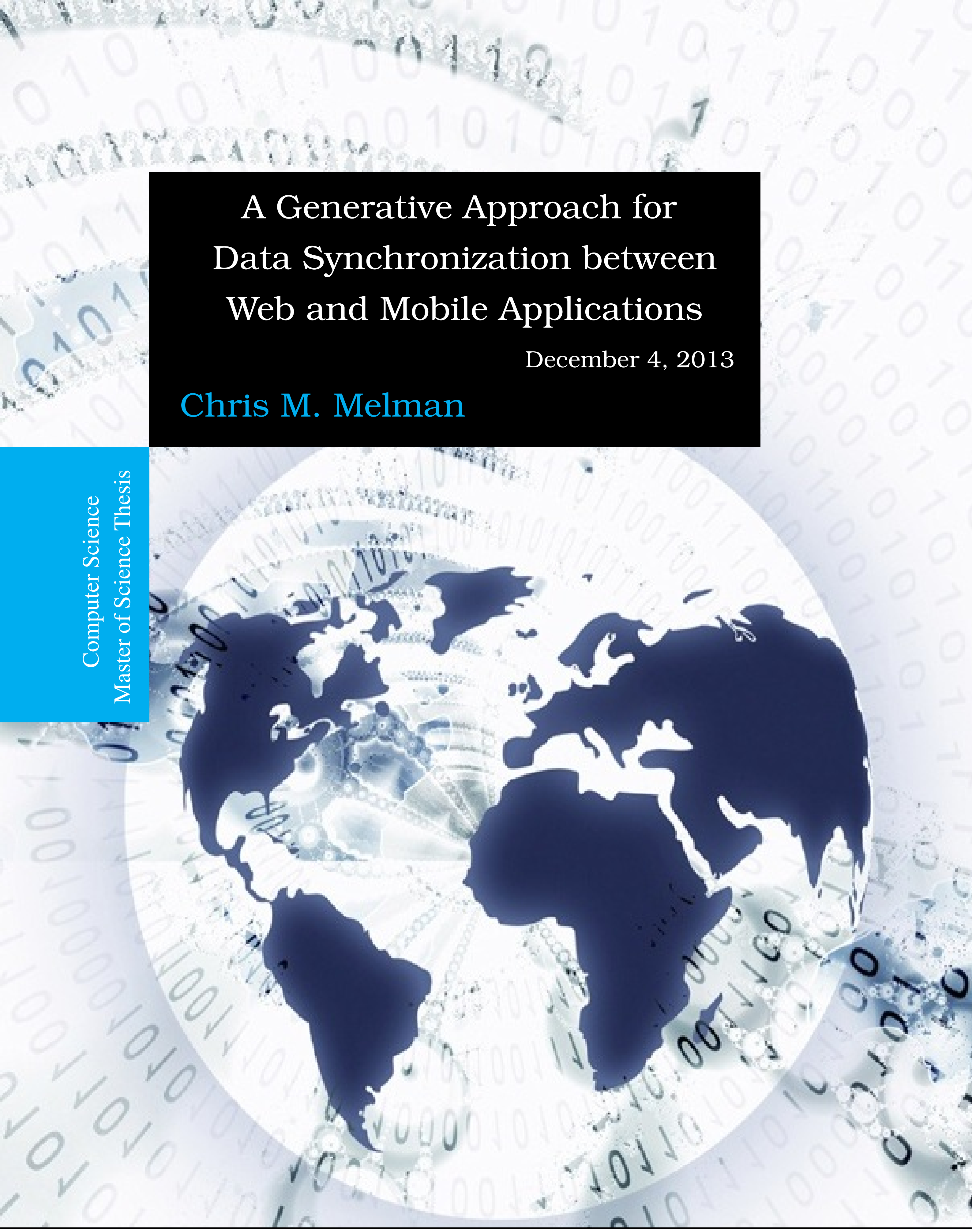
A Generative Approach for Data Synchronization between Web and Mobile Applications

December 4, 2013

Chris M. Melman

Computer Science
Master of Science Thesis



TU Delft Delft University of Technology

Software Engineering Research Group

A Generative Approach for Data Synchronization between Web and Mobile Applications

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Chris M. Melman

born in Haarlem, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Digits bursting into the world

A Generative Approach for Data Synchronization between Web and Mobile Applications

Author: Chris M. Melman

Student id: 1358359

Email: chrismelman@gmail.com

Abstract

Mobile developement is a relatively new and popular domain for applications. An increasing amount of web applications are releasing a mobile variant of their application, which requires sharing of data. Currently, the creation of a mobile version can be done in two ways: a specialized web UI for mobile devices, which lacks good abstractions for mobile hardware, or a manually implemented mobile application, which is more expensive and often causes a high amount of code duplication between web and mobile code.

This thesis presents a generative approach for data synchronization between web and mobile applications, which simplifies the creation of a standalone mobile application based on an existing web application. The generated framework is an incremental approach for synchronization of data in different representations. The framework uses object relations to determine selective data partitions to reduce the amount of data. Additional access control rules and validation expressions are used to enforce a secure and robust system.

Thesis Committee:

Chair:	Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor:	Dr. E. Visser, Faculty EEMCS, TU Delft
Daily supervisor:	Ir. D.M. Groenewegen , Faculty EEMCS, TU Delft
Committee Member:	Dr. G.H. Wachsmuth, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Bozzon, Faculty EEMCS, TU Delft

Preface

About this thesis This thesis consists of thee parts. The first part, Chapter 1 to Chapter 4, analyzes the three domains covering the solution space: Web, Mobile and Synchronization. The second part, Chapter 5, contains details on the motivating example YellowGrass Mobl. Which leads to the last part, Chapter 6 to Chapter 8, containing details on architecture, implementation and evaluation of the final solution of this thesis.

Acknowledgements I am grateful to Eelco Visser for his feedback, insightful conversations and the possibility to execute my own thesis proposal. I would like to thank Danny Groenewegen for his support on WebDSL , his task as daily supervisor, a sparring partner in discussions and delivering valuable feedback on my work. I would like to thank Zef Hemel for his support on Mobl. I am grateful to all the SLDE members for their support and good coffee conversations, but more importantly for providing a comfortable environment to produce my thesis. Last but not least, I am grateful to my family, friends and girl for support, motivation and affection during my studies.

Chris M. Melman
Delft, the Netherlands
December 4, 2013

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays it seems that smartphones and internet are products, which we cannot live without anymore. Both are relatively new technologies which have been adapted quickly into our society. This adaption seems to be triggering more and more applications to be available on those domains. This created a movement from developing conventional software to highly interactive web and mobile applications. However, due to the fast rise of both technologies there has been limited time for adaptation and creation of developer tools for the domain. Additionally, solutions for conservative software do not always apply in web and mobile software.

In our studies the first real contact with development for both domains was in the course Model Driven Software development. A part of the course was a project to create a social media application like facebook in a web and mobile variant. The development was done using Domain Specific Languages (DSLs) for the specific domain of web and mobile applications. Those languages: WebDSL and Mobl are developed by the Software Language Design and Engineering group (SLDE) using Spoofax.

The software for the lab was separated into three parts. Next to the web and mobile application an additional webservices layer is added as separate module. This layer is needed for communication between the web and mobile application because the mobile application operates on the same data. The data is stored on the server from the web application. After reviewing the code of the project, it showed a high amount of code and logic duplication that was added by the webservices. There was even functionality implemented in the webservices that was not available for the web application itself.

Not only in the course, but also in practice it is becoming normal that web application have a mobile variant. Those applications are mostly manual implementations of the web variant and require webservices to request data or send changes. While there are technologies that deliver abstractions for creating web applications and similarly, for creating mobile applications. However, the world lacks a solution that takes into account that both variants of an application are desired. This requires an infrastructure to share data.

A solution for sharing data is synchronization of data between devices. There has been a lot of research into the topic of synchronization which shows that the optimal solution is dependent on the context. We are interested in determining what emerges when more domains are inserted into the context. Including the question, can the difference between both domains be overcome within a synchronization framework?

This thesis uses a deeper analysis of the domains to deliver a better view on what are the restrictions are requirements for a synchronization framework within this context. A manual implementation of a synchronization framework and mobile application of YellowGrass is used as a motivating example. This delivers practical insight into the problem.

The goal of this thesis is to present a solution for the problem described in the previous paragraphs of missing technologies that overlap the gap of shared data between web and mobile application. The solution proposed in this thesis is a generative approach for a synchronization framework that is based on information from the existing code of the web application. This generative approach should consider the constraints and problems gathered from the analysis and infer abstractions from the motivating example. The synchronization framework should be simple and require only a small amount of interaction. An incremental solution is preferred to reduce the load in bandwidth and computation for the mobile devices. The webservices for synchronization imply additional security risks to the application and data, those risks should be considered in the design and implementation of the generated synchronization framework.

1.1 Research Questions

The research and engineering for this thesis is driven by the following research questions:

Research Question A *How do existing synchronizations solutions apply to the domain of web and mobile applications?*

The problem that this thesis tries to solve can be categorized into synchronization problems. As stated in the previous section, the problem and solution of synchronization is dependent on the context. So before answering the synchronization problem a deeper analyses is needed of the domains of web and mobile applications. With this analysis we can consider what possible solutions would fit and which additional constraints and requirements are included within the combination of both domains.

Research Question B *How can we automate the creation of incremental data synchronization between web and mobile applications?*

Secondly, how can we make a synchronization framework that works for a wide range of web and mobile applications. This question covers multiple aspects that are required from this approach. Firstly, how can we automate the creation so that it can be applied to various web applications. Secondly, what is needed for an incremental synchronization solution to reduce bandwidth and computation. Synchronization is often a process which needs to be a black box for the user which allows your data to be synchronized. This demands additional research into how it can be achieved to reduce the interaction. At last the synchronization has as side-effect that it delivers extra security concerns for a web application. How can we deal with the security problems without manipulation of the original application.

Research Question C *How can we optimize a synchronization algorithm to use a minimal amount of computation on mobile devices?*

The last part is to question if it is possible to optimize the solution for mobile devices. It is a known problem for the domain of mobile applications that mobile devices are restricted in factors such as computation, space and connectivity. Space and connectivity are not factors that can be compensated with other machines. However, computations can in some conditions be offloaded to other machines. This concludes the question stated above, how can we optimize the synchronization for mobile devices?

1.2 Outline

This thesis is organized as follows. Analysis of the web and mobile application domain and the possible impact on synchronization is described in Chapter 2. Chapter 3 gives insight in the synchronization problem and existing solutions. The information is combined to state the requirements for a solution(Chapter 4). The analysis is followed by Chapter 5 containing the explanation and evaluation of the motivating example. Chapter 6, Chapter 7 and Chapter 8 contain corresponding the architecture design, implementation details and evaluation of the final solution presented by this thesis. A summary of this thesis and answers to the research questions are given in Chapter 10. Finally, the future work is discussed in Chapter 11.

Chapter 2

Web & Mobile Applications

Web and mobile are a popular target platform for new applications. An application often has a version for both platforms, which requires data sharing or synchronization between devices. Synchronization solutions deviate from each other depending on their context. This fact calls for an inspection of the impact on synchronization by both domains.

This chapter explains the aspects of web (Section 2.1) and mobile (Section 2.2) applications and how they differ from each other. The second goal of this chapter is to give an introduction into the chosen target languages WebDSL and Mobl corresponding to their domains in Section 2.3. To conclude, Section 2.4 explains the influence of the domains from a synchronization viewpoint.

2.1 Web Applications

In the last decade there has been an increase in websites and web applications which also seem to be the trend for the near future [5]. Many of the desktop applications have moved or added a web version of their application. There are several reasons to prefer the web variant, but the most convenient is that it becomes easier to access the application and data for users anywhere.

This section starts with stating the important differences between conventional and web development (Section 2.1.1) and describes the important aspects of web applications (Section 2.1.2).

2.1.1 Conventional vs Web Engineering

The development of desktop applications and web applications deal with the same aspects. They each have other priorities and perception of those aspects. Mendes et al. state that the biggest differences between web and conventional software development are found in the following attributes: intrinsic characteristics, stakeholders and discipline variety [27].

Intrinsic characteristics of web applications

Web applications are remote datadriven applications that are available through the browser. Remote applications might not sound that more complicated, but having a remote access point directly raises problems: How to deal with, concurrency, load balancing, availability and other network characteristics.

One of the biggest concerns of web applications is security. Desktop applications store most of their information locally. In comparison, web applications have to store this on a remote server. The storage of data is already questionable for privacy reasons. However, it also increases the changes of getting sold, stolen or leaked without knowing.

Web applications have to combine more (smaller) technologies to build an application. Take a look at the Graphical User Interface (GUI), it uses at least three languages (HTML, CSS, JavaScript) to compose an interactive interface. Where conventional applications often use one language, in general the same language as the rest of the code, to compose an interface. The web browsers benefits from a universal display of the GUI, but comes at the price of the limitations in interaction possibilities of the web technologies: HTML, JavaScript and CSS.

Stakeholders

In conventional software development the stakeholders are predefined by the client and are accessible for questioning. Web development could also have predefined users if the software would be developed for internal usages. However, most of the web applications target are openly accessible. The open applications could define a target audiences. Nevertheless the applications are available for a wider group of people. Besides the unknown users, there is also the possibility that other software could interact with the application. The unknown and unreachable users make it harder to get a clear scope of functionality and requirements of a web application.

Multidisciplinary

In development of web software there are many aspects that have to be dealt with (Section 2.1.2). Conventional software implementation is often done by a development team of essentially general programmers. With web development it is not clear that a general purpose programmer can solve the issues of other areas. For example, GUI design or Network infrastructure are often done by specialists like graphic designers or hardware specialists. It is common to see a team of members with diverse specializations to create full web applications. When the application or the producing company becomes bigger, the gap of specialists diversity will reduce.

2.1.2 Web Application Aspects

The previous paragraph mentioned that web applications have to deal with multiple disciplines. Each of those aspects generally use different technologies and languages to implement correct behavior. The list below describes the most common aspects of web development and their possible influence on synchronization of data and therefore the solution of this thesis. The relations and the position of the aspects within a web application are displayed in Figure 2.1.

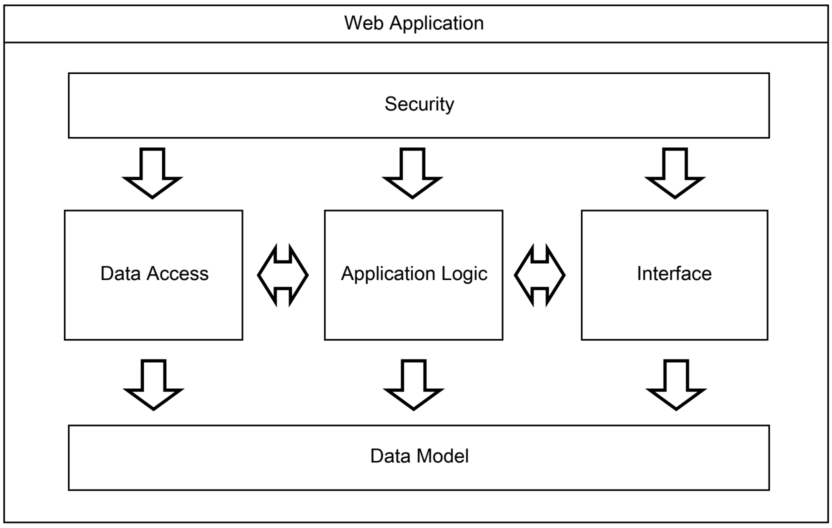


Figure 2.1: Decomposition overview of a Web application with its aspects

.

Data Model *Describes the relation between the available data, also describes the problem domain of the application.* Normally, programming languages allow to define a data model with relations directly in the code. The database structure also implies a data model for the stored data, which does not have to be the same as the one in the code of the application.

Data Access *Provides the possibility to manipulate stored data within the application code.* Programming languages support data modeling, yet data access is normally separated into libraries and are data storage dependent. Those libraries allow to access and modify stored data, through composition of queries.

Application Logic *Acts as glue between the data and the User Interface (UI) and describes the business logic of an application.* The application logic is an interesting and big part of the code base. However, it consists of normal computations just like most code in conventional software development.

Interface *Delivers a front end of the application for users to interact with the application.* Web applications often have two interfaces, one for people and another interface for external software. The goal of those interfaces are not similar, people prefer an attractive graphical interface. Where software does not need this sugar and only requires a thin layer in front of the application logic as interface.

Security *Protects the application from malicious users.* Security is one of the bigger concerns of web applications and most frameworks deliver a separate part to define security rules. Security is weaved through the other modules because it is required or requires information on different levels of the application.

2.2 Mobile Applications

The last couple of years the popularity of mobile devices has grown to a level that the majority of people in western countries own one or multiple mobile devices. Next to that there has been an enormous increase of quality and quantity on mobile internet, allowing people to access data from almost anywhere [4]. Combining those two factors caused higher usages of web applications on mobile machines. Most of the applications were not fully accessible because of the different interaction model of mobile devices. The different paradigm has triggered an increase in specialized applications for the mobile platform.

As continuation on the previous section about web applications, this section describes the differences between web and mobile applications in Section 2.2.1 and the hardware limitation of mobile devices in Section 2.2.2. The comparison of different platforms for creating applications can be found in Section 2.2.3.

2.2.1 Web vs Mobile Applications

The number of mobile applications created in the last couple of years has been higher than that of web applications. One of the reasons mobile applications are created is the fact that popular web applications need a mobile application to keep itself in the market. Development of a mobile application impacts the code base because for every functionality now it needs to be implemented in both the mobile and the normal application. The extra application leads to duplication of code and different implementations, what could even be more than one if there are different applications for different mobile operating systems.

Mobile and web applications have similar aspects to deal with, yet due to differences in the hardware and interaction model, they deviate in the how to deal with those aspects. To compare both platforms, this section evaluates the differences by reviewing the most common aspects that are described in Section 2.1.2.

Data Model
Mobile applications often do not have access to the same data as the actual application. Security and hardware limitations prefer that the data is shared on a need to know basis. Those restrictions on data have as consequence that mobile application use a simplified or ad hoc interpretation of the actual data as internal model. This simple model makes it easier to reason about data at the price of missing the bigger picture of the application domain.

Data Access Web applications normally persist their data to a database, but in general this is not the perspective of mobile applications. The mobile applications often require data from a remote servers which than could be manipulated locally. On application level it varies whether the data is persisted or just kept in memory. The same choice holds for changes, they could be stored locally to sync later or they could be only modified directly through server communication. The mobile devices have a limitation on bandwidth and space because of those restrictions a combination of both models is regularly used. Normally, this is the situation where the data is cached, although not fully persisted.

Application Logic
Mobile application have a limited set of computations and try to avoid to have much application logic because of multiple reasons:

- *Limited computation power available*, try to offload calculations to the server where possible.
- *Restricted set of data available*, makes it mostly impossible to compute certain values.
- *Duplication reduction* between applications as much as possible to improve on maintenance.

The reduction of application logic on mobile applications has as effect that some functionality is not offered in the mobile application or only on requests through a remote server.

Interface
The reason that there is a need for mobile applications is mostly because of the web UI is not easily accessible through the hardware of mobile devices. One of the biggest differences is the greatly reduced size and resolution of a screen and the use of touch instead of the conventional input devices. A simple solution that web applications offer are an modification of the pages to something that is accessible with mobile devices. Users are not satisfied with this solution because it is missing the look and feel of the native applications. Most mobile applications are reactive of style and optimized for touch devices with smaller screens. In general, the views are separated in smaller pieces and uses buttons and gestures to navigate easily between views.

Web applications and mobile applications can both offer interfaces for other software. Nevertheless, the functionality is not equivalent. Web applications have an interface to request data or functionality for other applications, where mobile applications might have an interface for other software to push notifications.

Security
Mobile applications have to deal with less security concerns because usually data is shared on a need to know basis. This implies that the mobile application only has data that is public available or within the boundaries of information that is available for the user. This restricted data set has as consequence that there is no need of protecting this data from the user. The local data is not available through remote access, so there is no need to protect against remote users. Sometimes applications choose for encryption of data as a security mechanism to keep private data inaccessible for stolen devices.

2.2.2 Hardware limitations

Web applications can run into hardware limitations when the applications are utilized by an increasing number of users. To keep the web application working and accessible within the normal time requirements additional hardware is needed. The mobile application developer runs faster into problems when it comes to hardware restrictions. The first cause is that the hardware is not within their control so they have no possibilities to know and modify the device. Secondly, is that the hardware of mobile devices has lower specifications and therefore more restricting on applications. The solution that is used for web applications is not working for mobile applications due to money and lack of control of the target hardware.

When taking the hardware limitations into consideration the first problem which arises is that mobile devices differ from each other on several properties. Another difficulty is that it is hard to have a good grip on the actual minimum on hardware requirements. The unknown parts of the specs are minimized where possible to keep the excluded group as small as possible. Those areas that are restricted by mobile devices are listed below with a description of the limitation it introduces.

Connectivity Mobile devices use Wi-Fi or a cellular connection to access the Internet. The mobile internet connections are slow and many providers only allow a restricted amount of bandwidth.

Computation power The processor of a mobile device is slower than those of desktops and use a less powerful architecture. Another disadvantage for the power of the processor is that it is more focused more on graphics calculations for the display output instead of general computations. The newer phones have multiple cores, however, that is not as sophisticated as the desktop variant.

Energy supply The energy supply of mobile devices is limited by the size of battery and the amount of energy that is consumed by the hardware.

Memory The memory in mobile devices come in two forms: Random-access memory (RAM) and storage. Both parts are small compared to the amount available on personal computers.

Input & Output The input and output devices of mobile devices are in general not interpreted as a restriction. however they are limited to the hardware of the device and differ from the conventional devices.

2.2.3 Development Platforms

There are several approaches to create a mobile applications: customized web pages, native applications, frameworks to generate mobile application, and DSLs for the mobile application domain. They can be split into two groups: The native and web solutions that are manually written applications (1). The framework and DSL solution abstract over common problems of mobile development (2). The next paragraphs describe the differences between mobile development platforms and are summarized in Table 2.2.

The specialized mobile pages are easy to create and take small amount of effort to develop. Just like a normal web page they get portability as free asset. Combined reducing the code duplication and costs to a minimum level. Still it is more common to develop mobile applications in a native form instead of a web variant. There are several other aspects which are in favor of the native application. The marketing reason would be that native applications are offered in a local marketplace and therefore the application is easier to be found and sold. In perspective of functionality, the native applications have access to more features of the device which could be a restriction that is critical for a mobile application. The developers take advantage of the development environment for mobile platforms that offers tooling and abstractions on UI elements, which makes it easier to develop and are also automatically optimized for mobile hardware [10].

	Web	**Native**
Implementation effort	+	-
Code duplication	+	-
Portability	+	-
Market place	-	+
Device feature access	+/-	+
Tool support	-	+
Mobile UI abstractions	-	+
Programming expertise	+	-

Table 2.2: Comparison between development platforms of mobile applications

DSLs and Frameworks try to abstract over aspects within the problem domain. The most popular frameworks for mobile applications are: Titanium[1] and Phonegap[2]. Both frameworks offer native functionality by wrapping the HTML5 UI in a native application with a browser widget. Additionally, they have some JavaScript Application Programming Interface (API) hooks to access device features like camera or contact list. The framework tries to combine the power of both web and native applications. The UI part is not as strong for current frameworks, Phonegap lacks total support of UI abstractions and Titanium has a solution library of small UI elements that are translated differently for different operating systems.

The differences between frameworks and DSLs is a more general question, which is similar to the WebDSL advantages in comparison to the popular web frameworks like better domain abstraction and error checking. In the case of mobile applications current frameworks try to combine the positives aspects of web and native applications, where the goal of a DSL is to have a more general view and abstractions on the whole set of sub-problems in the domain.

2.3 Target Languages

The implementation of the generative approach presented in this thesis needs target languages from both domains. The semantic details of the languages are not interesting for the synchronization problem, but a general overview of the languages is used as a

[1] http://www.appcelerator.com/platform/titanium-platform

[2] http://phonegap.com

guide and reference for some implementation details that are described further on in this thesis.

This section will give an overview on the target languages: WebDSL(Section 2.3.1) and Mobl(Section 2.3.2). Both languages are a DSL particularly developed for the corresponding domains of web and mobile applications.

2.3.1 WebDSL

Currently development of web applications often uses frameworks like: Ruby on Rails[3], Zend Framework[4] and Seam Framework[5]. Frameworks are constructed on General Purpose Languages (GPLs) and abstract over aspects of web applications by providing extensions or API to make life easier for programmers.

WebDSL[6] has the same purpose as those frameworks, to make life easier for web developers. WebDSL is a DSL which defines its own language instead of being based on an existing GPL. The compiler generates java code conform the Java Servlet API, which makes it possible to run code in a web container like tomcat[7]. WebDSL is build with Spoofax[8], which delivers in combination with a compiler an Integrated Development Environment (IDE) with editor services.

WebDSL agrees on the viewpoint of frameworks about abstraction and separation of concerns for areas of web development. WebDSL defines sublanguages for the different sub-domains as described in Table 2.3. The domains cover the aspects found in Section 2.1.2 and some of them are split up in to smaller parts to give better control and abstraction on the domain. All the sublanguages are combined through linguistic integration into one language. The sublanguages have similar syntax styles and can share common elements where semantically possible.

Type checking is one of the strengths of WebDSL. The language is statically typed making it possible to check for typing errors on compile time like Java. WebDSL is designed specifically for the web application domain. Which means that the semantics of the code gives the possibility to add extra checks and give errors that are more specific to the application, something a GPL cannot accomplish. The integration of the sublanguages makes it possible to cross check references and give feedback on compilation for unresolved references and incompatible usage. Spoofax provides inline errors in the editor, combining that with real-time analysis introduces the feature to show errors while typing, as shown in Figure 2.4. The integration and error checking on compilation is a great improvement of WebDSL compared to the popular frameworks. In frameworks errors often occur on run-time, making untested code conceivably fatal in a live system [19].

Further details about the syntax and semantics of WebDSL are not specifically interesting for the goal of this thesis, since other target languages with similar capabilities could have been used for this purpose. For the interested reader there are more papers with additional details of WebDSL [17, 18, 19, 34].

[3] http://rubyonrails.org

[4] http://framework.zend.com

[5] http://www.seamframework.org

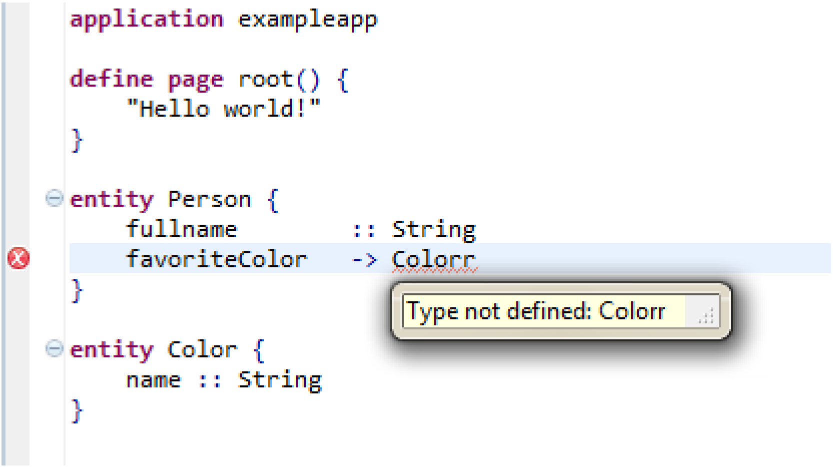
[6] http://webdsl.org

[7] http://tomcat.apache.org

[8] http://spoofax.org

Data Model	Language to define a persisted data model
Data Access	Built-in API to access stored data
HQL	Specialized query language to customize and optimize data access
Action	Java like language to define application logic
Tasks	Language to define reoccurring tasks
UI	Language to define User Interface design through templates and HTML
Services	Language for defining webservices
Access Control	Rule based language to define access control on UI elements
Validation	Check based language to define validation on data model and input
Search	Extension to provide indexing of entities and an API for search queries

Table 2.3: WebDSL sublanguages descriptions



```
application exampleapp

define page root() {
    "Hello world!"
}

entity Person {
    fullname        :: String
    favoriteColor   -> Colorr
}

entity Color {
    name :: String
}
```

Type not defined: Colorr

Figure 2.4: Inline display of an error in the WebDSL editor

2.3.2 Mobl

For this thesis the choice for a target languages on the mobile platform is Mobl, a DSL for mobile applications build with Spoofax. Mobl targets HTML5, JavaScript and CSS3 for easy portability and also offers support for integration with Phonegap, to generate native application. The design of Mobl is derived from that of WebDSL and shares the separation of concerns. However, the set is limited to a subset described in Table 2.5. The sublanguages differ for the biggest part in syntax and semantics from those of WebDSL to adapt for the mobile platform.

Compared to the frameworks Mobl offers an abstraction by reusable templates and a library, with UI elements and high-level controls, to give the applications a native look and feel. In the logic part Mobl uses a JavaScript like syntax with some abstractions and adds a type system. The best part is that Mobl rewrites normal functional calls into the form of the asynchronous style of JavaScript function calling.

Data Model	Language to define a (un)persisted data model
Data Access	Built-in API to access stored data
Action	JavaScript like language to define application logic
UI	Language to define User Interface design through templates and HTML
Services	Language for defining usage and handlers for webservices
Search	Extension to provide indexing of entities and an API for search queries

Table 2.5: Mobl sublanguages description

For the goal of this thesis Mobl is particularly interesting because of the automatic persistence of data and possibility to declarative define a model. With more effort other languages could have been used to generate the solution. For the interested reader, more details on Mobl are available in this paper [22].

2.4 Impact on synchronization

The previous sections tell the differences and the aspects that are introduced by web and mobile applications. The differences from conventional and web applications are not changing the problem of synchronization, except for the interesting fact of unknown users in web applications, which could be similar for the synchronization interface. On the other hand, mobile applications bear along hardware limitations and restricted software technologies, which deliver constraints for synchronization.

This section illuminates the impact of the domains web and mobile applications, broken down into software (Section 2.4.1) and hardware (Section 2.4.2) influences.

2.4.1 Software influence

It is certain that data plays an important role for synchronization, other areas of web and mobile application might as well interact with the synchronization algorithm. Therefore, the following paragraphs describe the relations between synchronization and the attributes that are found in Section 2.1.2.

Data Model

Depending on the synchronization algorithm the data model might be of high value, it is mainly used to interpret and map data between different formats. There is a high chance that mobile application needs another representation for the data model, since not all data might be shared and the technologies for data storage are not similar to those of desktops. In the situation of the solution, presented in this thesis, the data model is indeed needed because it uses different storage facilities and data representations.

Data Access

In synchronization data has a central position, therefore it would be an improvement if the access would be easy and without too much overhead. At the same time, there should be no limitation in the accessing data, since it might be necessary for the synchronization algorithm to specify queries that are not normal within the normal application data flow. In the case of WebDSL and Mobl, they have both abstractions and possibilities to manually specify queries, both on top of a specified framework. Those frameworks offer auto persisting of data. However, that means less control over the process to store and retrieve data, which might be needed for custom behavior.

Application Logic

Application logic in viewpoint of data synchronization is not interacting with the algorithm. The synchronization needs to take into consideration that application logic works at the same moment with the same data. For WebDSL the interfering is not a problem, since the persistence framework abstracts over those problems. The synchronization algorithm could be seen as part of the application logic since it is a glue layer between data and UI. It might also be regarded separately because it is not part of the business logic of the application.

Interface

Synchronization is done between multiple instance, to accomplish this, a communication layer is required. In the case of mobile and web applications, communication is normally implemented as webservices. More specifically RESTful webservices are often used because they are simple and have a minimal overhead on the data representation and communication.

Security

Security is an important factor for web applications. Therefore, it is also essential for synchronization with web applications. The sharing of data should be restricted based to specific user rights. There also exists a need to detect if the current principal is allowed to modify the data. WebDSL has a sublanguage which integrates security throughout the whole application and that is a good starting point to use or extend for synchronization purposes.

2.4.2 Hardware influence

The hardware limitations that are introduced by servers for the synchronization are similar to those of the application itself. It only deviates in the part that web page requests are smaller though, occurs often. Where synchronization requires more calculations however, on less frequent basis. The solution on scalability is similar to that of the application itself by improving or adding hardware. The limitations by mobile devices that are described in Section 2.2.2 have a bigger influence on the synchronization and will be reviewed per topic in the following paragraphs.

Connectivity

An internet connection is a required asset for a synchronization solution. Synchronization needs the connection to communicate between applications. Another influence is that synchronization cannot depend on a stable full-time connection to the server. The bandwidth and speed of internet connection motivate the synchronization to reduce the data traffic as much as possible or postpone the process until it has a better connection.

Computation power

There is always a certain amount of computation needed to use the synchronization algorithm on a device. However, to have a usable solution it is preferred that those computations are within the bounds of reasonable time consumption so that the mobile application is still usable and reactive.

Energy supply

Energy supply itself does not interfere with functionality of an application. There is a more general problem with energy usages. That is the use of internet connection and processor significantly increase the energy consumption. It would be good for the synchronization solution to keep the time of computation and connectivity low to increase the usability of the application.

Memory

The two parts of memory each have their own functionality in the device. RAM is used for access to data with increases speed, but is not persisted. When the data of the application does not fit in the memory it needs to use storage, which is much slower. This is general problem of insignificant amount of RAM, which leads to slower computations.

Storage of persistent data is done on other hardware and is in general bigger than RAM. The problem for synchronization is that it needs to store the retrieved data. Nevertheless, it can never fit same amount of data that is stored in database of the web application. Simplifications and restrictions are needed to address this issue. Although this might not be enough in the case of internet applications the maximum storage facility is reduced even further. Where IOS[9] tops everything by a limitation of a few Mb that can be increased by the user to 50 Mb maximum [16, 38].

Input and Output

The input and output where strictly not restrictions. They do not contribute any problems to the synchronization because they only have an impact on the GUI of the mobile application.

[9] `http://www.apple.com/nl/ios/`

2.5 Summary

The last decade there has been a movement of conventional software to web and mobile application development. Applications that both offer a web and mobile version run into the disadvantages: Code duplication and a need for communication layer between different application domains.

To have deeper analysis this chapter presents the three different areas where web development differs to conventional software: intrinsic properties, stakeholders and discipline variety. Secondly, evaluates the most common aspects used by web development (Section 2.1.2) which are displayed in an overview in Figure 2.1. Mobile applications share most of those aspects of web development. Nevertheless, the interpretation is different as described in Section 2.2.1.

Mobile applications are currently available in two variants: a customized web application and a native application. Both have their own advantages as summarized in Table 2.2. Frameworks for mobile applications try to combine the power of both platforms by wrapping web application in a native application with browser widget.

This chapter also introduce the two target languages WebDSL (Section 2.3.1) and Mobl (Section 2.3.2) for the corresponding domains of web and mobile applications. Both are a DSL written in Spoofax and include separation of concerns by sublanguages which can be found in Table 2.3 and Table 2.5. While both languages have their advantages over other solutions, they could be substituted with other languages.

The concluding part, Section 2.4, takes the aspect found earlier in this chapter and use them to give extra context in the synchronization framework. The hardware limitations of mobile devices, specifically those of connectivity, computation and memory have a considerable effect on the synchronization. In the following two chapters you will find the details of synchronization algorithms and uses the insight gathered from this chapter to formulate the requirements.

Chapter 3

Data Synchronization

Synchronization is a well researched problem and is used within diverse ranges of software applications. The problem is having data that is shared with multiple hosts. Each host has full copy of the data and require that their data is consistent with all other hosts. Direct connection with all hosts would make it possible to directly modify all objects on the different host. However, it is practical impossible to prevent conflicts and change all data on all hosts without blocking data access before applying. Additionally, it is not practical to require that all host are always connected for the system to be working. This shows the complexity and the core of data synchronization: identify updates in certain time range for host combination. Detection is needed within the system because multiple host could have changed the same data. A deterministic resolution approach is needed to acquire consistency of data over all hosts.

Data synchronization is used within multiple disciplines of computer sciences like: databases, file systems and version control. In big software solutions database systems are often replicated over multiple systems to distribute load or use as fallback. In this case data synchronization is used to keep databases equal to each other. In the case of file systems, synchronization is used to have files backed up or make it possible to easily share or cooperate on same documents. A popular application that is used for this purpose is Dropbox[1]. Version control systems for source code, like git or subversion, use data synchronization solutions to deal with distribution of change sets.

The current developments of cloud services made synchronization into a hot topic again. This thesis has the goal of data synchronization focused on web applications with multiple instances of the mobile application.

The previous chapter introduced the context of web and mobile applications. This chapter will focus more on the core of synchronization, presented as theoretical models with their pros and cons (Section 3.1), and a description of the activities that play a role in synchronization (Section 3.2).

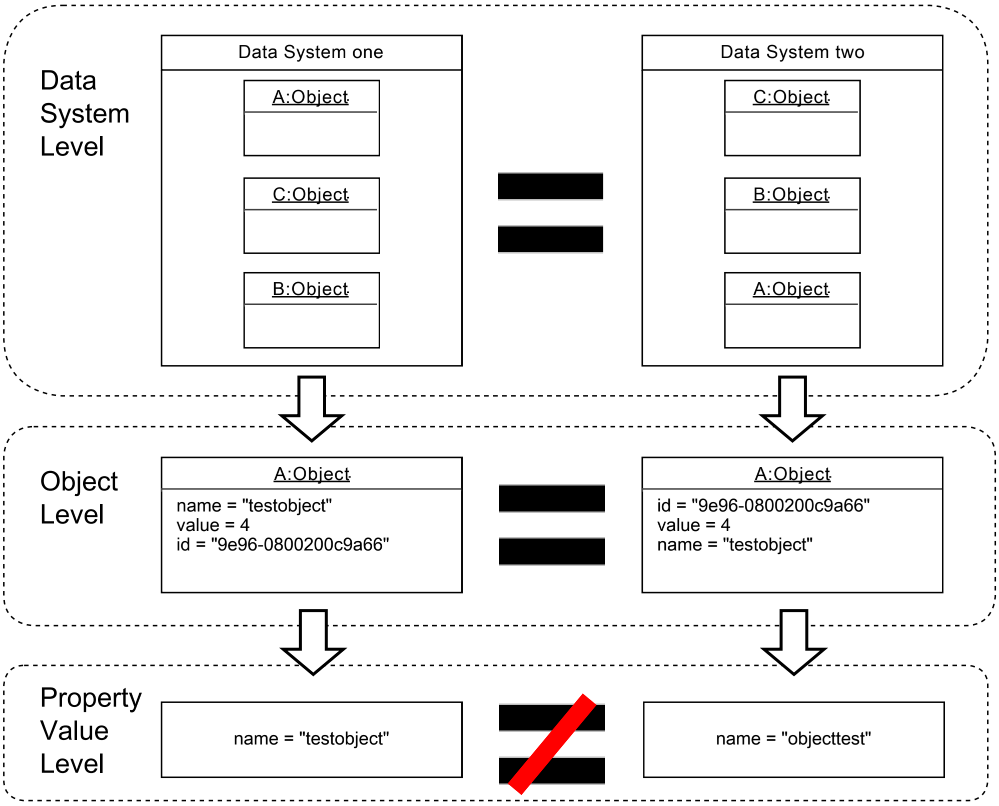
3.1 Theoretical Models

The theory splits data synchronization up into two domains: ordered and unordered. The difference can easily be explained by an example. In a text document the string “a b c” has another meaning than “ a c b ”, which means it is ordered data. In set theory

[1] `https://www.dropbox.com`

{a, b, c} is equivalent to {a, c, b} and therefore categorized as unordered. A similarity can be found in theoretical computer science, when inspecting collections, lists present ordered data and bags unordered data.

A first look would consider data with a data model as ordered data because it is structured, mostly uniform represented and some of the property values are definitely ordered. This assumption would make it hard to have an efficient solution for the synchronization problem. However, reality shows that it is a combination of the two, breaking the model representation down to a level of objects and separate the object to only include its own information makes it possible to handle the objects as unordered. This theory also holds on level of properties. However, the value in the properties must be handled as ordered data. An overview of ordered and unordered in object based systems is shown in Figure 3.1, where equality between two representations mean that data is unordered.



Equality means that both data systems represent same data because of unordered data

Figure 3.1: Combination of ordered and unordered data in an object based systems

For the solution of this thesis we could simplify the problem by perceiving the data as unordered. This means that the resolution of problems on property values can not be solved by merging, but has to be solved by selecting one version.

The unordered variant of the problem is also known in algebra as the set reconciliation problem. This problem is based on two remote clients, which each has a set of (different) integers. For both to know the updates they need to calculate the difference of both sets. The extra restriction is that the solution has to restrict the synchronization to a minimum amount of communication.

