponent:
package com.ibm.ai.plugin.api.samples;

public class SampleComponent {
    String name;
    public SampleComponent(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name = name;
    }
}

Extensibility can be also achieved by modifying existing functionality provided by the tool. For example, AMICO:CALC provides a generic adapter for services using TCP. Users can modify (or parameterise) this adapter to facilitate accessibility for a specific application using TCP protocol (e.g., a voice recognition application).

3.4.2 Composition Model

This element determines how components can be glued together through the following dimensions:

Mashup Strategy

This dimension identifies the mashup strategy used in SB mashup tools. We divide the mashup strategies into the following two categories:

- The first category concerns the time when a mashup application is created - design time or run time. Static mashup takes place during the design time when users plan the logic and components to be used in the mashup application. Components are selected and linked together, and finally compiled and
deployed. This strategy is suitable with a scenario where components and services are stable (e.g., in terms of availability) and rarely changing (e.g., data structure).

If the component providers frequently change the structure of data or business logic, static mashup is too restrictive and requires users to re-built mashup applications in order to adapt to the changes. *Dynamic* mashup may solve this problem by adapting to unpredictable changes at the run time. The Web environment is highly dynamic where new services become available and change frequently. Ideally, mashup applications should be able to transparently adapt to environment changes, and to adapt to customer requirements with minimal user interventions; and

- The second category concerns the way users build mashup applications which could be either *manual* or *automatic*. In *manual* mashup, users manually describe the composition logic (i.e., data flow and control flow) for mashup applications. This makes the development process of mashup applications tedious, error-prone and time consuming. On the contrary, automatic mashup does not require users to be involved in low-level development process. They only need to provide high level requirements or goals of mashup applications. The mashup tool will automatically suggest or choose composition plan based on given context or semantic of application and components.

### Orchestration Style

Orchestration describes the arrangement, coordination and management of components in a mashup application. It contains business logic of the mashup application specifically about execution order of components. There are three orchestration styles could be used in the spreadsheet paradigm, namely *flow-based*, *layout-based* and *event-based*:

- **Flow-based.** In flow-based orchestration style, mashup applications are built by using natural data flow and control flow created by cell dependencies in spreadsheet (i.e., linking one service’s output into another service’s input fields). For example, as shown in “Formula view” in Figure 2.5 the Mary’s
scenario can be implemented by “hooking” output of the SPL service in cell B2 with input field of the TSL service in cell B4. This orchestration style is naturally supported by spreadsheet paradigm and easy to use. However, as discussed in [131], it has limitations when users need to build a mashup application with a complex control flow logic.

- **Layout-based.** In layout-based orchestration style, users are allowed to explicitly define the execution order of components in mashup applications by organising spatial position of components. For example, in Husky, the execution order of a mashup application is defined by spatial organisation of services in a grid of cells. Husky transforms the spatial organisation of services’ events into their ordering in time. Time progresses from left to right and from top to bottom in cell blocks. A set of adjacent cells makes a sequence of events, while empty cells disjoin the workspace into temporally independent event sequences. Let consider the implementation of the Mary’s scenario by using Husky in Table 3.1. The execution flow is defined by a sequence of services in adjacent cells from C2 to C7. The conditional split defined in cell C3 will trigger the operation in cell C4 if the content of cell C2 is not empty (i.e., similar to “goto” or “jump” statement in some programming languages). The rest of the program is evaluated according to the following order: evaluate formula in cell C5, cell C6 and finally cell C7. By explicitly defining the control flow, this orchestration style gives the users the benefits of generality and extensibility. However, it may break the philosophy of spreadsheet paradigm and increase the learning curve [131].

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Orage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><a href="http://tts.com">http://tts.com</a></td>
<td>If [C2]&lt;”” Then Set Clock To [C4]</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Execute[A7] “Send”[C2][C5]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Layout-based Orchestration in Husky

- **Event-based.** In event-based orchestration style, the execution flow of a
mashup application is determined by *user actions* (e.g., button clicks and key presses) or *interactions with other services* (i.e., the completion of an action in one service may cause an action in other service). For example, in the A1 system, each cell is an arbitrary Java object. Each object is associated with a listener which is designed to manage events. The listener explicitly triggers the execution of a code block based on a condition. There are two listener constructs: (i) `on()` is used to trigger an operation based on events; and (ii) `when()` is used to trigger an operation based on a boolean expression. The Mary’s scenario can be implemented in A1 as shown in Table 3.2.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>SPLService(&quot;address&quot;)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>TTSService(&quot;address&quot;)</code></td>
</tr>
<tr>
<td>3</td>
<td><code>TSLService(&quot;address&quot;)</code></td>
</tr>
<tr>
<td>4</td>
<td><code>WorkbookService(&quot;address&quot;)</code></td>
</tr>
<tr>
<td>5</td>
<td><code>Orage;</code></td>
</tr>
<tr>
<td>6</td>
<td><code>Button();</code></td>
</tr>
<tr>
<td>7</td>
<td><code>on(B6) B1!Spell(B5);</code></td>
</tr>
<tr>
<td>8</td>
<td><code>when(B7 &lt;&gt; &quot;&quot;) B2!VoiceEN(B7);</code></td>
</tr>
<tr>
<td>9</td>
<td><code>when(B7 &lt;&gt; &quot;&quot;) B3!E2S(B7);</code></td>
</tr>
<tr>
<td>10</td>
<td><code>on(B9) B2!VoiceSP(B9);</code></td>
</tr>
<tr>
<td>11</td>
<td><code>on(B10) B4!Send(B7, B9);</code></td>
</tr>
</tbody>
</table>

Table 3.2: Event-based Orchestration in A1

Services definitions are defined from cell B1 to cell B4. The program starts when user clicks on the button defined in cell B6. Based on user-generated events (e.g., button clicked), the formula in cell B7 returns a spelling suggestion. Based on the value of cell B7 (i.e., the output of spelling service), two formulas in cell B8 and B9 are evaluated. Similarly, the executions of formulas defined in cell B10 and cell B11 are based on events generated by the executions of formulas in cell B9 and cell B10, respectively.

**Data Passing Style**

This dimension identifies how the data are passed between components. There are two data passing styles:

- *Data Flow*. In this style, data are passed from one component to another
component. For example, in Mary’s scenario, the data from output of the SPL service in cell B2 is directly transferred to input field of the TSL service in cell B4. This is the default data passing style of spreadsheet applications.

- **Blackboard.** In this style, data are read from and written to a shared data repository (e.g., variable). For example, AMICO:CALC uses variables (which encapsulate services’ data structure) to abstract the heterogeneity of services. Users are provided with various predefined functions (e.g., \texttt{Amico\_Read}, \texttt{Amico\_Write}, \texttt{Amico\_Read\_Delayed}, \texttt{Amico\_Write\_Delayed}) to get data from and to post data to services. The formula \texttt{Amico\_Read("SPL")} will receive a value when the spelling services is invoked. \texttt{Amico\_Write("SPL",B1)} will assign value in cell B1 to the variable SPL. This data passing style is used mainly in programming languages.

### 3.4.3 Development Environment

This element identifies characteristics of the development environment offered by a mashup framework through the following dimensions:

**Target Users**

We identify three types of users in SB mashup tools: developer, skilled user and novice user. A developer, who is the most skillful user, should be familiar with programming, Web technologies, different APIs as well as the usage of mashup tools. A skilled user has no programming skills but does have detailed functional knowledge about some technologies and a specific mashup tool. Novice user has only knowledge on the functionality of spreadsheet and be able to use some simple spreadsheet formulas and operations (e.g., cell reference, and copy/paste the content of a cell to another).

**Search-ability**

Search-ability expresses the ability of a mashup tool helping users in finding desired components (or even mashup applications) created by others. There are three types
of search as follows:

- **Text/Keyword-based.** Users can search on the title, description or tags of components by specifying keywords;

- **Browsing.** The tool displays a list of components in a typical file-finder or folder – sub-folder fashion; and

- **Context-specific suggestion.** The tool provides (or suggests) the needed component to users depending on the users’ context without having to search

### Community Features

One of the most important factors that could lead to the success of an End-user Development tool is the community support \[162\]. Some of the desirable community support features are as follows:

- **Sharing.** Components and mashup applications are created by a user but they can be run, copied or modified by other users;

- **Collaborating.** Users can co-operate to create or modify mashup applications; and

- **Discussion.** A framework is provided with a forum or chatting capability for users to exchange ideas and to help each other in order to build a community

### Software Engineering Aspects

This dimension concentrates on the software engineering aspect including the following features:

- **Debugging.** Users can locate and fix errors such as composition logic error and syntax error;

- **Testing.** Users can investigate the quality of mashup applications with respect to the context in which it is intended to operate; and
• Version control. Users can manage the changes and multiple users can access and modify mashup applications.

3.4.4 Runtime Environment

This element expresses how the results of mashup applications are delivered to users through the following dimensions:

Mashup Engine

We identify the following two types of mashup engine that can be used in the SB mashup tools:

• Mashup engine that based on spreadsheet evaluation engine (internal); and
• Mashup engine that developed as a plug-in to spreadsheet application (external)

On one hand, spreadsheet evaluation engine can be used for the data mashup by using natural data flow created by cell dependencies in spreadsheet. The mashup applications that are executed by using this mashup engine must use spreadsheet programming metaphor (e.g., using formula and cell referencing). For example, in StrikeIron, a very simple data mashup application can be implemented by linking one service’s output into another service’s input field. On the other hand, mashup engine can be separately developed from the conventional spreadsheet evaluation engine. In this case, the mashup applications do not need to conform to spreadsheet programming metaphor.

Execution Type

We identify the following three types of execution that can be used in mashup tools, namely centralized, distributed and hybrid. Centralized execution is similar to the client-server paradigm. In this case, the server is the central scheduler that controls the execution of the component services in a mashup application. The distributed
3.5 Analysis Results

paradigm, in contrast, expects the services to share their execution context. Each component service has its own coordinator, which has to collaborate with the coordinators of the other services in order to guarantee a correct ordered execution. A hybrid form of the distributed and centralized paradigms may be defined in the form that a coordinator controls not only one but a set of Web services [90].

Exception and Transaction Handling

Exception handling is a collection of mechanism supporting the detection, signaling and after-the-fact handling of unusual events whether erroneous or not [95]. In programming languages, exception handling is considered as a necessity part and exists in almost all of application development tools. However, existing spreadsheet mashup tools have very limited support for exception handling. In spreadsheet, exception handling is compatible with reasoning model of spreadsheet formula. For example, in Microsoft Excel, when an operation detects an exception, it will return one of seven possible error values (e.g., NULL, DIV/0, VALUE, REF, NAME, NUM, or N/A). The error value model in spreadsheet is different from the use of status flag in traditional programming languages in the sense that it can be ignored so that the error may not cause the whole spreadsheet program to crash.

A transaction is an atomic operation which may not be divided into smaller operations. Transaction handling is a mechanism supporting the ACID (i.e., Atomic, Consistent, Isolated, and Durable) properties of transactions.

3.5 Analysis Results

In this analysis, fourteen samples have been analysed by thirteen dimensions. A summary of the results is presented Table 3.3. In the following, we discuss the results.

The majority of the samples have the grid-based component data model. This design choice can be explained by the fact that by using grid-based model there is no need to extend traditional spreadsheet data model to present data. The issue of how to map structured data to spreadsheet data model has been addressed by existing
3.5. Analysis Results

<table>
<thead>
<tr>
<th>Data model</th>
<th>grid-based</th>
<th>object-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component access model</td>
<td>protocol(^a)</td>
<td>data format(^b)</td>
</tr>
<tr>
<td></td>
<td>(P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10})</td>
<td>(D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8)</td>
</tr>
<tr>
<td>Extensibility</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Mashup strategies</td>
<td>static</td>
<td>dynamic</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Orchestration styles</td>
<td>flow-based</td>
<td>layout-based</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Data passing styles</td>
<td>dataflow</td>
<td>blackboard</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Target user</td>
<td>developer</td>
<td>skilled user</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Search ability</td>
<td>text-based</td>
<td>browsing</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Community features</td>
<td>sharing</td>
<td>collaborating</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Software engineering aspects</td>
<td>debugging</td>
<td>testing</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Execution engine</td>
<td>spreadsheet</td>
<td>external</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
| (+) means the dimension is directly supported; (-) means the dimension is not supported; (NA) means the dimension is not mentioned in related publications.

\(^a\) \(P_1=HTTP; P_2=TCP; P_3=UDP; P_4=SMTP; P_5=REST; P_6=SOAP; P_7=RPC; P_8=application specific; P_9=ODBC; P_{10}=JDBC\);

\(^b\) \(D_1=HTML; D_2=XML; D_3=RSS; D_4=ATOM; D_5=JSON; D_6=XLS; D_7=RDF; D_8=Text\);
(+) means the dimension is directly supported; (-) means the dimension is not supported; (NA) means the dimension is not mentioned in related publications.

Table 3.3: Evaluation Results

<table>
<thead>
<tr>
<th>Execution types</th>
<th>centralized</th>
<th>distributed</th>
<th>AMICO:CALC</th>
<th>A1</th>
<th>Husky</th>
<th>Mashroom</th>
<th>SpreadMash</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Transaction &amp; exception handling</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

works. For example, SpreadMash provides a set of widgets for presenting complex data in spreadsheet paradigm, namely content, repeater, hierarchical, and index. StrikeIron provides a drag and drop interface for manually mapping data into target location in spreadsheet. The object-based data model is more sophisticated since it encapsulates object type in a cell. However, this data model requires more advanced knowledge from users and is more suitable with skilled users and developers than novice users. This is the reason of why there are only a few number of samples that are based on the object-based model.

Most of the samples provide support for REST and SOAP protocols using XML data format due to the prevalence of RESTful and SOAP-based services. Some samples allow users to “mashup” with other resources such as plain files, database, Web applications or even legacy applications. This greatly expands component resources from which users can make richer and better mashup applications. For example, AMICO:CALC supports various communication protocols (e.g., TCP, UDP, XML-RPC, Open Sound Control, HTTP, SOAP, SQL and some application-specific interfaces such as Sesame RDF, WordNet, etc) for interconnecting services with different interfaces.

In order to facilitate the mashup capability with a specific resource (e.g., a specific application), users need to write their own wrappers. Within the reviewed
samples, only A1 and AMICO:CALC allow users to extend Java class to write plug-ins/adapters for developing new functionalities of the samples. This could not be suitable with all users since it requires users to have programming skills.

All of the analysed samples use static and manual mashup strategies. Mashup development using these strategies is tedious, error-prone and time consuming while the mashup application is incapable of adapting to requirement changes. This limitation leads to the need to have an automatic and dynamic mashup tool supporting users in creating mashup applications.

Flow-based orchestration style is used in most of the samples since it conforms to natural spreadsheet programming metaphor (i.e., cell referencing). Layout-based orchestration is rarely used in mashup tools since it may break the functional programming nature of spreadsheet (i.e., it introduces procedural programming notions within spreadsheet). Event-based orchestration is based on the Event-Condition-Action form (i.e., on <event> if <condition> then <operation>) and is the preferred programming model for programmers.

The dominant data passing style is data flow. This could be explained by the fact that the data flow model is easy to use and exists in most of mashup tools (e.g., Yahoo pipes) and is naturally supported by the spreadsheet paradigm.

Most of the samples target at spreadsheet users who have limited programming skills. For example, users of Extensio Extender do not need to spend too much learning efforts in order to use the sample since the programming metaphor of the tool is similar to spreadsheet programming style. Some samples extend the spreadsheet paradigm to accommodate new orchestration styles and require users to learn a specific formula syntax in order to create a mashup application. For example, in A1, the user needs to learn how to write procedural codes within a cell formula. Moreover, Husky’s users need to learn the rule for spatial arrangement of services as well as four special objects (namely, Clipboard, Queue, TokenCenter and BrokerCenter) in order to define flow control of service invocations.

There is no sample that offers context-specific suggestion to the users. Using context-awareness may lead to the possibility of having a framework that relies on the context and incrementally guides users in developing mashup applications.
Browsing capability can make users feel conformable when they want to find desired components since it classifies components and services into different categories in a conventional manner. For example, StrikeIron provides browsing capabilities to its “Web service Marketplace” which has different categories (e.g., Communication, Financial, Marketing, E-Commerce, and Utilities). Text-base searching is useful when users know the names or descriptions of components. AMICO:CALC provide the text-based components searching in its middleware control panel.

To support collaboration among users, applications in A1 can be deployed as portlets in a J2EE-based Web portal server. In the portal, a user can execute or customise program deployed by other users. This is possible because all A1 spreadsheet is created in Java Swing-based client application which can be launched within the Web portal framework using Java WebStart. However in the reviewed tools, there is no approach providing a model to enable application reuse.

The surveyed mashup samples mainly focus on offering functionalities for creating mashup applications. The overall support for software engineering aspects such as debugging and version control is very limited. There is only AMICO:CALC supporting debugging. In AMICO:CALC, debugging is enabled through the use of middleware control panel for monitoring all the variables that are exchanged among services.

In the analysis, samples which adopt layout-based or event-based orchestration styles usually use external mashup engine. The dominant execution type is “centralized” and transaction handling is not supported.

All of samples have centralized execution type. For example, the execution of mashup program in AMICO:CALC is based on the AMICO middleware which is a centralized platform that facilitates adaptation, abstraction, and mediation for diverse service interfaces. The middleware maintains a list of variables which encapsulate data structures used by services and can be easily mapped into spreadsheet cells.

In summary, the existing SB mashup tools have provided many useful features for creating mashup applications. However there are plenty of rooms for further improvement. For example, a mashup tool should provide a generic development
environment for catering a wide range of mashup applications (e.g., data mashup, process mashup, and presentation mashup). In a further step, we intend to develop a generic purpose mashup development framework that leveraging on spreadsheet paradigm. This analysis results could be used as a basis to examine the suitability and applicability of a SB mashup tool.

The main limitation of this analysis is that we conducted a qualitative instead of quantitative survey, therefore, some of the observations could inherently be subjective. However, the results are only used to develop an initial understanding and create a foundation for further research. More rigorous surveys with quantitative measures are planned to add objectivity to the study.

### 3.6 Chapter Summary

In this chapter, we presented the landscape of mashups and positioned SB mashup tools in the area. The use of Web browsers as platforms for mashup development is widespread. When mashup applications are developed using Web browser platform, the users can leverage on interactive manipulations and intuitive user interfaces of browsers. The target user of tools based on browser is end users. Spreadsheet can be seen as an alternative mashup development paradigm for end users. IDEs and scripting languages are also available but intended primarily for professional developers. Programming by demonstration and automatic creation of mashup applications are appealing, however, still in their early stages.

We also conducted a qualitative analysis of state-of-the-art mashup tools using thirteen dimensions. We believed that the spreadsheet environment has inherent advantages over other mashup paradigm due to its popularity and familiarity with the users. We hoped that the analysis will be useful for the researchers and developers in the area to understand the landscape of SB mashup tools, the strengths and weaknesses of the current approaches, and to develop initial ideas on creating more advanced SB mashup frameworks.

In the next chapter, we will present our framework, named MashSheet, enabling generic mashup development based on spreadsheet paradigm.
Chapter 4

The MashSheet Framework

In this chapter we present our system MashSheet, a mashup development framework that supports generic application development. Our approach relies on spreadsheet paradigm, which is considered as one of the most successful End-user Development environments. We believe that spreadsheet paradigm can provide a programming environment that many users are already familiar with, hence would be a good environment for designing a generic purpose mashup development tool.

This chapter is organised as follows. The motivation and contributions of the MashSheet framework are presented in Section 4.1. We introduce the MashSheet’s architecture in Section 4.3. Section 4.4 and 4.5 present the application model and operators of MashSheet, respectively. We explain in Section 4.6 the processes of building and executing a MashSheet application with a running example. A discussion on advantages/disadvantages of MashSheet in comparison with the related work is presented in Section 4.7. Finally, we summarise this chapter in Section 4.8.

4.1 Introduction

The proliferation of Web services and online resources enabled new application development method, namely mashup, that combines together existing data and services to create ad-hoc, idiosyncratic applications. Mashup is increasingly popular amongst the Web development community. In ProgrammableWeb.com, for example, there are 5777 mashup applications and 3196 Web APIs registered. On average,
there are about 3 mashup applications and 3 Web APIs registered per day\textsuperscript{1}. The prevalence of mashup can be explained by the fact that nowadays many organisations (e.g., Yahoo, Google, and Amazon) open up their data via APIs, allowing users to exploit it in an easy and inexpensive way \cite{128}. Also, many Web-based tools have been developed to support the mashup activities, targeting users with little technical background (e.g., \cite{214} \cite{137}, and \cite{114}). On the back of the popularity of mashup, there have been a few suggestions to create mashup programming environments in spreadsheets, for example, StrikeIron \cite{63}, AMICO:CALC \cite{166}, Mashroom \cite{207}, and SpreadMash \cite{143}.

However, many mashup programming systems today are done in “special-purpose” and “hand-crafted” manner using a purposely created development environment \cite{128}. For example, MapCruncher \cite{42} is designed for users to create mashup applications involving maps. Swivel.com makes it easy for users to create graph mashup applications from multiple data tables. Yahoo Pipes \cite{80} is best suited to generate feeds. Due to the ad-hoc nature of the popular mashup tools, users are required to learn different tools, paradigms and syntaxes to write mashup applications. We believe that there is a need for a generic purpose mashup environment that can cater for a wider range of mashup applications with a uniform mashup paradigm.

The key contribution of this chapter is proposing a framework, named MashSheet, that facilitates an environment for mashup development and enactment. We focus on the generic aspect of the framework with the support of building mashup applications in three layers of the application stack: data, process and presentation.

To make the framework generic, we start with a generic data model that will support a wide-range of data types we need to deal with in the framework. We name the data type ‘MashSheet Object Type’ (MOT) and it is an XML-based data model. MOT is based on \cite{183}, where the conventional spreadsheet data model is extended to include complex data types to represent RSS feeds. Using MOT, we introduce two components to represent a Web service (\texttt{service}) and Web service output (\texttt{service-output}). At present, MashSheet can work with RESTful Web services, SOAP-based Web services and RSS data feeds. However, it is not difficult

\footnote{http://www.ProgrammableWeb.com/ the figures are as of 02 May 2011}
4.1. Introduction

to extend the range in the future (e.g., application-specific services).

MashSheet offers the following advantages over existing approaches:

- MashSheet defines all operators as spreadsheet formulas: data, process and visualisation. This means that the mashup operators would be like another spreadsheet formula to the users - the concept many users are already familiar with.

- MashSheet provides different classes of mashup operators to make the framework applicable to many mashup scenarios.

  - **Data operators**: In spreadsheet, the data are represented as simple data types (e.g., string and number). Web service invocation, on the other hand, often returns complex data type such as XML document. In other SB mashup tools, users have to “flatten” the complex data into two-dimensional grid of spreadsheet before data operations (e.g., sort, filter, and join) can be applied. This creates unnecessary step in the mashup application development process. In MashSheet, users can manipulate the complex data “as-is” by applying data mashup operators directly on MOT. In addition, any intermediary result created by the data operators can be reused by other data and visualisation mashup operators at any stage. This increases the reusability of intermediate data in the application.

  - **Process operators**: Evaluation of spreadsheet’s formula is driven by data dependencies between cells. This data-flow model allows users to define some process mashup operations using natural spreadsheet programming metaphor (e.g., sequence by using cell referencing mechanism).

However, the semantics of some control flow patterns are not inherently supported in spreadsheets and none of the existing SB mashup tools address this issue (e.g., exclusive choice and synchronisation). Supporting a basic set of control flow patterns is important for mashup component composing scenarios. We introduce an extension to the spreadsheet formula language so that the basic control flow patterns are supported within the mashup framework.
Visualisation operators. Data visualisation needs in mashups cover a wide range of options (e.g., grid, map, chart, and timeline). In the current SB mashup tools we investigated, data can only be represented in a grid of cells by using either simple grid (i.e., column and row) \([166, 63]\) or nested grid (i.e., hierarchical, index, and repeater) \([143, 207]\). Also, the data view needs to reflect any change in the data source and make it visible immediately to users.

We define visualisation operators as components to present Web services output data using different visualisation types (e.g., grid, chart, and map). The benefits of having these operators are as follows:

- **Flexibility**: the layout of data are automatically updated when there is a change in the data source;
- **Generic applicability**: we can apply different visualisation operators to produce different layouts of data; and
- **Easiness to use**: the operators can be called like spreadsheet functions

### 4.2 Reference Scenario

Let us consider the following scenario as our running example. Tom wants to create a mashup application that will help him find Points of Interest (POI) (e.g., restaurant and cinema) when he travels to a city. Tom considers the following five services:

- **RL**: with `read()` operation, it returns a list of restaurants with addresses;
- **CL**: with `read()` operation, it returns a list of cinemas with addresses;
- **DG**: with `getDR()` operation, it returns car driving direction;
- **WF**: with `getWeather()` operation, it returns weather forecast information; and
- **BU**: with `getBU()` operation, it returns bus itinerary

Figure [4.1](a) shows the scenario of the application. Tom inputs the address of a hotel, calls **RL** and **CL**, merges and sorts the results. He would use **DG** only when the
WF service reports rain, otherwise he prefers to use public transport, so information from BU will suffice. Finally, Tom visualises the direction results in a grid of cells.

Figure 4.1: The Tom’s Scenario

The structure of services (e.g., input and output) are illustrated in Figure 4.1(b). An example of the output data from RL is presented as follows:

```xml
<Restaurant>
  <Item>
    <Name>Grotta Carpi</Name>
    <Address>97-101 Anzac Pde</Address>
    <Postcode>2033</Postcode>
    <Rating>5</Rating>
  </Item>
  <Item>
    <Name>Golden Kingdom</Name>
    <Address>147-151 Anzac Pde</Address>
    <Postcode>2033</Postcode>
    <Rating>2</Rating>
  </Item>
  ...
</Restaurant>
```

And example of the data from CL is presented as follows:

```xml
<Cinema>
  <Item>
    <Name>Hoyts East Gardens</Name>
    <Location>152 Bunnerong Rd</Location>
    <Postcode>2036</Postcode>
  </Item>
  <Item>
    <Name>Ritz Randwick</Name>
  </Item>
</Cinema>
```
We will show how to realise this scenario throughout this chapter.

4.3 Overview of MashSheet

This section sketches the overall design of MashSheet and explains how it draws inspiration from spreadsheet. As depicted in Figure 4.2, the architecture of MashSheet comprises of four elements:

4.3.1 MashSheet’s Graphical User Interface (GUI)

This element (denoted 1) is a conventional spreadsheet interface which includes a grid of cells (denoted 1.1) and a mashup plug-in implemented as an add-in to the spreadsheet tool. Cells are capable of storing values and formulas, while mashup plug-in allows users to build mashups applications by specifying composition logic and layout information. The GUI plays a role as a development environment and users mainly interact with this component. The mashup plug-in includes the following elements:

Formula Editor

This element (denoted 1.2) allows users to enter mashup formulas. Note that MashSheet’s formula input area is different with the conventional formula input in spreadsheets. The reason for this design is that most of the spreadsheets do not allow developers to modify their evaluation engine with conventional input.

Component Explorer

This element (denoted 1.3) lets users to visualise structure of Web service (e.g., WSDL) and service output (e.g., XML).
4.3. Overview of MashSheet

Figure 4.2: The Architecture of MashSheet.
4.3. Overview of MashSheet

Component Repository Interface

This element (denoted 1.4) shows to the users a list of components (i.e., representations of Web services in the tool). It is a front-end of the “Component repository” element which will be defined below. Component Repository Interface also provides a search feature allowing users to search for a component by its name. To select a component for using in the grid, user can either drag and drop the component from Component Repository Interface into a cell or type its name in the Formula Editor.

