A Demonstrator Framework for Consistency Management Approaches

by

Arjya Shankar Mishra

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

A Demonstrator Framework for Consistency Management Approaches

Master's Thesis

Submitted to the Fakultät für EIM, Institut für Informatik
in Partial Fulfillment of the Requirements for the
Degree of

Master of Science

by
Arjya Shankar Mishra

Supervisors:
Jun. Prof. Dr. Anthony Anjorin
Prof. Dr. Gregor Engels

Paderborn, May 27, 2017

Declaration

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

City, Date

Signature

Contents

1 Introduction and Motivation

In the era of Information Technology, the usage of software and its applications is continuously increasing and has become an important part of our lives. This makes the software industry one of the largest industries in the world and many companies are built around the development of software. With the growing usage of software, the software development process has changed drastically and has become more solution-oriented. Nowadays, the entire focus is on making the software development process fast, less complex, and more human-friendly.

One of the approaches to reducing the complexity of software development is abstraction and separation of concerns [21]. In recent times, (software) modeling has become an effective way of implementing this principle. In a traditional approach, developers manually write programs and check the specifications based on software models, which is often costly, incomplete, informal, and carries a major risk of failure. In contrast, model-driven software development (referred to as **MDSD** from now on) improves the way software is built by moving the focus from programming language code to representing the essential aspects of software in the form of software models. It reduces development costs and increases the reusability and maintainability of software. The objective of MDSD [21] is to increase productivity and reduce time-to-market by enabling development at a higher level of abstraction and by using concepts closer to the problem domain at hand, rather than what is directly offered by programming languages.

The core idea of the MDSD approach is based on models, modeling and model transformations. In this approach, developers represent real world systems as models at a suitable level of abstraction. Different models can be used to represent different views of a system. Although these views are separate and result in models that can be independently manipulated by different developers, there are still numerous relations between models that must be taken into account to ensure that the entire system, described by the state of all models, is consistent. This phenomenon is handled by model transformation and certainly increases the developers productivity and quality of the models.

Bidirectional transformation (referred to as **bx** from now on) is a technique used to synchronize two (or more) models. Such models are related, but don't necessarily contain the same information. Changes in one model can thus lead to changes in other models [1].

Bidirectional transformation is used to deal with scenarios like[3]:

- change propagation to the user interface as a result of underlying data changes
- synchronization of business/software models
- refreshable data-cache incase of database changes
- consistency management between two artifacts by avoiding data loss

The bx community (`http://bx-community.wikidot.com/`) has been doing research in many fields including software development, databases, mathematics and much more, to increase awareness for bx [1][2]. As a result, many kinds of bx tools are being developed. These bx tools are based on various approaches, such as graph transformations e.g., eMoflon [9], bidirectionalization e.g., BIGUL [13], constraint solving e.g., Echo [12] and can be used in different areas of application [10].

1.1 Problem Statement

Bidirectional transformation is an emerging concept. In the past, many efforts have been made by conducting international workshops, seminars and through experiments conducted by developers / bx community to identify its potential. Also, in addition to the development of bx tools and bx language, benchmarks are being created for bx tools for systematic comparison [4].

Although a significant amount of work has been done on bx but a general awareness and understanding for basic concepts, the involved challenges, and reasonable expectations is not really given. Hence, there exist conceptual and practical challenges with building software systems using bx-tools and as a result bx tools and their applicability is still not widely known and used [3].

From experience with working with master students (future software developers in industry), we have identified a set of bx learning goals that can be targeted to improve the situation. Achieving these learning goals is, however, challenging as current possibilities (virtual machines, handbooks, etc) are either tool specific or ineffective because installation process to get the tool running is a time consuming process and sometimes requires technical expertise in a specific area/tool/programming language. In this fast paced world, people don't really want to spend much time in a process if it is time consuming and involves complexity. Even after you get the tool running, it doesn't help in understanding the bx concepts and its corresponding technological space in use because of lack of proper explanations.

1.2 Contribution

To solve the problems as described in Section 1.1, in this thesis, my goals are as follows:

- Design and implement an interactive demonstrator.
- Spread the basic concepts of bx to a wide audience and making them accessible and understandable.

An existing bx tool will be used as a part of the demonstrator to realize *bidirectional transformation*. The final prototype will be interactive and easily accessible to users to help them

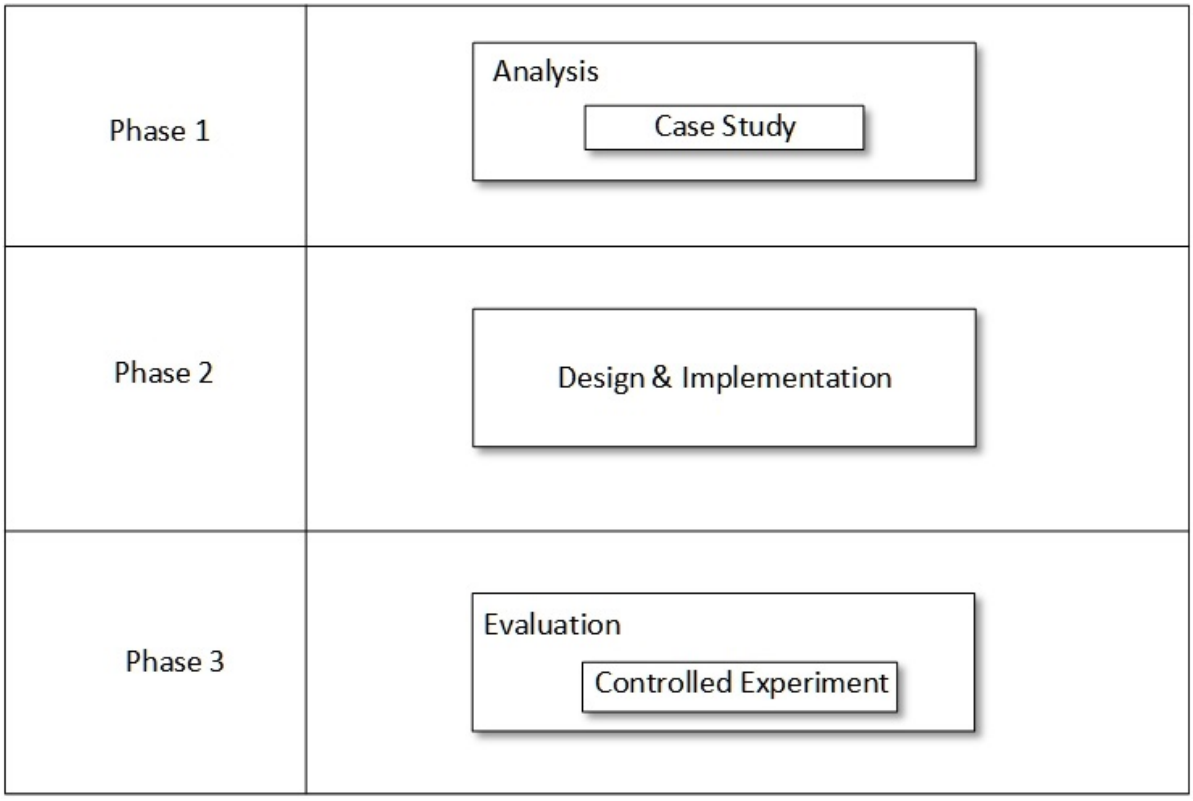

Figure 1: Approach Overview

understand the potential, power and limitations of bx.

My approach for providing a solution to the problems described in Section 1.1 consists of various stages which focuses on designing and implementing a successful bx tool demonstrator as shown in figure 1.

Analysis This stage is based on the research method called Case Study [18]. In any research, there could be many cases and each case could focus on a number of different research questions, each of which leads to a different direction in developing solution strategies [18]. Hence, this method aims at selecting cases that are most relevant to the research and research questions are pinned down to the exact problems in hand.

Initially, the case study was based on the existing work/research done on bx, existing tools available in the market, flexibility in usage, time and technical expertise required to use these bx tools, and the implementation of these tools in different areas. With the initial study and knowledge gathered, I had formulated the following associated research questions (referred to as **RQ** from now on):

- ***RQ1*** – What are the core requirements for implementing a successful bx demonstrator ?
- ***RQ2*** – What kind of interactivity and to what extent is it required in the bx demonstrator ?
- ***RQ3*** – Which goals can be particularly well addressed in a bx demonstrator and why ?
- ***RQ4*** – To what extent is such a bx demonstrator reusable?
  RQ4 can be split into the following sub-questions:
  RQ4.1 – Is the implementation of the demonstrator bx tool-specific ?
  RQ4.2 – Is the implementation of the demonstrator example-specific?
- ***RQ5*** – Is it possible to teach the concepts of bx through a demonstrator ?
- ***RQ6*** – Does an interactive GUI helps an user to increase his/her understanding of bx concepts ?

All of my work is directly or indirectly related to the above research questions.

Design and Implementation This stage is consist of all the steps related to designing and implementing the demonstrator by keeping a focus on the research questions.

First, I have constructed a few examples which can be implemented covering the requirements and showing the usability of bx tools through demonstrator. Then, taking account the availabilities of resources and usability factor, I finally chose the best suitable example to implement and build the final prototype. Section 5.1 describes list of all the examples.

Second step was to choose a bx-tool for my demonstrator. Based on the gathered information and taking account implementation related issues, I have chosen a bx-tool to be used as a part of the demonstrator to realize bx. Section 5.2 explains the process in detail.

Next step was to set up the entire application framework for implementing the demonstrator. First, I did some research by going through materials on software design patterns and web application architecture. Then, I prepared a few proof of concepts(POC) for checking the feasibilty of the architecture designs before finalising my application framework. Section 5.3 explains the process in detail.

Evaluation This stage is based on the research method called Controlled Experiments [18]. This experiment is based on one or more hypothesis, which guide all steps of the experimental design, deciding which variables to include and how to measure them. So, it is an investigation of one or more hypothesis where one or more independent variables are manipulated to measure their effect on one or more dependent variables. Learning goals were prepared along with the scenarios to explain them and questions were asked on these concepts to different groups of

participants. Finally, data is collected and analyzed to measure the outcome. Section 7 explains the process in detail.

1.3 Structure of Thesis

This document is structured as follows:

Chapter 1 (introduction) contains the introduction and motivation about the thesis with a solution strategy. Chapter 2 discusses the related terminologies with respect to bidirectional transformation. Chapter 3 describes the requirements for implementing a successful bx demonstrator. Chapter 4 explains the related work that has been done on bx in last few years and their related problems. Chapter 5 describes all the design decisions and its associated challenges during the implementation work. Chapter 6 provides in-depth explaination of the implementation details done in each layer along with UML diagrams. Chapter 7 contains the learning goals(based on research questions), feedback from user groups, and evaluation results . Last chapter summarizes all the work which was done as part of this thesis and draws useful conclusions followed by future work.

2 Foundation

This chapter provides an overview of my running example in Section 2.1 followed by definitions of some commonly used terminologies with respect to bx in Section 2.2. This chapter will lay the foundation for understanding the basics for the reader and will help him/her in apprehending the concepts explained in further chapters.

In this thesis, bidirectional transformation will be discussed by referring to a *Kitchen Model* and a *Grid Model* of a software system. Both the models describes the structure and behaviour of the real system "kitchen" but from different perspectives.

2.1 Running Example

This example is a simplified kitchen planner. Figure 2 describes the relation between the *Kitchen Model* denoted by *Kitchen* (right-hand side canvas) and the *Grid Model* denoted by *Layout* (left-hand side canvas) from GUI point of view.

Kitchen contains items e.g., sink, fridge, table etc. You can create, delete or move these items in the kitchen space, and press "Sync" to propagate your changes to the layout. The *Layout* consist of groups with certain number of blocks. This shows how much space the objects occupy as coloured groups of blocks organised in a grid. Here, the *Kitchen* corresponds to the

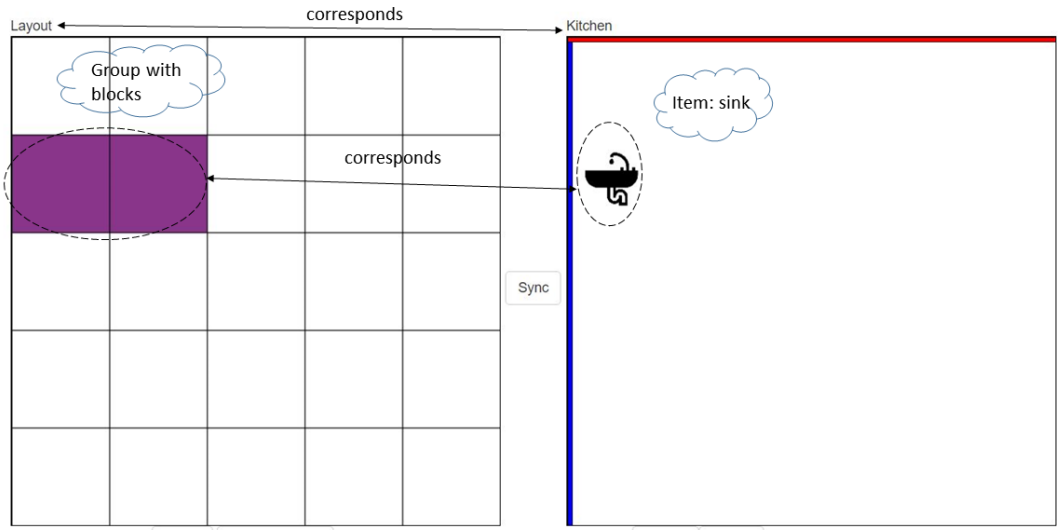

Figure 2: Running Example: GUI

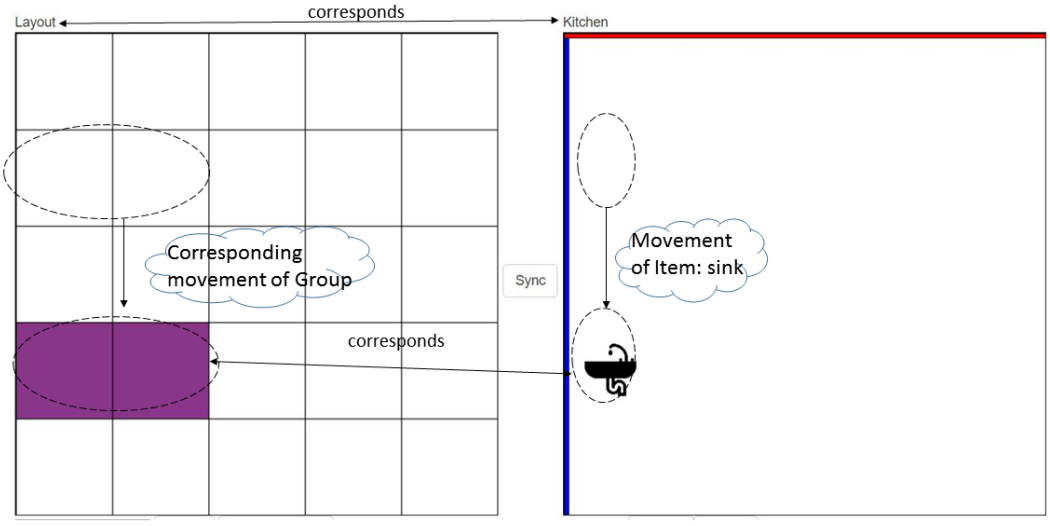

Figure 3: Running Example: GUI (consistency preservation)

Layout and a *Item* corresponds to a *Group*. This is described by Figure 2 where an item e.g., *sink* created in the *Kitchen* corresponds to a group in the *Layout*.