The next part describes methods that currently exist in unordered data synchronization and can be roughly separated in the following groups: wholesale, mathematical and incremental approach.

3.1.1 Wholesale Approach

The wholesale algorithms contain a straightforward approach. At the moment of synchronization sends all local data on the device to the remote device. The other device computes the differences and sends back the updates as displayed in Figure 3.2. The Slow sync algorithm of the Hotsync technology from Palm OS uses this technique to make it possible to synchronize the mobile device with multiple computers [26].

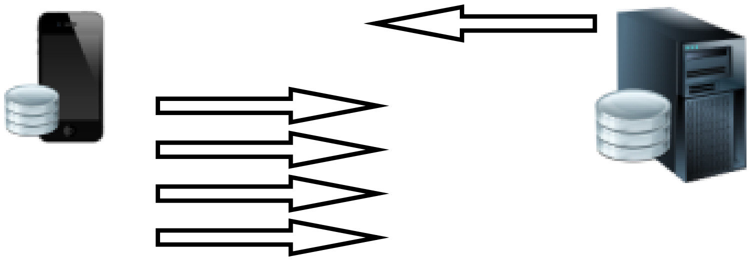


Figure 3.2: Graphical representation of the wholesale approach

3.1.2 Mathematical Approach

The mathematical approach consists of solutions for the set reconciliation problem, which are mapped to the synchronization domain (Figure 3.3). The computation of the symmetric difference between two sets of integers on different host with minimal communication can be mapped, since all data can be represented in integers. This mapping allows it to translate the set reconciliation problem into synchronization problems that are encountered in reality. The algorithm takes care of the core problems and minimizes on communication. CPIsync has implemented an algorithm of this group for PDA synchronization and shown big improvements over the slow sync algorithm [32]. Even if the solution would be optimal for the set reconciliation problem it does not have to be the best solution for practice, since it could use extra information to make the problem easier.

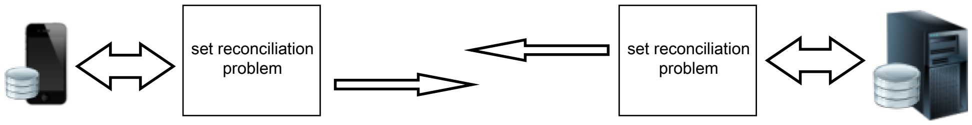


Figure 3.3: Graphical representation of the mathematical approach

3.1.3 Incremental Approaches

The most used synchronization variant currently are algorithms that use an incremental approach. A better description is that those algorithms only synchronizing changes of a certain time range, namely those in-between current and last synchronization as displayed in Figure 3.4. In general, the two following ways of tracking changes in certain time range can be distinguished: object versioning and logging changes.

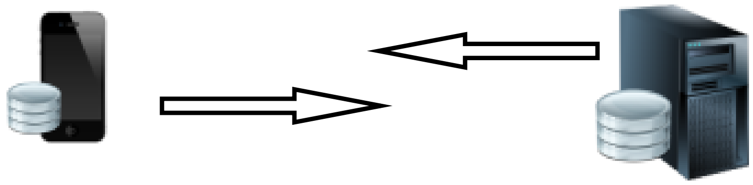


Figure 3.4: Graphical representation of the incremental approach

Tracking the status of an object with versioning is done by adding extra data to an object which stores version information. This extra information is used to only sent changed data instead of sending all data. There are several ways to implement versioning of objects. The solution often depends on the context of the synchronization problem.

Flag Approaches

For one-one synchronization the easiest ways is to track changes by setting flags for objects that are modified after the last synchronization. This only works if there is one remote point where it synchronizes with. An implementation of this approach is used in the Fast sync algorithm, part of the Hotsync technology [26] and Intellisync [29] solution from Nokia.

To adapt this method and make it usable for synchronization between multiple devices, additional flags are needed for every device. Flags for modifications are not really scalable, at a certain point the version information in an object will be bigger than the original data, which blows up the data increase on object creation. Several implementations of the SyncML [9] protocol use this solution to keep track of differences between devices.

Version Approaches

A fix that reduces the number of flags, is to keep track of version numbers on objects and compare the number to detect if the object has changed after the last update. An improvement on the weakness of this idea, is to use timestamps instead of version numbers on objects. This works because timestamps can be used to give a version that is can be used globally for all objects. This means that the timestamp, of the last synchronization with a certain device, is enough to calculate the changed objects in that time range.

Change log Approach

The second approach, that is used for tracking changes on data, is tracking changes in a log. The algorithm searches for all the changes that have occurred ahead of the last synchronizations and sends those to the other device to replay the changes on that device and vice versa. It could also work on sorted data, if the changes would have additional position information. This method works on a finer grained level which helps improving conflict resolution. Change log is an approach popular in source code revision control solutions.

3.1.4 Comparison

The previous paragraphs described the approaches to synchronization in theory and some of their properties. To have a better overview of the methods we compare on the following attributes:

- **Memory** The impact of the method on memory. For example, the additional memory that is needed.
- **Computation** The calculation that is needed with a focus on the impact for mobile devices.
- **Bandwidth** The amount of communication that is used in addition to sending the changes.
- **Network** The network architectures that are allowed by the approach: one-to-one, one-to-many and many-to-many

The wholesale approach biggest disadvantage is that all data is sent to the other host, which is very bandwidth expensive. The wholesale requires from the host to determine the changes by comparison. This is not a complex computation, but a high amount of data could make it expensive.

The mathematical approach uses the solutions of the set reconciliation problem, which is optimized for minimal communication. Nevertheless, this optimization comes at price of more computation. Adding a step of mapping from data to integers introduces overhead in computation and bandwidth.

The incremental approaches in general try to optimize on communication cost. The single flag approach is simple and has a minimum amount of overhead. The bigger problem comes that it is only applicable to synchronize with one device. Increasing the number of flags to the number of devices is a possible solution. However, applying this to a network with many devices would mean a high overhead compared to the size of the objects.

Using versions as replacement of flags reduces the overhead on the objects, but requires additional computation to identify the objects. The comparison needs to know version numbers for objects, which increases the bandwidth for sending version data. The timestamp as version reduces the bandwidth and makes it possible to query for changes instead of comparison for each object.

Tracking changes in a log is more expensive in space if there is a high number of changes since it needs to keep track of all changes. However, when the change log can be cleaned often it is not that expensive. The synchronization requires merging of changes instead of simple conflict resolution.

The weaknesses and strengths of the various approaches in those topics are partly described by Agarwal et al. [7, 8]. A overview of the comparison is given in Table 3.5.

In general, the timestamp approaches seems to be the most scalable. More specifically, depending on the context of the synchronization problem, the flag, timestamp versioning or change log are the best known approaches.

				Incremental				
		Wholesale	**Mathematical**	*F*	*MF*	*OV*	*TV*	*CL*
Memory		+	+	+	- -	+	+	+/-
Computation		+/-	- -	+	+	-	+/-	-
Bandwidth		- -	+	+	+	-	+	+
Network	1-1	+	+	+	+	+	+	+
	1-n	+	+	+/-	+	+	+	+
	n-n	+	+	-	+	+	+	+

F = Flags, MF = Multiple Flags, OV = Object Version, TV = Timestamp Version, CL = Change Log
- - = Verry Weak , - = Weak, +/- = Neutral, + = Strong

Table 3.5: Comparison between the synchronization approaches

3.2 Activities

Synchronization is often seen as a one step process of interchanging data, but this is not correct. The following three steps are separated in the process of synchronization [13]: identify updates, propagate updates and detect and resolve inconsistencies. The following paragraphs will describe in more details the responsibilities of those three procedures.

Identification of updates

The identification of updates is interesting when working with incremental updates instead of the wholesale method. Wholesale does a comparison to find changes. After sending all the data, this problem is trivial and will not allow any optimization. The identification of updates is not always seen as part of the synchronization itself and is regularly supplied by the actual data storage facilities. That facility has better control and overview on changes of the data and will be better in the view of separation of concerns. So in general the identification of updates could be given as parameter to the synchronization algorithm or should be accessible through easy queries on the objects properties. The following three type of modifications on data can be distinguish: mutation, creation and deletion. Those three groups in general require different interpretation, which means that the algorithm needs this as additional information next to the fact of modification.

Propagation of updates

The distribution of changes to the other client is the core of the synchronization. It is a simple step that takes care of the incoming requests for synchronization and maps data updates into serializable changes. The actual representation of data and execution order of this part depends on the algorithm, but in general it represents the communication layer for synchronization. This activity is in addition responsible for applying the changes to the database if they are allowed.

Detection and resolution of inconsistencies of data

The actual changes that are applied to the data could deliver inconsistency. This will be problematic to the application that is using the data. It is preferred to check the changes before finalizing them to storage. Meaning that this process must be weaved through the actual propagation of updates to find errors early on and regard or fix the modification. There are several ways to have inconsistencies, the normal conflicts are updates on objects that are already locally modified. Another way is introduced by users that could not be trusted. Those users need to be checked whether the data received is conform the specifications. Depending on the demands and the algorithm there are various ways to fix those problems, where the easiest way is to regard the changes.

A more single fitting purpose is security, which might be considered data inconsistency when a person changes data that it is not allowed to change, but that does not apply to all security issues. It could be perceived as a combination of this process and propagation of updates.

3.3 Summary

Data synchronization is a well researched subject and even has a mathematical representation of the problem (set reconciliation problem). The theory distinguishes three categories of synchronization algorithms (Section 3.1):

- **Wholesale** (Figure 3.2): All data is transferred to the remote client that calculated the differences.
- **Mathematical** (Figure 3.3): Solutions of the set reconciliation problem are mapped to the synchronization problem concept.
- **Incremental** (Figure 3.4): Solutions that track changes and only send changes from a specific time range to acquire incremental updates.

Each of the solutions has its advantages and disadvantages that are summarized in Table 3.5. The synchronization process can be broken down into three steps: update identification, propagation, and consistency detection and resolution (Section 3.2).

The next chapter will continue with the information gathered from this and previous background chapters to create requirements for a data synchronization framework between web and mobile applications.

Chapter 4

Synchronization Framework Requirements

The previous part has described the context and the theory of the synchronization problem for this thesis. This information is needed to formulate the requirements of the solution, which are useful as guideline for building and evaluating the solutions presented in this thesis.

There are several views regarding the tool requirements. The first separation can be found between the core synchronization problem and its context delivered by the web and mobile platform. The partitioning of the tool itself can be separated in the following two parts: The application developers part and the part that is actual used by the application. Both parts have different stakeholders and deal with different aspects of the solution.

There is also a general breakup of requirements, which is used in this section: functional (Section 4.1) and non-functional (Section 4.2) requirements.

4.1 Functional Requirements

The final solution should generate from extended WebDSL code a synchronization framework that allows mobile devices with a Mobl variant of the application to synchronize a restricted set of data with the WebDSL application. The solution will be focused on the generated Mobl part and integration. Nevertheless, it should be possible for other remote applications to use the synchronization framework. The functional requirements described below are summarized in Table 4.1.

4.1.1 Data Synchronization

The main part of the solution is the synchronization of data and can be split up in the processes as described in Section 3.2.

Identification of updates
The role of identification of updates in the solution is to find a generic way to get the updates after the last synchronization. The actual functionality of keeping track might not be part of the synchronization, instead it should be a low cost solution to find the updates. The web application needs a solution to acquire updates for all devices, while the mobile application only needs to keep track of its own changes.

Propagation of updates
The updates should be sent or received on request from the mobile application. To make this possible there is a need for webservices to send requests to the web application. Both applications also need a mapping from and to a serializable representation of the data model to communicate changes.

Detection and resolution of inconsistencies of data
The updates could bring along inconsistencies and therefore should be checked. This can only be limited to available knowledge that is declared in the code of the application. The second source of problems with data are changes on objects that are already changed by another source, which should be detected by the algorithm. Next to detection it should prevent that those updates are persisted, which creates an erroneous state of the database. This could be done by reverting or fixing the updates.

4.1.2 Interface

While the solution might try to reduce the interaction with users to a minimal, but there is still a need to define interfaces for each user. The interfaces can be separated for each of the following users groups:

Application developer In this group there can be a separation between the developer of the web and mobile application. The web developer needs an interface to define synchronization specific features. For the mobile application developer there is a demand for an interface that can be called within the actual application to send synchronization requests and interaction with the algorithm where needed.

Application user The actual user of the application should not experience a change in the web interface. For the mobile application there is a new interface itself, but that is not part of this solution. The actual experience with the end user should be limited to none. On the other hand, the developer might use part of the generated code for the interface to display status of the synchronization process.

Remote application developer There should be a possibility for a remote application to use this synchronization interface. This would be the same interface for the generated code. However, the remote application should create its own synchronization implementation for his software to supply the interface with the correct information.

4.1.3 Data restriction

In general, mobile applications do not have access to all data because of hardware limitations and security reasons. The synchronization framework should have similar functionality by restricting the data in several manners as stated below.

Global restrictions It should be possible to remove or simplify the data model that is synchronized with mobile devices. For example, remove properties that are application specific and not used by the mobile application.

User specific restrictions Only global restrictions are not enough, it should be possible to restrict data to a level that only part of the user group can retrieve or modify the data. For example, a user can only get his own address information and no other users are allowed to retrieve this private knowledge.

User defined restrictions Since there is a possibility that applications have a huge data set to transfer. The solution must be able to only synchronize parts of the data. The user must have the possibility to influence on which data is synchronized. At the same time, it should be usable. Those requirements forces the solution to partly derive the data selection.

4.1.4 Code generation

This thesis presents a generative approach to the synchronization problem and because of that a part of the requirements is generation of code. The generation can be split up into WebDSL and Mobl code and the required parts are listed below:

- WebDSL
  - mappers for objects to a serializable representation
  - mappers from serialized object to an internal data representation
  - synchronization algorithm including identification of updates and inconstancy handling
  - webservices for interaction with the synchronization by remote applications
- Mobl
  - modified model representation of the data model defined in WebDSL
  - mappers for objects to a serializable representation
  - mappers from serialized object to an internal data representation
  - functions to abstract calls to webservices
  - integration of synchronization parts into general functions

4.2 Non-Functional Requirements

The focus of the implementation is on fulfilling functional requirements, while often the success depends more on the non-functional qualities of the software. The previous section covers the functional aspects while this section will describe the requirements on non-functional topics: Usability, scalability, applicability, adaptability, security and robustness.

4.2.1 Usability

Nielsen et al. describe usability as: How well can users use the functionality [28]. In Section 4.1.2 already introduced the group of users and the specific interface for those users and is used in this section to define some usability requirements for the solution.

Goal	Requirement
Data synchronization	• Identification of updates on mobile and server side • Mapping data between serializable and local format • Sending updates between mobile and web application • Detection of inconsistencies in updates • Resolution of inconsistencies in updates
Interface	• Interface to customize and use generated code • Interface for synchronization process status • webservices for other applications to access data
Data restriction	• Restrict data model for synchronization • Restrict access to data based on user rights • Partial data synchronization selectable by user
Code generation	• Model representation for Mobl • Mappers to (de)serialize objects • webservices for communication • Synchronization algorithm • Integration functions

Table 4.1: Summary of partitioned functional requirements

Application developer

The developers of the WebDSL application needs to be able to enable the correct synchronization behavior with simple additions to the application code. The additional syntax needs to be in same style as the other sublanguages of WebDSL and have to be declarative, so that it is easy to understand the semantics of the written code. When it is possible, the solution should reuse information of the application code instead of adding duplication. Identically the Mobl developer should have almost no work to integrate the generated code into the application. Minimal and natural are the keywords to strive after for this user group.

Application user

The actual users do not have direct contact with the generated code. Still, it is part of the application and on execution could interfere with the usability of the total application. There are two general characteristics of mobile applications where the solution could interfere: reactiveness and offline capabilities. The reactiveness constraint is that while using the synchronization algorithm the mobile application or at least the GUI

Goal	**Requirement**
Usability	• Simple interfaces for users • Minimalistic interaction
Scalability	• Must be able to handle multiple clients • scale with bigger amounts of data
Applicability	• The framework can be generated for all data models that can be specified within the web application
Adaptability	• Modification possibilities to tweak generated code • Synchronization algorithm should not require changes
Security & Robustness	• Possibility to define access control on request/modification of data • Should be able to deal with non valid input of webservices • Protect database against invalid updates

Table 4.2: Summary of the non-functional requirements

part of it should not be blocked. The offline capabilities can be read as that an application is not obligated to be online all the time. This constraints that the synchronization cannot depend on its connection with the server and have to cache the changes locally.

Another problem is hardware limitations on mobile devices and should be considered while developing the algorithm. The following properties are influenceable by the solution: bandwidth, computation and memory. Memory and bandwidth can only be limited to size of actual data and requires to use an incremental approach. The solution for computation can be found by offloading computation work to the server side as much as possible.

Remote application developer

The remote application developers are not the main target group for the solution, but should be at least taken into account that they could use the synchronization webservices. Therefore, they should be simple and not send Mobl specific data. It should be possible to easily retrieve data from the application and with its own implementation of keeping track of changes could use the synchronization part as well.

4.2.2 Scalability

The scalability of the solution has two viewpoints: the number of clients it can handle to keep working and the size of data in the application without failure. The first viewpoint is a concern that is dependent on the usage of the application and how often people actually would synchronize their mobile client. Assuming synchronization is not required regularly, this problem is not that big and could be solved partly by introducing more machines with replicated databases.

The scalability of size of data is more interesting topic of synchronization, since this has a high influence for usage of the tool, for bigger applications. Smaller applications could already have a reasonable amount of data stored. Users do not see the direct problem. Although they do not like to wait too long for the data to be retrieved.

4.2.3 Applicability

The target for the software is web and mobile applications in general. This means that the generation, which is based on the sources of existing web applications, should allow to generate a working synchronization framework for every web application. The synchronization is based on the defined data model, meaning that the generation needs to cover all possibilities that are allowed in the model of the web application. However, it could be that technical limitations in the mobile application target languages restricts the possible options.

4.2.4 Adaptability

The goal for this solution interface is to be a minimal and simple for the users, this comes at a price of assumptions and default solutions. The synchronization algorithm itself should be complete and not require any modification. Some other parts of the generated code that influences the process or data could be customizable. For example, restriction on data is already part of the functional requirements. Other parts like data mappers and default integration functions in Mobl code are possible targets that need application specific adaptions.

4.2.5 Security and Robustness

Security issues could find place on the mobile and web variants of the application. The data restrictions (Section 4.1.3) should limit the security issues on the mobile devices. The web application has a harder job with security, since it cannot trust the clients because of the following two reasons: Clearly, the situation is that the synchronization webservices are open to any software and therefore cannot be trusted in general. The second user is the Mobl client that is generated and could be seen as safe. Nevertheless, both user and mobile application developer could change the data in a way that is not valid for the web application. To be sure that is not the case it is better to assume that also those clients cannot be trusted.

Similar to data restrictions on sharing data, the requirements state functionality for access control on data modifications. The web application developer remains responsible for security on the application and needs to define specific rules for data restrictions. Another possible breach that the framework needs to keep in mind is SQL injections, an attack which is a popular method to modify the database and gain control over an application [23]. The framework used for accessing data could already implement this. When this is not the case, input strings should be validated or escaped to prevent those attacks.

Robustness has to deal with the same factor as the previous paragraph that the users of the webservices cannot be trusted to deliver an input that is conform the expectations of the algorithm. Validation on the input is a part of the solution, this mean that at least the data is conform the presumptions. The input format of the webservices for changes could be different in format and it should not crash the application or break the algorithm.

4.3 Summary

The chapter presents requirements for a generative solution for synchronization between mobile and web applications. It uses the previous information of Chapter 3 to determine conditions for synchronization. The information from Chapter 2 is applied to add and adapt requirements for the domain of web and mobile applications.

The requirements are divided into functional (Section 4.1) and non-functional (Section 4.2) concerns. Generally, the solution must have the following functionality:

- Data synchronization framework
- Interface for all the user groups
- Data restriction possibilities
- Code generation for both web and mobile part of the framework.

The non-functional requirements are more general view of topics that contribute to the success of the tool. Important factors for this tool are: Usability, scalability, applicability, adaptability, security and robustness (Section 4.2.5). An overview of the most important requirements is summarized in Table 4.1 and Table 4.2.

The requirements described in this part will be used while developing the motivating example (Chapter 5) and the final solution (Chapter 6 and Chapter 7). They are also used to evaluate the result of the software.

Chapter 5

YellowGrass Mobl Motivating Example

In the previous chapters a theoretical approach is used to find data synchronization problems and state the requirements for the problem of data synchronization between web and mobile applications. We have chosen for a motivating example to discover and attempt to solve core problems before the abstraction and generation would be introduced. The YellowGrass application should show difficulties of the synchronization concepts in practice.

This chapter describes the motivating example. Starting with the approach description in Section 5.1 and introduction to the application YellowGrass[1] in Section 5.2. The second part gives an overview of the design (Section 5.3) and the implementation details (Section 5.4). Finalizing this chapter with an evaluation (Section 5.5) of the motivating example.

5.1 Approach

The synchronization problem within a new context of web and mobile applications is complex in such a way that direct implementation of the generation would be a naive approach. In this thesis we have chosen to follow the approach that is used by E. Visser in "WebDSL: A Case Study in Domain-Specific Language Engineering" [34]. The main steps are domain analysis, motivating example, generalization and generation as displayed in Figure 5.1. The domain analysis focus is on the theory of the problem and is in this thesis described in Chapter 2 and 3.

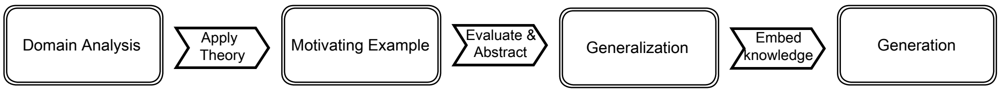


Figure 5.1: Approach used for developing this thesis solution

[1] http://yellowgrass.org

The motivating example stage is meant to implement a solution that is based on the theory delivered by the domain analysis. It adds a practical element that shows unexpected problems on implementation and integration aspects of the problem domain. The code base of the implementation is used as basis for the generalization.

The generalization is achieved by evaluating to find and solve shortcomings of the implementation of the example. Additionally, it requires abstraction on the example to have a generalized solution, which is usable for the generator. The generation is the final step in the approach and implements generation of the solution for a wider range of applications. This step is reached by making templates out of the generalized solution.

This chapter describes the architecture and implementation of the motivating example and an evaluation, which is used for the generalized solution. The outcome of the stages: generalized solution and generative approach are described in detail in Chapter 6 and 7.

The basic application was built from a simplified model of the application and only data was retrieved through some basic webservices. There were several iterations to improve the application and synchronization. Starting with simple retrieval of data, the following steps were implemented for synchronization: sending changes, retrieve updates and sending creations. The additional functionality was implemented during previous steps. This functionality is more focused on the context of mobile applications like: authentication, offline capability and validation. Those steps drove some of the changes in the framework. This chapter will use the last version of the motivating example to describe the details of the implementation. This includes interesting changes that were applied to improve the solution.

5.2 YellowGrass

This section describes the YellowGrass application which is used as basis for the mobile application. The description of the application is separated into an explanation of the functionality, the important pages of the GUI and the model.

5.2.1 Functionality

YellowGrass is a tag based issue tracker written as web application in WebDSL. The basic feature is to report and keep track of issues of (software) projects, through comments and status events. As stated in the first sentence it is tag based and those tags are used for the following purposes:

- grouping of issues on topics
- voting and following of issues by users
- assign issues to team members
- tagging of tags

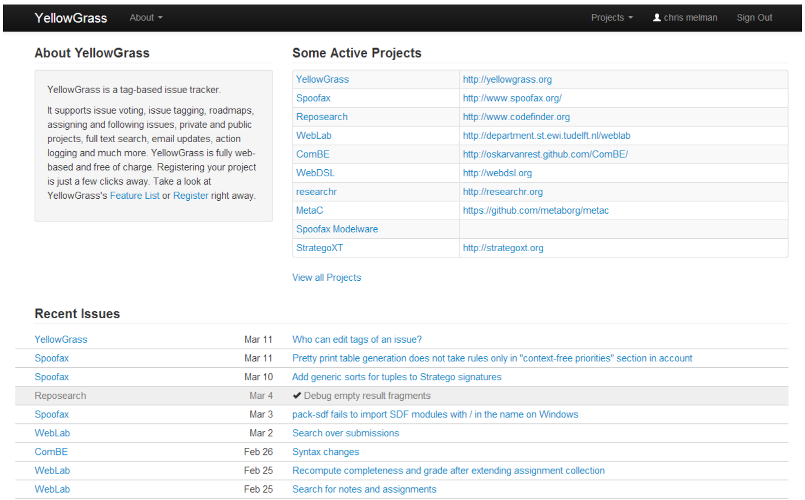
The tagging of tags (meta tagging), is meant for grouping of tags, currently this is used to mark special tags. For example, issues with a release tag are used to display a roadmap of the project.

5.2.2 Graphical User Interface

The previous sections describes the functionality of the application. Much of this functionality is linked with the GUI. This section displays the more interesting pages of YellowGrass and explains the information shown on the pages.

Home Page

The home page of YellowGrass is displayed in Figure 5.2. It gives an overview of the popular Projects and Issues. The top of the bar contains information of the current user.



About YellowGrass

YellowGrass is a tag-based issue tracker.

It supports issue voting, issue tagging, roadmaps, assigning and following issues, private and public projects, full text search, email updates, action logging and much more. YellowGrass is fully web-based and free of charge. Registering your project is just a few clicks away. Take a look at YellowGrass's Feature List or Register right away.

Some Active Projects

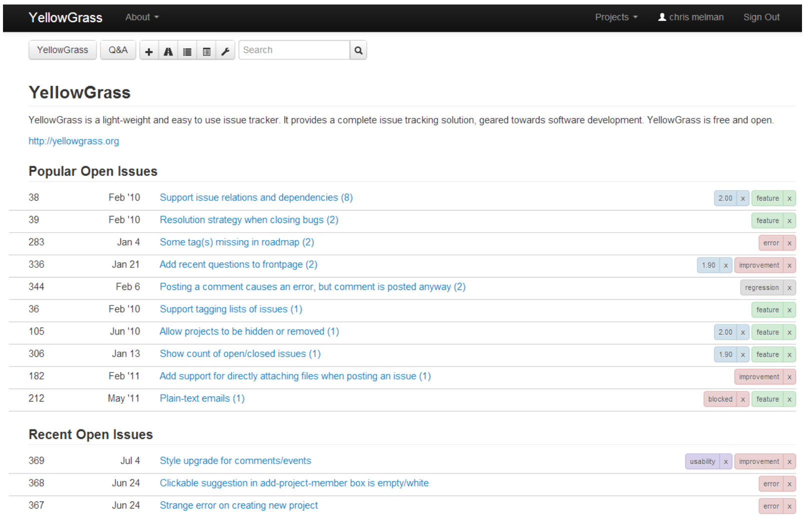
YellowGrass	http://yellowgrass.org
Spoofax	http://www.spoofax.org/
Reposearch	http://www.codefinder.org
WebLab	http://department.st.ewi.tudelft.nl/weblab
ComBE	http://oskarvanrest.github.com/ComBE/
WebDSL	http://webdsl.org
researchr	http://researchr.org
MetaC	https://github.com/metaborg/metac
Spoofax Modelware	
StrategoXT	http://strategoxt.org

View all Projects

Recent Issues

YellowGrass	Mar 11	Who can edit tags of an issue?
Spoofax	Mar 11	Pretty print table generation does not take rules only in "context-free priorities" section in account
Spoofax	Mar 10	Add generic sorts for tuples to Stratego signatures
Reposearch	Mar 4	✓ Debug empty result fragments
Spoofax	Mar 3	pack-sdf fails to import SDF modules with / in the name on Windows
WebLab	Mar 2	Search over submissions
ComBE	Feb 26	Syntax changes
WebLab	Feb 25	Recompute completeness and grade after extending assignment collection
WebLab	Feb 25	Search for notes and assignments

Figure 5.2: YellowGrass homepage



YellowGrass

YellowGrass is a light-weight and easy to use issue tracker. It provides a complete issue tracking solution, geared towards software development. YellowGrass is free and open.

http://yellowgrass.org

Popular Open Issues

38	Feb '10	Support issue relations and dependencies (8)	2.00 x feature x
39	Feb '10	Resolution strategy when closing bugs (2)	feature x
283	Jan 4	Some tag(s) missing in roadmap (2)	error x
336	Jan 21	Add recent questions to frontpage (2)	1.90 x improvement x
344	Feb 6	Posting a comment causes an error, but comment is posted anyway (2)	regression x
36	Feb '10	Support tagging lists of issues (1)	feature x
105	Jun '10	Allow projects to be hidden or removed (1)	2.00 x feature x
306	Jan 13	Show count of open/closed issues (1)	1.90 x feature x
182	Feb '11	Add support for directly attaching files when posting an issue (1)	improvement x
212	May '11	Plain-text emails (1)	blocked x feature x

Recent Open Issues

369	Jul 4	Style upgrade for comments/events	usability x improvement x
368	Jun 24	Clickable suggestion in add-project-member box is empty/white	error x
367	Jun 24	Strange error on creating new project	error x

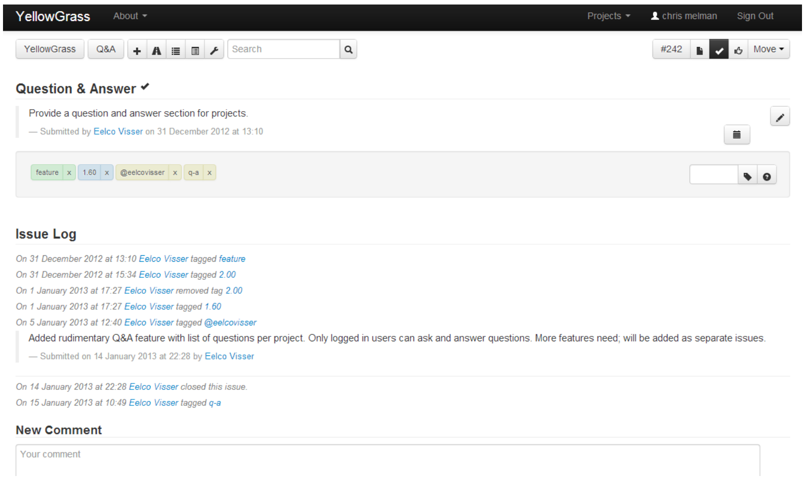
Figure 5.3: A YellowGrass Project page

Project Page

The project page of YellowGrass shows is a portal to issues, tags and member of the project. It allows to edit the project and navigate to wide range of related pages. Figure 5.3 shows an example of such a project pages. The page contains a description of the projects, interesting open issues and a navigation bar. This bar includes different navigation options for issues, a link to the roadmap and possibility to create a new issue.

Issue Page

The issues contains the most important data of the application. This makes the view an important part of the application as well. Figure 5.4 shows an example of an issue page. It displays basic information about the issue like description and title. Additionally, it contains status information in the form of a log. This log shows changes and comments made to the issue. Another important aspect is that it has an interface for managing tags on a issue.



Question & Answer ✓

Provide a question and answer section for projects.
— Submitted by Eelco Visser on 31 December 2012 at 13:10

Issue Log

On 31 December 2012 at 13:10 Eelco Visser tagged feature
On 31 December 2012 at 15:34 Eelco Visser tagged 2.00
On 1 January 2013 at 17:27 Eelco Visser removed tag 2.00
On 1 January 2013 at 17:27 Eelco Visser tagged 1.60
On 5 January 2013 at 12:40 Eelco Visser tagged @eelcovisser

Added rudimentary Q&A feature with list of questions per project. Only logged in users can ask and answer questions. More features need; will be added as separate issues.
— Submitted on 14 January 2013 at 22:28 by Eelco Visser

On 14 January 2013 at 22:28 Eelco Visser closed this issue.
On 15 January 2013 at 10:49 Eelco Visser tagged q-a

New Comment

Your comment

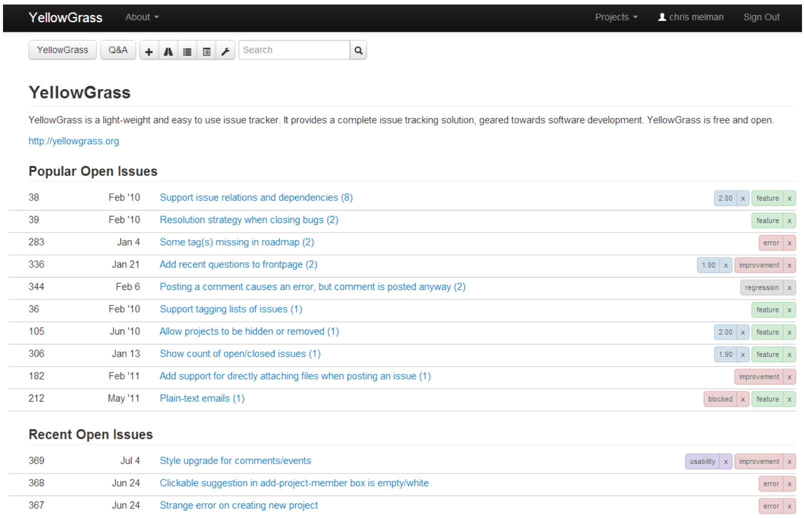
Figure 5.4: A YellowGrass Issue page

Roadmap Page

The roadmap is a feature of the application to keep track of changes related to version of the application. The roadmap page as displayed in Figure 5.5 is mainly to show the status and issues grouped by version description.

Mobile Scenarios

The described pages above show the important information of the application for users. The information from the previous paragraphs can be used to predict various scenarios that should be available in a mobile application of YellowGrass. Most of the pages are focussed on accessing information of the projects. The main information is currently stored in issues. This would mean that the mobile application should allow easy accessibility of issues with the related information like comments and tags. This includes the modification and creation of issues because that is probably the main feature what people will use the mobile application for. Additionally, the application should have some view to display information of the project including the roadmap.



YellowGrass

YellowGrass is a light-weight and easy to use issue tracker. It provides a complete issue tracking solution, geared towards software development. YellowGrass is free and open.

http://yellowgrass.org

Popular Open Issues

38	Feb '10	Support issue relations and dependencies (8)	2.00 x feature x
39	Feb '10	Resolution strategy when closing bugs (2)	feature x
283	Jan 4	Some tag(s) missing in roadmap (2)	error x
336	Jan 21	Add recent questions to frontpage (2)	1.90 x improvement x
344	Feb 6	Posting a comment causes an error, but comment is posted anyway (2)	regression x
36	Feb '10	Support tagging lists of issues (1)	feature x
105	Jun '10	Allow projects to be hidden or removed (1)	2.00 x feature x
306	Jan 13	Show count of open/closed issues (1)	1.90 x feature x
182	Feb '11	Add support for directly attaching files when posting an issue (1)	improvement x
212	May '11	Plain-text emails (1)	blocked x feature x

Recent Open Issues

369	Jul 4	Style upgrade for comments/events	usability x improvement x
368	Jun 24	Clickable suggestion in add-project-member box is empty/white	error x
367	Jun 24	Strange error on creating new project	error x

Figure 5.5: YellowGrass Roadmap page

5.2.3 Data Model

The pages and the functionality are based on the underlaying data model. The model is explained in this section to get a full grasp on the application. Figure **??** displays the model of the YellowGrass application. The simplified version of the model can be explained as a project with issues and members, where the issues have tags and a log of events for the specific issue. The next paragraph will explain each of the classes in more detail.

Project Central entity that represents the (software) projects which holds issues and users. Users can be split into members, who are working on the project, and followers, users that are interested in a project and are notified on updates. There is an additional private flag for projects that are closed to nonmembers.

Issue Representation of the issues and changes on a project. An issue has in addition status, author, events and tag information stored to document process of the issue.

IssueGhost A copy of issue to allow non registered users to create issues. The copy is used to store information until the issue is confirmed and transformed into a normal issue.

Tag Tags are used to group issues and it can contain other tags to operate as grouping on the level of tags.

User User entity is a representation of registered actors within the system. The user is mainly used to refer to as owner of objects. The entity stores personal information and a string representation of the user that is used as a tag name for special tags, which refers to users.

Event It is a generalization of the events that can happen within issues and uses a timestamp to track occurrences, which is used to sort the events.

Comment Comment is a proxy for the most occurring event of placing comments on an issue, which adds additional textual information about the issue and the process of implementation.

TagRemoval & TagAddition Events that are applied to the tags property of an issue and correspond to their names.

IssueClosed & IssueReopen Events that display the status changes an issue went through.

IssueMoved Special event that represents a move of an issue to another project and includes a link to the new issue.

This completes the list of the entities in the YellowGrass model and description of the YellowGrass application. The following sections will describe and evaluate the extension of the application, starting with the architecture design.

5.3 Architecture Design

The implementation of YellowGrass Mobl is meant as an experiment on feasibility of the synchronization between WebDSL and Mobl applications and will be used to deduce abstraction for further development.

This section explains the architecture design of the motivating example. The following viewpoints are used to give an overall impression of the functioning of the application: Context (Section 5.3.1), Decomposition (Section 5.3.2) and Control Flow (Section 5.3.3).

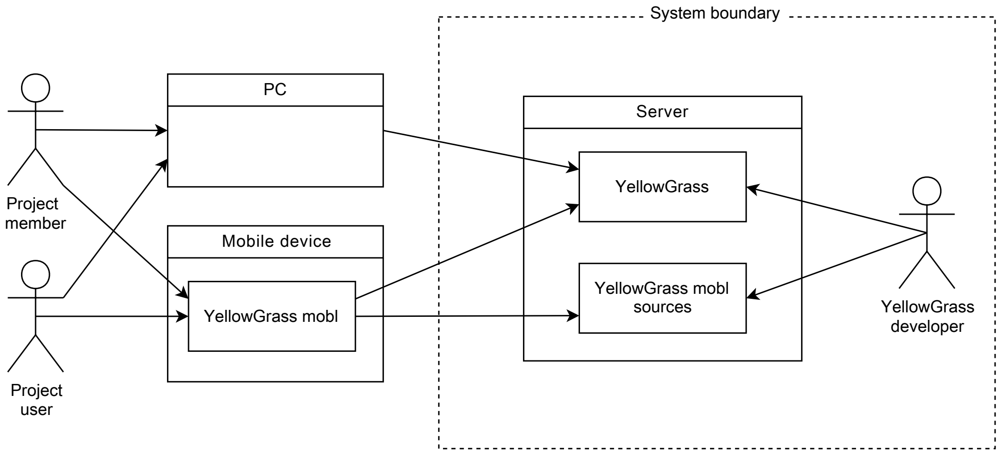
5.3.1 Context

YellowGrass Mobl is an addition to the original application.Figure 5.6 displays how the application and the Mobl additions are placed into the context. The boundary of the system restricted to the server with the YellowGrass applications, since other entities cannot be controlled. The role of the entities in the ecosystem are explained below.

YellowGrass The original web application has its original functionality and is extended with the purpose of central communication point for synchronization. The Mobl applications are able to request data and send changes that then will be processed by the web application.

YellowGrass Mobl The Mobl application implements part of the GUI of YellowGrass in a mobile variant. To display information it needs data, this is gathered through synchronization with the web application and persist it locally. The application also tracks changes to send them on the next synchronization attempt with the server.

YellowGrass Mobl sources The actual application runs on the mobile devices. Nevertheless, the application sources are served by the server.



Arrows display usage relations

Figure 5.6: Context diagram of the motivating example

PC The PC is a medium for accessing the web application and delivers a universal display through the medium of web browsers.

Mobile device The mobile device loads the YellowGrass resources and executes the Mobl application locally.

Server A central point that services as remote access point for both the web and the mobile application to request functionality or resources from the WebDSL application.

YellowGrass developer The person responsible for maintaining and developing both web and mobile application of YellowGrass. The developer is not a direct actor in the system. Although he has to deal with the code that is written for this motivating example.

Project member The project member is part of the user group that develops the project that uses YellowGrass to document issues. The member can use both the web and mobile application to modify the status and general information of the project on YellowGrass.

Project user The project user is a person that uses the application of a project, which is tracked on YellowGrass. The actor uses both applications to report and look up issues.

This concludes the context of the motivating example and will continue with the decomposition in the next section.

5.3.2 Decomposition

The motivating example consists of an extension to original web application and a new mobile application. The Mobl code can be separated into two parts depending if it is interesting regarding synchronization framework, the remaining code will be put into another module. An overview of the separations and components are shown in Figure 5.7. This section will describe the functionality and responsibilities for each of the components.

Webservices

The synchronization requires a communication layer between the web application and the mobile application. In the case of web applications it is common to use webservices for this purpose.

