Synchronization Techniques in Object Oriented Software Development Environments

Bringing a Synchronized State among Model and Code Components in Xtext Framework

Ozan Aksoy
Abstract

Synchronization Techniques in Object Oriented Software Development Environments

Ozan Aksoy

Code Synchronization is the process to achieve an equalized state among code domains. However, the equalization process is not clearly defined. An equal state can be achieved in various forms, such as destroying all irrelevant code parts, or just letting them exist without including them in the synchronization process. For this reason, a synchronization process is dependent on its synchronization behavior. This behavior is the characteristics of the process. Additionally, conditions of a synchronization process have an impact on the realization of the behavior. Behavior can only be realized if the conditions of compared states, environment variables, and abilities are configured in support of the behavior. Thus, it can be said that synchronization behavior is both the analyzer and the decision maker of a synchronization process.

When a synchronization process consists of a singular synchronization analysis and implementation process for two or more code bodies, then this can also be called a synchronization attempt. A synchronization attempt is analyzed under the provided synchronization behavior in limited, expected and exceptional conditions. Though this analysis, any synchronization attempt can be concluded in the synchronization behavioral expectations.

It was observed that a new synchronization theory is required to overcome the needs for the synchronization mechanisms. For this reason, synchronization as a concept has been researched and two contributions are generated. One of these is to establish a start point for synchronization studies, which is defined in nature of synchronization. Secondly, by using the findings from nature of synchronization, expected behaviors during synchronization have been analyzed as functional calculations. These functional calculation processes have been classified and they are documented for every classified case. Additionally, a methodology for analysis and design of synchronization mechanisms has been provided. Later, this is used on Xtext Framework.

This research is conducted for a double degree master program in Computer Science in Uppsala University and Software Engineering in Tongji University. The project is conducted in BMW ConnectedDrive Laboratories, Shanghai, China. It is supervised by Uppsala University, Tongji University, and Shanghai based BMW China ConnectedDrive Research Laboratory. It is aimed to satisfy the academic research needs of BMW AG.
# Table of Contents

## Chapter 1 Introduction

## Chapter 2 Synchronization Concept

2.1 Issue of Synchronization

2.2 Concept of Modeling in Defining Synchronization

2.3 Entity States and Nature of Dependency

2.3.1 Dependency between Entities of Different Realms

2.3.2 Capturing Complex Relations

2.4 Nature of Relations among Entity States

2.5 Code Artifacts between Mechanisms and Entity States

## Chapter 3 Synchronization Analysis

3.1 Analysis of Changes in Code Elements and Code Entities

3.1.1 Identifying Code Elements and Code Entities

3.2 Types of Changes on Code Artifacts

3.2.1 Types of Changes in Code Elements

3.2.2 Types of Changes in Code Entities

3.2.3 Changes in Code Entities and Refactoring

3.2.4 Conclusions on Changes inside and among Entities

3.3 Nature of Synchronization

3.3.1 Limits of Synchronization Ability

3.3.2 Desired Result

3.4 Using Nature of Relationship and Nature of Synchronization

## Chapter 4 Synchronization Behavior

4.1 Basic Behavior

4.1.1 Type 1 Basic Behavior

4.1.2 Type 2 Basic Behavior

4.2 Complex Behavior:

4.2.1 Complex Relation Resolution Toolset: Deep Synchronization Methods

4.2.2 Type 1 Complex Behavior

4.2.3 Type 2 Complex Behavior

4.2.4 Type 3 Complex Behavior

4.2.5 Type 4 Complex Behavior: Multi-Response Situations

4.2.6 Type 5 Complex Behavior: Confliction among Three Major Synchronization Parameters

4.3 Combinations
# Table of Contents

## Chapter 5 Design Methodology ................................................................. 67

5.1 Necessary Sub-Mechanisms for Synchronization ........................................ 67
5.2 Methodology for Constructing Synchronization Mechanisms .......................... 68
  5.2.1 Aimed behavior: Behavioral goals ......................................................... 69
  5.2.2 Overall Analysis of Problem Domain ...................................................... 69
  5.2.3 Segmentation of Target Domains ............................................................ 69
  5.2.4 Code Complexity Analysis ................................................................. 70
  5.2.5 Impact of Change Analysis ............................................................... 70
  5.2.6 Mechanism Model ............................................................................ 71
5.3 Major Sub-Mechanisms for a Synchronization Mechanism ............................ 74
  5.3.1 Change Notifier Mechanism ............................................................... 75
  5.3.2 Entity Relation Tracing Mechanism ...................................................... 75
  5.3.3 Code Generation Mechanism ............................................................. 76
  5.3.4 Behavior Decision Mechanism ........................................................... 76
5.4 Abstract Schemes for Synchronization Mechanism Design ............................ 77
  5.4.1 Tracer Mechanism Design ................................................................. 78
  5.4.2 Change Notifier Design ................................................................. 79
  5.4.3 Behavioral Decision Mechanism Design ............................................... 80
  5.4.4 Generation Mechanism Design .......................................................... 81
5.5 Discussions and Findings on Necessary Sub-Mechanisms ............................. 81
  5.5.1 Discussion on Traceability ................................................................. 81
  5.5.2 Discussion on Changes in Code Artifacts ............................................. 84
  5.5.3 Discussion on Serialization and Representation ..................................... 87
  5.5.4 Discussion on Similarity and Equality ............................................... 87

## Chapter 6 Applying Design Methodology to Xtext Framework ........................ 91

6.1 Aimed Behavior: ..................................................................................... 91
6.2 Overall Analysis ....................................................................................... 91
6.3 Segmentation of Target Domains .............................................................. 92
  6.3.1 Analysis of the Framework for Synchronization Attempts .................... 92
  6.3.2 Segmentation of the Mechanism Workflow ......................................... 93
6.4 Code Complexity Analysis ........................................................................ 96
  6.4.1 Code Entities in Domains ..................................................................... 97
  6.4.2 Code Elements in Target Domains ...................................................... 98
6.5 Impact of Change Analysis ........................................................................ 101
  6.5.1 Limitation Conditions .......................................................................... 102
  6.5.2 Expected Conditions ........................................................................... 105
### Table of Contents

6.5.3 Exception Conditions .................................................................................. 107
6.5.4 Target and Path Selection ........................................................................... 108
6.6 Mechanism Model: Suggestions ..................................................................... 111
6.7 Features of Synchronization Mechanism ......................................................... 112
   6.7.1 Compared States of Entities: ..................................................................... 112
   6.7.2 Time of Analysis: .................................................................................... 113
   6.7.3 Correct decision: ..................................................................................... 113
   6.7.4 Implementation of Decision: .................................................................... 113
6.8 Findings for Xtext Framework Synchronization Solution ..................................... 116
   6.8.1 Findings for Xtext Framework Synchronization Solution for Tracing Mechanism .................................................................................................................................................................................. 116
   6.8.2 Findings for Xtext Framework Synchronization Solution for Changes on Code Artifacts ............................................................................................................................. 117
   6.8.3 Findings for Xtext Framework Synchronization Solution in Serialization ........ 118
   6.8.4 Findings for Xtext Framework Synchronization Solution on Similarity and Equality Calculations .................................................................................................................. 118

Chapter 7 Conclusion .......................................................................................... 119

7.1 Future Work .................................................................................................... 122

References .......................................................................................................... 125

Abbreviations ....................................................................................................... 131

Appendix 1 Functional Relations .......................................................................... 133

Appendix 2 Nonfunctional Relation ...................................................................... 135

Appendix 3 Two Static State Conflict Conditions ............................................... 137

Appendix 4 Direction of Change Conflict Conditions .......................................... 139

Appendix 6 Type 2 Basic Behavior Starting Conditions ...................................... 143
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Abstract of a Synchronization Mechanism</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>Synchronization Basics between Two States</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Expected Realm Relation for Domain Specific Language Platforms</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Structure of an Entity</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Method Dependency Example</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Entities with Mapping to Properties and Methods Separately</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>Complex Dependency Relations</td>
<td>13</td>
</tr>
<tr>
<td>2.7</td>
<td>Mapping Entities with Captured Dependencies</td>
<td>14</td>
</tr>
<tr>
<td>2.8</td>
<td>Evolution of Objects</td>
<td>16</td>
</tr>
<tr>
<td>2.9</td>
<td>Evolved States of the Artifacts</td>
<td>17</td>
</tr>
<tr>
<td>2.10</td>
<td>Entity Declarations between Realms</td>
<td>17</td>
</tr>
<tr>
<td>2.11</td>
<td>Change Among Realms and Correspondents</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Functional Relation Synchronization Attempt</td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Processing of a Nonfunctional Relation Resolution</td>
<td>42</td>
</tr>
<tr>
<td>4.3</td>
<td>Synchronization Attempt of a Nonfunctional Relation</td>
<td>43</td>
</tr>
<tr>
<td>4.4</td>
<td>Functional Multi-response Cases in Plain State Division</td>
<td>45</td>
</tr>
<tr>
<td>4.5</td>
<td>An Example for Plain State Division</td>
<td>46</td>
</tr>
<tr>
<td>4.6</td>
<td>An Example Case for a Plain Separate Matching</td>
<td>48</td>
</tr>
<tr>
<td>4.7</td>
<td>An Example for Reverse Unified Response</td>
<td>49</td>
</tr>
<tr>
<td>4.8</td>
<td>Preparation of the Unified Response</td>
<td>50</td>
</tr>
<tr>
<td>4.9</td>
<td>Inferring the Unified Response</td>
<td>51</td>
</tr>
<tr>
<td>4.10</td>
<td>Example for Check Step of Type 1 Complex Behavior with Solution 3</td>
<td>53</td>
</tr>
<tr>
<td>4.11</td>
<td>An Example for Resolution of Type 1 Complex Behavior with Solution 3</td>
<td>54</td>
</tr>
<tr>
<td>4.12</td>
<td>Starting Conditions Example for Type 2 Version 1 Complex Behavior</td>
<td>55</td>
</tr>
<tr>
<td>4.13</td>
<td>Resolution of Type 2 Version 1 Complex Behavior</td>
<td>56</td>
</tr>
<tr>
<td>4.14</td>
<td>Starting Conditions Example for Type 2 Version 2 Complex Behavior</td>
<td>56</td>
</tr>
<tr>
<td>4.15</td>
<td>Resolution of Type 2 Version 2 Complex Behavior</td>
<td>56</td>
</tr>
<tr>
<td>4.16</td>
<td>Starting Conditions Example for Type 2 Version 3 Complex Behavior</td>
<td>57</td>
</tr>
<tr>
<td>4.17</td>
<td>Resolution of Type 2 Version 3 Complex Behavior</td>
<td>57</td>
</tr>
<tr>
<td>4.18</td>
<td>An Example for Starting Conditions of Type 3 Complex Behavior</td>
<td>58</td>
</tr>
<tr>
<td>4.19</td>
<td>An Example Resolution of Type 2 Version 3 Complex Behavior</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Figures

Figure 4.20 Reversing the Unified Response of a Sub-Entity or Element ..................60
Figure 4.21 Preparation for a Unified Reverse Response ..................................61
Figure 4.22 Creation of Reverse Unified Relations during Multi-Response Case ........62
Figure 4.23 Final Resolution of Reverse Unified Relations ..............................62
Figure 5.1 Creation of Selective Support over Basic Support .............................73
Figure 5.2 Elements that are required for Multi-Response Support ....................73
Figure 5.3 Complete Synchronization Support ..............................................74
Figure 5.4 Overview of Fundamental Synchronization Mechanism ....................77
Figure 5.5 External Traceability Information Storage Design ............................78
Figure 5.6 Internal Traceability Information Storage Design ............................79
Figure 5.7 Basic Design of Change Notifier Mechanism ..................................79
Figure 5.8 Basic Design of Synchronization Decision Mechanism ....................80
Figure 5.9 Basic Generation Ability Mechanism Design ..................................81
Figure 6.1 Mechanism of the Xtext Framework ..............................................92
Figure 6.2 Linear View of Code Generation Mechanism ..................................93
Figure 6.3 Expected Actions in Linear Workflow ...........................................94
Figure 6.4 Clustering Correlated Workflow Members ......................................95
Figure 6.5 Phases of DSL Development in Xtext Framework ............................96
Figure 6.6 Domain Model and Model Code ..................................................102
Figure 6.7 Domain Model and Generation Templates ......................................103
Figure 6.8 Model Code and Generation Templates ........................................103
Figure 6.9 Relation of Three Target Domains ...............................................104
Figure 6.10 Abstraction of the Synchronization Mechanism Design ..................112
List of Tables

Table 3.1 Entity and Element Segmentation ................................................................. 20
Table 4.1 Chart of Functional Calculation Parameters .................................................. 37
Table 4.2 Chart of Nonfunctional Calculation Parameters .............................................. 38
Table 4.3 Example Case for a Functional Relation ......................................................... 41
Table 4.4 Example Case for a Nonfunctional Relation .................................................... 42
Table 4.5 Stating Conditions for Type 1 Complex Behavior ............................................. 52
Table 4.6 Interim Step during Resolution of Type 1 Complex Behavior .............................. 52
Table 4.7 Starting Conditions for Type 2 Complex Behavior ............................................ 55
Table 4.8 Calculation Conditions during Type 2 Complex Behavior ............................... 55
Table 4.9 Starting Conditions for Type 3 Complex Behavior .......................................... 57
Table 4.10 Converted Nonfunctional Resolution Step Conditions ................................. 58
Table 4.11 Secondary Nonfunctional Resolution Step for Type 3 Complex Behavior ....... 58
Table 6.1 Sample Code for Code Entities ..................................................................... 97
Table 6.2 Same Code for Code Elements ..................................................................... 99
Table 6.3 Equivalent Code Elements Example ............................................................. 100
Chapter 1 Introduction

Software can be described as an evolving system. At the first phases of software development, most of the effort is spent on the documentation and the design of the project [1]. On later phases of development, the design began to lack the consistency with the changes of the implementation level. As it can be seen from the literature review of [2], these consistency issues can manifest itself throughout the development process in many forms, and identification and solution of these changes are resolved by using mechanisms that provide traceability of the artifacts and analysis of impact of the changes on the code and on the model.

With the introduction of the Domain Specific Language (DSL) frameworks, it can be stated that there is a challenge at tracing the changes from the model to the generated general purpose code. As an example, imagine that the defined DSL has been used to write a program for its domain solution. The real usable code will be based on a modeled object schema, which is created in defining the DSL [3]. During the development process, there will be a need for providing consistency and maintenance on related code bodies in the development framework [4]. From the observed frameworks such as Xtext DSL development framework on Eclipse development environment [5], JTL as explained in [6], and Rational Rose Modeling Environment [7], such a workflow was abstracted to following steps;

- A base for a domain model to be designed and created
- A workflow to generate an environment that uses the designed model
- A workflow to map and generate the platform code from the environment that holds the domain specific language code.
- And a target location to extract the workflow result(s).

In the given structure, target issue would be to find a solution for synchronization of changes among layers in the workflow.

As it sounds simple, it can be said that there are many aspects of the synchronization issue. First of all, we must understand the problem correctly. What we are trying to achieve here is creating a theory that will then be used as a guide to create the synchronization mechanisms. To achieve this target, one must understand the synchronization on object based environments. This understanding enables to create the analysis of the synchronization nature and opens the way for synchronization solutions in environments that are based on object oriented design.

In order to understand the needs for successfully implementing synchronization attempts,
two steps have been conducted. Firstly, concepts in modeling have been studied and, the composite features of the modeling were used in determining the structures of the code bodies. Secondly an analysis of synchronization attempts in object oriented development frameworks is conducted. This was achieved by a research thought the studies on the field, relations of objects as models, and functional calculations. Via these two studies, a new form of synchronization system has been formed.