Both artifacts are created and evolved together during the lifecycle of the application representing a kitchen workspace throughout. Thus, changes in one domain should be propagated to the other domain in order to ensure consistency between these related artifacts. This phenomenon is showed by Figure 3.

Model transformation and synchronization is discussed with this scenario throughout this thesis.

2.2 BX Basics

Definition 1 *(Model and Meta-Model)*

A model depicts the structure and/or behavior of a real system under discussion from a certain point of view and at a certain level of abstraction which helps in managing and understanding the complexities of a system [19] [20]. It is denoted as "M". Model creation helps in keeping a clear focus on selected concepts and rules relevant for a particular concern and omitting irrelevant details.

The set of concepts and rules used for forming models, is specified by a metamodel. Hence, a metamodel can be defined as a model's model [20].

Example: In my demonstrator, the example that I have implemented has two models i.e., *Kitchen* and *Grid*. Both the models represent the reality "Kitchen". A relation between reality and models is shown in Table 2.2. Whereas, figure 4 shows an abstract view of the *Kitchen* model and figure 5 depicts a concrete example of *Kitchen* model.

Definition 2 *(Modeling)*

Modeling is the process which allows a developer to specify, visualize and capture a variety of important characteristics of a system in corresponding models which helps in decision making during development stages [19]. Nowadays, Unified Modeling Language (UML) [19] is used to visualize models in an object-oriented fashion.

Example: In my demonstrator example, the process of identifying and constructing the "Kitchen Model" with the characteristics that is required for the task is an example of modeling. *Kitchen* model contains three classes i.e., *Kitchen*, *Item Socket* and *Item*. *Kitchen* class contains *itemSockets* and a *Item Socket* class contains an *item*. Figure 4 shows an abstract view of the *Kitchen* model.

Reality	Model	Aspects Covered
Kitchen	Kitchen Model	• Area of a kitchen as a white space • 4 Walls of a kitchen • contains the objeccts of a kichen • Creation, Movement, Deletion of kitchen objects
Kitchen	Grid Model	• Area of a kitchen as a block structure • 4 Walls of a kitchen • contains the objeccts of a kichen • Creation, Deletion of kitchen objects

Table 1: Model and Reality

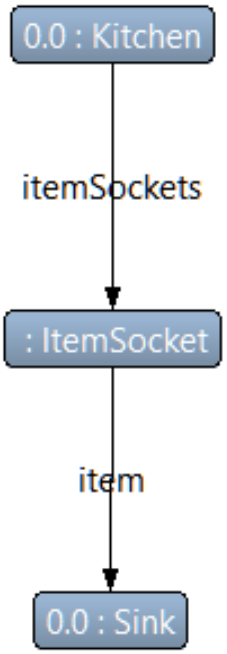

Figure 4: Abstract Kitchen Model

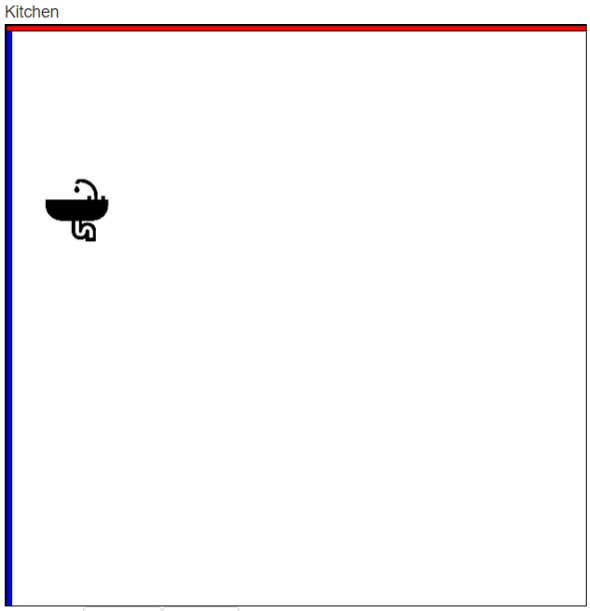

Figure 5: Concrete Kitchen Model

Model	**Delta (δ)**
Kitchen Model	• Creating a new Item • Deleting an existing Item • Moving an Item

Table 2: Examples of Delta in Kitchen Model

Definition 3 *(Delta)*

Delta is the change done to one or more properties of an artefact. It denotes the relationships between models from the same model space [6]. It is denoted δ: M $\longrightarrow$ M' where M' is an updated version of M.

Example: In my demonstrator example, deltas related to the "Kitchen Model" are described in Table 2.2. Whereas, figure 6 shows an abstract view of delta propagation and figure 7 depicts a concrete example of delta propagation, where creation of a new item e.g., sink causes the updation from Kitchen to Kitchen'.

Definition 4 *(Structural Delta)*

Structural Delta denoted as "s-delta" is the collection of changes done to a model to update it,

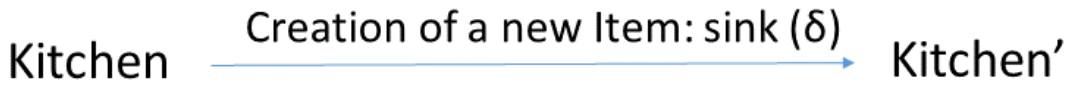

Figure 6: Delta Propagation (Abstract Example)

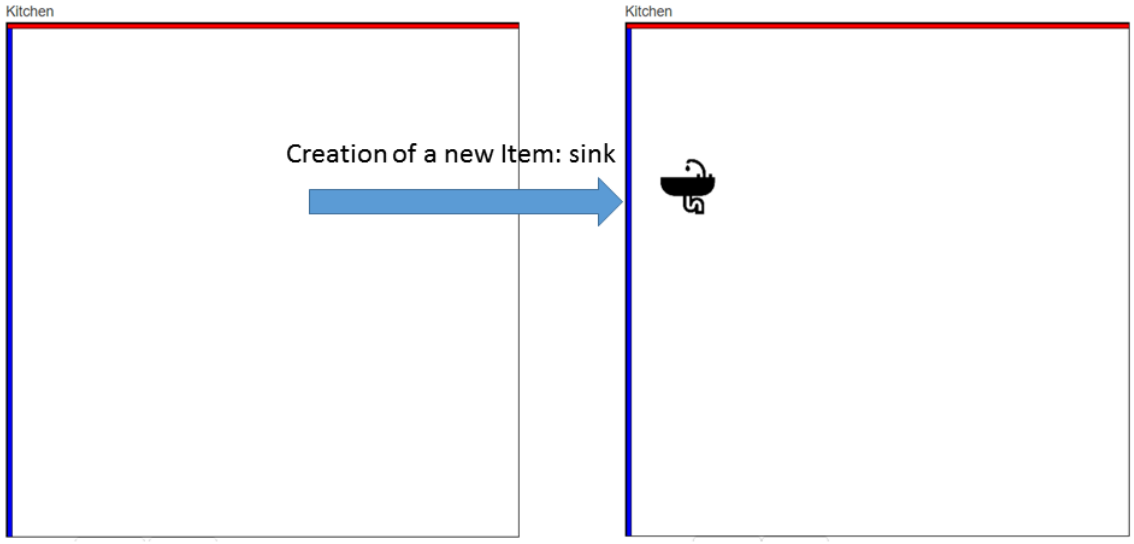

Figure 7: Delta Propagation (Concrete Example)

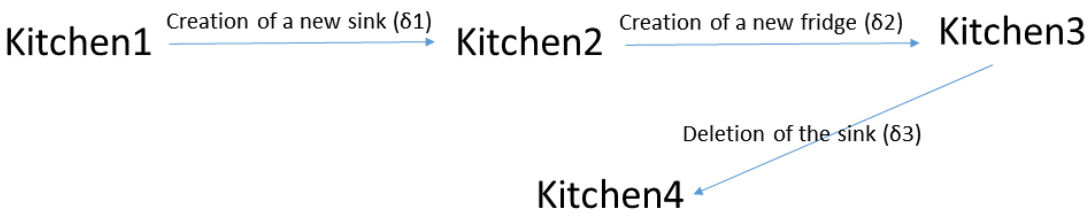

Figure 8: Model Space

which is compatible with structure [6]. The way the changes are applied to the original model to result in a updated model (models of the same model space) don't necessarily have to be in the order in which they are performed.

Definition 5 *(Operational Delta)*

Operational Delta denoted as "o-delta" is the collection of operational specification changes done to a model to update it [6]. It is important that the changes are applied to the original model to result in a updated model (models of the same model space) have to be in the order in which they are performed.

Definition 6 *(Model Space)*

Model Space describes all the states of an artefact and all the deltas which lead from one state to another.

Example: In my demonstrator example, an concrete example of a model space is shown in figure 8 with all the states i.e., Kitchen1, Kitchen2, Kitchen3, Kitchen4 along with its deltas i.e., $\delta 1$, $\delta 2$, $\delta 3$ of the *Kitchen* Model.

Definition 7 *(Correspondence Links)*

Relationships between models from different model spaces are called correspondence links, or just corrs [6]. A corr is a set of links r(a; b) between elements (a in A, b in B), which are compatible with the models' structure and denoted by double bidirectional arrows, e.g., R: A $\Longleftrightarrow$ B.

Example: In my demonstrator example, an example of a corr is r(group; itemSocket) where *group*, *itemSocket* are the elements of *Grid* and *Kitchen* model respectively. Figure 9 shows a

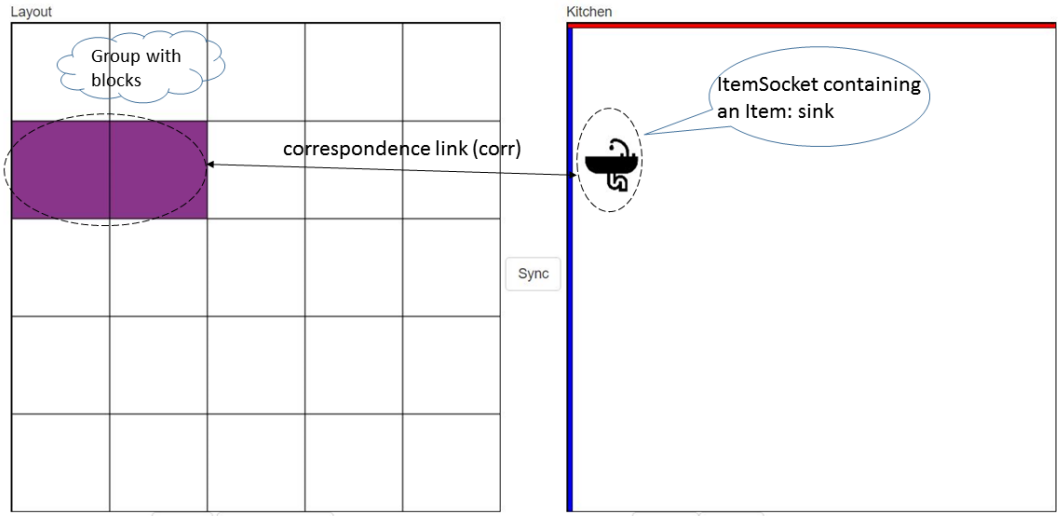

Figure 9: Correspondence Links

concrete example of the corr.

Definition 8 *(Transformation)*

Transformation is the process in which takes one or more source models as input and produce one or more target models as output (models from different model spaces) expressed in either same abstraction level or in different abstraction level following a set of transformation rules [21].

Definition 9 *(Forward Transformation)*

Forward Transformation is the transformation from the *Source* model to the *Target* model.

Definition 10 *(Backward Transformation)*

Backward Transformation is the transformation from the *Target* model to the *Source* model.

Example: In my demonstrator example, "Grid Model" is the *Source* model and "Kitchen Model" is the *Target* model. Forward transformation will cause "Grid Model" to transform

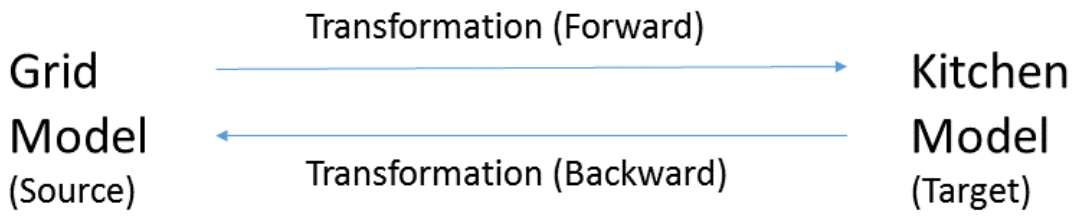

Figure 10: Transformation (Abstract Diagram)

into "Kitchen Model" and Backward transformation will cause "Kitchen Model" to transform into "Grid Model". Figure 10 shows an abstract example of the transformation process and figure 11 depicts a concrete example of the transformation process describing both forward and backward transformation.

Definition 11 *(Consistency)*

Changes in one model may or may not cause any change in other model (models from different model spaces) but their states must not contain any contradiction. It is checked on the set of all corrs related to both the models, R: A $\Longleftrightarrow$ B, i.e., the corr R is consistent, or R is inconsistent [6].

In layman's term, consistent is a state which always involves 2 or more objects/artefacts but doesn't involve ambiguity between them. This is the most important part of the model transformation and the entire focus revolves around it. To be consistent with each other the models have to be inline with respect to their states.

Example: In my demonstrator example, an example of a corr is r(group; itemSocket) where *group*, *itemSocket* are the elements of *Grid* and *Kitchen* Model respectively. Figure 2 and Figure 3 shows that changes in *Kitchen* model i.e., movement of sink should be propagated to the *Grid* model in order to ensure consistency.