WebDSL
The main functionality of the synchronization is implemented by the webservices component. Functionality of the webservices is to facilitate entry points for mobile applications, which enables the possibility to request or send data from the application. The component started simply with just map the requested objects to JSON and send those objects to the mobile application. However, the expansion of the motivating example, in functionality of synchronization, added extra responsibilities for this component. The final version has extended the responsibilities with the following synchronization concepts: apply and find updates, and also detect and resolve inconsistencies.

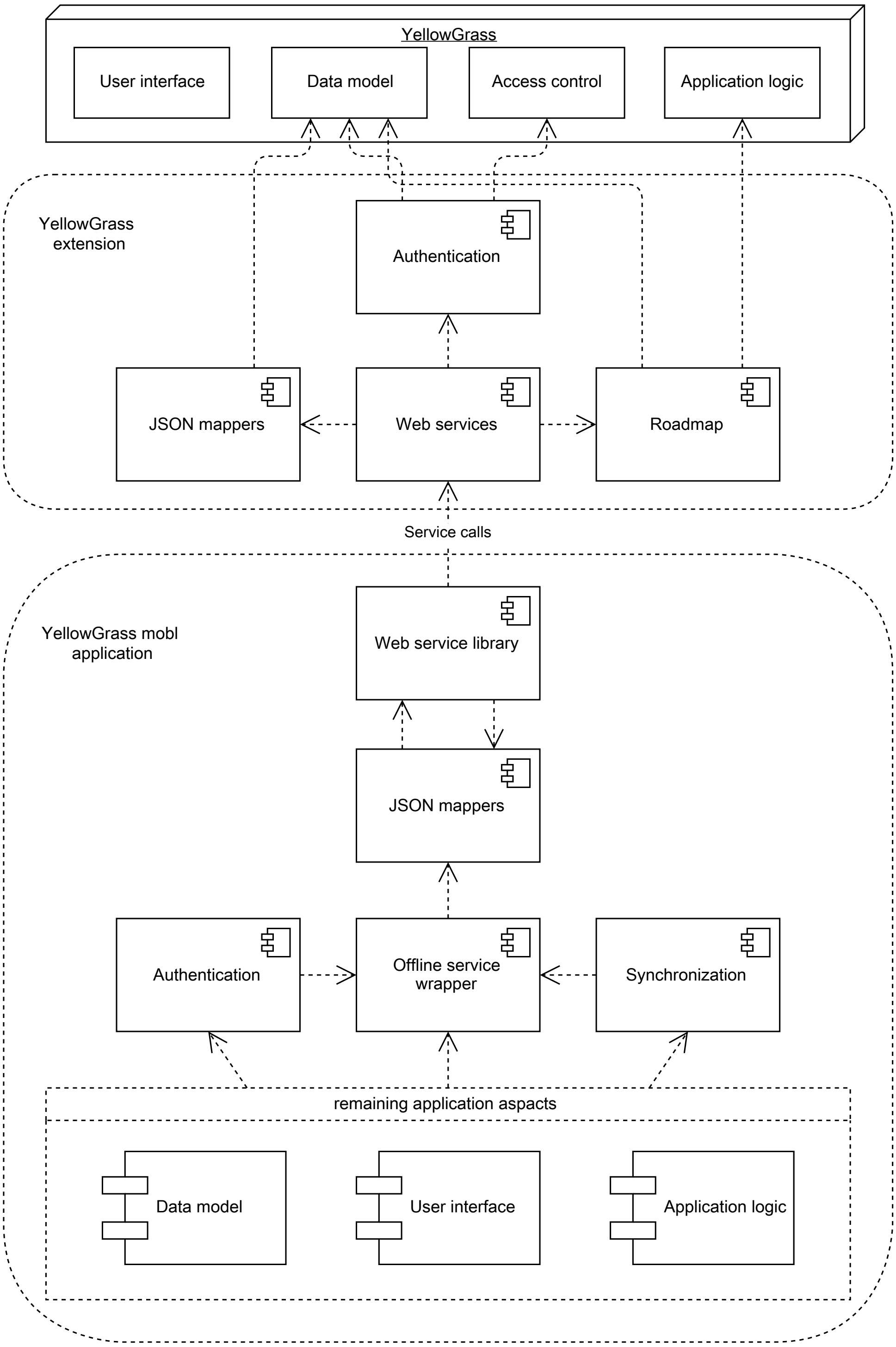
Mobl
The webservice library offers entry points for Mobl code to call the webservices that are specified in the webservices component of WebDSL. In addition, to making the services available in Mobl, this component also maps the parameters to a correct format that is expected by WebDSL services. The result that is handled by a mapper is specified in this component, but declared in the JSON mappers component.

Offline Service Wrapper

The offline service wrapper takes care of the possibility that a mobile device has no internet connection. Every service that is called is wrapped with a check whether there is an internet connection or not. The idea is that the function call does not have to take care of connection status. The specified actions differ for each function, but in general when there is no connection the function tries to return local data. When the functionality depends on up-to-date data, it will return an error since there is no valid return value.

JSON Mappers

The webservices are used to send information in the form of objects. However, the information stored on the devices is in a native format. This format is hard to interpret for the other application. This means that both the web and mobile application require mappings from and to a serializable format. In this case we have chosen for the lightweight JSON representation.



Arrows display dependency relations

Figure 5.7: Decomposition diagram of the motivating example

WebDSL
The translation is based on the data model from the main application and uses a default mapping from WebDSL types to known types in JSON. Each entity has three kinds of mappers:

- **Minimal**: Returning a representation which just contains information needed for references in other objects.
- **Simple**: Only containing the properties which are basic types of WebDSL, so without references to other objects.
- **Full**: A full representation of the object for properties that are mappable to JSON.

The mappers include for each entity a function that applies the values of a JSON object to the local storage.

Mobl
The mapping of Mobl entities to JSON is delivered by libraries. Those require a parameter which defines the properties to be included in the JSON representation of the objects. This component also declares calls and parameters for the mapping. The processing of the JSON objects in the result depends on the functionality of the webservice. In general, the mapper extract useful information from the JSON. This information is then persisted or returned to the offline wrapper component.

Synchronization

Synchronization is the core component to make the application working because the general functionality of the application is based on data gathered through the synchronization. While the core of synchronization is a part of the webservices in WebDSL the Mobl uses a separated module. The module manages integration of webservice calls by ordering and checking if all data is up to date. The webservices require specific input to make the synchronization algorithm work. This component gathers the required data from the database for each of the synchronization functions.

Authentication

Authentication is not directly coupled to the synchronization problem itself. However, YellowGrass contains data that should not be shared with all users. This requires a system for access control, which is only possible if there is authentication.

The authentication component handles the functionality of authentication for mobile devices. There are several forms of authentication functionality that is needed for mobile devices:

- Registration and coupling of mobile device to the user.
- Authentication of mobile device with credentials.
- The possibility to undo the authentication or registration.

WebDSL
The WebDSL component contains functions that extend the application with additional functionality to allow this different approach of authentication for mobile devices. It extends the model with a devicekey entity which is linked to the user entity.

Mobl
The Mobl components has similar functionality to enable the user to authenticate through the webservices. In addition, it stores information that is required for authentication, so the application can log in automatically. For security reasons it has the responsibility to delete local data when a user is logged out to ensure that sensitive data is not accessible by other users.

Roadmap

The roadmap is not directly expressed in the model. Instead a relatively expensive operation is used to calculate the roadmap. This module is created to make this information available without expensive calculations in the mobile application.

The roadmap component is responsible for the alternative of the calculation of the roadmap for the mobile application. This modification is an experiment to offload the calculation of roadmap, since in the original implementation it is a heavy computation for mobile devices. The component calculates the roadmap and maps the result to a JSON format which is usable for the Mobl application to represent the roadmap.

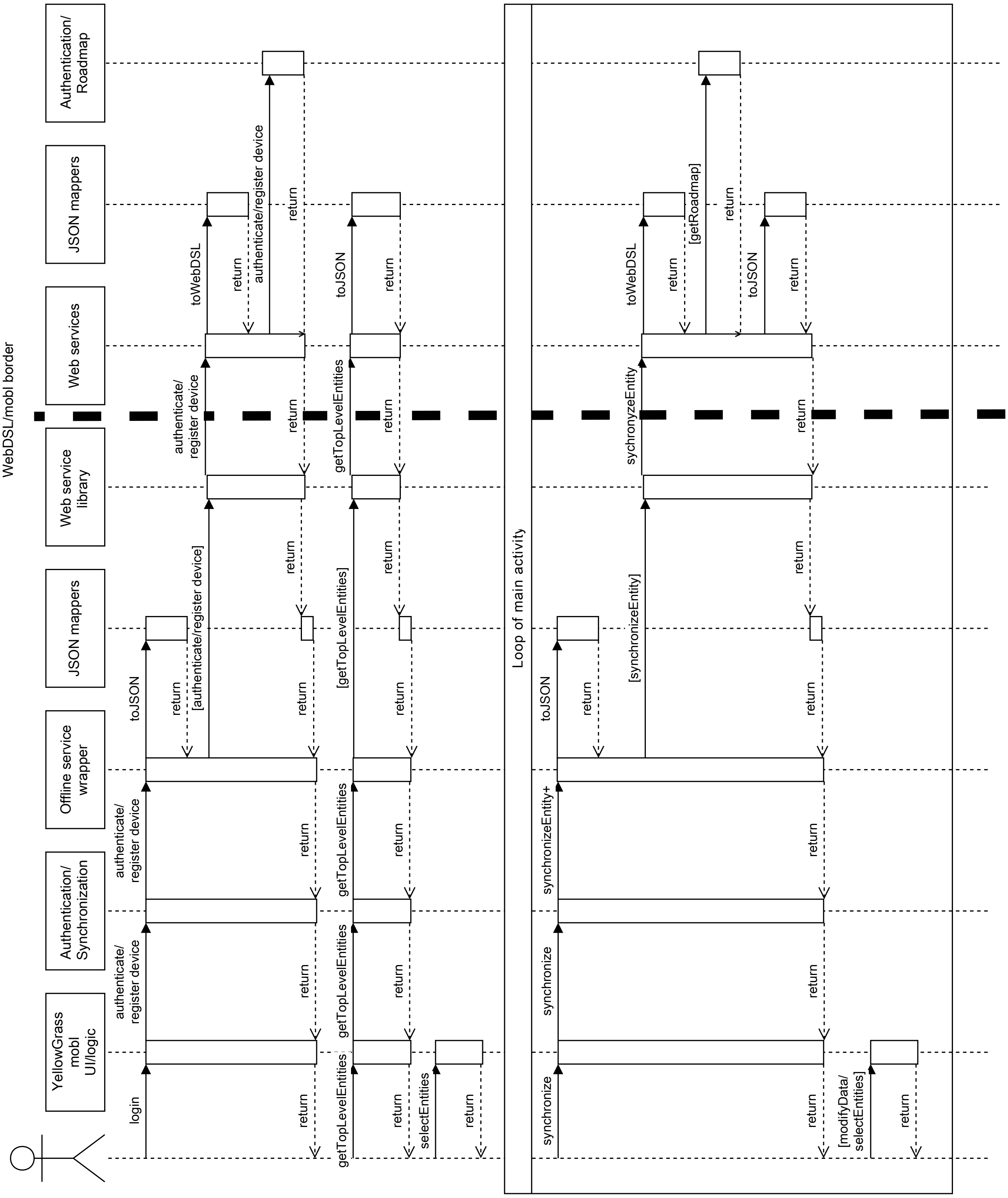
Remaining Mobl Modules

The Mobl application involved also other components in the development because it was built from the scratch it needed other functionality that were non specific to the synchronization. Those components where less interesting for the synchronization process, but have shown integration problems and shortcomings of the synchronization approach on time of development. The remaining part can be described as a simplified implementation of the original YellowGrass application in Mobl, which depends on the local data which is gathered through the synchronization, authentication and other webservices. The remaining segment contains the data model, user interface and application logic. The data model is most interesting of those modules, since most of the synchronization components are based on the model.

The total of those modules covers the complete functionality of the motivating example. More detail of the functionality will be explained in Section 5.4.

5.3.3 Control Flow

The previous section has shown the components and the dependencies, but misses the actual order and flow between the components. This section presents the workflow of the application as displayed in Figure 5.8.



[] = optional, / = or , + = multiple times

Figure 5.8: Sequence diagram of the motivating example

General Flow

Before discussing the various steps within the application, it is good to have an overview of the general flow in the application. The first step is to log in to the application with the mobile device. Following with getting the topLevel entities and select useful entities that represent partitions that the user want to have synchronized. Synchronize the entities to have the latest version locally. As last step, interact with the application which possibly effects the data. The steps of synchronization and interaction will be repeated to keep the data up-to-date. This flow is an abstraction and allows variations. Nevertheless, those deviations will not introduce new situations that are not covered by the previous steps.

Detailed Flow

This section will give more details over the steps described in the previous section. The following main steps that can be distinguished are: authentication of the user, selecting of partitions and synchronization.

Authentication
As described in the component description is authentication split up into two parts: Registration of device and authentication of user and device for current session. Depending on the current information stored on the mobile device it can directly authenticate, otherwise it has to request registration. The request goes through the offline wrapper, then mapped to JSON, which will be sent to the server. There it is mapped back to values of WebDSL, so it can be passed to the authentication component, which returns a result with possible registration key. The result goes in similar way back to the authentication component of Mobl.

Select Partitions
Synchronization requires a selection of entities, this is only possible if there is a minimal representation of objects making it possible to recognize and select partitions of data. In YellowGrass Mobl this is fulfilled by requesting simplified versions of selectable entities. This is done by simple request via a webservices, which maps the entities to a simplified JSON representation and sends them back. Afterwards, the top entities can be selected for the synchronization process.

Synchronization
The last step is the biggest and most occurring process element. It represents the synchronization process of the data, in the first request it will only request data. The synchronization has been split up into multiple blocks of objects. For each block it finds the changed objects and maps them to JSON representation. In WebDSL it is mapped to local representations and afterward applied to the database. In the second step it finds the updates (where needed uses the Roadmap component) and maps those objects to JSON so that they can be returned to Mobl. At the other side the mappers persists the changes to the local database so that it can be accessed and modified before the next round of synchronization.

The control flow concludes the architecture design of the motivating example. The next section will describe the more detailed implementation work.

5.4 Implementation Details

The previous section describes the architecture of the motivating example and with that has given an impression how the application works. This section gives deeper insight in some parts of the application. The more interesting parts of the application for synchronization can be found in: offline functionality (Section 5.4.1), synchronization algorithm (Section 5.4.2) and the model-to-model mapping (Section 5.4.3).

5.4.1 Offline functionality

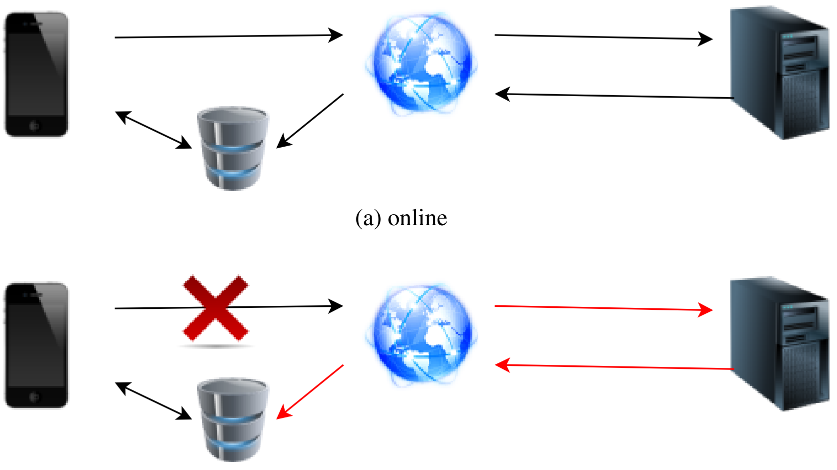
Most mobile applications are (partly) usable without internet connection. For web based applications like YellowGrass Mobl it is harder to be available without connection because of the remote resources. The same counts for data based applications, since those applications also require remotely stored data. This should be fixed by the synchronization solution itself.

Mobl is a client side based solution, therefore, it is possible to have the application working without internet when the sources are locally available. The current HTML5 specifications specify a solution for this problem in the form of an offline manifest, where it is possible to specify the behavior of a web application when there is no internet access available [37].

The second problem is the unavailability of webservices due to the fact that a mobile device can be offline. This requires a solution where the data is stored locally and tracking of changes. There are also functions that can only be executed when there is a connection, for example authentication. The first approach for offline webservices is based on request of data for specific functionality on the application and when there was no connection the local data was returned instead of the data from WebDSL. This method requires much knowledge about the functionality of each service to return correct local data. To improve the situation the default approach changed to work on local data and use synchronization to make the local storage up-to-date.
The functions that require a connection for execution started with authentication and roadmap functionality. The first attempt was to store the webservice calls until the connection was restored. There are some pitfalls in this concept, firstly discovery of the connection status. In theory it is simple to determine internet connectivity. However, several approaches that are used in practice where all problematic. Incorrect (in corner cases) or slow response are the general disadvantages even for the HTML5 proposed function for this purpose.

The other problem is how to let users know when the function is executed and how to show the result that is returned. Normally the function would be executed when the connection is restored. Although this moment is not always clear to the user. Possible interaction before or after the execution might be a valid solution. However, for the user this would make the workflow unpredictable and perhaps even incomprehensible.

In the last implementation most of this functionality was removed and only the authentication services required a connection. When there is no connection the authentication will use the last know state to keep the application usable. It tries to simulate the same behavior as with the synchronization that is accessible through local storage and it can be updated when there is a connection. This flow of data within the motivating example with off- and online state is displayed in Figure 5.9.



(a) online

(b) offline

red arrows indicate removed links due to missing previous link

Figure 5.9: Graphical display of connections between server and application in off- and online mode

5.4.2 Synchronization

Synchronization is main functionality and added value of the motivating example. Therefore, it is interesting to have more insight in the working of the synchronization. This section gives more details of the synchronization algorithm. It is split up in how the activities of synchronization are implemented and how the data is partitioned to make the data selectable.

Activities

In Chapter 3 describes that synchronization is split up in the three activities: detection of updates, propagation of updates, and detection and resolution of inconsistencies. The following paragraphs will explain how those activities are filled in within this example.

Identification of updates
The Mobl application only synchronizes with one source (the WebDSL application). This makes flag synchronization the best solution. Keeping flags for every object has a minimal amount of overhead and is simple to apply. The flags are required to manually set or unset. Nevertheless, it should be possible to set this automatically with modification of the Mobl compiler.

The identification of updates on the server side is more complicated because of the multiple clients it has to deal with. Versioning of objects is an obvious solution, since the version is already tracked by WebDSL. The problem is that for identifying the updates it requires the version of each object delivered by the previous synchronization, which adds a parameter for the synchronization webservices. Figure 5.10 displays the pseudocode of both parts.

```
function synchronizeFindUpdates(objects)
  updates :=  new Set()
  foreach object in objects where object.modified or object.created
    updates.add(object)
  return updates
```

(a) Mobl

```
function synchronizeFindUpdates(previousObjects)
  updates :=  new Set()
  foreach object in previousObjects
  currentObject := loadObject(object.id)
    if object.version < currentObject.version
      updates.add(currentObject)
  return updates
```

(b) WebDSL

Figure 5.10: Pseudocode for identification of updates used in the motivating example

Propagation of updates

The next step is the propagation of updates. This is a simple step, in case of Mobl application it sends an object with id and version for every object it needs to synchronize. If an object is modified or created it adds all other fields to the JSON representation of the object. WebDSL receives all object representations, depending on the version number and availability of other fields it decides what to do. In the case of extra fields available in addition to the identifier and version, the object is recognized that it is new or modified. If the version is 0 and identifier is unknown in the database it creates a new object and applies the modifications. For objects that have versions that are higher (this means that the object is not changed on the server) and have a known identifier, the server will load the object and apply the changes from the JSON object.

The last step is to find the changed objects, which is implemented by comparing the version numbers of the JSON object and the local object. When the version number of the local object is higher than the received version, the object is perceived as changed after the previous synchronization. Therefore, it needs to send a new version to Mobl where they are applied to the persisted data. Both algorithms are displayed in pseudocode in Figure 5.11.

Detection and resolution of inconsistencies of data

The final step of synchronization is the detection and resolution of inconsistencies. This step is simplified for the motivating example and is based on an underlying framework in WebDSL. WebDSL allows to specify validation rules on the data model. The code checks all modified objects if they satisfy to those validation rules. The database transaction is canceled when an update infringes one of the validation rules on the model, which means that the change is not persisted. As an additional check the version of the object is compared with the version on the server to prevent updating of objects that are already updated on the server. The validation and rollback is done in the background, which is expressed in the pseudocode. The other change is given by Figure 5.13, which is like a slight modification of the algorithm of Figure 5.11.

```
function synchronizePropagateUpdates(objects)
  JSON := new JSONArray()
  modified :=  synchronizeFindUpdates(objects)
  foreach object in objects
    if object in modified
      JSON.add(object.toJSON())
    else
      ref := new JSONObject(id=object.id, version=object.version)
      JSON.add(ref)
  updatesFromServer := sendAndRecieveUpdates()
  foreach JSONObject in updatesFromServer
    persistJSON(JSONObject)
```

(a) Mobl

```
function synchronizePropagateUpdates(JSONArray)
  foreach JSONObject in JSONArray
    if JSONObject.numberofproperties > 2
      if JSONObject.version == 0 and unusedId(JSONObject.id)
        createObject(JSONObject.id)
      localObject := loadObject(JSONObject.id)
      localObject.applyChanges(JSONObject)
  updates := new JSONArray()
  modifiedObjects := synchronizeFindUpdates(JSONArray)
  foreach object in modifiedObjects
    updates.add(object.toJSON())
  return updates
```

(b) WebDSL

Figure 5.11: Pseudocode for propagation of updates in the motivating example

```
function synchronizePropagateUpdatesAndPreventInconsitencies(JSONArray)
  foreach JSONObject in JSONArray
    if JSONObject.numberofproperties > 2
      if JSONObject.version == 0 and unusedId(JSONObject.id)
        createObject(JSONObject.id)
      localObject := loadObject(JSONObject.id)
      if JSONObject.version == 0 or
      JSONObject.version == localObject.version
        localObject.applyChanges(JSONObject)
  updates := new JSONArray()
  modifiedObjects := synchronizeFindUpdates(JSONArray)
  foreach object in modifiedObjects
    updates.add(object.toJSON())
  return updates
```

Figure 5.12: Pseudocode for preventing of inconsistencies of updates in the motivating example

Partitioning

One of the requirements is to only synchronize a selectable part of the data. To prevent that it is needed to select every object, partitioning is required. The approach used in the motivating example is done by viewing the data model as a graph, where property types are seen as outgoing edges. Data can be separated on relations with an object of the type TopLevelEntity, by selecting a specific entity to be the root (topLevelEntity). A possible problem is that there could occur cycles and even one object could relate to all other objects and break the partitioning solution. Links to objects of the type TopLevelEntity are removed to solve that problem.

The algorithm uses the status of local objects in Mobl to find update by finding objects that are incomplete, only having id and version, and request those objects from the server. Following the order of the graph makes this approach more efficient. To prevent more cycles the graph is transformed to a tree by deleting edges from node x to node y that are not part of the shortest path from the root to node y.

```
function model2tree(entities,topEntity)
  graph := model2graph(entities)
  foreach node in graph.nodes
    node.shortestPath := shortestPath(graph, topEntity.name, node.name)
  foreach edge in graph.edges
    if not edge.to.shortestPath.contains(edge)
      graph.remove(edge)
  return graph

function model2graph(entities)
  nodes := new Set()
  foreach entity in entities
    nodes.add(entityToNode(entity))
  foreach entity in entities
    from := nodes.get(entity.name)
    foreach property in entity
      to := nodes.get(property.type)
      from.add(to)
  return new Graph(nodes)
```

Figure 5.13: Pseudocode for transforming data model to graph tree representation

For YellowGrass, Project is the most obvious choice for selecting as the topLevelEntity, almost all data is coupled to the projects and the number of entities is within a normal range to use as selection mechanism. Also, from the view of the domain it is intuitive to select Project for coarse grained division of data. Applying the model2tree algorithm on the class diagram given in Figure **??** delivers the tree presented in Figure 5.14. A breadth-first approach is applied to decide the order of synchronization on entities.

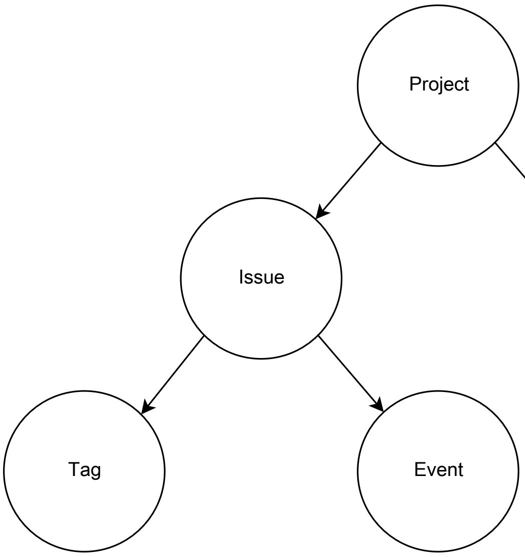


Figure 5.14: Tree representation of the YellowGrass model after applying the model2tree algorithm

5.4.3 Model-to-Model

The data model is very important for the data synchronization and would be easiest if they would be equal. This is not possible, since Mobl does not have the same expressiveness power for data models as WebDSL. The mapping was a manual process with in-between modifications when encountering problems. This section describes encountered problems and patterns that are found in the creation of the data model.

The approach is to keep as close as possible to the model of WebDSL, making a simple copy of the data model is a good start for the creation. Afterwards, simplify the entities by removing unwanted properties and entities that are not needed for the functioning of the mobile application. For all properties find a type in Mobl that is similar or at least can express the same data as the WebDSL type.

With this approach we found two nontrivial issues: List and Set, and inheritance hierarchy. For list and set Mobl has one possible replacement variant and that is the Collection type, which indeed can express both types. However, when this type is used it needs an inverse relation with another property, which is not always defined by the original model. Fake properties where added for the mapping of collections. Extending of entities is not supported in Mobl and needs a manual solution, in this case the choice was to just use Comment for replacement of the Event type as simplification. This solution is only possible because other entities where not used within the YellowGrass Mobl application.

In the decomposition viewpoint (Section 5.3.2) the Roadmap component is described as a modification to remove the roadmap calculation from the mobile application. To have the roadmap offline available it needs a model to persist the data, implemented by an additional Release entity. This entity is not available within WebDSL and cannot deal with modifications.

This completes the implementation details of the YellowGrass Mobl application. The next part of this chapter is the evaluation of the application.

5.5 Evaluation

The motivating example has as goal to discover problems that are encountered when solving synchronization between web and mobile applications and having a baseline for the final solution with a generative approach. This example is mostly tested on integration level by using the application and observe the state of the data on server and client side. The shortcomings encountered are discussed in Section 5.5.1. In addition, Section 5.5.2 describes the process and results of data chunking for synchronization.

5.5.1 Shortcomings

During the development and use of the motivating example certain shortcomings where exposed. Some of the flaws are fixed within the development and the previous sections described the interesting changes. The second category are problems that have a bigger impact or are more particular to the generative approach. Those problems will be discussed in this section and are listed below.

- The solution for finding updates on the server side delivers a considerable amount of overhead for the mobile client.
- The approach for data partitioning is incomplete.
- An additional component for heavy calculation like the Roadmap module is not an optimal solution for this type of problems.
- Lacking of robustness and security aspects.
- The choice for selecting Comment as representation for all sub types is not satisfying for all purposes.

Update Identification
The identification of updates on the server side is done by comparing the version numbers for each object. This requires to keep old version numbers for each device or the device has to deliver the version numbers, like implemented in the motivating example. The tracking of version numbers in the mobile application increases the overhead for the mobile device. The mobile application has to query and transform all the objects to JSON, which is relatively cheap operation, but becomes expensive through the huge amount of objects. Next to computation it also delivers extra bandwidth, with a small change set, the overhead could even be larger than the updates send and received.

Data Partitioning
The model2tree approach presented in Section 5.4.2 is used to separate and define the order of the synchronization. This intuitive approach to the problem was not correct because sometimes new objects where not retrieved on synchronization. A possible fix could be checking for objects that are incomplete and synchronize those objects until the number of object that are not synchronized is reduced to zero. This implementation would guarantee the completeness, but requires massive querying for checking the status of the objects. The solution also weakens the purpose of the tree, since it can be

executed in any order, still using the correct order will in most cases reduce the number of cycles needed to synchronize al entities.

The choice in YellowGrass to select Project as top level entity is obvious and the price of not being able to use GhostIssue is not problematic, since it is not used in mobile application. Removing some of the entities could become a bigger problem. For example, in the case that the model is more of a collection of graphs instead of one fully connected graph it is impossible to select only one root. To make all data available it is needed to have multiple top level entities and causes that the tree solution is not applicable anymore.

Heavy Computations

The Roadmap component was an experiment to delegate heavy computation from the mobile device to the server. The approach that is implemented is to calculate on the server and simulate an addition to the data model for the mobile application. This component shows that in principle it is possible to implement, but it is hard to totally simulate the behavior of normal entities. In the case of this application it shows a weakness of the model and in general could conclude that adjustment of the model or partitioning of the calculation is preferred.

Robustness and Security

At development of the motivating example some of the security and robustness issues are neglected, since it was outside the scope for this application. For security a few easy checks were added to not retrieve private projects and its data. Wrong input was not possible since it was restricted to the Mobl application developed during this process, which means that robustness problems are limited. The checking and resolving of inconsistencies in the updates where enough to have robust solution for this application. For further development and usage both should be lifted to a higher level.

Inheritance Hierarchy

The solution used for inheritance hierarchy within the data model is to select one entity within the sub types. While for YellowGrass this was a valid choice without losing too much of the data. This approach will not work for most of the situations where multiple of the subtypes are needed by the mobile application. A general approach to separate or combine the subtypes is probably a better solution.

This list of shortcomings were discovered during implementation and testing. The next section describes an experiment for further evaluation of the motivating example.

5.5.2 Webservice Data Chunking

One of the approaches tested to improve the scalability of the data synchronization, in the motivating example, is data chunking of the webservices. The bigger amounts of data applied at once to the mobile storage could be constraining. To find out what the influence is of chunking a test with splitting the objects by sending smaller packages. This section describes the test situation and the results of this experiment.

Experiment setup

The experiment is set up to discover whether data chunking influences the time of the synchronization. The best situation would be something similar to normal usage with one server and a mobile device. Nevertheless, simplification would not change the outcome as long as the results are produced by the same enviroment. The experiment is executed on one PC (quad-core 1.6 GHz and 4Gb ram), which contains the WebDSL and one Mobl instance. The Mobl application is based on HTML5 technologies and is therefore accessible through web browsers. In the experiment the Chrome browser is used, since it supported all HTML5 functionality that Mobl requires and the inspection tools gave possibility to inspect the process.

The experiment exists of two goals, the first is to find an optimal size of chunks to a medium sized data set of YellowGrass which contains 229 issues and the updates transferred are 224 kB of size. The second part is to take different sizes of data sets and find out if the effect of chunking holds on the whole range.

Results

The first experiment starts with some basic values for size and added some extra data points between 10 and 25 objects in a chunk to increase the precision in the range of the optimum. The results are displayed in Figure 5.15 and shows an optimal size around 20 objects in one chunk. Also, it is shown that small sizes for chunks make the effect negative while on bigger chunks the influence of chunking disappears.

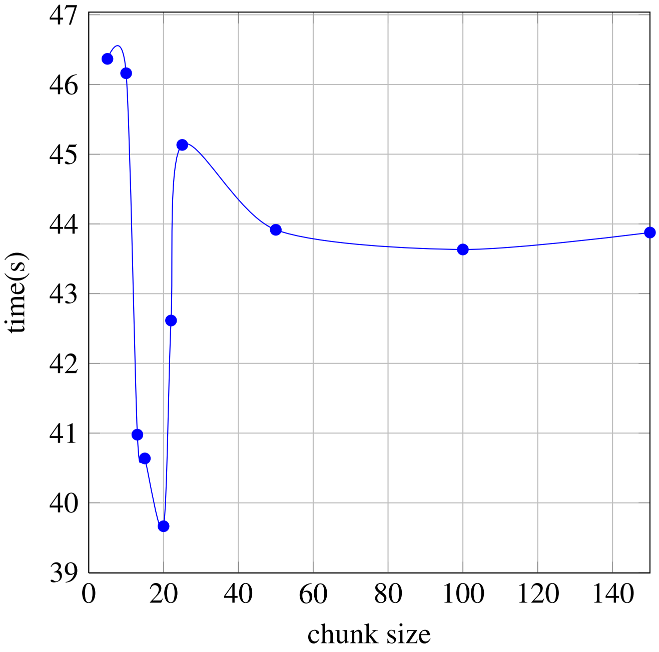


Figure 5.15: Results of chunking data of webservices with constant data set and varying chunk size

Using the optimal size found in the previous experiment, executed the next experiment on different data sets varying in size, expressed in number of issues and size of updates. Figure 5.16 shows a small tough increasing effect of the chunking. The increase of size of updates supports the positive effect and shows that number of objects seems to be more of concern for the algorithm then the size of the objects.

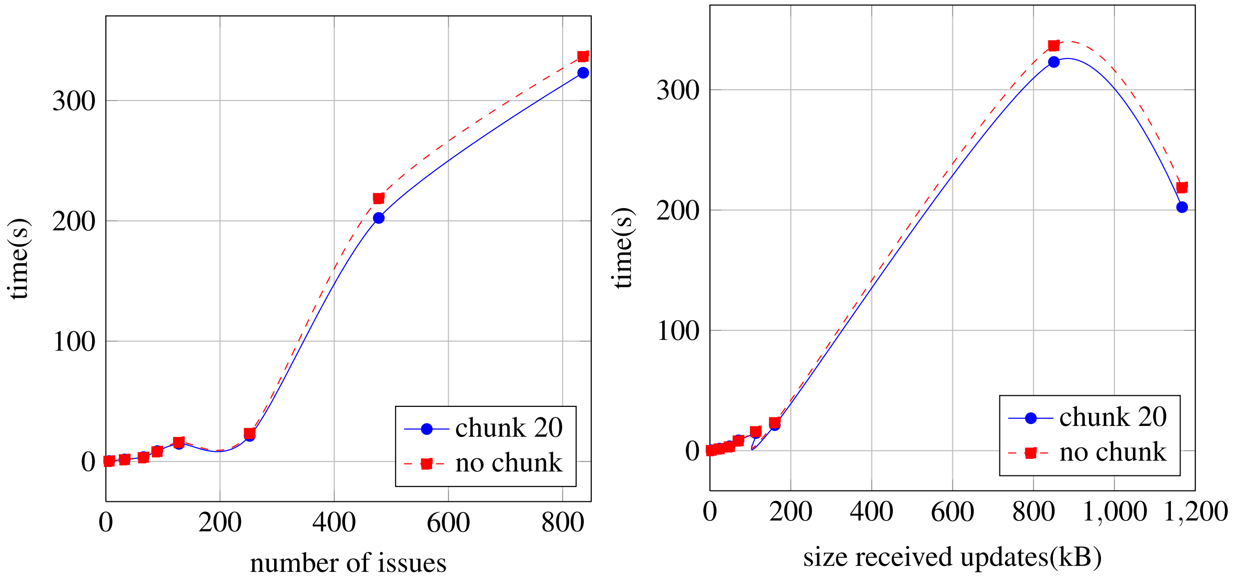


Figure 5.16: Results of comparing chunking and non chunking with varying size of data set

Discussion

For interpretation of the result we have to take into account the big variances in the data gathered from the experiment. The browsers seems to have problems with delivering constant values, but the average and the trimmed average of show a similar trend. The conclusions from the experiment is that chunking has a positive effect on the time needed for synchronization. On the other hand, the improvement is only a small part of total time consumed.

More worrying is the scalability aspect, while considering some overhead in the synchronization algorithm the time was still a lot higher than expected. Further inspection with the profiler shows that 98% of the time was spent in native browser code. In the case of synchronizing objects can be reduced to the handling of webservices and the usage of the local database. The webservice interaction should be limited and simple so the database querying is the most likely cause. Logging of the queries showed a huge amount of queries for search purposes. Therefore, the experiment is repeated with all the search annotations removed form the model. To show the effect of persisting changes an additional test is added, which is executing a second synchronization that contains no updates.

Results continued

The first experiment showed a general improvement of more than 50% in time reduction. The pattern found in the previous experiment is contradicted by the results of this experiment, since the optimum was found in bigger chunk sizes and also reduced the improvement in time to minimum amount as displayed in Figure 5.17. This experiment indicates that most likely the chunking effect only has some value when the search annotations are added.

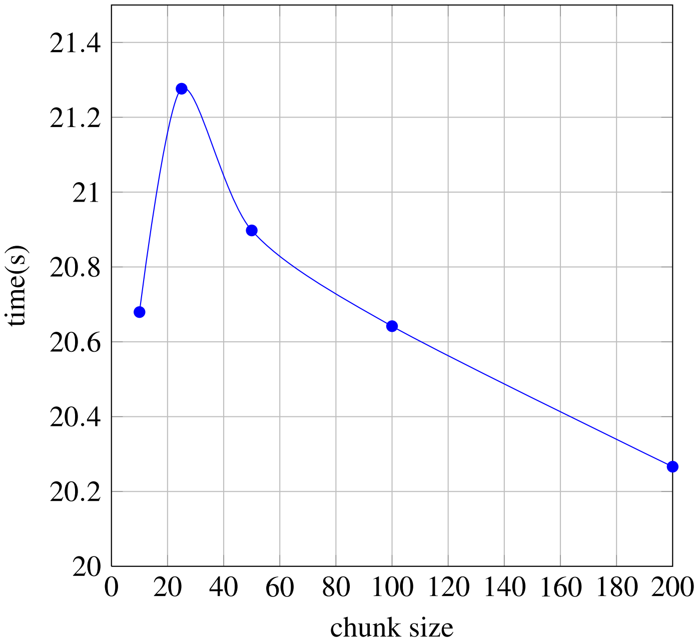


Figure 5.17: Results of chunking data of webservices with constant data set and varying chunk size (removed search annotations)

The results gathered by increasing the size of the data sets only supports the indication that the effect is minimal, since in general the chunked version seems to be slower or equal to the normal version as displayed in Figure 5.18. The more surprising part is that the reduction of time increases even more when data sets are bigger in size.
As stated before an additional experiment is added to find out what the overhead is in the synchronization algorithm. The results in Figure 5.19 shows that less than a second is required if no updates are encountered. The chunking effect is similar as in the previous results of this section.

Discussion continued

The new results show big improvements in the form of speedup and contradict the story of the effect of chunking. The second part where varying the size of data sets encountered similar results and even changing chunk of 20 to different values did not present any result to prefer chunking. Even with search enabled the improvement is small and comes with a price of additional complexity for the algorithm.