It can be said that “synchronization” or “code synchronization” is born out of a process that occurs between evolving structural states of corresponding code bodies. However, this synchronization process has to be configured in order to behave the way that was initially desired by the user. This necessity is born because two states of compared code bodies can achieve a synchronized relation in various ways. This can include loss of information, partially equalizing code bodies to each other, etc. Therefore, behavior of the synchronization process is a key factor that determines the outcome.

To achieve such classifications, synchronization process has been divided to different conceptual steps. These are synchronization attempt, synchronization behavior, and synchronization methodology. Throughout this research, a synchronization attempt is used to mean a singular synchronization process that involves at least two compared code bodies. It is concluded that a synchronization attempt carries a special nature. This nature is used as a defining concept on how the synchronization is analyzed and implemented. Moreover, it is used in defining the synchronization behavior.

Synchronization behavior is the logical process that calculates the characteristic of an outcome from a synchronization attempt, and it also tries to find the way that the synchronization attempt can be realized. Then it compares the possibility of whether the synchronization attempt is achievable in the desired characteristics or not. This behavioral analysis is achieved via functional calculations, comparative techniques, and synchronization tools that are created from basic cluster manipulations. These various behavioral aspects are classified in a hierarchic structure, which leads to a constant focus on simplifying the condition of compared states to a basic, calculation friendly form. It was observed that these simplified forms are also becomes more synchronized with each other.

Beyond behavioral aspects, there is a need for a general analysis and design methodology for setup of the behavior and the synchronization mechanism. This need is tried to be satisfied via a synchronization design methodology.

Synchronization methodology is an analysis and design process that is used in applying synchronization behavior to the target environment. Synchronization methodology is interested in how to realize the mechanism as a whole workflow. Therefore, it is interested in
five major issues that proclaim on synchronization attempts. Synchronization methodology can be used in realizing various levels of behavioral capabilities. Such a classification is done in a step by step evolving manner. Each step includes an increased behavioral ability. These behavioral abilities are discussed in synchronization behavioral tools.

In this research, it is claimed that synchronization attempts have five major issues. Synchronization methodology tries to find answers to these five major questions in realization of a synchronization mechanism. These are:

1. How to relate the correct code parts to each other?
2. How to generate the equivalent code in correlated code domains?
3. How to understand the changes on the code domain?
4. How to understand which part to change to reflect the changes on corresponding domain?
5. How to conduct the synchronization implementation decision?

From the five major issues of synchronization, a synchronization mechanism can be summarized in four abilities. These are:

![Figure 1.1 Abstract of a Synchronization Mechanism](image-url)
- Tracing Ability: Used in relation of code parts and storing or extracting this relation information
- Change Notification Ability: Understands and Notifies about the changes on related code bodies.
- Generation Ability: Checks whether if equivalent code can be generated, also implements the decided behavior.
- Decision Making Ability: Analysis the whole process and decides the outcome and implementation decision.

All these sub-mechanisms are used in creation of the desired outcome and they can be merged to each other or they can be divided within each other depending on the framework that synchronization design methodology is implemented. However, in this research, their basic abstraction is aimed to resemble these four sub-mechanisms. Throughout these classifications, intertwined interests of sub-mechanisms are taken in steps of design and development, which leads to less confusion during the implementation of synchronization behavior.

Since analysis and design of the synchronization mechanism is the main focus of this work; in this research, decision making process and reaction to impact of change is intensely studied. Code bodies in a development environment are divided to basic elements and more complex code entities. These entities and elements are discussed and analyzed, and they are mapped to a functional relation calculation process. Hence, a new decision making process has been introduced. This process is constructed via the influence of functional and nonfunctional relations between two domains, and it is used in providing a holistic outcome. Via these calculations, characteristics of a synchronization attempt can be analyzed prior to its implementation. Other issues of synchronization behavior have been discussed and related information from related fields of research has been provided as discussions.

This research starts with providing a discussion on synchronization as an analysis of the issue. A novel view of synchronization process has been provided with relation to development environments and domain specific language development processes. Outcomes of this synchronization analysis have been used in construction of synchronization behaviors. These expected synchronization behaviors are divided to different types. Moreover, tools for solving complex behavioral processes have been provided. On the following, a documentation methodology has been presented. This methodology aims to create a design pattern application which centers on using the synchronization behavior as the brain of the synchronization mechanisms.
Chapter 2 Synchronization Concept

In this section, what is understood from the synchronization in the thesis will be explained. This explanation is a declaration of the writers own design for theorizing synchronization. Influence for these explanations can be found in discussions on relation of sets \([8]\) and bidirectional relations \([6]\). Moreover, entity relations and dependency concept among related entity states are influential for the synchronization analysis and they are documented in this section.

In order to comprehend the synchronization analysis, issue of synchronization must be explained in the conceptual level. Understanding the issue of synchronization provides the basis to understand the fundamental nature of two synchronization implications. These two fundamental concepts are \textit{nature of synchronization} and \textit{nature of dependency}. Nature of synchronization is the fundamental variables which gives the characteristics of a synchronization attempt. Nature of dependency is about the relations among code artifacts that impacts the work and integrity of the code. Thus, nature of dependency creates the limitations and relations of a meaningful synchronization process. Additional to these two fundamental concepts, types of changes that can occur on the code body is studied with the goal of achieving a concrete classification of reasons for synchronization. After comprehension of these three topics, synchronization behavior is possible to be constructed with foreseeable outcomes, thus leading to possibility to analyze the synchronization implementation within a synchronization attempt.

It must be noted that the concept of synchronization that depends on different levels of code body complexities with different hierarchic order was provided from the fact that interested target environments are object oriented.

Understanding the synchronization concept is the entry point for a synchronization analysis. In the following chapter which is dedicated to synchronization analysis, nature of relations and segmented identification of code body is explained. Later, by using the information that is provided in previous sections, nature of synchronization factors is explained. Finally, conclusions from these intertwined concepts are tied as a conclusion in order to be used in construction of the synchronization behavior calculations.
2.1 Issue of Synchronization

In this work, synchronization is understood as creation of an equal state between two different entities. These entities are aggregated clusters of sub-entities. Sub-entities can be in two forms. These are complex and basic. Complex sub-entities are structures with modular phases of construction. On the other hand, basic entities are simpletons with no other sub-level complexity. Code entities will represent large scale and singularly identifiable code bodies, such as object classes, while methods and properties will be referred as code elements. This classification is possible in the fact that code elements are the carriers for the behavioral descriptions for an object. A behavioral code body is a set of functions and problem solving descriptions within a code element. The division for code elements and code entities lies in the concept of Modeling.

2.2 Concept of Modeling in Defining Synchronization

Modeling concepts and Meta Modeling has leaded to necessity for tracing and maintaining the relation of software artifact, which lost their relation with their model definitions during development process [9]. Synchronization process does involve this...
evolution; however, it expands this model to code artifact relation via its nature. Synchronization is linked on modeling about which parts of code body are related to each other and what structural limitations do they bring which can be used in creation of equalized states of code.

A software model is a skeleton of a real world idea that can reflect the structure of a logical entity. A software model underlines the identification and relation of the code artifacts. These code artifacts are built on attributes and parameters. These attributes and parameters carry the behavior of the code artifact, which is the logical workflow that methods use when they are called, or when parameters set as they are fetched. [4]

On the other hand, a model is a hierarchic structure which makes it possible to trace in a hierarchal form. This allows providing structural limitations on the code artifacts. Hence, it is both a defining and limiting context. [10]

Code parts that are the fundamental carriers of the behavioral aspects owns different characteristics compared to hierarchically higher, thus more complex code bodies which owns and uses them. Same behavioral result can be achieved in different logical structures, and it uses different forms of tools to achieve the same behavior among different platforms. This fact makes it difficult to realize a equalization of the behavioral code structures. [11]. However, a Meta Model which defines the model itself can be created. This Meta Model is not interested in the behavioral aspect of the code body, but it can target the limited and foreseeable order of a code artifact. [12]

Moreover, studies that researched the impact of change on behavioral code have found out that such and impacts are mostly derived on consistence of the declared dependencies within the code part, and it also impacts the higher structures. This results in decreasing the importance on direct need for equalizing the behavioral code parts. [13]

For all these reasons, methods and properties of the code artifacts have been declared as code elements in this research. This naming signifies that they are the end of a hierarchy within the mapped structure of a code body. Code bodies that are in higher hierarchic order are defined as code entities, meaning that they are mapped in a definable, capsulated structure, and they are ready to equalize to a common form.

A secondary effect of of modeling is division in characteristics of code relations. Models are structures that can be transformed to each other and to other types of platforms. However models do not need to carry code elements, they just need to define their existence in the code artifacts. [14]
Thus, two outcomes are achieved:

(1) Code elements act as a sub-entity which carries an identifier. Hence, it can be stated that all code elements are a type of code entity, however not all code entities are code elements

(2) Similarly, a model can define a code entity; it can be transformed into other equivalent models by mapping the definition that it carries. Also, a plain code body that is written in platform specific languages such as e.g. Java or C# carries an embedded model structure thus a model definition can be deduced from them. However, model definition to code definition has a dictating and clear definition for the code body of the implementable platform specific code, while plain code structure that carries the code elements has an accepting and mirroring nature. Thus, a relation of model to a model, model to a source code, and source code to another source code differs according to such a relation nature.

Therefore, in order to catch such relation characteristics, a model realm and a code realm have been introduced in this research. A code entity can have a domain that resides in a model realm or a code realm in relation to its corresponding domain. Same is true for the corresponding domain. A relation among model realm to code realm is different than a model realm to model realm, or a code realm to code realm relation. Moreover, same code domain can act differently according to direction of the dictation relation.

Introduction of modeling and model’s relation to its reflected code correspondent is very significant during designing and implementing the limitation conditions and expected conditions of two code bodies which act as realm to realm pair. The direction, the structure of the model relevance, the Meta Modeling that underlines the model itself, open ways to impact such relations in various fashions. This gives a significant relational characteristic to a code body in relation to its corresponding code body, which may reside in an opposite or a similar realm characteristic.

Such a division is most meaningful in built up for a synchronization analysis of a domain specific language. This is because, in current trends, a domain specific language has a workflow that carries a Meta-Model, a Model, a Code Generation Description, and a Generated Platform Specific code that is for final implementation [5] [3]. Realm specifications define the characteristics of limitations, expectation and exceptions of the relation and thus the synchronization process. In general, realm relations for a domain specific language are similar to Figure 2.2:

For singular model to code relations such as UML to source code, lower part of the Figure 2.2 is satisfactory. A model language that explains the definition of the code artifact is in dictating relation to code artifacts which acts as the source code, thus they are acting as
model realm to code realm relation. For an example case, this means that the UML model does provide the structure and generated code must follow the model, and the synchronization from code to model is limited to code parts that does not include behavioral code bodies of code elements that resides in code realm. On the other hand, a domain specific language grammar to developed domain specific code relation is similar to top-down left part of the same figure. This time previous model does behave like a source code and Meta Model owns the characteristics of a model code.

Another notable issue is on code realm to code realm and model realm to model realm relations. In the case of Figure 2.2, this is true for code generation templates and model codes. They behave in code to code relation. For a counter example, model to model relation can be mapped between two different model structures, such as XML schemas and UML descriptions [15].

![Figure 2.2 Expected Realm Relation for Domain Specific Language Platforms](image)

When relation is code realm to code realm, their dependency is not on the reflection of the structural changes. However, these two domains use each other without dictation to one another. Thus, the impact of change on one domain would only impact the usability of the resources, and it will not create a structural inconsistency, but a workflow failure. On the other hand, when relation is model to model, then the code bodies must reflect each other, thus dictating to one another. Unlike code realm to code realm relation, a change on the one
domain must be reflected on the corresponding structure of the co-domain.

It must be noted that, model to code and model to model relation is reflected though the relation of “Relates” and code to code relations is reflected by the term “Uses”.

2.3 Entity States and Nature of Dependency

State of an entity is a snapshot which captures the declaration of the entity itself including the condition of its elements. This declaration is a record of all significant information which describes the entity itself. Hence, it can be referred as both the map and the scheme of the “idea” that construct the entity from a model as a type of model. Action of synchronization can occur between states of the same entity, or between states of different entities. However, firstly the nature of these relations must be understood. In order to achieve this, one must look to the structure of the entity relations in the light of nature of dependency.

In order to synchronize the entities, desired state of an entity is compared with one or more entities. Simplest forms of synchronization can occur between such two different states. During an attempt to create equalization between two states, one must clarify the “Limits of Synchronization Ability” and “Desired Result. These two major concepts are combined in one topic called as the nature of synchronization. Assumption is that, the most desired synchronization result is a condition which keeps all or most of elements and creates complete equal state among entities. These two aspects of synchronization nature are results from the impact of change that occurs via any change that appears on the code.

An attempt to synchronize the elements must assess the condition of change in elements. Assessment is needed to understand the possible impact of the synchronization to elements of entities. This is not the nature of the entity as a whole but rather the nature of elements as a sub-entity. In order to satisfy this requirement, elements are needed to be assessed by conditions that are selected as requirements by the user of the mechanism.

When they are alone, objects are not useful software entities [4]. Therefore it is a necessity to create relations between objects. Relations of conceptual entities were observed from relations of real objects according to [16] and [17].

However, creation of relationships also leads to the issue of dependency. In the observation of this research, it has to be noted that synchronization of the software artifacts in different fields are affected by this relationship nature of the objects. Relationship nature identifies the nature of entity states. It does this by dictating the necessities for consistency among entities. This is an outer dictation factor. When compared to the action of declaring an entity as immutable, it is not constant as such an action, but it changes as dependencies as
they vary in software evolution process.

2.3.1 Dependency between Entities of Different Realms

In the context of this research, dependency is seen as an important point to figure out in the issue of synchronization. As pointed out in [9], objects are code artifacts that are capsulated in the string form. They exist with their properties and operations in this form, thus enabling more generalization. This is also the foundation for the modeling concepts [18].

In a model, class descriptions can be seen as entities of various forms. They can be either independent or dependent. Such a dependency can be in many forms [19]. However, the model is still in the form which will be in context of representation of the object, although is hold in a textual form. These textual representations of an object include various attributes and operations. In order to make them comparable to each other, we can declare both methods and attributes in property declaration. In this research, such a simplification is considered as extending the classification of object properties.

Properties have specific features which makes them different from each other. These differences vary according to the perspective of the comparison. It must be noted that the capability of the comparison mechanisms are directly affected by the data that is collected about the properties and the entities.

In Figure 2.3, we see a mapping of model and its entities. In this figure, all elements in the namespace are seen as properties of the class, including methods. We can state that if we declare classes as entities of a model, all attributes and methods are sub-entities which assemble the class. In the case of Figure 2.3, we declare all sub-entities as textual forms without trying to catch underlying the semantic mechanisms. Main reason for declaring such a simplification is about the complexity of underlying logical mechanisms. Also, in many cases this might not be relevant to the textual similarity. [9] Thus, these final sub-entity declarations of the complexity hierarchy are the code elements for this research.

When we look at the issue of synchronization in the entities and their extended properties perspective, it becomes an issue of correlation among extracted entities and properties of the model, and the generated code. However, such a perspective requires a solid mechanism where the design and the generated implementable code become comparable. This would also mean that, the structural differences in the generated code will be unified in a comparable model with the design model.