Definition 12 *(Unidirectional vs.Bidirectional Transformation)*

Unidirectional is the simplest one out of all kinds of transformation. It always takes one type of input and produces same type of output. The concept of consistency is very simple as the input model is consistent with the output model, only.

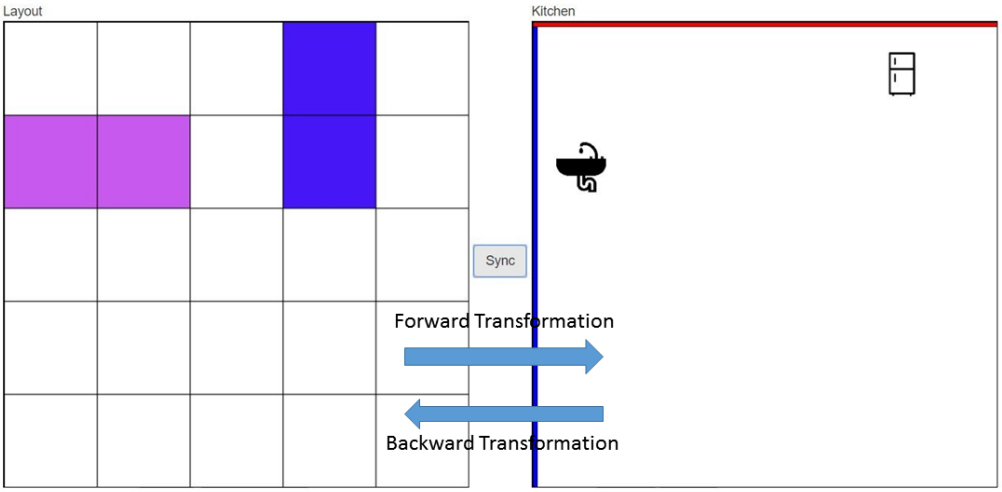

Figure 11: Transformation (Concrete Diagram)

Whereas, bidirectional transformation is a pair of transformation which takes place in both forward and backward direction. One model can be input sometimes and output at some other time. The concept of consistency is relatively complex as more than one model and the models must be kept consistent [17]. Source model (M_S) is transformed to target model (M_T) in forward transformation and vice-versa in backward transformation. It is denoted by "BX".

Example: In my demonstrator example, figure 12 shows an abstract example of the "BX" and figure 11 depicts a concrete example of the "BX" process where a *Grid* model and a *Kitchen* model is described as source and target respectively in the process of forward and backward transformation.

3 Requirements

This chapter explains all the requirements needed to implement a successful demonstrator. Section 3.1 describes all the functional requirements. Afterwards, Section 3.2 deals with all the non-functional requirements followled by Section 3.3, which describes some of the choices that I made during research and implementation work and its associated threats.

To avoid failure and for the smooth functioning of any project, the basic and foremost need of any project is requirement. Generally requirements are of two types: functional and non-

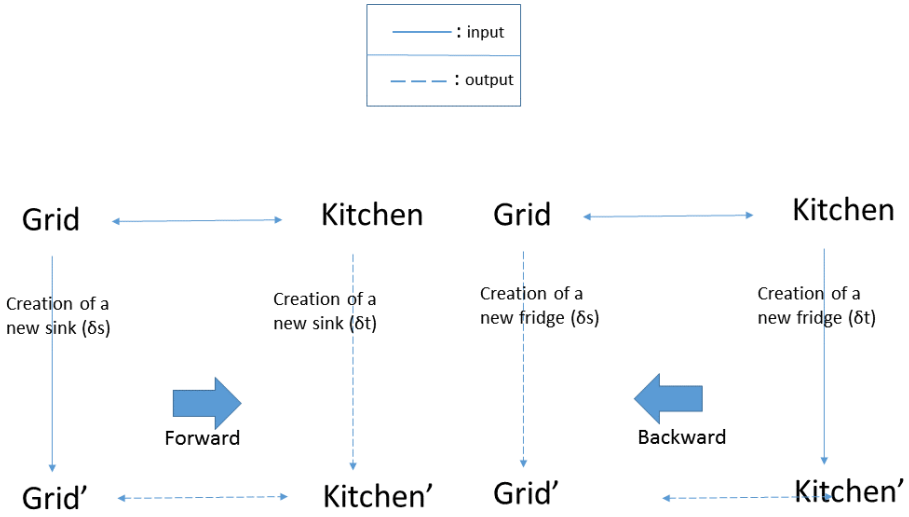

Figure 12: Bidirectional Transformation

functional.

3.1 Functional

Functional requirements define the behaviours of a system or in simple terms what a system should do. Hence it describes all the behavioural attributes of a system [35].

As my thesis is more focussed on the implementation of a demonstrator for consistency management based on a bx tool, following are some of the functional end results (referred to as **FREQ** from now on) I am trying to achieve:

FREQ1: Reduce the installation time of the bx tool
The demonstrator should not take much time to install and should be up and running very fast. By making the demonstrator available online, user needs nothing to install on his/her machine. User have to enter the url on the browser and hence just a click away from trying the demonstrator based on a bx tool.

FREQ2: Demonstartor should be fun to play with.
The demonstrator should be able to attract more users by exploiting the features of the GUI and colors. It should be interactive and fun to play with during the entire process so that the user

sticks to it.

FREQ3: Demonstrator's example should be simple
The user is going to try/use the demonstrator based on an example. This example should rather be less complex and less technical to understand to create interest and convince more target audience. Also, user should be able to relate to the bx concepts through the example.

FREQ4: Demonstrator should be able to guide the user
During the entire time while the user is using the demonstrator, it should be able to provide clear instructions/guidelines on what the user can do and try with it. User should not feel like what to do with the demonstrator after a few minutes. Demonstrator should guide the user with the all the aspects that is intended to be taught/learned with it.

FREQ5: Demonstrator should be informative
Rather being just a online playing tool, the user should learn certain concepts of bx by using the demonstrator. User should be able to relate to the concepts taught by the demonstrator with its scenarios.

3.2 Non-Functional

Non-functional requirements cover all the remaining requirements which are not covered by the functional requirements. Hence it describes all the quality attributes of a system [35].

Following are some of the non-functional end results (referred to as **NREQ** from now on) I am trying to achieve:

NREQ1: Public access
The demonstrator should be accessible world-wide to all. Being a demonstrator for a bx tool, it's accessibility shouldn't be restricted for a few groups. rather, it should be accessible to all whoever wants to use the tool.

NREQ2: Platform independent
Application should be platform independent i.e., it should run on any operating system and all kinds of browser. Different environment on user's machine should not affect the application.

NREQ3: Robost
Application architecture should be easily maintainable and extendable. With changing requirements and needs, application architecture should be able to accomodate new changes and future enhancements.

NREQ4: Easily deployable project

Application's deployment process on the web-server to get it running should not be complex. Rather, it should involve simple steps so that it can be carried out by anyone.

NREQ5: Products, technologies and tools
Updated version of the products, technologies and tools should be used as per the requirements during the implementation work so that future enhancements can ce carried out easily.

3.3 Choices and Threats

The implementation process and the final prototype was driven by many choices and threats. For example,

- selection of the bx tool and the most suitable example to implement has impact on deciding the usefullness of the demonstrator.
- seletion of the bx tool and the example to implement is more or less influenced by the ideas given by my supervisor.
- my decisions on designing the application's framework for implementation are impacted by the existing availability and usability issues of the finalized bx-tool.
- During evaluation, I might not have got proper feedback from my friends & colleagues due to my acquaintance and time constraints.

4 Related Work

This chapter sums up all the related work that has been done on bx. Section 4.1, 4.2, 4.3, and 4.4 describe the various ways that bx community and developer's group have tried to make the work done on bx visible to the world. Then, Section 4.5 explains the related problems.

Model transformation is a central part of Model-Driven Software Development [1] [2]. Bx community has been constantly doing research and development work in many fields to help people understand and increase awareness about bx. Nowadays, researchers from different areas are actively investigating the use of bx to solve a variety of problems. A lot of work has been done in terms of building usable tools and languages for bx. These tools can be used in various fields, for achieving *bidirectional transformation*. To understand these tools, several handbooks, tutorials and examples have been created so that users and developers can understand the core concepts. Following sections will describe these concepts in detail.

Figure 13: Web GUI for Demonstartor: Biyacc

4.1 Existing Demonstrator

First, I analyzed an existing demonstrator available along with the test cases of a domain-specific language, BiYacc [15] which is based on BiGUL [13]. BiYacc designed to keep the parsers and printers needed by the language designers unified, in a single program, and consistent throughout the evolution of a language. Based on bidirectional transformations theory, BiYacc constructs the pairs of parsers and reflective printers and gurantees that they are consistent.

Figure 13 shows the web interface of the demonstartor. It contains two views i.e, source and view. Depending upon the user selection, "souce" contains an arithmetic expression or a program with some comments. After the forward transformation is done, "view" is loaded with the parsed version (machine readable format) of the sources' text.

Being an online demonstrator, a user can try it out instantly and check how it works. It doesn't require any installation or technical expertise to get the example running.

4.2 Virtual Machines

Then, I analyzed a web-based virtual machine, e.g., SHARE [16]. Basically, this is a web portal used for creating and sharing executable research papers and acts as a demonstrator to provide access to tools, softwares, operating systems, etc., which are otherwise a headache to install [16].

This provides the environment that the user requires to execute his/her tool or program. Hence, it reduces the overhead of a user for maintaining and organizing all software framework related stuff and simplify access for end-users.

4.3 Handbooks & Tutorials

As a part of my research, I also analyzed some tutorials and tools and below are my findings.

Anjorin et al.[9] present the concept for *bidirectional transformation* using Triple Graph Grammars (denoted by "TGG") [7]. To demonstrate their core idea and the usage of the tool, they have described an example by transforming one model (source) into another (target) through TGG transformations [7][8]. The whole tutorial is about 42 pages long which guides the user to get the example running through a series of steps. These steps include installing Eclipse[1], getting their tool as an Eclipse plugin, setting up the workspace, creating TGG schema and specifying its rule and much more. If the user is able to execute each step correctly, then finally he/she can view the final output. It took me 4 days to get the tool up and running.

We have analyzed a tutorial[14] on a bidirectional programming language BiGUL[13]. The core idea with BiGUL is to write only one putback transformation, from which the unique corresponding forward transformation is derived for free. The whole tutorial is about 45 pages long which includes a lot of complex formulas, algorithms, and guides the user to get the example running through a series of steps. These steps include installing BiGUL, setting up the environment, achieving bx through BiGUL's bidirectional programming and much more. If the user is able to execute each step correctly, then finally he/she can view the final output.

[1] *Integrated Development Environment (IDE) for programming Java*

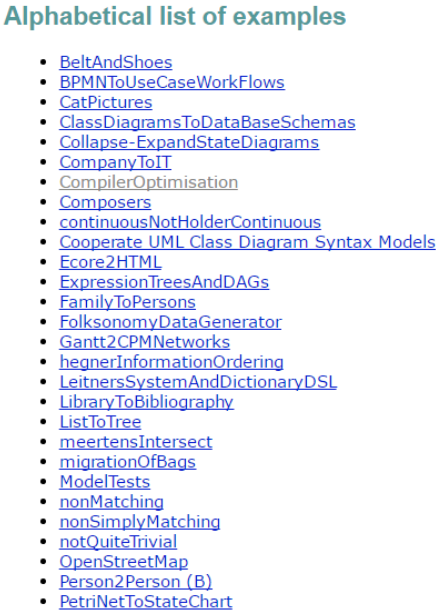

Figure 14: BX Example Repositories

4.4 Example Repositories

A rich set of bx examples repository [11] has been created based on many research papers. These examples cover a diverse set of areas such as business process management, software modeling, data structures, database, mathematics and much more.

Figure 14 shows a screenshot of the example repositories listed alphabetically. A user can find relevant information about the examples on the respective web pages. Some of the examples are very well documented along with class diagrams, activity diagrams, object diagrams etc. and source code of a few examples are available as well.

4.5 Discussion

Some associated problems that I found with the above paragraphs are as follows:

Existing Demonstrator The existing demonstrator's visual representation doesn't create interest in the target audience as it does not exploit the potential of using an interactive GUI and colors, etc. It just makes use of two text fields and is comparable to a console-based interface that is accessible online.

There is very limited guidance provided concerning what the user can do and try out with the

demonstrator. Especially for non-experts, it is not clear what to do with the demonstrator after a few minutes.

The existing demonstrator is based on a rather technical example that might not be relevant, interesting, or convincing for a large group of potential bx users.

Virtual Machines For security reasons, virtual machines are not like other web portals where a user need to simply sign-up and can host/create/access data, rather it includes a series of request-grant cycle for getting access to an environment and hosting/managing data. Also, some actions require special authorization and take time to complete the whole process.

User needs at least 3 Gigabyte or more space on his/her local system for configuring the virtual machines depending on the requirement of the environment, which sometimes creates an overhead.

Handbooks & Tutorials As described in handbooks and tutorials, Installation requires technical expertise and time consuming as the user typically has to setup and install the tool.

Required knowledge is too tool/area specific and it can be challenging if the user is not familiar with the technological space, e.g., User must possess knowledge about Java and Eclipse framework or a Java programmer might find the tool chain for Haskell unfamiliar.

Steps to get the example running needed technical expertise, e.g., In some cases, domain specific knowledge includes mathematics and specific coding language. What is showcased and discussed is tied to a specific technological space and might not be easily transferable to other bx approaches.

Not very helpful to understand what bx is before deciding which bx tool (and corresponding technological space) to use.

Example Repositories In today's fast paced and visually enriched world, "what you see is what you believe" and that is exactly what these examples are lacking in. None of the examples have a demonstrator showing how it works. Hence, it is very difficult for a user to realise the examples just by going through the documentation and UML diagrams. Even if source code is present, it takes a lot of time and requires technical expertise to set-up the framework and get the example running.

5 Design

In this chapter, I am going to describe the design steps realized during the implementation of the demonstrator. Section 5.1 describes all the examples that I have conceptualized before choosing the final one for the implementation. This is then followed by the description of the steps taken for selecting the bx-tool in Section 5.2. Afterwards, Section 5.3 deals with the decisions taken for finalizing the apllication's architecture design. Then, Section 5.4 describes the applications' framework in detail along with its components. At last, Section 5.5 describes the challenges that I faced while designing the entire framework and its components.