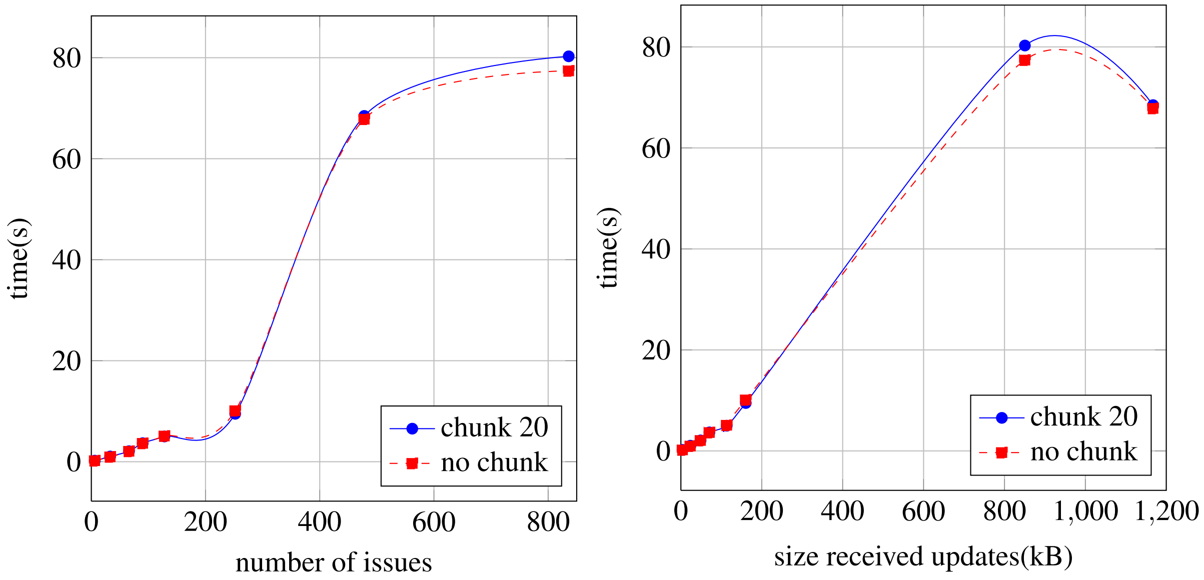


Figure 5.18: Results of comparing chunking and non chunking with varying size of data set (removed search annotations)

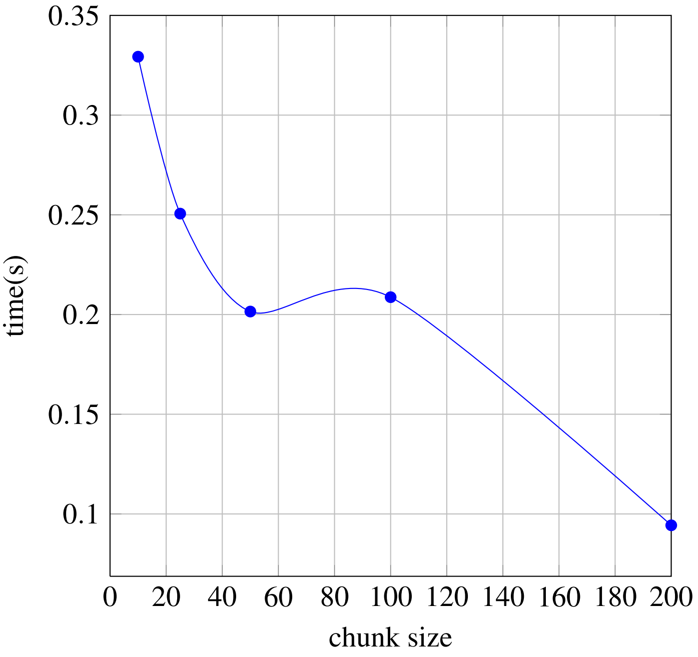


Figure 5.19: Results of chunking data of webservices (without updates) with constant data set and varying chunk size (removed search annotations)

The last plot displays that a small amount of resources is needed for synchronization without updates and therefore we can conclude that the core of the algorithm is just a small part of the total time spent for synchronization. The other time that is needed in the synchronization is in serialize and persisting of updates and cannot be reduced. The only way to limit the time is by reducing the size of updates.

In general the graphs do not show a signficant improvement when the data is chunked. The full synchronization requires a considerable amount of time, while the incremental updates prove to be more scalable. The most expensive part of the synchronization seems to be in the underlying persisting of updates in the mobile application. This is a restriction included with the choice of target language.

5.6 Summary

This chapter describes the process and the implementation of the motivating example for a synchronization framework between web and mobile application. It is based on YellowGrass, an existing web application for tracking issues. In the example a mobile variant is created, which is based on data delivered by synchronization. Discovery and solving of particular issues of synchronization is the goal, with consideration to reuse solutions for the generative approach.

The architecture is described in the following viewpoints: Context (Section 5.3.1), Decomposition (Section 5.3.2) and Control Flow (Section 5.3.3). The architecture describes the general working of the YellowGrass Mobl application and in the implementation details explain the interesting parts of the synchronization related code. Containing a detailed description of the following topics: offline functionality (Section 5.4.1), synchronization algorithm (Section 5.4.2) and the model to model mapping (Section 5.4.3).

Finally, this chapter evaluates the motivating example and has the following shortcoming that needs to be improved for the final solution:

- The solution for finding updates on the server side delivers a considerable amount overhead for the mobile client.
- The approach for data partitioning is incomplete.
- An additional component for heavy calculation like the Roadmap module is not optimal fix.
- Lacking of robustness and security aspects.
- The choice for selecting Comment as representation for all sub types is not satisfying for all purposes.

An experiment to test the effect of data chunking on webservices shows little too no improvement with chunked data. The tests show some other interesting facts: Indexing for search purposes in Mobl is really heavy on the synchronization process and the time consumption is mainly in the update propagation.

Chapter 6

Architecture Design

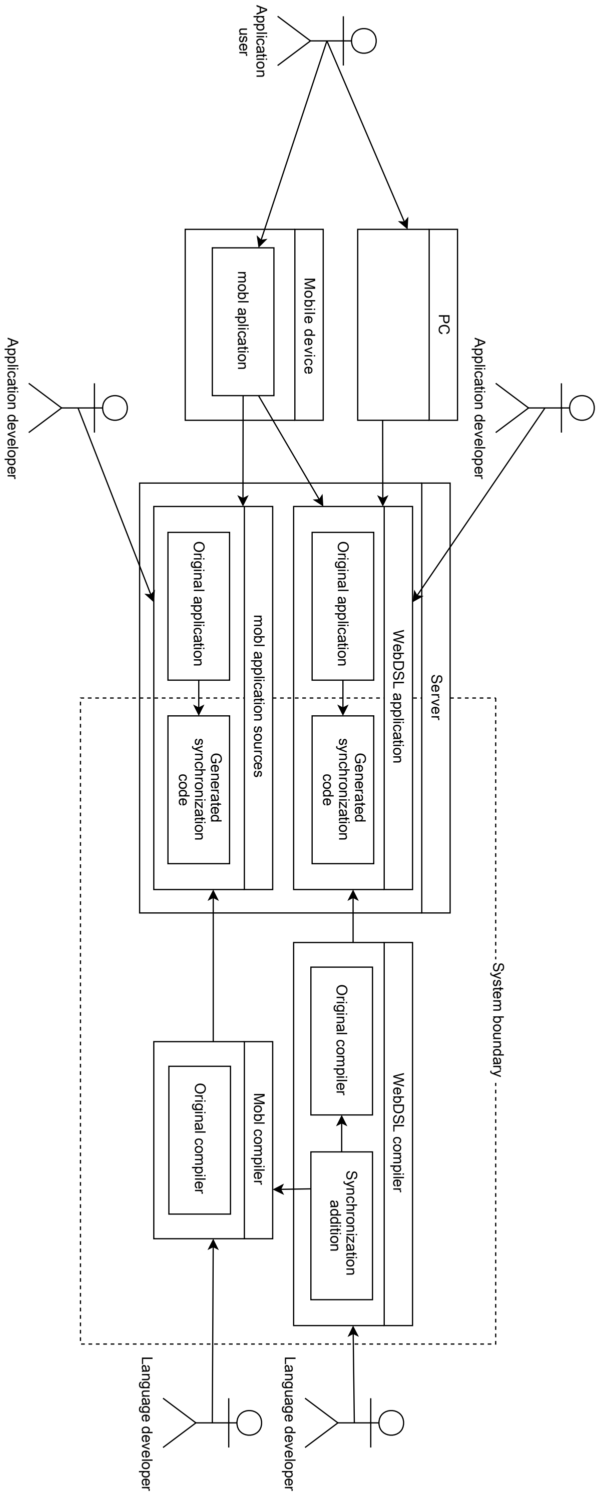
The final goal for this thesis is a solution for data synchronization between web and mobile applications, which requires a small amount of effort from the developer. In the previous chapter we introduced the motivating example. This example is a mobile application, which is a mobile application for the existing web application based on data synchronization. However, a generative solution requires a different approach, it has to find abstractions which are applicable to a wide range of web applications.

The total solution consists of an addition to the WebDSL compiler that generates WebDSL and Mobl code which forms a data synchronization framework. This chapter explains the architecture of the additional compiler components and generated code. Implementation details of the solution are discussed in Chapter 7.

To give a good overview of the architecture of the solution, multiple viewpoints are used. The viewpoints that are used in Section 5.3 will be extended to improve the understanding of the solution. The context, Section 6.1, of the solution is described as an overview of the system and it users to show how the solution fits into the picture. Section 6.2 breaks down the software in components to describe its functionality for each of the components. This is followed by a control flow, Section 6.3, to describe the usage and internal relations of the components. The data external to the system is explained in Section 6.4, which should give insight in the required information needed for the solution. The last viewpoint is security and robustness in Section 6.5. This viewpoint is added because they are important assets for web applications.

6.1 System Context

The context for this solution is similar to that of the motivating example presented in Section 5.3.1. The difference in generated code can be found in the abstraction on application level because instead of YellowGrass it applies to all sorts of WebDSL and Mobl applications. The generative approach adds components to the context namely, the compilers that are used for compilation of the applications. The system boundary has moved away from the application to the compiler part because there is no influence on the specific applications that can use the tool for data synchronization. The roles of the entities within the context are displayed in Figure 6.1 and explained in the following part.



Arrows display usage relations

Figure 6.1: Context diagram for generative solution

The direct users of the *compiler extension* are the *application developers*. Those developers take an *original WebDSL application* and use the compiler extension to generate code, the *synchronization framework*. The generated files are then used by the developer to include them in the *original WebDSL and Mobl application*. The compile result of the *combined code* for both applications are put on the *server* so they are ready to be used.

The *application user* can use both the web as the mobile version to access the application. The synchronization addition does not change anything for the user to the original *WebDSL application*. However, the user can now also use the *Mobl application* to access and modify the data. The synchronization allows the Mobl application to request data and send modifications.

The *system boundary* shows the part, which this thesis shows the scoop of the solution. The central part is the *synchronization addition to the compiler*, which depends on modules from the *original WebDSL and Mobl compiler*. The output of this addition, a set of generated WebDSL and Mobl files, is another important part of the solution. *Language developers* are responsible for maintainance and development of the compilers. This includes the additional module for synchronization and its output.

6.2 Decomposition

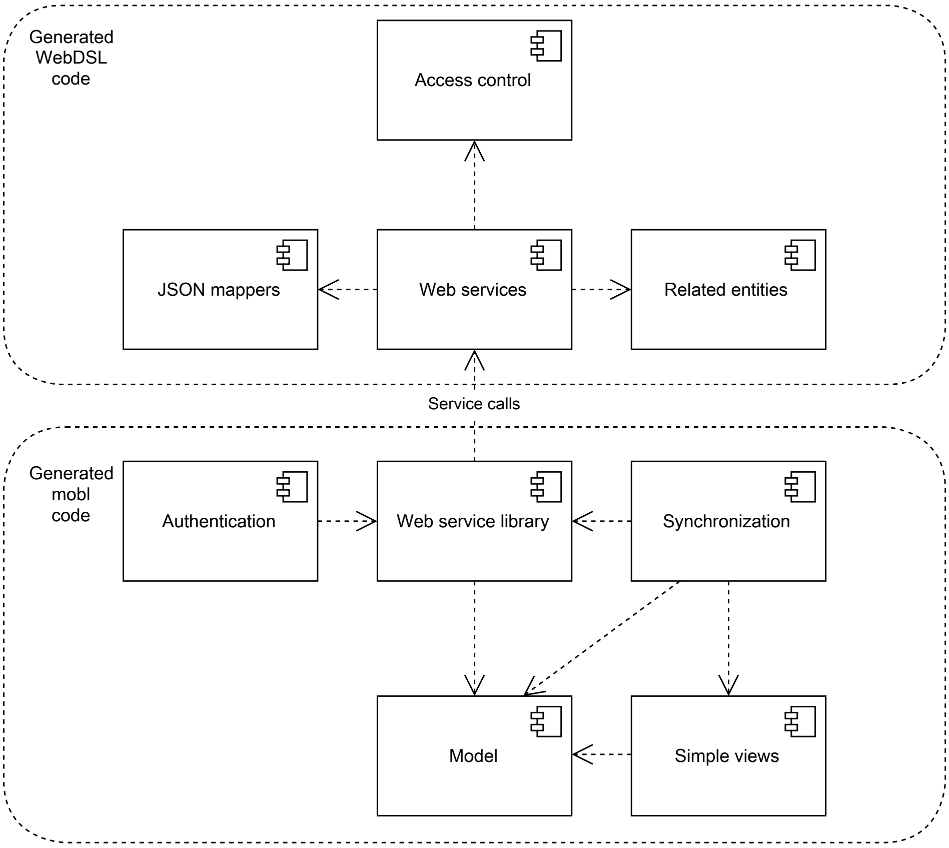
This section will state the decomposition of the generative solution. The decomposition is separated in two parts that of the generated code (Section 6.2.1) and of the compiler extension (Section 6.2.2). Both sections will describe its components and their responsibilities within the system.

6.2.1 Generated code

The framework that is generated by the compiler extension is evolved out of the motivating example. There are application specific parameters for the generation, but the general architecture is the same. It has a similar decomposition as presented in Section 5.3.2 because it is guided by the motivating example. Components with equal names are included in this solution. This does not imply the same responsibilities. An overview of the decomposition is shown in Figure 6.2.

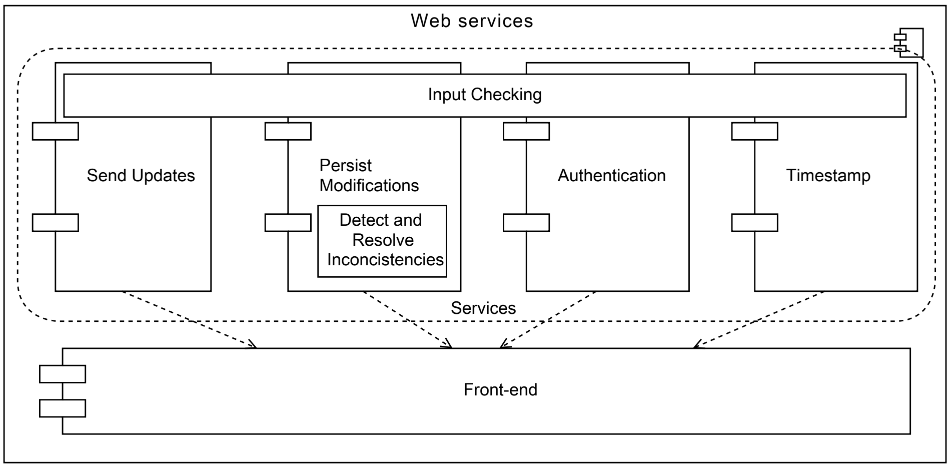
Webservices
The main point of the synchronization framework is focused in the webservices. The synchronization and the core algorithm of the synchronization can be found in this module.

The webservices component is the biggest component. We have chosen not to separate those concerns because generated code does not concern the same demands as manually written code. An advantage is that it improves and simplifies the control on the level of webservices. The seven main services and functionalities are displayed in Figure 6.3. The next paragraphs provided explanation of their responsibilities.



Arrows display dependency relations

Figure 6.2: Decomposition diagram for generated code of the generative solution



Arrows display dependency relations

Figure 6.3: Decomposition diagram for web services component

Front-end

The front-end of the webservices is an entry point to provide a general interface for webservices, which consists of registration, dispatching and basic error handling of services. The front-end can use the normal service names without name conflicts with existing web pages.

Send Updates

One of the activities of synchronization is the handling of updates. The component includes for each Entity a service that finds and sends updates based on the combination of, the timestamp of the last request and the partition identifiers.

Persist Modifications

The mobile application sends their modifications to the server in two forms: new and edited objects. Those updates are applied to the database when they do not create inconsistencies in the database.

Input Checking

The webservices are open for any form of JSON input. However, the services require specific formats and those are not checked on calling. The webservice module checks for every use of the JSON object if the parameter is available and contains the correct type. When this is not the case, the services will contineu to the next step and return an error. The entity specific input checking is handled by the mappers.

Detect and Resolve Inconsistencies

Detection of inconsistencies is done at the level of webservices. Before applying updates it checks if the object is out of date, and after the modifications the object is checked if it does still comply to the validation rules of the model. When some inconsistency is discovered during the process, a rollback is triggered so that the database stays correct.

Authentication

The authentication consists of simple services that have to be available for access control on synchronization. It provides registration and (de)authentication for users of mobile devices.

Timestamp

The updates require for an incremental approach a timestamp to define the last synchronization. The time on mobile devices cannot be used, since they could be different from the one used by the server. This difference is caused by manual setting of time or use of different timezone. To solve the problem, the component offers a request for current timestamp on the server.

Webservice Library
The module in the mobile part separates the synchronization and webservices. This module makes webservices available to the Mobl application. It maps the parameters to the right format and sends the request to the webservices. When the service returns it is passed to the specified mapper to handle the result. The mappers are included in this module, since the motivating example shows that the services are highly dependent on the mappers and are very small and simple.

Synchronization
This is the core of the synchronization in the mobile application. Most of the synchronization algorithm is calculated on the server. Nevertheless, the mobile application is responsible for delivering the correct input for the algorithm. This component is focused on integration and usability for the application developer. It delivers some integration functions so that synchronization can be minimized to a few function calls. It also delivers some views that can be used to display status from the synchronization, or for selecting topLevelEntities.

JSON mappers
The mappers have the responsibility to map objects to JSON format, to be serialized for the webservices. The serialize mappers are equal to those described in Section 5.3.2, which are split up in a minimal, simple and full version.

The modification mappers have to deal with two variants: modification and creation of objects. The synchronization algorithm takes care of this difference and makes it possible that the same mapper can be used for both instances. As stated before an additional responsibility for the mappers is to check if the JSON objects are in the correct format for that entity and return errors and warnings when encountered.

Related Entities
For the improvement of bandwidth and computation for the mobile devices, the synchronization algorithm has chosen another approach, where the objects requested are not added as parameter. This has as consequence that a different approach is needed to calculate the objects that are part of a partition. This component is responsible for calculating for each entity which objects are in the partition of a given object.

Model
The model is a simple component and has no real functionality. However, it is a important part of the synchronization framework. This component defines the persistent data model and is used throughout the whole mobile application. This includes the basis of the synchronization framework by enabling libraries for data access and mapping functions.

Access Control
The data of the web application should be secure. This requires an additional layer in the framework to restrict data access based on user information. The access control on the synchronization algorithm cannot be applied on the level of services because user access to objects depends on the requested data. This component specifies for every entity functions which can be used to check if a user is allowed to read, write or create an object.

Authentication

The access control requires authentication of users, but it can also be useful for other mobile application functionality. This component delivers an interface for the developer to use authentication within the mobile application. The component uses the credentials to register the device and stores the registration result to authenticate the device in the future, without requiring credentials. In case of deauthentication or denial of authentication the component is responsible for removing all local data to prevent data leakage.

Simple Views

At development of the motivating example, I discovered that a big part of the development time is in the GUI part of the application. This part of the application is not a required part for the synchronization. For this reason, the final solution includes a simple GUI to enable simple testing of the synchronization.

Normally views are highly dependent on the application and do not allow abstraction. The solution we used in this thesis for this problem is to generate a data browser. It enables the user to browse through the data by clicking on properties and modify simple property with basic types. The views are not meant for the final version of the application because it enables users to modify all data without restrictions.

Modified versions of those views are used as default to show the related object of synchronization errors. This is done to improve usability by showing users the objects instead of the identifiers. It is most likely that the end application prefers another approach of showing errors.

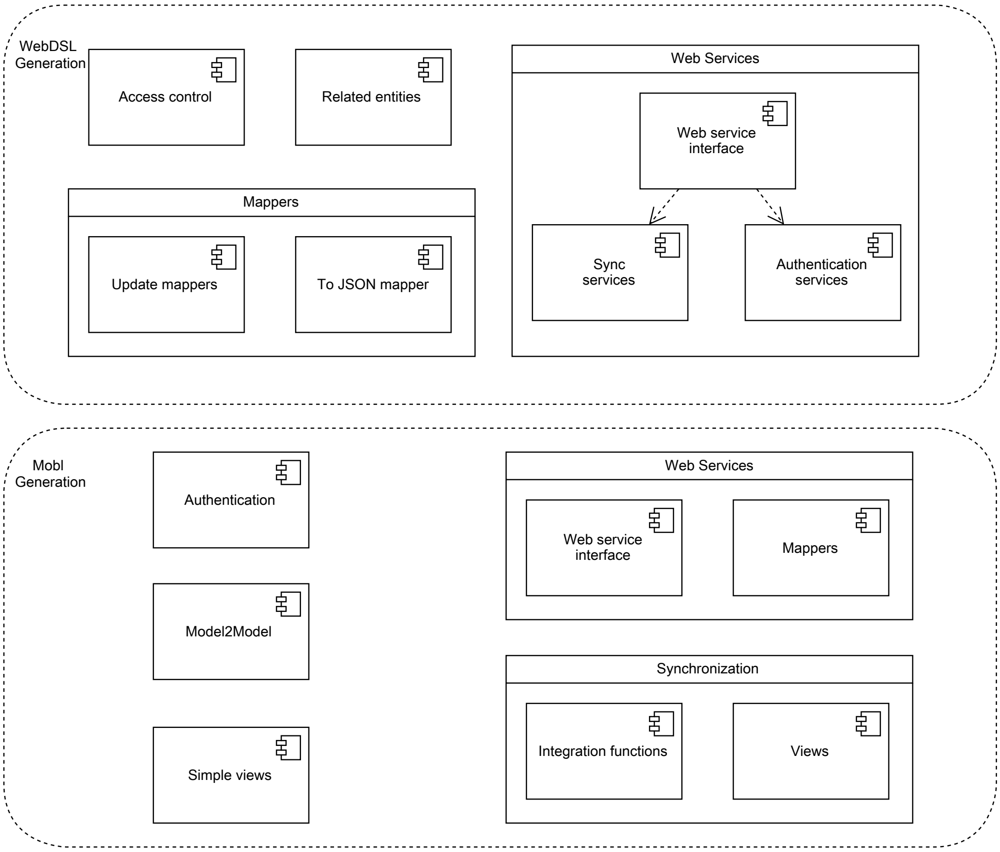
Next to the components of the generated code that are described in this section there are components in the compiler those will be described in the next section.

6.2.2 Compiler extension

The compiler extension is an additional set of components that are added to the compiler of WebDSL. The decomposition is based on the decomposition of the generated code and each part is responsible for generation of the similar named component. Those components are displayed in Figure 6.4. The diagram shows two outstanding particularities compared to the previous diagram.

The first characteristic is that this diagram has a low number of dependencies. The generation has a low-level of dependencies, since in general it gathers all required information from the source analysis that is part of the original compiler. This means that each of the components is independent of the information that is delivered by the other components of the extension.

The second difference is that some of the components are separated into smaller components, this has to do with the fact that some of the generated components are bigger and has several main responsibilities, those responsibilities are separated in the compiler by subcomponents.



Arrows display dependency relations

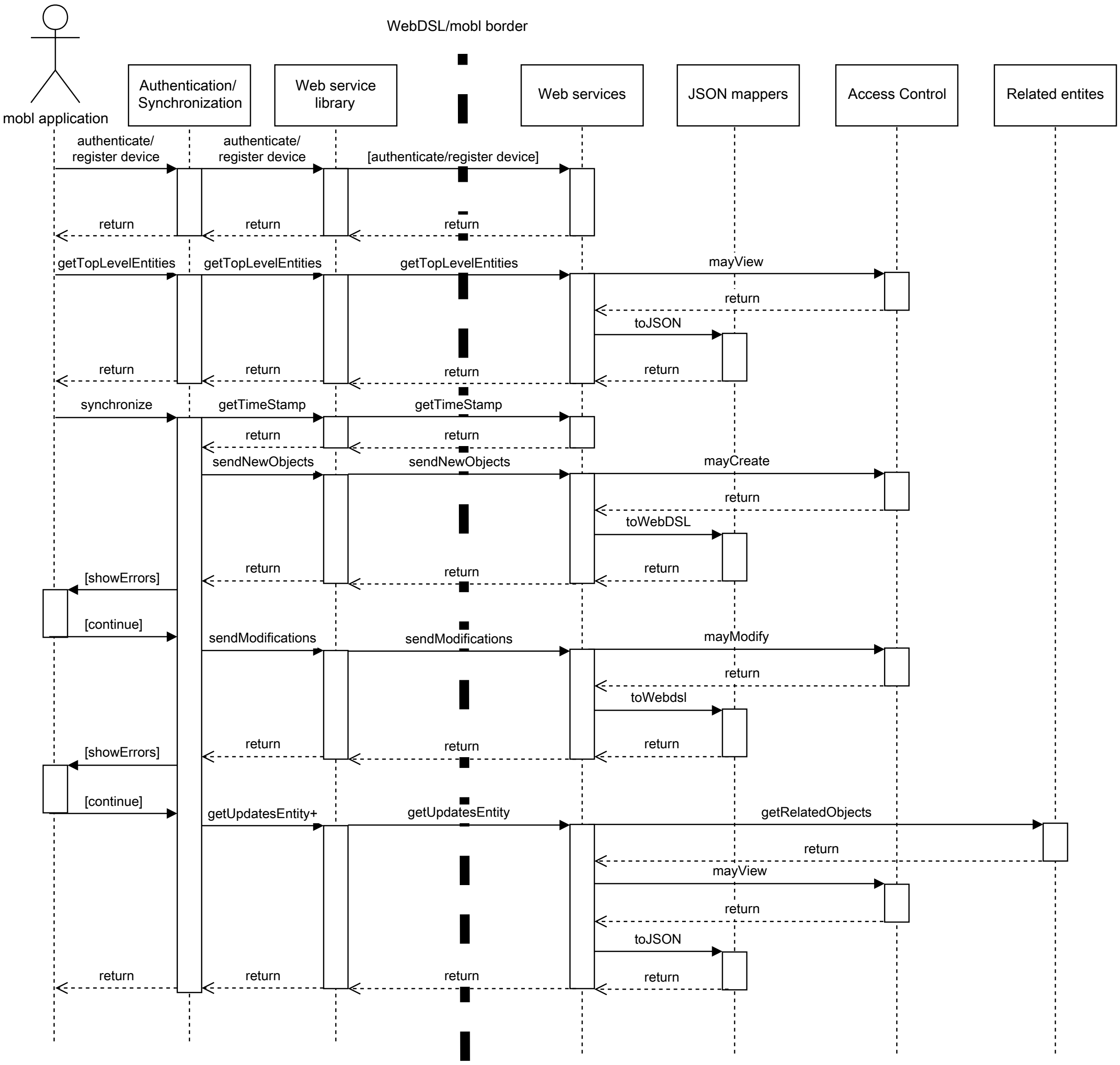
Figure 6.4: Decomposition diagram for compiler extension of the generative solution

6.3 Control Flow

The previous section introduces the components of the system and its dependencies. The control flow is meant to give a better impression of the execution and order of the system. This section also separates the two parts because they are executed independently from each other and therefore do not influence the execution order of the other code.

6.3.1 Generated Code

The generated code is evolved from the structure of the motivating example. Therefore, the code has a similar execution to that of Section 5.3.3. The abstraction that is applied does not influence that general flow, namely: authentication, get and select partitions and synchronization. The last step is a repeated process to send modifications and update the local data. The control flow of the generated code is displayed in Figure 6.5 and shows a simplification on the Mobl side. This is caused by the fact that the application workflow itself is unknown for the synchronization framework.



[] = optional, / = or , + = multiple times

Figure 6.5: Sequence diagram for the generated code of the generative approach

Authentication and Select Partitions

Authentication and selection of partitions are not really changed compared to that of the motivating example. The only considerable change is the allowance of multiple entities to represent a partition although this does not change the execution process. The difference that can be found in the sequence diagram is caused by applied abstractions and the change of responsibilities for each of the components. The only addition is that access control adds a check on objects to verify if objects are allowed to be sent.

Synchronization

The synchronization process has changed considerably by the modifications to the algorithm and therefore this paragraph will be more elaborated. The synchronization process has been split into: get timestamp, send new objects, send modified objects and request updates.

Firstly, a simple server request to get a timestamp from the server is needed to update the local timestamp of last synchronization. This is requested from the server to have a guaranteed correct and equal time range for next synchronization. The time of the mobile devices should not be used because the timezone can be different and can be modified by the user.

The sending of new objects, requires all new objects which are mapped to JSON so that the can be sent to WebDSL. On the WebDSL side the objects are checked on access control to verify that the user is allowed to create the object. When this is the case the objects are mapped to WebDSL objects and without inconsistencies are persisted to the database. Errors and warnings occurred in this process are returned to Mobl application, where they are possibly shown to the user to interact before continuing the synchronization process. The step of sending modifications is similar to this step.

The last step requires updates from the server, which is done separately for each entity. All identifiers of selected partitions with a timestamp of previous synchronization are send from the Mobl application. The identifiers are used to find all related entities for the partitions. After checking whether the objects are changed and allowed to propagate, the objects are transformed and returned to Mobl.

6.3.2 Compiler extension

The control flow within a compiler is different from normal applications. The extension to the compiler cannot be seen separately from the compiler. The process within the compiler can be interpreted as a pipeline of actions that are executed on the source to produce an executable. A simplified overview of the pipeline in the WebDSL compiler is displayed in Figure 6.6. More information on the execution order and pipeline of WebDSL are described in the paper ”Code generation by model transformation: a case study in transformation modularity” [20]. The next part will explain step for step the working of the pipeline.

The input *source files* delivered to the compiler are first *parsed*, which delivers an Abstract Syntax Tree (AST). This tree is first *normalized* to remove redundant syntax forms. This makes the input ready for *type and name analysis*. This analysis consists of *declaring* information, *renaming* to unique names and complete check of *type correctness*. After the first analysis, there is a stage to check for extra *constraints*. This allows additional checks related to domain and language knowledge. When no errors are encountered the AST is transformed to elements of the core language. This step is called *desugaring* and is meant to simplify the *code generation*. The code generation consists of two steps. Firstly, *model to model transformation*, which transforms the tree with core elements to elements of the target language Java. Secondly writing the Java AST to files.

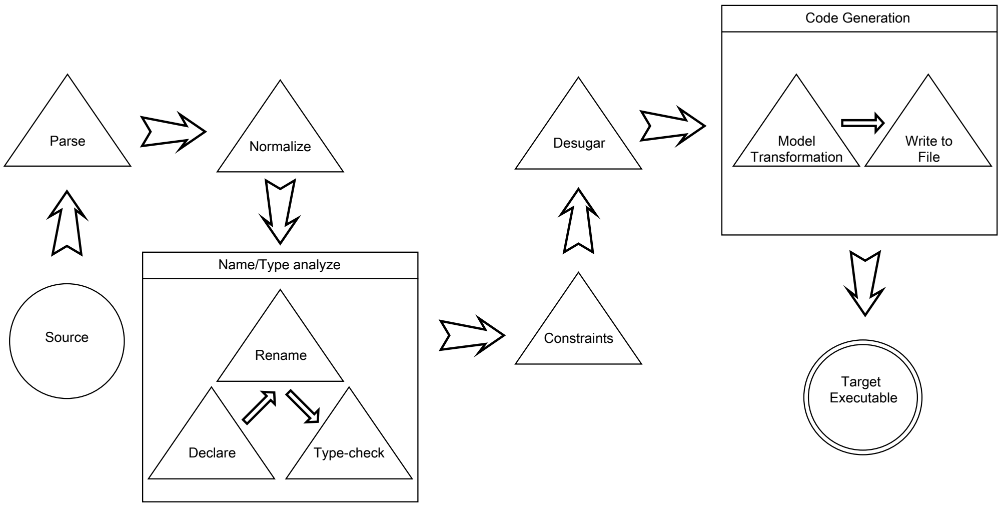


Figure 6.6: A schematic display of the WebDSL compiler pipeline

The compiler extension for synchronization is a code generation part and is added in the end of the pipeline. The additional syntax that was needed for the synchronization is added in the original compiler and required some additions in the other stages. The execution order shows the reason for the low number of dependencies, since the information used is already gathered in the previous stages. The implementation requires and specifies an order in the execution. Nevertheless, this order can be changed to any order of execution of the components.

This section wraps up the details of the working of the final solution more details will be explained in the next chapter. The following section will describe the external data used by the framework.

6.4 Information Architecture

The information architecture is meant to give insight in the information that is required by the solution. There is a high amount of internal passing of information on parameters, but often this is straightforward passing of objects. This section focuses on the data that is shared with external sources.

6.4.1 Generated Framework

The generated code of WebDSL and Mobl communicates through webservices. When looking at the level of the framework the communication is internal. However, the webservices are open to be used by other remote applications. WebDSL should not act different between the generated Mobl application or other applications that access the services. That means that both using the same API and the services should be considered external information. It is interesting for understanding the algorithm and possible developers for remote applications to see what parameters are required and what data is returned by the webservices.

There are eight sort of services with different purposes as described in Table 6.7. The services all return an error and result, which makes it possible to have result and error messages for the same request. This gives the services the possibility to return information about the process in addition to the result, which can be useful for the remote application actions.

The input and output for each of the services is given in Table 6.8. The services often require more complex structures to communicate, those types are explained in Appendix A

Service	Description
Register Device	Allows the user to register device with user credentials
Authenticate Device	Enables to authenticate device with devicekey
Logout	Logs out current device for the user
Get TopLevelEntities	Returns a list of all objects corresponding to a partition
Get Timestamp	Returns a timestamp of the current time on the server
Send New Objects	Enables creation of new objects for remote devices
Send Modified Objects	Enables modification of objects for remote devices
Get Updates Entity	Returns list of changed objects from given partitions

Table 6.7: List of webservices with corresponding description of functionality

Service	Parameter	Result	Error
Register Device	UserCredentials	DeviceKey	String*
Authenticate Device	DeviceCredentials	Boolean	String*
Logout	{ }	Boolean	String*
Get TopLevelEntities	{ }	EntityListObject*	String*
Get Timestamp	{ }	Long	String*
Send New Objects	EntityListObject*	{ }	[ErrorObject, String]*
Send Modified Objects	EntityListObject*	{ }	[ErrorObject, String]*
Get Updates Entity	PartitionListObject	EntityObject*	String*

{ } = empty object, * = zero or more , [] = grouping

Table 6.8: The webservice interface specifications of the generated code

6.4.2 Compiler Extension

The compiler uses different information than that of the generated code. This is because the generation is based on information of the source code that is available in the compiler (Section 6.3.2). The information it uses is restricted mostly to the data model. The information gathered from the original compiler is described in the following paragraphs.

Entities

Entities are used as core information of WebDSL applications. They are used to describe the data model of an application. Figure 6.9 shows an example of such an entity. This entity contains a list of properties, but it can contain also other code like functions. The information that is required for the generation of the synchronization framework is the name of the entity and the properties it contains.

```
entity Project {
  title       :: String (search)
  description :: WikiText
  owner       -> Person (inverse=Person.projects)
}
```

Figure 6.9: Example of an entity in WebDSL

Properties

The properties are part of an entity and contain its own information. The basic information of a property is its name and type. It is possible to add additional information to the properties with annotations. Figure 6.9 shows the additional annotations search and inverse relation. The model-to-model transformation tries to reuse this information for the generated Mobl model.

Hierarchy Information

The data model of WebDSL can contain additional information, that of the inheritance hierarchy of entities. An entity can define a relation of a parent entity. As example a *user* can be a specialized form of *person*, which is shown in Figure 6.10. This relation means that the child entities inherits the properties of his parent.

```
entity Person {
  firstName :: String
  lastName  :: String
  fullName  :: String
  projects  -> Set<Project>
}

entity User:Person {
  username :: String
  password :: Secret
}
```

Figure 6.10: Example of inheritance in data model of WebDSL

Security Information
The synchronization framework contains an addition to the original authentication code. To extend this functionality it requires the entity with the properties that are used for authenticating a user. Figure 6.11 shows an example of code that is used in a WebDSL application to define the principal information. This line enables authentication and access control for the WebDSL application.

```
principal is User with credentials username, password
```

Figure 6.11: Example of principal declaration in WebDSL

Type Information
The last part of information required from the compiler is type information. This includes the types of the properties, but also of expressions. The type information is mainly used for map WebDSL to Mobl types.

This concludes the information that is used from the original compiler for the code generation. However, there is additional information required for the generation.

DSL Addition

The information required of WebDSL applications requires an addition to the syntax. However, this addition needs to be inline with the design choices made in WebDSL. This requires that the addition is separated from the other DSLs to keep the separation of concerns. Another important aspect is that try to make the DSL as formal as possible.

This paragraph covers the additional syntax elements added. Those elements are explained with an example. Additionally, an explanation is given of what information it contains and where it is used for.

```
entity Project {
  title       :: String (search)
  description :: WikiText
  owner       -> Person (inverse=Person.projects)

  synchronization configuration {

  }
}
```

Figure 6.12: Example of synchronization configuration declaration

Main Declaration
The DSL contains a main entry to separate the synchronization code from the rest of the application code. An example of this is given in Figure 6.12. It shows an additional code part in the entity. We have chosen to add this to the entity body because all the required information is entity based. The syntax itself does not modify the code generation so it only needs to be added for specifying one of the options.

TopLevel Entity
The synchronization framework uses object relations for data partitions. Objects are used to represent data partitions. This requires from the user or developer to select a subset of objects to be partitions. This selection is done on entity basis. An entity can be selected to be a toplevel node for data partitioning. Figure 6.13 shows an example of such a declaration. This example displays that the toplevel entity requires a string property to represent data partitions. In the example we chose title as name property because it is a short, unique and effective description of the object. We have chosen to enforce this name property for the usability of the synchronization framework as end users. Selection of partitions by a meaningful string is prefered over the usage of UUIDs. In the framework it uses the UUID of the object to prevent conflicts by changing the title.

```
entity Project {
  title       :: String (search)
  description :: WikiText
  owner       -> Person (inverse=Person.projects)

  synchronization configuration {
    toplevel name property : title
  }
}
```

Figure 6.13: Example of toplevel entity declaration in synchronization configuration

```
entity Person {
  firstName :: String
  lastName  :: String
  fullName  :: String
  projects  -> Set<Project>

  synchronization configuration {
    restricted properties : firstName, lastName
  }
}
```

Figure 6.14: Example of restricted properties declaration in synchronization configuration

Property Restriction
The synchronization framework allows modification on the model to simplify the model on mobile application. This simplification must be selected by the developer to restrict the data model on entity basis. The configuration allows restriction on own properties only. The example in Figure 6.14 displays that the properties firstName and lastName are removed from the model on mobile side. Recommended is to remove properties from the model if they are not used for the mobile application. A possible choice is duplication of information like in the example, which also contains the fullName.

Security Rules

The framework has added support for access control, which will be explained in the next section. The basic concept is that a developer can define rules to restrict data access based on the user. Since it is based on the current user, the application should have declared a principal. This is done on object level. There are three different levels: read, write and create. The developer has to specify a boolean expression for each of those levels. Figure 6.15 shows an extension to the earlier example of the project entity. The added rules specify that that everybody can read the data. However, modifying the project object can only be done by the project owner. The restriction for creation of objects is that the user at least has to be logged in.

```
entity Project {
  title       :: String (search)
  description :: WikiText
  owner       -> Person (inverse=Person.projects)

  synchronization configuration {
    toplevel name property : title
    access read: true
    access write: currentPrincipal() == owner
    access create: isLoggedIn()
  }
}
```

Figure 6.15: Example of security rules declaration in synchronization configuration

This is concludes the DSL that is added to WebDSL, to provide the addition required information. The next section will state the security and robustness details of the framework.

6.5 Security and Robustness

Security and robustness are important factors for web systems, which in the motivating example did not have too much priority. This section explains what principles are applied to improve both aspects for the generated code. The aspects are to a certain extent also important for the compiler. However, the code does not influence those aspects at the level of architecture.

6.5.1 Security

In the case of the synchronization (in general also for web applications), security is focused on data. This is mostly protection against data leaking and prevention of unwanted modifications in the database. The first step in securing is the limitation of the model so that information like passwords are never shared. To further reduce the data leaking, every object can have an access control rule that specifies if an object may be synchronized by that user. Extra levels are added for creation and modification so that there is also protection for the database. This measure is totally dependent on the developer specifying correct rules. Another measure to protect the database is delivered by WebDSL, when only setting properties of objects, SQL injections are prevented.

Additional security is added to authentication process by using devicekeys instead of storing the user password. This allows the device to log in automatically without storing password of the user. The user can deregister the device if it is seized, preventing other people from misusing the application. Deauthentication notices by the authentication module will lead to resetting the local database. An extra security layer can be added for the webservices by accessing the application through HTTPS. However, this is not forced and should be manually enabled for web applications.

6.5.2 Robustness

Robustness is basically the code taking care of unexpected input without making the system unstable. The security measurements already reduces the changes of unexpected input by restricting users. Nevertheless, it is true that even trusted users could apply unexpected modifications to objects. The unknown remote applications using the webservices are even more free on their input.

The most important measure is taken by checking all JSON input on availability and correct type of the parameter before it is used. Additional to this check are the validation rules applied on objects, those are checked before persisting the changes to the database. This prevents inconsistent data which could lead to unwanted behavior of the system.