Therefore, as entities are assembled from the properties of various kinds, models are assembled from entities, where each of them is in model declaration. Changes in the model have an impact on the design and the generated code. This impact varies greatly on the
effected entities of the model. A major change in one entity of the design might be less effective in comparison to an impact of several smaller changes in several entities. The opposite can be true for different situations. This is heavily related to semantic dependencies of the properties. [6]

Additional to the work of Cicchetti, Di Ruscio, Eramo, and Pierantonio [6], examples of similar approaches in creating a comparable replication among realms can be found in the works of Alves-Foss, Leon and Oman [20], Ratanotayanon, Sim and Raycraft [21], and Diaz-Pace, Carlino, Blech, Soria and Campo [12]. In most of the works, method declarations are also included in properties of a class. However in OMG specifications properties are defined as the attributes of the class, and they do not represent the operations/methods of a class [22].

2.3.2 Capturing Complex Relations

There is a conceptual clash on the definition of properties in this concept. Properties are indeed structural features of a class [22]. For this reason methods such as Accessors and isKindOf are applied on them to achieve desired encapsulation features for an object. Methods are the behavioral features of an object. However, methods are also declared and they are also indexed in the namespace. Moreover, they are accessed by a method notation after an instance of an object is created. Some of these sub-entities are dependent on each other as it can be seen in Figure 2.6.

Alternatively, they might be dependent to other sub-entities of other entities. This is usually represented though more abstracted manner via dependency representation between entities. However, in the implementation phase, dependencies of entities can be directed to sub-entities. This does not have to be on single property and method. As methods get more complex, we can say that more properties will be related with some kind of relation characteristic.
We know that two or more classes can be depended on each other in order to function as desired. These are usually abstracted as semantic relations at the first phase of modeling. Semantic relations as associations are weak compared to other type of dependencies [4]. Among weak association dependencies, there might be more concrete relations between classes such as inheritance, generalization, and aggregation [22]. Additionally, analysis of the classes can be used to decide the multiplicity of the class. This denotes the multiplicity feature of class relations [4].

![Figure 2.5 Entities with Mapping to Properties and Methods Separately](image)

Basically, if we separate the properties and methods in our sub-entity declarations, we are able to catch more complex structures that are underlines in the model. Additionally, for a concrete code generation facility from the model, there must be a well-defined model at the first place. In this well-defined model, relations would be strongly mapped to each other. This mapping is similar to clusters with elements of their own. When a code body is analyzed and it is separated by the level of complexity, it becomes a layered overview which links to matched level of complexity on other compared state of itself or another entity state. In a case that is not well structured, code generation can work, but it would not be successful in creating functional code. Additionally, the model representation will fail to capture the fundamental functions of the code structure. [5]
The reason we discuss the abstraction level of the entities is related to the capacities of a synchronization mechanism. If we create more complex and detailed mapping mechanism in model to code and vice versa, it has the capacity to support more intelligent synchronization though the limitations that the concrete model form obtains. If one would only define a property mapping and declare all elements of an entity in name declarations, it will not be capable of understanding the dependencies in the semantics. But, understanding and correcting dependencies within the semantics requires well-structured model and generation body, which is much harder to implement on practice, harder to automate and it will need to include abilities to parse much of the code body to create the relational links. \[23\]

If one aims to reflect the changes in the code that is generated from the model back to the model, it is claimed that the impact of the changes in the generated code does not always have a major impact on the design. It is stated that most of the changes that is analyzed in the generated code “realm” does not need to reflect back to the model at all. A detailed study on the effects of changes in code-to-design relation can be seen in the work of Hammad, Collard and Maletic \[13\].

\[\]

\[\textbf{Figure 2.7 Mapping Entities with Captured Dependencies}\]

\textbf{Nature of dependencies} is deduced from the findings that are declared above. Thus, one of the two major factors that affect the decision mechanism and the end result of the synchronization attempt is the \textit{Nature of relations which is built in correlation to nature of dependencies}. In this work, \textit{Nature of relations} between two entity states is identified to be
similar with cluster relations in mathematics. Thus, in this research, they are named in the same manner.

2.4 Nature of Relations among Entity States

Nature of Dependency can be briefly explained to be causing that all dependent objects relate to each other in a manner in which:

- Direction of change dictates
- And resolved dependency is mapped.

Therefore, similar to create cluster-like mapping to understanding of functions in mathematics, code entity states are analyzed in four different states to each other which are inferred from two factors. These are \textbf{Functional Relations} and \textbf{Nonfunctional Relations}. These two forms of relations among entity states are collectively named as \textit{Nature of Relations}.

Functional relations that obeys the rule of function mappings fits into \textbf{Injective-Surjective} nature of sub-entities. Thus, a relation that is both injective and surjective at the same is \textbf{Bijective}. A resulting bijective relation without loss of information is the most desired outcome for a synchronization attempt. More detailed explanation for such relations among clusters can be found in [8]. Basically put, in this research, dependency relations are modeled as clusters and then relations are created from analysis of the relation as functions. Functional relation examples have been provided in Appendix 1. Significance of a functional relation comes from the fact that all elements of a domain have a correspondent in a co-domain. Thus, all elements that are included in the process can be mapped to a corresponding element or a sub-entity.

In a direction of relation from one entity state to the other, when sub-entities cannot be mapped to functional relations from the starting direction, this means a domain cannot relate all its members to its corresponding domain in functional fashion. Sub-entities that cannot be mapped in a functional relation are called nonfunctional relations. Thus, relations that do not fit into the provided example situations in Appendix 1 are identified as nonfunctional relations.

2.5 Code Artifacts between Mechanisms and Entity States

In order to create synchronization among the code artifacts, we must first encapsulate their state and find their matching correspondents in compared entity state. Hence, in order to identify code elements and entities between changing states, one must implement a specialized mechanism to the framework. These types of mechanisms are named as
traceability mechanisms.

If one wants to create equalization, then most basic need is to identify and track the related entities and code elements. According to [24], in general, code tracing is used in software maintenance, impact analysis and reuse of existing code. However, this time we must enhance this ability to fit the needs of synchronization.

Synchronization ability specifications and tracing ability specifications are directly related to each other. In order to synchronize any code body in the flow of the development, synchronization mechanisms have to create an encapsulation that refers to the related code bodies. In this research, the theme of entities and elements is used. However, in any definition, code entities should be identified and related to each other for healthy comparison.

2.5.1 Code Artifact Evolution

As it can be seen in Figure 2.9, an artifact description changes to different states during a software development progress. Object X has its initial state that is depicted as state X, which has changed to state Y in time. Unchanged sub-entities of software artifact exist in the intersection of two artifact descriptions. An area that reside in state X either corresponds to areas in state Y but they are not recognizable, thus they cannot be matches as the “same”, or state Y has obtained new sub-entities that do not relate those of state X [10]. These matches are available by using tracing mechanisms. [2]

By altering Bunge’s ontology, which is also explained in [9], to fit entity-element definitions, we can reach to the evolution of the code and its traceability. Where $X = < x, P(x) >$ and $X$ is defined as a unique entity, then $E(X)$ is a collection of code elements (properties defined as elements). Also, let’s say there is entity $Y$, correspondence of entity $X$, which changed in development phases. Where $m$ is the matching elements of evolving objects, intersection depicts the domain of $m$. 

![Figure 2.8 Evolution of Objects](image-url)
Unmatched defines the element states that are altered, added or obliterated.

1. In $\text{Ran}(X) = E(Y)$
2. $\text{Unmatched}(X) = E(X) - \text{Dom}(m)$
3. $E(X)$ is all identified properties of $X$
4. And $\text{Dom}(m)$ is all of the elements that are unchanged.
5. $\text{Unmatched}(Y) = E(Y) - \text{Ran}(m)$
6. $E(Y)$ is all identified elements of $Y$ which is in range of $X$
7. And $\text{Ran}(m)$ is the range of $\text{Dom}(x)$ which correlates to $Y$.

In a traceability mechanism that would regenerate the lost tracing link, basic evolving system function is as below:

(8)

$$\text{For } p \in E(X) \text{ and } q \in E(Y)$$

Among individual elements, one to one similarity search depends on

(9)

$$S(p, q) = \frac{1}{\text{Dom}(m(p))} \sum_{p \in \text{Dom}(m(p))} s(p, m(q))$$

In order to make this formulation work, one must also apply a similarity recognizer. However, this similarity recognizer mechanism can be overlooked in the cases of tracing have been successfully completed, such as behavioral code bodies, as discussed in [13]. In the cases that the similarity catcher is applied, it should identify the
textual elements and this function can be implemented to various forms.

Evolution of the code directly responds to the needs of the synchronization mechanisms, however catching similarities between the software artifacts are a dynamic challenge that changes among environments. As such, entities of this research are a direct relevance of Bunge’s ontology: Actually, entities are declared as substantial individuals with properties of their own. These properties will change in time, as development continues. If object declaration is dependent to model and the corresponding generated entity declarations is altered by developers in later phases of software development, then traceability is needed. [11]

In Figure 2.11, models and textual software code that related to them has been depicted. Model elements reside in model realm while code elements reside in code realm.

A model holds the descriptions for a code entity. A code entity in a model realm is actually a class description. It is an interface or similar scheme that holds the structural components. However, code realm includes both the realized entity code bodies and the expressions that implement the behavior for the software. [14]. Software evolution is a perceived reality which occurs throughout the software development [23]. However, code entities and code elements evolve in different fashions. If we are aiming for a software system that is consistent after the change which might occur at one point of the software development as evolution, we can say that this is also a process which goes thought some steps that is linked to each other via some type of relation(s) between the states of the software product.

In conclusion, a segmentation of the software development phases and separation of target domain are possible within development process. During this process, code elements and code entities are defined, used and changed, destroyed or replaced. This requires classification of code bodies and providing them identification. Hence, it is possible to build the mechanism which can provide the information that nature of synchronization can be used.
Chapter 3 Synchronization Analysis

In this section, code bodies of the code artifacts are analyzed in the light of the information that is provided in chapter 2. As a result, code elements code entities are separately handled because of their different behaviors in the synchronization processes.

In the following, types of changes that can occur on code elements and code entities are identified and their characteristics are used in finding the necessary abilities for reflection of the changes. By using the study of the types of changes and the nature of dependency, nature of synchronization has been identified and explained.

3.1 Analysis of Changes in Code Elements and Code Entities

One can see the code entities as software code body with a segmented part which aggregates together to form the entity. These segmented parts vary in complexity. In order to get the elements of an entity, code body must be analyzed.

Object oriented design features and object relations of Java Platform will be majorly used in this analysis. This is because that platform forms a significant influence on the interested development frameworks.

Because of object oriented design patterns [4], one can correlate the design patterns for a synchronization attempt. Moreover, object oriented patterns bring a beneficial limitation to things that need to be considered while a synchronization attempt, thus easing the burden of synchronization decision mechanisms. Some major limitations include the limitations on file structure, and manifestation model of projects.

Limitations on file structure’s state shows itself is in class declarations. Unless nested, class declarations cannot be in the same location as a file. All classes would have various depth and complexity for the statements and expressions that they hold [25]. When interfaces and abstract classes and methods are also added to the pattern matching, then an overview for the textual body of the code can be achieved. If one would correlate these patterns to entity – element design, then it would be beneficial to correlate entities with classes, and elements with class expressions and statements.

Manifestation model is the way that projects are declared in the development environment [25]. Declaration of elements brings a common form to synchronization attempts. Since all code bodies are textual elements, we need to trace the elements and find the corresponding element of the other entity that we are trying to synchronize.
These will require two abilities for the synchronization mechanism. One of them is to correctly determine the code artifacts. Second one is to understand what would be the impact of change to the elements in order to successfully transform the entity states in a synchronized form. In order to understand the impact of change, we must work on types of changes that occur on the code elements.

### 3.1.1 Identifying Code Elements and Code Entities

In a code body, code entity and code elements can be separated via analysis of scales. Classes, by the nature of the object oriented design, are collectives that hold the code elements. The code in Table 3.1 displays the example code body, GraphicDefinitionImp Java Class. Class declaration clause and every code part that supports the identification of the class are classified as a part of the code entities’ identification, while all statements and methods have been identified as code elements.

Depending on the programming language’s limitations and grammar design, there are different code body parts that will define the code entities and code elements. However, in this research one definition will stay true in general. A code entity, like a class in object oriented languages, is a code body that manifest though other code bodies as the main entry point to rest of its code elements, independent of the behavior of the software code. On the other hand, a code element is either a statement or an expression entry point, which can be called as a completion of an expression, but its sub elements are not independent from its behavior.

<table>
<thead>
<tr>
<th>Entity and Element Segmentation</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Code Entity Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>package org.xtext.car.car.impl;</td>
</tr>
<tr>
<td>import org.eclipse.emf.common.notify.Notification;</td>
</tr>
<tr>
<td>import org.eclipse.emf.ecore.EClass;</td>
</tr>
<tr>
<td>import org.eclipse.emf.ecore.impl.ENotificationImpl;</td>
</tr>
<tr>
<td>import org.xtext.car.car.CarPackage;</td>
</tr>
<tr>
<td>import org.xtext.car.car.GraphicDefinition;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cluster of Code Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class GraphicDefinitionImpl extends DefinitionImpl</td>
</tr>
<tr>
<td>implements GraphicDefinition</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>protected static final String IMAGE_EDEFAULT = null;</td>
</tr>
<tr>
<td>protected String image = IMAGE_EDEFAULT;</td>
</tr>
<tr>
<td>protected GraphicDefinitionImpl()</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>super();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
protected EClass eStaticClass()
{
    return CarPackage.Literals.GRAPHIC_DEFINITION;
}

public String getImage()
{
    return image;
}

public void setImage(String newImage)
{
    String oldImage = image;
    image = newImage;
    if (eNotificationRequired())
        eNotify(new ENotificationImpl(this, Notification.SET, CarPackage.GRAPHIC_DEFINITION__IMAGE, oldImage, image));
}

public Object eGet(int featureID, boolean resolve, boolean coreType)
{
    switch (featureID)
    {
    case CarPackage.GRAPHIC_DEFINITION__IMAGE:
        return getImage();
    }
    return super.eGet(featureID, resolve, coreType);
}

...
divides tiers for classification. These divisions can be easily formed to already structured platform languages such as Java, C or Objective-C; however it is needed to be identified for DSLs that is constructed by the developer. Therefore, one can come to a conclusion that there is a **matched level of complexity** among code artifacts.

### 3.2 Types of Changes on Code Artifacts

At the fundamental level, code artifacts are constructed from textual structures. When we are looking for the possible changes in textual bodies of code artifacts, we can separate the code body according to level of complexity. This enables conditions which are ideal for comparison among the captured states of the matched complexity levels. This matching is the **defined relation** for synchronization attempts in similarity. Two states can have different levels of similarity, with the highest similarity level defined as "same".

In this research, **sameness** means an entity or an element is completely identical between two comparable states. This means that, any change to the code body’s logical and semantic structure will break the sameness of two states of entities or elements. Thus, sameness state is less likely to pursue compared to being matching by a defined relation. Moreover, the matched level of complexity is mainly differentiated by matching of code elements and matching of code entities of the equal hierarchy.

For constructing and classifying the types of changes, [11] and [26] provided the influence for separation in type of changes of code elements and of code entities. For more of this discussion, please refer to conclusions from section 5.5.2.

In order to understand the behavior of the code body depending on the level of complexity, types of changes that can occur on the code elements and higher complexity level code entities have been studies. The findings from this analysis are then used in identification of the limitations, expected conditions and exceptional conditions on the code body parts.

#### 3.2.1 Types of Changes in Code Elements

By their nature, code elements are in a position that further synchronization attempts to deeper, more nested code body parts requires the support of a behavior analysis. However, synchronization of the behavior itself is semantically challenging and rather unnecessary for synchronization attempts [13]. Behavior of a code element is constructed from statements, expressions and fields that are prone to semantically complex changes.