5.1 Choosing an Example

To solve the problems as described in Section 1.1, the main idea is to design and implement an interactive bx tool demonstrator based on an intuitive example.

5.1.1 Construction

I have constructed a few examples for implementation as follows:

Task Management This prototype can be used for allocating tasks in a team. It contains two views e.g., supervisor's view and employee's view. A Supervisor can allocate tasks to their subordinates. An employee can view the tasks assigned to him. Then the task will go through a life cycle as the work progresses, i.e., Assigned, In Progress, Testing, Done. Supervisor's view shows aggregate information from multiple projects and multiple employees, but does not contain detailed information, e.g., tasks have fewer states than for assigned employees. Bx rules control how updates are handled and states are reflected in the different views of the project, e.g., the employee's view will be updated for each state change, whereas the supervisor's view is only updated when a task is completed and not for intermediate changes.

Quiz This prototype can be used for an online quiz game. It contains two views e.g., administrator's view and participant's view. There will be a large set of questions related to different areas, e.g, history, geography, politics, sports, etc. The administrator can select the areas from which the questions will be shown to the participant and initiate the game. The participant can override the selection of the areas and start the quiz. Randomly questions will be shown to the participant from the selected areas with 4 options. The administrator's view contains less information than the participant's view, e.g., only the result of each question will be shown to the administrator, whereas participant can see questions along with its options. As soon as the

participant chooses the answer to any question, bx rules control how updates are handled and states are reflected in the different views of the project.

Playing with Shapes It contains two views e.g., low-level view (depicts UI[2] for low-level language, i.e., UI with less functionality) and high-level view (depicts UI for high-level language, i.e., UI with more functionality). User will draw a geometric shape, i.e., triangle / square / rectangle / circle with some notations similar to the shape on the low-level view and if the notations are correct, the high-level view tries to recognize the shape and draws it with default parameters and vice-versa. Basically the transformation will happen between a low-level language and a high-level language and bx rules control how updates are handled and states are reflected in the different views of the project. In high-level view, more functionalities will be present, i.e., moving one shape from one place to another, creating a clone of an existing shape, etc. which is not possible in low-level view.

Arranging a Kitchen It contains two views e.g., low-level view (depicts a grid structure containing blocks) and high-level view (empty space which depicts UI for kitchen). High-level view has more functionalities such as creating/ deleting/ moving an kitchen item, etc. out of which only a few will be available in low-level view. User will create/ delete/ move a kitchen item, i.e., sink / table on the high-level view and if changes done on the high-level view are according to the rules defined in the bx tool then items will be reflected on the low-level view with same colored blocks and vice-versa. Basically the transformation will happen between a low-level language and a high-level language and bx rules control how updates are handled and states are reflected in the different views of the project.

Person and Family It contains two views e.g., Family view and Person view. In Family view, it contains many families and each family consist of members. Whereas, Person view contains persons (the members of each family). We assume that the surnames are unique and allow us to differentiate between different families. Addition of a new person to the Person view will be reflected on the family view and vice-versa. Also, due to the uniqueness of the surnames, person created will be automatically assigned to the related family. Bx rules control how updates are handled and states are reflected in the different views.

5.1.2 Selection

Selecting an example to implement through the demonstrator was not random, rather I have taken many factors into considerations before choosing the final one. It was a very important decision, as selection of the example and its implemenation will directly effect the research

[2] *short for User Interface*

questions *RQ2*, *RQ4.2*, *RQ6* described in Section 1.2 and requirements *FREQ2*, *FREQ3* described in Section 3.1.

Hence by taking into account all these factors, I have finally chosen the *Arranging a Kitchen* example to be implemented by the demonstrator as a part of my research. Following were the driving factors for the selection of the example:

- any user can relate to the example very well as everybody is familiar with a kitchen and its environment.
- example is very simple and no technical details are involved.
- interactivity and rules won't be a overhead for the user, rather intuitive.
- associted scenarios are part of day to day life, so user will be able to relate to the learning concepts through the example.

5.2 BX Tool Selection

Next step in the design process was selection of a bx tool which takes care of the bx part of the demonstrator and upon which the entire framework of the demonstrator will be constructed. My gathered information during the case study phase led the foundation for the selection process.

5.2.1 Theory

I further investigated on the existing bx tools from the point of view of practical application and usage of these tools in terms of building softwares. Even after a significant amount of work has been done by developers community and bx community, the main problems are still revolving around below points [3]:

- the conceptual or theoretical challenges, practical challenges associated with using bx, and tool/technology challenges involved with building software systems that supported or exploited bx.
- limited benchmarks for comparison of complete bx solutions.
- no common repository of bx scenarios or problems that can be used to test and evaluate bx solutions.
- there exists tools to support particular bx scenarios but with very limited interoperability and integration.

To focus particularly on the above issues, bx-community had conducted a series of technical workshops at relevant conferences and organised week-long intensive research seminars in the

year 2013 before the *BX 2014* workshop [3]. Some of the main outcome of these seminars are listed below:

- focus on the need of benchmarks and further categorizing them into functional and non-functional ones.
- scenarios for bx were developed based on database, synchronization, model-view architecture etc.
- software tool support for bx was shown in terms of demos which includes tool like eMoflon, Echo etc.

5.2.2 Selection

Again, it was a very important decision, as selection of the bx tool and the implemenation on the top of it will directly effect the research questions *RQ1*, *RQ4.1* described in Section 1.2 and requirements *FREQ1* described in Section 3.1.

The outcome of the seminars as discussed in previous section 5.2.1 led me to concentrate and analyze the bx tools like eMoflon, Echo, BIGUL. Also, I came across the benchmark [5] [6], the first non-trivial benchmark where Anjorin et al. has provided a practical framework to compare and evaluate three bx tools. After analyzing these tools, benchmarks and taking into consideration the reserach questions and requiements, I have finally chosen *eMoflon* as the bx tool to handle the bx part of my demonstrator. Following were the driving factors for the selection of the bx tool:

- sample implementation with framework to implement the eMoflon tool was already available.
- my supervisor/prof. is a core member of the eMoflon community which gave me added advantage of knowing the tool inside out.
- extra knowledge about the tool can be really handy and it actually helped me in solving the implemenation issues/challenges regarding the tool as described in Section**??**.

5.2.3 Benchmarx

A bx benchmark (referred to as **Benchmarx** from now on) is a bx example that has a precise and executable definition of a binary consistency relation on source and target artefacts; an explicit definition of, or a generator for, input artefact elements; a set of precisely defined update scenarios for certain input artefact elements; and a set of executable metric definitions [3].

Anjorin et al. [6] tried to solve the main problem with benchmarking bx tools by creating a common design space, in which different bx tools architecture can be accomodated irrespective of the fact that they can have different input data.

Design Space

5.3 Architecture Design

Last step in the design process was application architecture design which is the most important part of my thesis and also the starting phase of the implementation of the prototype.

5.3.1 Design Decision

I decided to build a web appliaction to address the requirements *FREQ1*, *NREQ1* and *NREQ2* described in Section 3.1 and 3.2.

Web application development has evolved drastically from single page design to very complex layering structures since the beginning of World Wide Web. Many design patterns [22] [23] consisting of different technologies and programming languages were proposed, adopted and implemented in different time to address the demands of customers and users on web. With the rapid changes occuring on World Wide Web, technologies are becoming obsolete and losing its demand day by day. Nowadays, the main focus is on improving the user interaction and allow the developers to build powerful web applications rapidly.

I have analyzed a few design patterns and commonly used architecture designs in todays world by working on a few Proof of Concepts (PoC). Main goal of my thesis is to design and implement a demonstrator based on a bx-tool as described in Section 1.2 and prior to this stage, I have already selected *eMoflon* as my bx-tool in Section 5.2. So, The main idea was to check the feasibility of the architecture and the data flow within its components on the top of the interface provided by the *Benchmarx* (proposed by Anjorin et al. [6]) to access the bx-tool. While working on the PoCs, I came across a few problems such as, maintainability and reusability of code, dependencies of components etc. To avoid these problems and to address the requirement *NREQ3* described in Section 3.1, I finally chose Model-View-Controller (referred as **MVC** from now on) architecture for my application framework. Following were the driving factors for the selection of the MVC architecture [23] [26] :

- it differentiates the layers of a project in Model, View and Controller for the re-usability and easy maintainance of code.

- it splits responsibilities into three main roles which allows the developers to work independently without interfering in each others work and for more efficient collaboration.
- due to the sepration of concern, same Model can have any no.of Views. Enhancements in Views and other support of new technologies for building the View can be implemented easily.
- a person who is working on View does not need to know about the underlying Model code base and its architecture.

5.4 Architecture Layers

Figure 15 shows a very high-level architecture diagram of my prototype. It consist of 3 components e.g., Model, View and Controller. View is responsible for all the graphical user interface management and consist of technologies like HTML, CSS, JavaScript and Jquery. Controller is responsible for event handling and basically consist of the technology Servlet. Model is responsible for all the tasks related to business logic, business rules, data, meta-models, state of meta-models etc. and consist of java classes.

Request-Response Cycle Figure 16 shows a detailed architecture diagram with a complete request-response cycle described below:

1. User interacts with the application on a browser and user actions are sent to the Controller as requests.
2. After accepting the request, Controller decides on how to handle it and send it to the controller helper for further processing.
3. After processing the request, controller helper send it to the Model.
4. Model processes the request and generates the response, which is sent back to the controller through controller helper.
5. After receiving the response, Controller prepares the data and send it to the View.
6. View processes the data and finally the final response is generated and loaded on the browser.

Below sub-sections describes the designing process of each component e.g., Model, View and Controller in detail.