Another possibility to get unstable systems is delivered by concurrency in the system. This is not the case in the solution, since all the service calls are executed in separate threads and do not interact with other system functionality. This gives as advantage that wrong services calls are isolated from the rest, making it impossible to damage other processes.

There is no possibility that the algorithm shows unexpected behavior, since the synchronization framework has no functionality for suspicious code to be executed. The algorithm is of course dependent on the input of the services. Although it cannot create never-ending loops.

6.6 Summary

This chapter introduces the final solution, a generative approach for synchronization between web and mobile applications, by describing the architecture of the system on the hand of the following viewpoints: Context (Section 6.1), Decomposition (Section 6.2), Control Flow (Section 6.3), Information Architecture (Section 6.4), and Security and Robustness (Section 6.5).

The context gives an overview of the system as displayed in Figure 6.1, which is a generalization of the motivating example plus an addition of the compilers that are used for the code generation.

Decomposition of the generated code as displayed in Figure 6.2 has the motivating example as starting point and therefore it has similar components. However, some of the responsibilities have changed. The compiler extension is basically a one to one mapping for every component of generated code, it has a component that is responsible for generating the equally named component. Figure 6.4 shows that the components in the compiler have almost no dependencies.

The reason for the low number of dependencies can be found in the fact that the pipeline of the WebDSL compiler (Figure 6.6) delivers decoupling of stages. The execution of the compiler extension finds place in the last stage namely, code generation. The control flow of the generated code (Figure 6.5) is in general the same to that of the YellowGrass Mobl. The biggest difference is caused by the modifications in the synchronization algorithm.

The information flow is more interesting on the level of communication with external systems. In the case of the generated code this is done through webservices. The webservices interfaces are summarized in Table 6.8 and Table A.1. The code generation requires information from the source code. For customization of the generated framework addition information was needed and is delivered by an extension of the language.

Security and Robustness are two important non-functional aspects for web systems. Specialized access control for objects verifying the input are the most substantial additions in the generated code to support both aspects. The next chapter will go into more details of the solutions and explains interesting implementation properties of the solution.

Chapter 7

Implementation

The previous chapter introduced the generative approach to data synchronization for web and mobile applications, by explaining the architecture from several viewpoints. This chapter focuses on the interesting details of the implemented and generated code of this thesis solution.

The generated code is a product from the compiler extension code. That makes it a separate process as discussed in the previous chapter. For that reason this chapter will use the separation between runtime and generation to discuss the implementation details in Section 7.1 and Section 7.2.

The solutions described in Chapter 5 and 6 plus the first part of this chapter is focused on general solution that is non-specific to the target languages WebDSL and Mobl. The second goal of this chapter is to explain some target language specific difficulties that needed to be solved to get the synchronization framework working (Section 7.3).

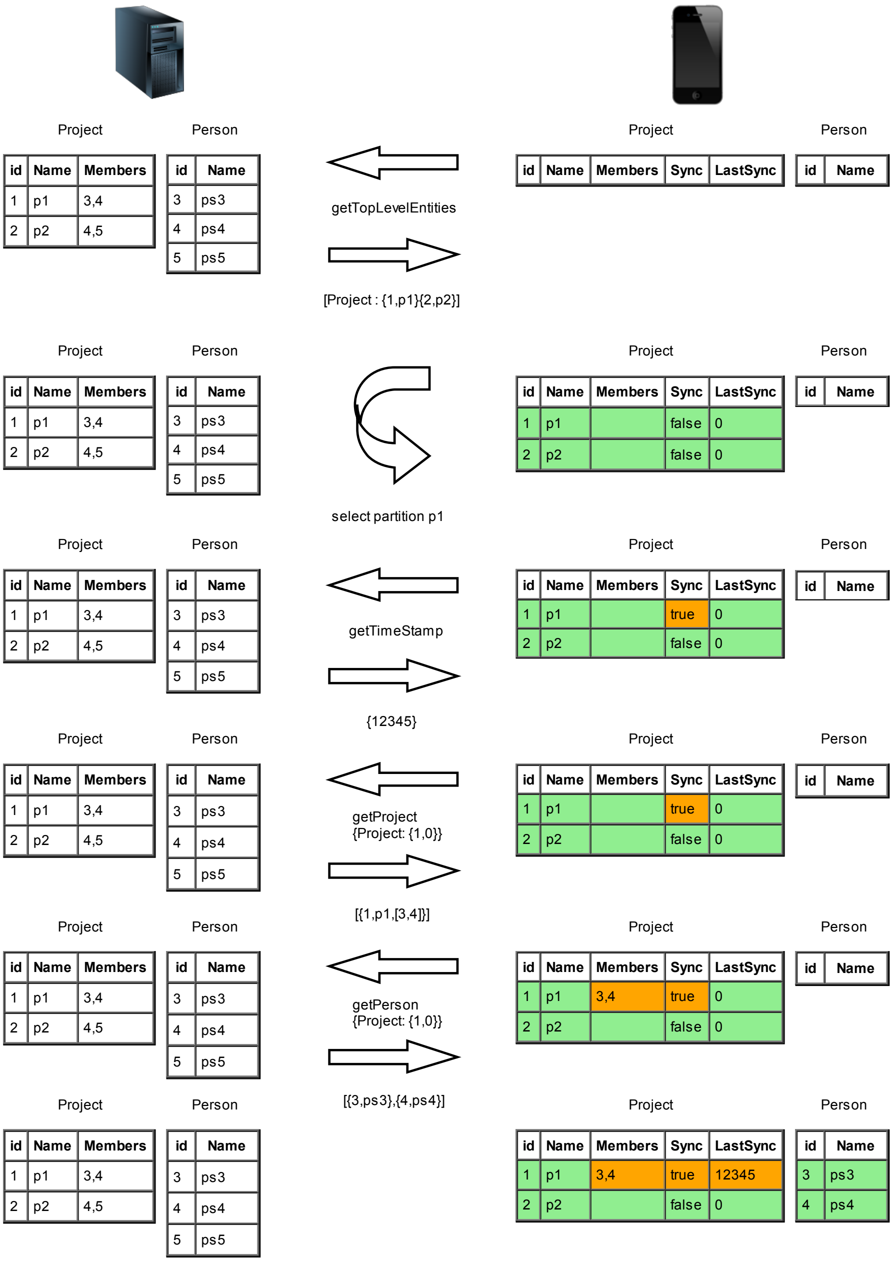
7.1 Generated Code

The motivating example was a starting point for the generated code. Nevertheless, some of the modifications are thorough and change how the code works. A big part of the implementation is a straightforward translation of the functionality described in the architecture design (Chapter 6). The more complicated implementations are found in the following functionality: Synchronization (Section 7.1.1), Related entities (Section 7.1.2) and Simple views (Section 7.1.3).

7.1.1 Synchronization

The main functionality for the generated framework is synchronization. It has to deal with several aspects that all have to be integrated into the synchronization algorithm. The algorithm has changed to reduce computation and bandwidth on the mobile devices. The pseudocode of the synchronization steps can be found in Appendix B

An example of the synchronization is displayed in Figure 7.1. This example shows first full synchronization cycle. The mobile application first requests all the partitions, this makes it possible to select one for next step by setting the true flag. After that the timestamp is requested for later purposes. The next two steps represent the request of all data objects for the selected data partition. The last step is to modify the lastSync property of the partition with the timestamp requested earlier. The rest of this section will explain each of the steps of the synchronization activities for both server and client side.



Project

id	Name	Members
1	p1	3,4
2	p2	4,5

Person

id	Name
3	ps3
4	ps4
5	ps5

Project

id	Name	Members	Sync	LastSync

Person

id	Name

Project

id	Name	Members
1	p1	3,4
2	p2	4,5

Person

id	Name
3	ps3
4	ps4
5	ps5

Project

id	Name	Members	Sync	LastSync
1	p1		false	0
2	p2		false	0

Person

id	Name

Project

id	Name	Members
1	p1	3,4
2	p2	4,5

Person

id	Name
3	ps3
4	ps4
5	ps5

Project

id	Name	Members	Sync	LastSync
1	p1		true	0
2	p2		false	0

Person

id	Name

Project

id	Name	Members
1	p1	3,4
2	p2	4,5

Person

id	Name
3	ps3
4	ps4
5	ps5

Project

id	Name	Members	Sync	LastSync
1	p1		true	0
2	p2		false	0

Person

id	Name

Project

id	Name	Members
1	p1	3,4
2	p2	4,5

Person

id	Name
3	ps3
4	ps4
5	ps5

Project

id	Name	Members	Sync	LastSync
1	p1	3,4	true	0
2	p2		false	0

Person

id	Name

Project

id	Name	Members
1	p1	3,4
2	p2	4,5

Person

id	Name
3	ps3
4	ps4
5	ps5

Project

id	Name	Members	Sync	LastSync
1	p1	3,4	true	12345
2	p2		false	0

Person

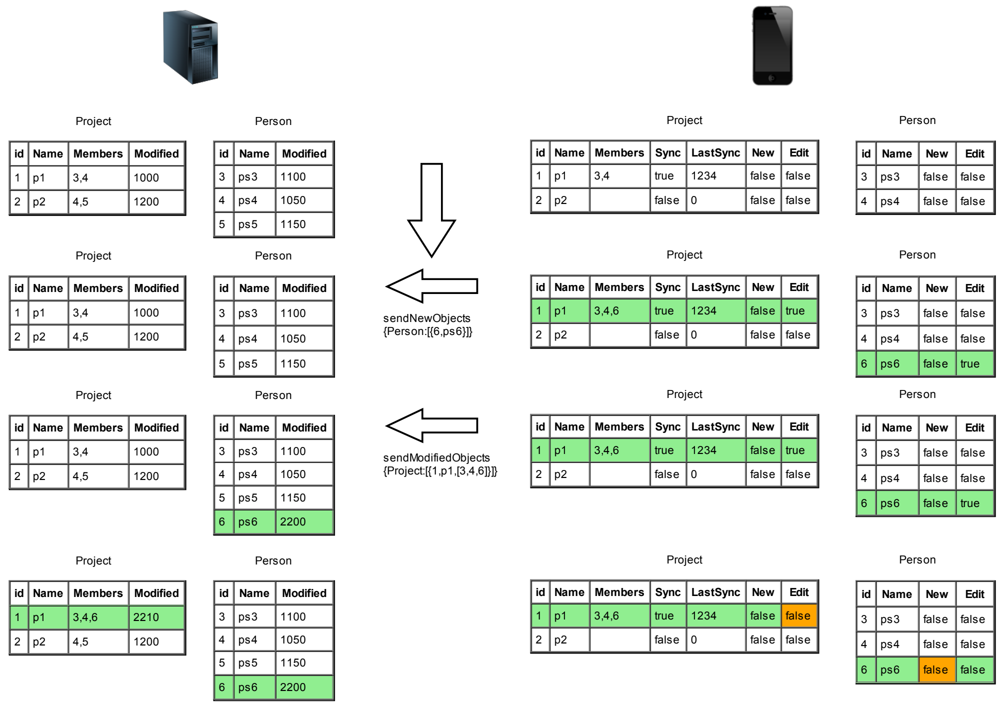
id	Name
3	ps3
4	ps4

green = modified data, orange = second modification

Figure 7.1: Example of a synchronization with a clean mobile application

Identification of Updates

The first step of synchronization is identification of updates. In the improved synchronization algorithm we have chosen to for a new separation in the process. It is split up in first synchronizing the updates from the mobile client to the server. After that synchronizing the updates from the server to the mobile device.



Project

id	Name	Members	Modified
1	p1	3,4	1000
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4	true	1234	false	false
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

Project

id	Name	Members	Modified
1	p1	3,4	1000
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4,6	true	1234	false	true
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false
6	ps6	false	true

Project

id	Name	Members	Modified
1	p1	3,4	1000
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150
6	ps6	2200

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4,6	true	1234	false	true
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false
6	ps6	false	true

Project

id	Name	Members	Modified
1	p1	3,4,6	2210
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150
6	ps6	2200

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4,6	true	1234	false	false
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false
6	ps6	false	false

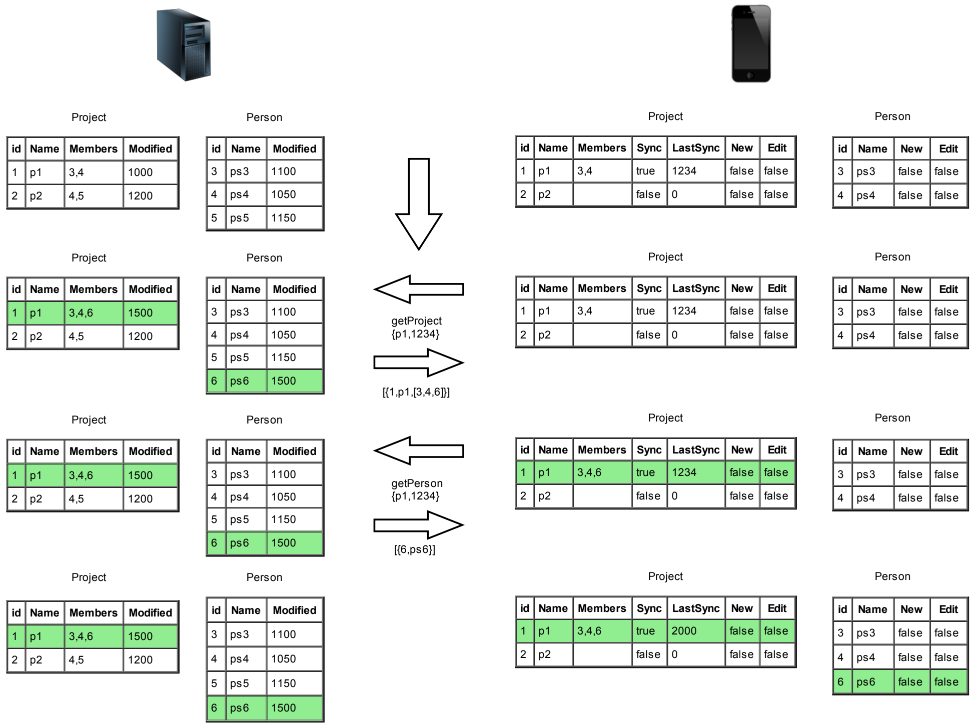
green = modified data, orange = second modification

Figure 7.2: Example of a synchronization with a changed data in the mobile application

The algorithm for update identification in Mobl splits synchronization of new and modified objects and does not require a list of objects that have to be synchronized. The new function queries for each entity in the data model if instances of that entity are new by checking if the created flag is set and then put those objects in a list. Discovering of new objects functionality is displayed as second step inFigure 7.2 and. The two additional fields new and edit are used to check for modified and created objects.

The server side identification got more complicated because it gets a list of partitions instead of all objects that are requested. Each entity in the data model has a separate function that returns the related objects of the given partitions. For each TopLevel object it finds all the objects of the entity type in a partition, which is explained more elaborated in Section 7.1.2.

A last check is added to filter out the objects that are not changed after the previous synchronization. This can be done by comparing the timestamp it gets from the request and the last modified timestamp of the object. As you can see in the second and third step of Figure 7.3. The time in modified field is higher than that of the timestamp from the request.



Project

id	Name	Members	Modified
1	p1	3,4	1000
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4	true	1234	false	false
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

Project

id	Name	Members	Modified
1	p1	3,4,6	1500
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150
6	ps6	1500

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4	true	1234	false	false
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

Project

id	Name	Members	Modified
1	p1	3,4,6	1500
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150
6	ps6	1500

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4,6	true	1234	false	false
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

Project

id	Name	Members	Modified
1	p1	3,4,6	1500
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150
6	ps6	1500

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4,6	true	2000	false	false
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false
6	ps6	false	false

green = modified data

Figure 7.3: Example of a synchronization with a changed data in the web application

Propagation of Updates

The second step, the propagation of updates, can be separated in two main activities: sending the updates, and converting and persisting the updates to local values. A simplified approach would be to map the updates to JSON and send them over. The receiving side would get the JSON and map them to the local values in the database.

For the mobile application the function is reduced to those simple steps. The synchronization function first sends the new objects, then the modified objects, as represented in second and third step of Figure 7.2. This can be followed by an update request for every entity, which are mapped and persisted locally, which is displayed from step two in Figure 7.3. An explanation of the overall synchronization process is described in Section 6.3.1.

The synchronization process is driven by the mobile application, which has separated the synchronization algorithm in three stages. This required the following three features on the side of WebDSL: processing modifications, processing creations and propagation of local updates.

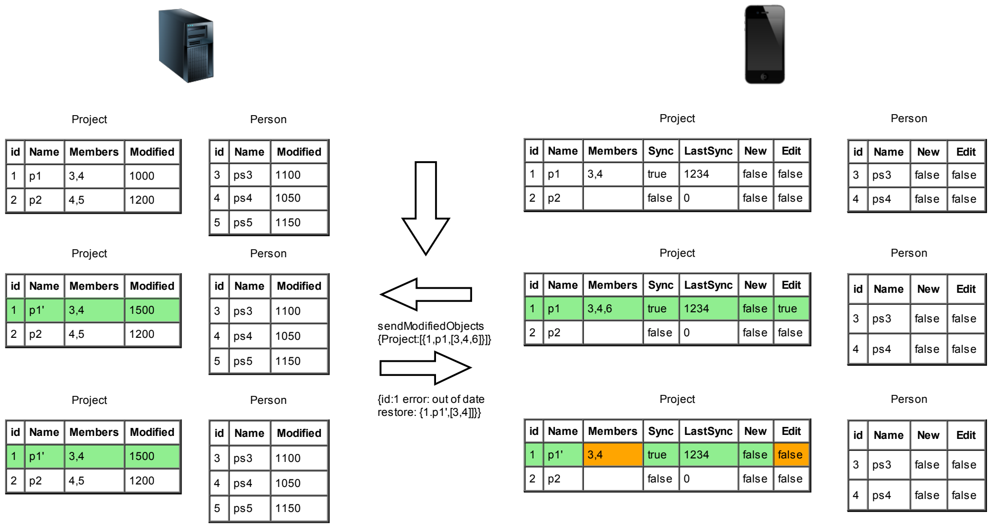
The processing of modified objects is implemented as follows: Finding the correct type for each of the objects. Based on the type and the id of the object, it can load the local object. The object is then used to map and apply changes that are specified by the input. This mapping is done by the editmapper that is generated for every Entity.

The creation of objects is similar to that of modifying objects. However, it is possible that several new objects are created that refer to each other. This situation could create errors on applying updates therefore an extra step is added. The additional code creates all the objects before the properties are set.

Detection and Resolution of Inconsistencies

The last part contains detection and resolution, which are integration tasks. Detection is done in entity bound functions. This information is gathered in a global synchronization algorithm function. With this approach it can decide and take action based on the errors. The main goal is maintaining an error free database on the server. The generated framework also implements some functionality to prevent inconsistencies in the database on the mobile application.

Detection
The detection of inconsistencies is often done in entity bound functions because there is information available to verify whether there are inconsistencies within the updates. This causes that the detection is spread throughout the generated code in several elements. For the server application, detection is done by three different checks: outdated updates, validation rules and additional mapper input checks.



Project

id	Name	Members	Modified
1	p1	3,4	1000
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4	true	1234	false	false
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

Project

id	Name	Members	Modified
1	p1'	3,4	1500
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

sendModifiedObjects
{Project:[{1,p1,[3,4,6]}]}

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4,6	true	1234	false	true
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

{id:1 error: out of date
restore: {1.p1',[3,4]]}}

Project

id	Name	Members	Modified
1	p1'	3,4	1500
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1'	3,4	true	1234	false	false
2	p2		false	0	false	false

Person

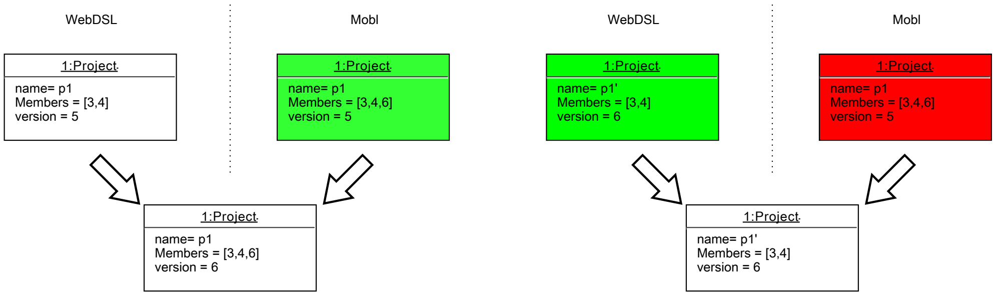
id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

green = modified data, orange = second modification

Figure 7.4: Example of synchronization with conflicting changed objects

A conflict with changed object on both sides is shown in Figure 7.4. The algorithm checks if the local object is not changed after the previous synchronization. This is done by comparing the version number of the local object and the update. This is shown in Figure 7.5. The update is only applied when the version is greater or equal to the local object, otherwise an error is returned.

The validation of objects is functionality that is created by the WebDSL compiler based on the rules specified on the entity. It consists of a method call that validates the object and returns a list of errors for violated rules. The returned errors are added to a list of errors for the object. The last step is implemented by additional checks on the mapping of the objects and consists of additional type input checks. The mapper can be overwritten for adaptation purposes, which means it allows the developer to add other checks on the input.



(a) No conflict

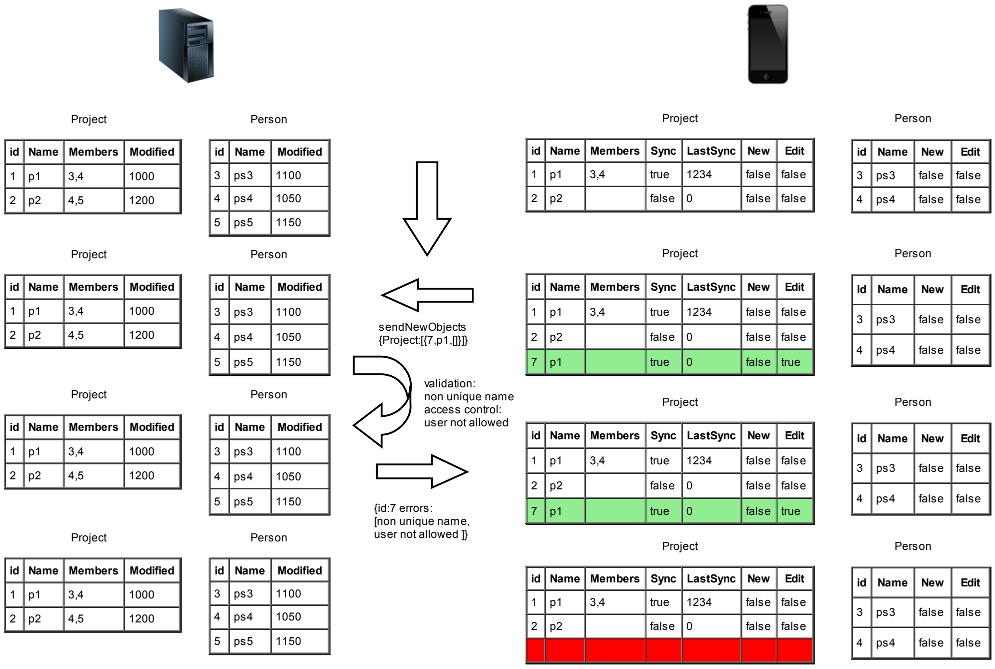
(b) Outdated object

green = correct update, red = incorrect update

Figure 7.5: Example situations conflict detection and resolution for objects

The detection of inconsistencies with new objects is similar to that of modified objects, but there is no possibility that a new object is out of date. An example of this is shown in Figure 7.6.

The mobile application does not have its own functionality to detect inconsistencies because local changes are sent and checked by the server. The possible inconsistencies are found in the step that propagates the updates. For detection of incorrect objects it uses the errors based on updates returned from the server.



Project

id	Name	Members	Modified
1	p1	3,4	1000
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4	true	1234	false	false
2	p2		false	0	false	false

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

Project

id	Name	Members	Modified
1	p1	3,4	1000
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

sendNewObjects
{Project:[{7,p1,[]}]}

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4	true	1234	false	false
2	p2		false	0	false	false
7	p1		true	0	false	true

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

validation:
non unique name
access control:
user not allowed

Project

id	Name	Members	Modified
1	p1	3,4	1000
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

{id:7 errors:
[non unique name,
user not allowed]}

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4	true	1234	false	false
2	p2		false	0	false	false
7	p1		true	0	false	true

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

Project

id	Name	Members	Modified
1	p1	3,4	1000
2	p2	4,5	1200

Person

id	Name	Modified
3	ps3	1100
4	ps4	1050
5	ps5	1150

Project

id	Name	Members	Sync	LastSync	New	Edit
1	p1	3,4	true	1234	false	false
2	p2		false	0	false	false
🟥	🟥	🟥	🟥	🟥	🟥	🟥

Person

id	Name	New	Edit
3	ps3	false	false
4	ps4	false	false

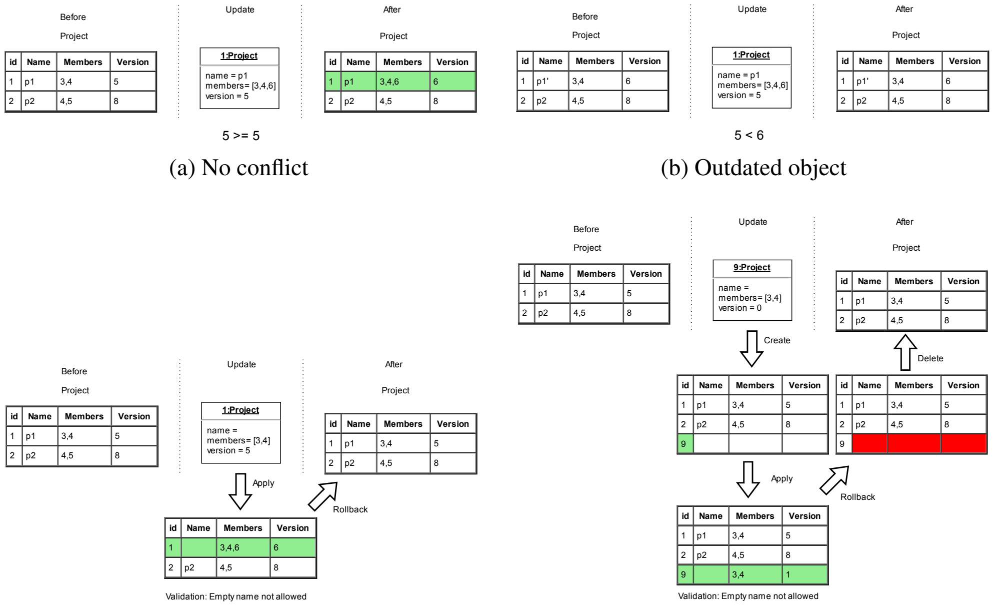
green = modified data, red = deleted data

Figure 7.6: Example of synchronization with conflicting new objects

Resolution

The resolution of problems often depends on the source. The solution used in this thesis is to have a high-level approach for resolution. This gives the opportunity to keep the process simple and have the responsibility of the resolution in one place.

The resolution of outdated objects is not needed, since it is detected before applying the changes, see Figure 7.7b. This is possible because the version can be checked before applying changes. The checking of correctness for objects might depend on a multiple fields of an object. Therefore, we have chosen to use rollback functionality of the database transactions, which cancels out the changes after they are applied to the object. The new transaction for every object makes it possible that only updates of the current object are not applied. The example in Figure 7.7c displays this situation. The modifications to new objects is similar to those applied for modified objects. Although an additional step is needed for new objects, which is deleting the object from the database to prevent ghost objects. This is shown in the last part of Figure 7.7.



(a) No conflict

(b) Outdated object

(c) Incorrect modification

(d) Incorrect creation

green = correct update, red = deleted data

Figure 7.7: Example situations conflict resolution for updates in web application

The mobile application did not need functionality for detection of errors. However, the errors can influence the incorrectness of the local database. After propagation of new and modified objects, it is possible to interact with the local data based on the errors. It is needed to clean up the local database before continuing synchronization.

The cleanup is integrated within the general mobile application synchronization function. Both modifications and creation have a similar approach: first clearing the flags of the correct objects to prevent that correct updates are applied multiple times. Followed by the possibility for the application to interact with local data. This gives the possibility to fix incorrect objects and retry the synchronization. When continuing the synchronization the algorithm assumes that the errors are approved so that the invalid changes can be rejected and restored to a correct state.

The first step is to select all objects which have a flag for modified or created. The difference of new and modified can be found in the way to resolve the inconsistencies. New objects are simply deleted from the database as displayed in the last step of Figure 7.6. The modified objects are restored to valid state given by the server and changing the dirty flag to false. This is shown in step three and four of Figure 7.4.

Access control

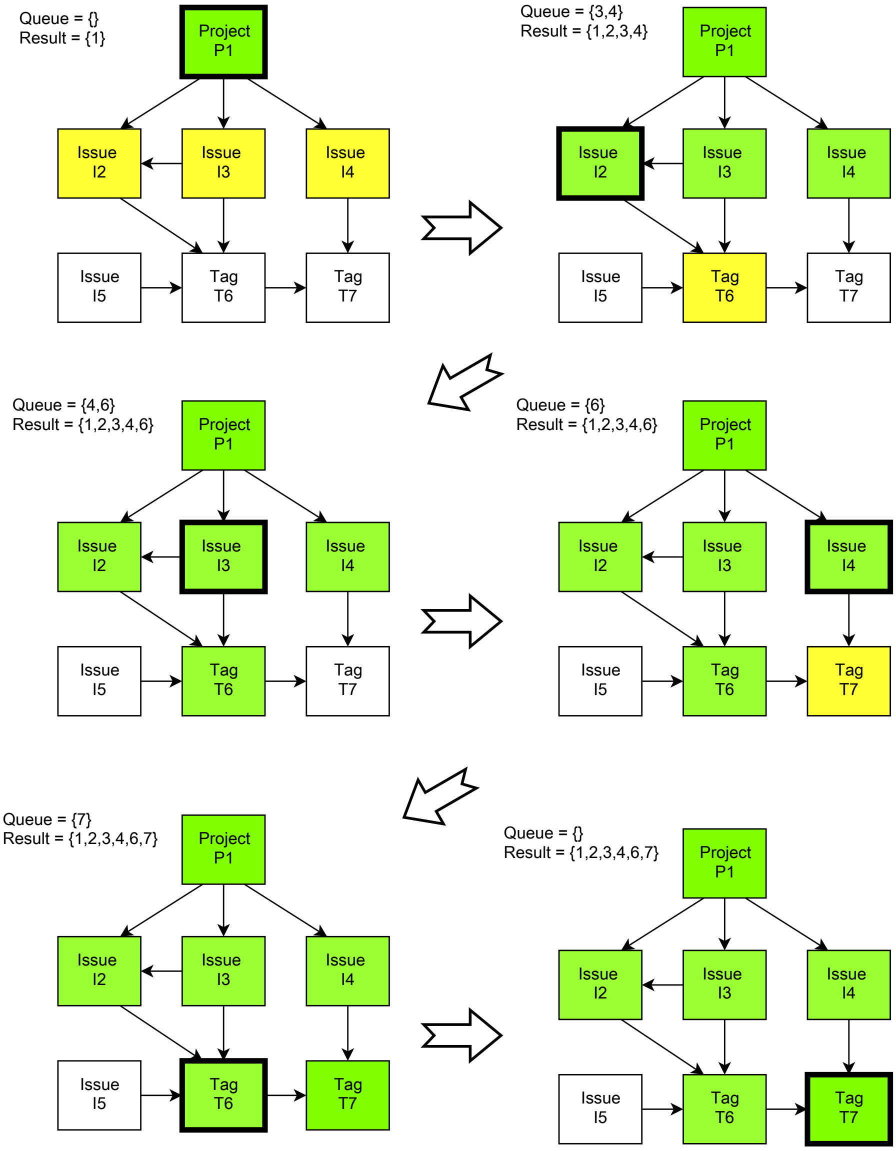
An addition to the synchronization algorithm is access control, customized rights for objects based on the current principal. Figure 7.6 displays an example where the access control is triggered, which restricts the user from creating the object. The integration is to check the specified rules for each modified object but also for each object that is propagated to the mobile application. The difference is that for the creation check a static function is used. This is done because at the moment of creation there are no values that can be used to authenticate the user. Both creations and modifications are checked before applying the updates to prevent bypassing of security rules.

7.1.2 Related Entities

The previous section explained the core of synchronization however, data partitioning is needed to make synchronization scalable for good usability. In the final solution we have chosen to use a version based on the idea used in the motivating example. The idea used in the example is to have Toplevel objects which could be selected and then request all objects that are linked to the object by properties recursively. In Mobl this is simply tracked by the local database, since only objects that are related are stored locally. To improve the synchronization for the mobile devices this calculation is moved to the server. This requires a different approach for the calculation of related entities.

Figure 7.8 shows an example execution of the idea behind the calculation of related entities. A Toplevel object is chosen to start and puts all objects linked by outgoing edges in the result and queue. While the queue is not empty it takes an object and adds the related entities of that object into the queue en result if they are not already discovered.

The serialization in code of this approach is a straightforward transformation of the described steps in the previous paragraph. However, the synchronization framework requires some tweaking to make it usable in the current framework. The implementation of the algorithm is shown in Figure B.8.



green = included entity, yellow = new related entity, bold line = selected object

Figure 7.8: Visual representation of the calculation for related entities

7.1.3 Simple views

The simple views display data through GUI and allows browsing to other objects by clicking on properties. The properties with simple types are easy to display by using standard input fields as displayed in Figure 7.9a. Similar approach could be used for properties with entity type. This would mean that a 'inputField' template has to be generated for each entity and the same for the collection view (Figure 7.9b). Higher abstraction and templates as first class citizens allow to abstract even more in the views. Figure 7.9c shows that the view is given by the parameter in the template. The required view is deducted from the type of the property. This approach has less code and more abstraction, making it easier to reuse within the application. For example, the display could be used to display the objects that are related to the returned errors.

The views conclude the interesting implementation details of the generated framework. The next section will describe the implementation details of the compiler that generates this framework.

```
entity Project {
  title        : String
  description  : String
  size         : Num
  owner        : Person
}
screen viewProject(object : Project) {
  inputfield("title", object.title)
  inputfield("description", object.description)
  inputfield("size", object.size)
}
```

(a) View with simple type properties and model

```
screen viewProject(object : Project) {
  inputfield("title", object.title)
  inputfield("description", object.description)
  inputfield("size", object.size)
  fieldbutton("owner", object.owner)
}
template fieldButton(labeltext : String, object : Person) {
  button(labeltext, onclick = { goto showPerson(object) } )
}
```

(b) Specialized button template for entity types

```
screen viewProject(object : Project) {
  inputfield("title", object.title)
  inputfield("description", object.description)
  inputfield("size", object.size)
  fieldbutton("owner", object.owner, showPerson)
}
template fieldButton(labeltext : String, object : Entity, view : Control<Entity>) {
  button(labeltext, onclick = { goto view(object) } )
}
```

(c) Improved specialized button template for entity types

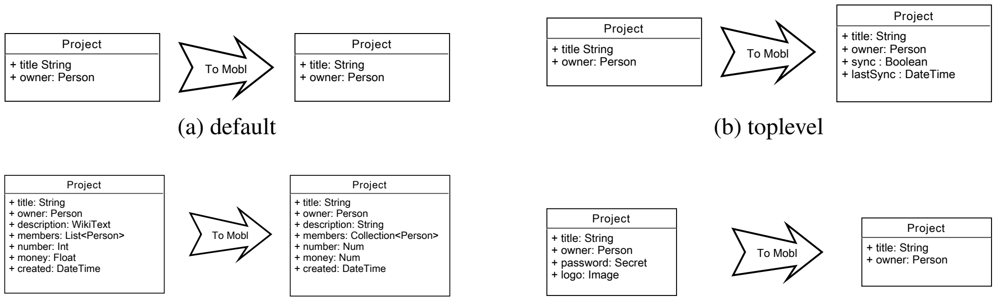
Figure 7.9: Example for implementation of simple views in generated Mobl code

7.2 Compiler Code

The general working of the compiler extension is generation based on templates with application specific parameters. Those templates can already be found in the pseudocode in the previous sections, where several placeholders are used to represent application specific information. A big share of the templates for the framework are generated for every entity within the data model. That means that in general the generation is a map over the list of entity names for a template. Many templates are straightforward and interesting details of the generated code are already discussed in the previous section. This section contains more detailed information where the generation deviates from the straightforward mapping in this case the following parts are included: model-to-model, authentication and access control.

7.2.1 Model-to-Model

The mapping from model to model is in principle pretty simple, for every entity make an entity for Mobl. Then for every property in an entity make a property in the equally named entity. The only thing it needed afterwards, is to map every type from WebDSL to a type in Mobl to complete the properties.



(a) default

(b) toplevel

(c) specialized types

(d) unsupported types

Figure 7.10: Examples for model-to-model transformations in different situations

Figure 7.10 shows four examples for the model-to-model transformation. The first one shows the default situation where the model stays the same. In Figure 7.10b the entity has been selected as toplevel entity, which gives the Mobl entity two extra properties for the functionality of the synchronization framework. The third example shows that the specialized types of WebDSL are represented by the more general type in Mobl entity. For example, the types Email and Wikitext do not exist in Mobl and need a different representation.

The last example shows what happens in the situation of non supported types. The entity remains however, the properties with unsupported types are removed from the model. The main reason for not supported types are listed below:

- **Mapping**: There are types that are not easy to map if there is not a type that represents a similar values. Even if there is a possible mapping it should still make sense and be usable. For example, the File is not supported in Mobl, but could be represented as a string of the actual byte values. This string representation itself does not give a useful representation and is almost impossible to manipulate without invalidating the file.

- **Security**: There might security issues when mapping a type, in case of WebDSL this only applies for the Secret type. It implies that the value is meant to be private and not shared. Even when it would be encrypted it is better to never share those values.

- **Size**: The space on mobile devices is limited and this makes it unwanted to have some values taking all space. File and Image are such types which are in average huge in size compared to other values. In practise some basic types like String and List could also become large. On the other hand, this is normally not the case.

7.2.2 Authentication

The authentication module is an extension on the current authentication system in WebDSL. The mobile devices use device keys instead of user password to authenticate itself on the server. There are some particular aspects that are not found in other generation parts. Figure 7.12 shows an example of the generated authentication module of WebDSL. The code contains many templates that do not require input for generation. The more interesting aspects of the templates is that it adds an entity and extends the principal entity with a property and some help functions that are left out from the example. This is only added when authentication is enabled for the web application as well.

```
entity User {
  username :: String
  password :: Secret
}
principal is User with credentials username, password
```

(a) Example program

```
extend entity User {
  devicesKeys -> Set<DeviceKey>
}
entity DeviceKey {
  description :: String
  key         :: String := id
}
```

(b) Generated authentication module

Figure 7.11: Example of authentication module generation

7.2.3 Access control

The access control on objects is an extension to the current existing rules for pages and templates. A similar implementation is used for objects, the developer can specify three boolean expressions for an entity. The synchronization algorithm uses the generated functions which contains the evaluation of those rules. An example for the generation of those three functions is shown in Figure 7.12. There is a small difference between the three functions, namely the *mayCreate* function is static. There is only one expression for every method, this makes it easy to create a return statement from the expression. The generation of the functions bodies have a fallback true expression in the case there is no rule specified for the specific entity.

This completes the details of the implementation for the general solution. Nevertheless, the implementation also contains interesting implementation work related to the chosen tools. This will be explained in the next section of this chapter.

```
entity Project {
  title       :: String (search)
  description :: WikiText

  synchronization configuration {
    access read: true
    access write: currentPrincipal() == owner
    access create: isLoggedIn()
  }
}
```

(a) Example program

```
extend entity Project {
function mayRead() : Boolean {
  return true;
}
function mayWrite() : Boolean {
  return currentPrincipal() == owner;
}
static function mayCreate() : Boolean {
  return isLoggedIn();
}
```

(b) Generated access control module

Figure 7.12: Example of access control generation

7.3 Technical Difficulties

While this thesis presents an implementation for WebDSL and Mobl, the focus is on a solution that is applicable to other target languages in the domain of web and mobile applications. The implementation has some interesting problems and solutions that are specific to WebDSL and Mobl. This section will describe interesting problems that influenced the final solution. The difficulties encountered can be separated into two parts: limitations encountered by the model to model transformation (Section 7.3.1) and synchronization algorithm (Section 7.3.2).

7.3.1 Model Limitations

Programming languages always contain their own way of specifying data models within the application. Most of those differences are syntactic and therefore can be expressed in other languages as well. In the case of WebDSL and Mobl the syntax is similar to each other. However, the semantics differ since they are designed for different domains. The mobile domain does not require the same complexity, since mobile applications deal with simpler data structures. The restrictions that are encountered within the mapping of the model are: no model hierarchy support, restricted collection support and expensive search indexing.