In this research, it is claimed that synchronizing a method’s behavior is sufficiently done without analysis of the behavior itself. The reasons are below:
(1) Statements and expressions are prone to complex semantic changes. These changes can either become successful, or they can create an inner conflict in the code element itself. However they are not interesting for analyses for outer scale impacts. [13]

(2) Moreover, code elements are identified as dependent to their code body. This reasoning lies from the fact that the return values of the code element are a result of code body that defines its expressions. However, for the code element itself, the identifier has three major features that define it. These are return type of the code element, name of the code element, and if there is any, types of parameter(s) of the code element. Such as

```java
public void setImage(String newImage){…}
```
or

```java
char c = ...
```

At above example, public classifies the access rules for the code element, thus, it is important for Java environment. However more solid identification for the code element comes from the return type of the method `void`, the name of the method `setImage` and the parameter type of the method (String newImage). In the parameter section, general conviction indicates that parameter name itself is not valid enough to indicate the identity of the method, but rather the parameter type is of importance. In the second example, `char c` is able to represent the code element alone. Explanation for this classification can be found in [4] and [9].

One of the claims of this research is that, one can separate the types of changes on code elements are as follows: Different types of changes are deducted from the nature of code elements in three major ways. These are obliteration of a code element, substitution of a code element and modification of a code element. However, modification of a code element can also be separated to three different natures. These are subtraction in code element’s code body, addition to code element’s code body and alteration of the code element’s code body.

Although modification can be separated to these three minor sub categories, if the behavioral code is not interesting for deepness of synchronization attempt, this information is not fundamental for the mechanism.

Definitions of these keywords are below.

**Obliteration**: Code element’s identifying clause is missing. Also, there is no found track for an editing option for the name of the type or the method. Thus, method is considered as erased, or obliterated.

**Substitution**: Code element’s identifying clause is replaced by another code element. Substitution is strongly related to Modification, with the difference of code element’s name. In obliteration, method has been totally lost; in substitution, either name or return type or
parameter features of code element have been changed.

**Modification:** In modification, either because of a substitution or because of a change on the textual body of the code element, textual form has to be changed. Modification has three different versions.

**Subtraction:** If the code body is a block of text, then some code has been erased from code block, without disturbing the pattern of the previous code, thus the defined relation information is kept. Hence, this enables textual integrity that can relate to previous code version.

**Addition:** If the code body is a block of text, then some code has been added to code block, without disturbing the pattern of the previous code, hence enabling textual integrity that can relate to previous code version.

**Alteration:** Code body has been changed and it is not possible to understand whether if previous code body and new code body was related.

An obliteration of a code element is easier to understand compared to substitution or modification. However, there is an overlap between obliteration, substitution and modification. Understanding that there was an obliteration of the code element, but not a substitution is intertwined. A both substitution and modification means a destruction of code integrity. A heuristic code identification mechanism would perceive all changed code element identifiers as obliterated. Also, substitution and modification overlaps in code change attempts that would disturb the code element identifiers. Distinctive features of these two types of changes can be easier to understand with the following scenario.

Let’s consider a situation that a developer has replaced a method in a class by another method from either another class or a written a new method. If developer wants to replace the method in object instances that calls the method too, then substitution is the desired outcome from the change operation. A substitution can also lead to a modification in cases which the modification consists of changing the identifier features. However, as deducted from [23], changing code element identifier’s features has the possibility to lead to consistency and dependency problems.

3.2.1.1 Conclusions on Changes on Code Elements

Because of the overlap between obliterations, substitutions and modifications, *ability to create equivalents* plays a major role in determining the importance of deducing which of those types of changes has occurred among code artifact states. Depending to the decisions for assessing a synchronization attempt, code elements can be separated to various forms of dept of success in synchronization to the most desired outcome. Three points are important to such
ability:

(1) **Ability to deduct patterns from behavioral code body:** As explained, at least in object oriented design, this ability is not interesting for most cases of synchronization attempts, but it has effects in perception of dependency among code elements and code entities that owns those code elements.

(2) **Ability to follow edits on the code element identifiers:** This means that, the synchronization mechanism owns and implements a design that can relate code element identifiers even after they have been changed. Thus, a change identifier can be followed though this editing history.

(3) **Ability to catch developer’s intention:** When there are multiple possible outcomes from the sequence of edits, it is a requirement to understand the intention of the changes. Unless there is only a single path that leads to the editing attempts, mechanisms needs to question the developer for their intentions.

### 3.2.2 Types of Changes in Code Entities

Code entities are composed from code elements. Code elements that are bound with code entities carry their own types of changes. On the other hand, since code entities are bound in a hierarchy and also since they are consisted of a cluster that owns identifiable separate code bodies, nature of code entities carry some changes to code elements. A code entity can be consisted of other code entities, and this may or may not change its level of complexity. Such as, a single class is a basic code entity, while a class with a nested class has still the same level of complexity but with more interchangeable code parts. However, a single Java Project is a bigger bulk of code structure that exceeds the complexity level of other two. It has a different scale and with additional attributes. Hence, it has a different matched level of complexity. Influence for these classifications is explained in section 5.5.2.

Important thing to notice is, in a synchronization attempt, unless dependency clashes occur, code elements that belong to a code entity is isolated from the effects of changes in other code elements that belong to code entity and code entities’ identifiers. However, opposite is no true. Code entities must reflect all changes that occurred on them to correspondent pairs in order to keep the level of similarity close to sameness.

Since code elements are independent from their code body’s behavior, and since they can be identified with their sub structures, they can manipulate their sub-entities. Additionally, code elements can change types that code entities inherit from code elements. This is a direct result of object oriented design pattern in structure of objects schemas [4] and also directly relates to project structures that have higher level of complexity [23].
In conclusion, additional to three major types of changes that are entity level obliteration, substitution, and modification; elements and entities react to changes among each other. A sub entity can survive outside of the code entity, thus it can be transferred to other entities as a substitution or an addition modification while it is registered as a subtraction modification in the previous entity. Entities can obtain first-born sub-entities or lose existing sub-entities. Lastly, sub-entities can change their appearance order within the code entity.

On the code entity’s identification level, code entity itself carries all three major possible change actions, obliteration, substitution and modification.

**Entity Obliteration**: Code Entity’s obliteration means that entity is textually destroyed and its elements are either transferred to other entities or they are also obliterated. Entity’s identifying clause is missing. Also, there is no found track for an editing option for the name of the entity name or other identification features is dropped. Thus, entity is considered as erased, or obliterated. However, entity’s elements can continue their existence in other entity or entities.

**Entity Substitution**: Code entity’s identifying clause is replaced by another identifier. Entity substitution is strongly related to obliteration. Difference of substitution from obliteration is that, a new identification replaces the manifestation of the entity with the new declaration. Also, relation between entity modifications and entity substitution is lessened because of hierarchic order that marks the separation of identities is visible.

**Entity Modification**: Additional to in-element modifications and substitutions, in-entity modifications appear.

*Element Subtraction*: An element in the entity is obliterated.

*Element Addition*: A new element body has been added to Entity.

*Element Alteration*: An element has gone through a substitution, or an element level modification.

*Reordering*: Order of the sub-entities was altered.

3.2.3 Changes in Code Entities and Refactoring

One of the major reasons to cause a synchronization necessity is *refactoring* in one of the related code elements or entities. Refactoring does not affect the behavior of code element [11]; therefore it fits to most cases of synchronization attempts. Code entities carry the types of changes in code elements to a different scale. These changes correlate to refactoring attempts in a class. As explained in [11], these refactoring attempts are separated to seven actions. These are “Replacement”, “Extraction”, “Merge”, “Split”, “Merge into a new class”, “Recombine”, and “Recombine into new classes”.
If one were to take account of the similarities for refactoring and synchronization attempts, then, there can be a classification for which of the refactoring cases are interesting for a synchronization mechanism for minimum requirement.

Basically, all of the seven cases are interesting for a synchronization mechanism with one difference. Synchronization will mean to transfer the textual and logical changes that refactoring applied to a code entity, and taking these changes and applying them to another code entity state that will relate to changed code entity. This is done in various forms as it was discussed in types of changes on code elements and code entities. Thus, refactoring and synchronization are related but they are not the same. If necessary, Identifiers must be synched to related definitions after refactoring.

3.2.4 Conclusions on Changes inside and among Entities

If one would perceive the types of changes in code element as basic change types, and if code entity is an order of code elements, then it can be claimed that by using the basic types of changes in different orders, any change request can be implemented. This is achievable though a tracing ability \[9\]. In this research, it is claimed that this is true as long as it is in the bounties of factors in nature of synchronization.

It is concluded that, in order to achieve synchronization of the identified type of changes on code entities, a synchronization mechanism must deploy three major ability classifications. These are \textit{ability to create equivalents}, \textit{ability to trace elements and entities}, and \textit{ability to understand the impact of change}. It must be noted that code element level abilities are still necessary and thus it is inherited to code entities’ necessary abilities for their code element resolutions. Excluding the \textit{ability to deduct patterns from behavioral code body}, other two abilities are plays a role in code entity level synchronization attempts. These are explained in the following.

Entities are dependent on to the (1) \textit{ability to create equivalents} on themselves in order to determine the scale and dept of a synchronization attempt. Compared to element level ability assessments, code entities require more complex structures with more complex abilities. Especially, (2) \textit{ability to catch developer’s intentions} is an important functionality for resolution of complex situations. This is because when cases of overlapping entity relations are found, these are cases that developer has to make a decision over one other. Thus, it is a requirement to understand the intention of the user from the synchronization attempt. Therefore, mechanisms need to question the developer for the final intentions. This condition is mostly required in two conditions. One of these is nonfunctional relations among entity states, and the other one is when non-injective surjective functional relations appear.
Other code entity level necessary abilities are:

(3) **Ability to capture entity states**: An entity or an element has to be compared to a different state of either itself that has different time value, state condition or a relatable but initially different another entity. Thus, there is a necessity to capture these states. This can be achieved in various forms. For example, by following the either in an edit history form from “Ability to follow edits on the code element identifiers” as in [26] or by capturing snapshots for versioning as in [11], this can be realized. Additional forms of similar capabilities exist in related studies, as stated in [2].

(4) **Ability of trace elements and entities**: An entity has to map its sub-elements and its own identification to comparable state by selected from of traceability. Traceability will require to either inserting the traceability information to code body manually or automatically. Another way to achieve traceability is infer the traced artifacts “on the fly” or deduce the relations by recreation relations between states. Depending of the abilities of the tracing ability, synchronization methods can be enhanced to cover a greater intelligent workflow. As example to such approaches, code entities can be identified in a case of change as an evolving system [9], or numerical methods can be used to recreate the tracing links that can be lost through development [27]. However, it is suspected that there is an upper limit with methods to retrieve the traceability [27]

(5) **Ability to understand the impact of change**: Impact of change relies heavily on both to Ability to capture entity states and Ability of trace elements and entities. In brief, any change that occurs in the entity has different scales of disturbance on the consistency and integrity inside and among code entities. Thus, in order to successfully conduct a synchronization attempt in code entity level, these are the needs to analyze such impacts. Impact of change must be considered on four main factors

- Compared states of Entities: Which entity states are relates?
- Time of Analysis: When was entity state registered for comparison?
- Correct decision: What synchronization behavior to conduct?
- Implementation of Decision: How to generate the Synchronization Attempt?

In later phases of this research, these findings are used in determining the necessary mechanisms for design and realization of the synchronization mechanism.

### 3.3 Nature of Synchronization

Behavior of synchronization differs according to four different variables for synchronization nature. First two is about clarifying the limits of synchronization. Other two
is about the deciding the outcome of the action. These are listed and explained below:

3.3.1 Limits of Synchronization Ability

Synchronization is limited by two major factors. These are the nature of the entity state, and the ability of the synchronization mechanism or the state itself to create an equivalent form of the correspondents. Nature of entity state is about whether if the entity state is able to allow itself to be changed or not. On the other hand, ability to create equivalents is mostly related to traceability and code generation ability within the mechanism.

3.3.1.1 Nature of Entity States

This variable dictates whether a state can be changed in any manner. If a state is static, then the state in comparison to other state cannot be changed. It can only be the start point for direction of change, thus in the dictating position. On the other hand, if a state is dynamic, it can change itself. Hence, it can accept the demands of synchronization or it can be in dictating position.

Nature of Entity states is related to two main factors. Other minor factors also exists, yet in greater picture, there two factors are fundamental. These are Relationship Nature of code entities and Mutability. Relationship Nature is the dictated limits for a reflection of changes to an entity state from another. Relationship nature overlaps with dependencies among entity states, and therefore it is discussed as a separate condition in section 2.3. Mutability comes from a common programming practice. Independent from dependency relations, an entity state can declare that some sub-entities or a code entity itself is immutable. Both Relationship Nature, and Mutability can be broken if consistency among dependency relations are not interesting for the synchronization attempt. However, in this case compared states are forced to become dynamic. This decision is prone to create malfunctions.

3.3.1.2 Ability to Create Equivalents

In order to create equality among the entities, elements of the entities must be either abolished or generated. In order to generate new elements which equals to other entity or entities, there must be a mechanism to put equivalent elements which can be correlated between entities.

One can assume that a mechanism don’t have complete coverage to create software elements which are equivalents of the other element in every situation. Moreover, this ability would be partial, which would mean some entities that are required to generate for a complete synchronization cannot be supported while other entities can be regenerated. These conditions
are interrelated with the desired equality result and nature of entity states. Synchronization mechanism would need to follow the acceptable outcome of the controller and started of the synchronization attempt. This started must declare the desired level of equality for the synchronization mechanism to operate successfully.

### 3.3.2 Desired Result

Synchronization process has to be configured in order to behave the way that is desired. This necessity is born because two states of compared code bodies can achieve a synchronized relation in various ways. Such ways include e.g. loss of information to bring an equalized form, partially equalizing code bodies to each other, a complete equalization by generation of all corresponding elements. Therefore, behavior of the synchronization process is a key factor that determines the outcome.

The desired result can be assessed by two factors. These are the desired level of equality at the end of a synchronization attempt, and the direction of change during the synchronization. Desired level of equality is identified by two parameters. These are whether if the result is allowed to be partial or should it be a complete synchronized state at the end of the synchronization, and whether if it can achieve the partial or complete synchronization by loss of information as data or not. On the other hand, direction of change is about which entity state should become equal to the other, thus it is about which side dictates the changes from their code body, while the other side accepts the changes on their code body.

#### 3.3.2.1 Level of Equality

Equality at the level of sameness is complete similarity of two different states. This similarity varies according to structure of elements. For the largest pool of coverage among entities, equality can also be assessed by linkable results of two elements in entities. However, shift at description would change the requirements in ability to create equivalents.

In this research, bijective state of two entities that would link to each other as correspondents at the result of synchronization is accepted as “complete equalization” synchronization. However, in conditions where this is not a requirement for synchronization to succeed, two entities can be equalized partially instead of a complete equalization. Decision of level of equality depends of ability to create equivalents, or generation ability. In cases in which a consistent equal element cannot be created or not desired, a partial synchronization can be preferred instead of complete synchronization. As a result, one of the parameters that identify the level of equality is whether if a partial synchronization result is accepted as an outcome or not. This brings a division between a complete synchronization attempt and a
partial synchronization attempt. It is named as **completeness** attribute of the desired result.

Another feature of desired result is whether if compared states have to lose information in order to create synchronization. This information is mostly about the losses of sub-entities or relation links. In cases where synchronized artifacts are able to create complete synchronization and the desired level of equality states that they can lose data to be completely synchronized, sub-entities would be depicted. However, in cases where desired equality demands that no information should be lost at the end of the synchronization attempt, then either the process would be terminated, or process should be altered to new condition.

In code elements, desired level of equality is possible in three ways. One of them is generating and replacing equal or similar code body which has the same behavior. Second one is replacing the code body with corresponding textual body. And last one is demanding the user to enter code that would be equal or similar to be synch code.