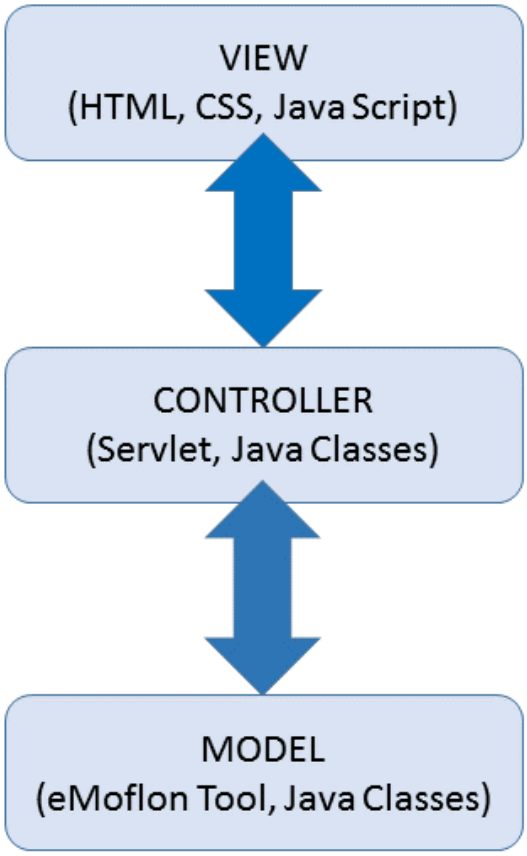

Figure 15: High Level Architecture Diagram

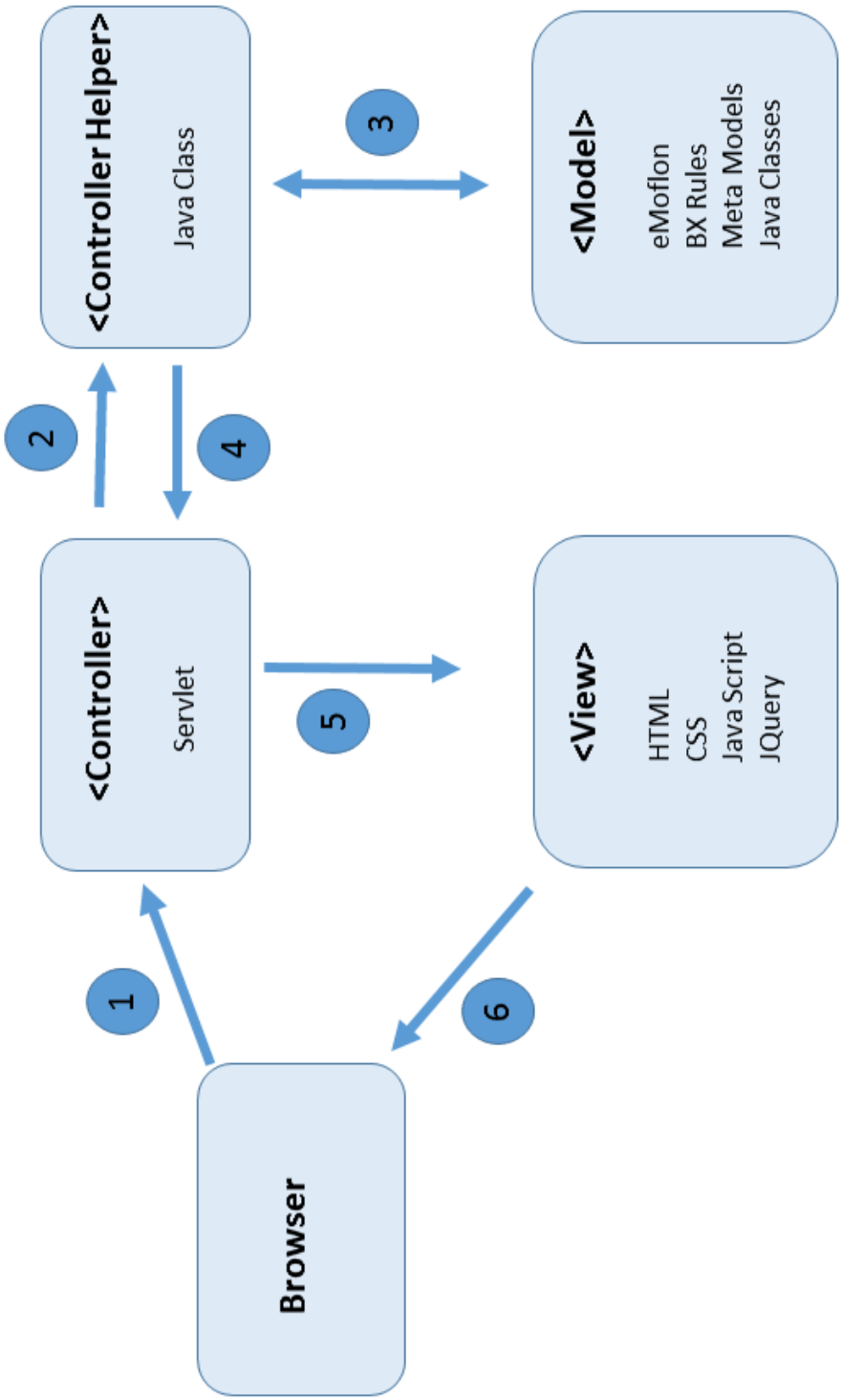

Figure 16: Detail Architecture Diagram

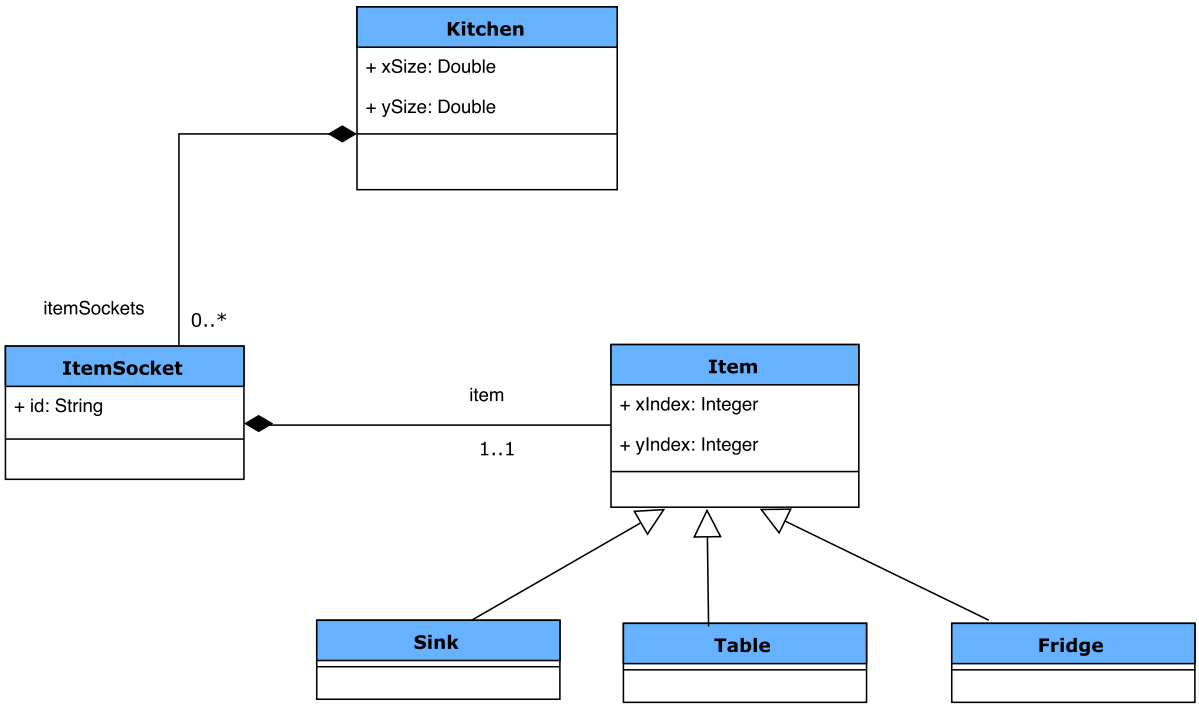

Figure 17: KitchenLanguage Class Diagram

5.4.1 Model

Model is mainly responsible for encapsulating the access of data and handling the business logic of the application [26] [24]. It ensures the data abstraction and provides methods to access it, due to which the complexity of writing the code on developers' part is highly reduced [25].

In my case, Model is consist of java classes and the *eMoflon* tool. The java classes consist of interfaces and its concrete implementation designed to access the bx-tool eMoflon on the top of the framework provided by the *Benchmarx*. The tool contains all the meta-models, state of the meta-models and the associated transformation rules.

eMoflon specific Models For implementing the example *Arranging a Kitchen* through *eMoflon* tool, first task was to create related models inside the tool. The bx tool will keep these models, create and manage instances of these models by applying associted transformation rules as per the requirement. As the example has two different views, it requires two different model in the tool as well i.e., `KitchenLanguage` to represent high-level view and `GridLanguage` to represent low-level.

Figure 17 describes the structure of the `KitchenLanguage` model in a class diagram from an object-oriented point of view. It contains three classes i.e., `Kitchen`, `ItemSocket`, and

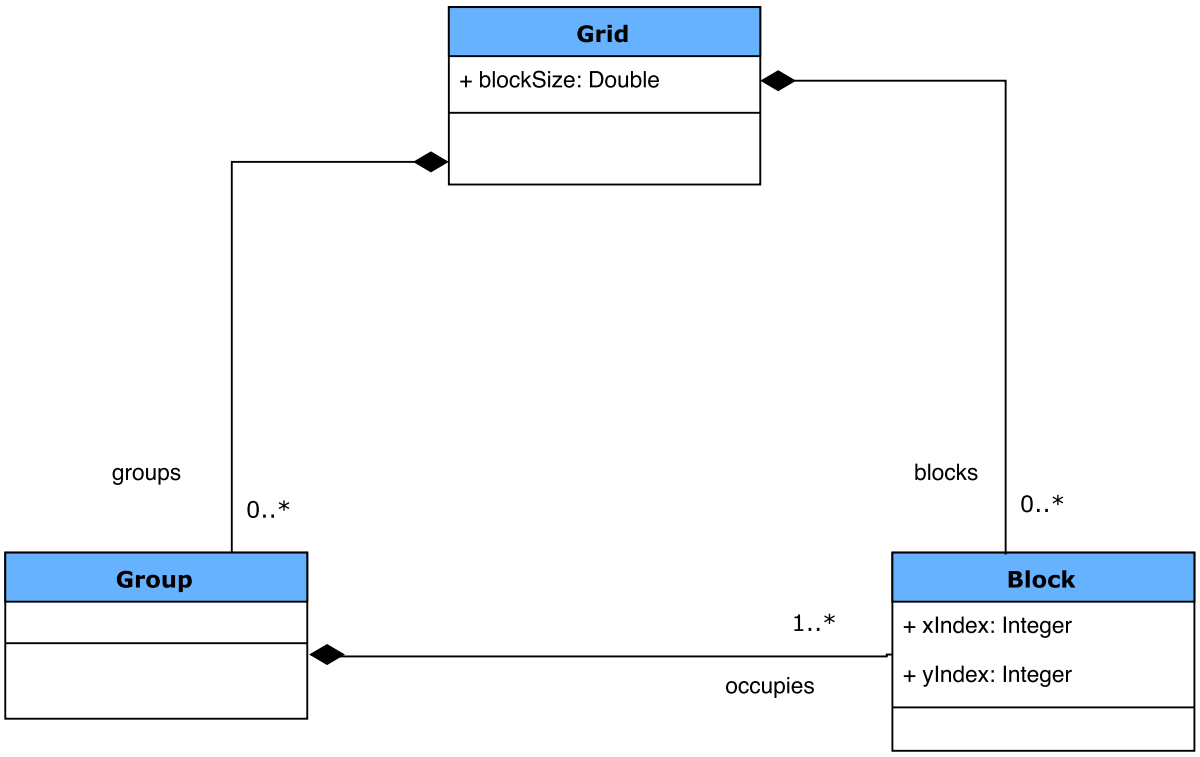

Figure 18: GridLanguage Class Diagram

`Item`. `Kitchen` class has the attributes *xSize* and *ySize* to describe its size and contains zero or more itemsockets. `ItemSocket` class has the attribute *id* and contains exactly one item. `Item` class has the attributes *xIndex* and *yIndex* to describe its position. `Sink`, `Table`, and `Fridge` are different types of items.

Figure 18 describes the structure of the `GridLanguage` model in a class diagram from an object-oriented point of view. It contains three classes i.e., `Grid`, `Group`, and `Block`. `Grid` class has the attribute *blockSize* to describe the size of each blocks contained inside it, contains zero or more itemsockets, and zero or more blocks. Each `Group` class occupies one or more blocks. `Block` class has the attributes *xIndex* and *yIndex* to describe its position and is being occupied by one group.

UI related Models Now, the second step in modeling was to create models for user interface to handle and visualize the bx tool specific models. Thus, UI models are necessary to deal with the user actions and act a connecting bridge between the user actions and the bx tool specific models. While user interacts with the demonstartor, actions i.e., changes performed by the user in the UI will be captured in the form of the UI models and will be sent to the controller helper. Conversion between the UI models and bx tool specific models happens inside the controller helper before sending the request to the bx tool and after receiving the response from the bx

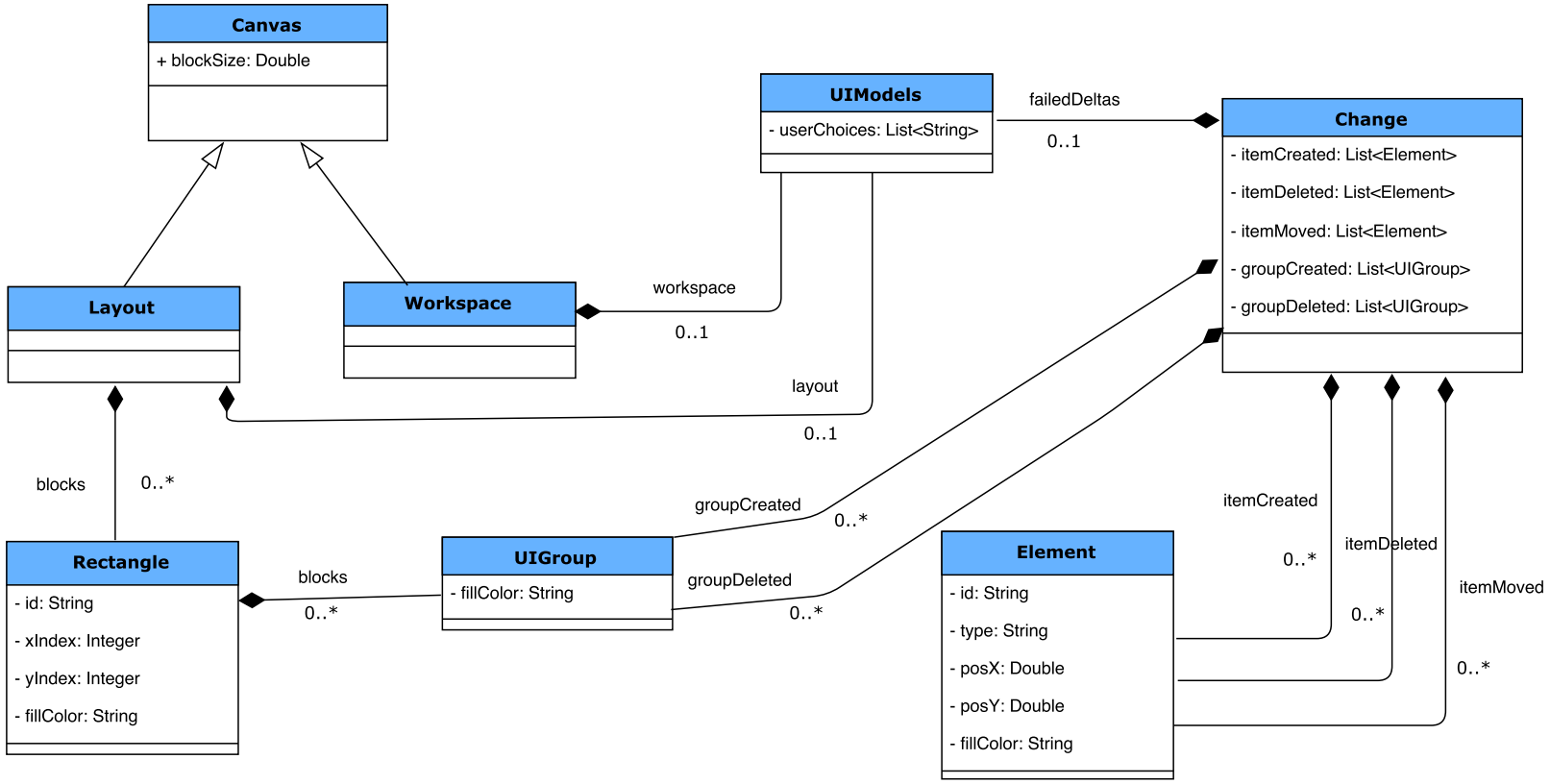

Figure 19: Structure and Relationship between UI Models

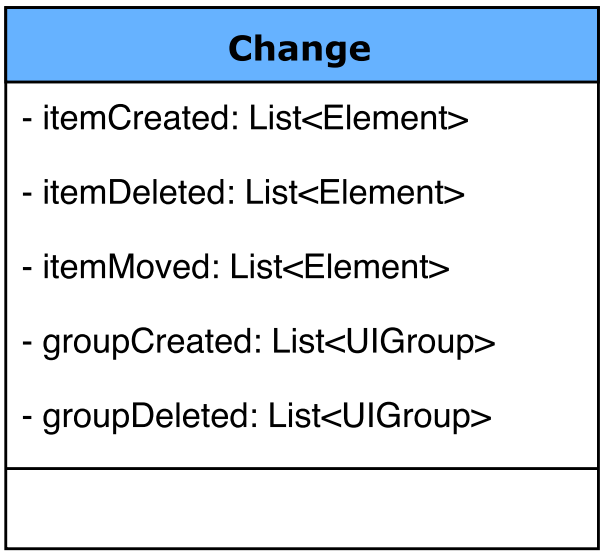

Figure 20: Class Diagram for Change

tool. View always receives the response in the form of UI models and visualize them.

Figure 19 describes the structure and relationship between the models related to user interface in a class diagram. UI models consist of classes like `Canvas`, `Layout`, `Workspace`, `UIModels`, `UIGroup`, `Element`, `Rectangle`, and `Change`. `Canvas` is the parent class for `Layout` and `Workspace`. It has attributes *height* and *width* to describe its size representing both the views. `Workspace` represents the high-level view in the user interface and contains zero or more objects (elements) and an objectTypeList. It represents the eMoflon specific model `KitchenLanguage` on the UI side. `Layout` represents the low-level view in the user interface and contains zero or more blocks and groups. It represents the eMoflon specific model `gridLanguage` on the UI side. `UIGroup` has attribute *fillColor* to uniquely identified as a new group on UI and contains zero or more blocks. It represents the eMoflon specific model `Group` on the UI side. `Rectangle` has attributes *id*, *xIndex*, *yIndex*, and *fillColor* to deal with the UI related actions. It represents the eMoflon specific model `Block` on the UI side. `Element` has attributes *id*, *type*, *xPos*, *yPos*, and *fillColor*. It represents the eMoflon specific model `Item` on the UI side. `UIModels` is the class which carries all the information related to UI. After receiving the response from the bx tool, it is converted into `UIModels` and send it to view for visualization. It has a attribute userChoices to deal with the user options and contains a layout, a workspace, and a set of failedDeltas wrapped inside a `Change` class.