Inheritance Hierarchy

As experienced in the motivating example Mobl does not support inheritance hierarchy within the data model. In WebDSL application this is often used, which means that it has to be solved to make the synchronization framework usable. The solution used in the motivating example, only support one class of the hierarchy, is not good enough for general solution.

There are several ways to remove the hierarchy out of the data model. Figure 7.13 displays an example of inheritance and the effect of several solutions. The solution used for the motivating example is displayed in Figure 7.13b, which directly shows its shortcoming of losing information. A better solution is to simulate inheritance in a flat data model. There are two solutions to do this, separating (Figure 7.13c) or combining (Figure 7.13d) the classes. Separating the hierarchy is done by deleting all hierarchy and include all the properties of the parents in the class itself. The side-effect is that for every property referring to a superclass it needs to add a property for all the subtypes. The combining of classes into one class is done by collecting all properties in the parent and adding an extra field that contains the actual type of the object. This combining does not require other entities to deal with the modifications.

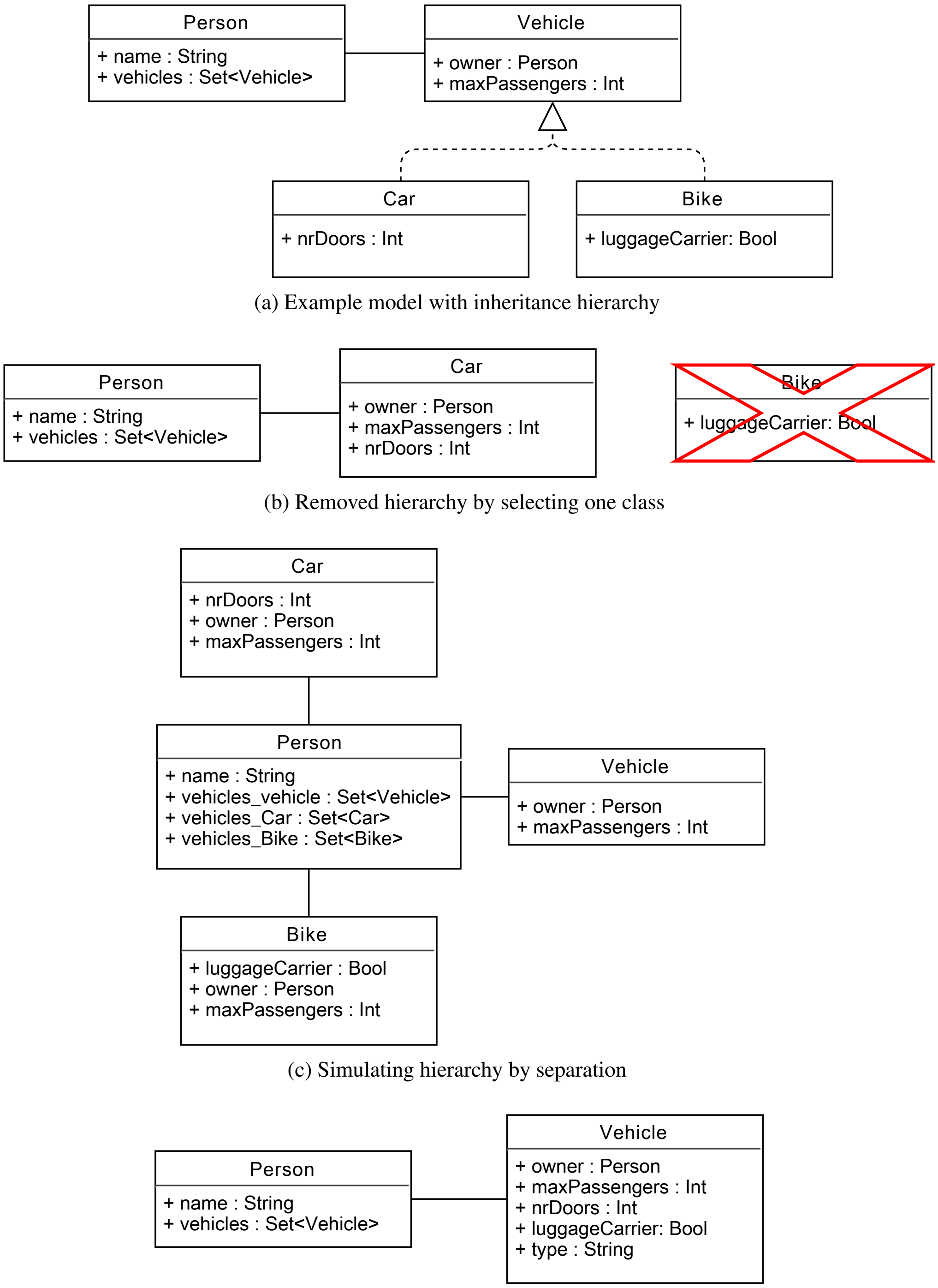
Separation and combining are both good solutions to simulate inheritance hierarchy in Mobl, we have chosen to only use combining. This has mainly to do with the fact that the extra properties on an entity for every child class makes it hard to use, since requiring the actual value of a property referring to such a class must be done by combining the values of multiple properties. The other reason is that the impact is restricted to the entity itself.

The impact on the transformation from model-to-model is limited to remove all sub classes and move all properties to the parent. Additionally, it influences all other WebDSL code that has to deal with the JSON input. This has as effect that many of the generation templates have an extra version with added checks to find out the actual type of the input.

Collections

Collections in WebDSL exists in the form of sets and lists where Mobl only supports the general collection type to represent both forms. In addition, Mobl requires an inverse property for every collection property, where WebDSL allows the annotation instead of forcing it. In Figure 7.14 shows an example of the steps that are used for collections in the model-to-model transformation. The approach can be described by the following steps:

- First find all collection properties that do not have an inverse annotation or the inverse property is excluded with restricted property option.
- Followed by generating an extra property for the entity, which already includes an inverse property itself.
- The last step is to get the stored inverse properties from the compiler and add them to the property. This step is done in the stage that the properties are mapped from WebDSL to Mobl.



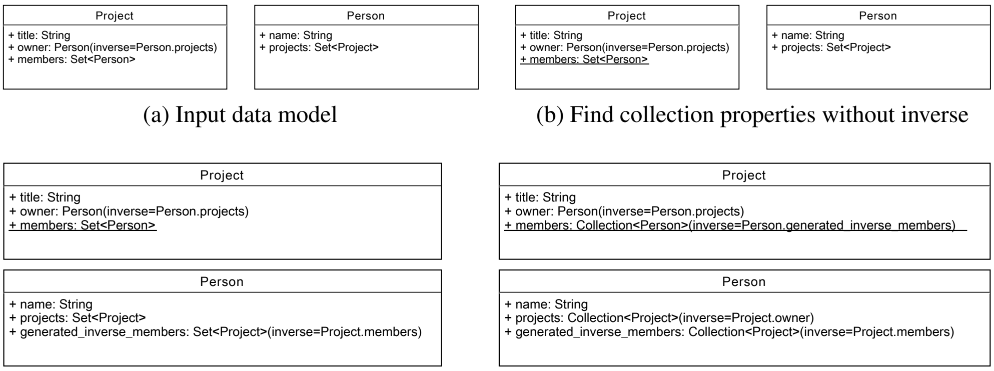
(a) Example model with inheritance hierarchy

(b) Removed hierarchy by selecting one class

(c) Simulating hierarchy by separation

(d) Simulating hierarchy by combination

Figure 7.13: Solutions by example for missing inheritance hierarchy support in data model


Project
+ title: String
+ owner: Person(inverse=Person.projects)
+ members: Set<Person>
Person
+ name: String
+ projects: Set<Project>
Project
+ title: String
+ owner: Person(inverse=Person.projects)
+ members: Set<Person>
Person
+ name: String
+ projects: Set<Project>
Project
+ title: String
+ owner: Person(inverse=Person.projects)
+ members: Set<Person>
Person
+ name: String
+ projects: Set<Project>
+ generated_inverse_members: Set<Project>(inverse=Project.members)
Project
+ title: String
+ owner: Person(inverse=Person.projects)
+ members: Collection<Person>(inverse=Person.generated_inverse_members)
Person
+ name: String
+ projects: Collection<Project>(inverse=Project.owner)
+ generated_inverse_members: Collection<Project>(inverse=Project.members)

(a) Input data model

(b) Find collection properties without inverse

(c) Create extra property

(d) Map to Mobl and include extra inverse annotations

Figure 7.14: Solutions by example for no inheritance hierarchy support in data model

Search

WebDSL and Mobl have the functionality that allows auto indexing of objects. However, we found in Section 5.5 that the indexing of objects in Mobl is significantly slowing down the synchronization. The search is not part of the synchronization and influence the speed. Therefore, we have chosen as default behavior in the transformation is to remove all search annotations and add only search annotation to the name property of TopLevelEntities. The added annotations are required to make it easy to find and select objects. This approach is not optimal and could be improved by adding extra syntax that allows the developer to specify search annotations for the Mobl entities.

7.3.2 Synchronization Limitations

The synchronization algorithm requires control over objects and database. This is often not within the capability of the normal code and should be implemented within the compiler. The following three features had to be implemented within the compiler to support the synchronization solution: Timestamp versioning on objects, automatically setting of flags and control on the rollback of transactions.

Timestamp Versioning

The motivating example uses the version numbers of the objects to identify updates. This is done automatically for entity objects that are persisted. However, to improve the synchronization algorithm it is better to have timestamps instead of version numbers. This change requires a hook in the current versioning system and uses 2 extra properties on all entities: created and modified. The created property is only set for new objects, while the modified property is set when at least one of the properties has changed.

Auto flagging

Mobl has an internal persisting framework, which tracks changes to persist them to the database. Non of that information is persisted or directly accessible. The compiler is extended such that each entity has two extra properties: created and dirty, which are boolean flags to track status of the objects. In addition, the persistence framework had to be adapted to automatically set those flags on creation or changes of the property. There are some corner cases which need to be handled. The first intervention that needs to be excluded, is that when changing the flag it should not be seen as modification of the object. This needs to be prevented because when setting the flag it would trigger itself to be set creating an endless loop. The second problem is that it cannot distinguish differences from the synchronization and the changed made by the application. Therefore, an additional flag in the system framework is added which disables the tracking on synchronization. This means that changes and creations during synchronization are not discovered.

Rollback of Transactions

To resolve inconsistencies in the database rollback functionality is used, which is part of WebDSL functionality. The original compiler approach is an automatic validation and possible rollback at the end of each request. Synchronization can be seen as a batch of changes. For batches it is unwanted to have validation and rollback on the whole set of changes, since one incorrect update would revert all changes in the batch. The second problem of the original approach is that it only uses the validation rules on the model, other inconsistencies cannot be used to trigger a rollback.

We added functionality to the compiler to allow to start and rollback transactions manually. Those functions allow to split up the batch in separate transactions and only fixes the changes with inconsistencies. There is a problem with splitting up the transactions for creating objects. The objects have to be created before the actual values are applied. This variation forces the use of different transactions, which makes it impossible to roll back only a single object. We have chosen to do a rollback for all new objects. Although the mobile application can retry the propagation with changes to solve the conflicts.

7.4 Summary

This chapter shows implementation details of this thesis solution separated into three parts: The framework delivered by the generated code, The compiler extension that generates the framework and difficulties that are specific to the target languages WebDSL and Mobl.

The generated code basis started from the motivating example. Changes have been applied to abstract and improve on shortcomings of the motivating example. The biggest differences to the code can be found in synchronization, data partitioning and view generation. The synchronization has changed by using a different approach that is compatible with the new data partitioning. The synchronization functions have some extra responsibilities as is described in Section 7.1.1. The data partitioning is needed to have an approach that allows users to select parts of the data to retrieve without too much effort. A few entities are used to specify the partitioning representation. The partition is calculated by related entities, this calculation is shown by an example in Figure 7.8. The view generation is added to have an easy data browser which allows a quick start when developing the Mobl application and is explained in Section 7.1.3.

The compiler code is basically template generation that is often based on the list of entities. However, some of the generation parts are more complicated. This is the case for the following parts: model-to-model, Mappers, Related entities, authentication and Access control. The generation of those components are explained in Section 7.2.

The last part of this chapter includes some details that are specific to the target languages of this thesis. There are two main parts where we encountered difficulties:

- **model-to-model transformation** is required for the synchronization framework to work. The general problem is that not all the properties of the data model can be expressed in Mobl. The following difficulties were encountered and explained in Section 7.3.1: no support for inheritance hierarchy in data model, collection properties require inverse annotations, expensive indexing of objects and restricted set of types.

- **synchronization algorithm** requires some control on the level of the persistence framework and database, but this abstracted by the language. The following modifications where made in the compiler code for a functional synchronization framework: Automatic timestamp for modifications, automatic setting of flags for Mobl on modifications and manual control on transactions to allow rollback of single inconsistencies.

This and previous chapter describes the solution of this thesis and its implementation. The next chapter will evaluate the solution to check if it satisfy the requirements and discover possible weaknesses.

Chapter 8

Evaluation

The two previous chapters introduced a generative approach for a synchronization framework between web and mobile applications. Chapter 4 states a list of requirements for functionality and non functional requirements that the framework should fulfill. This chapter uses this list of requirements as guideline to evaluate the solution. The chapter is separated into evaluation of functional (Section 8.1) and non-functional (Section 8.2) requirements. To validate the scalability of the framework it includes an experiment (Section 8.3 and Section 8.4).

8.1 Evaluation of Functional Aspects

The functional requirements are objective definitions of functionality that should be included and working in the product. The evaluation of those requirements can be done by objectively checking the list. The functional requirements for this solution are summarized in Table 4.1. In a simple interpretation it could be stated that the product can only be delivered if all the functional requirements are fulfilled. The previous chapter describes the implementation of the solution, which can be used to review the requirements. In addition to that, the product has been tested at development time by adapting the model and reviewing the results and differences between each version. Next to manual testing there are some integration tests on the webservices to check the functionality of the synchronization framework. This section will have a quick review on the main features: synchronization, interface, data restriction and code generation.

Synchronization
The general steps that in synchronization can be described as: identification, propagation, and detection and resolution of inconsistencies on updates. The first two steps: identification and propagation are required minimum for a working synchronization approach. Section 7.1.1 explains in detail which algorithms are implemented for synchronization within the solution. The implementation covers the three steps fully.

To propagate updates it is required that the objects are transformed into valid and serialized instances on both sides. In the case of Mobl most of the functionality is built in already, only a mapping to the correct function calls is added. WebDSL lacks this functionality and therefore needs generated mappers based on the data model. The functionality of the generation of those mappers is explained in Section 6.2.1.

Interface
The current implementation has four different interfaces:

- A small set of additions to WebDSL syntax to adapt the code generation (Section 6.4.2).
- Webservices that allow communication with mobile devices and other remote applications (Section 6.4.1).
- Mobl integration functions and views to insert the synchronization into the application.
- A generated data browser for the Mobl application (Section 7.1.3).

Those four cover the interfaces that are specified in the requirements and gives possibilities for all user groups to interact with the solution where required.

Data Restrictions
The underlying problem of data restriction is that in most cases it is not possible (or wanted) to have all data available for the mobile applications. The requirements state three approaches to reduce the amount of data by restrictions. The solution implements all three with following parts of the implementation:

- Restricting properties on data model for simplification.
- Access control on objects to define user related restrictions of synchronized objects (Section 7.2.3).
- Selectable data synchronization implemented as partitions with TopLevel entities and related entity relation (Section 7.1.2).

Code Generation
The requirements state that the solution requires a list of elements that should be generated. This might not satisfy the goal of this thesis, which is a complete synchronization framework. The fact that small amounts of additional code are needed after generation shows the completeness of the generated code. The additional code that is required for mobile application is limited to the import and calling of synchronization framework. This only delivers a simple application with an available data browser and the possibility to synchronize data.

This completes the list of functional evaluation of the solution. However, the evaluation is not complete. The requirements also contain non-functional demands. The next part of this chapter will evaluate those requirements.

8.2 Evaluation of non-Functional Aspects

The non-functional requirements are complicated to define in strict rules. Instead, they are often described in how the system should be functioning when it encounters certain (abnormal) conditions. This softer approach of stating requirements has a disadvantage on validation. A simple checklist does not fit this purpose a more persuasive text with valid arguments is a different solution. Experiment results can be used to support the arguments and make it more objective. This section will argue on each of the topics that are summarized in Table 4.2 and how the tool fulfills those requirements.

Usability

The usability requirements state two general focus points for usability: simple and minimal. There are different interfaces for each of the user groups: application developer, mobile application user and remote application (developer).

The application developer has to deal with two interfaces: The additional syntax that is required for the generation and the generated Mobl code which has to be included within the application. The minimal syntax that is required is a simple definition that states the TopLevel entity. Additionally, it is possible to specify some restricted properties and access control rules. This is as minimal as it can be, since this information cannot be extracted from the normal application code and is required for a functional framework. Defining some boolean expression and references should not be complex for a developer.

The integration within the application could be easy and minimalistic by just including two function calls. The functions allow some parameters for tweaking. There is a possibility that other behavior or interaction is wanted for an application. This has as disadvantage that it loses the minimalistic character, when the developer chooses to change the functions. There are additional abstractions in the generated code on different levels of the synchronization algorithm for easy reuse of parts. Something similar is applicable for the generated WebDSL code. WebDSL allows overriding of functions, which makes it possible to tweak some of the functions like the mappers. This is not an optimal interface, but should only be needed exceptionally.

The synchronization is a process that should not require much interaction with the user. The current approach in this tool has three points of interaction: selecting the partitions, triggering the synchronization and error handling on feedback of synchronization steps. Those points can be implemented to work without interaction to the user, but this requires specialized code based on application behavior. Meaning that the current implementation gives a minimalistic and simple interface that is applicable for all applications, but optimal result can only be achieved by implementation of the application developer.

The remote application can use the webservices that are generated as part of the synchronization framework. Those services are meant for synchronization with mobile applications and require some information from the remote application. The design is optimized for the communication in the framework. A second focus was to keep it understandable and usable for other applications. It fails to fully satisfy the requirement on simplicity, but that is due to the trade-off on other (more) important requirements.

In addition to the minimal and simple requirements, the framework should also take into account the limitations of mobile devices. The incremental approach of updates should reduce the bandwidth and computation costs. To achieve low cost for mobile device the computation has been moved as much as possible to the server. The mobile client only has to deal with mapping and query of local data.

Scalability

One of the bigger requirements for a synchronization solution between web and mobile clients is the scalability factor. The nature of web applications is that it can handle numerous users at the same time. For the solution this meant that it should be possible for all users to have mobile devices that can synchronize with the server. The impact on the server can be separated in two dimensions that of CPU time and disk space.

The normal usage of web applications deals with many small requests, since every page request requires some small amount of computation. The synchronization requires more computation on request. On the other hand, the number of request is significantly smaller. This is due to the fact that the synchronization only consist of a limited number of webservice calls and does not have to be executed frequently. It only requires a single thread on a server for a device and is limited by the speed of the mobile device (Section 8.3.3) to process the results, which means a high amount of idle time which can be used to handle other requests.

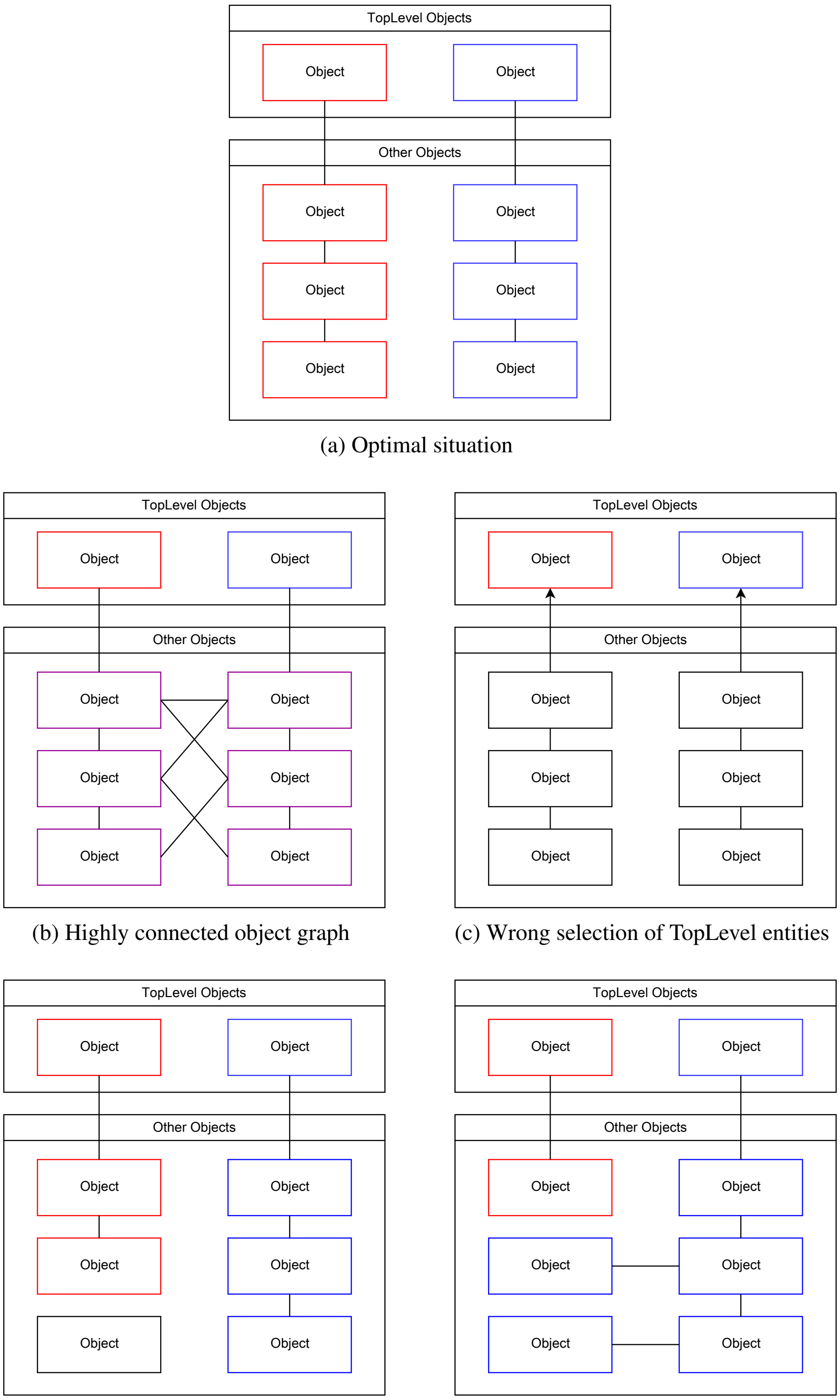
The impact on space is even less constraining, the server does not keep any extra information for synchronization algorithm. Therefore, extra devices will not increase the database size and cannot constrain the scalability.

In addition to the scalability found in the number of clients, there is also the size of the application that could affect the scalability of the synchronization framework. The impact of increasing size in data (model) is evaluated by the experiment described in Section 8.3.

Applicability

The applicability requirement is that the solution should be able to generate a synchronization framework for every WebDSL application. In the strict case it is true that every application that has selected a TopLevel entity is able to generate a functioning synchronization framework, since the generation deals with all possible inputs of models. There are some remarks on this notion because not all types of WebDSL are (fully) supported and some data models might not have a perfect performance in the framework.

The non supported types is an issue that is derived from the constraints of the target language and mobile domain and is discussed in Section 7.3.1. The performance penalty in some models has to do with the calculation of related entities. The wrong selection of TopLevel entities or fully (non) connected object graphs show the weakness of the partitioning approach. Figure 8.1 displays situations that cover those weaknesses of the partitioning approach and are discussed in the following paragraphs.



(a) Optimal situation

(b) Highly connected object graph

(c) Wrong selection of TopLevel entities

(d) Unreachable object

(e) Unbalanced object graph

Figure 8.1: Visual representation of object graphs representing weaknesses of data partitioning approach

The optimal object graph for the partitioning solution used in this thesis is displayed in Figure 8.1a. The first unwanted situation is that the graph is highly connected, which most likely delivers the situation that there is no partitioning anymore (Figure 8.1b). A possible solution is to remove the edges that are crossing the partition, which is possible in the generation by adding restricting properties.

Selection of TopLevel entities is a manual task. When the developer does not really have good insight the data model and the object graph it is possible that a situation appears as showed in Figure 8.1c. The solution is often quite simple by selecting other TopLevel entity. However, this requires some extra effort from the developer to inspect the data model and object graph.

There are two other situations that might be encountered and possibly deliver unwanted behavior. A possibility is that not all the objects are (indirectly) connected to a TopLevel object (Figure 8.1d). The question is if this delivers wanted or unwanted behavior. If it is an unused object, the behavior is as it should be. When it is an isolated entity type, the choice can be made to make it a TopLevel entity which resolves the problem as well.

The last situation is something which most likely will happen in all cases, namely an unbalanced partitioning (Figure 8.1e). It is questionable if this is problematic or unwanted. This depends on the situation, for example in YellowGrass the behavior is related to the fact that the size of the projects are diverse, which creates uneven partitioning. In this case, it is not incorrect since only related entities to the project are retrieved. However, when the separation by related entities is not as directly related to the application it is most likely that it will retrieve unwanted objects.

Adaptability

Adaptability is a double-edged sword for most code generation approaches [36]. The goal is to have a complete solution that is directly usable in all cases. However, the reality often shows that there are cases that are not covered by the tool. A way to solve this is to allow adaptability or insertion in the generated code. This approach has as weakness that there is no control on this additional code and therefore cannot be checked on correctness. The requirements for this thesis are similar: The focus should be on a complete covering implementation which works for all inputs, but there must be possibility to adapt the generated code for tweaking certain aspects.

The generated code can be separated into two parts: core synchronization algorithm and helper functions. The core should never be adapted for application specific features, since those are requested through helper functions. The helper functions on the other hand can be adapted to deliver different input to the core algorithm. There are two ways to adapt those functions. One is the interface delivered by additional syntax in WebDSL(Section 6.4.2). The second way of tweaking is to override or extend functions in WebDSL. This allows to change any function in the code base including the generate code. This should only be done on helper functions like mapper or related entities and only when it is not possible to tweak with the DSL extension.

Security and Robustness

The security and robustness are two topics in software projects that are always problematic. Big and important systems often require perfect scores on these topics. Although reality shows that even those systems contain flaws. In the case of the synchronization framework, robustness is very easy to achieve, since it does not interact with other processes in the system and cannot change the working of the web application. The only thing it might do is change data that changes the working of some other aspects of the application. However, this is more a security issue then a robustness flaw.

Security is a much harder problem for a synchronization framework, since there is a possibility that (untrustworthy) users can change data directly in the database. The synchronization framework has three checks added for security reasons:

- Access control to limit the data access based on the user (Section 7.2.3).
- Validation on the objects before persisting them to the database (Section 7.1.1).
- Additional input checks on the objects protecting against incorrect input parameters.

The biggest disadvantage is that the security of the synchronization is bounded by the effort of the developer to specify correct and complete access control rules and validation expressions. Without those there is a high chance that a user can abuse the weaknesses to steal protected data or corrupt the database.

8.3 Experiment in the Scalability of the Synchronization Framework

Web applications have to deal with scalability because of database size and user amounts. This means that the synchronization framework must be able to deal with those amounts as well. The solution tries to reduce the amount by partitioning and restricting the data. However, it is not clear what the influence is when changing the variables for size and complexity of data.

The experiment uses a similar setup as used for the experiment in the motivating example (Section 5.5.2). Containing a laptop with tomcat instance as web server and a chrome browser to simulate the mobile devices. Each test has its own application with specific goals and will be explained separately. Each of the experiments consists of two parts: A full synchronization that returns all the full objects in the partition and a second synchronization which returns no objects since nothing has changed in between. The last variant is used to test the overhead of the algorithm and find out if the bottleneck is on the server or client side. This als shows the advantage of using an incremental approach. The non clean update should should be in same range with a full synchronization that has the same number of objects. This is because the algorithm is not significantly different in those cases, only a higher overhead which is displayed by the second synchronization.

The goal of the experiment is to test scalability in size of the data. There are multiple parameters that influence the size and complexity of the database. The parameters that could influence the data are separated into objects (Section 8.3.1) and entity (Section 8.3.2) specific parameters.

The second part of the experiment, testing the framework on existing web applications is described in Section 8.4.

8.3.1 Object Level Scalability

On object level there are three ways to influence the complexity and size of data: The number of objects, the size of objects and the number of edges. Each of those parameters are tested with different sizes to test if there is a relation between size and execution time. The second step is to determine how big the impact is and whether this is problematic for scalability.

The first experiment uses a simple model containing 6 entities as displayed in Figure 8.2. Increasing the number of objects of *A*, which are all linked to the TopLevel object, gives an increase of 5 extra objects, namely for every entity an extra object in the total partition.

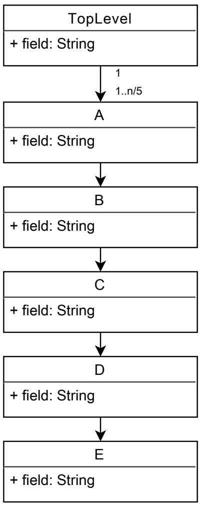
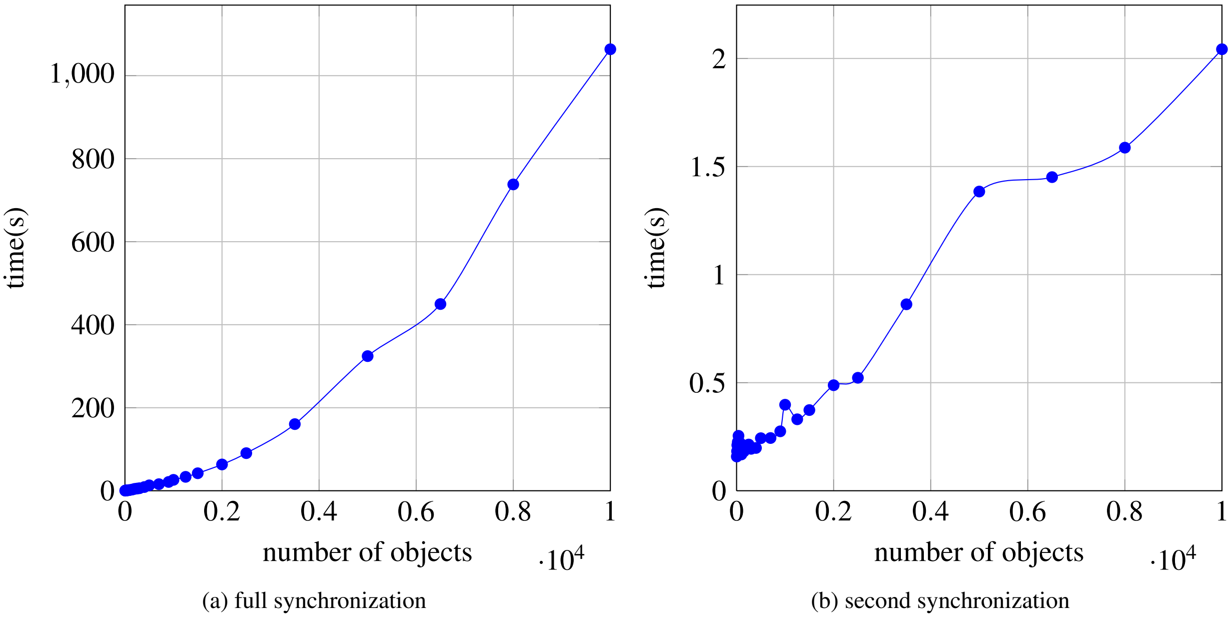


Figure 8.2: Class diagram of linear model for the experiment with increasing number of objects

Figure 8.3 shows the results for the experiment of increasing number of objects in a partition. The line increases more than linear, but still polynomial with the number of objects transferred. The second synchronization shows a linear increase although not as steep as the total synchronization. Secondly, it shows that the overhead of the synchronization algorithm is not so high. In the case of 10000 objects the overhead in the algorithm is smaller than 2‰ and seems to point to a bottleneck on the client side.



(a) full synchronization

(b) second synchronization

Figure 8.3: Results of scalability experiment based on increasing number of objects in partition

The next experiment is increasing the size of the objects. The previous model makes it hard to increase all the objects to a wanted format. We simplified the model to include the TopLevel with a single entity that contains more properties (Figure 8.4). The extra properties are inserted to make it easier to scale up the size of the objects. The experiment has a small adjustment, instead of using the total synchronization time it only looks at the synchronization of the objects of type A.

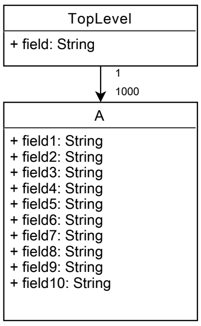
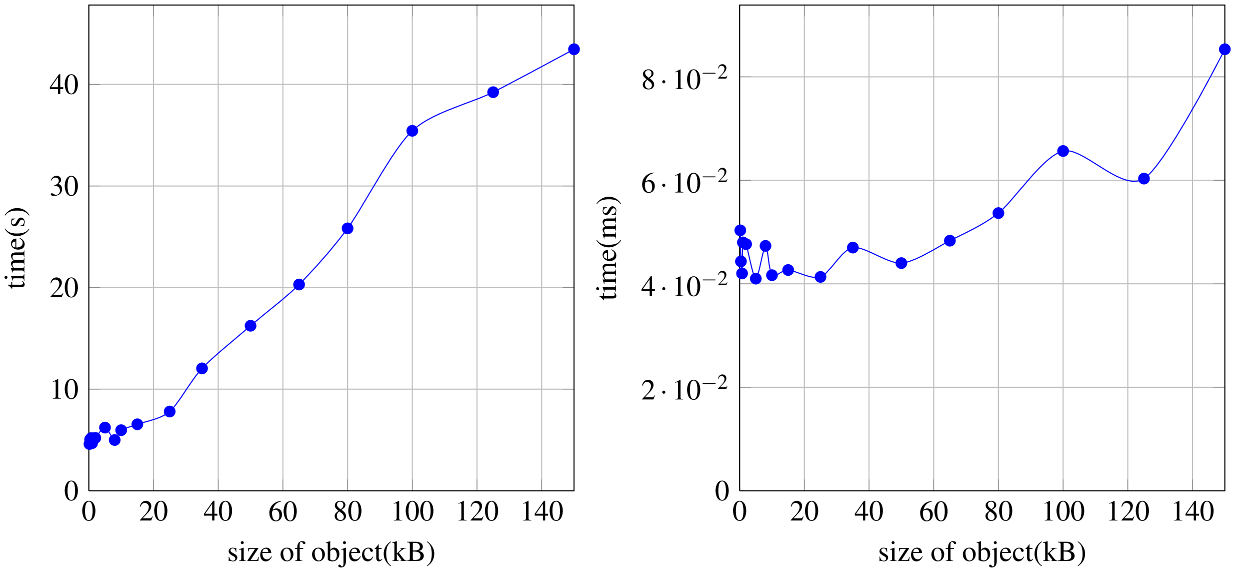


Figure 8.4: Class diagram for the experiment with increasing size of objects

The results of this experiment (Figure 8.5) seem to display a linear increase when the objects become have a significant size. It should be taken into account that average size of object bigger than a few kB, could become problematic for the database size limitations on mobile devices. The effect in the second synchronization do not display a significant relation in the smaller values, however the bigger object sizes show a linear increase in time. It is within expectations that the size of objects in kB should



(a) full synchronization

(b) second incremental synchronization

Figure 8.5: Results of scalability experiment based on increasing size for each of the 1000 objects

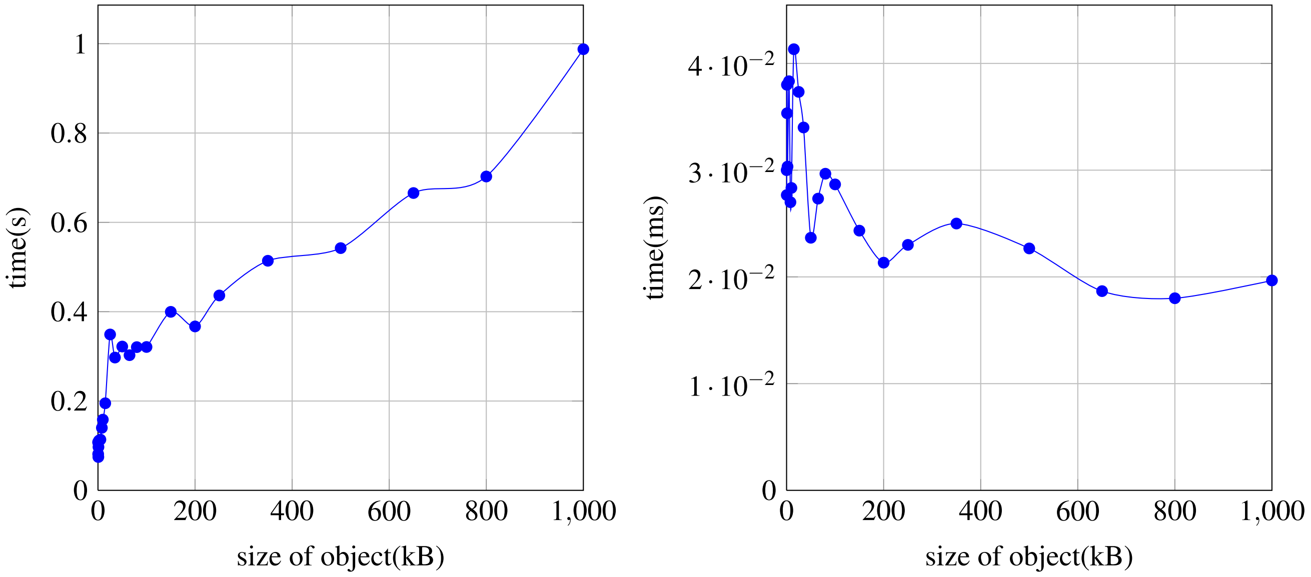
only influence the server when loading the object and mapping the object. The second synchronization does not use any mapping, which means that the increase is due to the loading of the bigger objects.

The increase of time to more than 40 second on 150 kB object size is more worrying, since there is a chance that bigger objects are used in the system. Section 8.4.1 tells more about the size of objects in two existing web applications. This data supports the assumption that most objects are small and that there is a verry small amount of objects wich is in the range of 10 to 1000 kB. To test the impact of increasing the size of a few objects. The experiment is repeated with only 10 objects instead of 1000.

The results show that there is still a linear increase on the size in kB for the objects. However, even at 1000 kB for each object the increase in time is less than a second (Figure 8.6). Which is more likely scenario for a normal system and is better scalable than a high average for the object size.

The last parameter in the object level experiments is the number of edges. An edge in the object graph is a reference from an object to another object by property value. Changing the number of edges cannot be reduced lower than the number of objects, since the partition algorithm requires at least one incoming edge for every object. So instead we test by increasing the number of incoming edges for each object. Figure 8.7 shows adapted model for this experiment. The edges are generated and linked to random objects instead of having copies of each edge.

The impact of the increasing number of edges looks like a less than linear relation between number of edges and synchronization time required as shown in Figure 8.8. The relation in the second synchronization seems to be linear, which makes sense, since the edged are used for the calculation of the partition.