Log

This element (denoted 1.5) displays the output message of each operation.

4.3.2 Component Repository

This element (denoted 2) is a repository of all mashup components available in the tool. Users can choose an external Web service and create a component that allows them to interact with the service within the tool. This service component can be assigned a friendly name (i.e., alias) and can be called in the MashSheet formula editor. For example, in the Tom’s scenario, DG is a component representing a SOAP-based service that provides driving direction information.

4.3.3 Mashup Engine

This key element (denoted 3) is responsible for evaluating the mashup formula (i.e., composition logic) and “wiring” components together. Since most of spreadsheet tools do not allow users to modify their formula evaluation mechanisms, mashup engine could be developed as an extension to spreadsheet evaluation engine. Mashup engine is also responsible for maintaining the formula evaluation context and facilitate the reaction to cell modification by triggering the re-evaluation of dependent formula (i.e., upon a service invocation returns, the corresponding references need to be updated with the returning value). We will explain the working of the engine with examples in the later sections.
4.3.4 Wrappers

Wrappers (denoted 4) facilitate interoperability among services which have different data formats (e.g., text, CSV, XML, RSS, Atom, KML, HTML, JSON, and YAML) or use different access protocols (e.g., SOAP, REST, HTTP, and SMTP). For example, we need different wrappers to create components correctly serving Tom’s scenario, such as SOAP wrapper for DG, RSS wrapper for RL. We rely on existing works (e.g., RSS.NET library [60]) to provide wrappers for services.

4.4 Application Model

MashSheet extends conventional spreadsheet to provide its application model. In this section, we leverage the formal definitions of spreadsheet in [84] to define the MashSheet application model. Figure 4.3 illustrates the model.

![Figure 4.3: MashSheet Application Model](image)

4.4.1 MashSheet Application

MashSheet application \((S)\) is a spreadsheet containing a collection of formulas \((F)\) and values \((V)\) embedded into a spatial structure. The spatial structure is a rectangular grid, whose elements can be addressed by pairs of integers. We denote a set of addresses as \(A = N \times N\) and an address as \(\delta (\delta \in A)\).

\[
S : A \rightarrow F.
\]
4.4.2 MashSheet Formula

MashSheet formulas \((f)\) can either be values \((v)\), refer to other formulas through cell addresses \((\delta)\) or mashup operations \((\omega)\) embedded into spatial structure.

\[
f \in F : v|\delta|\omega(f,..,f) .
\]

Values \((v \in V)\) will be defined in the Section 4.4.6 below. Address \((\delta)\) uses absolute address (i.e., a cell is identified by its absolute coordinates in the grid). Operations \((\omega)\) include mashup operations which will be defined later in Section 4.5.

4.4.3 MashSheet Cell

An element \((a,f) \in S\) is called a MashSheet cell. We use the notation \(C_S(a) = (a,S(a))\) to yield the cell (that is, address together with the formula) stored at address \(a\) in the MashSheet application \(S\). The evaluation of a cell \((a,f)\) in the context of MashSheet is denoted by \(\|(a,f)\|_S\) and is defined as follows:

\[
\|(a,v)\|_S = v .
\]

\[
\|(a,\delta)\|_S = \|C_S(a + \delta)\|_S .
\]

\[
\|(a,\omega(f_1,..,f_n))\|_S = \|\omega\| (\|(a,f_1)\|_S ,.., \|(a,f_n)\|_S) .
\]

In the above definition \(\|w\|\) refers to the function denoted by the predefined mashup operation \(\omega\). The semantics of cells containing formulas with circular references is undefined, like in most of conventional spreadsheets. The semantics of a MashSheet application \(S\) is simply given by the semantics of its cells:

\[
\|S\|_S = \{(a,\|(a,f)\|_S)|(a,f) \in S\} .
\]

4.4.4 MashSheet Cell Type

In MashSheet, a cell value \((v)\) can be either a simple data type \((\alpha)\) (e.g., number, string) or complex data type. We model the complex type as XML-based data
type and name it MashSheet Object Type (MOT). By using MOT, we build two components: service (SC) and service – output (OC) to represent Web service and its output data, respectively.

$$(\alpha, MOT) \in T.$$ 

$$\alpha ::= Num \mid String \mid Bool \mid Datetime \mid Undef.$$ 

$$MOT ::= SC \mid OC.$$ 

**Service Component (SC)**

MashSheet service component is a triplet \((\text{Name}, \text{Type}, \text{URL})\), where:

- **Name** is an unique identifier of a Web service (WS) to be used in MashSheet;
- **Type** is the service type of the WS; and
- **URL** is unified resource locater of the WS

When a SC is bound to a MashSheet cell, MashSheet creates an instance of the SC in the application. We call the instance of the SC as ‘bound SC’ and denote as \(SC^*\). A \(SC^*\) inherits all the attributes of the SC and has a new attribute: Location which is the address (a) of the SC in MashSheet.

**Service-output Component (OC)**

MashSheet service-output component is a triplet \((\text{Name}, \text{OprCell}, \text{OutputCell})\), where:

- **Name** is an unique identifier of a service – output component;
- **OprCell** is the address (a) of the \(SC^*\); and
- **OutputCell** is the address (a) of the OC

For example, let us consider a simple MashSheet application as follows:
4.4. Application Model

create('DG'; http://www.ecubicle.net/driving.asmx?WSDL')
(A1, bind('DG'));
(B2, invoke(A1.getDR(C1, C2)))

The application contains three formulas. All formula are entered into the Formula Editor of MashSheet GUI by user. The first formula creates \( SC_{DG} = \{DG, SOAP, http://www.ecubicle.net/driving.asmx?WSDL\} \). Component DG is now available for user to choose from the component repository.

User, then, needs to bind component DG to a specific cell. The evaluation of the formula in cell A1 creates \( SC_{DG} = \{DG, SOAP, http://www.ecubicle.net/driving.asmx?WSDL, A1\} \). The component DG is now registered with cell A1. From now on, user can refer to elements of DG by using its cell address (e.g., A1.getDR()).

The evaluation of an operation provided by DG will create a service-output component. The evaluation of the formula in cell B2 creates: \( OC_{DG} = \{DG, A1, B2\} \).

4.4.5 MashSheet Object Type

We use regular tree languages and tree automata from \[161\] to define MOT as follows.

MOT is a quadruples \((N, T, B, P)\), where:

- \( N \) is a finite set of nonterminals which are regular expressions represented by capitalised Italic characters (\( \epsilon \) is a null sequence of nonterminals),
- \( T \) is a set of terminals which are symbols represented by lowercase characters (\( pcdata \) is a special terminal character matched with any node in the tree),
- \( B \) is a set of start symbols, where \( B \in N \),
- \( P \) is a set of rules in the form of \( X \rightarrow a r \), where \( X \in N, a \in T, \) and \( r \) is a regular expression over \( N \).

Figure 4.4 and Figure 4.5 show examples of data produced by RL’s read() and CL’s read() operations (in Section 4.2) and their MOT representations, respectively. A simple data type (e.g., text) can also be mapped in to MOT by using a generic
4.4. Application Model

Figure 4.4: Example of RL Data and its MOT Representation.

<Restaurant>
  <Item>
    <Name>Grotta Carpi</Name>
    <Address>97-101 Anzac Pde</Address>
    <Postcode>2033</Postcode>
    <Rating>5</Rating>
  </Item>
  <Item>
    <Name>Golden Kingdom</Name>
    <Address>147-151 Anzac Pde</Address>
    <Postcode>2033</Postcode>
    <Rating>2</Rating>
  </Item>
  ...
</Restaurant>

N = [Restaurant, Item, Name, Address, Postcode, Rating, PCData];
T = [restaurant, item, name, address, postcode, rating, pcddata];
B = [Restaurant];
P = {Restaurant \rightarrow restaurant (Item),
    Item \rightarrow item (Name, Address, Postcode, Rating),
    Name \rightarrow name (PCData),
    Address \rightarrow address (PCData),
    Postcode \rightarrow postcode (PCData),
    Rating \rightarrow rating (PCData),
    PCData \rightarrow pcddata (e),
}

Figure 4.5: Example of CL Data and its MOT Representation

<Cinema>
  <Item>
    <Name> Hoyts East Gardens</Name>
    <Location>152 Bunnerong Rd</Location>
    <Postcode>2036</Postcode>
  </Item>
  <Item>
    <Name> Ritz Randwick</Name>
    <Location>39-47 St. Pauls Street</Location>
    <Postcode>2033</Postcode>
  </Item>
  ...
</Cinema>

N = [Cinema, Item, Name, Location, Postcode, PCData];
T = [cinema, item, name, location, postcode, pcddata];
B = [Cinema];
P = {Cinema \rightarrow cinema (Item),
    Item \rightarrow item (Name, Location, Postcode),
    Name \rightarrow name (PCData),
    Location \rightarrow location (PCData),
    Postcode \rightarrow postcode (PCData),
    PCData \rightarrow pcddata (e),
}
wrapper for simple data. For example, a simple text string “Grotta Carpi” can be mapped into MOT as follows:

\[
\begin{align*}
N &= \{\text{SimpleType}, \text{PCData}\}; \\
T &= \{\text{simpletype}, \text{pcdata}\}; \\
B &= \{\text{SimpleType}\}; \\
P &= \{\text{SimpleType} \rightarrow \text{simpletype (PCData)}, \text{PCData} \rightarrow \text{pcdata(e)}\}
\end{align*}
\]

### 4.4.6 MashSheet Cell Value

MashSheet cell values \((V)\) include simple values \((S_v)\) and complex values \((C_v)\). Simple values \((S_v)\) include number, boolean, and string. Complex values \((C_v)\) include XML data. In order to display complex value in MashSheet cell, a representation of the value is presented. For example, a string representing the name of service is used in a cell containing an \(SC^*\).

### 4.5 MashSheet’s Operators

Inspired by mashups operators provided in existing frameworks, we believe that MashSheet operators need to be in three categories: data mashups, process mashups and visualisation. Data mashup operators allow users to retrieve data from sources, to process, mix and publish data as another data source. Process mashup operators let users to automate process by orchestrating services. Visualisation operators enable users to visualise data by different views.

MashSheet provides operators \((\omega)\) that support data, process (orchestration of services) and visualisation operations in a mashup application. They are implemented as spreadsheet functions and can be used in spreadsheet formulas \((F)\):

\[
<\text{formula}> ::= \text{Blank} | <\text{expr}>
\]

The operators can be classified into four main categories: life cycle, process, data and visualisation:

\[
<\text{expr}> ::= \text{Constant} | <\text{ref}> | <\text{lExpr}> | <\text{cExpr}> | <\text{dExpr}> | <\text{pExpr}>
\]
4.5. MashSheet’s Operators

In this section, we introduce these operators with syntaxes, semantics and examples.

4.5.1 Life Cycle Operators

MashSheet has three life cycle operators for the purpose of managing \( \text{SC} \) and \( \text{SC}^* \) in the application: component creation, component deletion, service binding:

\[
<lExpr> ::= <\text{compCreate}> | <\text{compDel}> | <\text{bind}>
\] (4.3)

Component Creation

Component creation (denoted \( \text{create}() \)) registers a new \( \text{SC} \) to the component repository by importing the service interface referenced. The syntax is defined as follows, where \( \text{URL} \) is uniform resource locator of a Web service and \( \text{alias\_name} \) is a user defined name of the component representing the service. The \( \text{alias\_name} \) must be unique in the component repository and the system will detect the \( \text{URL} \) to make sure that the user-entered URL is in corrected format (e.g., WSDL document of a service).

\[
<\text{compCreate}> ::= \text{create}(<\text{alias\_name}>; <\text{URL}>)
\] (4.4)

\[
<\text{URL}> ::= \text{Constant}
\]

\[
<\text{alias\_name}> ::= \text{Constant}
\]

The \( \text{create}() \) operator does not produce any data value in the MashSheet grid. It only operates on the repository of components and creates a representation of a Web service to be used in MashSheet. For example, the following formula creates a component (i.e., \( \text{SC}_{DG} \)) representing the driving direction Web service to be used in the framework:

\[
\text{create}(\text{DG}; \text{http://www.ecubicle.net/driving.asmx?WSDL});
\]
4.5. MashSheet’s Operators

Component Deletion

Component deletion (denoted \texttt{delete()}) is an inverse operator of \texttt{create()}. It removes the referenced \texttt{SC} from the component repository. The syntax is defined as follows, where \texttt{alias\_name} must exist in the component repository in order to correctly run this operator:

\[
< \text{compDel} >::= \text{delete}(< \text{alias\_name} >) \quad (4.5)
\]

\[
< \text{alias\_name} >::= \text{Constant}
\]

Similar to \texttt{create()}, \texttt{delete()} does not affect the MashSheet’s cell values since it is repository management operator. For example, the following formula removes the component \texttt{SC}\_DG from the repository:

\[
\text{delete('DG')};
\]

Component Binding

Component binding (denoted \texttt{bind()}) binds a \texttt{SC} to a specific cell and creates \texttt{SC\_}. When the \texttt{SC\_} is created, cell address (\texttt{a}) is used to identify the component and its operations are accessible by using dot notation. The incentive of using \texttt{bind()} is that users can interact with a specific Web service invocation within cells. We do not provide an automatic binding mechanism of components into cells. This way, the users have some freedom in spatial arrangement of components in the grid of cells. A service can be created in one cell, and be bound to a different cell location. Users can freely arrange a group of components in the grid, according to their preferences.

The syntax of component binding operator is defined below, where \texttt{alias\_name} is the name of a component:

\[
< \text{bind} >::= \text{bind}(< \text{alias\_name} >) \quad (4.6)
\]

\[
< \text{alias\_name} >::= \text{Constant}
\]

\texttt{bind()} has one parameter which is \texttt{SC} and return \texttt{SC\_}. That is, it would be ex-
pressed as $\| (a, \text{bind(SC)}) \|_S \to SC^*$. 

For example, evaluation of the following cell binds $SC_{DG}$ to cell B3 and creates $(SC^*_{DG})$:

$$(B3, \text{bind}('DG'))$$

Thereafter, Tom can refer to the `getDR` operation of $DG$ as $B3.getDR(String, String)$. 

### 4.5.2 Process Control Operators

MashSheet has three process operators for the purpose of supporting orchestration of Web services: simple invocation, extended choice and synchronisation. They are used to support modeling basic control flow patterns \[199\] in the mashup applications.

$$< cExpr > ::= < invoke > | < extIF > | < sync >$$  \hspace{1cm} (4.7)

**Simple Invocation**

Simple invocation (denoted `invoke()`) is defined in much of the same way as other SB mashup tools, however, in MashSheet, the output is held in an $OC$ for further access and manipulation. The syntax is defined below, where $\text{ref}$ is coordinates of the cell containing a $SC^*$ (i.e., the cell has component binding of a Web service), $\text{opr}$ is the name of service’s operation that bound to the address $\text{ref}$, $\text{subExpr}$ (if exist) is the input parameter of the $\text{opr}$.

$$< invoke > ::= \text{invoke}(< \text{ref} > . < \text{opr} > ([< \text{subExpr} >]))$$  \hspace{1cm} (4.8)

$$< \text{ref} > ::= < \text{cellID} >$$

$$< \text{cellID} > ::= \text{Constant}$$

$$< \text{opr} > ::= \text{Constant}$$

$$< \text{subExpr} > ::= \text{Constant} | < \text{ref} >$$
invoke() allows users to invoke asynchronous (e.g., one-way) or synchronous (e.g.,
request-response) operations offered by Web services. The interaction uses either
push/pull (e.g., for SOAP service) or publish/subscribe (e.g., for RSS data feeds)
mechanisms. invoke() has one parameter which is SC* and returns OC. That is,
it would be expressed as \(((a,\text{invoke}(SC^*))_s \rightarrow OC\). For example, evaluation of
the following formula invokes the getDR() operation provided by the SC^*_DG in cell B3 and
holds the output data (i.e., OC^*_DG) in cell B5:

\((B5,\text{invoke}(B3,\text{getDR}(B1,B13)));\)

**Extended Choice**

Extended choice (denoted iff()) is provided to model the exclusive choice pattern
[199] in conventional spreadsheet. Let us consider the exclusive choice case in Tom’s
scenario (Section 4.2), the pseudo code is represented as follows:

If WF =‘rain’ Then DG Else BU

The semantically correct interpretation of this situation is that there is only one
execution between DG and BU. But in conventional spreadsheets (and any other
spreadsheet mashup tools we investigated), both DG and BU will run in order for the
code/formula to be evaluated.

The syntax of iff() is defined below, where expression is a boolean expression,
invoke is a Web service operation. The part “ELSE <invoke>” can be absent. In
this case, there is only one service invocation run when the expression is evaluated
to True.

\(< extIF > ::= \text{iff} < expression > \text{then} < invoke > [\text{else} < invoke >] \quad (4.9)\)

\(< expression > ::= < subExpr > [= | != | < | > | && | || | <! subExpr >]\)

\(< subExpr > ::= \text{Constant} | < ref >\)

\(< invoke > ::= \text{INVOKE}(< ref > . < opr > ([< subExpr >]))\)
iff() has a boolean and two SC* parameters and returns OC. The evaluation of the cell containing iff() would be expressed as \((a, \text{iff}(\alpha, \text{SC}^*, \text{SC}^*)) || S \rightarrow \text{OC}\). The semantic of iff() is defined as follows:

\[
\| (a, \text{iff}(\alpha, \text{opr}_1, \text{opr}_2)) \|_S = \begin{cases} 
\| (a, \text{opr}_1) \|_S & \text{if } \alpha = \text{True} \\
\| (a, \text{opr}_2) \|_S & \text{if } \alpha = \text{False}
\end{cases}
\]

For example, Figure 4.6 illustrates the exclusive choice case in the Tom scenario. Depending on the weather condition reported by the weather forecast service (WF). If WF reports rain, DG will be run, if not the BU (Bus Route Information) will be invoked.

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotel address</td>
<td>115 Meeks St, Kingsford</td>
</tr>
<tr>
<td>bind('RL')</td>
<td>bind('DG')</td>
</tr>
<tr>
<td>bind('BU')</td>
<td>bind('WF')</td>
</tr>
<tr>
<td>invoke(A2.read())</td>
<td>invoke(B3.getForecast(B1))</td>
</tr>
</tbody>
</table>
| iff B4.weatherInfo='Rain' then
  invoke(B2.getDR(B1, A4.Restaurant[0].Item[0].Address))
else
  invoke(A3.getBU(B1, A4.Restaurant[0].Item[0].Address)) |
```

Figure 4.6: The iff() Example
User input is in cell B1; Service bindings are in cells A2:B3; Simple invocations are in cells A4:B4; Exclusive choice is in cell A5.

**Synchronisation**

Synchronisation (denoted sync()) is provided to correctly support the semantic of AND-join (synchronisation) and XOR-join (simple merge) in MashSheet [199]. The syntax of sync() is defined below, where invoke is the service’s operation provided by a SC*.

\[
< \text{sync} > ::= \text{sync}( < \text{invoke} > ; < \text{invoke} > ) +; \text{AND} | \text{XOR} \quad (4.10)
\]
sync() has parameters which are SC* and a condition indicating the semantic of sync() (i.e., AND/XOR) and returns a Boolean value (α) indicating the joining status of participating service invocations: \( \| (a, \text{sync}(\text{SC}^*, \ldots, \text{SC}^*, \alpha)) \|_S \rightarrow \alpha \). The semantic of AND-join is defined as follows:

\[
\| (a, \text{sync}(opr_1, \ldots, opr_n, AND)) \|_S = \begin{cases} 
  False & \text{if any } \| (a, opr_k) \|_S | k \in (1, n) \text{ has not evaluated} \\
  True & \text{if all } \| (a, opr_k) \|_S | k \in (1, n) \text{ has evaluated}
\end{cases}
\]

The semantic of XOR-join is defined as follows:

\[
\| (a, \text{sync}(opr_1, \ldots, opr_n, XOR)) \|_S = \begin{cases} 
  False & \text{if no } \| (a, opr_k) \|_S | k \in (1, n) \text{ has evaluated} \\
  True & \text{if any } \| (a, opr_k) \|_S | k \in (1, n) \text{ has evaluated}
\end{cases}
\]

The default timeout setting for sync() is initially set at 15000 milliseconds. This value can be changed by using the administration web pages provided for MashSheet administrator. For example, Figure 4.7 illustrates a synchronisation in the Tom’s scenario. Tom uses the RL and CL components to obtain a list of restaurants and cinemas, respectively. He combines these two lists into a single list and wants to update the combined list only when both of the lists have updated.

### 4.5.3 Data Manipulation Operators

MashSheet provides ten data operators: union(), join(), merge(), merge_field(), rename_field(), filter(), filter_field(), extract(), download() and sort() for manipulating with OC data.

\[
<dExpr> ::= <union> | <join> | <merge> | <merge_field> | <filter> | <rename_field> | <filter_field> | <extract> | <download> | <sort>
\]

(4.11)
4.5. MashSheet’s Operators

Figure 4.7: The sync() Example

User input is in cell B1; Service bindings are in cells A2:B2; sync() is in cell A3. Users check the value of A3 to see whether two invocations are synchronised or not.

The main differences of these operators in comparison with data manipulation operators in other SB mashup tools are as follows:

- Input parameters of MashSheet data operators are MOT data (e.g., XML data). Therefore, instead of mapping data into the grid before we can run the operator (like in other tools), MashSheet data operators can run directly on MOT data. The operators can perform operations either on the structure of data (e.g., merge() and filter_field()) or on the data itself (e.g., filter() and sort());

- The output of a MashSheet data operator is held in a cell for further processing by other data or visualisation operators instead of being immediately visualised in cells. These operators are motivated by the fact that intermediate data in a mashup application should be reused by others operators and visualised only when needed.

Union

Union operator (denoted union()) combines two OCs in different structures into an OC containing all the data from participating data sets. The syntax is defined below,
where \( \text{ref} \) is the address of an \( \text{OC} \):

\[
< \text{union} > ::= \text{union}(< \text{datasource} >; < \text{datasource} >) \quad (4.12)
\]

\[
< \text{datasource} > ::= <<< \text{cellID} >>>
\]

The semantic of \( \text{union}() \) is expressed as follows:

\[
\| (a, \text{union}(\text{OC}, \text{OC})) \|_{S} \rightarrow \text{OC}
\]

We denote the data set in the first \( \text{OC} \) as \( \text{OC}_1(N_1, T_1, B_1, P_1) \) and the data set in the second \( \text{OC} \) as \( \text{OC}_2(N_2, T_2, B_2, P_2) \). \( \text{union}() \) runs on \( \text{OC}_1, \text{OC}_2 \) and produces a new \( \text{OC}(N, T, B, P) \) with the MOT representation is as follows:

\[
\begin{align*}
N &= \{ \text{Union} \cup N_1 \cup N_2 \} \\
T &= \{ \text{union} \cup T_1 \cup T_2 \} \\
B &= \{ \text{Union} \} \\
P &= \{ \text{Union} \rightarrow \text{union}(B_1, B_2) \cup P_1 \cup P_2 \}
\end{align*}
\]

For example, in the following program, the formula in cell C3 combines the data produced by formulas in cells A2 and B2:

\[
\begin{align*}
(A1, \text{bind}(\text{RL})); \\
(B1, \text{bind}(\text{CL})); \\
(A2, \text{invoke}(A1.\text{read}())); \\
(B2, \text{invoke}(B1.\text{read}())); \\
(C3, \text{union}(<<<A2>>>; <<<B2>>>));
\end{align*}
\]

The \( \text{OC} \) data in cell C3 is represented as follows:

\[
<\text{Union}> \\
<\text{Restaurant}> \\
<\text{Item}> \\
<\text{Name}>Grotta Carpi</\text{Name}> \\
<\text{Address}>97-101 Anzac Pde</\text{Address}> \\
<\text{Postcode}>2033</\text{Postcode}> \\
<\text{Rating}>5</\text{Rating}> \\
</\text{Item}> \\
<\text{Item}> \\
<\text{Name}>Golden Kingdom</\text{Name}> \\
<\text{Address}>147-151 Anzac Pde</\text{Address}> \\
<\text{Postcode}>2033</\text{Postcode}> \\
<\text{Rating}>2</\text{Rating}> \\
</\text{Item}> \\
</\text{Restaurant}>
\]
Join

Join operator (denoted join()) combines elements of two OCs according to a condition. It is similar to inner join operator in SQL language. The syntax is defined below where ref is the address of an OC, expression is an expression that accepts comparison, position or XPath expressions.

\[
< \text{join} > ::= \text{join}(<\text{datasource}>; <\text{datasource}>; <\text{expression}>) \tag{4.13}
\]

\[
< \text{expression} > ::= < \text{subExpr} > [ = | \neq | < | > | \& | \| | \| \| < \text{subExpr} > ]
\]

\[
< \text{subExpr} > ::= \text{Constant} | < \text{ref} >
\]

The semantic of join() is expressed as follows:

\[
\{(a, \text{join}(OC, OC, \alpha))\} \rightarrow OC
\]

We denote the data set in the first OC as \(OC_1(N_1, T_1, B_1, P_1)\) and the data set in the second OC as \(OC_2(N_2, T_2, B_2, P_2)\). join() runs on \(OC_1, OC_2\) and produces a new \(OC(N, T, B, P)\) with the MOT representation is as follows:

\[
N = \{\text{Join} \cup N_1 \cup N_2\}
\]

\[
T = \{\text{Join} \cup T_1 \cup T_2\}
\]

\[
B = \{\text{Join}\}
\]

\[
P = \{\text{Join} \rightarrow \text{join}(B_1, B_2) \cup P_1 \cup P_2\}
\]

For example, in the Tom’s scenario, if he wants to select all restaurants and cinemas that have the postcode equal to 2033. The following program combines data in cells A2 and B2 and holds the output in cell C3:
(A1, bind('RL'));
(B1, bind('CL'));
(A2, invoke(A1.read()));
(B2, invoke(B1.read()));
(C3, join(<<A2>>;<<B2>>;A2.restaurant.item.postcode = 2033 AND B2.cinema.item.postcode = 2033));

The data in C3 is represented as:

```xml
<Join>
<Restaurant>
  <Item>
    <Name>Grotta Carpi</Name>
    <Address>97-101 Anzac Pde</Address>
    <Postcode>2033</Postcode>
    <Rating>5</Rating>
  </Item>
  <Item>
    <Name>Golden Kingdom</Name>
    <Address>147-151 Anzac Pde</Address>
    <Postcode>2033</Postcode>
    <Rating>2</Rating>
  </Item>
</Restaurant>
<Cinema>
  <Item>
    <Name>Ritz Randwick</Name>
    <Location>39-47 St.Pauls Street</Location>
    <Postcode>2033</Postcode>
  </Item>
</Cinema>
</Join>
```

**Merge**

Merge operator (denoted merge()) combines two OCs in a uniform structure into an OC containing all the data from participating OCs. The syntax is defined as follows, where ref is the address of an OC.

\[
\text{merge}() := \text{merge}(\text{<datasource>};\text{<datasource>})
\]  

(4.14)

Evaluation of the cell containing merge() is defined as follows:

\[
\|(a, \text{merge}(\text{OC}, \text{OC}))\|_s \rightarrow \text{OC}
\]

We denote the first OC as \(\text{OC}_1(N_1, T_1, B_1, P_1)\) and the second OC as \(\text{OC}_2(N_2, T_2, B_2, P_2)\). The merge() operator runs on \(\text{OC}_1, \text{OC}_2\) and produces a new OC\((N, T, B, P)\) with the
MOT representation is as follows:

\[ N = \{ N_1 \cup N_2 \} \]
\[ T = \{ T_1 \cup T_2 \} \]
\[ B = B_1 = B_2 \]
\[ P = \{ P_1 \cup P_2 \} \]

For example, let us consider the following cells. The formula in cell C3 merges the output data produced by the RL’s read() and CL’s read() operations in cell A2 and B2:

\[
\begin{align*}
(A1, \text{bind('RL')}); \\
(B1, \text{bind('CL')}); \\
(A2, \text{invoke(A1.read())}); \\
(B2, \text{invoke(B1.read())}); \\
(C3, \text{merge(<<A2>>;<<B2>>))});
\end{align*}
\]

**Merge-field**

Merge-field operator (denoted \texttt{merge-field()}) combines selected attributes in two OCs into an OC with a user-defined structure. The main difference between \texttt{merge()} and \texttt{merge-field()} are as follows.

- \texttt{merge()} combines two OCs of the same data structure, \texttt{merge-field()} does not require the two data sets to be of the same data structure.

- all data fields in the two data sets will be combined by using \texttt{merge()} operator. \texttt{merge-field()} will combine a subset of the fields in the two data sets.

The syntax is defined below, where \texttt{ref} is an address of a OC, \texttt{field} is name of the attribute that the user wants to include in the new OC. At least two fields from the two OCs need to be defined.

\[
< \texttt{merge-field} >::= \texttt{merge-field(} < \texttt{datasource} >; < \texttt{datasource} >; < \texttt{field} > + )
\]

(4.15)

Evaluation of the cell containing \texttt{merge-field()} is defined as follows:
\| (a, merge-field(OC, OC, \alpha, ..., \alpha)) \|_S \rightarrow OC

We denote the first OC as \( OC_1(N_1, T_1, B_1, P_1) \) and the second OC as \( OC_2(N_2, T_2, B_2, P_2) \). \( \text{merge-field}() \) runs on \( OC_1, OC_2 \) and produces a \( OC(N, T, B, P) \) with the MOT representation is as follows:

\[
\begin{align*}
N &= N \in \{ N_1 \cup N_2 \} \\
T &= T \in \{ T_1 \cup T_2 \} \\
B &= \{ \text{Merge-field} \} \\
P &= X \rightarrow a r, \text{ where } X \in N, a \in T, \text{ and } r \text{ is a regular expression over } N
\end{align*}
\]

For example, let us consider Tom’s scenario, where he wants to merge the names and addresses of the restaurants with the names and addresses of the cinemas to create a new data set. The following program merges data in cells A2 and B2 and puts the result in cell C3:

\[
\begin{align*}
(A1, \text{bind('RL')}); \\
(B1, \text{bind('CL')}); \\
(A2, \text{invoke(A1.read())}); \\
(B2, \text{invoke(B1.read())}); \\
(C3, \text{merge-field}(<<A2>>;<B2>>; A2.restaurant.item.name; A2.restaurant.item.address; B2.cinema.item.name; B2.cinema.item.location));
\end{align*}
\]

In the first two formulas, Tom binds RL and CL to cells A1 and B1, respectively. He then invokes the \( \text{read()} \) operations provided by RL and CL in cells A2 and B2. Now, the results of invocations are stored in cells A2 and B2 (i.e., two OCs). Finally, in the last formula, Tom combines the data of two OCs in cells A2 and B2 by using \( \text{merge-field()} \) operator in cell C3. Tom selects only a subset of fields in two data sets (i.e., name and address from the first OC and name and location from the second OC). The OC data in cell C3 is as follows:

\[
\text{<Merge-field>}
\text{<Item>}
\text{<Name>Grotta Carpi</Name>}
\text{<Address>97-101 Anzac Pde</Address>}
\text{</Item>}
\text{<Item>}
\text{<Name>Golden Kingdom</Name>}
\text{<Address>147-151 Anzac Pde</Address>}
\text{</Item>}
\text{<Item>}
\text{<Name>Hoyts East Gardens</Name>}
\text{</Item>}
\]
4.5. MashSheet’s Operators

Rename-field

Rename-field operator (denoted rename-field()) renames an attribute in a data set to a new name. The syntax is defined below where ref is the address of an OC, field is the old name of the OC’s attribute, name is the new name of the OC’s attribute. The field and name parameters must come in pair and at least one pair needs to be defined.

\[
<\text{rename-field}> ::= \text{rename-field}(<\text{datasource}>;[<\text{field}>;<\text{name}>]+)
\]

Evaluation of the cell containing rename-field() is defined as follows:

\[
||(a,\text{rename-field}(OC,\alpha,\alpha,..,\alpha,\alpha)||_S \rightarrow OC
\]

We denote the data set in OC as \(OC_1(N_1,T_1,B_1,P_1)\). We assume that a set of attributes’ old names that wants to be renamed is \(N_{\text{old}}\) and a set of new names corresponding to the old names is \(N_{\text{new}}\). \(T_{\text{old}}\) and \(T_{\text{new}}\) are sets of terminals corresponding to the nonterminals \(N_{\text{old}}\) and \(N_{\text{new}}\), respectively. rename-field() runs on \(OC_1\) and produces \(OC(N,T,B,P)\) with the MOT representation is as follows:

\[
N = \{N_1 \cup N_{\text{new}}\} - N_{\text{old}}
\]

\[
T = \{T_1 \cup T_{\text{new}}\} - T_{\text{old}}
\]

\[
S = \{\text{Rename - field}\}
\]

\[
P = X \rightarrow a r, \text{ where } X \in N_v, a \in T_v, \text{ and } r \text{ is a regular expression over } N_v
\]

For example, let us consider the result that Tom achieve after applied the merge-field() operator as shown above. Now, Tom wants to rename the attribute location to name in this data set. In the following application, the formula in cell B4 renames location to name:

\[
(A1,\text{bind}('RL'));
\]

\[
(B1,\text{bind}('CL'));
\]
(A2, invoke(A1.read()));
(B2, invoke(B1.read()));
(C3, merge-field(<A2>;<B2>;A2.restaurant.item.name;
A2.restaurant.item.address;B2.cinema.item.name;B2.cinema.item.location))
(B4, rename-field(<C3>;C3.merge-field.item.location;
C3.merge-field.item.address));

and obtains the result as follows:

```
<Merge-field>
  <Item>
    <Name>Grotta Carpi</Name>
    <Address>97-101 Anzac Pde</Address>
  </Item>
  <Item>
    <Name>Golden Kingdom</Name>
    <Address>147-151 Anzac Pde</Address>
  </Item>
  <Item>
    <Name>Noys East Gardens</Name>
    <Address>152 Bunnerong Rd</Address>
  </Item>
  <Item>
    <Name>Ritz Randwick</Name>
    <Address>39-47 St.Pauls Street</Address>
  </Item>
</Merge-field>
```

Filter

Filter operator (denoted `filter()`) selects a specific subset of entities in an OC based on a condition and places the result in a new OC. The syntax is defined below where `ref` is an address of an OC, `expression` is an expression that accepts comparisons, position or XPath expression.

```
<filter> ::= filter(<datasource>; <expression>)                         (4.17)

<expression> ::= <subExpr> [|=|!=|<|>|&|||!<subExpr>]

<subExpr> ::= Constant| <ref>
```

The semantic of `filter()` is expressed as follows:

```
\| (a, filter(OC, \alpha)) \|_S \to OC
```
We denote the data set in the original $\mathcal{O}C$ as $OC_1(N_1, T_1, B_1, P_1)$. \texttt{filter()} runs on $OC_1$ and produces $\mathcal{O}C(N, T, B, P)$ with the MOT representation is as follows:

\begin{align*}
N &= N_1 \\
T &= T_1 \\
B &= B_1 \\
P &= P_1
\end{align*}

For example, in the Tom’s scenario, he wants to obtain a list of restaurants with rating greater than 4. The following formula in cell A3 is used to filter the list of restaurants:

\begin{align*}
(A1, \text{bind('RL')}); \\
(A2, \text{invoke(A1.read())}); \\
(A3, \text{filter(<<A2>>; A2.restaurant.item.rating > 4)});
\end{align*}

and obtains the result as follows:

\begin{verbatim}
<Restaurant> \\
  <Item> \\
    <Name>Grotta Carpi</Name> \\
    <Address>97-101 Anzac Pde</Address> \\
    <Postcode>2033</Postcode> \\
    <Rating>5</Rating> \\
  </Item> \\
</Restaurant>
\end{verbatim}

Filter-field

Filter-field operator (denoted \texttt{filter-field()}) selects a specific set of attributes from an $\mathcal{O}C$ and places the result in a new $\mathcal{O}C$. There are two variants of \texttt{filter-field()}. Users can either specify the list of fields to be filtered or use XPath expression to filter the fields. The syntax is defined as follows, where \texttt{ref} is the address of an $\mathcal{O}C$, \texttt{field} is name of $\mathcal{O}C$’s field needs to be excluded from the original data set. At least one \texttt{field} must be defined in the syntax to correctly run this operator. \texttt{expression} is an expression that accepts comparisons, position or XPath expression.

\begin{align*}
< filter_field >::= & \texttt{filter-field(< datasource >;[< field >]+)} \\
< filter_field >::= & \texttt{|filter-field(< datasource >; < expression >)} \quad (4.18)
\end{align*}
Evaluation of the cell containing filter-field() is defined as follows:

\[ \|(a, \text{filter} - \text{field}(OC, \alpha, .., \alpha))\|_{S} \rightarrow OC \]

We denote the data set in the original data set as \( OC_{1}(N_1, T_1, S_1, P_1) \). filter-field runs on the \( OC_{1} \) and produces a new data set \( OC \ (N, T, B, P) \) with the MOT representation is as follows:

\[
\begin{align*}
N &= N \in N_1 \\
T &= T \in T_1 \\
B &= B_1 \\
P &= P \in P_1
\end{align*}
\]

The main difference between filter() and filter-field() is as follows. While filter() selects a subset of data values, filter-field() selects a subset of data fields.

For example, in the previous example, Tom does not want to have postcode information in the data created by RL. The formula in cell A3 is used to exclude the “postcode” field:

\[
\begin{align*}
(A1, \text{bind}(\textit{RL})); \\
(A2, \text{invoke}(A1.\text{read}())); \\
(A3, \text{filter-field}(<\text{A2}>; A2.\text{restaurant}.item.postcode));
\end{align*}
\]

and obtains the result as follows:

```
<Restaurant>
  <Item>
    <Name>Grotta Carpi</Name>
    <Address>97-101 Anzac Pde</Address>
    <Rating>5</Rating>
  </Item>
  <Item>
    <Name>Golden Kingdom</Name>
    <Address>147-151 Anzac Pde</Address>
    <Rating>2</Rating>
  </Item>
</Restaurant>
```

Extract

Extract operator (denoted extract()) selects a value of an entity in an OC based on a condition and returns the result as a simple data (e.g., text). The syntax is
defined as follows where ref is an address of a OC, expression is an expression that accepts XPath expression.

\[
<extract> ::= extract(<datasource>; <expression>) \quad (4.19)
\]

\[
<expression> ::= <subExpr> [\neq \mid < \mid \&\mid \mid \mid \mid \mid < \mid ]
\]

\[
<subExpr> ::= Constant \mid <ref>
\]

The semantic of the cell containing extract() is expressed as follows:

\[
\| (a, extract(OC, \alpha))\|_S \rightarrow \alpha
\]

The main different between extract() and filter_field() is that extract() returns simple data while filter_field() returns MOT data.

For example, in the Tom’s scenario, he wants to obtain only the name of the first restaurant in the list of restaurants. The following formula in cell A3 is used to extract the restaurant’s name in the list:

\[
(A1, bind('RL'));
(A2, invoke(A1.read()));
(A3, extract(<<A2>>; //Restaurant/Item[1]/Name/text()));
\]

and obtains the result as follows:

Grotta Carpi

Download

Download operator (denoted download()) converts data in a URL to an OC data. The syntax is defined below where URL is the uniform resource locator of a data set. Currently, MashSheet supports the download for CSV data, text data, and HTML data.

\[
<extract> ::= download(<URL>) \quad (4.20)
\]

\[
<URL> ::= Constant
\]

The semantic of the cell containing download() is expressed as follows:
For example, let us consider a scenario when the invocation result of a SOAP-based Web service is an URL to a CSV file. This scenario occurs in real-world since normally the data returned by financial services are often too big and could hardly fit into a single SOAP message. Therefore, the data are normally encapsulated into a file (e.g., csv or text) and the Web service returns a URL to this file. The example shown here will be further discussed in Chapter 7. The following formula is used to download data in the given URL and creates MOT data in cell A3:

\[
\langle A3, \text{download}(\alpha) \rangle_S \rightarrow OC
\]

The MOT data in cell A3 can be further manipulated as a non-MOT data by using the extract() operator.

Sort

Sort operator (denoted sort()) is used to select a data set in a specific order. The syntax is defined as follows, where ref is the address of an OC, subExpr is a sorting criteria, ASC or DES is sorting order (ASC for ascending or DES for descending).

\[
< \text{sort} > ::= \text{sort}(< \text{datasource }>; < \text{subExpr }>; \text{ASC}|\text{DES})
\]

Evaluation of the cell containing sort() is defined as follows:

\[
\langle (a, \text{sort}(OC, \alpha, \alpha)) \rangle_S \rightarrow OC
\]

We denote the data set in the original data set as \( OC_1(N_1, T_1, S_1, P_1) \). sort() runs on the original data set \( OC_1 \) and selects a new data set \( OC(N, T, B, P) \) with the MOT representation is as follows:

\[
N = N_1 \\
T = T_1 \\
S = S_1 \\
P = P_1
\]
4.5. MashSheet’s Operators

For example, Tom wants to sort the list of restaurants by increasing rating order. He enters the formula in cell A3 as shown in the program below:

\[
\begin{align*}
& (A1, \text{bind(‘RL’))}; \\
& (A2, \text{invoke(A1.read())}); \\
& (A3, \text{sort(<<A2>>; //Restaurant/Item/Rating; ASC))};
\end{align*}
\]

and obtains the result as follows:

\[
\text{<Restaurant>}
\text{<Item>}
\quad \text{Name} \text{Golden Kingdom}</Name>
\quad \text{Address} \text{147-151 Anzac Pde}</Address>
\quad \text{Postcode} \text{2033}</Postcode>
\quad \text{Rating} \text{2}</Rating>
\quad </Item>
\text{<Item>}
\quad \text{Name} \text{Grotta Carpi}</Name>
\quad \text{Address} \text{97-101 Anzac Pde}</Address>
\quad \text{Postcode} \text{2033}</Postcode>
\quad \text{Rating} \text{5}</Rating>
\quad </Item>
\text{</Restaurant>}
\]

4.5.4 Visualisation Operators

MashSheet provides three visualisation operators: \texttt{grid()}, \texttt{chart()} and \texttt{map()}:

\[
< pExpr > ::= < grid > | < chart > | < map > \quad (4.22)
\]

The goal of these operators is to separate the presentation layer from data layer so that users can visualise OC data using different layouts (e.g., grid, chart, and map). The separation is important, especially when users want flexibility and reusability in creating their own display of the OC data. Basically, visualisation operators map a data space into a visual space. In our context, data space is an OC and visual space is a set of cell addresses (A). It is noted that MashSheet visualisation operators do not change the cells’ values.

Grid

Grid operator (denoted \texttt{grid()}) allows users to visualise OC data in a grid of cells. There are two types of \texttt{grid()}: column and row. Column (row) grid visualises
4.5. MashSheet’s Operators

Data as a horizontal (vertical) sequence of columns (rows) where a column (row) is constructed as a vertical (horizontal) sequence of attributes in \( \mathbb{N}^* \) (\( \mathbb{N}^* \) is a subset of \( \mathbb{N} \) and is constructed by selecting all “leaf” nodes in the OC’s tree).

The syntax is defined as follows, where \( \text{datasource} \) is the address of an OC; \( \text{ref} \) is address of the top-left cell in a cell range where users want to visualise the data set. The type of grid operator is identified by the second parameter (i.e., row or column). \( \text{expression} \) is an optional parameter which is used to filter the data to be visualised (\( \text{expression} \) accepts comparisons, position or XPath expression).

\[
< \text{grid} > ::= \text{grid}(< \text{datasource} >; \text{ROW}|\text{COLUMN}; < \text{ref} >; < \text{expression} >) \tag{4.23}
\]

\[
< \text{expression} > ::= < \text{subExpr} > [=|!=|<|>|&&|||< \text{subExpr} >]
\]

\[
< \text{subExpr} > ::= \text{Constant}|< \text{ref} >
\]

Semantic of the cell containing \( \text{grid}() \) is defined as:

\[
\|(a,\text{grid}(\text{OC},\alpha,\alpha,\alpha))\|_S \rightarrow (a,\ldots,a)
\]

Visualisation will be started at the first child node of the root element in an OC. For row-grid, the children of the root are printed in the first column from the starting cell. Next, their children are indented and printed in the next column, this pattern applies to each level for all the descendants. On the other hand, text of a particular element is placed in the next column of the element. All attributes of a particular element are placed in the same row one by one while their values are printed in the next row. The visualisation of the column-grid has the same structure as the row-grid except that the next column becomes the next row and the next row becomes the next column.

For example, Figure 4.8(a) and 4.8(b) present two scenarios where Tom visualises the data produced by the RL’s \( \text{read}() \) operator in cell B1 by a range of cells which the top-left cell is cell A3.

Let us consider another example, where Tom wants to display the list of restaurants’ names in a block of cells starting from cell A4, he enters the following formula in cell A3 as shown in the following program:
Figure 4.8: Visualise RL Data by: (a) Column, (b) Row

(A1, bind('RL'));
(A2, invoke(A1.read()));
(A3, grid(<<A2>>; column; A4; //Restaurant/Item/Name/text());

The fourth parameter used in A3 is an XPath expression using to filter the data in the A2 to get a set of names.

*General design guidelines.* When the size of result generated from a grid operator affects the position of another part of the application, the general design guideline are as follows:

- If the grid operator visualise data by “row” then all parts of the application that occupy the needed position will be advanced equally by rows. The number of rows to be advanced is determined by size of data to be visualised.

- If the grid operator visualise data by “column” then all parts of the application that occupy the needed position will be advanced equally by columns. The number of columns to be advanced is determined by size of data to be visualised.

- The advancement process is automatically done by MashSheet tool. User does not need to manually move the affected part.
Chart

Chart is an important visualisation type of data in spreadsheet. However, conventional chart operators in spreadsheets cannot apply to OC data in our framework since they can only take simple data as parameter. In order to provide flexible presentation for OC data in chart layout, MashSheet introduces three operators: column, bar, and pie (denoted column, bar and pie, respectively).

The syntax is defined below, where the first datasource is the address of an OC, the second datasource and the third datasource are two subsets of OC in the first datasource that can be used as axes in the chart. The second and third datasource can be achieved by applying data manipulation operations (e.g., filter-field) to the first datasource.

\[
<\text{chart}> ::= <\text{column}> | <\text{bar}> | <\text{pie}>
\]  
\((4.24)\)

\[
<\text{column}> ::= \text{column}(<\text{datasource}>;<\text{datasource}>;<\text{datasource}>)
\]  
\((4.25)\)

\[
<\text{bar}> ::= \text{bar}(<\text{datasource}>;<\text{datasource}>;<\text{datasource}>)
\]  
\((4.26)\)

\[
<\text{pie}> ::= \text{pie}(<\text{datasource}>;<\text{datasource}>;<\text{datasource}>)
\]  
\((4.27)\)

column(), bar(), pie() do not produce any data value in the grid of cells. It produces a visualisation of data in the component explorer (i.e., GUI) of the MashSheet framework. The semantic of the cell containing either column() (or bar(), or pie()) is defined as follows:

\[
\|(a, \text{column}(OC, OC, OC))\|_S \rightarrow a
\]

\[
\|(a, \text{bar}(OC, OC, OC))\|_S \rightarrow a
\]

\[
\|(a, \text{pie}(OC, OC, OC))\|_S \rightarrow a
\]

For example, Tom wants to visualise the data obtained by the RL service. He only wants to view two attributes: name and rating. He enters the following formula in cell A5 as shown in the below program and obtains a column chart in the component explorer:
Map operator (denoted map()) lets users to visualise data in an OC by using a map interface. This operator creates a map object with points indicating the addresses. The syntax is defined as follows:

\[
< \text{map} > ::= \text{map}(< \text{address} >)
\]

\[
< \text{map} > ::= \text{map}(< \text{from} >; < \text{to} >)
\]  \hspace{1cm} (4.28)

There are two variants of the \text{map}(). If there is one argument, the location of an address will be pinned on the map. If there are two or more arguments, the driving direction from the first address to the other addresses is shown on the map.

\text{map}() does not produce any data value in the MashSheet grid. It shows a map interface in the component explorer. Evaluation of the cell containing \text{map}() is defined as follows:

\[
\| (a, \text{map}(\text{OC}[\text{OC}])) \|_S \rightarrow a
\]

Let us consider the following program, evaluation of the formula in cell B2 will obtain data produced by the RL’s read() operation in cell B1 and display it in a map interface:

(A1, bind(‘RL’));
(B1, invoke(A1.read()));
(B2, filter – field(<<B1>>;B1.restaurant.item.address));
(B2, map(B2));

The syntaxes of MashSheet operators are summarised in Table 4.1.
4.5. MashSheet’s Operators

```
(4.1)  <formula> ::=  Blank | <expr>
(4.2)  <expr> ::=  Constant | <ref> | <lExpr> | <cExpr> | <dExpr> | <pExpr>
(4.3)  <lExpr> ::=  <compCreate> | <compDel> | <bind>
(4.4)  <compCreate> ::=  create (<alias_name>; <URL>)
(4.5)  <compDel> ::=  delete (<alias_name>)
(4.6)  <bind> ::=  bind (<alias_name>)
(4.7)  <cExpr> ::=  <invoke> | <extIF> | <sync>
(4.8)  <invoke> ::=  invoke (<ref> . <opr> ([<subExpr>]))
(4.9)  <extIF> ::=  if <expression> then <invoke>
       [else <invoke>]
(4.10) <sync> ::=  sync ([<invoke>; <invoke>] +; AND | XOR)
(4.11) <dExpr> ::=  <union> | <join> | <merge> | <merge_field> |<rename_field> | <filter> | <filter_field> | <extract> |
       <download> | <sort>
(4.12) <union> ::=  union (<datasource>; <datasource>)
(4.13) <join> ::=  join (<datasource>; <datasource>; <expression>)
(4.14) <merge> ::=  merge (<datasource>; <datasource>)
(4.15) <merge_field> ::=  merge-field (<datasource>; <datasource> ; <field> +)
(4.16) <rename_field> ::=  rename-field (<datasource>; [<field> + ; <name>] +)
(4.17) <filter> ::=  filter (<datasource>; <expression>)
(4.18) <filter_field> ::=  filter-field (<datasource>; <field>+)
       | filter-field (<datasource>; <expression>)
(4.19) <extract> ::=  extract (<datasource>; <expression>)
(4.20) <download> ::=  download (<URL>)
(4.21) <sort> ::=  sort (<datasource>; <subExpr>; ASC|DEC)
(4.22) <pExpr> ::=  <grid> | <chart> | <map>
(4.23) <grid> ::=  grid (<datasource>); ROW | COLUMN;
       <ref>; <expression>)
(4.24) <chart> ::=  <column> | <bar> | <pie>
(4.25) <column> ::=  column (<datasource>; <datasource>; <datasource>)
(4.26) <bar> ::=  bar (<datasource>; <datasource>; <datasource>)
(4.27) <pie> ::=  pie (<datasource>; <datasource>; <datasource>)
(4.28) <map> ::=  map (<datasource>)
       | map (<datasource>; <datasource>)
       <URL> ::=  Constant
  <alias_name> ::=  Constant
  <ref> ::=  <cellID>
  <datasource> ::=  <cellID>
  <cellID> ::=  Constant
  <opr> ::=  Constant
  <field> ::=  Constant
  <name> ::=  Constant
  <subExpr> ::=  Constant | <ref>
  <expression> ::=  <subExpr> | =| | > | & | || | <subExpr>
```

Table 4.1: MashSheet Operators’ Syntaxes
4.6 Building MashSheet Applications

4.6.1 A Running Example

In this section, we discuss the process of building and executing a MashSheet application. MashSheet uses spreadsheet programming metaphor for its programming paradigm. MashSheet GUI comprises two modes in a single interface: development and evaluation representing for its development and evaluation environments. Users enter the development mode when they edit a cell’s formula and right after they quit the development mode (by clicking on “Eval” or “Save” button), the execution mode is triggered. Formula evaluation result can only be saved to a cell when users hit the “Save” button. The “Eval” button only evaluates the formula. If users do not save evaluation result to the cell, the result will be lost when they select a different cell. A MashSheet application is incrementally developed by entering formulas in cells step-by-step. At each step, MashSheet evaluates not only the entered formula but also formulas in the depended cells (in both control flow and data flow). Users can visualise the formula evaluations results at every step. MashSheet leverages the “lazy evaluation” mechanism of the spreadsheet paradigm to re-evaluate formulas in the cells.

In the following, we use the Tom’s scenario as our running example. Figure 4.9 shows how the Tom’s scenario can be implemented using MashSheet. First, Tom defines cell B1 as the input area for entering hotel’s address. He uses create() operator to register five components to the repository (not shown in the figure):

```plaintext
create('RL';'http://RestaurantList.com');
create('DG';'http://DrivingGuide.com');
create('CL';'http://CinemaList.com');
create('BU';'http://BusItinerary.info');
create('WF';'http://WeatherForecast.net');
```

He, then, binds five components to cells A3:B4, C3:

```plaintext
(A3,bind('RL'));
(B3,bind('DG'));
```
Figure 4.9: A running example implementing Tom’s scenario.

Note that in this figure we combine formula view and evaluation view in one worksheet. In fact, the formulas will be evaluated to concrete data in the corresponding cells. The formula view displays formula in bold text beginning with “=“ character. The evaluation view presents data in normal text.
Tom is now ready to invoke services and build the scenario:

- He invokes the read() operations provided by the RL and CL components in cells A6 and B6 respectively. He also invokes the getWeather() provided by the WF component in cell C7:

  \[
  \text{invoke}(A3.\text{read}()); \\
  \text{invoke}(C3.\text{read}()); \\
  \text{invoke}(B4.\text{getWeather}(B1));
  \]

- Tom uses a sync() formula in cell A8 to check the synchronisation status of the two read() operations provided by the RL and CL components:

  \[
  \text{sync}(\text{invoke}(A3.\text{read}());\text{invoke}(C3.\text{read}());\text{AND})
  \]

- After both components RL and CL are completed, he merges the data in cell A6 and B6 and puts the result into cell A9:

  \[
  \text{if}(A8 = \text{True}, \text{merge}(<A6>;<B6>))
  \]

- He then sorts the content of cell A9 and puts the result into cell A11. The sorting is done by the rating property (i.e., A9.Restaurant.Item.Rating) in ascending order:

  \[
  \text{sort}(<A9>;\text{//Restaurant/Item/Rating};\text{ASC})
  \]

- In cell A13, Tom visualises the data in cell A11 by a grid of cells which the top-left cell of the grid is cell A14:

  \[
  \text{grid}(<A11>;\text{ROW};A14)
  \]

- Since the data in cell A11 is MOT data type, Tom needs to extract the data so that it can be used in other services. He enters the following formula in cell C11 to get the address of the first restaurant in the list:

  \[
  \text{extract}(\text{firstRestaurant}(A11),\text{Address})
  \]

---

The evaluation result of each operation is displayed on the component explorer (Figure 4.2 part 1.3) when associated cells are clicked.
4.6. Building MashSheet Applications

(C11, extract(<<A11>>;//Restaurant/Item1/Address/text()));

- Tom enters the formula in cell A18 to obtain the direction from his hotel to the first POI in the list (in case WF reports rain, he uses DG otherwise he uses BU):

  
  
  (A18, iff C7.weatherInfo = 'Rain' then invoke(B3.getDR(B1,B14))
  else invoke(A4.getBU(B1,B14)));

- Finally, he visualises the direction produced in cell A18 in a grid of cells which the top-left cell is cell A21 by entering a formula in cell A20:

  
  
  (A20, if(B18 = 'True', grid(<<A18>>;COLUMN;A20)));

4.6.2 How to Create an Application?

To build a MashSheet application, we need to consider two aspects:

- Control flow defines the order in which the cells’ formulas are evaluated and the condition under which a certain formula may or may not be evaluated,

- Data flow defines how data are passed between the evaluations of the formulas

We observe that spreadsheet evaluation engine only depends on data flow through cell referencing mechanism. In this framework, we extend the conventional spreadsheet paradigm to support both control flow and data flow in a MashSheet application but still using spreadsheet-like language. We consider the basic control flow patterns as defined by [199] and have some development rules as follows:

**Sequence.** Sequence can be modeled in MashSheet either by spatial arrangement of cells’ formulas in a MashSheet application or by using cell referencing (the spatial arrangement of cells is considered first).

- Spatial arrangement: We define the spatial arrangement of the cell formulas as the evaluation order. Two formulas are sequentially evaluated if their cells are located in two adjacent cells. The evaluation order progresses from left to
right and from top to bottom of the cell locations. For example, the group of cells A3: B4, C3 in Figure 4.9 are sequentially executed in the order: A3 → B3 → C3 → A4 → B4.

- Cell referencing: Two formulas are also considered sequentially evaluated if they have *input-output dependency* to each other. For example, the formula in cell A6 is evaluated after the evaluation of formula in cell A3 in Figure 4.9.

**Parallel split.** Parallel split can be also modeled by either spatial arrangement of cell’s formulas or using cell referencing.

- Spatial arrangement: Two formulas are evaluated in parallel if their cells are located in two *non-adjacent cells without data dependency*. For example, cells A6 and C7 in Figure 4.9 are evaluated in parallel.

- Cell referencing: Two formulas are evaluated in parallel if they have *data dependency with the same data and have no data dependency to each other*. For example, cells A5 and B6 in Figure 4.10 are evaluated in parallel.

![Figure 4.10: Parallel Example with Input-output Dependency](image)

**Exclusive choice.** Exclusive choice can be modeled using `iff()` operator. For example, in Figure 4.9 the formula in cell A18 models an exclusive choice pattern.

**Synchronisation and Simple Merge.** Synchronisation and Simple Merge can be modeled using `sync()` operator with AND and XOR parameters, respectively. For example, the `sync()` formula in A8 performs *synchronisation* for the formulas in A6.
and \texttt{B6} are executed in parallel, whereas the \texttt{sync()} in \texttt{B18} performs \textit{simple merge} because the service invocations in \texttt{iff()} is done as \textit{exclusive choice}.

## 4.7 Related Work

In this section, we present the related work of MashSheet in the context of mashup development tools.

Research and development on building tools and methods to support mashup development are abundant. In order to ease the mashup development tasks, a variety of tools and methods have emerged. They facilitate the environments with simple, graphical or textual interfaces, technical abstractions, and sets of predefined operators. According to Fischer et al. [119], mashup tools can be classified into six categories using mashup paradigm as criteria for classification: integrated development environment, scripting language, spreadsheet, wiring/piping, programming by demonstration, and automatic creation of mashup applications. Because the landscape of mashup tools is broad, in this section we narrow the review and discussion of the related work to a few notable tools in these six categories.

**Integrated development environment.** There are a number of integrated development environments (IDE) in widespread use today, including Eclipse [17], NetBeans [46], Microsoft Visual Studio [45].

Eclipse is an open source development platform that enables building Web 2.0 applications by using Web Tools Platform (WTP) [76] or PHP Development Tools (PDT) [54]. In addition, it can also serve as a runtime framework for executing applications. According to Maresca et al. [156], Eclipse can be used to satisfy all mashup applications’ needs, allowing developers to create mashup applications in all levels of the application stack: data, process and presentation.

Microsoft Visual Studio is an IDE facilitating the entire application development process from design to deployment. When used in conjunction with .NET framework, it enables developers to create service-oriented applications without very deep technical details such as SOAP, HTTP, WSDL [177]. Mashup programming in Mi-
4.7. Related Work

Microsoft Visual Studio is done primarily by using JScript, a Microsoft’s scripting language based on the ECMAScript. Debugging capabilities are also provided for users to support the development process.

The main difference between MashSheet and these IDE tools is that MashSheet targets users who have little programming skills and technical knowledge while Eclipse and Microsoft Visual Studio are mainly used by professional programmers. These IDE tools provide unlimited capability to developers when creating mashup applications. MashSheet, however, has a limited set of mashup operators.

**Scripting language.** Scripting languages, either general purpose (e.g., Javascript [66]) or domain specific (e.g., Swashup [157], Google Mashup [29]) can be used for developing mashup applications.

JavaScript was originally created and included in Netscape Navigator Web browser version 2.0 in 1995 to support relatively simple Web content scripting and animation tasks [193]. Since then, it is common to have JavaScript in applications on the Web. Mashup APIs are generally written in JavaScript, while Web services deployed on the Web are commonly developed using Java or .NET technologies. For example, Google Maps provides JavaScript API that makes geospatial data easily accessible to users [190]. However, users have to face some problems when using this scripting language for developing mashup applications as follows:

- If mashup APIs are not provided in Javascript, users have to bridge the integration gap between Javascript and the languages used by the APIs; and
- There are some security issues when implementing mashup applications using Javascript such as cross-site scripting [209], cross-site request forgery [208]. These issues may lead to threads that make the created applications unsafe or unusable.

Swashup provides a domain specific language for mashup development focusing on description and propagation of data in mashup applications [157]. It is implemented by using Rails Web applications development framework [139] to provide some first-class language concepts such as data mediators, services APIs, protocols,
4.7. Related Work

choreographies, user interfaces [200]. Swashup can be used to develop mashup applications involving data, but it cannot handle all scenarios thus cannot be as a generic purpose mashup development framework.

The main benefit of MashSheet and other scripting based mashup environments is that applications created by these tools can be easily distributed in source code form. The ability to deliver applications in source code form makes it easier to reuse applications in whole or piecemeal fashion. In comparison with mashup development using IDEs, applications are binary files that cannot easily be reused. Unlike most other scripting-based mashup creation environments, MashSheet uses spreadsheet-like language for mashup development. The target user of scripting-based mashup tools are programmers or technical users while MashSheet targets at normal users (without programming skills). MashSheet requires users to install the plug-in in conventional spreadsheet tools (e.g., Microsoft Excel) while some scripting based mashup environments do not need to install plug-in or extension (e.g., Google Mashup Editor) in Web browsers.

Tools based on spreadsheet paradigm. SB mashup tools (e.g., StrikeIron [63], AMICO:CALC [166], Mashroom [207], and SpreadMash [143]) allow users to create mashup applications by using spreadsheet programming metaphor (e.g., using formula and cell referencing). The “mashup” between services is made by using natural data flow created by data dependencies between cells (i.e., linking one service’s output into another service’s input fields), reading/writing to a share data repository, leveraging on events, or using mashup operators [131].

StrikeIron allows users to pull data from SOAP-based Web services into Excel worksheet and create mashup applications by ‘wiring’ output data of one service with the input parameter of another service (via cell referencing). The main limitation of this tool is that it does not support all of the basic control flow patterns. In addition, data visualisation in StrikeIron is limited to conventional spreadsheet data visualisation (i.e., data are visualised by a grid of cells and a cell can only accommodate simple types such as number and string).

AMICO:CALC lets users to create mashup within Calc spreadsheet by using
AMICO\_Read and Amico\_Write functions. By using a combination of the Read and Write functions, users can model basic control flow patterns in their mashup application. The main limitation of AMICO:CALC is that it does not provide data manipulation and visualisation operators.

Mashroom relies on nested table algebra and spreadsheet programming metaphor to simplify the mashup creation process. Mashroom is not a generic mashup framework since it focuses only on data mashup and its data visualisation is limited to nested table.

In SpreadMash, Web data are modeled using entity relations, then browsed and visualised using widgets. SpreadMash, however, is limited to data importation and visualisation. It does not mention control flow and is not a generic mashup tool. Spreadator [183], as a SpreadMash’s earlier prototype, mainly provides means for accessing/manipulating Web data from spreadsheet.

MashSheet builds further on the early ideas presented in SpreadMash and Spreadator, focusing on generic applicability and mashup component composition. To the best of our knowledge, MashSheet is the first tool in this category that can provide generic applicability to different scenarios. An important note on all SB mashup tools is that there is no approach support for application source code reusing.

Tools based on the wiring/piping paradigm. Wire (or pipe)-oriented mashup tools minimise code entry problems by providing graphical building blocks (i.e., widgets) to create mashup applications. The widgets are represented for mashup components and operators while the links between them are represented for data flows. The tools in this category adopt a flow-style orchestration specification for creating mashup applications (e.g., Yahoo Pipes [80], Microsoft Popfly [44], and Marmite [214]).

All these tools run in Web browsers, and have a Web-based graphical user interface that has been implemented using the capabilities of the underlying plug-in. To our experiment, Yahoo Pipes is the easiest and the most intuitive tool. The visual capabilities of Yahoo Pipes is comprehensive, however, its usage scope is limited to processing and creating new feeds. It allows users to aggregate, manipulate, and
merge Web content in different formats such as HTML, XML, JSON, CSV, RSS, Atom, RDF, Flickr, Google Base, Yahoo Local and Yahoo Search. The output of a pipe can be exported as news feeds (e.g., RSS and Atom) or embedded into other Web sites (e.g., iGoogle and My Yahoo). Microsoft Popfly is built on top of the Silverlight Rich Internet Application platform. It provides a more comprehensive set of APIs than that is offered by other tools in the class of Web-based mashup development environments.

All these tools have two separate user interfaces: development interface and execution interface. In the development interface, the users can connect different Web services to each other by dragging the blocks representing them into the main display, drawing “wires” to connect those services, and then editing and filling in the relevant attributes (e.g., to determine which particular properties from the services will be included in the generated mashup application). In the execution mode, the created mashup application can be immediately tested inside the Web browser and the result is visible to the users. MashSheet has these two interfaces combined in a unique conventional user interface of spreadsheet. The formula view of the Mashsheet represents for development interface where users can edit the content of a cell (e.g., enter mashup formula in cell, or edit cell value). The evaluation view represents for execution environment where all cells’ formulas are evaluated. By leveraging on spreadsheet paradigm, MashSheet enables an interesting features such as immediate visual feedback of applications executions.

**Tools based on programming by demonstration.** Programming by demonstration (or by examples) mashup tools allow users to create a mashup application by applying the same programming behavior (with any necessary adjustment) on antecedent programming tasks. It enables users with little programming skills to learn from provision example and borrowing it as a “template” on their developing program. Key players in this category include Potluck [137], Karma [196], and Dapper[13]. The simplest form of programming by demonstration can be done in spreadsheet is by using “macro”. Macro is an abstraction that defines how a simple input is recorded as a sequence of actions, which can be replayed again. The most popular language used for macro in Microsoft Excel spreadsheet is called Vi-
Visual Basic for Application (VBA). VBA was pretty powerful and had the ability to do most operating system functions. However, it is mostly targets at professional programmers who are familiar with programming concepts and syntaxes.

Karma, Potluck, and Dapper are application specific systems that provide programming-by-examples languages for integrating data retrieval, modeling, cleaning, and integration. By using a progressive approach to composing data and graphical user interfaces, these tools are appealing to the non-programmer users.

**Automatic creation.** Automatic creation of mashup applications is the most advanced paradigm of mashup development. It does not require users to involve in the low-level development process. Users only need to provide high level requirements or goals of a mashup application. The mashup tool will automatically suggest or choose composition plan based on given context or semantic of application and components. For example, Matchup [126], Intel Mash Maker [114], and Ngu et al. [164].

The research and development activities on creating automatic mashup development environments are still on-going and have limited achievement.

### 4.8 Chapter Summary

In this chapter, we presented MashSheet, a framework extending conventional spreadsheet paradigms to facilitate Web services “mashup” in the spreadsheet environment. The users, who are familiar with spreadsheet concepts, can use pre-defined operators as spreadsheet formulas to orchestrate Web services, manipulate/visualise data to create mashup applications. The key innovation of MashSheet is a collection of operators that supports orchestrating Web services, manipulating and visualising data created by the services. MashSheet operators are classified into four main categories: *lifecycle* (e.g., *bind*), *process* (e.g., *invoke*), *data* (e.g., *merge*), and *visualisation* (e.g., *grid* and *chart*). The life cycle operators are used for managing the representations of Web services within the framework. The process operators underpin coordination of Web services. The data operators are used for manipulat-
ing the data generated by Web services’ invocations. The goal of the visualisation operators is to separate the presentation layer from data layer so that users can visualise the same data using different layouts, without having to re-invoke services. Our tool is unique in that:

- The syntax of the operators is based on a paradigm that many users are already familiar with;

- The operators support basic control and data flow (rather than simple sequences); and

- There is a separation of data and presentation for flexibility

In the next chapter, we will present how to facilitate application reuse in the MashSheet framework.
Chapter 5

Reusability in MashSheet

In this chapter, we study the reuse issues in SB mashup tools. This chapter is organised as follows. We present an introduction to the problem of reusing mashup applications in Section 5.1. A motivating scenario is presented in Section 5.2. The related works are introduced in Section 5.8. We present MashSheet template and MashSheet template repository in Section 5.3 and Section 5.4, respectively. The processes of creating and using templates are detailed in Section 5.5 and Section 5.6. A running example is illustrated in Section 5.7. Finally, we summarise this chapter in Section 5.9.

5.1 Introduction

Mashup development is a knowledge intensive task. Being able to leverage on the services’ meta-data and already-built composition logics when building new mashup applications can significantly improve the performance and user experience [92]. Users of a mashup tool should be able to transfer and reuse knowledge attained after each mashup development scenario.

However, so far, the collective efforts into SB mashup have been focused on providing a basic service composition and execution solution. We argue that, for a SB mashup environment to mature, we need a way of discovering and reusing already-built mashup applications within the spreadsheet environment. After all, the idea of reuse is an important and integral part of the mashup initiation.
5.1.1 Research Questions and Challenges

The following are research questions for facilitating the reuse of applications in SB mashups:

- **What is a “reusable” mashup application in the spreadsheet paradigm?**
  When it comes to mashup development using the spreadsheet paradigm, there is a problem arising from the fact that we have no good definition for the boundary of a mashup application (i.e., input and output) laid out on a spreadsheet. This makes it difficult to describe clearly which part(s) of a worksheet are intended for reuse. The spreadsheet paradigm does not impose any constraint on the spatial layout of data.

- **How to design a repository of applications in the spreadsheet paradigm?**
  It is important to provide an approach to build an extensible and shared repository of already-built mashup applications. The relationships between applications, users and requirements (e.g., input and output constraints) must be managed within the spreadsheet environment. And the repository should provide users a low cost and rapid accessibility to the applications.

- **How to assist users during the reuse process in the spreadsheet paradigm?**
  Since the target users of the SB mashup tools are required a minimum amount of knowledge and skills, the process of creating a reusable application should be simple and easy to initiate and complete the tasks. The complexity should be hidden from the users as much as possible. Also, having a simple means for discovering reusable applications would be desirable as well.

5.1.2 Contributions

Although the idea of reuse has already existed in spreadsheets (e.g., [83, 140, 115], and [127]), the mashup tools in this area (e.g., StrikeIron [63], AMICO:CALC [166], SpreadMash [143], and Mashroom [207]) have not been able to provide solutions yet.

We propose a reuse model to facilitate reuse activities in MashSheet. Specifically,
we make the following contributions as extensions to MashSheet:

- We conceptually define a reusable MashSheet application (a.k.a, MashSheet template) by using meta-data relating to the application (e.g., input, output, and creator). The templates act as documents to describe the functionality of a mashup application without referring to any concrete data values. This means that templates are used to keep the code of applications, and can be applied to different data sets. We propose MashSheet Template Definition Language (i.e., MTDL), an XML-based language, for modeling MashSheet templates;

- We design a repository of MashSheet templates and define operators for manipulating templates in the repository. The operators have spreadsheet-like syntaxes that make it easy for users to use them in the MashSheet framework;

- We define the process of creating, instantiating, and executing MashSheet templates in reused applications. We extend MashSheet language proposed in Chapter 4 to include MashSheet template management. To the best of our knowledge, MashSheet is the first SB mashup framework which incorporates reuse.

## 5.2 Reference Scenario

Let us consider a scenario where Tim, a sales manager, wants to create a mashup application for managing purchase orders. Instead of developing the application from scratch, which may be labor-intensive, time consuming and error-prone, he searches for existing mashup applications that may provide a full or partial solution. In his PO - Management application, Tim needs the following:

- Calculating the discount rate granted to customers depending on their credit history;

- Checking the inventory for the availability of items in the order; and

- Converting the value for different currency types
Let us also assume that the above three functions are already developed by his colleagues at some points as individual MashSheet applications and available for reuse. Tim builds his application using the existing applications as templates as follows:

- **Discount – Rate**:
  - Input: CustomerID (text)
  - Output: DiscountRate (number)

- **Inventory – Check**:
  - Input: ItemID (text), Quantity (number)
  - Output: Availability (boolean)

- **Currency – Conversion**:
  - Input: Source-currency (text), Target-currency (text), Amount (number)
  - Output: Amount-in-target-currency (number)

Figure 5.1 illustrates the application scenario. In PO – Management application (i.e., the first worksheet), the values in columns D and H are from the outputs of Inventory – Check and Discount – Rate applications (i.e., the second and the third worksheet), respectively. The values in column J are achieved from the output of the Currency – Conversion application (i.e., the fourth worksheet).

We will show how to realise this scenario throughout this chapter.

## 5.3 MashSheet Template

Leveraging on the definition of a MashSheet application in the previous chapter (i.e., Section 4.4), in this section, we conceptually define a MashSheet template.

### 5.3.1 Definition

Given an existing MashSheet application (S), a MashSheet template (T) is defined as a tuple: \( (Name, Location, Creators, Users, In, Out, Area, Tags) \), where:

- **Name**: A unique name for the template.
- **Location**: The location where the template is stored.
- **Creators**: The creators of the template.
- **Users**: The users of the template.
- **In**: The inputs of the template.
- **Out**: The outputs of the template.
- **Area**: The area or domain of the template.
- **Tags**: The tags associated with the template.
5.3. MashSheet Template

Figure 5.1: The Tim’s Scenario

- **Name** is user-defined name for $T$,

- **Location** is unified resource locator for $T$,

- **Creators** is a list of users who are granted permissions as the creator of $T$ (i.e., can publish, find, load, and unpublish $T$),

- **Users** is a list of users who are granted access permissions (i.e., can find and load $T$). The creator of a template creates a list of users that he wants to share the application. Only users in the list can find and load the template. A wildcard (e.g., *) can be used to indicate that the template can be used by everyone,

- **In** is a collection of cell addresses (a) in $S$ indicating the inputs of $T$,

- **Out** is a cell address (a) in $S$ indicating the output of $T$. We define **Out** as a single output since the return value of a MashSheet template is a single MOT data only. This complies with the existing design of spreadsheet that evaluation result of a formula in spreadsheet cannot be rendered in multiple cells,
• **Area** is a quadruple \((\text{startRow}, \text{endRow}, \text{startCol}, \text{endCol})\) identifying the boundary of an area in \(S\) intended for reuse. Cells in the rectangular area defined by **Area** are identified with the following information:

  - **Formulas** is a list of tuples \((\text{Coordinates}, \text{Formula})\) representing a list of MashSheet cells’ formulas \((f)\) and corresponding addresses \((a)\). A “special” cell address named \text{float} is represented for a formula that does not belong to any cell (i.e., formula contains a life-cycle operator),

  - **Values** is a list of tuples \((\text{Coordinates}, \text{Value})\) representing a list of cells’ values \((\alpha)\) and corresponding addresses \((a)\). It is noted that only simple cell values \((\alpha)\) are stored in \(T\). MOT values (i.e., SC and OC) and conventional cell formula (e.g., \text{SUM}(A1:A3)) are not stored in \(T\),

• **Tags** is user-defined tag names for \(T\).

MashSheet templates act as documents to describe the functionality of a MashSheet application, similar to defining a function with input and output, without making references to concrete data sets in the application. A template contains a piece of code expressed in terms of MashSheet language, and it can be applied to different data values, other than the ones used in the original application. A MashSheet template is physically saved as an XML document with the file name determined by the template name and extension “.mtdl” (e.g., Currency – Conversion.mtdl).

### 5.3.2 Example

Figure 5.2 illustrates the template representation for the Currency – Conversion application. The cell range \(A1:B5\) is defined as a reusable area in which cell values and MashSheet formulas are saved for reuse. Inputs of the template are values of cell \(B1:B3\). Output of the template is the value of cell \(A5\).

The MashSheet template \(T_{\text{Currency–Conversion}} = (\text{Name}, \text{Creators}, \text{Users}, \text{In}, \text{Out}, \text{Area}, \text{Tags})\), where:

• **Name**: Currency-Conversion
5.3. MashSheet Template

Figure 5.2: A MashSheet Template Created from Currency-Conversion Application

- **Location:**
  
  http://mashsheet.cse.unsw.edu.au:8080/Library/Currency-Conversion.mtdl

- **Creators:** {tim}

- **Users:** {tom}

- **In:** {B1, B2, B3}

- **Out:** {A5}

- **Area:** {1,5,1,2}

  - **Formulas:**
    
    - (float, create(currency,'http://www.xignite.com/xCurrencies.asmx
      ?WSDL'));
    
    - (A4, bind(currency));
    
    - (A5, invoke(currency.ConvertRealTimeValue(B1,B2,B3)))}

- **Value:** {{A1, Input currency}; (A2, Output currency); (A3, Amount)}

- **Tags:** {Financial, Exchange rate}

5.3.3 MTDL

To properly represent the MashSheet templates, we propose the MashSheet Template Definition Language (MTDL), an XML-based declarative language. More detail of MTDL specification is presented in Appendix A.
5.4. Template Repository

The following document illustrates the definition for the Currency-Conversion template in the Currency-Conversion.mtdl file:

```xml
<mtdl xmlns="http://mashsheet.cse.unsw.edu.au:8080/MashSheet"
version="1.0"
><name>Currency-Conversion</name>
<location>http://mashsheet.cse.unsw.edu.au:8080/Library/Currency-Conversion.mtdl</location>
<user login="tim" role="creator" />
<user login="tom" role="user" />
<cell id="B1" type="input" />
<cell id="B2" type="input" />
<cell id="B3" type="input" />
<cell id="A5" type="output" />
<area startRow="1" endRow="5" startCol="1" endCol="2" />
<formula id="null" value="create(currency,
http://www.xignite.com/xCurrencies.asmx?WSDL)" />
$formula id="A4" value="bind(currency)" />
<formula id="A5" value="invoke(currency.ConvertRealTimeValue(B1;B2;B3))" />
$value id="A1" text="Input currency" />
$value id="A2" text="Output currency" />
$value id="A3" text="Amount" />
<tag name="Financial" />
<tag name="Exchange rate" />
</mtdl>
```

The MTDL document is automatically generated by MashSheet when user creates a template. This means that if the Currency-Conversion application is used and published by different users, different versions of the MTDL file will be created for each user. Each version of MTDL file may be different because users may have different setting for template’s meta-data (e.g., creators, users).

5.4 Template Repository

MashSheet templates need to have a mechanism to enable discovery and maintainability. In this section we define a repository that can be used to manage and distribute MashSheet templates.

UDDI is an XML-based registry catalog for SOAP services. It provides data and meta-data about services and mechanisms to classify, catalog and manage them [203]. Similar to UDDI, MashSheet template repository enables discovery and consumption of existing MashSheet applications. UDDI enables the management of service registry in two different ways: centralized and distributed, MashSheet template repository only supports centralized management. Centralized management means that all service (or template) registration information are stored and managed in a centralized way by means of CRUD (Create, Read, Update, Delete) operations.
(i.e., publish, unpublish and update service (or template) descriptions). Distributed service management means that there are different service registries and they share the service information to each other in the “read-only” manner. The centralized option provides an efficient way of managing services (or templates) while the distributed option provides more interoperability between service registries.

5.4.1 Definition

MashSheet template repository is an application accessible over a network, with which the users can publish, find, load, and unpublish templates. The repository stores template files and contains a database for managing templates’ data. The database provides information basing on three types: user data, service data and meta data. User data consists of information on the creators and users of templates. Service data includes Web services’ names, URLs using in the templates. Meta data includes tagging data and information related to location of template files (i.e., *.mtdl files).

Figure 5.3 illustrates the data model of the repository. A user is identified by an identifier (i.e., ID), name and password. A template is identified by ID, name and location of the template in the server. A user may have many user templates and a template may belong to many user’s templates. Each user’s template is identified by its ID and a user’s role (i.e., the permission that is given to the user to access the user’s template). A template may contain many services. A service is identified by its ID, name and URL. Each template also have many tags, and each tag is identified by its ID and name.

5.4.2 Operators

We define five operators for manipulating MashSheet templates in the repository. Users can either enter operators with the syntaxes defined below or use graphical interface to build and evaluate the formulas.

- Publish operator is used to create a template from an existing application
with the syntax as follows\textsuperscript{1}:

\begin{verbatim}
publish (name : text; creator : text; [user : text]; [input : text]; output : text;

top – left : text; bottom – right : text; [tag : text]);
\end{verbatim}  \hspace{0.5cm} (5.1)

where:

- \texttt{name: text} is a mandatory parameter indicating name of the template,
- \texttt{creator: text} is a mandatory parameter indicating a username who has full permissions to the template (i.e, \texttt{publish}, \texttt{find}, \texttt{load}, \texttt{unpublish}),
- \texttt{user: text} is an optional parameter indicating a username who has \texttt{find} and \texttt{load} permissions to the template.
- \texttt{input: text} is a mandatory parameter indicating the cell address used as input of the template.
- \texttt{output: text} is a mandatory parameter indicating the cell address used as output of the template.

\textsuperscript{1}The square bracket ([]) indicates that the parameter can be repeated to define multiple inputs.
- top-left:\textit{text} is a mandatory parameter indicating the top-left cell address of the tabular area in the application that intended for reuse,
- bottom-right:\textit{text} is mandatory parameter indicating the bottom-right cell address of the tabular area in the application that intended for reuse,
- tag:\textit{text} is optional parameter indicating the annotation of the template.

\textit{Example:} the following formula creates \textit{Currency – Conversion} template:

\begin{verbatim}
publish(name:Currency-Conversion; creator:tim; user:tom; input:B1; input:B2; input:B3; output:A5; top - left:A1; bottom - right:B6; tag:Financial; tag:ExchangeRate );
\end{verbatim}

- \textbf{Find} operator is used for various inquiry purposes:

  - Find by the creator name:

    \begin{verbatim}
    find (user : text);
    \end{verbatim}

    For example, the following formula lists all templates created by Tim:

    \begin{verbatim}
    find(user : Tim);
    \end{verbatim}

  - Find by service names:

    \begin{verbatim}
    find ([service : text]);
    \end{verbatim}

    For example, the following formula lists all templates that contain the service named \textit{Currency}:

    \begin{verbatim}
    find(service : Currency);
    \end{verbatim}

  - Find by tags:

    \begin{verbatim}
    find ([tag : text]);
    \end{verbatim}

    For example, the following formula lists all templates that contain two tags “Financial” and ”ExchangeRate”:

    \begin{verbatim}
    find(tag : Financial; tag : ExchangeRate);
    \end{verbatim}
– Any combination of above criteria:

\[ \text{find} \ (user : text; [service : text]; [tag : text]); \]  

(5.2)

The \text{find} \ operator includes sophisticated boolean operation (e.g., nested AND, OR, XOR, and NOT logical operator), however, the current implementation only supports logical AND.

• **Load** operator is used for loading a template into the MashSheet tool:

\[ \text{load}(template : text); \]  

(5.3)

For example, the following formula loads the Currency-Conversion template into MashSheet tool to create Currency-Conversion application:

\[ \text{load}(template : \text{Currency – Conversion}); \]

• **Execute** operator is used for calling the application execution of an already loaded template:

\[ \text{template}(name; [input]); \]  

(5.4)

where \text{name} is the name of a template, \text{input} is a cell reference to a cell value that can be used as input parameter to the template. For example, the following formula in cell J8 executes the Currency-Conversion with parameters in cells K5, K6, I8:

\[ (J8, \text{template}(\text{Currency – Conversion};<<K5>>;<<K6>>;<<I8>>)); \]

• **Unpublish** operator is used to remove a template from the repository:

\[ \text{unpublish} \ (template : text); \]  

(5.5)

For example, the following formula removes Currency-Conversion template:

\[ \text{unpublish}(template : \text{Currency – Conversion}); \]

The syntax definitions of MashSheet template repository operators are summarised in Table 5.1.
5.5 Template Creation

Figure 5.4 illustrates the process of publishing a template into the template repository:

- Firstly, a user has existing MashSheet application and wants to reuse the code. He reviews and revises (if needed) the application;
- Secondly, the user can either enters formula in the formula input box (using publish operator) or uses graphical interface to create and publish template.

From the system point of view, the template creation process contains two sub-processes:

- Creating template file in a network folder by using MTDL; and
- Inserting template data into the repository’s database
It is noted that the template creation process conforms to the following grounding rules:

- All the MashSheet cells’ formulas ($f$) are saved to the template;
- All simple cells’ values ($\alpha$) are saved to the template;
- MOT cells’ values and spreadsheet cells’ formulas (e.g., $\text{SUM}(A1 : A3)$) are ignored and not saved to the template; and
- Formatting properties (e.g., color, border, font) are not saved to the template.

Database manipulation. The process of publishing a template also contains repository’s database manipulations. The repository database (shown in Figure 5.3) is exposed as data service (i.e., REST) so that users can run CRUD (i.e., Create, Read, Update, Delete) operations without the need to set up database connection.

When a user publishes a template, the following tables will be inserted with new records: Template, User_Template, Service, and Tag. When the user unpublishes a template, the corresponding record will be removed from table User_Template.

5.6 Template Usage

Figure 5.5 illustrates the process of using a MashSheet template:
5.6. Template Usage

- Firstly, the user finds a suitable template in the MashSheet repository;
- Secondly, the user loads the template into MashSheet to create an application;
- Thirdly, the user can preview/edit the application and link with other application(s); and
- Finally, the user executes the application.

5.6.1 Find

We formulate the template finding problem in MashSheet as follows: Let \( W = \{w_1, w_2, \ldots, w_m\} \) be a set of Web services. Let \( T = \{t_1, t_2, \ldots, t_n\} \) be a set of templates, where each \( t_i \) contains a set of Web services \( w_{s_i} \) such that \( w_{s_i} \subseteq W \). Each \( t_i \) has been created by a creator \( name_i \) and has a set of tags \( tag_i (tag_i \subseteq G, G = \{g_1, g_2, \ldots, g_k\} \) is a set of tag names). Template finding task with the user request \( U \) is defined as follows:

- Given a username (\( UNAME \)) in \( U \), find all templates \( t_i \in T \) so that \( name_i = UNAME \). For example, find all templates which names similar to “Currency-Conversion”;
- Given a set of Web services (\( WS, WS \subseteq W \)) in \( U \), find all templates \( t_i \in T \) so that \( WS \subseteq w_{s_i} \). For example, find all templates which contains “Google Maps” and Xignite’s Currency web; and
- Given a set of tags (\( TAG, TAG \subseteq G \)) in \( U \), find all templates \( t_i \in T \) so that \( tag_i \subseteq TAG \). For example, find all template which contains tag names: “Financial” and “Convert”.

5.6.2 Load

LOAD is an inverse function of SAVE. It loads data from a template into the MashSheet tool to create application. Given a template name, MashSheet locates, gets and parses the template file (i.e., *.mtdl). For authentication purpose, it checks to see if the user has permission to load the template. If the user has permission to load
the template, MashSheet will load the data into the worksheet. A user is granted a permission to load a template when the creator of this template adds his username to the user-list of the template. A user is required to login into the MashSheet system by using the MashSheet GUI before manipulating with the templates. For each \(<Formula>\) element, it creates a MashSheet cell and adds the formula to this cell. For each \(<Value>\) element, it adds the value to corresponding spreadsheet cell.

### 5.6.3 Execute

Figure 5.6 shows a working example in Tim’s scenario. The upper worksheet

![Figure 5.6: A Running Example](image)

is the PO – Management application. Cell J8 contains an execution instance of Currency – Conversion application (i.e., lower worksheet) which converts an amount of money from one currency to another currency.

Implementation of the Currency – Conversion is as follows:
5.6. Template Usage

- The inputs are located in cells B1, B2 and B3;
- Cell A4 contains intermediate value which is a binding of a currency conversion Web service; and
- The final result (i.e., output) is located in cell A5

Notice that in the execution instance of the Currency – Conversion shown in Figure 5.6, the input values (i.e., values of cells B1, B2, and B3) are obtained from the PO – Management by using conventional cross-sheet links:

```
P0 – Management!K5;
P0 – Management!K6;
P0 – Management!I8
```

The value in cell J8 of the PO – Management is achieved from cross-sheet link of cell A5 in the Currency – Conversion. If the user changes any value in cells K5, K6, I8 of the PO – Management, new result of execution will be calculated and displayed in cell J8.

**Application definition and execution instance**

*MashSheet application definition* is abstract in the sense that it does not include any data from a specific execution instance. This means that when the user loads a template into MashSheet, the system creates an *application definition*. When there is an invocation (e.g., an application execution call in cell J8 of the PO – Management), the data are presented and the system creates an *execution instance*. Thus, the lower worksheet in the example above is shown not an application definition but an execution instance of the Currency – Conversion.

In the Tim’s scenario, if he has seven application execution calls in cell J8: J14, there will be seven execution instances of the Currency – Conversion. These execution instances are populated with real data originating from a specific site (i.e., the input cells where the application is called). These execution instances are worksheets with the same layout and formulae with the application definition but populated
with different data. For simplicity, MashSheet hides execution instances from the user since if there are many execution calls in an application it may be distracting.

**Names of execution instances**

Since there would be more than one execution instance of an application definition, there is a need to provide a mechanism for identifying each instance. We do so by naming each application execution instance by the name of its application definition plus the absolute coordinates of its execution call (in the bracket). For example, in the Tim’s scenario (Figure 5.6), the execution instances for application execution calls in cell J8, J9,..J14 will have names Currency-Conversion(PO-Management!J8), Currency-Conversion(PO-Management!J9),.., and Currency-Conversion(PO-Management!J14), respectively.

**Visibility of execution instances**

As above mentioned, we hide all execution instances (i.e., worksheets) for simplicity. However, if the user wants to debug an execution instance, he can right click on the cell that contains execution call of the application and select “Show execution instance” option. MashSheet will display the execution instance as a worksheet. For example, in our reference scenario, if Tim wants to see the execution instance of the application call in cell J8, he selects cell J8 and selects the option. The execution instance Currency-Conversion(PO-Management!J8) will be shown as a worksheet.

**Triggering mechanism**

The evaluation of each execution instance is determined by cell evaluation order of cells containing executions calls. MashSheet relies on a dependency graph of cells in worksheet to determine the evaluation order and triggers executions of instances.
5.7 A Running Example

In this section, we present the process of reusing MashSheet templates to create a MashSheet application. We use Tim’s scenario (in Section 5.2) as our running example.

As assumed in the scenario, three MashSheet applications Discount – Rate, Inventory – Check and Currency – Conversion have been developed. The implementation of these applications are as follows:

- **Currency – Conversion**:
  - Inputs are located in cells B1, B2, B3
  - Output is located in cell A5
  - MashSheet formulas in the application are as follows:
    ```javascript
    create('currency';'http://www.xignite.com/xCurrencies.asmx?WSDL';)
    (A4,bind('currency'));
    (A5,invoke(A4.ConvertRealTimeValue(B1,B2,B3)));  
    ```

- **Inventory – Check**:
  - Inputs are located in cells B1, B2
  - Output is located in cell A4
  - MashSheet formulas in the application are as follows:
    ```javascript
    create('check';'http://localcheck.net/Inventory.asmx?WSDL';)
    (A3,bind('check'));
    (A4,invoke(A3.CheckInventory(B1,B2)));  
    ```

- **Discount – Rate**:
  - Inputs are located in cells B1
  - Output is located in cell A3
  - MashSheet formulas in the application are as follows:
create('discount';http://localcheck.net/Discount.asmx?WSDL');
(A2,bind('discount'));
(A3,invoke(A2.Calculate(B1)));

We also assume that three MashSheet templates are created by authors of these applications and published in the MashSheet template repository. And Tom has been assigned permission to find and load these templates. The definitions for these template are as follows:

- **Currency – Conversion:**
  - Name: Currency-Conversion
  - Location:
    http://mashsheet.cse.unsw.edu.au:8080/Library/Currency-Conversion.mtdl
  - Creators: {tim}
  - Users: {tom}
  - In: {B1, B2, B3}
  - Out: {A5}
  - Area: {1, 5, 1, 2}
  - Formulas:
    * (float, create(currency,'http://www.xignite.com/xCurrencies.asmx
      ?WSDL'));
    * (A4, bind(currency));
    * (A5, invoke(currency.ConvertRealTimeValue(B1, B2, B3)));
  - Value: {(A1, Input currency); (A2, Output currency); (A3, Amount)}
  - Tags: {Financial, Exchange rate}

- **Inventory – Check:**
  - Name: Inventory
  - Location: http://mashsheet.cse.unsw.edu.au:8080/Library/Inventory.mtdl
  - Creators: {tim}
– Users: \{tom, mary\}
– In: \{B1, B2\}
– Out: \{A4\}
– Area: \{1, 4, 1, 2\}

* Formulas :

\[
\begin{align*}
\text{create('check';'http://localcheck.net/Inventory.asmx?WSDL');} \\
(A3, \text{bind('check')}); \\
(A4, \text{invoke(A3.CheckInventory(B1, B2))});
\end{align*}
\]

* Value: \{(A1, ItemID); (A2, Quantity)\}

– Tags: \{Management\}

• Discount – Rate:

– Name: Discount
– Location: http://mashsheet.cse.unsw.edu.au:8080/Library/Discount.mtdl
– Creators: \{tim\}
– Users: \{tom, mary\}
– In: \{B1\}
– Out: \{A3\}
– Area: \{1, 3, 1, 2\}

* Formulas :

\[
\begin{align*}
\text{create('discount';'http://localcheck.net/Discount.asmx?WSDL');} \\
(A2, \text{bind('discount')}); \\
(A3, \text{invoke(A2.Calculate(B1))});
\end{align*}
\]

* Value: \{(A1, CustomerID)\}

– Tags: \{Management, Promotion\}

Now, Tim is ready to build his PO – Management application:

• Tim loads three templates by using the load() operator:
load(template: Currency – Conversion);
load(template: Inventory);
load(template: Discount);

These operations create three worksheets that contain application definitions.

• In the PO – Management application, Tim enters the following formulas in cells D8-D14 to call application executions of the Inventory template:

  (D8, template(Inventory;<<C8>>;<<E8>>));
  (D9, template(Inventory;<<C9>>;<<E9>>));
  (D10, template(Inventory;<<C10>>;<<108>>));
  ... and ...
  (D14, template(Inventory;<<C14>>;<<E14>>));

• In the PO – Management application, Tim enters the following formulas in cells H8-H14 to call application executions of the Discount template:

  (H8, template(Discount;<<B8>>));
  (H9, template(Discount;<<B9>>));
  (H10, template(Discount;<<B10>>));
  ... and ...
  (H14, template(Discount;<<B14>>));

• In the PO – Management application, Tim enters the following formulas in cells J8-J14 to call application executions of the Currency – Conversion template:

  (J8, template(Currency – Conversion;<<K5>>;<<K6>>;<<I8>>));
  (J9, template(Currency – Conversion;<<K5>>;<<K6>>;<<I9>>));
  (J10, template(Currency – Conversion;<<K5>>;<<K6>>;<<I10>>));
  ... and ...
  (J14, template(Currency – Conversion;<<K5>>;<<K6>>;<<I14>>));

The Tim’s PO – Management is illustrated in Figure 5.6.
5.8 Related Work

To understand the importance of the problem, we review the reuse issue in both mashup tools and spreadsheets.

5.8.1 Reuse in Mashup Tools

There are mainly two forms of reusing mashup applications: data and computation. Data reuse is about making the same data sets available across multiple mashup applications. Normally, data reuse is supported via import/export functions in mashup tools. Computation reuse allows the computational logic inside a mashup application to be shared with other applications (e.g., via the use of templates).

Major mashup tools from big software vendors (e.g., Yahoo Pipes [80], IBM Mashup Center [34], and Intel Mash Maker [36]) facilitate sharing and reusing those artefacts within the user community. The shared mashup applications can be run and copied to workspaces for modification by other users.

Yahoo Pipes allows the users to save and execute “pipes” in their repository on Yahoo servers. When developing a pipe, the last widget in the widget pipeline is always the pipe output, which represents the result and can be published as a RSS feed or XML and JSON data set. Users can search for pipes created by others by using different criteria (e.g., formats, tags, sources, and modules).

In [180], Riabov et al. proposed a framework that simplifies the mashup creation process by reusing existing “pipes”. The users express their “mashup wishes” through the use of pipes’ tags. The tag taxonomy is defined in a hierarchical way and the users can refine their “wishes” by incrementally adding tags. The mashup engine will rely on the “wishes” to suggest a mashup application.

Intel Mash Maker also includes “community” features enabling the reuse of created mashup applications. When the users navigate to a Web page previously used in a mashup application created by another users, the Web browser will automatically display mashup icons that allow the users to reuse the exiting application.
5.8.2 Reuse in Spreadsheets

There are mainly two approaches to reuse data and computation in spreadsheets:

In the first approach, a spreadsheet can be seen as a template for the purpose of keeping validity of spreadsheet data. Whenever a user wants his spreadsheet to conform to a template, he can choose one in a list of templates to apply. When a template is applied to a spreadsheet, all of the user’s actions (e.g., inserting a row, deleting a column, and updating a value) will be checked to see whether or not it conforms to the template. GenCel [115] and RDF123 [127] are two examples of this approach. GenCel creates a copy of a user’s spreadsheet on its backend server and makes it synchronised with the original one (kept by the user) by triggering the back-end server to load the instance whenever the user enables Gencel add-in in Microsoft Excel. RDF123 transforms a spreadsheet data to a RDF graph, which is driven by a map, then publishes it online to encourage reuse.

In the second approach, a spreadsheet can be seen as a function that performs computation and generates output for given inputs. The benefit of using spreadsheet in this context is that users are allowed to directly maintain the business logic of their applications. [140] and [21] are examples in this approach. In [140], Jones et al. extend Microsoft Excel by allowing users to create user-defined functions. The main idea is that the user can save his computation in a worksheet as a function and reuse it later in other worksheets as a function call. Microsoft Excel service [21] targets the reuse of Excel computations created by users as a calculation service, component of a larger orchestration.

In this chapter, we aim at addressing the reuse issues in the MashSheet framework by using existing spreadsheet’s reuse metaphor (i.e., consider MashSheet applications as functions to be called in other applications). Existing SB mashups (e.g., StrikeIron [63], AMICO:CALC [166], Mashroom [207], and SpreadMash [143]) have not provided a solution for reusing existing mashup application. They only provide basic service mashup development and enactment solutions. Our work is the first attempt to facilitate the reuse in the same class of mashup tools (i.e., using spreadsheet paradigm).
5.9 Chapter Summary

In this chapter, we have given an overview of the reuse issue in the areas of mashup development and spreadsheet. Given that there exists various already-built mashup applications, we raised the need for (i) a model that enables to describe reusable applications, and (ii) a repository that facilitates storage and manipulation of already-built applications. We, then, have proposed a model and language for describing reusable mashup applications in the MashSheet framework. In particular, we introduced the notion of template to capture the functionality of an application without reference to any particular instance. We have proposed a language, named MTDL, as an XML-based declarative language allowing users to describe MashSheet templates. We have also presented a reference model for the MashSheet template repository and defined operators for manipulating the templates. The operators have spreadsheet-like syntaxes and can be used in the MashSheet formula editor. We have defined the processes of creating, finding, loading, and executing MashSheet templates and illustrated them in a scenario.

In the next chapter we will present the implementation of the MashSheet framework.
Chapter 6

Implementation

This chapter is devoted to the implementation of the MashSheet framework\[133\]. We implemented MashSheet as a plug-in component in Microsoft Excel. We will discuss how the same implementation of MashSheet can be realised in other spreadsheet tools (e.g., OpenOffice Calc and Gnome Gnumeric). The overview and demonstration of the system is available at: http://www.cse.unsw.edu.au/~hpaik/mashsheet.

This chapter is organised as follows. We present the design and implementation details of MashSheet in Microsoft Excel in Section 6.1. In Section 6.2, we discuss the possibility of implementing MashSheet in other spreadsheet tools. Finally, we summarise this chapter in Section 6.3.

6.1 MashSheet Implementation in MS Excel

Our implementation extends the prototype introduced in [183], which is mainly designed for importing data services into a spreadsheet. The following are the implementation choices we made about MashSheet:

Spreadsheet choice: Microsoft Excel

We chose Microsoft Excel as spreadsheet representation for the mashup framework. This choice comes from the fact that Microsoft Excel is the most popular spreadsheet tool [159]. The design of MashSheet is generic and it can be applied to several
6.1. MashSheet Implementation in MS Excel

spreadsheet tools (e.g., OpenOffice Calc, Gnome Gnumeric, or Google Spreadsheet). The generic applicability comes from the fact that MashSheet’s abstractions (e.g., application model, MashSheet Object Type) are not depending on any particular spreadsheet tool). In addition, the implementation of MashSheet as a spreadsheet’s plug-in is feasible in all spreadsheet tools. We will show how it can be done in Section 6.2 below.

Development choice: Visual Studio Tools for Office

As presented in Chapter 2, there are three common ways to extend the functionality of Microsoft Excel spreadsheet: Visual Basic for Application, COM/ Automation Add-ins, and Visual Studio Tools for Office (VSTO). We selected VSTO since it is the most evolving extensibility model and has a clear road map from Microsoft. In addition, it has maximal functionality and performance in .Net framework.

Language choice: C#

The language of choice for implementing MashSheet is C#.

Web server: Apache Tomcat.

We used Apache Tomcat Web server to host a network folder named “Library” and provides two Web pages for registering and administrating template users. Apache Axis is also installed to provide the server-side infrastructure for deploying and managing Web services and the client-side API for invoking Web services. Both these tools are open-source products and available for download.

Database server: PostgreSQL

We adopted PostgreSQL as database management server to store all the template information. The template repository database is exposed as a data service so that MashSheet can access the database via Web service tools. We use Axis2 POJO framework and Java to implement the data service.
Table 6.1 summarises the products and technologies used for the implementation of MashSheet.

<table>
<thead>
<tr>
<th>Product</th>
<th>Version</th>
<th>Usage description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft Excel</td>
<td>2007</td>
<td>Spreadsheet tool for developing MashSheet</td>
</tr>
<tr>
<td>Visual Studio</td>
<td>2008</td>