Actions performed by the user on both the views are tracked by the `Change` class. High-level view (Kitchen) allows three actions i.e., creation of a new item, deletion of an item and movement of an item. Whereas, low-level view (Grid) allows two actions i.e., creation of a new group and deletion of a group. Hence, `Change` class track all of these five actions separately.

Figure 20 shows the class diagram of the `Change` class. It contains itemCreated, itemDeleted and ItemMoved as a list of `Elements` to capture creation, deletion and movement of an item respectively in high-level view. It also contains groupCreated and groupDeleted as a list of `UIGroups` to capture creation and deletion of a group respectively in low-level view.

5.4.2 View

The View handles the graphical user interface part of the application. Hence, it contains all the graphic elements and all other HTML elements of the application. View separates the design of the application from the logic of the application due to which the front end designer and the back end developer can work separately without thinking about the errors which could have showed up incase of an overlapping [26] [24]. View controls how the data is being displayed, how the user interacts with it and provides ways for gathering the data from the users.

In my case, the technologies that I am using for View are HTML, CSS and JavaScript and JQuery.

External Design For the visualization of the example *Arranging a Kitchen* as explained in Section 5.1, first task was to design the low-level and high-level views along with its functionalities.

Both the views represent a kitchen area and its certain behaviour. High-level view has more functionalities and flexibility in usage than the low-level view. As both the views are independent of each other and resonates a confined area in which certain task related to animation/graphics has to be performed, I chose *Canvas* [27] HTML element as a container for my views. *Canvas* was the best fit for my views as it provides great support and application for creating animation and drawing graphics on web.

For high-level view, I kept the *Canvas* clean to represent an empty kitchen space where addition and manipulation of different objects such as, sink, table, fridge etc. can be done. Figure 21 shows a sample of the high-level view (Kitchen View). To make the kitchen space more realistic, I have even added water outlet on western wall and electrical fittings on northen wall so that the user can relate to it. For low-level view, I have filled the *Canvas* with grids/blocks. The low-level view represents exact kitchen space as shown in high-level view but, divided into blocks. Also, the blocks restrict certain flexibility and functionality compared to high-level view. Figure 22 shows a sample of the low-level view (Grid View).

Next step was to handle the user interactions in the process of performing various task on both the views. In web development, javascript is the most used language for handling the user intearctions and programming the behaviour of web pages [31]. Hence, I have analyzed a few canvas libraries available in market e.g., Fabric.js [28], Processing.js [29], Pixi.js [30] by working on a few Proof of Concepts (PoC). The main idea was to check the feasibility and

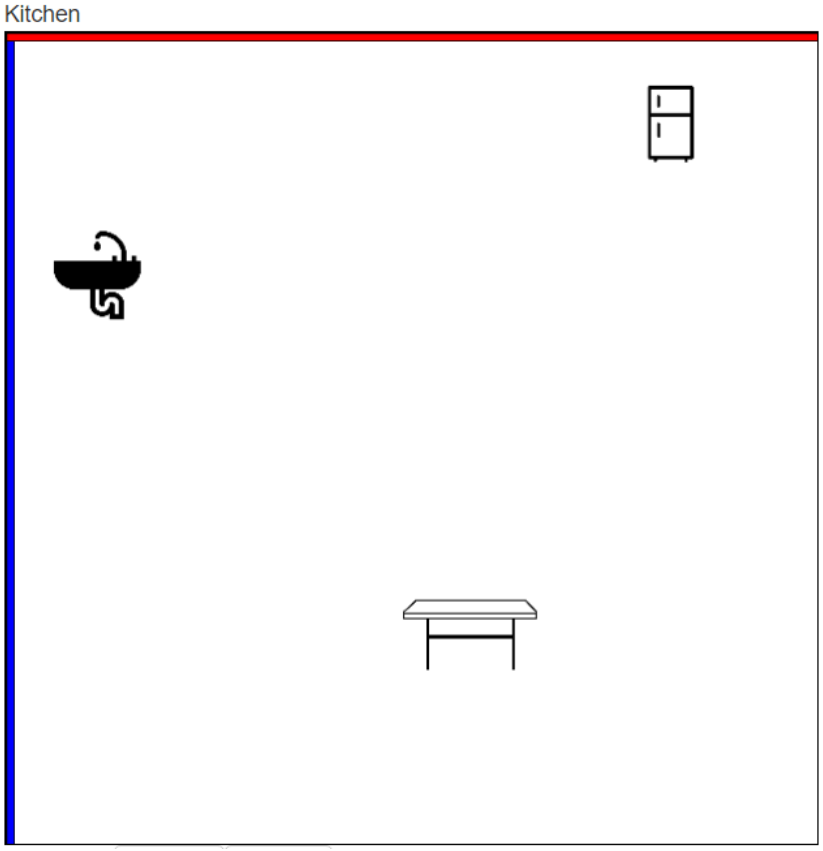

Figure 21: High Level View

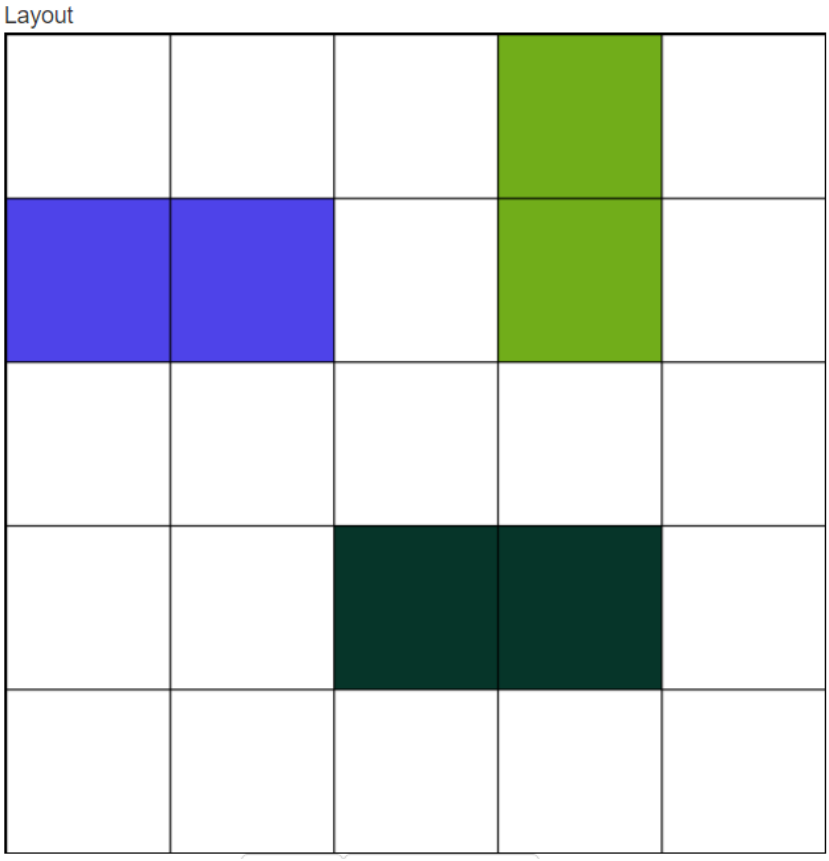

Figure 22: Low Level View

support for interactivity to perform different user defined tasks on the *Canvas*. Finally, I chose Fabric.js as my javascript library for handling the user interactions because of below factors [28]:

- It is good at displaying large number of objects on canvas.
- It handles object manipulation such as, moving, rotating, resizing for any kind of object.
- It has a great support for rendering and displaying object of any kind.

Internal Design Second task in the process of visualization was defining the user actions with respect to both the views, capturing them, and displaying the views after transformation is done.

In high-level view, user can perform addition, removal and movement of the kitchen objects as it is done in everyone's house. A new object can be added and an already existing object can be moved or removed within the empty space available in the high-level view with the different mouse events. To make the example more realistic, I have used similar images of the kitchen objects as shown in figure 21. Every object is tracked on the basis of its position in the view and every change i.e., creation, deletion or movement of objects is captured inside a *delta* by the high-level view and send it to the controller for further processing.

As low-level view offers less functionalities than high-level view, user can perform only addition and removal of the kitchen objects. Low-level view represents the kitchen space divided into blocks. Hence, each kitchen object is represented in the form of a group, combining any number of block(s) arranged in vertical or in horizontal direction filled with a unique color everytime a new object is added. For example, sink is represented by two horizontal blocks attached to one another and filled with blue color as shown in figure 22. As each object is identified with a group having an unique color, user can generate different colors by multiple mouse clicks on the non-occupied blocks while creating a new group. For deletion of a group, user can blur the color of an occupied block. Every group is tracked on the basis of the block's position that it is consist of along with its color and every change i.e., addition or removal is captured inside a *delta* by the low-level view and send it to the controller for further processing.

After the changes are done on either view, to see the effect on the other, user can click on the synchronization button and both the views will be updated.

5.4.3 Controller

The Controller is mainly responsible for event/action handling and basically manages the relationship between a View and a Model [25]. These actions are triggered while a user is interacting with the application on a web browser. It accepts the user requests, interacts with the Model, receives the response and generates the View.

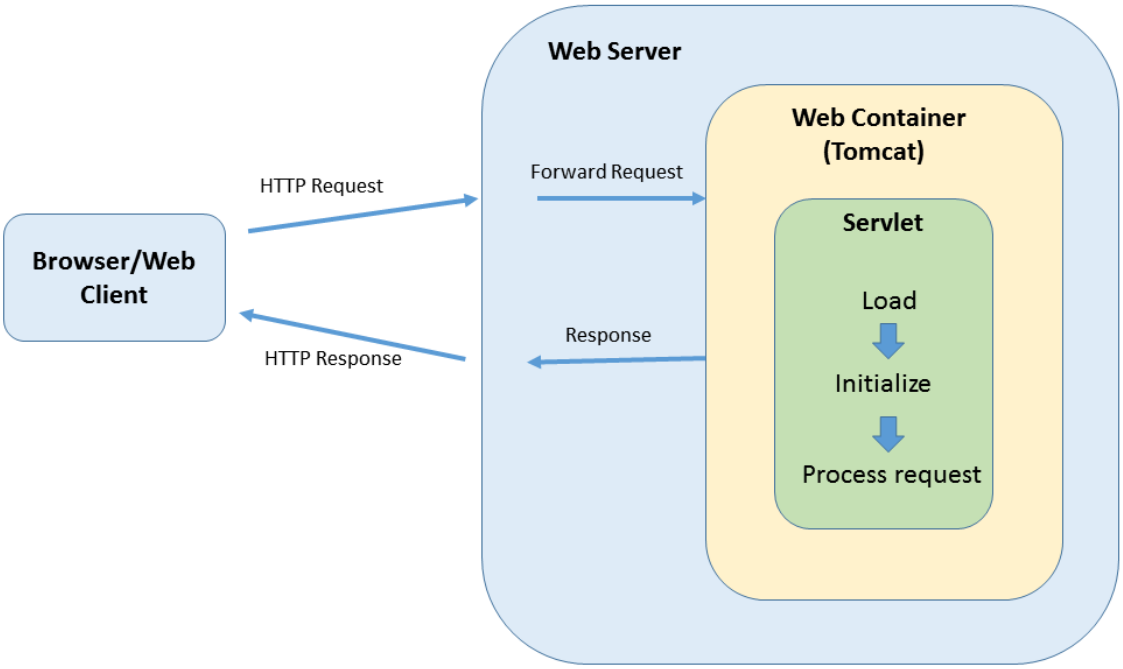

Figure 23: Servlet Lifecycle

In my case, I am using a thin Controller which is a Servlet along with a Controller helper consist of a java class.

Servlet & Controller Helper Servlet is a technology which provides a component-based, platform-independent method for building web-based applications [34]. Servlet is built on java platform to extend the capabilities of a web-server which makes it robust and scalable. It resides inside a web-server to generate dynamic content.

Servlet is capable of handling all kinds of client-server protocol but popularly and mostly used with the HTTP, the HyperText Transfer Protocol. A web container is essentially required to run a servlet. A web container is a component of the web server that interacts with the servlets and manages the lifecycle of servlets. In my case, I am using *Apache Tomcat* as my web container.

Figure 23 describes the complete lifecycle of a servlet w.r.t. a web server and a web container. The lifecycle steps are described as follows [34]:

1. Web server receives the `HTTP request` from the client interacting through a browser.
2. After accepting the request, web server forward the request to the web container i.e.,

tomcat.

3. Web container sends the request to the `Servlet class`.
4. If an instance of the servlet does not exist, the web container

   loads the servlet class, then creates an instance of the servlet class and initializes the servlet instance by calling the `init` method.
5. After that, web container invokes the `service` methods (normally HTTP methods i.e., `get` , `post` , `put` , `delete`) of the servlet class by passing the request and response objects and the actual processing of the request is done and response is generated.
6. Web container sends the response to the web server. Afterwards, web server creates the HTTP response and send it back to the client.

In my application, controller i.e., servlet is strongly coupled with a controller helper i.e., java class. Figure 24 describes the workflow of the Controller in my project. After invoking the service method of the servlet class, if an instance of the controller helper class does not exist, the servlet creates an instance of it and calls the appropiate method by passing the `UI Model` object. Inside the method of the controller helper, the conversion of the `UI Model` to the eMoflon specific models happens and data is sent to the `Model` for further processing. After processing, controller helper receives the response from the `Model` and conversion of the eMoflon specific models to the `UI Model` happens before sending the data to the servlet.

5.5 Challenges

During the entire designing process as explained in previous sections of this chapter, I came across a few challenges. This section describes them in detail.

Architecture Design – *Avoiding circular dependency between projects* The first hurdle that I faced in the designing process was with application framework design. The challenge was correctly maintain the dependencies of projects without creating a circular dependency and strong coupling among each other. To overcome this challenge, basic concepts on design patterns and MVC architecture helped me in building the web architecture and maintaining the dependencies among the components in a correct way.

UI Design – *Represent one workspace in two different views* UI design was the second big challenge that I faced during the designing process.