(a) full synchronization

(b) second incremental synchronization

Figure 8.6: Results of scalability experiment based on increasing size for each of the 10 objects

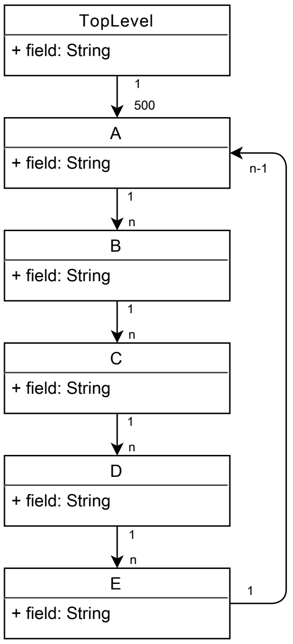
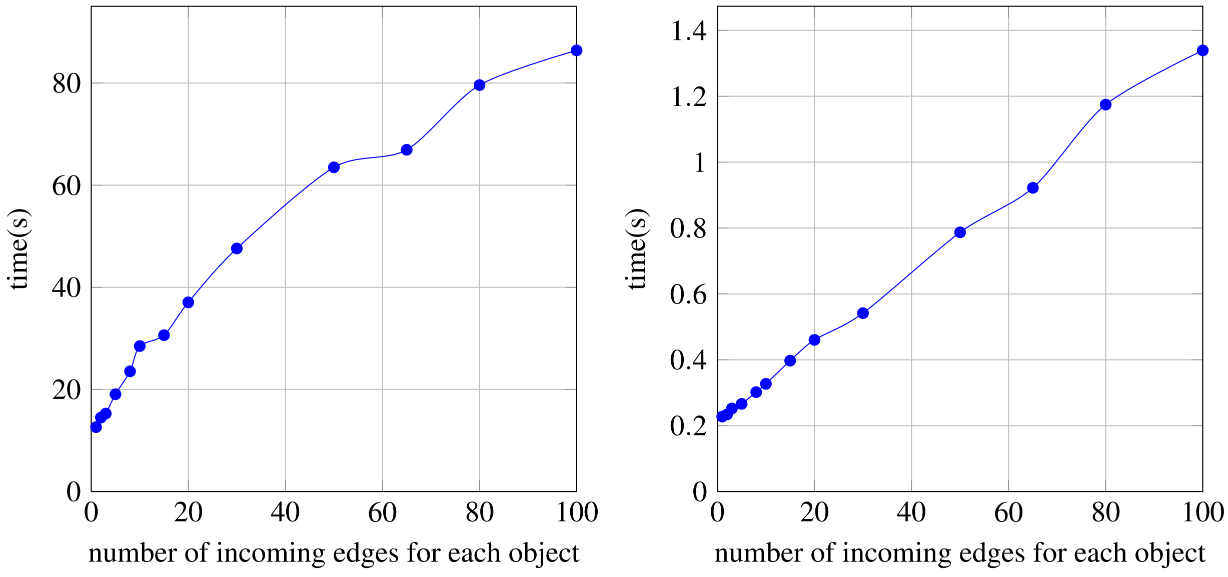


Figure 8.7: Class diagram for the experiment with increasing number of incoming edges for each object



(a) full synchronization

(b) second incremental synchronization

Figure 8.8: Results of scalability experiment based on increasing incoming references for each of the 500 objects

8.3.2 Entity Level Scalability

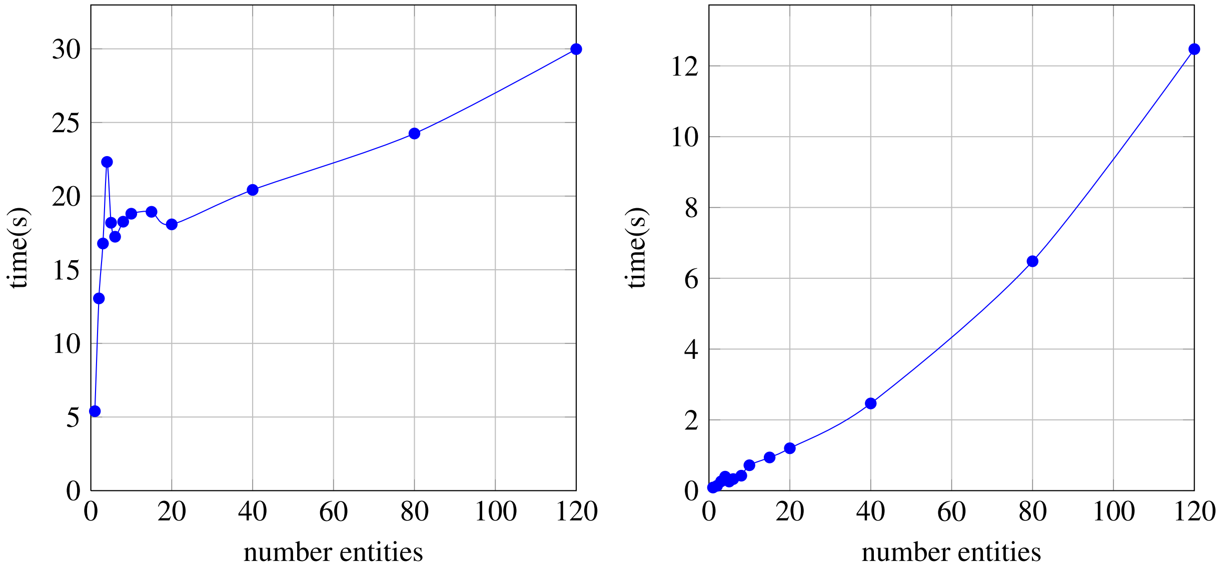
The data model itself is not per definition changing the size of the data. Nevertheless, the increasing complexity of the model might be a side-effect of increasing functionality of the application. Another reason for testing the complexity of the model is that the code generation depends on entities and properties, which definitely means that more code has to be executed when the data model increases in size. The two parameters that mark the increase of the data model are the number of entities and properties.

In the first experiment the number of entities is tested. The data model is changing for every input, but the basic model is similar to the one in Figure 8.2. The variation can be found in the length of the chain for the number of entities. To reduce the influence of the other factors the number of objects and edges are kept the same for every execution.

The results (Figure 8.9) show a big increase for the first additional entities. After a more stable part of the following 15 entities, a linear relation seems to be raising. In comparison to the previous results there is a lot more overhead in synchronization algorithm. Comparing the two synchronization times it seems that biggest part from the increase in time of the full synchronization is caused by the increase in overhead.

The other test on data model level is that of the number of properties. A customized variant of Figure 8.4 is used to get a good and controllable situation for the experiment. The values in the properties are reduced in length to counter the influence of the increasing object size due to additional properties.

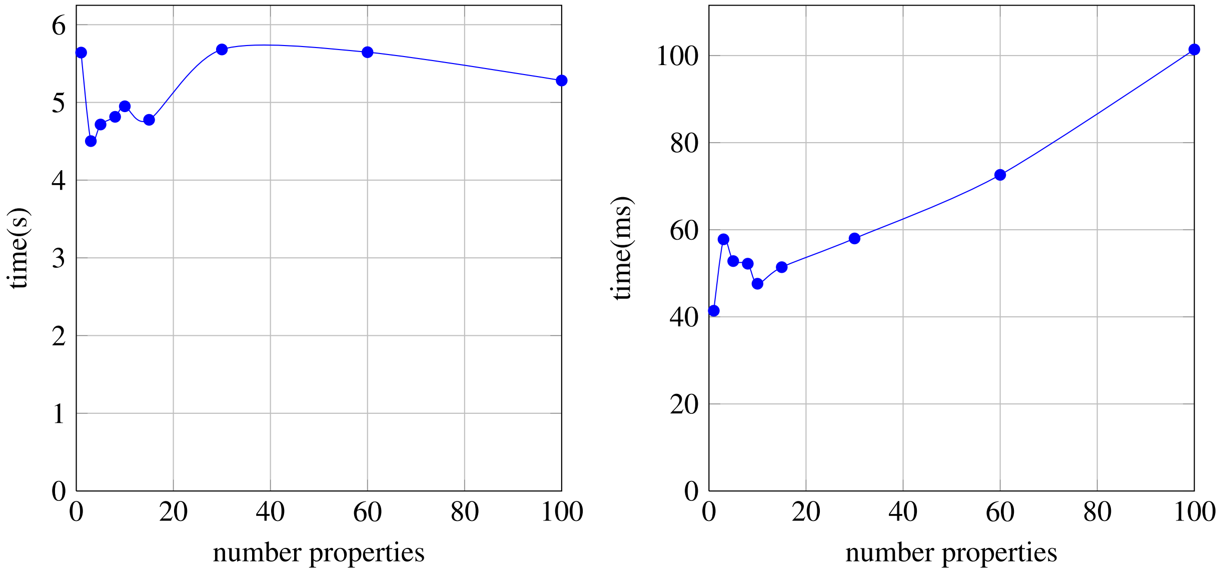
The result of this experiment are displayed in Figure 8.10. The total synchronization results show some fluctuations. However, there is a small or even no effect on time for additional properties. The second graph shows a linear increase for extra properties. The additional price of 50 milliseconds for 100 properties seems to be very little. Certainly when you take into account that 100 properties on an entity is abnormal amount.



(a) full synchronization

(b) second incremental synchronization

Figure 8.9: Results of scalability experiment based on increasing number of entities in data model with 720 objects



(a) full synchronization

(b) second incremental synchronization

Figure 8.10: Results of scalability experiment based on increasing the number of properties for an entity with 1000 objects of constant size

Figure 8.9a and Figure 8.10a show a few fluctuations that are outside the expected range. The values are averages of several runs to counter fluctuations in measure times. However, the stability of execution time on the browser is not as stable as in a normal experimental setup. This is moste likely the cause for the fluctuations in the results of both diagrams.

8.3.3 Discussion

The results of the experiment show in the worst case a linear relation for increasing data size influenced by the following parameters: number of object, size of object, number of edges and number of entities in the data model. A linear relation between input size and execution time could be interpreted as scalable process. An additional factor for is that the experiment has a direct connection and neglects the value of data transfer. This will increase the time for synchronization, although sending data is a linear process bound to the amount of data that is transferred. The additional price for data communication will not make the solution less scalable.

The more negative part of the result is that the synchronization itself is not cheap, the results show high synchronization times for increasing number of edges and objects. Certainly the 1000 objects is within the range of normal amounts, but that already takes more than 25 seconds. Luckily the results of second synchronizations results in times that are far better and within amounts that users will accept.

The difference shows that most of the time is on the client side and has to do with persisting the changes to the local database. This is not something that can be offloaded to the server, which keeps it problematic that it cannot be improved on that factor. This most likely tells us that the implementation of Mobl's persistence framework is expensive and the bottleneck for the synchronization.

The numbers do not directly imply failure for the solution. While it is true that the full synchronization is expensive, it is only required once after that the time will be reduced significantly to only retrieve new and changed objects. Additionally, the synchronization can be done in the background, which allows the application (partly) to be used while synchronizing. Still improvements on the time are preferable for a successful synchronization framework.

The next section show continues the experiment on existing web applications, to validate if the results of this section is in line with the framework applied to real applications.

8.4 Experiment with Existing Web Applications

The experiment in the previous section was focused on testing the influence of the input parameters of the framework with a simple application. This section uses existing applications and applies the framework to those applications. The experiment tests the applicability and includes testing the scalability of the framework on bigger databases. The applicability can be separated into: Does the solution generate a correct working framework. The second part is; can we define usable partitioning for the users so that the framework is usable.

This section starts with some insight in the size of the data of the existing applications. We revisit the web application of the motivating example YellowGrass (Section 8.4.2). This gives the possibility to compare the final solution to the manual implemented example. The second application is Researchr[1], which is currently the largest WebDSL application.

[1] `http://researchr.org`

8.4.1 Object and Database Sizes

In the previous part of the evaluation we already tested on scalability of the input factors. This section states a couple of numbers on the two web applications. Those numbers should give better insight in the size of the current applications and gives some comparison material for the previous results.

Table 8.11 displays the average, maximum size and number of objects of both applications. The total database of YellowGrass has around 10 thousand objects, which is the same number as the maximum used in Figure 8.3. The Researchr database is much bigger with 25 million objects. The average object size of YellowGrass is bigger, but both are around 0.5 kB. Researchr contains the biggest object with a size of slightly more than 600 kB.

Figure 8.12 and Figure 8.13 display the distribution of the object sizes in the databases. We observe that both applications have the biggest share of objects in the first 5 kB. Less than 1% of the objects are found in the rest of the range. In general, the distribution of object sizes of both applications are similar.

Application	**Objects**	**Average Object Size(B)**	**Maximum Object Size(kB)**
YellowGrass	10399	627	106
Researchr*	25178824	467	683

* = the average and maximum are based on 2% random objects from the the database

Table 8.11: Numbers on the data in the databases from the existing applications YellowGrass and Researchr

The basic set of calculations for this section became complex when we scaled the problem up from YellowGrass to Researchr. The running time of those calculations for YellowGrass is less than a minute. Estimated time of those calculations on a server is more than a week. After more analysis, we found that the time was mostly spend on database queries. Some of the objects took more than 10 seconds to load. Therefore, we used 2% random objects of the Researchr database to calculate those numbers.

8.4.2 YellowGrass

The motivating example is developed on YellowGrass, which is explained in Section 5.2. The goal of testing the framework on this application is to test if the framework is usable and how it performs compared to the manual implementation. The experiment tests the synchronization based on size of the objects in the partitions. This contains a first full and a second incremental synchronization.

The first part of the experiment used the number of objects in a partition as size. Figure 8.14 shows that there is a linear increasing line for both synchronizations. The numbers of the full synchronization are in line with Figure 8.3. Comparing both second synchronizations show that the ones of YellowGrass are more than a second slower. The results based on the size of the objects instead of the number of objects supports those results, as is displayed in Figure 8.15

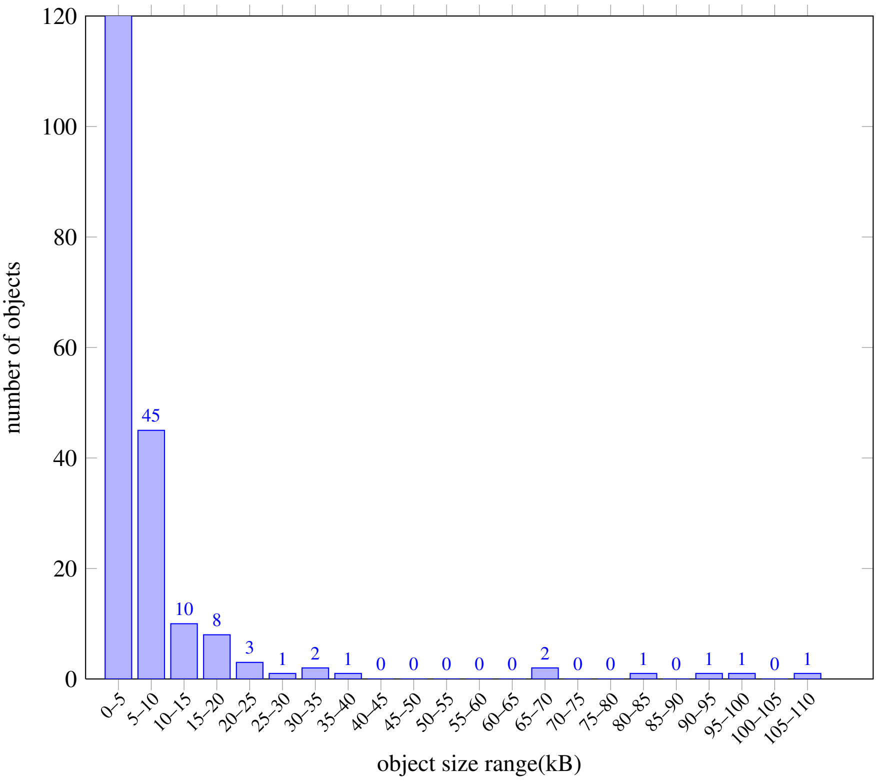


Figure 8.12: Distribution of the object sizes in the database from the existing application YellowGrass

Generated Framework vs. Motivating Example

YellowGrass is used as basis for the motivating example and the framework evolved out of that result. Evaluating this framework with YellowGrass gives the possibility to compare the generated solution to the manual implementation. Where the framework improved on some of the shortcomings of the manual implementation, it is also more general than the motivating example which could influence the performance as well. Figure 8.16 displays the synchronization times for the corresponding partitions. The smaller partitions seem to be faster in motivating example, while the bigger are up to 30 seconds faster than the motivating example. The second incremental synchronization for the motivating example is only covered by Figure 5.17, which represents a medium sized partition. Comparing that to the results of Figure 8.17b, can conclude that the motivating example is faster in the second synchronizations. Combining this with the other comparison, tells us that the generated framework has more overhead.

Some notes have to be placed on the comparison because there are several differences between both versions. The motivating example has a simpler model, which included less properties and less objects for the same partition. On the other hand, the motivating example has extra functionality for the roadmap, which is not included in the generated framework. There has also been some development on the WebDSL compiler, which might influence the performance of the web application. However, in the experiment of the motivating example we already encountered that the time spend on the server side, for bigger partitions, is minimal compared to time spend on the mobile device.

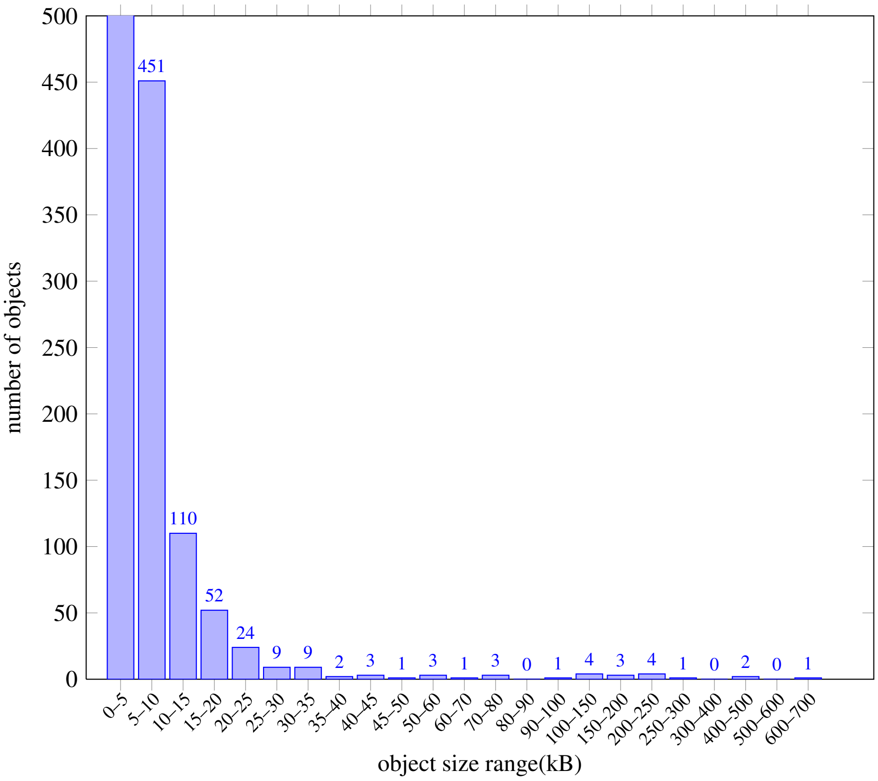


Figure 8.13: Distribution of the object sizes from 2% random objects in the database from the existing application Researchr

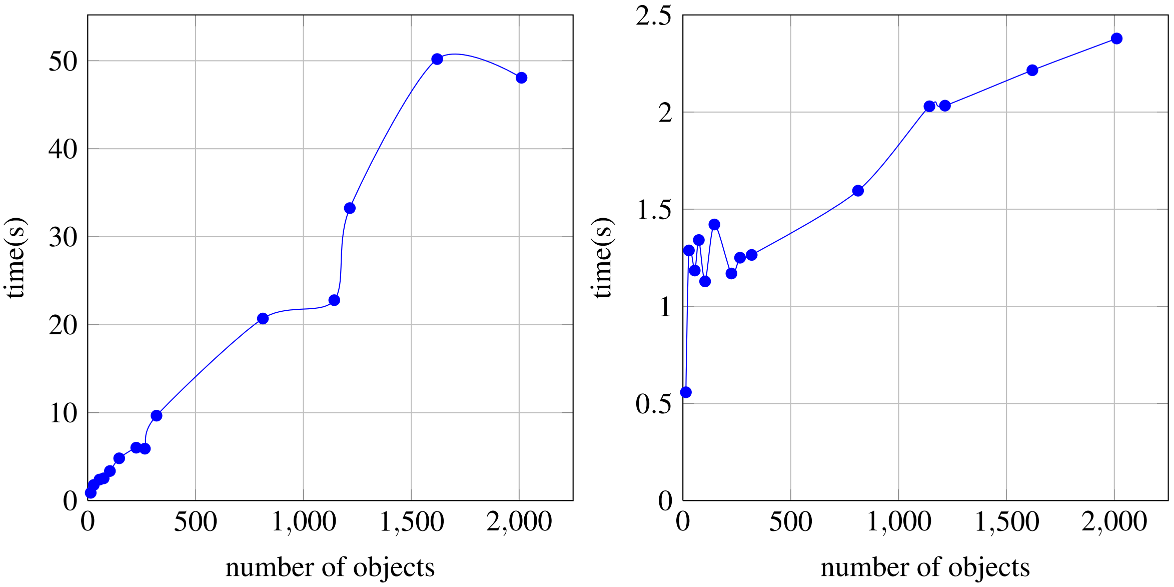
8.4.3 Researchr

Researchr is a WebDSL application that indexes scientific publications. The application allows users to find, collect, share and review publications[35]. For the experiment we first explain the important details of the data model. Followed by an explanation of the partitioning choices which are needed to generate a framework. Completing the section with the results of the experiment.

Data Model

The data model is an important aspect for a synchronization framework. The model of Researchr is big and complex. Since this is important we will discuss the most important entities including: Publication, Person, User, Bibliography, Journal, ConferenceSeries, Alias, Tag and PublicationList

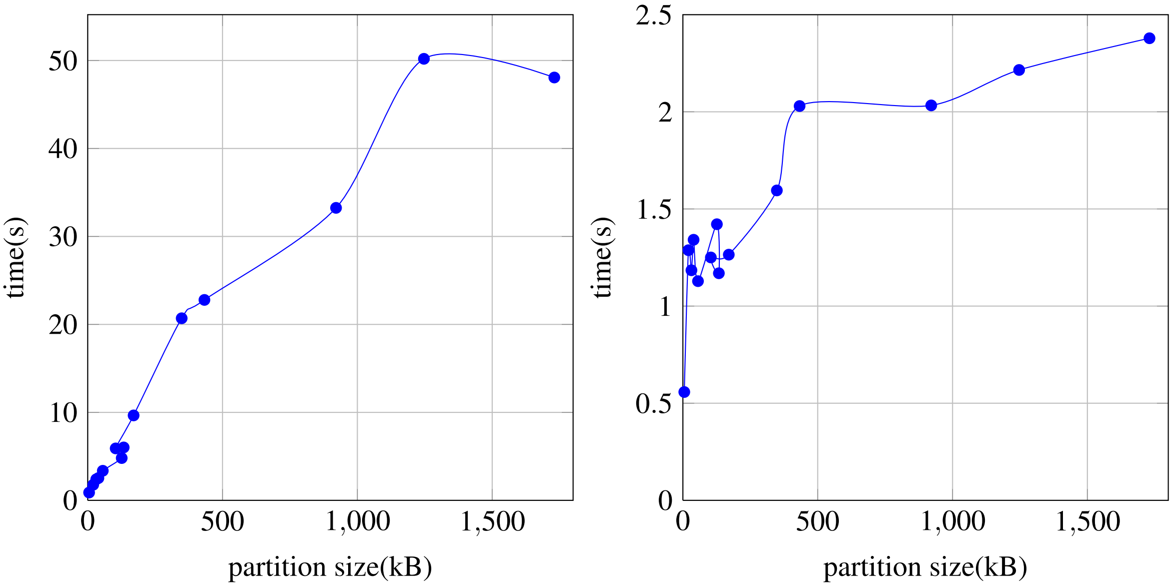
The most important entity is the *Publication*, since the functionality of Researchr is focused on publications. It has several subtypes such as *Proceeding*, *Book*, *MasterThesis* and *Article*. A publication contains some standard information like: title, abstract and year. The entity also contains references to wide range of other entities, but the most important ones are the authors and publications.



(a) full synchronization

(b) second incremental synchronization

Figure 8.14: Results of scalability experiment with existing application YellowGrass based on increasing size of partition measured by the number of objects in a partition



(a) full synchronization

(b) second incremental synchronization

Figure 8.15: Results of scalability experiment with existing application YellowGrass based on increasing size of partition measured by the sum of the size for each object in a partition

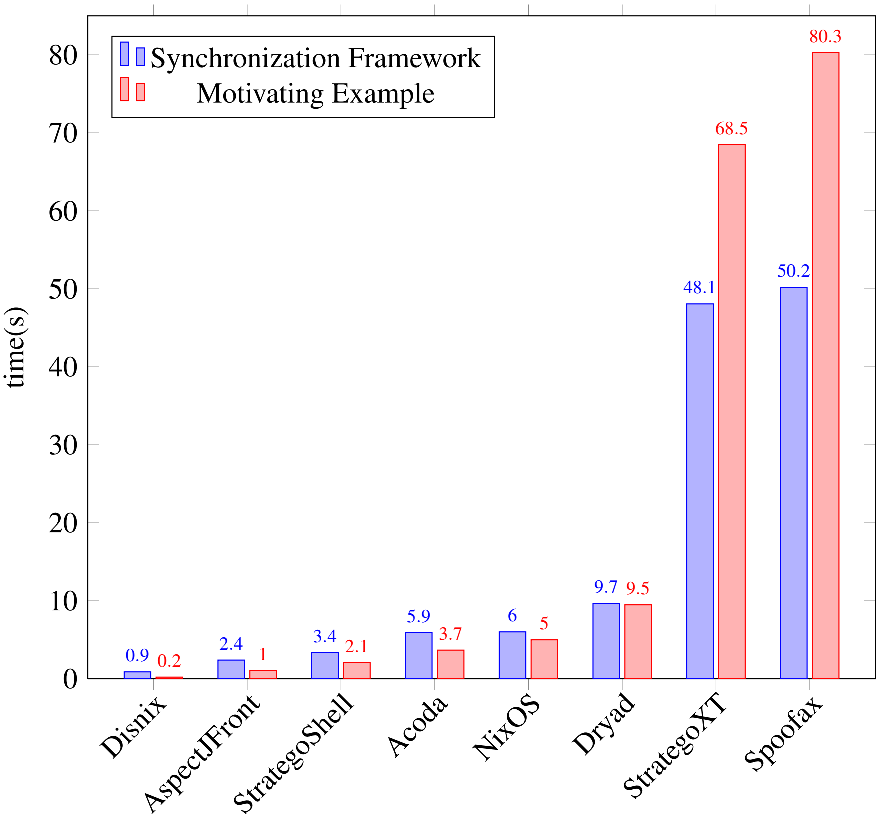


Figure 8.16: Comparison of performance between motivating example and synchronization framework

The application contains several entities that represent people within the application. The most important ones are *Person* and *User*. Person represents a person that is unique within the application. It contains some basic information about the person and references to publications which he collaborated on. User is a common entity whitin a web system to represent users. They contain user related application information like: email, username and password. It also contains references to other entities to represent personal preferences. Both entities share a link to couple both as reference so that it is easy to couple information from both objects for the same person. *AbstractAuthor* is another entity that is often used to represent a person in the system. This entity is closer coupled to publications since it represents an author of a paper. This means that the system contains a lot of duplicates for a person because each publication has a new object for each author.

Bibliography represents a list of publications composed by a user or a user group. The bibliography contains often publications of the same topic. Users can use this to keep track of publications, for example a list of publications that the user examined for his own publication.

Journals and conferences are two common ways to publish scientific work. The application contains *Journal* as higher level which contains *JournalVolume*s and *JournalIssue*s with their corresponding publications that were published in it. Similar *ConferenceSerie* has *Conference*s with its proceedings.

Alias is an entity that is used to couple different names to the same object. For example, a person can be known by several names or notations of the same name. Instead of duplicating information an Alias object is used to represent those variants.

*Tag*s are used to couple group objects of different types with a specific topic. This gives the possibility for users to create relations between objects that are not yet defined by the model.

PublicationList is entity to keep track of certain publication lists which share a common object. For example, *TagPubicationList* keeps a list of all publications with a certain tag. This entity can be replaced with search functionality and can be seen as redundant. [33].

Partitioning

The synchronization framework requires that the developers specify the toplevel entities and restricted properties if necessary. The explanation of the model and the applications show that publications are the main data of the application. This means that toplevel entities should give a group of publications as partition. Additionally, toplevel entities should not have to many objects so that a user is still able to easily select partitions. The combination of those factors can be found with the entities: User, Person, Journal, ConferenceSeries, Bibliography and Tag.

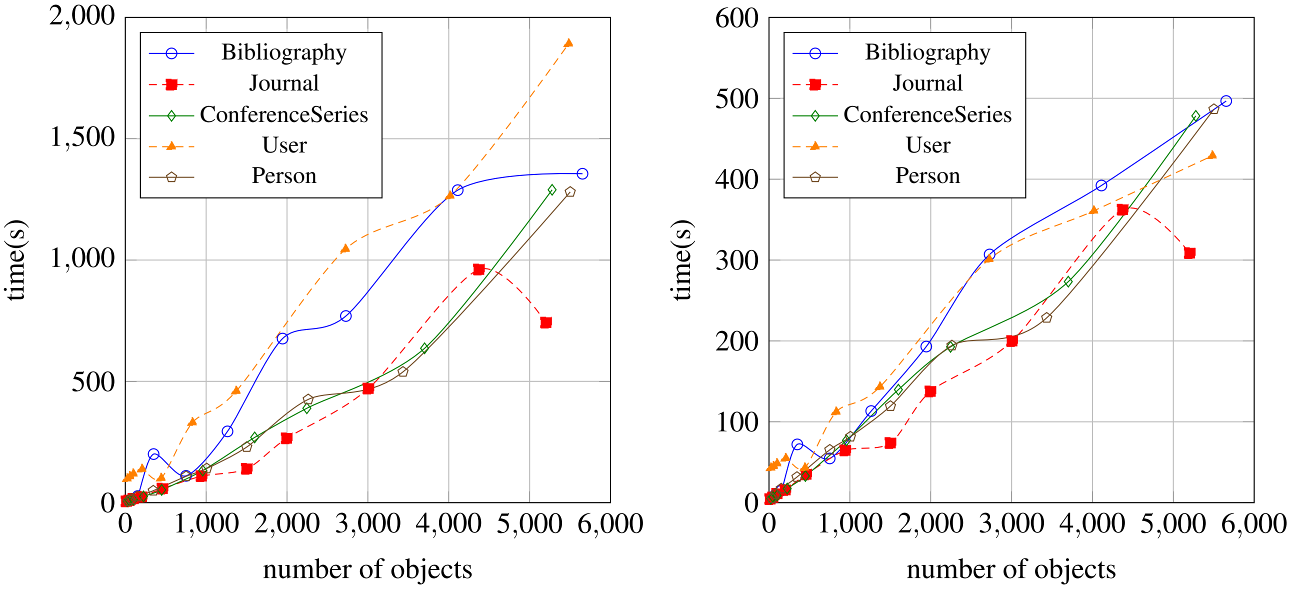
As stated before the querying performance of the Researchr database was problematic especially for the Tag entity. Which forced us to remove this entity from the model for the purpose of synchronization. References to PublicationList where removed because of redundancy. Another problem we encountered was highly connected object graph which required some extra restricted properties. This requires experience with the application and understanding of the model. While we focus on delivering the best partitioning there is high possibility that our choices do not deliver the optimal solution.

Each of the five topEntities has between 1000 and 3000 objects and gives a total of around 11000 partitions. This is still high amount of partitions, but would be usable in term of finding and selecting partitions. A bigger problem is that getting and storing all the partition information in the Mobl application is expensive. It takes around 45 minutes to store all those objects.

Results

The experimental setup has changed to a more powerful computer. This was necessary because the application required more memory than there was available in the previous setup. The new computer has 6 gB of ram and a quadcore cpu(3.0 GHz).

The goal of the experiment is to test the scalability of the framework when used on an existing application with a bigger database. Figure 8.17 shows that there is a big difference in performance compared to YellowGrass. The line still looks linear increasing to the input, however, at a much higher cost. The full synchronization is around factor 5 slower, while the second synchronization is 80 times slower compared to the results of Figure 8.14. This means that the overhead and time spend on the server for a bigger and more complex application like Researchr has a big impact.



(a) full synchronization

(b) second incremental synchronization

Figure 8.17: Results of scalability experiment with existing application Researchr based on increasing size of partition measured by the number of objects in a partition

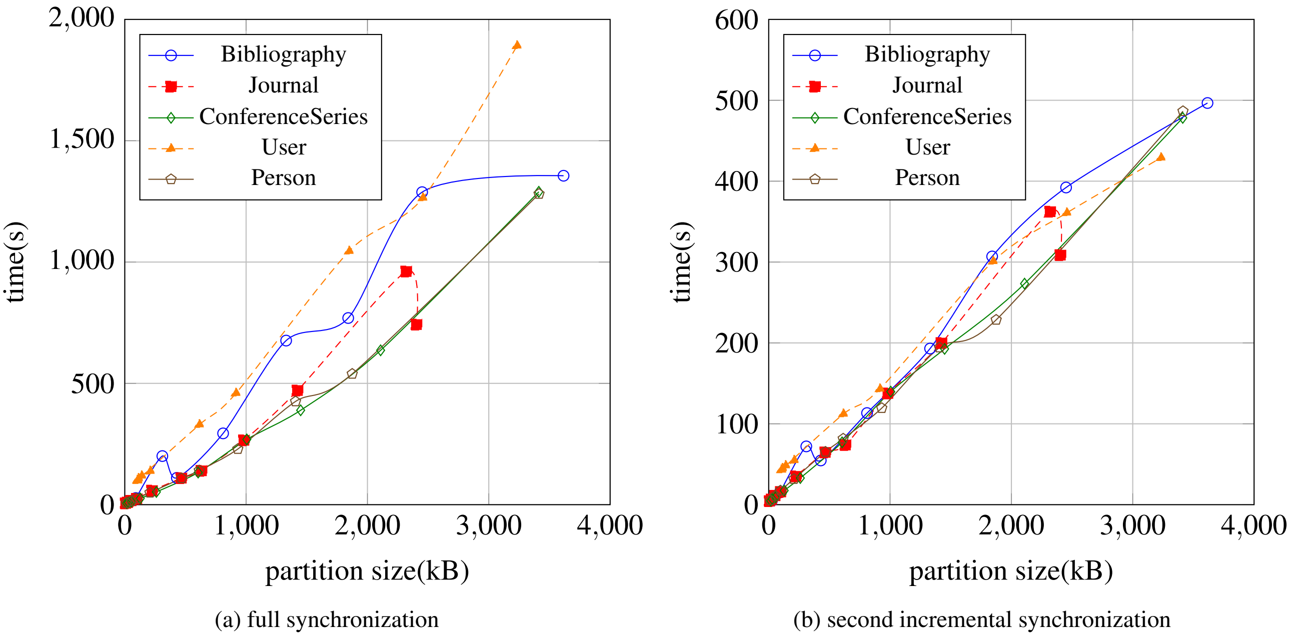
The entity partitions differ from each other in time for same partition sizes. The biggest difference between those partitions is the composition of objects. This shows that some of the object types require more time than others. When comparing the partitions in object sizes this difference is already smaller, which is displayed in Figure 8.18.

8.4.4 Discussion

The results of YellowGrass show similar numbers as the previous experiment. The performance is linear, however full synchronization takes a lot of time mostly on storing the objects on the mobile device. The synchronization framework shows improvements over the manual implementation of the motivating example. This shows that the adaptions done to improve weaknesses from the motivating example had a positive impact on the performance.

The experiment of Researchr contains two parts, the partitioning and the scalability. The partitioning for Researchr shows that it is possible with current approach to specify useful partitions in the data. In general, the partitioning of such a big database is problematic. The number of partitions and size of partitions are dependent of each other. Removing more data from the partitions is a possibility. However, there is a high chance that this restricts the functionality of the mobile application.

The partitioning of Researchr shows that even highly connected object graphs can be partitioned with current approach. We discovered two improvement possibilities, which could have given slightly better results. The first possibility is to make distinction between restricted and removed properties. It is currently possible to make



(a) full synchronization

(b) second incremental synchronization

Figure 8.18: Results of scalability experiment with existing application Researchr based on increasing size of partition measured by the sum of the size for each object in a partition

distinction by overriding the related function, but it would be better to make a setting of this possibility. Secondly, is an improved way to specify related entities. The solution allows to specify inclusion and is similar for all inputs. Nevertheless, there is a chance that the same object has a different roll in other partitions. An option is that the function can decide on possible parameters to include properties. Another adaption would be that you want to specify that an object is related. However, it should not use the related objects of that object.

The results of Researchr show a scalability issue. Only smaller partitions are usable, the bigger partitions require such amount of time that it conflicts greatly with the usability. The application is already usable when it is synchronizing, even so, the time required for synchronization of a partition is in our opinion to high. Of course one of the causes is the size of the partitions. Nevertheless, there are other factors that produce the problem. A basic problem is the time spend on queries for loading objects on the server. This increased the partition calculation time greatly. The time spend on queries is problematic for each synchronization approach, even the simple job of making JSON objects of all the data in the system was not feasible within the normal time restrictions. The increasing size of the database from YellowGrass to Researchr shows that there is an issue of scalability for WebDSL. The increase in database size was not equal to increase for the time spent on loading objects. The Mobl application shows a similar effect. The mobile application cannot deal with a large number of objects and this is mainly because of the persistence framework used by Mobl, which we have already discussed in previous experiments.

The overhead could be reduced by combining the object request into one service. This would reduce the overhead cost and reduce the server time by a constant factor, but will put more strain on the mobile application and has negative effect on the usability for remote application developers.

We have to conclude that the total solution for synchronization including the WebDSL and Mobl application have a scalability issues in size and complexity of data. It is possible that big improvements on Mobl and WebDSL persistence allow to re-evaluate the current approach. A better solution to the problem is a different approach of selecting data.

8.5 Summary

This chapter evaluates the solution based on the requirements stated in Chapter 4. The functional requirements are straightforward and easy to confirm. The solution generates a full synchronization framework containing functionality for all three steps in the process. It contains interfaces for (remote) application developers and application users. Additionally, the tool supports data restrictions through: property restriction, object access control and selectable data partitioning.

The non-functional requirements are more complicated to confirm and can be conflicting with each other. The following topics are discussed for fulfilling of requirements:

- **Usability**: The interfaces for the users are minimized and simplified where possible and keeps in mind the additional constraints delivered by mobile devices.
- **Scalability**: The synchronization requests are heavier than the average page request. Although it should be no constraint for the amount of mobile devices due to the limited number of requests needed for synchronization.
- **Applicability**: The generation supports all possible data models from WebDSL, but not all possibilities will have a good partitioning in the framework. The implementation of data partitioning is a general solution for separating data by relationship. Figure 8.1 shows several situations to cover the weaknesses of the partitioning approach.
- **Adaptability**: The current implementation allows some adaptability through extra WebDSL syntax plus full possibility to adapt functions of the generated code.
- **Security and Robustness**: While the solution provides a possibility to specify access control rules and validation expressions. Both levels of success are restricted to the effort of the developer to specify those rules.

An experiment is used to evaluate the scalability in data size and complexity for the framework. The results show a linear relation between time and the number of objects, edges, entities and the size of objects. The full synchronization can take a considerable amount of time although this is only required on a clean database and can be done asynchronously. The incremental approach should allow the further updates to be much shorter in time.

The experiment with YellowGrass shows similar results as the other experiment and has improved compared to the motivating example for total synchronization of bigger partitions. The results of the experiment with Researchr show that the solution is applicable to more complex and bigger applications. On the other hand, it exposes scalability issues for the total synchronization solution, while the application can be used in s.

In Chapter 9 shows related work to place the solution in a context. In addition, we extend the evaluation and discuss possible improvements in future work (Chapter 11).

Chapter 9

Related Work

This thesis introduces a data synchronization for web and mobile applications. Currently, there are several implementations of mobile applications that use some sort of data sharing gathered from mobile applications. We give an overview on general approaches used in popular applications and compare them in functionality to the solution in this. Secondly, we look at existing frameworks that offer data synchronization solutions for mobile applications.

9.1 Existing Approaches in Popular Applications

There are many applications with different purposes or categories. We can separate five types of applications when we look at data communication: games, social networks, information systems, messaging platforms and applications that use files. Each of those categories have a different purpose roles for data, which requires a different approach.

Games
In the past games where mainly offline applications that did not share any data. However, with the increasing quality and quantity of the internet connections the game development has been shifting to online functionality. The same has been happening for mobile games. Where even a big share of the single player games share data for several purposes.

The information shared is often that of the current status of the game. This information is stored locally and only synchronized when the internet connection is available. This allows gamers to play the game when there is temporarily no connection. On synchronization the server is always correct, which prevents users from cheating by local data modification.

The biggest difference with the synchronization of this thesis can be found in the fact that the game data is often small and does not require different representations for the client and the server.

Social Networks
Social networking is a popular activity, certainly under the younger people. The social applications allow to create a network of information shared with other users. Each

social network has a different purpose, but in general, the data they use is similar and is limited to texts, photos and videos. Most of the applications have diverse information. Nevertheless, the data views can be separated into a three sort of views: latest activity overview, search functionality and data view pages. Often, the application loads on start the latest activity to update the overview, the other pages are loaded on request. The data for the overview is cached locally so you can browse the activities offline. However, all other functionality requires a connection to gather or send data.