<td>IDE for developing MashSheet</td>
</tr>
<tr>
<td>VSTO</td>
<td>3.0</td>
<td>Framework for developing Microsoft Office products</td>
</tr>
<tr>
<td>.NET framework</td>
<td>3.5</td>
<td>Framework allows languages interoperability</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>6.0.32</td>
<td>Web server</td>
</tr>
<tr>
<td>Axis2</td>
<td>1.6.0</td>
<td>SOAP engine</td>
</tr>
<tr>
<td>JDK</td>
<td>1.6</td>
<td>Java development environment</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>9.0</td>
<td>Database management</td>
</tr>
<tr>
<td>RSS.NET</td>
<td>1.0</td>
<td>Generic wrappers for RSS data feeds</td>
</tr>
<tr>
<td>AXIS POJO</td>
<td></td>
<td>Expose a database as a data service</td>
</tr>
</tbody>
</table>

Table 6.1: Products used for MashSheet Implementation in Microsoft Excel

6.1.1 System Architecture

Figure 6.1 depicts the high-level architecture of MashSheet. It comprises of two following parts: client-side and server-side. The client-side part is Microsoft Excel tool with MashSheet plug-in and the server-side is a Web server where all the MashSheet templates are stored.

In the client-side part, MashSheet plug-in is developed as an application-level customisation of Microsoft Excel spreadsheet. A mashup engine is developed separately from the conventional spreadsheet evaluation engine to evaluate MashSheet’s mashup operators. All events and notifications between Excel and the mashup engine are handled by the workbook model. The graphical user interface of MashSheet plug-in is shown in Figure 6.2.

There are three components in the server-side part: a database, a network folder, and a user administration Web pages. The database is implemented using PostgreSQL to store meta data of templates (e.g., creator, user, input, output, and location). We adopt Axis Pojo framework and use Java to expose the database as a REST service so that it can be accessible as a Web service. Apache Tomcat is used as Web server to host a network folder named “Library”. This network folder is used to store all the MashSheet template files. Administration Web pages are
provided allowing users to register accounts for the template repository.

MashSheet Plug-in GUI

We extend the interface of Microsoft Excel by using a Visual Studio for Office add-in project. Custom Excel functions and features are accessible through Add-ins menu in spreadsheet. The main menu of MashSheet is defined as a Task Pane in the interface. Figure 6.2 illustrates the MashSheet GUI and its components.

Mashup development tab.

- **Formula editor.** Since Microsoft Excel does not allow developers to modify its conventional evaluation engine, we decided to create a separate formula evaluator for mashup operations. Therefore, there are two kinds of formula in MashSheet:
  - Default Excel formula is used for all computation that does not involve mashup operations; and
  - MashSheet operators is used for mashup operations and template repository manipulations

- **Component explorer.** Component explorer allows users to graphically browse the content of cells. If the data in a cell is MOT data then it can be hierarchically viewed in the component explorer. We facilitate a drag and drop
Figure 6.2: MashSheet GUI on Microsoft Excel
manipulation of data from the component explorer to a grid of cells so that
users can put a data element in the tree view into a cell. Component explorer
is also used as a visualisation window for other cell’s data types. For example,
if the data in a cell is a valid URL, the component explorer will act as a Web
browser and display a Web page with the given URL. Users also view special
visualisation (e.g., map, chart) in the component explorer.

• Component repository. The component repository is used to accommodate
Web services that will be used in MashSheet. In a nutshell, the component
repository lets users to select a Web service, and then to determine a name for
that Web service that could be used in Excel. In the current implementation,
we provided adapters allowing access to Web data in three sources: RESTful
Web services, SOAP-based Web services, and RSS data feeds.

Mashup reuse tab. The graphical user interface (GUI) of MashSheet template
repository is shown in Figure 6.3. The GUI consists of four elements: Operation
selector (denoted 1), Parameter input (denoted 2), Evaluation result (denoted 3)
and Template Repository (denoted 4). In the “Operation selector” element, users
can choose one of the five template repository operators to execute. The “Parameter
input” element contains essential user input information for the operators in 1. In
the “Evaluation result” element, the execution results of the operators in 1 are
displayed. The “Template repository” element lists all MashSheet templates.

Template Repository

The repository stores template files and contains a database for managing templates’
data. The database provides information based on three types: user data, service
data and meta data. User data consists of information on the creators and users of
templates. Service data includes Web services’ names, URLs of services included in
the templates. Meta data includes tagging data and information related to locations
of template files (i.e., *.mtdl files). Figure 5.3 illustrates the data model of the
repository. A user is identified by an identifier (i.e., ID), name and password. A
template is identified by ID, name and location of the template in the server. A
Figure 6.3: MashSheet Template Repository GUI
user may have many user’s templates and a template may belong to many user’s templates. Each user’s template is identified by its ID and a user’s role (i.e., the permission that is given to the user to access the user’s template). A template may contain many services. A service is identified by its ID, name and URL. Each template also has many tags. And each tag is identified by its ID and name.

Table 6.2 shows the database design of the template repository.

<table>
<thead>
<tr>
<th>Table name</th>
<th>Field name</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users</td>
<td>UserID</td>
<td>integer</td>
<td>primary key</td>
</tr>
<tr>
<td></td>
<td>Username</td>
<td>varchar(20)</td>
<td>user login name</td>
</tr>
<tr>
<td></td>
<td>Password</td>
<td>varchar(20)</td>
<td>user login password</td>
</tr>
<tr>
<td>Service</td>
<td>ServiceID</td>
<td>varchar(100)</td>
<td>primary key</td>
</tr>
<tr>
<td></td>
<td>ServiceName</td>
<td>varchar(100)</td>
<td>name of a service</td>
</tr>
<tr>
<td></td>
<td>URL</td>
<td>varchar(200)</td>
<td>URL of the service</td>
</tr>
<tr>
<td></td>
<td>TemplateID</td>
<td>integer</td>
<td>foreign key to the Template</td>
</tr>
<tr>
<td>User_Template</td>
<td>UserTemplateID</td>
<td>varchar(100)</td>
<td>primary key</td>
</tr>
<tr>
<td></td>
<td>UserID</td>
<td>integer</td>
<td>foreign key to Users</td>
</tr>
<tr>
<td></td>
<td>TemplateID</td>
<td>integer</td>
<td>foreign key to the Template</td>
</tr>
<tr>
<td></td>
<td>UserRole</td>
<td>varchar(10)</td>
<td>“creator”/“user”</td>
</tr>
<tr>
<td>Template</td>
<td>TemplateID</td>
<td>integer</td>
<td>primary key</td>
</tr>
<tr>
<td></td>
<td>TemplateName</td>
<td>varchar(100)</td>
<td>name of a template</td>
</tr>
<tr>
<td></td>
<td>Location</td>
<td>varchar(200)</td>
<td>location of MDTL file</td>
</tr>
<tr>
<td>Tag</td>
<td>TagID</td>
<td>integer</td>
<td>primary key</td>
</tr>
<tr>
<td></td>
<td>TagName</td>
<td>varchar(100)</td>
<td>name of a tag</td>
</tr>
<tr>
<td></td>
<td>TemplateID</td>
<td>integer</td>
<td>foreign key to the Template</td>
</tr>
</tbody>
</table>

MashSheet templates are stored as MTDL files on the server. The specification of MTDL is presented in Appendix A of this dissertation.

6.1.2 Data management issues

In this section, we discuss issues relating to the data management in MashSheet.

Data storage

Conventional spreadsheets (e.g., Microsoft Office, OpenOffice Calc) have a limitation when dealing with complex data type or storing multiple values in a cell. For example, only simple data type (e.g., number, text) is allowed to store in a cell.
To overcome this limitation, we provided a solution in MashSheet to map complex data (e.g., an XML document) to a single cell (see Chapter 4). All intermediate data in MashSheet are stored locally as XML files (e.g., result of service invocation operation, result of data filtering operation). A file name is defined by address of the cell accommodating the data set. For example, result of the “Restaurant service” invocation in cell B2, worksheet Sheet1 is stored locally in the machine as %MashSheet\directory%\Data\Sheet1-B2.xml. Data in every cell is traceable by identifying the filename corresponding to the cell addresses.

There is a problem when the size of XML file (i.e., intermediate data) is large. When user changes his focus to cells so quickly, it may take time (e.g., few seconds) for the system to read the entire XML file and load it into the MashSheet GUI. This may cause user dissatisfaction since the system does not provide a good immediate response. To alleviate this situation, we provided a solution to enable a “quick view” of the XML document. Initially, a cell does not display the full content of the document. It only displays the name of root element and only after user expands the hierarchical architecture of the document in the Component Explorer, the system will load the full content of the file.

Data importation

To select data importation mechanism for MashSheet, we considered the following existing approaches:

- Dynamic Data Exchange (DDE) is a mechanism implemented in many popular programs in Windows (e.g., Microsoft Excel and Microsoft Word) for exchanging data among processes. For example, users can define a link between a cell in Excel and a value in another application. When the application causes the value changed, the cell’s value in Excel will be changed accordingly.

- Real-time Data (RTD) server is an automation server enabling communications with Microsoft Excel. The interaction between an RTD server and Excel program is done by using a hybrid push-pull mechanism (i.e., (i) When there is a change in value, the RTD server notifies Excel (push), and (ii) A request for data update from Excel causes a data update in the server (pull)).
• Microsoft Office Excel XML Tool Add-in\textsuperscript{1} \cite{181} is used to aid Excel users in working with XML data, XML list and XML maps. With it, user can map a user-specified XML schema to fields and then import or export data in spreadsheet.

• StrikeIron \textsuperscript{63} allows users to directly drag and drop Web services from multiple sources into Microsoft Excel spreadsheets. Users have to predefined the data structure in the grid before the data mapping and importation can be taken place.

We considered data importation issue in term of (i) How efficient it is when importing a large amount of data? (ii) How fast it is when we update those data? (iii) What is data refresh mechanism?

In DDE, when the number of DDE formulas in a sheet becomes very large, performance can be greatly reduced. DDE is limited to 50 items per request and maximum 50 DDE links. One of the reasons for reduced performance is that simultaneous requests will greatly increase the number of Windows messages to be transferred and each DDE link heavily consumes system resource.

On the other hand, RTD function behaves differently to other worksheet functions: its function value is updated whenever new data becomes available from the RTD server and the workbook is able to accept it (i.e., when Excel is not busy and the RTD throttle interval has elapsed). The data access in RTD is efficient because only updated values are refreshed in spreadsheet (i.e., partial updates only).

In XML mapping tool, user has to map data into the spreadsheet before the importation can be taken place. It updates the whole dataset regardless some values in that is not changed. In our experiment to map spreadsheet with an XML file, it takes a long time (about 7 minutes) to import 10,000 rows of data into spreadsheet. Evaluating performance of StrikeIron is quite different from other approaches because StrikeIron does not directly get the data from the data repository. It gets the data by invoking web services. Thus, there are many factors that may impact the performance of StrikeIron such as network throughput, number of web service invocation.

\textsuperscript{1}We will call XML Mapping tool for short
DDE and RTD cannot retrieve a collection of data in one query (i.e., it only returns one value for one formula). However, RTD server can return multiple values by combining values into one delimited string. A programmer can write a VBA wrapper function to parse the string into an array that can be put into worksheet. On the contrary, XML mapping tool and StrikeIron can provide data for a collection of cells in a single query.

In terms of data refreshment, there are two approaches: push and pull. In push approach, the server tries to push real-time data into spreadsheet. The main limitation of this approach is that it may push data into spreadsheet when spreadsheet is not ready for it (e.g., while spreadsheet is doing calculation or has a modal dialog box that need to be handled). This leads to dropped updates and even causes a crash. In pull approach, spreadsheet application pulls data from the feed. The main limitation of this approach is that they may repeatedly make request to feed even there have not been any updates so that it will cause overhead to the server. DDE and RTD use hybrid push and pull approach while the other two methods use pull mechanism (i.e., XML mapping, StrikeIron).

Table 6.3 summarises our experiment on different data importation approaches in spreadsheet.

Based on the above evaluation, we considered which approach to be used for MashSheet. We decided that the data for importation in MashSheet is not dynamic (i.e., they did not change too frequently), so we chose a solution to provide faster importation of a large amount of data rather than a solution that serves better in frequently update scenarios. We used a hybrid push-pull approach based on an RTD server.

<table>
<thead>
<tr>
<th></th>
<th>DDE</th>
<th>RTD</th>
<th>XML mapping</th>
<th>StrikeIron</th>
</tr>
</thead>
<tbody>
<tr>
<td>Performance</td>
<td>Reduce when number of DDE increased</td>
<td>Good for dynamic data</td>
<td>Good for static data</td>
<td>Depending on external factors</td>
</tr>
<tr>
<td>Update mechanism</td>
<td>Hybrid push/pull</td>
<td>Hybrid push/pull</td>
<td>Pull</td>
<td>Pull</td>
</tr>
<tr>
<td>Can return multiple value</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 6.3: Evaluation of Data Importation Approaches in Spreadsheet
Error handling

Mashups are highly distributed applications that accomplish goals by executing services provided by partners. This makes the development of mashups more flexible and modular. However, it also brings more challenges to the development and maintenance of quality mashups in practice. In a classical distributed system, correctness can be ensured by statically checking the composition of the components that make up the application against properties of interest (e.g., [89, 120, 122, 123]). However, in the case of mashups, there are various factors that makes the correctness checking process more complicated. For example, Web service’s behaviour may be changed at execution time, so statically checked properties of applications may be insufficient throughout the application’s lifetime.

In MashSheet, we considered the following problems:

- **Runtime monitoring.** Monitoring MashSheet as it runs provides a change to recover from an error once a problem has been detected. MashSheet applications are monitored for both functional (e.g., assertions) and non-functional (e.g., performance) while the applications are running. According to [188], there exists several criteria for a monitoring system: (i) On-line vs Off-line, (ii) Global vs Local, (iii) Passive vs Active. In MashSheet, we provided a simple mechanism for runtime monitoring through logging windows. We offered an off-line, global and passive solution for runtime monitoring issue. More advanced technique will be considered as the future work of the framework.

- **Error recovery.** When an error is detected at runtime, a system should try to compensate all completed activities and restart once the error has been fixed. This is done by adding more behaviours to the application and may result in a significantly bigger and slower program. Currently, we do not support error recovery mechanism in MashSheet. For any system violation, we only provide a restart option for application execution.
6.1.3 MashSheet Implementation Issues

Evaluation Engine

In the Microsoft Office 2007 products (e.g., Microsoft Excel 2007) developers are able to handle events generated in custom code by using “Event Manager”. The “Event Manager” provides mechanism to control registered events when a document (e.g., a spreadsheet worksheet) is initially loaded. This could be done via `InternalStartup()` method, and then the delegates can be constructed to encapsulate a reference to a method that handles the events’ custom code.

In MashSheet, we utilise this mechanism to create a new evaluation engine for mashups. Our evaluation engine provides the following functionalities:

- Preprocessing mashup formula on user input. This step parses formula into methods so that the mashup engine can evaluate the code. It also takes into consideration the spatial location of cells’ formulas to determine the evaluation order;

- Evaluating the statements and returning values to the cell. When a formula returns MOT data to a cell, a string representation of the MOT data is generated so that the cell still has a valid conventional Excel value (i.e., that can be used by conventional spreadsheet evaluation engine and displayed in the grid).

MashSheet Object Type

Since we cannot create a new cell data type in Microsoft Excel, we need to find a new way to define MOT data. To implement MOT, we create a “virtual” grid of cells as in conventional spreadsheets. We call cells in the virtual grid as MashSheet cells. A MashSheet cell is only created when we enter MashSheet mashup formula in a cell. For example, in a MashSheet application, we have:

\[(B3, bind('DG'))\]
The evaluation of the formula in cell B3 will create a new MashSheet cell with coordinate B3 representing the MOT data it created (i.e., SC data). Figure 6.4 illustrates the MOT representation in MashSheet as a virtual cells.

![Figure 6.4: MOT’s Implementation in MashSheet](image)

In virtual grid, the solid box represents a MashSheet cell, the dotted boxes represents blank MashSheet cells.

Each MashSheet cell contains an XML data type and can be used as primitive data type in MashSheet formulas. Most evaluation results of MashSheet operators are MOT data (except extract). However, the invocation result of a Web service is not always an XML data (i.e., the Get operation provided by a REST service could return a simple text data). In this situation, we need to serialise service output to MOT. In order to do that, we leverage on XMLSerializer class in .Net framework to convert any unpredictable object type into an XML document.

Referencing and Dependency

Similar to dependency graph in Microsoft Excel, a dependency graph of virtual cells is maintained in MashSheet to facilitate the lazy evaluation of the MashSheet cells. Each MashSheet cell will have a list of depending cells and a list of depended cells. For example, if cell A1 has a formula containing a reference to cell B7, then A1 is the depending cell of B7 while B7 is A1-depended cell. To provide a connection between the dependency graph of Microsoft Excel and the dependency graph of MashSheet,
we define an “Event Interceptor” element to propagate every change/update notification of Excel’s dependency graph to MashSheet’s dependency graph, and vice versa. This enables the transparency of two grid models.

Wrappers/Adapters

Basically, Web services allow users to access application logic and data through standard protocols (e.g., HTTP, SMTP). In terms of making distributed components, Web service is similar to DCOM\textsuperscript{2}. However, the main advantage of the Web services is that any application on any platform can access the Web services as long as it uses standard Web protocols, and understands the XML encoded messages. Here we have the mechanism to utilise different Web services in MashSheet:

- **RSS feeds.** We use an open source .NET library for RSS feeds called RSS.NET\textsuperscript{3} for generic RSS adapters.

- **SOAP-based Web services.** For SOAP services, there is no generic adapter defined. Therefore each SOAP service needs to be added to the tool we create a proxy code and object representation for it to be executable within .NET framework. There are two steps to utilise a Web service using .NET framework:
  - *Generate Web service proxy code.* In general, proxy source code is similar to the way that IDL\textsuperscript{4} compilers generate source file for DCOM proxy. Microsoft .NET SDK provides user with tools for creating and consuming Web services (i.e., \texttt{wsdl.exe}). WSDL tool takes WSDL file as an input and creates proxy for Web service’s consumer. For example, the following command creates proxy file for **Weather Forecast** service that has \texttt{wsdl} file in \texttt{Weather.wsdl}:

    \begin{verbatim}
    wsdl.exe Weather.wsdl /namespace:Weather /out:Weather.cs
    \end{verbatim}

    The file \texttt{Weather.cs} will contain proxy code for **Weather Forecast** Web service.

\textsuperscript{2}Distributed Component Object  
\textsuperscript{3}Available in http://rssdotnet.com  
\textsuperscript{4}Interface Description Language
6.2. Discussion

- *Utilise.* After the proxy file is generated, it will be utilised as a library function. The utilising step comprises of compiling and creating DLL\(^5\) file. Users then can refer to it as a DLL library in their applications. At first, the process of creating DLL file can be done by using the *csc* command. For example, the following command will generate *Weather.dll* from the proxy file:

\[
\text{csc /target:library Weather.cs}
\]

- **RESTful Web services.** For REST services, in current implementation we provides generic wrapper for the *Get* operation. The wrappers for *Post*, *Put*, *Delete* are left for later development phase.

### Visualisation Libraries

We use third party libraries for implementing functionalities of two visualisation operators: Google Map API\(^6\) for *map()* and Raphael Javascript Library\(^7\) for *chart().*

### 6.2 Discussion

The whole design behind MashSheet is generic and can be applied to other spreadsheet applications. In this section, we discuss the possibility of implementing MashSheet in other spreadsheet tools.

#### 6.2.1 OpenOffice Calc

The design of MashSheet framework can be implemented in OpenOffice Calc spreadsheet by using OpenOffice.org API plug-in for NetBeans\([52]\). OpenOffice.org API plug-in for NetBeans, developed by Sun Microsystems, allows developers to create OpenOffice.org extensions through four types of extension: Add-ons, Calc Add-ins, Components and Client Applications. The extensions are developed as Java ap-

---

\(^5\)Dynamic Link Library

\(^6\)http://code.google.com/apis/maps/index.html

\(^7\)http://raphaeljs.com/
applications and managed by Extension Manager. They can be used directly in the normal user interface of the OpenOffice products.

Table 6.4 summarises the recommended products and technologies that can be used for the implementation of MashSheet in OpenOffice Calc.

<table>
<thead>
<tr>
<th>Product</th>
<th>Version</th>
<th>Usage description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenOffice Calc</td>
<td>2.3</td>
<td>Spreadsheet tool for developing MashSheet</td>
</tr>
<tr>
<td>NetBeans</td>
<td>5.5.1</td>
<td>IDE for developing MashSheet</td>
</tr>
<tr>
<td>OpenOffice SDK</td>
<td>2.3</td>
<td>Framework for developing OpenOffice</td>
</tr>
<tr>
<td>OpenOffice API</td>
<td>3.5</td>
<td>Framework allows developing OpenOffice extension using NetBeans</td>
</tr>
<tr>
<td>plug-ins for NetBeans</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>6.0.32</td>
<td>Web server</td>
</tr>
<tr>
<td>Axis2</td>
<td>1.6.0</td>
<td>SOAP engine</td>
</tr>
<tr>
<td>JDK</td>
<td>1.6</td>
<td>Java</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>9.0</td>
<td>Database management</td>
</tr>
<tr>
<td>AXIS POJO</td>
<td></td>
<td>Allow exposing a database as a data service</td>
</tr>
</tbody>
</table>

Table 6.4: Products recommended for MashSheet Implementation in OpenOffice Calc

<table>
<thead>
<tr>
<th>Product</th>
<th>Version</th>
<th>Usage description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gnome Gnumeric</td>
<td>1.10.16</td>
<td>Spreadsheet tool for developing MashSheet</td>
</tr>
<tr>
<td>PyGTK</td>
<td>2.24.0</td>
<td>IDE for developing Python applications</td>
</tr>
<tr>
<td>GNOME Structured File Library (libgsf)</td>
<td>1.14.19</td>
<td>I/O abstraction for reading/writing compound files</td>
</tr>
<tr>
<td>Gnome Application Library (libgal)</td>
<td>2.4.3</td>
<td>A collection of widgets and other helper functions</td>
</tr>
<tr>
<td>Python plug-in for Gnumeric</td>
<td>xx</td>
<td>Framework allows developing extension</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>6.0.32</td>
<td>Web server</td>
</tr>
<tr>
<td>Axis2</td>
<td>1.6.0</td>
<td>SOAP engine</td>
</tr>
<tr>
<td>JDK</td>
<td>1.6</td>
<td>Java</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>9.0</td>
<td>Database management</td>
</tr>
<tr>
<td>AXIS POJO</td>
<td></td>
<td>Expose a database as a data service</td>
</tr>
</tbody>
</table>

Table 6.5: Products recommended for MashSheet Implementation in Gnome Gnumeric

6.2.2 Gnome Gnumeric

MashSheet can be also implemented in Gnumeric spreadsheet by using Python and C. Table 6.5 summarises the recommended products and technologies that can be used for the implementation of MashSheet in Gnome Gnumeric.
6.3 Chapter Summary

In this chapter, we have presented the implementation of MashSheet. We have discussed in details the design and implementation of MashSheet in Microsoft Excel, including the supporting libraries and tools chosen. We have also proposed the possibility of implementing MashSheet in other spreadsheet tools to prove the generic applicability of our design.

In the next chapter we will present the evaluation of MashSheet.
Chapter 7

Evaluation

In this chapter, we present the three-way evaluation of the MashSheet framework.

Firstly, in Section 7.1, we apply MashSheet to case studies in various application domains to assess whether the design of the tool can adequately support mashup development and investigate any issue. Secondly, in Section 7.2, we present an evaluation on the expressiveness of the MashSheet language. Thirdly, in Section 7.3, we put forward a user study to evaluate the usability of the framework. Finally, a summary of the chapter is presented in Section 7.4.

7.1 MashSheet Applications

We will present four scenarios, each showing different aspects of the tool. In the first scenario (Section 7.1.1), we use the Google News service to show data import, data filtering and multiple visualisations of Web service data. In the second scenario (Section 7.1.2), we present a simple mashup application using the Craigslist and Driving Guide services. In the third scenario (Section 7.1.3), we present a case study within a financial analysis application domain. We show how to reuse an already-built mashup application in this scenario. Finally, we show a composition in MashSheet using a bioinformatics scenario in Section 7.1.4.
7.1.1 News Feed Reader Scenario

With an ever increasing number of news data and information on the Internet, it is impossible for people to read everything that might be relevant. Therefore, it would be useful to have an application that assists readers in dealing with this problem possibly by filtering a number of news articles that they should read, and by visualising a list of news according to the readers’ preferences. In this section, we assess the capability of MashSheet in importing, filtering and visualising data from Web services.

**Application scenario:** Let us consider a scenario when Tom wants to pull news items relating to a keyword from a news provider and put into his spreadsheet. He uses a News service and gets items in different topics: politics, technology, economics, music, and sport. He, then, filters the news items using a keyword according to his need. Finally, he visualises the news in his favourite presentation format.

**Setup:** In this application, we use Google news Web service. Google news Web service is a computer-generated news aggregator that aggregates headlines from news sources. The Web service is in RSS format.

**Implementation:** The application can be implemented by using MashSheet as follows:

- Tom creates a component for Google news service to be used in MashSheet:

```plaintext
create('google';
'http://news.google.com/news?pz=1&cf=all &ned=au&hl=en&output=rss')
```

- A keyword is entered in cell B1 and Tom binds the google component into cell A2:

```plaintext
(B1,Michael Jackson)
(A2,bind(google))
```
• He invokes the `Read()` operation provided by the `google` component in cell B2. The result of invocation in cell B2 will be displayed in the component explorer (i.e., in the tree view) of the MashSheet tool.

\[(B2,\text{invoke(A2.Read())})\]

• All news items in cell B2 are filtered by the keyword in cell B1 by using the following formula in cell A4:

\[(A4,\text{filter(<<B2>>://rss/channel/item/title[fn:contains(title,'MichaelJackson')])})\]

• Tom visualises the results in cell A4 by entering the following formula in cell B4:

\[(B4,\text{grid(<<A4>>;column;A5)})\]

• Any change of value in cell B1 will trigger the recalculation and the worksheet will be updated.

Tom can also customise the visualisation of the news feed as follows:

• He only wants to view the title and link of the news items horizontally. He enters the following formulas in cells B4, B5:

\[(B4,\text{grid(<<A4>>;row;A7://rss/channel/item/title/text())})\]
\[(B5,\text{grid(<<A4>>;row;B7://rss/channel/item/link/text())})\]

The fourth parameter in above formulas is a XPath expression allowing Tom to customise his view.

• He can also view the feed vertically by enter the following formulas in cells A13, B13:

\[(A13,\text{grid(<<A4>>;column;B14://rss/channel/item/title/text())})\]
\[(B13,\text{grid(<<A4>>;column;B15://rss/channel/item/link/text())})\]

Figure 7.1 shows a screen shot of this implementation in MashSheet.
7.1. MashSheet Applications

Figure 7.1: Implementation of the News Feed Reader Application in MashSheet.
7.1. MashSheet Applications

**Discussion:** This scenario illustrates a basic application of a mashup tool being served as a feed reader with filtering capability. There is no mashup in the scenario in that there is no mixing of different data sources, however, the scenario shows some basic functions of a mashup tool: data importation, data manipulation (e.g., filter) and data visualisation.

StrikeIron cannot implement this scenario since it only supports Web services using SOAP interfaces.

AMICO:CALC can implement this scenario however users may experience a difficulty when creating Google news module and defining variables. AMICO requires users to put extra effort in configuring its java library for service adapters. Moreover, users have to manually define the layout of variables in the grid by using a combination of `Amico_Read` and `Amico_Write` operators.

In MashSheet, users can easily bind a Web service into a cell and invoke its methods without any configuration step. MashSheet’s users can utilise XPath to customise the grid visualisation and can also view data as a tree view or as HTML page in the component explorer. The main difference in data manipulation between MashSheet and AMICO is that MashSheet data operators run on XML data (e.g., Web service output) while AMICO:CALC uses spreadsheet’s functions to manipulate with simple data type (e.g., text, number).

Mashroom and SpreadMash facilitate users to import and visualise data feed in spreadsheet interface. While SpreadMash provides a set of widgets for users to import and visualise data, Mashroom relies on nested relational model for defining data visualisation in a nested table only.

Yahoo Pipes allows the creation of a data feed which consists of one or more data sources. It also has a set of operators, each of which performs data manipulation tasks (e.g., sort and filter). It perfectly serves this scenario with a user-friendly Web based interface. But the only form of data sources supported in Yahoo Pipes is RSS/Atom.

Damia, Marmite and Intel MashMaker also can facilitate users to implement this scenario with a few steps.
In summary, in this scenario, MashSheet is capable of implementing functions that are commonly seen in other mashup tools (e.g., import). In addition, MashSheet also provides flexible data manipulation and visualisation by incorporating XPath scripts into the operator syntax.

7.1.2 Housing Direction Scenario

As reported by ProgrammableWeb.com, 21% of mashup applications listed in this Web site involve map\(^1\), featuring mapping mashup as one of the most popular types of mashup in this Web site. By using APIs (e.g., Google Maps), mapping mashup provides useful applications such as Housing Map \(^2\), United States Postal Service tracking\(^2\), BBC News Map\(^3\), or Library Location \(^4\).

Inspired by the Housing Map application, in this section, let us consider a scenario where users want to combine property listings from Craigslist with driving direction information and map from Google to assist themselves in house searching.

**Application scenario:** Tom is moving to San Francisco for his new job and is in search of an apartment for rent. On the Internet, he uses Craigslist listing service to get properties’ advertisements. His office address is “750 Kearny Street, San Francisco, CA” and he wants to view driving direction from his work place to each property’s address in the list. Google map and direction services may help him in these functions.

**Setup:** In this application, we use two Web services: “Craigslist” \(^{12}\) and “Driving Guide” \(^{16}\).

- Craigslist is an online classified advertisements Web service. It provides sections devoted to housing, job, sale advertisements. We are interested in “San Francisco for rent” service - a RSS data feed for listing rental property in San Francisco.

\(^1\)The figures are as of November 2010 from \texttt{http://www.programmableweb.com/mashups}
\(^2\)\texttt{http://usps.ems-tracking.net/}
\(^3\)\texttt{http://dev.benedicteoneill.com/bbc/}
“Driving Guide” provides a SOAP-based Web service to find driving direction from one address to another address.

**Implementation:** The application is implemented by using MashSheet as follows:

- **Tom creates components for services used in MashSheet:**
  ```plaintext
create('craigslist';'http://sfbay.craigslist.org/apa/index.rss')
create('driving';'http://www.ecubicle.net/driving.asmx?WSDL')
```

- **His office address is entered in cell B1. He, then, binds two components into cells A2, B2:**
  ```plaintext
  (B1,750 Kearny Street San Francisco CA)
  (A2,bind(craigslist))
  (B2,bind(driving))
  ```

- **He invokes the Read() operation provided by the craigslist component in cell A4. The result of invocation in cell A4 will be displayed in the component explorer (i.e., in the tree view) of the MashSheet tool.**
  ```plaintext
  (A4,invoke(A2.Read()))
  ```

- **Tom visualises the result in cell A4 by entering the following formulas in cells B4, B5:**
  ```plaintext
  (B4,grid(< A4 >;row;A7; // * [name()] ='item'/title/text()))
  (B5,grid(< A4 >;row;B7; // * [name()] ='item'/link/text()))
  ```

- **To get the directions from each property’s address to Tom’s office, he enters the following formulas in cells C7 – C16:**
  ```plaintext
  (C7,invoke(B2.GetDirections(< A7 >,< B1 >,km,true)))
  (C8,invoke(B2.GetDirections(< A8 >,< B1 >,km,true)))
  (C9,invoke(B2.GetDirections(< A9 >,< B1 >,km,true)))
  (C10,invoke(B2.GetDirections(< A10 >,< B1 >,km,true)))
  ..
  (C16,invoke(B2.GetDirections(< A16 >,< B1 >,km,true)))
  ```
• The direction in cell C7 can be visualised by using formula in cell D7 as follows:

$$(D7, \text{grid}(<<C7>>; \text{row}; E7; //\text{drivingdirection/route/text}())$$

• Tom views the first property in the list in the Google map by enter the following formula in cell D4. The map is shown in component explorer of the MashSheet tool:

$$(D4, \text{map}(<<C7>>)$$

Figure 7.2 shows a screen shot of this implementation in MashSheet.

**Discussion:** This scenario shows a very common mashup application scenario where a Map service is mixed with another type of service(s).

Among the SB mashup tools, only MashSheet can implement this scenario. StrikeIron cannot implement the scenario due to the wrapper issue (i.e., it can only deals with SOAP-based services). AMICO:CALC does not support mashups at presentation level, which means a map cannot be rendered in Calc spreadsheet. Data visualisation in SpreadMash and Mashroom is restricted to grid only, so users cannot render a map in their application.

Yahoo Pipes focuses on merging feeds or enhancing existing feeds by transforming them by using data operators. However, this particular scenario cannot be implemented using Yahoo Pipes since it does not support SOAP-based Web services. Damia goes beyond Yahoo Pipes by providing a more generic data model that can accept a larger set of data sources (e.g., Notes, Excel, XML and SOAP-based Web services provided by StrikeIron marketplace). Damia can implement this scenario, however, both MashSheet and Damia have a limitation on data extraction and cleaning. They assume that the Craigslist service always returns addresses of properties in the “title” section. If the addresses are not in the “title” or the addresses are vague, then the driving direction service neither find the direction nor display the property on the map.

Unlike MashSheet and Damia, Marmite and Intel Mash Maker provide mechanism to easily extract content from one or more Web pages. The addresses of properties are achieved through machine learning techniques or programming by
7.1. MashSheet Applications

Figure 7.2: Implementation of the Housing Direction Application in MashSheet
demonstration techniques. This relieves the users’ pains in manually mapping data from the output of one service to the output of another service.

In summary, in this common scenario, MashSheet is the only tool among SB tools can implement the application. However, it still has limitations and does not serve to implement the scenario as good as Marmite or Mash Maker do, due to some manual work involved in data extraction.

7.1.3 Financial Market Analysis Scenario

Financial market research and analysis is an information intensive application domain that requires researchers to filter through enormous datasets to find the information required.

As another type of application domain we study, we chose an event study scenario in financial market data. Event study, a statistical method in financial market research, assesses the impact of an event on the value of a firm [116]. In this scenario, we will also show how our reuse concept can be applied.

Our event study scenario measures the degree to which a company’s value reacts to news. According to Binder [91], the standard method for event study is as follows:

- Collect multiple occurrences of a specific type of events for firms that fit a certain criteria,
- Find stock price changes for those firms in periods around the event date as well as changes in a market-wide index,
- Look for abnormal returns compared with usual returns, or returns adjusted for market rate of return, and
- Run further regressions for explain the abnormal returns.

Application scenario: We identify a scenario in the domain to find the correlation between abnormal stock prices and news data relating to the stocks\textsuperscript{4}. In

\textsuperscript{4}This scenario is inspired by [216]
particular, we are looking for significant price movements of stocks in major pharmaceutical companies traded on the New York Stock Exchange (NYSE). We will then find whether there is any correlation between the price movement and relevant information available from the news.

Figure 7.3 illustrates the scenario using BPMN\textsuperscript{5}. There are two mashup applications that serve this scenario\textsuperscript{6}:

- The first mashup application, named “Abnormal Detection”, imports stock price data of two companies (i.e., Pfizer and S&P 500), then applies, to each data set, a time-series function to normalise the data, then merges the data sets. After that an analysis service called ‘Abnormal Returns’ is invoked to detect any abnormal pricing behavior in the dataset. The result is visualised with a visualisation service.

- We reuse the code in the first mashup application to be used in a new mashup application, called “Correlation”, which runs News import service to find the correlation between abnormal stock pricing and news (e.g., any impact of news topics on the stock prices). The scope of the reuse is marked by a box (dotted lines) in Figure 7.3.

\textsuperscript{5}Business Process Modeling Notation, \url{www.bpmn.org}  
\textsuperscript{6}PFE.N and .SPXT are Reuters Instrument Code for Pfizer and S&P 500 stocks on the NYSE, respectively.
Setup: SIRCA\textsuperscript{7} and ADAGE project\textsuperscript{8} provide Web services that are designed to manipulate stock market data and news data. In the implementation of the scenario, we utilised the following Web services:

- **TRTH Import Service:** \textsuperscript{[70]} This Web service imports events from the Thomson Reuters Tick History (TRTH) System \textsuperscript{[195]} into the shared event repository. These events can cover company stocks from every financial exchange worldwide, currency exchange rates, indexes and interest rates. A selection of events can be imported using criteria such as period and exchange.

- **News Import Service:** \textsuperscript{[47]} This Web service imports news articles and reports from a news archive provider such as Thomson Reuters. Imported data are stored as News events.

- **Time Series Building Service:** \textsuperscript{[69]} This Web service processes events to produce time series data sampled at equal time intervals. This service is highly customizable as it allows the sampling period to be modified and a number of measures (such as the return, spread and vwap) to be included in the time series.

- **Abnormal Returns Service:** \textsuperscript{[1]} This Web service analyses trading price time series of a stock by using market level data (like index price and interest rate) to compute abnormal returns and therefore detect unusual price movements.

- **Visualisation Service:** \textsuperscript{[74]} This Web service allows a time series data set to be visualised as a chart.

- **News Service:** \textsuperscript{[48]} This Web service allows users to get news articles and reports from a news archive provider such as Thomson Reuters in a given time frame.

Implementation:

- Firstly, we need to implement the first mashup named “Abnormal Return”:

\textsuperscript{7}http://www.sirca.org.au/
\textsuperscript{8}Ad-hoc DAta Grids Environments, available at http://cgi.cse.unsw.edu.au/~soc/adage/
Components for import, time-series and ‘Abnormal return’ Web services are created by using the following formulas:

\[
\text{create('import';http://soc-server2.cse.unsw.edu.au:14080/axis2/services/TRTHimportService?wsdl)};
\]

\[
\text{create('timeseries';http://soc-server2.cse.unsw.edu.au:14080/axis2/services/TimeseriesService?wsdl)};
\]

\[
\text{create('abnormalreturn';http://soc-server2.cse.unsw.edu.au:14080/axis2/services/AbnormalreturnService?wsdl)};
\]

We, then, bind the import, timeseries, and abnormalreturn components to cells: A1, A9, and C1, respectively:

\[
(A1, \text{bind(import)})
\]

\[
(A9, \text{bind(timeseries)})
\]

\[
(C1, \text{bind(abnormalreturn)})
\]

To make the mashup application descriptive, we enter labels for input parameters in cells A2–A5, and A10–A12:

\[
(A2, \text{sec(String)})
\]

\[
(A3, \text{startDate(DateTime)})
\]

\[
(A4, \text{endDate(DateTime)})
\]

\[
(A5, \text{dataSourceURL(String)})
\]

\[
(A10, \text{eventId(String)})
\]

\[
(A11, \text{TimeIntervalDuration(String)})
\]

\[
(A12, \text{TimeIntervalUnit(String)})
\]

The input values are entered in B2–B6, and B12–B13:

\[
(B2, \text{PFE.N})
\]

\[
(B3, 01-Sep-2001)
\]

\[
(B4, 01-Oct-2001)
\]

\[
(B5, http://mashsheet.cse.unsw.edu.au:18080/FinDataSimple.csv)
\]

\[
(B11, 6)
\]

\[
(B12, \text{hour})
\]

The operations provided by Web services are invoked as follows:
7.1. MashSheet Applications

\[(A6, \text{invoke}(A1.\text{importMarketData}(<B2>, <B3>, <B4>, <B5>)))\]
\[(A7, \text{extract}(<A7>; /result/text()))\]
\[(A14, \text{invoke}(A9.\text{buildTimeSeries}(<A7>, <B11>, <B12>)))\]
\[(A15, \text{extract}(<A14>; /result/text()))\]
\[(A16, \text{download}(<A15>))\]
\[(C2, \text{invoke}(C1.\text{abnormalReturn}(<A16>)))\]

- Secondly, we create a template for the already-built “Abnormal Return” application.

\[
\text{publish(name: AbnormalReturn; creator: dat; user: helen;}
\text{input: B2 – B6; input: B12 – B13; output: C2; top – left: A1;}
\text{bottom – right: B13; tag: Finance)}
\]

- Thirdly, we reuse the template in the second mashup named “Correlation”

  - We create components for the ‘News’ and ‘Visualisation’ services in the second mashup application:

    \[
    \text{create(news:}\ http://soc-server2.cse.unsw.edu.au:9000/
    \text{AdageNewsWebServices?wsdl)}
    \]

    \[
    \text{create(visualisation:}\ http://soc-server2.cse.unsw.edu.au:14080/
    \text{axis2/services/VisualizationService?wsdl )}
    \]

  - The components are bound to cells A1 and B1:

    \[(A1, \text{bind(news)})\]

    \[(B1, \text{bind(visualisation)})\]

  - Input parameters are entered in cells A2–A8:

    \[(A2, \text{PFE.N})\]

    \[(A3, \text{SPXT})\]

    \[(A4, 01-Sep-2001)\]

    \[(A5, 01-Oct-2001)\]

    \[(A6, \text{http://mashsheet.cse.unsw.edu.au:18080/FinDataSimple.csv})\]
7.1. MashSheet Applications

We, then, load the AbnormalReturn template into the MashSheet tool:

\[
\text{load(template: AbnormalReturn)}
\]

- We execute the AbnormalReturn service in cell B3 for the PFE.N security code:

\[
(B3, \text{template}(\text{AbnormalReturn}(<A2>;<A4>;<A5>;<A6>;<A7>;<A8>))) )
\]

- We execute the AbnormalReturn service one more time in cell B4 for the .SPXT security code:

\[
(B4, \text{template}(\text{AbnormalReturn}(<A3>;<A4>;<A5>;<A6>;<A7>;<A8>))))
\]

- We also invoke the news component using the results from the output of the AbnormalReturn application:

\[
(B5, A1.\text{getNews}(<B2>))
\]

- Finally, user visualises the result:

\[
(B6, B1.\text{visualize}(<B3>))
\]

Figure 7.4 and 7.5 show screen shots of creating the AbnormalReturn template and reusing the template in the Correlation application.

Discussion: In this scenario, we realised an application in the financial analysis domain by using MashSheet. We also showed how to reuse already-built mashup applications in MashSheet. To our best knowledge, MashSheet is the first SB mashup tool that enables the reuse of already-built mashup applications. Currently, state-of-the-art spreadsheet mashup tools (e.g., StrikeIron, AMICO:CALC) cannot implement this scenario.

7.1.4 Evolutionary Relationships of Organisms

A scientific workflow is a chain of scientific tasks expressed in terms of tasks and their dependencies. Typically, it is used as data-driven application for comparing
We create a template from existing AbnormalReturn application. The required parameters are entered in the MashSheet menu.

Figure 7.4: Implementation of the Financial Market Analysis Application in MashSheet: Create Template
We use the template (i.e., execute the application) in cells B3 and B4.

Figure 7.5: Implementation of the Financial Market Analysis Application in MashSheet: Use Template.
observed and predicted data. It contains a wide range of components for querying databases, data transformation, data mining, and execution of codes [153]. Scientific workflow tools are used to create, edit, publish and execute workflows that analyse information about scientific problems, experimental results and problem solutions.

In this application, we investigate the possibility of using MashSheet as a scientific workflow tool, specifically in the bioinformatics application domain.

**Application scenario:** We consider a workflow\(^9\) where biologists want to find the evolutionary relationships between the different organisms of the interested genome sequence. They use the BLAST sequence alignment tool to find the gene in different organisms that is similar to the interested gene. Then they use the ClustalW tool to find the evolutionary relationships between genes. In this scenario, the workflow contains data types mismatch between the two Web services (i.e., the output of BLAST are protein identification (ID) and protein name while the input of ClustalW is protein sequence. The process of converting protein ID to protein sequence can be done by expert knowledge and a “protein data bank” Web service. Figure 7.6 illustrates this workflow.

![Figure 7.6: Bioinformatic Application Scenario](image)

\(^9\)The scenario is inspired by [100]
Setup: In this application, we use several bioinformatics Web services. DNA Database Bank of Japan\(^{10}\) has published a number of biological tools as Web services like Blast, ClustalW, DDBJ, Fasta, and others. We utilise the Web services provided by DDBJ for our application:

- **BLAST service at DDBJ** [5]. BLAST algorithm [4] allows biologists to compare DNA and protein sequences with an existing body of knowledge to find similar sequences in other organisms. Because of its usefulness, most genomics sequence data sources provide a BLAST interface to allow scientists to easily identify species or homologous species of an input sequence. The WSDL of DDBJ BLAST Web service is available at [5]. It provides eight Web operations, however, we are only interested in the `simpleSearch` operation.

- **ClustalW service at DDBJ** [9]. ClustalW is a general purpose global multiple sequence alignment program for DNA or proteins. It produces biologically meaningful multiple sequence alignments of divergent sequences. It calculates the best match for the selected sequences, and lines them up so that the identities, similarities and differences can be seen. The WSDL of DDBJ ClustalW Web service is available at [9]. It provides four Web operations, however, we are only interested in the `analyzeSimple` operation.

- **RCSB PDB Web service at PDB** [55] provides Web operations for accessing protein database. The WSDL of RCSB PDB Web service is available at [56].

Implementation: The application can be implemented using MashSheet as follows:

- User creates components for the Web services to be used in MashSheet:

  ```
  create('blast'; 'http://xml.nig.ac.jp/wsdl/Blast.wsdl')
  create('clustalw'; 'http://xml.nig.ac.jp/wsdl/ClustalW.wsdl')
  create('pdb'; 'http://www.rcsb.org/pdb/services/pdbws?wsdl')
  ```

- Input parameters (i.e., protein sequence, BLAST program name, database name) are entered in cells B1, B2, B3:

\(^{10}\)DDBJ
7.1. MashSheet Applications

(B1,'inputsequence')
(B2,blastp)
(B3, PDB)

- blast, clustalw, and pdb components are bound to cell A5, B5, and C5, respectively:

  (A5, bind(blast))
  (B5, bind(clustalw))
  (C5, bind(pdb))

- User invokes the searchSimple operation provided by the blast component in cell A6 with input parameters reference to cells B1, B2, B3:

  (A6, invoke(A5.searchSimple(<<B1>>, <<B2>>, <<B3>>)))

- User visualises the result in cell A6 by using the grid operator in cell A7:

  (A7, grid(<<A6>>; column; B8; //xml/proteinid))

- Since the output of blast’s searchSimple cannot be fed directly to the clustalw’s analyzeSimple operation, the user needs to select appropriate output from A6 to be used in the analyzeSimple operation. Since the user only invoke the pdb’s fastaQuery operation when the input parameter is “Chain A” proteins, he enters the following extended choice operators in cells C8–C9:

  (C8, iff(FIND('A', B8)='#VALUE!' then invoke(C5.fastaQuery(<<B8>>))))
  (C9, iff(FIND('A', B9)='#VALUE!' then invoke(C5.fastaQuery(<<B9>>))))

- User visualises the data in cells C8 – C9 by using grid operators:

  (E8, grid(<<C8>>; column; D8))
  (E9, grid(<<C9>>; column; D9))

- The data in cells C8 – C9 are merged in cell D10:

  (D10, merge(<<C8>>; <<C9>>))

- The analyzeSimple operation provided by clustalw is invoked in cell A11.
(A11.invoke(B5.analyzeSimple(<< D10 >>)))

- To prepare for visualisation, the user uses the following formula

(A12.filter(<< A11 >>; //xml/proteinid))

(A13.filter(<< A11 >>; //xml/letter))

- The result in cell A11 can be visualised by the chart operator in cell A14:

(A14.column(<< A11 >>; << A12 >>; << A13 >>))

Discussion: In this scenario, we use MashSheet to implement a scientific workflow in bioinformatics application domain. In comparison with scientific workflow tools (e.g., Taverna [169]), MashSheet exposes the following benefits and limitations:

Benefits:

- **Programming environment.** A scientific workflow tool should allow users to create, to execute, and to view the results of an executing workflow. Typically, when a scientist conducts data-driven experiments, he uses various analysis and querying tools in sequence, and in most cases, results (i.e. outputs) from the previous step of the experiment are fed into inputs of the next step. MashSheet, like other mashup tools, helps the scientist to semi-automate the process by offering an environment for composing services in “piping” fashion. Moreover, MashSheet uses spreadsheet interface - a productivity tool that most scientists are familiar with. The process of creating MashSheet application is as easy as writing formulas in spreadsheet.

- **Web service interfaces.** When scientific tools are exposed as Web services, the process of creating and executing them becomes fairly simple since services can communicate by producing and consuming compatible XML documents. In fact, many bioinformatics tools are available today in the format of Web services. MashSheet currently supports three service interfaces: RSS, REST and SOAP.

- **Incremental programming.** MashSheet allows users to bind Web services into cells, to obtain input/output parameters through XML-based definition of
7.1. MashSheet Applications

services, to execute and to view intermediate results in cells. Scientists are able to inspect and interpret results of the application (e.g., does this result make sense?), to examine execution traces and data dependencies (e.g., which results were “tainted” by this input dataset?), and to debug runs (e.g., why did this step fail?). They can refine the application design and re-run at any stage.

- **Data visualisation.** The philosophy of MashSheet visualisation operator is that data should be only visualised when needed. Intermediate data are only visualised when users want. This provides the MashSheet’s users flexibility in terms of visualising their application output.

- **Data flow and control flow.** It is clear that both control and data flow techniques are needed for scientific workflow languages. If we limit the language to one or the other, we will limit the usefulness of the tools. MashSheet users are facilitated by both data flow and control flow modeling capabilities.

**Limitations:**

- **Data transformation.** The implementation of scientific workflow in MashSheet requires some manual data transformation efforts to mediate between consecutive processing steps and between the data sources and data sinks. For example, in our scenario the BLAST Web service provides output in either HTML, text or XML format while the ClustalW Web service only accepts the input in FASTA format.

- **Large-scale of data.** Scientific workflows often contain many tasks, involve large data sets, and require intensive computation. However, MashSheet has limitations involve large data (e.g., limited number of rows and columns in a worksheet, and the length of text data in a cell).

- **Provenance.** Scientific workflow tools need to support not only data provenance but also workflow evolution provenance (i.e., a form of versioning for configured workflow). Currently, MashSheet does not support provenance.
7.2 Evaluation of the MashSheet Language

The workflow patterns model introduced in [199] is a reference framework for assessing workflow systems or languages in terms of their ability to model control flow structures. Using the workflow patterns, a system/language is evaluated by examining whether it provides direct support, partial support or no support for each pattern. Direct support means that the system provides a feature/construct corresponding to the description of the pattern. Partial support means that the system provides direct support but in limited situations (e.g., a pattern can be expressed via extended workarounds or programmatic extensions). No support means that the system does not provide direct support [199].

In this section, to evaluate the expressiveness of our mashup language, we assess how MashSheet fares in modeling the basic workflow patterns: sequence, parallel split, exclusive choice, simple merge, and synchronisation. More sophisticated workflow patterns (e.g., structural patterns, patterns involving multiple instances, routing patterns) can be used to evaluate MashSheet in the future work.

In the following, we will show whether it is possible to realise the patterns with the facilities offered by the MashSheet framework and by the benchmarking tools.

We choose AMICO:CALC and StrikeIron as benchmarking tools. First, they are available for download. There is no other other SP mashups tools we know of that are released publicly. Second, StrikeIron is a good indicator for what many other pipeline-based mashups can support. We consider AMICO:CALC as the best SP mashup tool so far in terms of supporting both pipeline-based and workflow-based models. So, without loss of generality, we believe we can compare our tool to the two.

7.2.1 Pattern 1: Sequence

General Description

“A sequence pattern describes the structure where a component service starts after the completion of another component service in the same process” [199].
7.2. Evaluation of the MashSheet Language

Implementation

**MashSheet.** This pattern is directly supported in the MashSheet framework. In fact, a sequence can be expressed in MashSheet either by spatial arrangement of cells’ formulas in a mashup application or by using cell referencing. Two formulas are sequentially evaluated if their cells are located in two adjacent cells. The evaluation order progresses from left to right and from top to bottom of the cell locations. Besides, two formulas are also considered sequentially evaluated if they have input-output dependency to each other.

**AMICO:CALC.** AMICO:CALC can model the sequence pattern by using a combination of *Amico_Read* and *Amico_Write* operators. Using *Amico_Read*, users can read value from variable (i.e., get the invocation result of a service). Using *Amico_Write*, users can assign value to variable (i.e., pass data as input parameter to a service). As shown in Figure 7.7, a sequence of service invocations is expressed. The output of the “spelling-suggestion” service is used as input for the “tts-input” service.

![Sequence Diagram](image)

*Figure 7.7: AMICO:CALC Approach to Express the Simple Sequence Pattern (This figure is derived from [166]*)

**StrikeIron.** The sequence pattern is not directly supported by StrikeIron in the sense that there is no tool’s feature or language construct provided by StrikeIron helps users to express the pattern. However, the input-output constrains defined among cells in spreadsheet can force the invocation of a service happens after another invocation. Therefore, the pattern is only partially supported by StrikeIron.
7.2. Evaluation of the MashSheet Language

7.2.2 Pattern 2: Parallel Split

General Description

“A parallel split describes the structure where a single thread splits into multiple threads which can be executed in parallel” [199].

Implementation

MashSheet. The parallel split pattern in MashSheet can be also expressed by using either spatial arrangement of cells’ formulas in a mashup application or by using cell references. Two formulas are evaluated in parallel if their cells are located in two non-adjacent cells without input-output data dependency. Besides, two formulas are evaluated in parallel if they have data dependency with the same data and have no data dependency to each other. Therefore, the parallel split pattern is directly supported in MashSheet.

AMICO:CALC. Figure 7.8 shows how the parallel split pattern is directly supported in AMICO:CALC by using a combination of Amico_Read and Amico_Write operators. In fact, three Amico_Write operators are considered as three outgoing split branches.

![Figure 7.8: AMICO:CALC Approach to Express the Parallel Split Pattern](This figure is derived from [166])

StrikeIron. StrikeIron directly supports the parallel split pattern. StrikeIron users are allowed to select the “update all” feature in the tool to concurrently re-
invoke all the Web service connections.

7.2.3 Pattern 3: Exclusive Choice

General Description

“An exclusive choice pattern describes the structure where, based on a decision or process control data, only one selected branch is activated and executed” [199].

Implementation

**MashSheet.**  MashSheet directly supports the exclusive choice pattern by using the `iff()` operator.

**AMICO:CALC.**  Figure 7.9 illustrates how to model the exclusive choice pattern in AMICO:CALC. The pattern is directly supported by using a combination of Amico_Read, Amico_Write and If operators.

![Figure 7.9: AMICO:CALC Approach to Express the Exclusive Choice Pattern](This figure is derived from [166])

**StrikeIron.**  StrikeIron has no direct support for the exclusive choice pattern. In fact, the built-in IF function in Microsoft Excel tool cannot represent the exclusive choice since it is not a process-oriented operator.
7.2.4 Pattern 4: Simple Merge

General Description

“A simple merge pattern describes the structure where more than one branches converge without synchronisation and only one of them has ever been executed” [189].

Implementation

MashSheet. This pattern is directly supported in MashSheet by using either input-output dependency between cells’ formulas and \texttt{sync()} operator.

AMICO:CALC. Figure 7.10 illustrates a simple merge scenario in AMICO:CALC. Three \texttt{Amico\_Read} operators are represented for three incoming branches for the merge.

![Figure 7.10: AMICO:CALC Approach to Express the Simple Merge Pattern](This figure is derived from [166])

StrikeIron. StrikeIron directly supports this pattern by using cell referencing mechanism.
7.2.5 Pattern 5: Synchronisation

General Description

“A synchronisation pattern describes the structure where multiple parallel branches converge into a single thread synchronised” \[199\].

Implementation

**MashSheet.** MashSheet directly supports this pattern by using the `sync()` operator.

**AMICO:CALC.** As pointed out in \[166\], the synchronisation pattern is partially supported in AMICO-CALC.

**StrikeIron.** StrikeIron provides no support for this pattern. In fact, users cannot model synchronisation pattern in StrikeIron.

7.2.6 Summary

Table 7.1 summarises how the basic workflow patterns are supported in MashSheet, AMICO:CALC and StrikeIron. We assign “+” as direct support of a pattern, “+/−” for partial support and “−” for no support.

<table>
<thead>
<tr>
<th>Workflow patterns</th>
<th>MashSheet</th>
<th>AMICO:CALC</th>
<th>StrikeIron</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>+</td>
<td>+</td>
<td>+/−</td>
</tr>
<tr>
<td>Parallel Split</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Exclusive Choice</td>
<td>+</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>Simple Merge</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Synchronisation</td>
<td>+</td>
<td>+/−</td>
<td>−</td>
</tr>
</tbody>
</table>

Table 7.1: Patterns Supported in some SB Mashup Tools

The table shows that by allowing basic control flow patterns to be presented in the mashup applications, the expressiveness of MashSheet framework is better than state-of-the-art SB mashup tools such as AMICO:CALC and StrikeIron.
7.3 User Study

In this section, we put forward a user study to evaluate the usability of the MashSheet framework. We carried out user experiments to collect practical feedback and to observe how MashSheet performs as a tool. In the study, we compare MashSheet with another SB mashup tool, AMICO:CALC. We consider AMICO:CALC as our benchmarking tool for this study because, according to our analysis in Chapter 3, it has similar goals to ours (i.e., supporting both implicit and explicit mashups). Also, it is publicly available and there is no special requirements for installing and executing AMICO:CALC.

Our claims for the contribution of the MashSheet framework are as follows:

- Overall, users take less time to build the same mashup applications in MashSheet compared to AMICO:CALC; and
- Users with varied programming experiences and technical knowledge can build MashSheet mashup applications and reuse existing applications.

7.3.1 Participants

We invited volunteers from different computer background to take part in this study. We chose our target group for invitation based on candidates’ work or field of study. The list of candidates is formed through the following three groups:

- **Group 1**: computer science researchers and research students. We invited members of Service Oriented Computing research group in the School of Computer Science and Engineering, the University of New South Wales;

- **Group 2**: computer science students. We invited students in COMP9322 - Service Oriented Architecture class in semester 2, 2011, the University of New South Wales; and

- **Group 3**: computer users, and finance and business students. We invited members of Securities Industry Research Centre of Asia-Pacific (SIRCA) and
postgraduate research students in the School of Business Law and Taxation, the Australian School of Business, the University of New South Wales

The participation was completely voluntary so there was no coercion or an inducement involved. However, to increase the chance of the candidates accepting the invitation, we were targeting the group of people who might find the research topic relevant or interesting to their work or study. For example, RESTful Web service composition and mashup are topics taught in COMP9322.

All of the participants had never used MashSheet and AMICO:CALC before and had varied background in using mashup tools and spreadsheets. They were required to report their skill levels of using spreadsheet tools, programming experience and knowledge in Web technologies. They were in one of the following three levels:

- **Novice**: participants know how to use at least one spreadsheet tool (e.g., Microsoft Excel or OpenOffice Calc) to analyse numerical data, have no programming experience, and possess a fair knowledge on Web technologies (e.g., XML and Web services),

- **Intermediate**: participants know how to use spreadsheets to do advanced features (e.g., formatting, sorting, charting, and importing XML), have some programming experiences, and possess a fair knowledge on Web technologies (e.g., XML, Web services),

- **Expert**: participants know complex Excel functions to manipulate and analyse data, know how to extend at least one spreadsheet tool using a programming language (e.g., VBA for Excel), and possess a good knowledge on Web technologies (e.g., XPath, SOAP, REST, and RSS)

Fifteen volunteers were recruited to take part in the study, comprising of five participants in Group 1, five participants in Group 2, and five participants in Group 3. Among them, eight participants reported themselves as expert, two as intermediate, and five novice-level users.
7.3.2 Tasks

There were three tasks in the user study. The participants were asked to implement three applications introduced in the Section 7.1 in this chapter (i.e., Section 7.1.1, Section 7.1.2 and Section 7.1.3) by using MashSheet and AMICO:CALC\textsuperscript{11}.

The tasks cover the basics of how to (i) invoke different Web services (RSS, SOAP, REST), (ii) mashup using cell references, (iii) manipulate the data generated by a Web service (e.g., filter), (iv) visualise data (e.g., grid, map) and (v) reuse existing mashup applications.

The first task involved building an application from a simple RSS data source. The second task required to build a mashup application combining data and functionalities from multiple Web services. The third task involved using the templates for reusing already-built applications.

At the beginning of the study, we handed out to the participants a short introduction of the MashSheet and AMICO:CALC tools, a summary of the operations’ syntaxes and a clear definition of the tasks need to be done. To ensure fairness in the evaluation, we allowed the participants to complete their tasks in a given computer. The computer is Dell XPS Laptop with Intel Core2Duo 2GHz, 3GB RAM, and contains Windows 7 Professional 64-bit SP1. At the end of the study, we asked the participants to complete a questionnaire form. A copy of this form is included in the Appendix B.

7.3.3 Procedure

The study process consisted of the following four main steps:

- **Preparing.** In this step, we contacted and invited the participants. A guideline was drafted to include system overview, goal of the study, scenarios used in the study, and syntaxes of the operations. Also, all of the software and tools required for the study were installed and configured.

\textsuperscript{11}Note that the third scenario cannot be implemented by using AMICO:CALC so that we only use MashSheet to implement this scenario.
7.3. User Study

- **Pre-investigation.** On the day of the study, we conducted a 30-minute tutorial session for the participants prior to the main study. In the tutorial, we handed out the materials for the study, introduced the goal and the tools overview, study scenarios and a demonstration on usages of MashSheet and AMICO:CALC. We gave time to the participants to understand the tasks correctly and to have any questions they had answered.

- **Main investigation.** In this step, the participants were asked to use MashSheet and AMICO:CALC to perform the tasks. In every stage of the tasks, the participants can get hints and advices from the study investigator, if needed. We recorded time spent by the participants to perform each task by each tool, which we summarised by mean and median later. If a participant did not complete a task in 10 minutes, the task was considered as incomplete and the time was counted as 10 minutes.

- **Post-investigation.** The participants were asked to complete a post-investigation questionnaire and invited to make additional comments at the end of the study, and through follow up emails.

### 7.3.4 Observations

Figure 7.11, Figure 7.12, Figure 7.13 and Figure 7.14 show the mean and median performances of the participants to complete the first and the second task by using MashSheet and AMICO:CALC, respectively.

Overall, the participants’ performances, measured in terms of “time-to-complete-each-task”, were better in MashSheet than in AMICO:CALC. In fact, in all operations required to complete the tasks, the time needed to run operations in AMICO:CALC was generally twice longer than the time needed in MashSheet.

Based on our observations, the main reasons why MashSheet outperforms AMICO:CALC can be explained as follows:

On average, the time spent to implement `create()` operation in AMICO:CALC was four times greater than in MashSheet. The reason for this difference is that the participants struggled with defining and configuring the modules and variables
7.3. User Study

Figure 7.11: Completion Time for the First Task (Mean)
The “grid() with filter” operation cannot be implemented in AMICO:CALC

Figure 7.12: Completion Time for the First Task (Median)
The “grid() with filter” operation cannot be implemented in AMICO:CALC
7.3. User Study

Figure 7.13: Completion Time for the Second Task (Mean)

The “grid() with filter” (shown as grid() in the figure) and map() operations cannot be implemented in AMICO:CALC.

Figure 7.14: Completion Time for the Second Task (Median)

The “grid() with filter” (shown as grid() in the figure) and map() operations cannot be implemented in AMICO:CALC.
needed for using Web services in AMICO:CALC. The main obstacle was that the middleware used for the configuration task is built outside of the spreadsheet tool, and the process of creating the modules and variables required the participants to have some basic knowledge on compiling and running java classes. In contrast to AMICO:CALC, in MashSheet, the participants only need to specify the names and URLs of Web services to create necessary components.

In AMICO:CALC, there is no feature or language construct that has the same goal as bind() operator. However, even when we sum up the time spent for create() and bind() in MashSheet, the result was still three times smaller than the amount of time the participants had to spend on defining and configuring variable in AMICO:CALC.

In MashSheet, the operation names of a Web service are clearly visible in the component explorer. The definitions (e.g., input and output) of a Web service’s operation can be dragged from the component explorer and dropped into the grid of cells by clicking onto the operation name. The participants, then, can easily invoke the operation by calling the invoke() operator. In AMICO:CALC, however, to specify a service invocation, the participants need to use a combination of Amico_Read and Amico_Write operators. The participants were confused on the semantics of the Amico_Read and Amico_Write operations. Consequently, the time spent for the invoke() operator in AMICO:CALC was longer.

In AMICO:CALC, when users define modules and variables to access a Web service, they need to flatten the output data from service invocation to a set of (variable, value) pairs. In addition, the output of a service invocation (i.e., value of variable) is not immediately visible within the spreadsheet. Users need to write a formula to extract the value in each variable to display in spreadsheet. In MashSheet, however, we extend the existing spreadsheet data model so that service output data can be accommodated in a cell. Hence, the participants can directly run data visualisation operators on the output data of a service invocation as it is. This significantly lowered the learning barrier and time to invoke visualisation operators. As a result, the time and effort needed to spend for the grid() operator in AMICO:CALC was 1.5 times longer than in MashSheet.
In both MashSheet, AMICO:CALC, the participants struggled with implementing the grid() operator with filtering capability. In MashSheet, the participants needed to use XPath expressions to filter the data, whereas in AMICO:CALC they needed to re-configure the module in order to filter data. No participant could complete the grid with filtering capability in AMICO:CALC. In MashSheet, only the expert users who were familiar with XPath syntax could complete the operation.

There is no visualisation operator supported like the map() operator in AMICO:CALC, so all the participants did not complete the task of visualising data on a map in AMICO:CALC. By contrast, the participants had no problem with running map() operator in MashSheet.

Figure 7.15 shows the average completion time of the participants to complete the third scenario by using MashSheet. Overall, the average completion time spent by participant for the create template operation is highest (i.e., 3 minutes) since users have to enter several parameters to the operator (e.g., name, creator, user, input cells, output cell, area, and tag). The time spent for finding and loading template is reasonably low (i.e., half a minute) because the syntaxes of the template find and load operators are simple. The average time spent for the execute operator is 1.5 minutes since users have to make some cell references in the formula. In terms of creating a template from already-build mashup applications, the participants did not find the needed difficult to grasp. However, since the participants had to provide quite a lot of data into a single formula to define a template, there were high chances
for errors (e.g., typo, incorrect cell referencing addresses). Most of the participants had at least one incorrect try when implementing this operation.

In terms of finding and loading template, they were successfully implemented because MashSheet provides a simple syntax for these operators. In terms of executing, the intermediate and expert-level participants had no problem since the metaphor of template is similar to functions in conventional programming languages. The novice-level participants could implement the operation, but were not sure if they did correctly or not.

Figure 7.16 and Figure 7.17 show the completion time (mean and median) of three participant groups (defined in Section 7.3.1) in implementing three tasks (defined in Section 7.1.1, Section sec:7.scenario2, and Section 7.1.3). In general, the completion time of participants in group 1 and group 2 are similar. We believe this is because the participants in these two groups have more advanced computer skills and background (computer science researchers and students) than users in group 3 (computer users and finance/business students).

7.3.5 Post-study Feedback

We asked the participants to complete a questionnaire form at the end of the study. The feedback from the participants was encouraging. Twelve people said that they
understood the design of the MashSheet framework after the user study, while three reported a little comprehension. There were thirteen people who completed the tasks by themselves, while two people needed help and hints throughout the tasks. All of the participants rated the suitability of using spreadsheets for mashups as good.

However, there were some criticisms of the MashSheet tool. The participants claimed that they committed some errors when entering mashup formulas and MashSheet did not help them to avoid errors by warnings. In addition, the syntaxes of MashSheet operators were not easy to remember and the participants had to refer to the syntax summary sheet frequently.

It is clear that the criticisms and weaknesses pointed out by the participants were not originated from the design and architecture of MashSheet, but mainly derived from the limited time available for implementing MashSheet to date. They can be addressed by implementing a syntax suggestion mechanism in the future release of the MashSheet tool.

We also collected the suggestions from the participants on the design of MashSheet. Some participants suggested an assistant feature in MashSheet to support users in writing mashup formulas. The expert-level participants suggested in-line text suggestion mechanism (i.e., when typing a formula, the system will try to predict the rest of the formula and suggest users with the correct syntax). Some participants
suggested using a wizard-like feature to write mashup formulas (i.e., similar to the Microsoft Excel’s formula wizard). In addition, some participants suggested that a cell’s colour should reflect its formula and value. For example, MashSheet should make the colour of a cell containing the `bind()` operator as grey, and `invoke()` as cyan and so on. It could help users quickly identify and locate a cell’s formula in a mashup application.

All suggestions from the participants are greatly appreciated and we will consider to include in the future implementation of the tool.

The user study in this dissertation is conducted on a small number of users (i.e., 15 participants) and there is no specific focus on the expertise level of each group. In the future, we will provide more thoughtful study (e.g., greater number of participants) and study the impact of MashSheet on each group of users (e.g., how it impacts on non-programmer group of users).

## 7.4 Chapter Summary

In this chapter, we put forward evaluations of the MashSheet framework.

We presented four different use cases in different application domains for applying MashSheet. We also presented an evaluation of MashSheet language to assess its expressiveness. The evaluation showed that by allowing the basic control flow patterns to be presented in the mashup applications, the expressiveness of MashSheet language is better than the benchmarking tools.

We presented a user study to evaluate the usability of our framework. Our user study indicated that, using MashSheet, users with all levels of computing experience implemented mashup scenarios in half the time compared to the benchmarking tool. The questionnaire also showed that users are comfortable and satisfied with the MashSheet framework as a mashup tool.
Chapter 8

Conclusions and Future Work

In this chapter, we summarise the contributions of this dissertation in Section 8.1. This is followed by research directions for future work in Section 8.2.

8.1 Concluding Remarks

Information integration, application integration and component-based software development have been among the most important research areas for decades. Recently, mashups have emerged and been considered as a key enabler in self-service data and application integration. While mashup researches have taken off and over thousands of mashups and APIs are currently widely available, the problem of adopting mashups in wider application domains and leveraging different platforms for mashup development have become a priority in both academia and the industry. Besides, spreadsheets are popular productivity tools and are useful in easy manipulation, analysis, visualisation and reporting of data. In [184], Scaffidi et al. estimate that there will be 55 million spreadsheet users in America alone in 2012, which provides a compelling argument for creating a mashup environment customised for spreadsheets.

In this dissertation, we have focused on the problem of providing a generic mashup framework using the spreadsheet paradigm. The first contribution of this dissertation is the identification of possible ways of implementing mashups in spreadsheet environment: implicit, event-based, using shared process data, and explicit.
Our study shows that the four approaches have different benefits and limitations in terms of expressiveness and usabilities.

Below, we summarise the most significant contributions of this dissertation:

1. **An analysis of the state-of-the-art SB mashup tools** (Chapter 3). We have characterised SB mashup tools and proposed a set of dimensions for analysing them from the view of data and application integration systems. We have proposed a general architecture of SB mashup tools including four major components. We have also conducted a quantitative analysis on state-of-the-art SB mashups using these dimensions. The analysis result could be useful for the researchers and developers in the area to understand the landscape of SB mashups, the strengths and weaknesses of the current approaches, and to develop ideas on creating more advanced SB mashup frameworks.

2. **The MashSheet framework for generic mashup application development and enactment** (Chapter 4). The main contribution of this dissertation is the MashSheet framework. MashSheet allows users to develop mashup applications using spreadsheet-like formulas. The key innovation of MashSheet is a collection of operators that supports coordinating Web services, manipulating and visualising data created by the services. MashSheet applications are incrementally built and data in each intermediary step is only visualised when needed. It makes the mashup application concise and easy to follow, as well as keep the computation logic separate from presentation.

   - **Generic data model.** We have proposed a generic data model that can accommodate a wide-range of data types. Using this data model, we introduced two components to represent a Web service and Web service output.

   - **Coordinating Web services.** We have proposed an extension to spreadsheet paradigm and formula language so that users can coordinate Web services in MashSheet applications. To coordinate Web services, MashSheet users can either use (i) spatial arrangement of the cell formulas as the evaluation order; (ii) cell referencing mechanism; or (iii) use process constructs.
8.1. Concluding Remarks

- Manipulating Web data. We have provided operators for manipulating Web data. The MashSheet data operators can perform operations either on the structure of data, on the data itself or transforming from complex data to simple type data. In addition, MashSheet data operators can manipulate the complex Web data as-is. The result of each data manipulation operation is held in cell for further processing by other data or visualisation operators.

- Visualisation of Web data. We have defined visualisation operators in MashSheet as components to present service output data using different visualisation types.

- MashSheet implementation in Microsoft Excel. The design of MashSheet is generic and can be implemented in different spreadsheet tools. We have discussed the possibility of implementing MashSheet in OpenOffice Calc, Gnome Gnumeric and implemented the prototype for MashSheet in the Microsoft Excel spreadsheet (Chapter 6). Using this prototype, we have applied to real-world scenarios and showed that the framework can adequately support mashup development (Chapter 7).

The evaluation on expressiveness of the MashSheet framework (Chapter 7) shows that the MashSheet framework is more expressive than state-of-the-art SB mashup tools such as AMICO:CALC and StrikeIron in terms of modeling basic workflow patterns in its applications. In addition, the case study (Chapter 7) also shows that MashSheet is generically applicable to different application scenarios.

3. Reusability of mashup applications (Chapter 5). We have investigated the problem of discovering and reusing already-built mashup applications in SB mashups. In order to enable users to reuse already-built applications, we have proposed a reference model facilitating the reuse of mashup applications (Chapter 4):

- We have conceptually defined a reusable application as a template. A MashSheet template can be used to define which parts of the already-built application are intended for reuse. Templates act as documents to
describe the functionality of applications without reference to particular instances. This means that templates are used to keep the code of applications to be used with different data sets. We have proposed an XML-based language for defining template (MTDL - MashSheet Template Definition Language).

- We have designed a repository that facilitates storage and manipulation of already-built applications. The meta data defining relationships between applications, users and requirements (e.g., input, output constraints) are managed within the repository. Users can access and manipulate templates by using operators in a low cost way.

We have conducted a user study of the MashSheet framework (Chapter 7). We have experienced that users took less time to build the same mashup applications in MashSheet compared to AMICO:CALC. Also, users with different level of programming experiences and technical knowledge could build MashSheet applications and reuse existing applications.

8.2 Future Directions

The work presented in this dissertation can be extended in many directions, some of which have been discussed in the earlier chapters, but we summarise in this section.

8.2.1 A Quantitative Analysis of SB Mashup Tools

We have introduced a qualitative analysis on the state-of-the-art SB mashups in this dissertation. We believe that end-users mashups in general and SB mashups in particular is an important area, which will attract a lot of attention from both academia and industrial community. However, the main limitation of the analysis is that we have conducted a qualitative instead of quantitative survey, so some of the observations could be inherently subjective. We plan to conduct a more rigorous survey with quantitative measures in the future study.
8.2.2 An Extension to Web Services Wrappers

For the sake of enabling the MashSheet framework to work with different types of Web services, we developed wrappers and adapters. However, as mentioned in Chapter 4, currently we only support achieving Web data from three sources: SOAP-based Web services, RSS data feeds and RESTful Web services. A possible research direction is to extend the number of wrappers and adapters to accommodate more services’ interfaces in the MashSheet framework. For example, multimedia applications, and legacy application (e.g., using low level TCP and UDP interfaces).

8.2.3 Software engineering aspects

As an application development tool, MashSheet needs to support users in various software engineering aspects, such as error handling, testing or debugging. We intend to provide these features to ensure the correctness of MashSheet applications developed by users.

8.2.4 Workflow patterns

We will investigate how MashSheet can support more workflow patterns [199]. For example, (i) Advanced branching and synchronization patterns (e.g., multi-choice, structured Synchronizing merge, multi-merge, structured discriminator), (ii) Multiple instance patterns (e.g., Multiple instances without synchronization, Static partial join for multiple instances, Dynamic partial join for multiple instances), (iii) State-based patterns (e.g., deferred choice, interleaved parallel routing, milestone), and (iv) Iteration patterns (e.g., arbitrary cycles, structured loop, recursion).

8.2.5 An Approach to Semi-automatic Mashups

Another possible direction for our future work is to provide a semi-automatic mashup framework. In order to provide such a framework, users’ requests and components need to be stated in a way that rich semantic information can be incorporated when components are queried. We try to embed intelligence into the manual process of
mashups: service selection, binding and composition through the smart use of context. We intend to construct a user model that reflects user preferences, such as usage characteristic, interest, expertise and community model that measure the interest of users on a mashup application or mashup component. Based on these two models, our framework incrementally guides users on the process of building mashup application. In addition, when a requirement changed (e.g., interest changed) the system will automatically suggest new mashup plan to the users. Our research prototype will have two basic components: a composer and an inference engine. The inference engine stores the information about known component, user model and community model in its knowledge base and has the capability to suggest composition plans. The composer has a spreadsheet grid interface that enable users to incrementally build their mashup applications. The users start the composition process by selecting preferred profile and one component service. A query is sent to the inference engine to get the suggestion on possible mashup plans. The composers then get the results and display the plans for user to select.
Bibliography


[64] SuperCalc spreadsheet.  

[65] Table Oriented Programming - A Practical, Intuitive, and Consistent Way to Organize and Process Data and Algorithm Collections.  
http://www.geocities.com/tablizer/top.htm  

http://www.javascript.com/  

[67] The Official YAML Web Site.  
http://www.yaml.org/  

http://www.w3.org/  

http://soc-server2.cse.unsw.edu.au:14080/axis2/services/TimeseriesService?wsdl  

[70] TRTH Import Service.  

[71] Uniform Resource Identifier.  

[72] Using DDE Functions with a DDE Client Application.  

[73] VisiCalc Information Web page.  
http://www.danbricklin.com/visicalc.htm  

[74] Visualisation Service.  
http://soc-server2.cse.unsw.edu.au:14080/axis2/services/VisualizationService?wsdl  


[102] David Churches, Gabor Gombas, Andrew Harrison, Jason Maassen, Craig Robinson, Matthew Shields, Ian Taylor, and Ian Wang. Programming Sci-
scientific and Distributed Workflow with Triana Services. *Concurrency and Com-


[127] Lushan Han, Tim Finin, Cynthia Parr, Joel Sachs, and Anupam Joshi. RDF123: From Spreadsheets to RDF. In Proceedings of the 7th International Semantic Web Conference (ISWC ’08), Karlsruhe, Germany, pages 451–466, 2008.


[154] Bertram Ludascher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific Workflow


[212] Andrew Witkowski, Srikanth Bellamkonda, Tolga Bozkaya, Aman Naimat, 
Lei Sheng, Sankar Subramanian, and Allison Waingold. Query by Excel. In 
Proceedings of the 31st International Conference on Very Large Data Bases 
(VLDB ’05), Trondheim, Norway, pages 1204–1215, 2005.

[213] Jeffrey Wong and Jason Hong. What Do We ”Mashup” When We Make 
Mashups? In Proceedings of the 4th International Workshop on End-user 

[214] Jeffrey Wong and Jason I. Hong. Making Mashups with Marmite: Towards 
End-User Programming for the Web. In Proceedings of ACM CHI 2007 Con- 
fERENCE on Human Factors in Computing Systems (CHI ’07), San Jose, CA, 

Challenges of Service Composition in E-Marketplaces. In Proceedings of the 
12th International Workshop on Research Issues in Data Engineering: Engi- 
neering E-Commerce/E-Business Systems (RIEDE’02), Washington, DC, USA, 

[216] Lawrence Yao and Fethi A. Rabhi. Modelling Exploratory Analysis Processes 
for eResearch. In Proceedings of the 21st Australasian Conference on Infor- 
mation Systems (ACIS ’10), Brisbane, Australia, pages 14–24, 2010.

[217] Louis E. Yelle. The Learning Curve: Historical Review and Comprehensive 


[220] Jin Yu, Boualem Benatallah, Fabio Casati, Florian Daniel, Maristella Mat- 
era, and Regis Saint-Paul. Mixup: A Development and Runtime Environment
Appendix A

MashSheet Template Definition Language (MTDL) Specification

• Version 1.0
• Date: 15 Jun, 2011
• Author:
  – Dat Dac Hoang, UNSW
  – Hye-young Paik, UNSW

A.1 Abstract.

The MashSheet Template Definition Language allows the definition of MashSheet template creating from a MashSheet mashup application. Its syntax is based on XML format. This document describes the syntax and semantics of MTDL elements.

A.2 Introduction.

MashSheet Template Definition Language, abbreviated as MTDL, defines a class of documents called MashSheet templates. It is built based on Extensible Markup

Language and has the main goal of describing elements contained in a template document, including name, location, user, cell, area, formula, value, tag.

MashSheet templates act as documents to describe the functionality of MashSheet applications without reference to particular instances. This means that templates are used to keep the “code” of applications to be used with different data sets.

Example. The following document exemplifies the definition of Currency-Conversion template using MTDL:

```xml
<mtdl xmlns="http://mashsheet.cse.unsw.edu.au:8080/MashSheet"
version="1.0">
<name>Currency-Conversion</name>
<location>http://../Library/Currency-Conversion.mtdl</location>
<User login="tim" role="creator" />
<User login="tom" role="user" />
<Cell id="B1" type="input" />
<Cell id="B2" type="input" />
<Cell id="B3" type="input" />
<Cell id="A5" type="output" />
<Area startRow="1" endRow="5" startCol="1" endCol="2" />
<Formula id="null" value="create(currency, http://www.xignite.com/xCurrencies.asmx?WSDL)" />
<Formula id="A4" value="bind(currency)" />
<Formula id="A5" value="invoke(currency.ConvertRealTimeValue(B1; B2; B3))" />
<Value id="A1" text="Input currency" />
<Value id="A2" text="Output currency" />
<Value id="A3" text="Amount" />
<Tag name="Financial" />
<Tag name="Exchange rate" />
</mtdl>
```

A.3 MTDL Document Structure.

A.3.1 The MTDL Root Element

`<mtdl>` is the root XML element in a MTDL document. It has the following attributes:

- namespace: `http://mashsheet.cse.unsw.edu.au:8080/MashSheet`
- version: the version of the MashSheet template descriptor

A `<mtdl>` element contains a list of elements: `<name>`, `<location>`, `<user>`, `<cell>`, `<area>`, `<formula>`, `<value>`, `<tag>`
A.3.2 MTDL Elements

**Name.** This element indicates the name of MashSheet template.

**Location.** This element specifies location of the MashSheet template.

**User.** This element specifies a list users who have permission to access the MashSheet template. It has the following attributes

- login: the login name of user who want to access the MashSheet template. This login name is provided through a registration process,
- role: this attribute specifies the role of login name for the template. Its value can be either “creator” or “user”

**Cell.** This element specifies parameters of the MashSheet template. It is a list of cells in the MashSheet application. It has the following attributes:

- id: this attribute is the coordinate of a cell in MashSheet application to be used as parameter of the template,
- type: this attribute is the type of parameter. Its value can either be “input” or “output”

**Area.** This element specifies reusable boundary of the MashSheet application. It is a rectangular area in a MashSheet application that we want to make a template for reuse. It has the following attributes:

- startRow: this attribute specifies start row of the reusable rectangular area of cells,
- endRow: this attribute specifies end row of the reusable rectangular area of cells,
- startCol: this attribute specifies start column of the reusable rectangular area of cells,

- endCol: this attribute specifies end column of the reusable rectangular area of cells

**Formula.** This element contains formula of a MashSheet cell in the application. It has the following attributes:

- id: This attribute is the coordinate of the cell,
- value: This attribute contains MashSheet formula that belongs to the cell

**Value.** This element contains value of a spreadsheet cell in the application. It has the following attributes:

- id: This attribute is the coordinate of the cell,
- type: This attribute contains spreadsheet value of the cell

**Tag.** This element contains the description of tag name used for a template. It has the following attribute:

- name: This attribute is user-defined tag name for the template.

### A.3.3 MTDL Schema Definition

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://mashsheet.cse.unsw.edu.au:8080/MashSheet"
    xmlns:cm="http://mashsheet.cse.unsw.edu.au:8080/MashSheet">

<x:element name="mtdl">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="name" type="xs:string" minOccurs="1" maxOccurs="1"/>
            <xs:element name="location" type="xs:string" minOccurs="1" maxOccurs="1"/>
            <xs:element name="users" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>
            <xs:element name="cell" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>
            <xs:element name="area" type="xs:string" minOccurs="1" maxOccurs="1"/>
            <xs:element name="formula" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="value" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element name="tag" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
</x:element>

<x:element name="user">
    <xs:complexType>
        <xs:attribute name="login" type="xs:string" use="required"/>
        <xs:attribute name="role" type="xs:string" use="required"/>
    </xs:complexType>
</x:element>
</xs:schema>
```

</xs:element>
<xs:element name="cell">
    <xs:complexType>
        <xs:attribute name="id" type="xs:string" use="required" />
        <xs:attribute name="type" type="xs:string" use="required" />
    </xs:complexType>
</xs:element>

<xs:element name="area">
    <xs:complexType>
        <xs:attribute name="startRow" type="xs:nonNegativeInteger" use="required" />
        <xs:attribute name="endRow" type="xs:nonNegativeInteger" use="required" />
        <xs:attribute name="startCol" type="xs:nonNegativeInteger" use="required" />
        <xs:attribute name="endCol" type="xs:nonNegativeInteger" use="required" />
    </xs:complexType>
</xs:element>

<xs:element name="formula">
    <xs:complexType>
        <xs:attribute name="id" type="xs:string" use="required" />
        <xs:attribute name="value" type="xs:string" use="required" />
    </xs:complexType>
</xs:element>

<xs:element name="value">
    <xs:complexType>
        <xs:attribute name="id" type="xs:string" use="required" />
        <xs:attribute name="text" type="xs:string" use="required" />
    </xs:complexType>
</xs:element>

<xs:element name="tag">
    <xs:attribute name="name" type="xs:string" use="required" />
</xs:element>
Appendix B

User Study Material

This appendix provides the questionnaire for MashSheet user study.

Survey No:
Participant name:
Date/ Time:

Questionnaire

Part 1: Background

Question 1.A. Do you know what MashSheet does? Yes/ No

Question 1.B. How do you rate the suitability of using spreadsheet for mashups?

- Not Suitable
- Neutral
- Suitable

Question 1.C. What is your skill level of using Microsoft Excel spreadsheet?

- I have no idea what Microsoft Excel is
• I know how to use Microsoft Excel to analyse numerical data and do calculations

• I know how to use Microsoft Excel to do advanced features (e.g., formatting, sorting, charting, importing XML)

• I know how to extend functionalities of Microsoft Excel using programming language (e.g., VBA)

Part 2: Implementation task 1

Question 2.A. Were you able to complete the task? Yes/No Please select the most accurate description of your experience with this task:

• I knew what to do with it right away

• After re-reading the material provided I understood what to do

• I completed the task myself but I am not sure if I did it correctly

• I needed help from others to complete the task

• Something was unclear about the task and I was unable to figure out what to do.

Please list the reasons that hamper you to finish the task (e.g., documentation, software, task):

Question 2.B. How long did it take you to finish the task?

Question 2.C. What were the system errors you experienced during the task?

Question 2.D. What features would you suggest to the developers of this tool?

Part 3: Implementation task 2
**Question 3.A.** Were you able to complete the task? **Yes/No** Please select the most accurate description of your experience with this task:

- I knew what to do with it right away
- After re-reading the material provided I understood what to do
- I completed the task myself but I am not sure if I did it correctly
- I needed help from others to complete the task
- Something was unclear about the task and I was unable to figure out what to do.

Please list the reasons that hamper you to finish the task (e.g., documentation, software, task):

**Question 3.B.** How long did it take you to finish the task?

**Question 3.C.** What were the system errors you experienced during the task?

**Question 3.D.** What features would you suggest to the developers of this tool?

**Part 4: Implementation task 3**

**Question 4.A.** Were you able to complete the task? **Yes/No** Please select the most accurate description of your experience with this task:

- I knew what to do with it right away
- After re-reading the material provided I understood what to do
- I completed the task myself but I am not sure if I did it correctly
- I needed help from others to complete the task
- Something was unclear about the task and I was unable to figure out what to do.
Please list the reasons that hamper you to finish the task (e.g., documentation, software, task):

**Question 4.B.** How long did it take you to finish the task?

**Question 4.C.** What were the system errors you experienced during the task?

**Question 4.D.** What features would you suggest to the developers of this tool?

**Part 5: Other**

**Question 5.A.** Do you have any other comments?