As a final note for assessment of the desired result and the synchronization outcomes, it is not known if equality of the generated code or manually fed code can be ensured by using similarity calculations during code generation, which are aimed to be implementable in synchronization mechanisms. However, it is known that similarity calculations are not aimed to reflect changes. Thus they are not used in creating similarity, calculated to an extent which was fit to desired result. They are aimed to calculate if two separate code bodies are somehow match able to a behavior, context, or notation order. Therefore, they can be used to check the generated synchronized results to understand their level of equality, but they are not fit to create the results in current status. For this reason: *A holistic outcome assessment is found more fitting for calculation of possibly to achieve desired synchronization outcome.*

### 3.3.2.2 The Direction of the Change:

Direction of change is the desired direction which is decided by the starter of the synchronization attempt. Synchronization mechanism assesses the possibility of the synchronization within given desired direction.

For automated mechanisms, direction of change can be inferred from the nature of a state. In any condition, independent from their state nature, entity states can demand a stance while entering the synchronization process. This stance has options of either “to accept” or “to dictate”. Declaring an accepting stance would mean that the entity state will try to generate equivalent code elements and sub-entities on itself, while on the other hand dictating stance declares that the entity will not accept changes on itself but dictate its form on the other side.

In order to activate a synchronization attempt, at least one state must be set to acceptance stance. When only one side is in dictating stance, and the compared entity state is in accepting
stance, then direction of change is decided to be from the dictating side to accepting side. After the analysis of the relation, resolution of the synchronization attempt can be achieved in one or two steps. If both states are in accept stance, and if there is ability to create equivalents in both states, and also if they are both in dynamic nature or forced to become dynamic, then direction of change has the option of becoming both directions. These result in more complex type behaviors. This can involve two or more steps for resolving the synchronization attempt.

Nature of states is more decisive in shaping the synchronization nature. However, in conditions where nature of entity states is not enough to lead the synchronization attempt, desired direction of change is helps to deduce the behavior. Moreover, when desired direction of change is not compatible with the desired synchronization attempt, then synchronization needs to either terminate or it needs to assess if desired level of equality is achievable by changing the scope of synchronization according to impact of change in synchronization attempt.

3.4 Using Nature of Relationship and Nature of Synchronization

In conclusion, by using the given information on dependency, one can conclude that there is a correlation between **Nature of Synchronization** and **Nature of Dependencies**. Entity dependencies resolve themselves very similar to domain to domain function relations [8]. Hence, it is concluded that, *one can create a holistic overview that deducts the expected result of synchronization attempt by analyzing this relation between two compared entity states through functional relation analysis.*

Dependency among code elements dictates to **nature of entity states, thus to the code entity states**. This is also linked to limits of synchronization ability, because the limitations that generation ability has to endure for synchronization of the behavioral code body equivalents, as mentioned in code element and code entity comparison. As a result, captured entity state owns three major parameters, which are independent from the behavioral code limitations. These are the State Nature, Direction (Stance for Change Behavior) and Generation Ability. These are used as state characteristic parameters for synchronization behavioral calculations in later phases of the research.

**State Nature:** Can be set to *Complete* or *Partial*

**Direction (Stance during Synchronization Attempt):** Can be set to *Dictate* or *Accept*

**Generation Ability:** Generation ability for a defined relation pair on the entity state can be “*Yes, it exists*”, or “*No, it does not exist*”.

Additional to characteristic parameters, there are four conclusions that dictate the design
of the synchronization mechanisms. These four conclusions are deduced from the discussions on nature of synchronization, and nature of dependencies. These points must be considered when planning the limitations and expected conditions for a synchronization mechanism.

1) Model to Code change direction does not directly affect the behavior of the code elements in general. However they create dependency inconsistency when a code part that reflects the behavior from another code element is lost as the impact of change.

2) Code behavior can be analyzed by similarity studies. However, code behavior does not create heavy disturbance on dependency consistency. In most cases, it is confined as textual consistency issues. Therefore it can be stated that, changes in the code is not in need of synchronization, thus it is not reflected to model structure in most cases. Exceptions to these are cases that result in inconsistencies on the dependencies in the code element’s body. This means to create dependency conflicts on statements, expressions and field declarations in a way that creates a need for feature mappings to resolve the conflicted states. When this function is not possible, then complete transfer of behavior is not possible. As a result, synchronization must be either aborted or should be accepting partial synchronization outcomes.

3) When model to code relations carry constraints on identifiers and scheme declarations, or validity and formatting options for related code entity declarations, model to code synchronization has to include dependency conflicts that disturbs declared constraints, or it must be aborted. However, in most cases, this is not a possibility. This is because the code declarations already cannot go a change outside of model constraints. This scenario is only valid if the model declarations are forced to change via synchronization mechanisms ignorance to relationship nature and nature of entity states.

4) When relationship nature is followed, it would be resolved as injective and surjective relations among entity states of code bodies which has the same matched level of complexity. This complexity level is not about the mass of the code body, but it is a reflection of matched structural hierarchic equality. Such as project to project, classes to classes, expressions to expressions
Chapter 4 Synchronization Behavior

Synchronization behavior is the negotiation process among captured and compared entity states in order to determine whether if it is possible to create the desired level of equality as a final result at the end of the synchronization attempt. In this research, synchronization behavior has been constructed in a methodology that is sufficient to create a decision mechanism for platforms that can be abstracted to code-entity and code element design.

Use of *Nature of Synchronization* on code element and entity states reflects to the behavior in a synchronization attempt. By different conditions that dictate the nature of synchronization, one can conclude four major factors. Among the matched code entities which have the same complexity level, synchronization behavior follows the nature of synchronization. These can be categorized in two decision parameters.

One of these decision parameters is whether if the compared entities are allowed to partially synchronize. This condition has been categorized as complete or partial synchronization. Other decision parameter is whether if synchronization process results with loss of information in code entities. This loss is holistically determined by other three factors; *nature of entity states, ability to create equivalents,* and *direction of the change.* Partial synchronization gains more meaning when Relationship Nature of entity states is considered. The outcome of the synchronization attempt differs in collected data either from the user’s preferences or information from deduced states.

Thus, a synchronization attempt’s outcomes can be categorized as:

1. **Complete Lossless Synchronization**
   
   In this case, a synchronization attempt results in bijective relation among entity states and it can reach to this condition without losing data as information on related states.

2. **Partial Lossless Synchronization**
   
   Synchronization attempt cannot result as bijective; however it keeps all information while keeping consistent resolution to correspondent direction.

3. **Complete Lossy Synchronization**
   
   Synchronization attempt results in a bijective relation among entities, however in order to achieve this condition, it sacrifices some of the data from an entity state or more states.

4. **Partial Lossy Synchronization**
   
   Synchronization attempt cannot result in a bijective state; however it loses information in order to bring two states in a more synchronized form. These are conditions with highest level
of complexity in their synchronization attempt.

By using this categorization, one can also deduce the desired results for the synchronization attempts. As explained in nature of synchronization, desired result is separated to two factors. One of these factors is level of equality, the other one is direction of change. In a synchronization attempt, synchronization behavior sets the desired result to be achieved via feeding the desired level of equality. Thus, the categorized list of outcomes is also the list of desired level of equalities. Mechanism uses the direction of change information to validate whether if the desired level of equality is possible to be achieved. Lastly, the agreed direction of change has to negotiate with state natures and generation ability in order to deduce if the desired outcome is possible.

It must be stated that, as a design decision, above list of desired results also provides the level of acceptability. Such as, if the desired result accepts the characteristics of complete lossy synchronization, this would mean that all behavioral analysis results which gives complete lossless, partial lossless, and complete lossy conditions would be accepted, while partial lossy results would be rejected. However, this condition can be altered by changing the order of the desired result. For example, if completeness is more desired than preserving data, then complete lossy condition can be set to second place at the order, while the partial lossless is set as the third desired condition. Moreover, desired condition can be set to be confirmed by the user of the synchronization method, thus ensuring the correct behavior for each attempt; however this creates a necessity for intervention to the synchronization process and breaks the level of autonomous control of the synchronization process.

Negotiation among entity states follows a refined order. First negotiation among the parties is on direction of change. Two sides register if they want to accept or dictate the change attempts. Second negotiation is on whether if the agreed direction of change is compatible with the nature of the entity states. If nature of entity state is compatible to receive the changes, it is required to check the generation ability for corresponding sub-entities. If generation ability is confirmed on accepting side(s), then relationship nature of compared states are deduced to understand their functionality. If there is a conflict during negotiating parameters, it is sent either to complex behavior cases and then resolved as complex type behavior, or synchronization attempt is declared as “not implementable”.

Once the functionality among states is deduced, synchronization attempt enters to two distinctive processes. A synchronization attempt can either stage a functional process or a nonfunctional process. In this research, for a functional process, a complete lossless synchronization has been claimed to be the best outcome from a synchronization attempt. By trying to decrease the complexity type of the relation, synchronization behavior tries to
find out the way to achieve a result that is closest to the initially declared desired result.

When assessed among each other, relations of the compared states can be classified in different level of equalized states. That is, being in a functional or a nonfunctional relation with each other has an impact on how already equal the compared states are at the beginning of the synchronization attempt. This observation has been used in creation of the holistic result calculations for this research. It is claimed that, synchronization level of compared states can be listed in a complexity hierarchy, and a synchronization level can be increased by decreasing the complexity level of the relations. Thus, from the initial analysis of the state conditions, types of behavioral conditions are classified and they are listed as level of complexities of a synchronization behavior. As a result, synchronization mechanisms can also be classified in different levels, according to the support that they provide to resolve the level of complexities.

In order to achieve such an outcome, synchronization behavior uses combinations and compositions in synchronization attempts during synchronization process. For example, non-functional processes are expected to follow a functional process to deduce the outcome of a synchronization attempt. Therefore, if possible, the methodology constantly tries to achieve Complete Lossless synchronization outcome within a functional relation. However, all attempts are not bound to achieve this result, and thus when desired level of equality is achieved and further level of equality is not possible, synchronization attempt is accepted as ready to initialize for implementation.

Table 4.1 Chart of Functional Calculation Parameters

<table>
<thead>
<tr>
<th>Functional Relation Calculation</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
<th>Injective</th>
<th>Surjective</th>
<th>Data Loss</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Static / Dynamic</td>
<td>Dictate / Accept</td>
<td>Yes / No</td>
<td>Yes / No</td>
<td>Yes / No</td>
<td>Allow / No</td>
<td>Complete / Partial</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static / Dynamic</td>
<td>Dictate / Accept</td>
<td>Yes / No</td>
<td>Yes / No</td>
<td>Yes / No</td>
<td>Allow / No</td>
<td>Complete / Partial</td>
</tr>
<tr>
<td>Able to Synch</td>
<td>Yes / No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loss</td>
<td>Lossless / Lossy</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete Loss</td>
<td>Complete / Partial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Stop / Continue</td>
</tr>
</tbody>
</table>

Table 4.1 displays the related information for functional relations. Table 4.2 displays information required for resolving non-functional code to functional code.
In this research, it is claimed that above table gives an overview of the synchronization behavior for **functional relations**, and they can be used as a scheme to create any type of synchronization mechanism. Entity relations that do not fit into functional relations are resolved in a nonfunctional synchronization process. Functional relations are provided in Appendix 1 for reference.

It can be stated that, all synchronization mechanisms have to be converted from nonfunctional relations to functional relations to achieve best possible synchronization level for their condition. Such conversions differ in sequence of the data collection about synchronization attempt, which also ends up determining the outcome of the synchronization attempt. Data collection is either done directly or indirectly from users of the mechanism.

**Nonfunctional relations** have to be resolved before a functional relation can be mapped correctly on to entity states. To resolve nonfunctional relations, synchronization mechanism tries to create a corresponding sub-entity for every sub-entity of the dictating state. However, there are cases that functional relations cannot be achieved. The chart below displays the related information that affects the outcome of the resolved attempt. In this case, Nonfunctional synchronization attempts are required when synchronization mechanism detects that synchronization direction has nonfunctional relations. Nonfunctional relations are provided in Appendix 2 for reference.

### Table 4.2 Chart of Nonfunctional Calculation Parameters

<table>
<thead>
<tr>
<th>Nonfunctional Relation Calculation</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
<th>Equality</th>
<th>Multi-Respond</th>
<th>Data Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Static / Dynamic</td>
<td>Dictate / Accept</td>
<td>Yes / No</td>
<td>Complete / Partial</td>
<td>Yes / No</td>
<td>Allow / No</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static / Dynamic</td>
<td>Dictate / Accept</td>
<td>Yes / No</td>
<td>Complete / Partial</td>
<td>Yes / No</td>
<td>Allow / No</td>
</tr>
<tr>
<td>A Injects B</td>
<td>Yes/ No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B injects A</td>
<td>Yes / No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Separate Matching</td>
<td>Yes/No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&gt; Nonfunctional</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>&gt; Functional Relation</td>
<td></td>
</tr>
</tbody>
</table>

If it can, all sub-entities are mapped to a relation on the correspondent entity state during the resolution of a nonfunctional relation. By doing this, it tries to construct a functional outcome by injection of the correspondent sub-entities. This requires that, the dictating state must accept partial synchronization results during the nonfunctional resolution process.
Therefore, it can be stated that all attempts that result in nonfunctional relations are also partial lossless synchronizations and if functional relation is not followed, partial complete synchronization level is a satisfactory default outcome.

Additional to direction and ability parameters, both functional and non-functional relations possess condition parameters specialized for their process. For functional relations, injective and surjective relation characteristics are important. Moreover, the provided desired results for singular entity state conditions must be in consideration. When analyzed, it can be seen that injective condition has an impact on data loss condition, and surjective condition has an impact on completeness of the synchronization outcome. When assessed together, surjective and injective condition parameters also signal a nonfunctional relation, thus it directs the synchronization process type.

On the other hand, for nonfunctional relations, a different effect is analyzed. Since a nonfunctional relation is not aimed to result in a final synchronization step, it checks if complex type conditions and deep synchronization mechanism are necessary or not. Thus, a nonfunctional condition checks if there as a occurrence of a “one to many” or “many to many” relation among corresponding elements. This check result is then reflected to determine the requirement to use the techniques designed for dealing with such conditions. Moreover, a necessary condition for another complex type behavior is checked thought the control of the desired result parameters of individual entity states. This check determines if there is a situation in which separation of entity states affects to individual code parts.

It must be noted that, both nonfunctional and functional processes are all designed to be in all or nothing behavior. Reason for this decision is to guarantee overall consistency in a synchronization attempt.

Synchronization attempts also can be categorized to two distinctive behavior complexity levels. In this research, behavior complexity is segmented to Basic Behavior Types, and Complex Behavior Types. Basic behaviors are the cases where there are no hierarchic statement conflicts in negotiations among nature of state, direction, or the generation ability. Thus they can be solved in a single functional process or a simple combination of a nonfunctional and a functional process. These behaviors are well ordered decision flows that lead to plain, unambiguous decisions. On contrary, complex behavior are cases where negotiations between entity states fail because:

- Conflicting States Natures with Directions
- Partial Generation ability
- Multi Responses

Nonfunctional processes are designed to be used in combinations. Combinations are a
broader analysis mechanism, constructed from either nonfunctional or functional or both processes. They are sequenced together to satisfy the desired level of equality within the closest reach to lossless complete synchronization outcomes. Combinations lead to two major results. Firstly, they enable solutions for complex behavior analysis. Secondly, they are used in basic synchronization behaviors that cannot be achieved in single functional process.

In conclusion, synchronization behavior provides all necessary parameters and relational information in order to create a synchronization decision mechanism that can be implemented to frameworks that are fit to code-entity and code element design. Thus it can be used as a methodology to create any such system.