The biggest difference is found in the fact that it requires an internet connection for most of the functionality. This makes it impossible to use it with longer periods of no available connectivity.

Information Systems

Many applications offer information to the user for example: latest news or movie details. They are very diverse in the quantity and the dynamics of the data. This delivers a range of synchronization approaches. When data is big and often changing there is no added value in synchronization. On the other hand, there are applications that allow to synchronize full data, which makes it possible to fully use the application offline. The information applications have an in between variant of data synchronization. For example, news applications often use the activity overview approach of social networks to load latest news and caches it locally to read it without connection.

There are several data synchronization approaches used by information systems. The biggest difference is found in the fact that it only requires reading of data and not the modification aspects of synchronization.

Messaging Platforms

Messaging has always been a popular communication alternative to calling on mobile phones. This created a market for applications that allowed users to message each other for free. Those applications extended to not only send text, but also photos, videos, sounds and location information. The message sending can be seen as a synchronization of data between two or more post mobile devices. The server allows sending of messages without requiring that both devices are connected. This system works like a mail system where the server has a mailbox for every user and delivers it when it is connected.

This approach allows creation, but does not allow modifications to already sent data. Additionally, it has a different architecture namely, the mobile devices have central role and the server is used as a hub.

File Utilities

File utility applications are in general editors or viewers of various file formats. Those tools are used on documents that are stored locally. While files can be transferred to the mobile device through several ways. A popular approach is to use cloud storage services like dropbox[1]. Since the storage is big and the application only requires a few documents it selects the documents that should be synchronized.

The biggest differences with synchronization solution for this thesis can be found in the data. The cloud storage facilities use a simple model to represent files. This includes that files do not refer to each other.

[1] https://www.dropbox.com

9.2 Mobile Data Synchronization

In Chapter 3 we discussed the background of synchronization problem it gives an overview of different approaches with several examples. This section will describe various existing approaches from full database solutions up to frameworks for full data synchronization in mobile applications.

The several approaches presented in the synchronization chapter had their strengths and weaknesses as summarized in Table 3.5. This shows that the incremental approaches are more suitable for synchronization with mobile devices. SyncML [31] was given as an example for such an approach. SyncML is actually just a protocol developed for unifying several data synchronization approaches. This protocol is only meant for synchronization layer and should work independent of the data source. It provides a general solution for synchronization problems, but can be modified on almost every aspects. To be independent of the data source, the protocol requires on synchronization input from the client to deliver the changed data. While this protocol is more than a decade old it is still widely used for backing up a synchronizing data on mobile devices. Next to that it has provided a basis for custom synchronization solutions in mobile applications. It is difficult to compare SyncML since there are several implementations that differ in functionality. Generally, we could say that SyncML is a broader solution and covers more than just web and mobile application domain. However, it misses the application awareness, which leads to more implementation effort for a developer for similar functionality.

Xmiddle[25] is a data synchronization middleware solution for data sharing of XML data. It uses trees representation, which can have shared subtrees with other devices. There is no server and the devices can all have different data with shared parts for different devices. The subtrees are synchronized when the devices are connected with each other. This is a full application implementation and is based on XML data storage, which is not optimal for direct access of big data. The middleware is limited in extra functionality like security and validation.

The tree representation for data synchronization is also used by Syxaw[24]. The synchronization solution is not based for XML data stores, but for file synchronization. Nevertheless, the solution is XML-aware, which meant that when XML files are synchronized they are merged. This means that it could be used for the same purpose as Xmiddle. Syxaw provides an interface to share subtrees of the file system with other devices. It uses a central object provider to access the files for the application, but also provides extra functionality for the file synchronization. This object provider can be extended by developers to change for example merge approaches.

Industrial Solutions

The previous solutions are non commercial solutions for synchronization on mobile devices. On the other hand, there are several solutions used by the industry. This section gives an overview of popular synchronization solutions used in industry. One of those is the Microsoft Sync Framework [1, 6], which provides data and file synchronization for different platforms. The synchronization solution is an extra layer on the data source. It used extra storage for synchronization metadata. The synchronization is based on .NET databases however, it allows other sources like RSS feeds as data input. The framework does not force a specific architecture so it can support peer-to-peer network as well as a client-server approach. The basic synchronization only requires a few lines of code to specify the synchronization scope and the connection. However, for most cases you need to customize behavior like conflict resolution. The solution is quite complete, but it is bound to Microsoft technologies. This makes it hard to cover the mobile domain for a wide range of devices.

Oracle has already technology for replicating databases on servers. It used this knowledge for a data synchronization solution for mobile devices as Oracle Database Mobile Server[3]. It uses a server to communicate synchronization messages between the central database and mobile clients. The synchronization is a background process on the mobile devices and separated from the application. It supports SQL lite, which is a popular lightweight embedded database that can be used on diverse range of mobile devices. A special feature is automatic synchronization, which is based on condition to trigger the synchronization. It can also synchronize by manually calling the API. The synchronization solution is more separated from the application, which comes with less influence on the actual synchronization process. This makes it harder to have custom validation and authentication of users.

Mobilink [2, 15] a part of SQL anywhere takes a low-level approach because it works on the level of the database. Similar to the solution form oracle this database replication approach runs as a seperate process. The architecture of SQL Anywhere requires a server (Mobilink) for the synchronization process between a central database and mobile clients. This architecture enables that it can handle diverse databases and sources like XML and Excel files. To restrict the data it uses conditions that specify row and column filters for every table. The users are special entities in the database for authentication and for user based data filtering.

The previous solutions were based on problems of data replication on data store level. However, db4o[2] delivers a persisting solution for Java and C# and includes an object database. The database engine abstracts away most of the data storage facility functionality. The db4o has its own replication framework namely, db4o Replication System (dRS)[30, 14]. It can be used to synchronize with remote applications including mobile variants. The abstraction of the underlying database forces dRS to abstract on the data storage. The API adds a replicate call for objects to specify that object needs to be replicated to another device. It is a simple approach that allows high influence on synchronization. However, it comes at a price that it requires more manual code to create a more complex solution. The data store abstraction gives the possibility for more freedom in databases by using Hibernate as an alternative persistence framework. The dRS takes a similar approach to that of the solution of this thesis by having abstractions on the problem domain. Nevertheless, it misses abstractions on complex topics like automatic data selection.

Burckhardt et al. [12] describe that the consistency of shared data with high number of mobile devices is challenging. Therefore, they propose a different approach on data synchronization for mobile applications. Their solution starts with simplifying the possible data to a restricted set of cloud types. Every client keeps a log of changes on data. On synchronization the clients sends those change logs to the server. The server combines the change logs into a graph of changes and solves conflicts where encoun-

[2] http://db4o.com

tered. When combined the server sends the values that have been changed since the last synchronization. This makes the mobile device up-to-date, which means that it can start with a clean log. This reduces the data storage on the clients. The server keeps all change logs in a graph. When new information is introduced by other clients it can recalculate the values. This solution is comparable to the newer source code version solutions like git[3] and mecurial[4]. Those version control systems uses graphs to keep track of changes and can be used to combine other versions. This solution is specialized for the use of mobile applications. However, it is questionable if the simplification of types can be combined to make it usable in combination with web applications.

[3] http://git-scm.com

[4] http://mercurial.selenic.com

Chapter 10

Summary and Conclusions

This chapter contains a summarization (Section 10.1) of this thesis and the conclusions based on the research questions in Section 10.2. The last part of this chapter states the software contributions in Section 10.3.

10.1 Summary

The thesis consists of three parts: Analysis and Requirements, Motivating Example and Final solution. Each are discussed separately in the continuation of this section.

Analysis and Requirements

Chapter 2 introduces and analyzes the popular development domains of web and mobile applications to find characteristics that influences the synchronization process. The hardware limitations of mobile devices are part of the context and constrain the possible synchronization approaches. It describes the three different areas where it differs the most compared to conventional software, namely: intrinsic properties, stakeholders and discipline variety. The chapter contains an introduction to the target languages WebDSL and Mobl. Both are DSLs written in Spoofax and include separation of concerns by sublanguages. The target languages could be substituted with other languages. However, the selected languages deliver an advantage by additional abstractions over the domain.

Synchronization is a well known and researched subject in different domains like database replication and source code version control. Chapter 3 analyzes the existing solutions and knowledge of the synchronization problem to require insight for a solution. Synchronization can be separated into three stages: identification, propagation, and consistency detection and resolution of updates. The solutions for synchronization problems can be separated into three categories:

- Wholesale (Figure 3.2): Full transfer of data to calculate differences.
- Mathematical (Figure 3.3): Solutions for the set reconciliation problem mapped to synchronization domain.
- Incremental (Figure 3.4): only sends updates that are calculated locally based on meta data in objects.

The known solutions have each their advantages and disadvantages, which are summarized in Table 3.5.

The analysis of the synchronization, web and mobile domain leads to a list of requirements that are stated in Chapter 4. Those requirements are used as guideline for development and validation of the motivating example and final solution. The functional requirements describe a generated synchronization framework based on the source code of web application. It should contain interfaces for (remote) application developers and application users. Additionally, it must be possible to synchronize partial data for the mobile domain requires that does not allow the huge amounts of data. The non-functional requirements state that the solution should consider the following topics as important: usability, scalability, applicability, adaptability, security and robustness.

Motivating Example

The application YellowGrass Mobl is used as motivating example to gather practical insight in addition to the information discovered during the analysis of the domains. The example serves supplementary goal of delivering code that can be used for templates in the code generation of the final solution. Chapter 5 describes the architecture design and the interesting implementation details of the example, wich is a manually implemented data synchronization solution between WebDSL and Mobl variant of YellowGrass. The evaluation shows that much of time is spend on the mobile client and for better results it should reduce calculations by offloading it to the server.

Final Solution

The final solution is based on the analysis of the domains and the evaluation of the motivating example. The solution includes an additional layer of abstraction, which is generation of the framework. This code generation is based on the source file of the web application. The solution generates additional code for WebDSL and Mobl to create a synchronization framework. Chapter 6 describes the architecture design of the solution, which started from the motivating example and evolved into the final solution. The synchronization framework contains a data browser as an initial GUI. Security in an important factor for web systems and requires extra attention when extending the application. Therefore, we included access control on objects into the framework, which creates a dynamic security system for the data based on current principal.

The highlights of the implementation details are explained in Chapter 7. The generation is mainly based on templates in combination with entity information. Model-to-model transformation is an exception on the template based approach and is based on mapping of types from WebDSL to Mobl (Section 7.2.1). One of the more interesting topics is an explanation of the partitioning approaches, which is based on object relations and is described in Section 7.1.2. The solution covers an approach that is can be used with different target languages. Nevertheless, there are other problems that were encountered in the process, which are language specific. Examples of additional limitations specific to the target languages include: no support for hierarchy in data model, collection properties require inverse annotations and automatic tracking of status for objects.

Chapter 8 contains the evaluation of the solution based on the requirements listed in the similar named chapter. The functional requirements are covered with following description. The solution generates a full synchronization framework containing functionality for all three steps in the process. It contains interfaces for (remote) application developers and application users. In addition, the tool supports data restrictions through: property restriction, object access control and selectable data partitioning.

The evaluation is extended with an experiment, to test the scalability of the framework based on various parameters to increase the size and complexity of data. The results show a linear relation between time and the number of objects, edges, entities and the size of objects. The full synchronization takes a considerable amount of time, nevertheless the incremental update reduce the time to a tolerable level. Second part of the experiment was applied on existing applications. The results on YellowGrass showed some improvement compared to the motivating example. Researchr exposed scalability problems in synchronization time with bigger applications.

10.2 Conclusions

Research Question A *How do existing synchronizations solutions apply to the domain of web and mobile applications?*

Synchronizations is a well-known solution used in several problem domains, where several of those include extensive research. The current knowledge that the synchronization approach used depends on the domain, which is in this case web and mobile applications. Chapter 2 analyzes both domains and explains the differences between conservative, web and mobile application development.

The existing approaches described in Chapter 3 shows examples of implementations that are used in the domain of mobile devices. The incremental solutions and in particular the timestamp versioning and the changelog are good applicable to the domain of web and mobile applications. However, the existing solutions lack some specialization to be used in practice. The next paragraphs will explain the differences that need to be solved.

The interesting software related topics that introduce extra complexity are: data model, interface and security. Data is an important part of synchronization and uses a model for interpretation. The domain of web and mobile differ in the form that often the mobile application uses a simpler representation for the same type of data. The different representation implies that the synchronization requires transformations steps between different formats so that both applications can use the same data.

Synchronization requires an interface for communication between the clients. Synchronization solutions often have their own implementation of communication layer that is used for direct communication with remote hosts. Web applications have their own standard for communication layer, namely: webservices. The difficulty introduced with webservices is that they are open accessible for the world. They could be secured to only accept certain applications. However, it is more common that the services are open for other applications as well.

Web applications are open and accessible for everyone and contain data from all the users of the system. This amplifies the need for a good security measurements compared to conventional software. The data synchronization would extend the web application and allows access to require data and possibility to modify data. This gives additional entry point that should be protected against erroneous input and restrict access based on user rights.

The hardware for web applications are one or multiple servers that serve the web application. Those servers are quite powerful and should not restrict the calculation or space that is needed for synchronization. Nevertheless, they should deal with many requests. This demands two properties of a synchronization solution: It should be non blocking to allow other page or service requests. It should be scalable in the form that it can handle synchronization for many mobile clients.

The mobile application runs on mobile devices which differ much in several hardware specifications. The synchronization should be possible on wide range of devices so it should assume low hardware specifications. The synchronization is mostly constrained by the connectivity, available memory and computation power. The connection on mobile devices delivers two problems:

- The server is not always available, meaning that the synchronization should allow that it can work even with gaps of disconnection.

- The connection is slow and expensive, meaning that the synchronization should reduce the amount of data that is communicated.

The available amount of memory on mobile applications is limited to a small share of the space on the device. Therefore, it cannot handle the amount of data that is used for the database from the web application. Simplification and reduction of objects is necessary to make synchronization feasible for a mobile platform. The computation on mobile devices is slower and less powerful than average computers, which implies that it cannot be used for expensive computations.

Research Question B *How can we automate the creation of incremental data synchronization between web and mobile applications?*

Chapter 6 and Chapter 7 describe a solution that generates an incremental data synchronization framework. The generation mostly contains of templates that are filled with application information. This means that the solution is based on the source of the web application and requires the following information: entities, properties, type information, security rules, entry point, property restriction.

A complete synchronization solution requires implementation of: detection of updates, propagation of updates, and detection and resolution of inconsistencies. Incremental updates are required for a usable solution in mobile and web applications. The incremental approach requires additional information in the detection to discover updates in specific time range.

The synchronization algorithm is part of the framework, which at least should contain the following components in addition to the algorithm:

- A (generated) **data model** for the mobile client to interpret data on mobile devices
- **Mappers** to transform objects and updates to (de)serialized forms
- A **Communication layer** to propagate updates to remote devices

The base components should be extended with the following components to fully satisfy the demands:

- **Access Control** to secure the synchronization framework based on user privileges
- **Authentication** is required to use the full possibilities of the access control system
- **Object Validation** to improve inconsistency detection of updates
- **Partitioning functions** for partial data synchronization
- **Integration functions** to reduce interaction and improve usability

Research Question C *How can we optimize a synchronization algorithm to use a minimal amount of computation on mobile devices?*

The first question already encountered the restrictions of the mobile devices. To increase the usability of the synchronization framework it should focus on dealing with the restrictions. Memory and connectivity are hard restrictions which cannot be solved by software. Instead, they should be considered as environment of the problem. Computation on the other hand is much more flexible, since the work can be calculated remotely and then be used locally.

This requires from the synchronization that most of the work is done on the server to reduce load for devices. The first step in synchronization is the identification of updates. This calculation depends on which sort of synchronization solution is used. The only approach that allows to offload this computation is wholesale, since it requires the server to have all data from the remote to calculate the differences. The incremental solutions only require a small amount of computation to find updates and can be formulated into database queries. Finding updates on the server requires information of the status of objects on the mobile device. Sending version information for all objects must be prevented to reduce communication cost.

Propagation of updates is a process of sending the updates to the other client. For mobile devices this includes sending the local changes and persisting the remote changes. The computation seems to be minimal, but actually requires some amount of computation for mapping to different data representation. The full mapping cannot be calculated on the server side, since it has to obey the format of the webservice protocol. The possible reduction can be found in using a protocol with data representation close to that of the mobile application.

The last step for synchronization is detection and resolution of inconsistencies in updates. Detection and resolution should always be done to prevent an incorrect database on the server. The reason to calculate this on the server is that mobile clients cannot be trusted. Resolution for those inconsistencies are also required on the mobile application, which prevents that incorrect data is kept in the system. Including detection and resolution results in the feedback from the server would prevent that mobile clients need local detection and resolution calculations.

10.3 Software Contributions

In addition to the conclusion on the research questions this thesis contains the three following main software contributions:

WebDSL extension: generating data synchronization frameworks

The first technical contribution is an extension to WebDSL that is able to generate a synchronization framework for Mobl (and possible other remote) application, based on the source code of the web application. The framework contains the following special functionality:

- A (restricted) representative model of the WebDSL application in Mobl
- Mapper functions to transform updates to several representations
- Keeping track of changed objects for incremental updates
- Automatic Data partitioning for selective synchronization
- Validation of updates by model validation, preventing data inconsistencies
- Access control for: read, create and modify restrictions on object level
- Additional functionality for easy integration within the WebDSL and Mobl application

Data browser

The fourth addition is a generated data browser for the Mobl application. It uses the data model extracted for the synchronization to generate a view for every entity. A developer can use those views to display objects and edit simple type properties. Buttons are added for properties that refer to other objects to browse through the local data. This extension is meant as a quick start for developing a mobile application.

Remote device authentication module

The last additional feature of the contributions is an additional module for the WebDSL application that allows remote devices to authenticate with the server using a device keys instead of a password. This allows more secure authentication and possibility to de-authenticate devices remotely without changing the password. This module is required for use of full functionality of the framework and also contains a Mobl module. However, it can be used separately for other authentication purposed with remote devices.

Chapter 11

Future Work

This chapter contains interesting future work related to the solution of this thesis and can be separated into the following topics: fine grained synchronization, automatic partitioning and total solution.

11.1 Fine Grained Synchronization Framework

The current solution in this thesis is mainly based on the level of objects. However, this approach has its weaknesses namely:

- Updates contain unnecessary data, so can be reduced in size.
- Updates might trigger false positives in form of conflicts, since updates are done on different properties.
- Access control only allows total restriction of objects, instead of partial properties.
- Small modification for property requires rewrite of total function for entity.
- Applications with high amount of object have to deal with big or many partitions.

The following paragraphs describe possible approaches which can be used to improve or solve those weaknesses.

Reduced Updates
The current implementation traces updates with extra properties tracking the status on object, by flags or timestamps (Section 7.1.1). This allows only status for objects and does not tell which properties are changed. To reduce the size of updates and remove unnecessary data it is required to track status on properties.

A possible solution is one that is used in the synchronization solution by Burckhardt et al. [12]. Their solution is based on keeping track of changes instead of status on objects or values. The weakness is that it misses grouping of updates and it is expensive in space to keep multiple versions of properties. This approach could work on mobile devices where the log is short, since it can be cleared after synchronization. However, the server cannot clear this log after synchronization. It can only be cleaned when it knows that all devices are up to a certain point in the log.

Another solution is that of status tracking on the level of property values instead of objects. This approach has a higher initial overhead for objects, but is constant in relation to the amount of changes. This delivers a more scalable approach for client and server.

Improved Inconsistency Detection
The solution provides inconsistency checking based on the versions of the current object in the database and in the update (Section 7.1.1). This is a pessimistic approach because there is a possibility that the changes do not interfere with each other. When keeping track of local changes it is possible to compare versions of the property values to detect inconsistencies. This will decrease the number of false positives and therefore increase the user friendliness of the framework.

Property Restriction with Access Control
The access control for the synchronization framework allows to write boolean expressions that define rules to restrict data access based on the current principal (Section 7.2.3). However, in practice it is not always that strict that a user is restricted for the whole object. A better way is to use a restricted set of properties. The last sentence already gives away the core of a possible solution. This would be an modification that returns a set of allowed (or restricted) properties instead of using a boolean value.

Fine Grained Adaptation of Generation
The WebDSL source is used as input for the code generation. Some additional syntax was added to adapt the generated functions (Section 6.4.2). This does not completely cover the possible adaptations that can be used within the synchronization framework. In the current implementation those functions can be overwritten. However, this requires implementation for the whole entity and includes boilerplate code. Possible extensions to the WebDSL compiler can be added to specify modifications on property level. Currently, for a solution we would focus on the mapping of property values to and from (de)serialized form. Nevertheless, there could be other adaptions for other functions like related entity calculation.

Fine Grained Partitioning
The evaluation with Researchr showed that there is a scalability issue. Most of the time is spent on persisting the object to the local database. Reducing the partition sizes would be a good approach. Certainly, when you take into consideration that current mobile applications also try to limit the amount of objects. The current solution allows to define smaller partitions, but this will create a high amount of partitions. This is not really user friendly, since it increases the time to synchronize the list of partitions and the easiness of selecting the partitions.

A possible approach is on-demand requesting of objects. There are two disadvantages to this approach, which influences the usability.

- **Offline Capability**: When the mobile application is offline you can only use available objects. This effect is the same for the current solution. However, the current solution gives the possibility to easily request a set of objects before loosing connectivity. While the on-demand approach would be limited to the objects that have been used before.

- **Reactivity**: The mobile application has to wait for the object to be returned from the server. The time required for an single object from the sever can increase to more than 10 seconds. Even if the would be around halve a second the interactivity will be dramatic for a user.

11.2 Improved Data Partitioning

The data partitioning used in this thesis, based on object relations, is not optimal for all situations. The following problems are encountered with the use of current solution:

- The automatic partitioning is nontrivial and could deliver non optimal divisions.
- The partition calculation is expensive compared to web page request and is calculated for every synchronization request.

The following paragraphs suggest possible approaches to counter the problems that have been described above.

Partitioning Based on Run-Time Information
The current partition is based related on the information in the source code of the web application. This information is limited to the entities and properties plus some extra information delivered by the WebDSL extension (Section 7.1.2). The problem is that the source code only gives information how entities relate. However, the partitioning should be based on the object level information. This is in some situations problematic for the related entity calculation (Figure 8.1). The developer can tweak the behavior of the calculation although it requires from him to have good insight in the object graph of the application.

It would be preferable if this calculation or tweaks could be done automatically. This might be possible if it is calculated at run-time, where it can use the object information available that is available. Additionally, it could use other information like the usage of relations to determine weights for edges. It is not clear if this can be tweaked on the run-time or that it requires modification of the code.

Caching of Related Entities
Every request for updates in the synchronization framework requires to gather all objects of a partition, which is done using the calculation of the related entities (Section 7.1.2). It is a relatively expensive calculation and is executed often, certainly when increasing the users of the mobile application.

Another approach of this calculation is to store the value as set of entities as property of an object. It needs to recalculate the value when the object has changed one of its outgoing edges. It is much more efficient to calculate on change instead on request when the application deals with many users. However, there is a problem that the related entities calculation is based on the objects of the outgoing edges. This means that change of an object should also trigger calculation of related entities for objects that have outgoing edges to the current object. This requires reverse information for incoming edges, which delivers extra overhead. A similar problem occurs for calculation of search indexes for objects. This indexing problem is solved in Hibernate Search [11].

11.3 Total Solution

The motivation for this thesis is based on the fact that many web applications also have a mobile variant, which creates code duplication. This synchronization framework is a first step to a possible more general goal of simplification of creation of mobile application for web applications. The next step would be to generate more of the mobile application based on the source code of the web applications. We can separate two sorts of code that is lacking in the generation: Action and UI code. The next paragraph will discuss those two topics and what are the obstacles.

Action Code
It seems an interesting approach to do similar calculations of the web application in the mobile variant. This is most likely limited to the calculations that are simple and do not require a big share of the database to be available. Code like methods could be interesting for functionality in the application. It would be an improvement to have functions for validation expressions and access control rules to enhance the synchronization framework. Those functions could be used to validate before the propagation and reduce the number of errors that could possibly be returned. This approach has as restriction that it can not calculate the collisions with possible other (remote) updates.

A first step that could be done to extend the synchronization framework is to port the validate expressions. This requires that the expressions in validation rules are transformed to Mobl equivalent variants. The syntax of both languages are quite similar and would seem to be fairly easy to adapt. This transformation will run into problems because it is missing the functions that can be called within the expression. Extending the transformation to include possible methods and functions will expose the missing of library functionality. Those should be implemented manually to cover those possibilities. Assuming that this is possible it would seem to be fine. Nevertheless the predicate that the syntax similarity implies semantic equality is fragile. So all transformations require to preserve semantics to be sure that this would work. Related to this topic is PIL. PIL is a language with basic building programming building blocks and transformation to other languages. This enables generation of implementations for multiple target languages with one code base. [21].

UI code
The solution of this thesis generates a data browser to have a simple UI. The generated UI would be much more usable if it would use the existing templates to generate views for the mobile application. UI transformation triggers some different difficulties as explained in the next paragraphs.

The code that is used for UI is a different perspective than that of action code. Where action code wants to preserver semantics of the code, UI is more concerned about displaying the same information in an optimal way for the target device. The target devices for mobile and web applications differ in screen size and input hardware. This actually requires a paradigm shift to display similar information.

Simple input and output templates could be mappable from a web variant to one that can be used for mobile devices, since they are simple and do not require much knowledge about the other view elements. On the other hand, the bigger views, which are combinations of multiple templates have to deal with reordering of elements or even splitting up in separate views. This is often a form of taste of the users and developers, which is a subjective interpretation. It seems that because of subjectivity, the best case scenario could be delivered using heuristics that could guess a transformation on views. The generation of some basic templates might already reduce workload and duplication to a minimum.

Bibliography

[1] Introduction to microsoft sync framework. http://msdn.microsoft.com/en-us/sync/bb821992.aspx, 2009.

[2] Mobilink: Getting started. http://dcx.sybase.com/1200/en/pdf/mlstart12.pdf, 2010.

[3] Oracle database mobile server: Developers guide. http://docs.oracle.com/cd/E35865_01/doc.1120/e29740.pdf, 2012.

[4] Cisco visual networking index: Global mobile data traffic forecast update, 2012Ű2017. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html, 2013.

[5] February 2013 web server survey. http://netcraft.com, February 2013.

[6] Introduction to sync framework database synchronization. http://msdn.microsoft.com/en-us/sync/bb887608, July 2013.

[7] Sachin Agarwal, David Starobinski, and Ari Trachtenberg. On the scalability of data synchronization protocols for pdas and mobile devices. *Network, IEEE*, 16(4):22–28, 2002.

[8] Sachin Kumar Agarwal. *Efficient reconciliation of unstructured and structured data over networks*. PhD thesis, Boston University, Boston, MA, USA, 2006. AAI3186484.

[9] Open Mobile Alliance. Syncml specifications. http://technical.openmobilealliance.org/technical/syncmlindex.aspx.

[10] Appcelerator. Naive vs. html5 mobile app development: Which option is best? http://www.appcelerator.com.s3.amazonaws.com/pdf/appcelerator-whitepaper-native-html5.pdf, 2012.

[11] Emmanuel Bernard, Hardy Ferentschik, Gustavo Fernandes, Sanne Grinovero, and Nabeel Ali Memon. Hibernate search, apache luceneŹ integration : Reference guide 4.3.0.final. http://docs.jboss.org/hibernate/search/4.3/reference/en-US/pdf/hibernate_search_reference.pdf, 2013.

[12] Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P Wood. Cloud types for eventual consistency. In *ECOOP 2012–Object-Oriented Programming*, pages 283–307. Springer, 2012.

[13] Todd Ekenstam, Charles Matheny, Peter L. Reiher, and Gerald J. Popek. The bengal database replication system. *Distributed and Parallel Databases*, 9(3):187–210, 2001.

[14] Eric Falsken. Enabling the mobile enterprise with db4o. `http://db4o.com/about/productinformation/whitepapers/db4o%20Whitepaper%20-%20Enabling%20the%20Mobile%20Enterprise%20with%20db4o.pdf`, 2006.

[15] Eric Giguère. Mobile data management: Challenges of wireless and offline data access. In *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 227–228. IEEE, 2001.

[16] Google. Managing html5 offline storage. `https://developers.google.com/chrome/whitepapers/storage`, 2012.

[17] Danny M. Groenewegen, Zef Hemel, and Eelco Visser. Separation of concerns and linguistic integration in WebDSL. *IEEE Software*, 27(5):31–37, 2010.

[18] Danny M. Groenewegen and Eelco Visser. Integration of data validation and user interface concerns in a dsl for web applications. *Software and Systems Modeling*, 12(1):35–52, February 2013.

[19] Zef Hemel, Danny M. Groenewegen, Lennart C. L. Kats, and Eelco Visser. Static consistency checking of web applications with WebDSL. *Journal of Symbolic Computation*, 46(2):150–182, 2011.

[20] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code generation by model transformation: a case study in transformation modularity. *Software and Systems Modeling*, 9(3):375–402, June 2010.

[21] Zef Hemel and Eelco Visser. PIL: A platform independent language for retargetable DSLs. In Mark G. J. van den Brand, Dragan Gasevic, and Jeffrey G. Gray, editors, *Software Language Engineering, Second International Conference, SLE 2009*, volume 5969 of *Lecture Notes in Computer Science*, pages 224–243. Springer, 2009.

[22] Zef Hemel and Eelco Visser. Declaratively programming the mobile web with mobl. In Kathleen Fisher and Cristina Videira Lopes, editors, *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, OOPSLA 2011*, pages 695–712, New York, NY, USA, 2011. ACM.

[23] Stephen Kost. *An Introduction to SQL Injection Attacks for Oracle Developers*. Integrigy Corporation, 2004.

[24] Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. Syxaw: Data synchronization middleware for the mobile web. *Mobile Networks and Applications*, 14(5):661–676, 2009.

[25] Cecilia Mascolo, Licia Capra, Stefanos Zachariadis, and Wolfgang Emmerich. Xmiddle: a data-sharing middleware for mobile computing. *Wireless Personal Communications*, 21(1):77–103, 2002.

[26] Julie McKeehan and Neil Rhodes. *Palm OS programming: the developer's guide*, chapter 14-16. O'Reilly Media, Incorporated, 2001.

[27] Emilia Mendes, Nile Mosley, and Steve Counsell. The need for web engineering: An introduction. *Web Engineering*, pages 1–27, 2006.

[28] Jakob Nielsen and JoAnn T Hackos. *Usability engineering*, volume 125184069. Academic press San Diego, 1993.

[29] Nokia. Managing mobility: An it perspective. `http://www.majorcities.eu/generaldocuments/pdf/nokia_managing_mobility.pdf`, 2006.

[30] Jim Paterson and Stefan Edlich. *The definitive guide to db4o*. Apress, 2006.

[31] Ligang Ren and Junde Song. Data synchronization in the mobile internet. In *Computer Supported Cooperative Work in Design, 2002. The 7th International Conference on*, pages 95–98. IEEE, 2002.

[32] Ari Trachtenberg, David Starobinski, and Sachin Agarwal. Fast pda synchronization using characteristic polynomial interpolation. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1510–1519. IEEE, 2002.

[33] Elmer van Chastelet. A domain-specific language for internal site search. Master's thesis, TU Delft, August 2013.

[34] Eelco Visser. WebDSL: A case study in domain-specific language engineering. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007*, volume 5235 of *Lecture Notes in Computer Science*, pages 291–373, Braga, Portugal, 2007. Springer.

[35] Eelco Visser. Performing systematic literature reviews with researchr: Tool demonstration. Technical Report TUD-SERG-2010-010, Software Engineering Research Group, Delft University of Technology, Delft, The Netherlands, May 2010.

[36] Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013. http://dslbook.org.

[37] W3C. A vocabulary and associated apis for html and xhtml: W3c candidate recommendation 17 december 2012. `http://www.w3.org/TR/2012/CR-html5-20121217/browsers.html#offline`, 2012.

[38] George White. Web sql database gotchas. `http://cantina.co/2010/10/08/web-sql-database-gotchas/`, 2010.

Appendix A

WebService Data Types

This table displays all non default JSON data types that are used by the generated webservices for synchronization.

Type	Fieldname	Type	Description
UserCredentials	username	String	username of current actor
	pw	String	password of current actor
	devicename	String	unique description for device
DeviceKey	key	String	unique key to authenticate device
DeviceCredentials	username	String	username of current actor
	devicename	String	unique description for device
	devicekey	String	unique key to authenticate device
EntityListObject	name	String	type of the objects in the list
	value	EntityObject*	list of objects
PartitionListObject	name	String	type of the TopLevelEntity
	value	PartitionObject*	list of partition objects
PartitionObject	id	String	identifier for object representing partition
	lastSynced	Long	timestamp of previous synchronization of partition
ErrorObject	ent	String	type of object
	id	String	identifier to represent object where errors occurred on
	errors	ErrorTypeObject*	list of errors occurred on entity
	restore[1]	EntityObject	server representation of object to fix local object
ErrorTypeObject	message	String	error message
	type	String	level of the error
EntityObject	Propertyname*	PropertyType	JSON representation of entity object

* = zero or more

[1] Only available in ErrorObjects of Send Modified Objects service.

Table A.1: The specification of types used by the webservices of the generated code

Appendix B

Pseudo Code Synchronization Algorithm

This Chapter contains the pseudo code for the synchronization algorithm used in the final solution.

```
function synchronizeFindNewObjects()
  updates :=  new JSONArray()
  [[ // blok is replicated for each entity in data Model
    objectlist := JSONObject()
    objectlist.name := "<ENTITY>"
    newObjects := JSONArray()
    foreach object in <ENTITY>.all() where object.created
      newObjects.add(object.toJSON)
    objectlist.value := newObjects
    updates.put(objectlist)
  ]]
  return updates
```

(a) Mobile

```
function synchronizeFindUpdatesFor<ENTITY>(partitionList)
  updates :=  new Set()
  foreach objectlist in partitionList
    switch objectlist.name
    [[ // blok is replicated for each TopLevelEntity in the data Model
      case "<TOPLEVELENTITY>":
        foreach topObject in objectlist.value
          topLevelObject := loadObject(topObject.id, "<TOPLEVELENTITY>")
          partition := getAll<ENTITY>ForPartition(topLevelObject)
          timestamp := topObject.lastSync
          foreach object in partition where object.modified > timestamp
            updates.put(object)
    ]]
  return updates
```

(b) Server

<X> = placeholder for entity name

Figure B.1: Pseudocode for identification in generated synchronization framework

```
function synchronize()
  timestamp := getTimestamp()
  newObjects := synchronizeFindNewObjects()
  result := SendNewObjects(newObjects)
  modifiedObjects := synchronizeModifiedObjects()
  result := sendModifiedObjects(modifiedObjects)
  partition := getSelectedPartitions()
  [[ // blok is replicated for each entity in data Model
    updates := getUpdates<ENTITY>(partition)
    foreach JSONObject in updates
      <ENTITY>.persistJSON(JSONObject)
  ]]
  foreach object in partition
    object.lastSync := timestamp
```

(a) Mobile

```
function sendModifiedObjects(entityLists)
  foreach entityListObject in entityLists
    switch entityListObject.name
  [[ // blok is replicated for each Entity in the data Model
      case "<ENTITY>":
        foreach object in entityListObject.value
          localObject := loadObject(object.id, "<ENTITY>")
          edit<ENTITY>Mapper(localObject, object)
  ]]
```

(b) Server processing modifications

```
function sendModifiedObjects(entityLists)
  var errors := JSONArray()
  foreach entityListObject in entityLists
    switch entityListObject.name
  [[ // blok is replicated for each Entity in the data Model
      case "<ENTITY>":
        foreach object in entityListObject.value
          localErrors := JSONArray()
          localObject := loadObject(object.id, "<ENTITY>")
          if(object.version >= localObject.version)
            edit<ENTITY>Mapper(localObject, object, localErrors)
            foreach error in localObject.validate()
              localErrors.put( createWarning(error))
          else
            localErrors.put( createError("outdated object"))
          if(localErrors.length > 0)
            errors.put(createErrorObject(localObject, localErrors))

  ]]
  return errors
```

<X> = placeholder for entity name
createWarning and createErrorObject are functions which transform the input to correct JSON format

Figure B.3: Improved pseudocode for detection of inconsistencies in generated synchronization framework

```
function sendModifiedObjects(entityLists)
  var errors := JSONArray()
  foreach entityListObject in entityLists
    switch entityListObject.name
  [[ // blok is replicated for each Entity in the data Model
      case "<ENTITY>":
        foreach object in entityListObject.value
          localErrors := JSONArray()
          localObject := loadObject(object.id, "<ENTITY>")
          if(object.version >= localObject.version)
            edit<ENTITY>Mapper(localObject, object, localErrors)
            foreach error in localObject.validate()
              localErrors.put( createWarning(error))
          else
            localErrors.put( createError("outdated object"))
          if(localErrors.length > 0)
            errors.put(createErrorObject(localObject, localErrors))
            if (containsError(localErrors))
              rollbackAndStartNewTransaction();
  ]]
  return errors
```

<X> = placeholder for entity name
createWarning and createErrorObject are functions which transform the input to correct JSON format

Figure B.4: Improved pseudocode for resolution of inconsistencies in generated synchronization framework

```
function synchronize()
  timestamp := getTimestamp()
  newObjects := synchronizeFindNewObjects()
  errors := SendNewObjects(newObjects)
  clearNew(getIDs(errors))
  returnErrors(errors)
  clearNew([])
  modifiedObjects := synchronizeModifiedObjects()
  errors := sendModifiedObjects(modifiedObjects)
  clearDirty(getIDs(errors))
  returnErrors(errors)
  restoreObjects(errors)
  clearDirty([])
  partition := getSelectedPartitions()
  [[ // blok is replicated for each entity in data Model
    updates := getUpdates<ENTITY>(partition)
    foreach JSONObject in updates
      <ENTITY>.persistJSON(JSONObject)
  ]]
  foreach object in partition
    object.lastSync := timestamp
```

<X> = placeholder for entity name

getIDs makes a list of object identifiers of all the objects in the error list.

returnErrors gives the possibility to interact based on the errors, possibly could stop the synchronization.

Figure B.5: Improved pseudocode in mobile application for resolution of inconsistencies in generated synchronization framework

```
function clearDirty(excludedObjects)
  [[ // blok is replicated for each Entity in the data Model
    foreach object in <ENTITY>.all() where object.dirty == true
      if(object.id not in excludedObjects)
        object.dirty := false
  ]]

function clearNew(excludedObjects)
  [[ // blok is replicated for each Entity in the data Model
    foreach object in <ENTITY>.all() where object.created == true
      if(object.id not in excludedObjects)
        objecte.delete()
  ]]

function restoreObjects(errors)
  foreach error in errors
    if(error.restore)
        switch error.ent
        [[ // blok is replicated for each Entity in the data Model
          case "<ENTITY>":
          <ENTITY>.persistJSON(error.restore)
        ]]
```

<X> = placeholder for entity name

Figure B.6: Pseudocode of resolution functions for mobile application in generated synchronization framework

```
function getUpdates<ENTITY>(partitionList)
  result := JSONArray()
  updates := synchronizeFindUpdatesFor<ENTITY>(partitionList)
  foreach object in updates where object.mayRead()
    result.add(object.toJSON())
  return result
```

<X> = placeholder for entity name

Figure B.7: Improved pseudocode in mobile application for resolution of inconsistencies in generated synchronization framework

```
function getAll<ENTITY>ForTopEntity(TopLevelEntities)
  todo := new Queue()
  seen := new Set()
  found := new Set()
  foreach topLevelEntity in TopLevelEntities
    seen.add(topLevelEntity)
    if(topLevelEntity instanceof <ENTITY>)
      found.add(topLevelEntity)
    else
      related := getSetWhereNotSeen(seen, topLevelEntity.getRelatedEntities())
      todo.addAll(related)
      seen.addAll(related)
  while(todo.length > 0)
      entity := todo.next()
    if(entity instanceof <ENTITY>)
      found.add(entity)
    if(!isTopLevelEntity(entity))
      related := getSetWhereNotSeen(seen, entity.getRelatedEntities())
      todo.addAll(related)
      seen.addAll(related)
  return found

function getSetWhereNotSeen(seen, addition)
  newObjects := Set()
  foreach object in addition where object not in seen
    newObjects.add(object)
  return newObjects
```

<X> = placeholder for entity name
isTopLevelEntity is a help function that checks whether an object is a TopLevelEntity
getRelatedEntities method returns a set of objects accumulated for all entities in its properties

Figure B.8: Pseudocode for gathering object of partitions in generated synchronization framework