First problem was with the design, look, and feel of the high-level and low-level view. As per the requirement, both the views should represent the same workspace area but in different ways

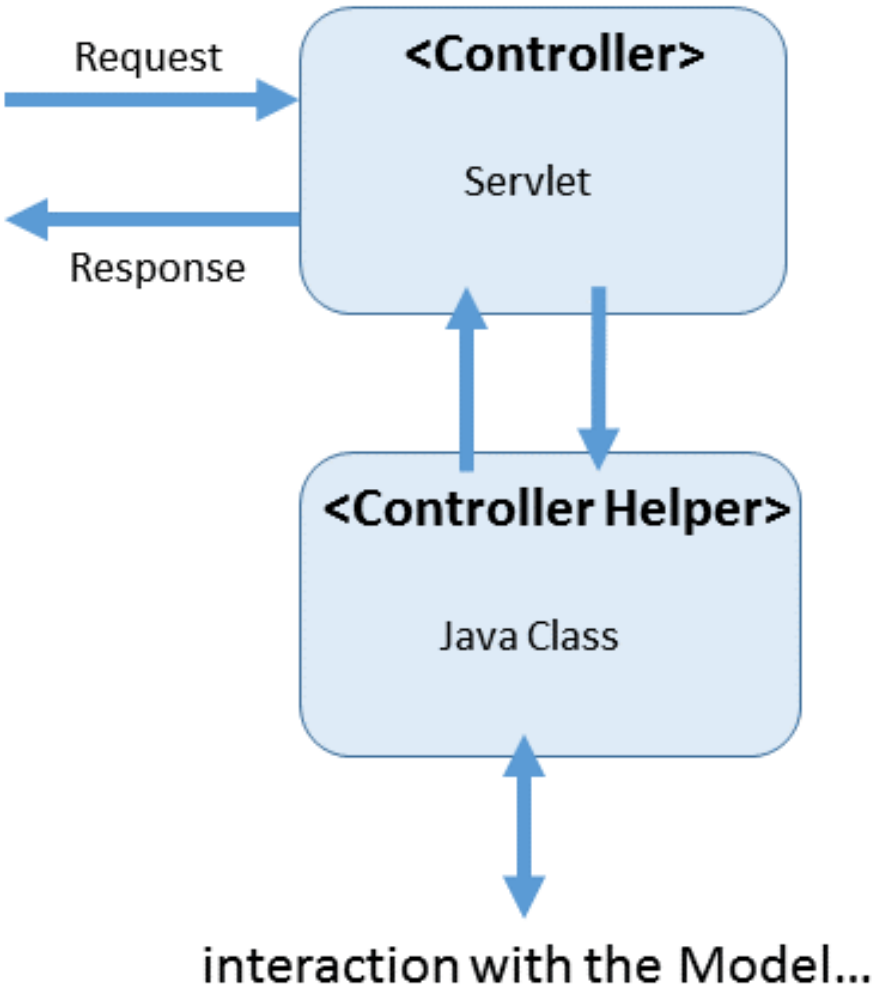

Figure 24: Workflow within the Controller

i.e, different action space, functionalities, and fexibility in interaction. So the challenge was to represent one workspace in two different views separated by functionalities and flexibility in use. Hence, I went through a few UI patterns [32] and scenarios based design principles [33] for knowing the basics. Finally, I and my supervisor after a few discussions chose an empty canvas as shown in figure 21 to represent high-level view and a canvas filled with blocks as shown in figure 22 to represent low-level view.

UI Design – *Capturing different changes(delta) done in the views* Second problem was with conceptualizing the structure of *delta* (refer Definition 3) by differentiating the functionalities e.g., changes done by user in the high-level and low-level view. So the challenge was to capturing different changes done in both the views but bundling them into one structure as the *eMoflon* tool requires one input for all the changes done for transformation process. As mentionaed earlier, high-level and low-level view are different from each other based on their functionalities. For example, in high-level view three types of changes are allowed i.e., creation of a new item, deletion of an item, and movement of an item. Whereas, in low-level view two types of changes are allowed i.e., creation of a new group and deletion of a group. Hence, the structure of changes in both the views will be different as well. For separation of concerns, I decided to maintain five different variables for five different kinds of changes i.e., `createdItem`, `deletedItem`, `movedItem`, `createdGroup`, and `deletedGroup` but, bundled them into one single class `Change`. Also, a user can perform a series of changes even of the same type i.e., creating more than one item, deleting more than one item, creating more than one group etc., so I decided to keep every variable as a `List` item. Fig 20 shows the class diagram of the `Change` class and section 5.4.1 explains the structure in deatil.

6 Implementation

In this chapter, I am going to describe the implementation process and its related details of the demonstrator in detail. Section 6.1 describes the overview of the implementation of the demonstrator with component, activity, and sequence diagrams. This is then followed by the detail description of the implementation steps involved in each layer i.e., Model, View, and Controller in Section 6.2. At last, Section 6.3 describes the challenges that I faced while implementing the entire framework and its components.

6.1 Core Details

This section provides an overview of the implementation steps involved with the demonstrator. First, I introduce the components of the tool and their interconnection. Then, I describes the general workflow of the tool by showing in which order which activities are executed. Finally, the communication between the components is presented.

6.1.1 Component Architecture

The architecture has three components to signify Model-View-Controller (MVC) pattern. Figure 25 describes the high-level architecture of the tool in form of a component diagram.

On the top, `View` (Web Browser) component is present which contains a graphical user interface and functionalities that belong to the user. With the changes on `View` component, data are being provided through interface `IDoAnalysis` to the `Controller` component and after the calculations are done, the results are sent back to the `View` through interface `IProvideResults`. Both `View` and `Controller` resides on the same machine. `Model` (Bx Tool) component encapsulates and manages the state of all models by communication with the `Controller` through interface `IRules`.

6.1.2 General Workflow

6.1.3 Interaction between the Components

6.2 Architecture Layers

This section describes each architecture layers i.e., Model, View, and Controller in detail from implementation point of view.