4.1 Basic Behavior

Basic behavior covers the simple, well constructed cases. These cases limit the possible parameter values in determining synchronization behavior. These limitations are drawn from dependencies among nature of synchronization factors.

In basic behavior, all synchronization attempts can be resolved to the level compatible with the desired result either in single synchronization process of a functional relational calculation, or it can be resolved in two steps which starts with a nonfunctional resolution preparing the conditions for a single functional calculation during the synchronization process. Therefore, these are identified as basic complexity conditions which are fundamental for synchronization behavior. These two basic conditions are classified as the following:

4.1.1 Type 1 Basic Behavior

Type 1 basic behavior is synchronization attempts that can be analyzed in a single functional process between two entity states. Figure 4.1 is an example of a type 1 basic behavior. In this example condition, the element \( m \) is able to be generated on the co-domain entity state B. Thus, an equivalent of \( m \) has been generated on the entity state \( m \) during a functional process. Moreover, other correspondent elements, \( k \) and \( l \) are also checked on equivalence, but in this case they were not found necessary to be equalized, since they were already equals.

For here, it can be stated that the actions that occur during the basic behavior is the following:

- Create the domain elements from the dictating side that does not have a correspondent on the accepting side.
- If the corresponded exists, check and if necessary re-equalize the code according to set level of equality conditions for code elements and code entities.
Table 4.3 Example Case for a Functional Relation

<table>
<thead>
<tr>
<th>Functional State Nature</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
<th>Injective</th>
<th>Surjective</th>
<th>Equality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Partial</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Partial</td>
</tr>
<tr>
<td>Able to Synchronize</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loss</td>
<td></td>
<td></td>
<td>Lossless</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complete</td>
<td></td>
<td></td>
<td>Complete</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td></td>
<td></td>
<td>Complete Lossless Synchronization</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1 Functional Relation Synchronization Attempt

Briefly, it can be stated that specifications and conditions for ability to create equivalents are also true for capturing and identifying the level of equality among code bodies. These are direct reflection of type of changes that can occur on the code body. For example, if code element’s behavioral code body is not analyzed for the level of equality, then this code body parts are not to be included in the comparison process. Hence, synchronization process will be blind to the changes in such code body parts.

**Starting conditions:** All starting conditions for basic type behavior 1 can be found in Appendix 5

4.1.2 Type 2 Basic Behavior

These synchronization attempts can be analyzed in single nonfunctional and functional process combination. Figure 4.3 depicts the first resolution of such a combination attempt. Code elements $b$ and $a$ are able to be generated on the co-domain entity state, thus they are injected with an equivalent code.

As the outcome of this process, a functional relation between entity state A and entity state B is formed. As the second step, the functional relation can be used in determining the possible outcome for the synchronization process by activation of a type 1 basic behavioral...
calculation. Thus, the type 2 basic behavior condition is downgraded to a type 1 basic behavioral condition. It is also true that during this first non-functional step the synch level of two entities have been increased. However, it also must be noted that in the second attempt that follows the nonfunctional calculation, there can be situations in which the functional synchronization step is not successful or the functional relation is already sustaining the desired condition, thus none of the generation processes are to be activated in the second step.

Table 4.4 Example Case for a Nonfunctional Relation

<table>
<thead>
<tr>
<th>Nonfunctional Relation</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
<th>Equality</th>
<th>Data Loss</th>
<th>Multi-Respond</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
<td>Complete</td>
<td>Allow</td>
<td>No</td>
</tr>
<tr>
<td>A Injects B</td>
<td>Yes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B injects A</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Separate</td>
<td></td>
<td></td>
<td>No</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matching</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Functional Relation</td>
</tr>
</tbody>
</table>

Figure 4.2 Processing of a Nonfunctional Relation Resolution

Briefly put, in cases of type 2 basic behaviors, all nonfunctional codes are resolved to Basic Type 1 conditions. Thus, basic Type 2 conditions must be successfully followed by Basic Type 1 behavior. An example for such a transformation can be seen in Figure 4.2.

**Starting Conditions:** All starting conditions for basic type 2 behavior can be found in Appendix 6
4.2 Complex Behavior:

Complex behavior is conditions where basic behavioral conditions are not available, yet synchronization attempt is possible with deeper analysis or with the aid of deep synchronization methods. There are three major synchronization methods which are designed for satisfying these deep synchronization needs. These are “State Division”, “Separate Matching” and “Reverse Unified Response”. There are five types of complex behavior that are studied. Type 1, 2, and 3 complex behaviors can be defined as the fundamental resolvable complex behavioral conditions. Type 4 is a specialized sub-type for resolving “one to many” or “many to many” dependency relation conditions. Additionally, type 5 behaviors are a set of highly complex behaviors that are designed to create synchronized states forcefully. There are three major conditions that open a need for a forced synchronization attempt.

4.2.1 Complex Relation Resolution Toolset: Deep Synchronization Methods

Deep synchronization methods are a set of complex behavior resolution tools in which code entities can be divided between their correlated sub-entities from another state or within the same entity state. Deep Synchronization is used in

- Solving multi-response clashes
● When sub-entities are inconsistent in deciding which side can dictate the change
● When generation ability is inconsistent among sub-entities

Deep Synchronization is based on dividing matching related states via creating reflections of the instances. For this reason, deep synchronization is able to use the below three methods:

● State Division
● Separate Matching
● Reversed Unified Response

Deep synchronization methods are explained in the following.

4.2.1.1 State Division

State division is a method to handle *inconstant parts of an entity state* and *consistent parts of an entity state* separately. These divided parts handled in different states and once they are processed, they are recombined to a single state entity.

Consider a case where a state entity is inconsistent at some of its relations. For example, generation ability for some of the entities is not available. Since both nonfunctional process and functional process declare that generation ability is not possible in cases where all code elements does not reply to generation query with a positive response; these cases declare their generation ability query is either partial or unavailable. Generation ability can be impaired or in conflict either by lack of generation descriptions for the respondent codes element or sub-entity, or because of a “many to one” relation in a functional relational mapping.

In the first case, state division works as a method to force the synchronization process to check all generation ability options for the given entity state pair. Thus, it would try to resolve the maximum level of synchronized level available among the entity states. In the second condition, again, the generation would raise a fail warning; however this time, a functional multi-response case is the reason for the failure. In this case, divided states from many relation sides would direct to the same element on the correspondent side. Problem is to choose the method of how to reflect the most desired synchronized case.

In plain state division method, functional multi-response cases are identified as conflict with the generation process, thus they are not included in the generation
Chapter 4  Synchronization Behavior

In order to solve this inconsistency, state division creates a new correspondent for each divided state and synchronization process is applied to individual states. As a result, the common relation would be broken, and a new relation would emerge after divided states are recombined. An example of a plain state division process can be seen in Figure 4.5.

If partial synchronization is allowed, then synchronization attempt divides the synchronic sub-entities from inconsistent ones. This is done by clustering sub-entities in individual states and then processing them in their divided states. Divided states are later recombined into a singular view of the original entity state.

Synchronization mechanisms hold entity state information in various fashions. These design decisions are made in consideration of impact of change and tracing mechanism that is used in the synchronization. During the retrieval of the synchronization information on compared states, complex cases can be refined into basic cases, or entity states can be handled in single edit attempts that have narrow impact in the code consistency. In these refined cases, state division has less room to be used.

Figure 4.4 Functional Multi-response Cases in Plain State Division
Figure 4.5 An Example for Plain State Division
4.2.1.2 Separate Matching

When a domain entity has more than one correspondent, it cannot use functional calculation tools. In order to solve such cases, states are taken away from the calculations and processed in separate process.

When a sub-entity is resolved to several entities as *one* entity to *many* corresponding entities, then synchronization methodology perceives such cases as nonfunctional processes. In nonfunctional processes, ideal end result is a partial lossless synchronized state. Therefore, non-functional process prepares the process for reverse unified response. However, during the creation of unified response, *one to many* style resolutions has to be handled according to desired result of the synchronization attempt. Corresponding entities are divided to individual states and handled in a one to one mapping.

According to desired outcome, occurrence of a separate matching results in different behavioral decisions. These decisions have to be handled by the synchronization mechanism. Below, four different desired result conditions are explained within relation to possible outcomes of synchronization attempt.

**Complete Lossless:** If a separate matching will occur during synchronization, a complete synchronization is not possible. Only possible case for this to happen is to inject all corresponding elements by a reverse non-functional process after severing link information from corresponding sub-entities except most prior correspondent sub-entity.

**Complete Lossy:** Since complete synchronization is not possible when there are multi-responses, other way to reach to a complete synchronized state is to destroy all other sub-entities other then most prior sub-entity. However, this would mean to lose all other correspondent information.

**Partial Lossless:** Links are kept and separate matching is applied to all correspondent states. Hence, there is not a loss of information from dictating or accepting sides. After this kind of implementation, direction from correspondent to the dictator keeps a non-functional relation.

**Partial Lossy:** There are special cases where inconsistency level makes synchronization attempt to use other synchronization tools on separated states. Thus, they end up inconsistent in their synchronized outcomes. This includes conditions where generation ability is different among divided states or developer’s intentions are processed in a result that causes losses in various individual state comparisons. In such cases, the last reached synchronized state is a combination of several states with different characteristics.

As explained, complete synchronization is not directly achievable if a multi-response
during a nonfunctional process occurs. Therefore, it is expected to use one of the synchronization mechanism’s code element and code entity abilities to resolve decisions over forcing the synchronization attempt closer to desired result. However, in this research, in cases where non-functional process is concerned, it is advised to accept Partial Lossless synchronization. It is accepted as the most beneficial result for nonfunctional processes.

Figure 4.6 An Example Case for a Plain Separate Matching
4.2.1.3 Reverse Unified Response & Unified Response

Reverse unified response is a method to handle multi-response cases. It is used in bringing a temporary *one to one* relation into a nonfunctional *one to many* or nonfunctional *many to one* relationship. By creating unified instance of “to be synchronized” sub-entity, unified response can force the relation of two states to become ready to complete lossless synchronization during a functional process. Thus, unified response helps to create temporary reflection states that are ready for type 1 complex behavior or a basic behavior.

Unified response cases appear when a functional relation as a *many to one* multi response case. Information for synchronization is a deduced from the common information that is carried among the responding code elements and sub-entities such as their identifier information. However, this creates the necessity to collect the shared information in the corresponded unified form in the specific case of each synchronization attempt. Thus, it results in a condition where differences among common information of relative code parts from the same entity have to be chosen by a priority selection. However, there is problem if just the unified response to be applied. This problem is about the consistency condition in which the common information from different correspondents overwrites their information in unified respondent, and as a result, they lose their equality level, rather than gaining more synchronized state with their corresponded. This problem is handled by forcing a nonfunctional resolution on opposite direction. This solution is explained in the following:

A reverse unified response case appears when a nonfunctional relation has *one to many* relation situation. A reverse unified response has two parts that is also embedded to its name.

![Diagram](image1)

**Figure 4.7 An Example for Reverse Unified Response**

A unified respondent that directs incoming or outgoing information to common identifiers of sub-entities, and a reversing process that dissolves the unified response image back to originals. Unified response is applied to original sub-entities. Therefore, unified response
reverts from the unified view to original condition. This reverse case can be solved by using state division and separate matching during resolution of the unified response back to originals.

When analyzed, every unified response is able to be resolved to a reverse unified response. Therefore, designer of the synchronization process can use three different approaches:

1. If the consistency among the unified elements is to be kept, every “many to one” case can be handed as a reverse unified response. This would force the opposite direction on the synchronization attempt. Thus, it will bring a consistent form to all related sub-entities.

2. If reflection of latest information while keeping data integrity is preferred, then plain state division is a better choice for implementation. This would mean to use state division by first dividing the entity state and applying a priority based selection to choose the unified response candidate, which is also a part of reverse unified response process. An example of deducing the unified response is depicted in Figure 4.8 and Figure 4.9. Significantly, creation of the unified response is dependent on the approach of how to deduce the unified case. Thus, this approach is open for improvement. However, in current case, priority based selection is applied in order to keep a consistent form with the latest version. Currently, a latest update is suggested as a default option for inferring priority.

Figure 4.8 Preparation of the Unified Response
Chapter 4  Synchronization Behavior

If Complete Lossless is the desired result

Allow only the most prior to be the unified response; Try to inject others with a nonfunctional process. If it is not possible to do so, then alternate the process or terminate.

If Partial Lossless is the desired result

Allow only the most prior to be the unified response; Try to inject others with a nonfunctional process. If it is not possible to do so, then keep simple state division to save the code that is not able to be synchronized.

If Complete Lossy is the desired result

Allow only the most prior to be the unified response; Don’t include others in the process.

If Partial Lossy is the desired result

Allow only the most prior to be the unified response; Don’t try to inject others with a nonfunctional process.

Figure 4.9 Inferring the Unified Response
(3) Finally, unified response can be deduced with a different selection method that is other than priority based selection. These cases are not in the interested of this research, but it must be noted that such decisions are in the hands of the developer and a possible area for further research.

4.2.2 Type 1 Complex Behavior

This condition appears when two states are dynamic states and when both of them accept changes from each side, and both sides have functional relation to each other. Starting conditions are when two states have accepting stance and both of them have the generation ability.

Table 4.5 Stating Conditions for Type 1 Complex Behavior

<table>
<thead>
<tr>
<th>-</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Condition Analysis:**

In order to understand the right combination to resolve these synchronization states, separate conditions are needed to be created. Via these conditions, complex behaviors can be downgraded to basic type behavior. If two sides resolve a functional relation to each other, the side which gets the priority must be decided.

**Step 1:** Original Conditions are same as Table 4.5.

**Step 2:** Change one of the state’s accept direction to dictate direction from original condition.

Table 4.6 Interim Step during Resolution of Type 1 Complex Behavior

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Complexity Status: Which side is prior?**

Mechanism has to decide which dynamic state is allowed to dictate to the other state.

**Suggested Solution 1:** Activate ability to catch developer’s intention

Developer decides which side is prior in synchronization, thus that side will earn the dictation.

**Suggested Solution 2:** Activate impact of change – time of analysis on code entities

Mechanism decides which side is prior in synchronization by comparing time of entity state creation and storage, thus that side will earn the dictation. Expected priority level is highest for the last edited version and lowest to oldest edited version.
**Suggested Solution 3:** Activate ability to follow edits on code element identifiers

Mechanism decides which element is prior in synchronization by comparing time of edits on sub-entities, thus that element will earn the dictation, but not whole entity state. Expected priority level is highest for the last edited version and lowest to oldest edited version. Use Deep Synchronization is used in these solutions. Entity states are divided to sub-entities as individual entity states and match to correlate with sub-entity with same **matched level of complexity.**

![Diagram](image-url)
4.2.3 Type 2 Complex Behavior

Two sides are both dynamic and they are both ready to accept changes. When their relation is resolved, one of the sides has non-functional relation. Synchronization attempt depends on resolving the nonfunctional relation to a functional relation. There are three different types of conditions that mimic the type 2 complex behavioral conditions. These are displayed in the following figures.
Table 4.7 Starting Conditions for Type 2 Complex Behavior

<table>
<thead>
<tr>
<th>Functional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Condition Analysis:**

Nonfunctional relation must either be resolved, or it must be divided to a functional to functional relation match via deep synchronization. Deep synchronization is applied to nonfunctional state to divide the state to a matched state with relatable sub-entities and an unmatched state that cannot join to that matched sub-entity cluster. In order to understand right combination to resolve these synchronization states, separate conditions are needed to be created. Via these conditions, complex behaviors can be downgraded to basic type behavior. If two sides resolves a functional relation to each other, rest of synchronization steps are similar to type 1 complex behavior.

**Step 1:** Original Conditions are same as Table 4.7. In this step, if A to B has functional relation, while B to A has nonfunctional relation, then it follows the second step.

**Step 2:** Resolve the non-functional relation to a functional relation.

Table 4.8 Calculation Conditions during Type 2 Complex Behavior

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Step 3:** Apply Type 1 Complex Behavior resolution from this point on.

Figure 4.12 Starting Conditions Example for Type 2 Version 1 Complex Behavior
Figure 4.13 Resolution of Type 2 Version 1 Complex Behavior

Figure 4.14 Starting Conditions Example for Type 2 Version 2 Complex Behavior

Figure 4.15 Resolution of Type 2 Version 2 Complex Behavior
4.2.4 Type 3 Complex Behavior

Two sides are both dynamic and they are both ready to accept changes. When their relation is resolved, both have non-functional relation to each other. Synchronization attempt depends on resolving one of the nonfunctional relations to functional relation, thus converting type 3 complex situations to type 2 complex behavior situation.

<table>
<thead>
<tr>
<th>Functional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Condition Analysis:

Both of nonfunctional relations must either be resolved, or they must be divided into a functional to functional relation match via deep synchronization. All multi-response cases must be resolved. Type 3 complex behavior is similar to applying two Type 2 complex behaviors. In a type 3 complex behaviors, similar to type 2 complex behavior, once one of the cases are resolved; others evolve to a functional state or a closer version of a functional relation.

Step 1: Original Conditions from Table 4.9 applies as original conditions.

Step 2: Resolve the non-functional relation to a functional relation

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Step 3: If not resolved by a side product of step 2, then resolve the non-functional relation for opposite side.

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure 4.18 An Example for Starting Conditions of Type 3 Complex Behavior
4.2.5 Type 4 Complex Behavior: Multi-Response Situations

In a synchronization attempt, when a sub-entity is related to more than one corresponding sub-entity, this causes a conflict on synchronization decisions. A multi-response can occur both in functional and nonfunctional states, thus it is not sufficient to apply basic behaviors to deal with the multi-response situations without providing necessary behavioral setup for the synchronization.

Multi-responses are related to tracing mechanisms of synchronization mechanism framework. Depending on how the tracing methodology is applied, changes in relations would require different types of resolutions. For example, if a tracing mechanism is using an internal trace link and it is static on whole code entity, there would be cases that it is not possible to dissolve the relation from one-sub entity unless forced. [28].

In this research and also in related studies of Hammad M., studied and tested models have shown that multi-response from one domain to another appears mostly between meta-models and their generated code. Thus it can be concluded that code realm correspondents from models correspond to meta-models similar to [15], or model to generated code similar to [23]. However, there is a difference in how to resolve multi-responses in functional relations and nonfunctional relations. Difference is usually that model realm holds the singular side, while the code realm holds the multi-response.

Since methodology works on non-functional to functional flow, nonfunctional relations prepare the conditions to resolve multi-response elements. In nonfunctional relations, multi-
response relations are composed to a singular responding unit. This is a deep synchronization method called *reverse unified response*. A unified response hides the multi-response situation temporarily. During the resolution of a functional process, all corresponding elements of a reverse unified response are linked to a subversion which is hold in a separate functional process.

![Figure 4.20 Reversing the Unified Response of a Sub-Entity or Element](image)

While unified response itself creates conditions to apply a functional process, it resolves itself with deep synchronization methods or a priority application at the back of the major functional process.

4.2.5.1 Priority on Code Elements in Functional Process during Multi-Response

In any functional process among code elements, priorities have to be set for the process to make selections work successfully. In order to infer priorities, synchronization mechanism’s abilities have to be activated. This is necessary because functional multi-response cases are “many to one” relations and they have to select which element would be send to correspondent as unified response. Therefore, there has to be a selection among sub-entities. These selections are inferred using a kind of priority over other options. Inferring priorities can be implemented in various fashion, however in this research, it is advised to use the suggested solutions that are explained below:

**Suggested Solution 1:** Activate ability to catch developer’s intention

In this case, developer feeds the priority information for the process. If synchronization attempt resolution is calculated as possible, then synchronization implementation asks to developer how to handle multi-responses. As basic support for priority selection, developer can choose to dissolve the other links except a link or links, or they can be hidden while synchronization implementation.

**Suggested Solution 2:** Activate ability to follow edits on code element identifiers

Instead of developer, same decisions are automated via an analysis of time of change on sub-entities. If the process involves code entities that have higher matched complexity level than code elements, process continues among sub-entities of those code entities.
4.2.5.2 Using State Division during resolution of a multi-response case

If a unified response is not possible, then failing sub-entities are either dissolved or hidden during functional processes. Necessity to hide or dissolve a link to a sub-element is deduced from desired result and generation abilities in combination to information from synchronization mechanism’s abilities. When relation links are decided to be separated, it will either be corresponding to another instance of its correspondent in reverse process or it will not be in the process at all.

If a sub-entity is not included to reverse unified entity state process, then it will be dissolved; and if the link is kept on a separate instance, it will be hidden from other sub-entities. As a result, its identity will be unrelated to corresponding domain. With a nonfunctional process, this unrelated sub-entity can be injected to corresponding entity state. These decisions are given separately during processing reverse unified response. Since reverse unified response will call state division on itself in order to solve the consistency problems. Thus mainly state division method is used in resolving these cases.

However, dissolution of a sub-entity breaks partial lossless nature of nonfunctional process. If dissolution is not activated, nonfunctional process will pass the sub-entity without handing it. This will let future functional processes to handle deciding information losses for multi-responses.

Figure 4.21 Preparation for a Unified Reverse Response
4.2.6 Type 5 Complex Behavior: Conflicton among Three Major Synchronization Parameters

Conflicting states appears when two compared entity states declare parameters that make synchronization inapplicable. These conditions are when:

- **Two static states are checked for a synchronization attempt:**

  This condition manifest itself when both sides enter to synchronization behavior analysis as statics, thus both sides are unable to accept changes that will occur on themselves. They
can be found in Appendix 3.

- **Direction of change is in conflict with static and dynamic counterparts:**

  This condition manifest itself when accepting and dictating sides are not able to do so because both sides are trying to dictate their state on the other side at the same attempt, or the accepting side in not able to do so because of its nature of state is static, or because it does not possess generation ability on accepting side. They can be found in Appendix 4.

- **Generation ability is heavily impaired or totally unavailable:**

  Conditions that have identified generation ability as unavailable can be summarized as all situations that have both sides with no generation ability and all conditions where direction of change is persisted to be on static side.

  When analyzed, it can be seen that, even though they are all aggregated as type 5 complex behaviors, there is an order among identification of conflicts. This is because conflict conditions overshadow each other. Expected conditions is that, firstly two static states conflict will be identified, secondly direction of change conflict will be identified, and lastly impaired generation ability will be identified as the conflict type to be resolved as type 5 complex behavior.

  In this research, two static states conflicts and direction of change conflicts are suggested to be resolved in two steps. For conditions of impaired generation ability, *manual generation feed* is suggested as a solution. However, it must be noted that developers of the generation ability mechanisms are able to add other solutions to resolve situations where generation ability is unavailable. These additional abilities are to be assessed by the developer of the framework and they are to be placed in behavioral analysis.

  These three type 5 conditions can be intertwined to each other. During resolving the two static state conflicts, if direction of change is also found to be conflicting, then also the direction of change conflict must be resolved, and if during that resolution a heavy generation ability conflict is found, then it also must be resolved. Important point is, in a holistic analysis, *two static states conflict overshadows the direction of change conflict*, and *direction of change conflict overshadows impaired generation ability conflict* at the initial phase. This is also true for resolutions that are initialized for direction of change conflicts. If a direction of change is not resolved in its first step, then it will act as *two static states conflict*. If this *two static states conflict* behavior is resolved to impaired generation ability conflict type, then generation ability conflict solution steps must be activated.

  In resolving two static state situations, first step is to open negotiations with the user via ability to catch developer’s intention; and if that does not create a solution, then mechanism can try to deduce the intentions; and if possible, mechanism can suggest an available
alteration to nature of state parameters. Moreover, if nature of state parameter change is not possible, mechanism can ask permission to use altered reflections of entity states to analyze possible outcomes as if the parameters are changed. Reflections are snapshots of entity states with changed nature of state parameter and direction of change parameter. If allowed, synchronization applies behavioral analysis on reflections of entity states. Developer can choose to implement the reflection state’s results instead of original states. This ability is limited by whether if using reflections or placing reflection results with originals are permitted and able in framework configurations.

This scenario is possible when a synchronization attempt is identified as type 5. Synchronization mechanism raises a warning that declares conflict and questions the user. At this point, user can either try to reconfigure the parameters of the synchronization attempt, or it can choose to create reflections and alter their nature and direction. If allowed, synchronization can work the behavioral analysis and outcomes from reflections of entity states. These outcomes can be implemented instead of real outcomes. However using reflection results must require developer’s permissions.

On the other hand, resolving direction of change conflicts has the similar steps with resolving two static state checks. User must be informed of the conflict and mechanism oversees if there is a possibility to resolve the direction of change conflict by just changing the direction of change parameters. If it can find such a solution then it must be followed with a change of direction and other conflict behaviors.

Decision for changing direction is either deduced by the mechanism or user can change the direction manually. New direction can be deduced and suggested to user by analysis of generation ability and nature of state. If this first step fails, then it is concluded that type 5 complex behaviors has to be resolved in two static state conditions or impaired generation ability conditions. In second step, synchronization attempt is reanalyzed and if direction of change does not result in a new type 5 conflict, this means that synchronization attempt is resolved to other basic or complex behavior types. Moreover, reflections can be used in creating an alternative solution. Reflections are accessible when two static type conflicts emerge during direction of change conflict resolutions.

A special condition for direction of change conflicts are when two sides declare that they want to dictate. Unless resolved by one side changing their direction parameter to “accept”, a direction cannot be identified. Therefore if this behavior cannot be changed in the first step of direction of change resolution, then the step will repeat itself, hence asking user to change the direction or abort the synchronization attempt.

If a generation ability analysis cannot be resolved at least to a type 4 complex behavior,
this means that mechanism couldn’t find a solution as a partial generation. In such a case, it must be noted that generation conditions behaves differently on a code entity and a code element. If a code element cannot be generated while all other parameters are sufficient for a successful synchronization, then code change equivalent has to be fed by other means by the user of the mechanism. These are manual generation feeds. However, manual generation feeds are prone to mismatches since they user feed a code element with a different behavior. In order to supply a consistency support for manual generation feeds, similarity calculations can be beneficial yet they are in need of testing. Related discussion on use of similarity calculations on synchronization frameworks can be found in section 5.5.4.

4.3 Combinations

Combinations are methods that combine two or more steps of synchronization behavior attempt analysis processes in order to analyze and resolve synchronization behavior. Most basic combination is a nonfunctional process that is followed by a functional process.

In this research, synchronization behavior has been designed to be directed to a functional relation in order to create a consistent resolution at the end of a combination. Thus, it can be stated that all synchronization attempts are constantly downgraded to a functional process.

All given types that are explained in complex basic behavior sections can be used in chains to each other in order to bring a successful and acceptable synchronization solution. Highly complex cases that exceeds the given complex behavior resolutions can be seen as a self generating solution that repeats the synchronization steps over itself and adds them on a solution tree. This three is generated during synchronization behavior analysis. After behavior analysis is resolved to a desired result and accepted for implementation, solution tree is run on real entity states, creating the synchronization implementation.

4.4 Compositions

Synchronization analysis can handle two states that are compared to each other to understand the possible outcome of a synchronization attempt. However, in cases where more than one state is compared to each other at the same synchronization attempt, this is actually a combination of several entity states that are chained together in order to resolve a functional process. Compositions are combinations that appear on the same synchronization attempt at the same processes. Importance of compositions is the process order. Composition process order must be analyzed and configured during the analysis the environment that
synchronization mechanism will target. At the step of impact analysis in synchronization methodology, compositions process order must be understood and set for the synchronization attempts.

4.5 Reordering

Reordering is a less important ability that is a part of code entity abilities. It means to synchronize the order of appearance of the code entities with the dictating side. Ordering behaves very simple. Since code entities are encapsulated, they are easy to reorder. In most cases, Sub-code entities in a code entity can reorganize their structure and they don’t need to synchronize neither their code body nor code identifiers. Reordering is not included in synchronization behavior types, thus it is not a tool on synchronization. However, it is important for frameworks that depend on the order of the declaration of the sub-entities. In such cases, reordering of the code elements and sub-entities must be supported.

During a reordering process, there is only one parameter comparison. Those parameters are the direction of change parameters. Accepting side reorganize its appearance of sub-entities with the same order of the dictating side. Sub-entities with no corresponded from dictating side are not replaced but they are either pushed up or down at the appearance order. How to handle the reordering of sub-entities with no correspondents can vary in behavior and this does not have a profound effect in studied environments. Therefore in this research, this decision of setup for reordering is advised to be left to developers.
Chapter 5 Design Methodology

At this section, how to apply the synchronization behavior analysis and designing a synchronization mechanism for a specific target domain solution is studied and explained. Goal of synchronization methodology is to provide the basics for a systematic approach that leads the developers in their pursuit to realize the synchronization behavior analysis and implement the analyzed solutions. Code object based synchronization behavioral analysis can be applied to a platform that fits into code entity – code element scheme. Therefore synchronization methodology can be applied to frameworks and systems that synchronization behavioral analysis can be applied.

5.1 Necessary Sub-Mechanisms for Synchronization

Hammad, Collard and Maletic discuss that, in their empirical research, they found the confirmed changes on the code are mostly based on the changes in behavior part of methods. By behavior, what is meant is the part of the code body, where the method functionality is described. Since the model is not to be directly affected by the changes that are not targeting the identification of the field or method declarations, it does not create a requirement to reflect the changes on the model. [13]

Most of the code-to-design change requirements appear when there is a code change which is directed to the model identifications or dependency relations among classes. These are mostly application interface related changes. This is an expected outcome for object oriented language development platforms, since the design of the classes are separated to interface and the implementation part, and design models are focused on the interface part of an object. [4]

We can relate the difference in the design-to-code impact and the code-to-design impact to the description of entities and their properties. If the properties are not changing in the declaration level, then there is a significant chance that changes which might occur will not need to be directed to the model. However, this does not mean that they are not indeed required to be reflected at all [13]. Implementation declaration of a method body can hold dependencies to other classes or libraries [19]. These necessary abilities are discussed in section 5.3.

As mentioned before, in this context, properties are used to describe the attributes and methods of the class. Class descriptions are written in the files that carries the code in the
textual basis. By using this to our advantage, one solution to overcome the issue of bringing generated code and design model in comparable form resides in a persistent unified form format for the model and code bodies. [23]. This persistence and the format of the unification is a discussion related to serialization. This topic is discussed in the section 5.5.3 in relation to sections 5.5.1 and 5.5.2.

A code file can be analyzed and semantics of the code can be extracted, however these services have to be linked, and the resources in the code body have to be traceable to an extension that will lead the mechanism to entrap the related elements to each other. A tracer mechanism is a requirement for such ability. However, it must be noted that tracer techniques vary greatly among each other, and suitability of tracer techniques over one other is a matter that has to be decided according to the target domain. This topic is discussed in the section 5.5.2.

Additionally, in domain specific language developments frameworks, beyond the required mechanism to trace the links from model-to-code or code-to the model, there are specific problems in reflecting the changes in the source code to the code generation facilities in a domain specific language framework. This is also true for changes in the code generation descriptions.