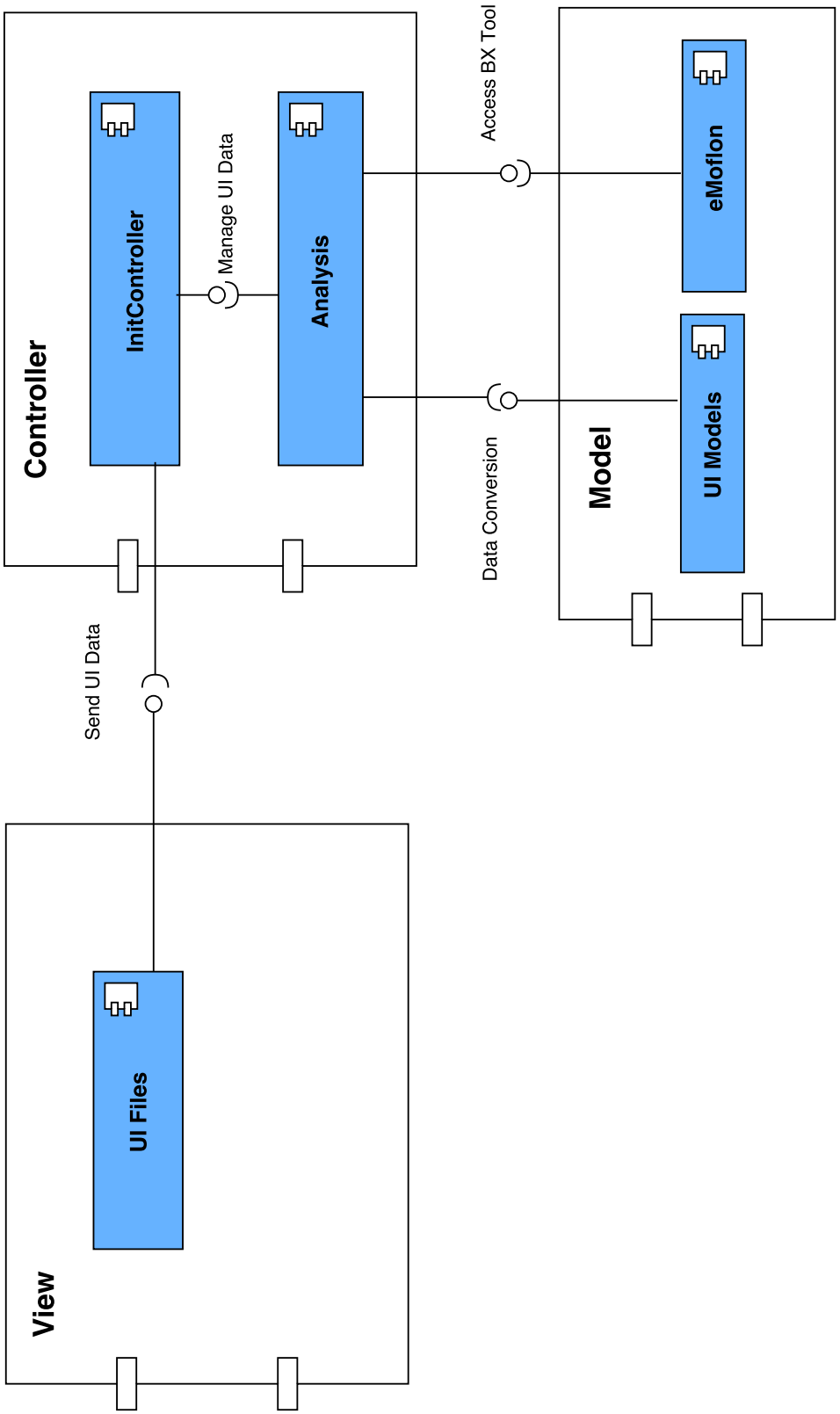

Figure 25: Component Diagram of Demon-BX Tool

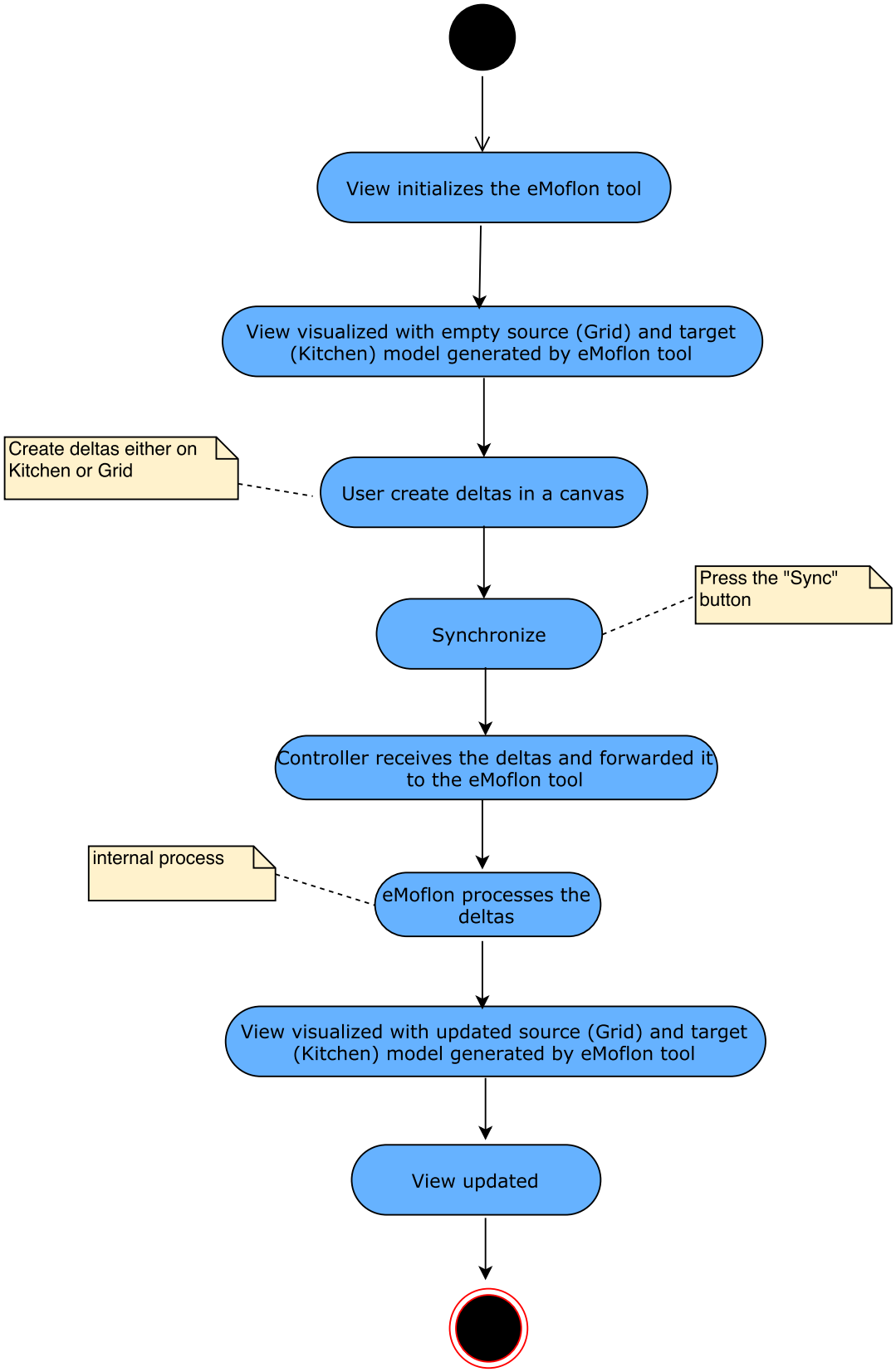

Figure 26: Activity Diagram for General Workflow of Demon-BX Tool

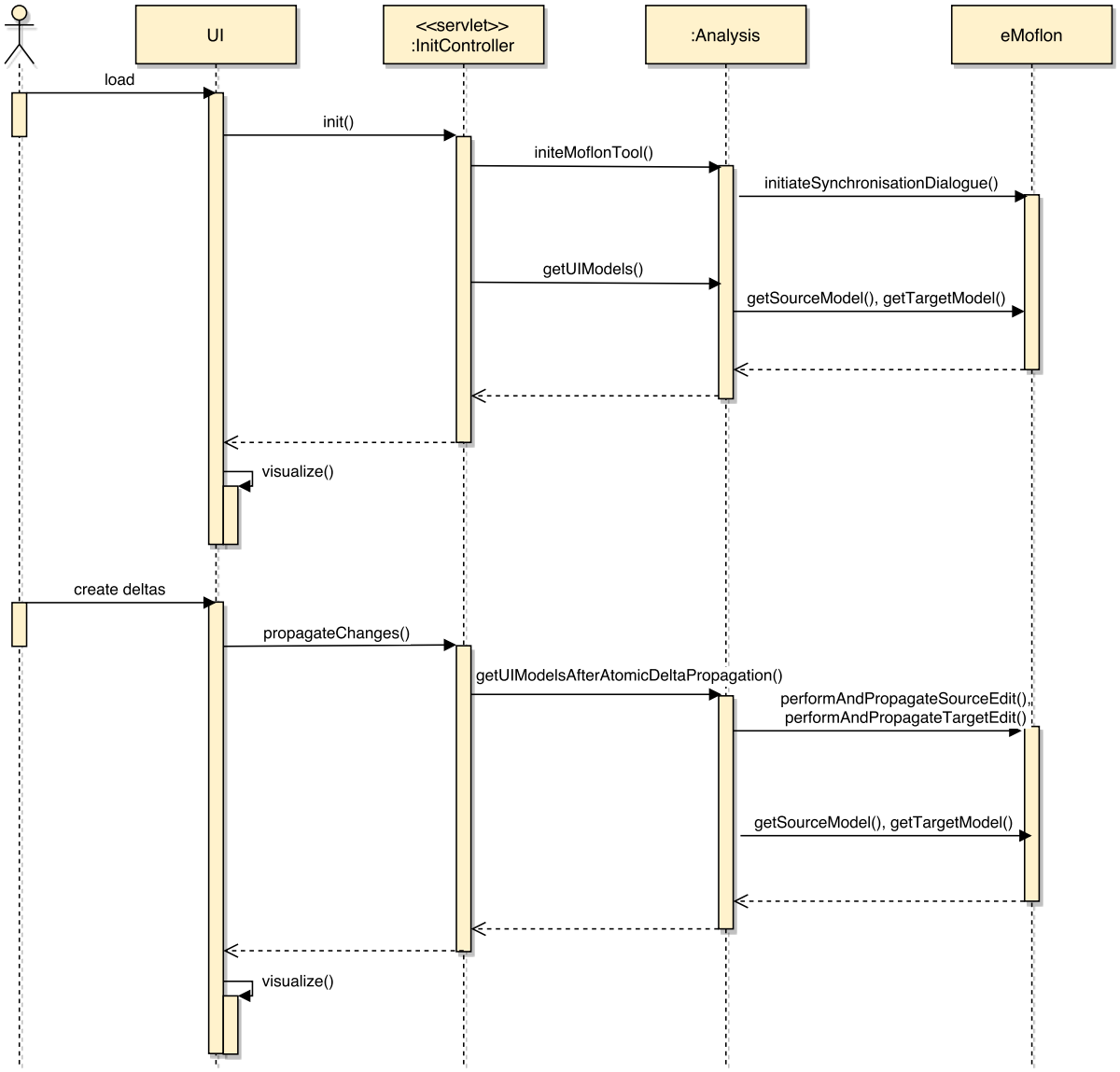

Figure 27: High Level Sequence Diagram of Demon-BX Tool

6.2.1 Model

Transformation Rules

6.2.2 View

6.2.3 Controller

controller and helper - how servlet is implemented with actual class name and func.

6.3 Challenges

During the entire implementation process as explained in previous sections of this chapter, I came across a few challenges. This section describes them in detail.

Bx Tool selecting the type of delta (refer def.) implementing atomic deltas to track successful/failed changes translation. User Choice Selection – interupting the tool inbetween and start the process again

UI Implementation Designing the user interactions with high-level view was relatively easy as user have to deal with the objects with actions such as, mouse movements and button clicks. But, the real challenge was to handle the user interactions in low-level view's block structure. In low-level view, only addition and removal of objects are allowed and objects will be shown by unique colors.

User Choice Selection

Handling Multiple User Request

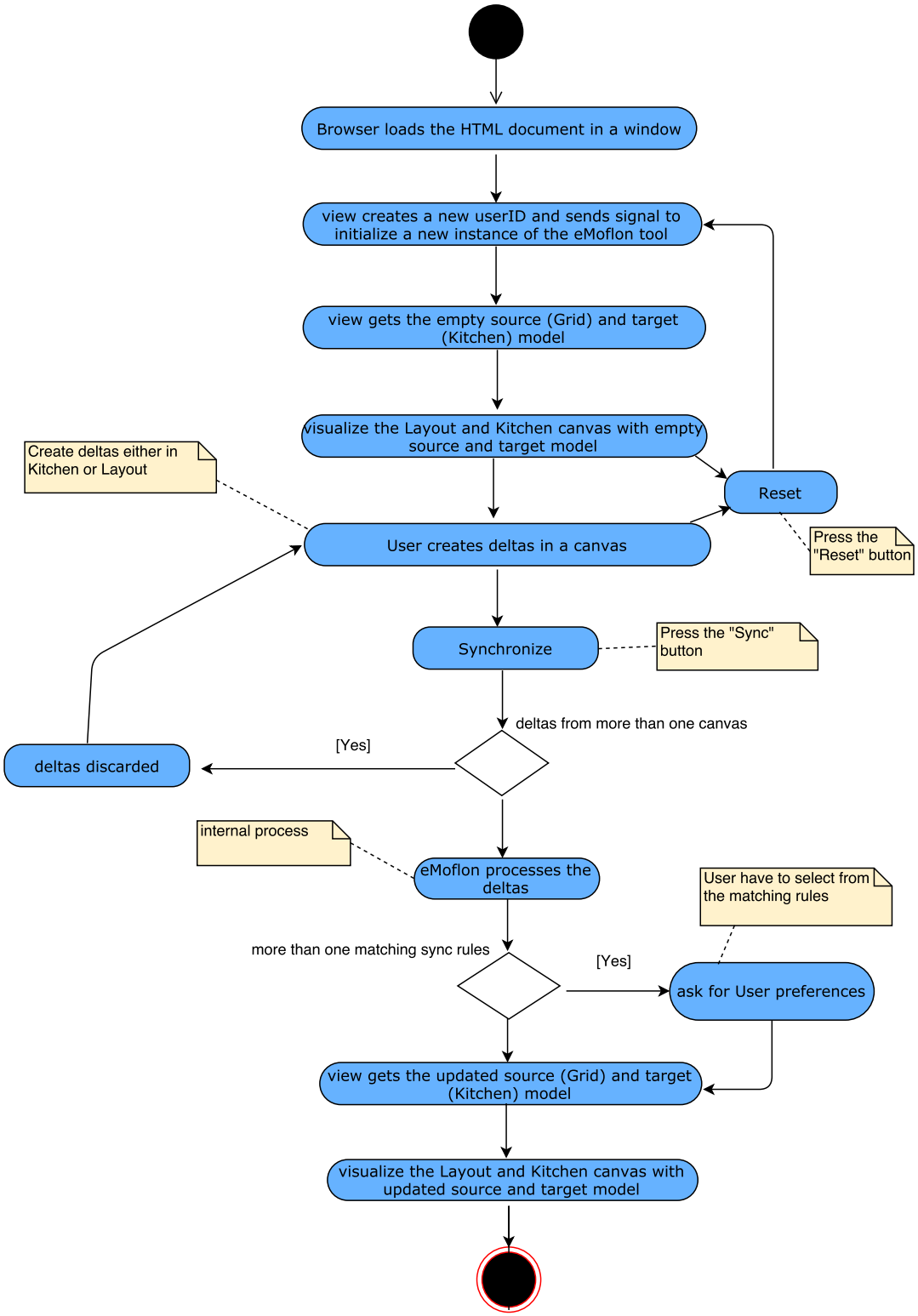

Figure 28: Activity Diagram for View

7 Evaluation

7.1 Discussion and Feedback

7.2 Evaluation and Results

8 Summary and Future Work

8.1 Conclusion

Finding the limitations of the tool, benchmarx.

8.2 Future Work

1. Another tool 2. Another example 3. Tool as a webservice

9 Appendix

The Appendix can be used to provide additional information, e.g. tables, figures, etc.

List of Figures

List of Tables

References

[1] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, *Bidirectional Transformations: A Cross-Discipline Perspective*, GRACE International Meeting , Shonan, Japan, 2008. 1, 2, 17

[2] Z. Hu, A. Schürr, P. Stevens, and J. Terwilliger, *Dagstuhl Seminar* #11031 on Bidirectional Transformations "bx", Dagstuhl Reports, Vol. 1, Issue 1, pages 42-67, January 16-21 , 2011. 2, 17

[3] J. Gibbons, R. F. Paige, A. Schürr, J. F. Terwilliger and J. Weber, *Bi-directional transformations (bx) - Theory and Applications Across Disciplines*, [Online]. Available: https://www.birs.ca/workshops/2013/13w5115/report13w5115.pdf. 1, 2, 24, 25

[4] R. Oppermann and P. Robrecht, *Benchmarks for Bidirectional Transformations.* Seminar Maintaining Consistency in Model-Driven Engineering: Challenges and Techniques, University of Paderborn: Summer Term 2016. 2

[5] A. Anjorin, A. Cunha, H. Giese, F. Hermann, A. Rensink, and A. Schürr, *Benchmarx*. In K. S. Candan, S. Amer-Yahia, N. Schweikardt, V. Christophides, and V. Leroy, editors, Proceedings of the Workshops of the EDBT/ICDT 2014 Joint Conference (EDBT/ICDT 2014), CEUR Workshop Proceedings, pages 82-86. CEUR-WS.org, 2014. 25

[6] A. Anjorin, Z. Diskin, F. Jouault, Hsiang-Shang Ko, E. Leblebici, and B. Westfechtel, *Benchmarx Reloaded: A Practical Benchmark Framework for Bidirectional Transformations*. In R. Eramo, M. Johnson (eds.): Proceedings of the Sixth International Workshop on Bidirectional Transformations (Bx 2017), Uppsala, Sweden, April 29, 2017, published at http://ceur-ws.org. 9, 11, 13, 25, 26

[7] A. Schürr. *Specification of graph translators with triple graph grammars.* In E. W. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG 94*, volume 903 of LNCS, pages 151-163, Herrsching, Germany, June 1994. 19

[8] A. Bucaioni and R. Eramo, *Understanding bidirectional transformations with TGGs and JTL*, in Proc. of the Second Workshop on Bidirectional Transformations (BX 2013), no. 57, 2013. 19

[9] A. Anjorin, E. Burdon, F. Deckwerth, R. Kluge, L. Kliegel, M. Lauder, E. Leblebici, D.Tögel, D. Marx, L. Patzina, S. Patzina, A. Schleich, S. E. Zander, J. Reinländer, and M. Wieber, *An Introduction to Metamodelling and Graph Transformations with eMoflon. Part IV: Triple Graph Grammars.* [Online]. Available: https://emoflon.github.io/eclipse-plugin/release/handbook/part4.pdf 2, 19

[10] BX Community, [Online]. Available: http://bx-community.wikidot.com/ 2

[11] BX Community, [Online]. Available: http://bx-community.wikidot.com/examples:home 20

[12] N. Macedo, T. Guimaräes and A. Cunha, *Model repair and transformation with Echo*. In Proc. ASE 2013, ACM Press, 2013. 2

[13] Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu, *BiGUL: A formally verified core language for putback-based bidirectional programming.* In Partial Evaluation and Program Manipulation, PEPM'16, pages 61-72, ACM, 2016. 2, 18, 19

[14] Z. Hu and H.-Shang Ko, *Principle and Practice of Bidirectional Programming in BiGUL*, Tutorial [Online]. Available: http://www.prg.nii.ac.jp/project/bigul/tutorial.pdf 19

[15] *BiYacc, tool designed to ease the work of writing parsers and printers*, [Online]. Available: http://biyacc.yozora.moe/ 18

[16] *SHARE - Sharing Hosted Autonomous Research Environments*, [Online]. Available: http://is.ieis.tue.nl/staff/pvgorp/share/ 19

[17] A. Bucaioni and R. Eramo, *Understanding bidirectional transformations with TGGs and JTL*, In Proc. of the Second International Workshop on BX, 2013. 14

[18] S. Easterbrook, J. Singer, M.-A. Storey, & D. Damian. *Selecting Empirical Methods for Software Engineering Research. Guide to Advanced Empirical Software Engineering*, 285-311, 2008. Retrieved from http://doi.org/10.1007/978-1-84800-044-5_11 3, 4

[19] J. Rumbaugh, I. Jacobson and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*, Pearson Higher Education, pages 1-21, 2004. 7

[20] S. Beydeda, M. Book and V. Gruhn, *Model-Driven Software Development*, ACM Computing Classification, pages 1-8, 1998. 7

[21] S. Sendall and W. Kozaczynski, *Model transformation: the heart and soul of model-driven software development*, IEEE, Vol. 20, Issue: 5, pages 42-45, Sept.-Oct. 2003. 1, 12

[22] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns*, Addison-Wesley, Boston Massachusetts USA, vol. 47, 1995. 26

[23] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design*, In Nierstrasz O.M. (eds) ECOOP 1993 - Object-Oriented Programming. Lecture Notes in Computer Science, vol 707. Springer, Berlin, Heidelberg. 26

[24] J. Deacon, *Model-view-controller (mvc) architecture*, Computer Systems Development, pages 1-6, 2009. 30, 34

[25] D. Distante, P. Pedone, G. Rossi and G. Canfora, *Model-driven development of Web applications with UWA, MVC and JavaServer faces*, Lecture Notes in Computer Sci-

ence (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 4607, pages 457-472, 2007. 30, 37

[26] E. Freeman, E. Robson, K. Sierra and B. Bates, *Head First Design Patterns*, O'Reilly, USA, pages 536-566, 2004. 26, 30, 34

[27] *MDN, "Canvas API",* Retrieved May, 2017, from https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API. 34

[28] *Fabric.js,* [Online]. Available: http://fabricjs.com/. 34, 37

[29] *Processing.js,* [Online]. Available: http://processingjs.org/. 34

[30] *Pixi.js,* [Online]. Available: http://www.pixijs.com/. 34

[31] D. Goodman and M. Morrison, *Javascript Bible*, Wiley Publishing Inc., 6th Edition, pages 1-8, Indianapolis, Indiana, 2007. 34

[32] J. Tidwell, *Designing interfaces*, O'Reilly, vol. XXXIII, Pages 81-87, 2012. 41

[33] K. Kusano, M. Nakatani and T. Ohno, *Scenario-based interactive UI design*, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13, 391, 2013. 41

[34] J. Hunter and W. Crawford, *Java servlet programming*, vol. 37, 2001. 38

[35] P. Sharma and S. Dhir, *Functional & non-functional requirement elicitation and risk assessment for agile processes*. International Journal of Control Theory and Applications, vol. 9, pages 9005-9010, 2016. 15, 16