Once we are able to trace the model entities in generated source code, and once we are able to bring the model and the source code in a comparable form, we would still need a decision mechanism that can conclude the necessary action for the related committed change. This mechanism can be described as the brain which carries the information of the synchronization behavior. It has to be capable of analyzing the data that is collected by the tracer mechanism. Additionally, it would strongly need to support the synchronization of the code generation facility in the domain specific framework, so that the mechanism does not get out of synch with the code within the code generation workflow. This topic is discussed in the section 5.5.4.

5.2 Methodology for Constructing Synchronization Mechanisms

In the following, design methodology will be explained step by step. Moreover, the detailed explanations, discussions and findings on the structure of synchronization mechanism models are provided. These sub-mechanisms are all linked to the discussions and findings within respective sections.
5.2.1 Aimed behavior: Behavioral goals

Aimed behavior or behavioral goal is a documentation that gives the initial task description for the synchronization mechanism design. Complexity of the desired behavior of a synchronization mechanism would change according to minimum requirements of aimed behavior. Aimed behavior does not include the description of the inner mechanics of a synchronization attempt. Rather, it is the desired outcomes from a synchronization mechanism.

5.2.2 Overall Analysis of Problem Domain

In overall analysis, the conditions of the related technologies and frameworks are studied and an overall picture presented in order to simplify the solution implementation. There are two steps of an overall analysis. Overall analysis includes research on background technologies, framework, programming languages and other related fields. After this step, an overview of the interested system is deduced as a simplified form of the all related elements. This overview will help to focus on the correct parts of the interested domain or domains, so that the mechanism can be cleared from the other unrelated parts.

5.2.3 Segmentation of Target Domains

Segmentation is a process where the outcomes of the overall analysis are restudied in order to bring the parts of the mechanism that can be combined in an abstraction. These abstractions are constructed by following conventions:

- In the work sequence of a process, steps that lead to a major result such as a code product, a change on a code body, or a setting are in interest to each other are considered for aggregation.
- If these interested parts are dependent to a code entity that is acting as the accessing point for the user of the environment, then these points can be used to create simplified versions of the overall system.
- These access points must also cover a dependency relation to other access points that are selected which same reasoning.
- These simplified versions are then constructed to become abstractions, and they are used in representing all related parts of the system.

As a result of segmentation process, problem domain is divided to target domains which have a distinctive uniformity and they are able to create dependency relations which are able to impact other target domain or domains of the change on themselves.
5.2.4 Code Complexity Analysis

After segmentation of the problem domain, new target domains are analyzed to deduce the code bodies that fit to code entity and code element specifications. This step is a very important for mapping the synchronization behavior, since those entities and elements will be declared to have the matched level of complexity among other target domains and on their own entity states.

Second importance of these classifications is that the storage of the entity state versions is expected to follow the entity conventions that are declared at this step.

Code entities and code elements are defined for the target domains. There are two major factors to identify this behavioral separation,

- Code entities are identified as the code bodies that fit to descriptions of section 3.1.1.
- Moreover, changes of code elements must resemble the behavior of types of changes in code elements. On the other hand, code entities must resemble the type of changes in the code entities.

5.2.5 Impact of Change Analysis

Once code entities and code elements of the target domains are identified, target domains are ready to be analyzed for a code element and code entity type of changes. Thus, target domains that are interesting for the solution to the aimed behavior are picked, their limitations, expected state conditions and exceptional state conditions are studied. Conditions that does not fit to either one of three are also aggregated as “other”. Finally, all conditions that are studied are applied to find the relation of impact of changes and the path that these changes must follow for transferring these changes.

5.2.5.1 Limitations Conditions

Limitation conditions are:

- Unavailable conditions that is restricted by the nature of the target domain
- Restrictions that dependency from one domain to the other has by design
- Behavioral limitations that directs the impact of change on limited outcomes

Limitations are very helpful to decrease the workload of the developer of the synchronization mechanism. Moreover, limitation conditions decrease the number of possibilities of the changes, thus they open the way for creating unambiguous behavioral outcomes and generation ability.
5.2.5.2 Expected Conditions

In expected conditions, conditions that can occur while trying to achieve the aimed behavioral outcome in given limited conditions are studied. Expected conditions are a guiding document for design of the synchronization mechanism. Thus, they are a guide to follow when realizing the aimed behavior on the interested problem domain.

5.2.5.3 Exception Conditions

Exception conditions are events that can occur outside of the expected conditions, however they are known and they can be identified. In exception conditions, the behavior does not fit into expected inputs or outputs. However, since they are identifiable, users of the mechanism can be guided elegantly, or the mechanism can be set to handle the exception conditions separately, according to their nature.

5.2.5.4 Other Conditions (Optional)

This is for documenting that how the mechanism will react under circumstances where unidentified conditions occur. This is an optional declaration document.

5.2.5.5 Target and Path Selection

Target and Path selections are the conclusions deduced from limitation, expected, exception and other conditions. During target selection, domains that is a part of solutions to achieve the aimed behavior are selected as the targets, and the path that reflect the impacts of change and how this impact is realized on different types of changes on code elements and code entities are analyzed. Therefore, a discussion on a change from a target domain that is directed to other target domain is documented.

During target and path selection, all expected impact of change conditions are considered. Hence, it is where the specifications that define how the aimed behavior can be realized on the problem domain are documented for every type of change.

5.2.6 Mechanism Model

Mechanism model is the solution that is brought up by the developer. While target and path selection provides a general documentation of type of change conditions, mechanism model has to choose the level of complexity support. In this research, five variants of selected complexity support as the target behavioral aspect are proposed. The need for target behavioral aspects are either deduced from the aimed behavior or it is dictated by the limitation conditions.
Other than target behavioral aspect, mechanism is also designed to an optimal condition support for a default result. Default design support would change the generation abilities that are necessary for determining minimum level of synchronization and allowed loss of information. These are the same desired conditions of synchronization behavior’s expected outcomes.

**Complete Lossless Default:** Design is expected to support generation of all possible outcomes.

**Partial Lossless Default:** When design cannot support a generation, it is expected to discard code entities and elements that cannot be generated, but keep the discarded code on synchronized outcomes.

**Complete Lossy Default:** When design cannot support a generation, it is expected to discard code entities and elements that cannot be generated and destroy these entities and elements.

**Partial Lossy Default:** When design cannot support a generation, it is expected to discard code entities and elements that cannot be generated, but keep some of discarded code on synchronized outcomes while destroying others.

From the above information, it can be stated that partial lossless default support aims for a minimal requirement for a synchronization condition with minimum requirement of user’s intervention to the mechanism. This is because partial lossless synchronization behavior will aim to keep the information loss nonexistent. On the other hand, as mentioned in synchronization behavior, complete lossless synchronization is the most ideal condition to achieve. This signals that most desired condition and most achievable condition are not the same. However, complete lossless synchronization default support is most plausible to sustain in situations where there are only two target domains that is found by the segmentation of the problem domain.

5.2.6.1 Basic Support

In these cases, mechanism is designed to reflect the synchronization behaviors that fit into basic type of behavioral cases without a deep synchronization method. Starting basic conditions are provided in Appendix 5 and 6. Common aspect of basic complexity support is the holistic availability of generation ability. This means, even if one of the elements cannot be generated on the other side, holistic outcome will signal the outcome is not possible, and it will not run the synchronization unless generation ability is provided.
Basic support is the simplest form for synchronization support. Since it does only allow overall consistent conditions, it has highest level of keeping consistency among entity state versions among the same target domain. However, it is also designed to not start synchronization attempts that can actually be applied with deep synchronization tools.

5.2.6.2 Selective Support

Figure 5.1 Creation of Selective Support over Basic Support

Selective complexity support means to use state division method for a partial synchronization support. This approach is superior to basic support in terms of creating a partial synchronization relation among entities and elements that fit into synchronization attempt. Selective complexity support is expected to resolve all basic behavioral conditions and up to complex type 2 version 1 behavioral conditions that does not have a requirement for separate matching and a reverse unified response.

5.2.6.3 Multi-Response Support

Multi-response complexity support means that mechanism can resolve multi-response cases. In this research, there are two synchronization methods for specially dealing with multi-response cases. These are separate matching and reverse unified multi-response.

Figure 5.2 Elements that are required for Multi-Response Support

Therefore, if a mechanism can support resolution of multi-response cases, it is assumed that mechanism has support for separate matching and reversing unified response abilities. Thus, multi-response complexity support has to have all deep synchronization tools in the mechanism. It must be noted that, this approach is necessary only if problem domain creates multi-response conditions in order to achieve the aimed behavior conditions.
5.2.6.4 Generation Feed Support

Generation feed support is an elegant and guided way to replace unavailable generation situations with manual generation feeds from the developer. It must be noted that generation feed support is a type 5 complex behavior and it can be applied independent from basic, selective and multi-response support conditions.

5.2.6.5 Complete Support

These mechanisms are expected to support all behavioral conditions that are defined in synchronization behavior. Complete support is the most complex level of support that is identified in this research.

5.3 Major Sub-Mechanisms for a Synchronization Mechanism

In this research, one of the conclusions is that, once impact analysis is complete, a synchronization mechanism can be placed by using the four major sub-mechanisms. This synchronization mechanism must target at least two target domains as a flow. If a synchronization mechanism targets the same target domain, then synchronization behavior analysis applies to the solution; however, it behaves as a versioning system. In this research, it was concluded that, in order to realize a basic synchronization mechanism, four major sub-mechanisms are required. The work for De Lucia, Fasano and Oliverto was the starting point for the identification of the necessary sub-mechanisms [29]. These necessary sub-mechanisms are defined as:

- Change Notifier Mechanism
- Entity Relation Tracing Mechanism
- Code Generation Mechanism
- Behavior Decision Mechanism

It must be stated that, influence for defining these four sub-mechanisms are deduced from the discussion and findings on change in code artifacts and impact of change. It was stated that there are four major specifications of a change analysis. These are:

- Compared states of Entities: How are the entity states are captured and what is the
relation of the compared entities?

- Time of Analysis: What was the timing of capturing the entity state condition?
- Correct decision: What are the possible outcomes of the relations?
- Implementation of Decision: How to realize the analyzed decision?

For more detailed discussion on impact of change analysis, please refer to section 5.5.2.

Traceability is used in solving the issue of compared state of entities. Time of analysis issue was combined with triggering and identifying occurring changes, thus an interim sub-mechanism was suggested with under the name of “change notifier mechanism”. In order to solve the correct decision issue, synchronization behavioral analysis and decision making processed was introduced. Lastly, implementation of the decision was suggested to be solved in a code generation mechanism which is controlled by the decision mechanism.

For all studies that were influential on design decisions for this research, please refer to section 5.5. In the following sections, these four mechanisms is discussed and explained.

5.3.1 Change Notifier Mechanism

Change notifier mechanism is a sub-mechanism to deduce the impact of change on target domains and it is interrelated with the tracing mechanism. Therefore, change notifier is an interim mechanism between tracing mechanisms and behavioral decision mechanism. It uses change analysis in order to deduce the changes on interested targets and pass its results to the behavioral decision mechanism.

According to knowledge of the researcher, serialization techniques are used in storing a version of a code entity. A discussion serialization can be found in section 5.5.3. However, it must be noted that time of analysis and implemented relation tracing mechanism has an effect on necessary features of captured entity states. This is because traceability mechanisms can provide the necessary level of information about a state. This can be seen in the work of Benestad, Anda and Arisholm [2]. Moreover, making the correct decision and implementation of a decision is passed to behavioral decision mechanism.

5.3.2 Entity Relation Tracing Mechanism

Tracing mechanism is heavily related to capturing and linking the compared entity states. Tracing mechanism is necessary for realizing a synchronization attempt because of the need to relate the correspondent code body parts during a functionality based synchronization calculation. Without a way to relate these code parts, it is not possible to define the code elements separately, and it is not possible to calculate the realistic loss and possible synchronization outcome characteristics. However, it must be noted that traceability can be
achieved in a lot of different ways. Some of these traceability approaches was found interesting for classification and implementation of tracing code entity and code element relations. These are discussed in section 5.5.1, and also findings from overall traceability approaches that are found related to necessities of tracing mechanisms are provided in section 5.5.1.1 and 5.5.1.2.

Moreover on traceability, from the findings of traceability mechanisms that are studied in the field, it can be stated that a selected type of serialization method is possible to use in order to enhance the abilities of the tracing mechanism. These practices includes storing the information of links in a serialized code body, or using uniformed serialized model code and generated code for comparison. Such topics are discussed in section 5.5.3.

5.3.3 Code Generation Mechanism

Code generation means that a sub-mechanism that is specialized on creation of new code bodies to realize the synchronization attempt are available for synchronization mechanism. It must be noted that the design and needs for code generation abilities would differ according to target and path that was in interest of realization. Therefore, code generation mechanisms must be developed according to changing needs. Target domains have different necessities for generation of the code body and code elements. *Minimum necessary generation capabilities can be deduced by the target and path selection analysis.*

On the other hand, in this research, possible uses of similarity calculations on understanding and enhancing code generation and code matching abilities was found plausible to be used code generation mechanisms. However, a study that uses the equivalent approach was not found by this research. Such possible uses, together with relevant similarity and equality calculation studies are provided in section 5.5.4.

5.3.4 Behavior Decision Mechanism

Behavior decision mechanism can be called as the brain of the mechanism. By gathering information from other three major mechanisms, behavior mechanism applies *the synchronization behavior analysis* and conducts the mechanism’s communication channel with the user.

It is significant to note that, development of synchronization behavior types and tools was heavily influenced by *impact of change* or in other naming *change in the code artifacts* related studies. All influential findings and a discussion on general aspects of change in the code artifacts have been provided in section 5.5.2.
5.4 Abstract Schemes for Synchronization Mechanism Design

The design for synchronization mechanisms would require different workflows for achieving the target behavior. Beyond the synchronization behavior, the design of the mechanism is determining in providing automation or manual initialization of the synchronization mechanism. Internal and external tracing information storage, automation and manual feed operation support is capable of forcing changes in synchronization mechanism design. Therefore, in this section, fundamental sub-mechanisms will be depicted in abstractions. It must be noted that detailed and specialized design features are required for target environments.

Overview of the basic synchronization mechanism that can support synchronization behavior is similar to design that is shown in Figure 5.4.

![Image of a flowchart showing the design of a synchronization mechanism](image.png)

Figure 5.4 Overview of Fundamental Synchronization Mechanism
In fundamental design, the work process starts with an initial request to the change notifier. Automation of the synchronization process is achievable though using this initialization message. For example, in conditions where the initial request message is tied to a save/commit action in the development framework, then the synchronization attempts would be automatically called at the end of all successfully committed save attempts. It is significant to note that, the model in Figure 5.4 does not depict the initial information feeds like desired result for the synchronization.

In further design solutions, each sub-mechanism’s basic workflow will be displayed.

5.4.1 Tracer Mechanism Design

As can be seen in Figure 5.5 and Figure 5.6, tracer mechanism is able to request information from the existing code elements by directly accessing them. Tracer mechanism is the gateway of the synchronization mechanism in order to gather the required relation information on the code entities. However, it must be noted that tracer mechanism does not achieve an overall analysis for the changes on code bodies, but it is only responsible on analyzing the links that is available on the compared code bodies. It is speculated by the researcher of this work that tracing information deduction feature can be altered to increase the deduction of the code tracing ability by creating relations with an “on the run” analysis. However, analyzed studies did not have such abilities accept creating a maintenance process that is able to replace and repair previously defined code relations [11].

![Diagram of Tracer Mechanism Design](image)

**Figure 5.5 External Traceability Information Storage Design**

This tracer design abstraction is for an external link storage capacity as suggested from the discussion on traceability studies that is discussed in section 5.5.1. On the other hand,
internal storage process is also achievable by generation of the link processed during code
generation. Furthermore, previously unrelated code elements and entities can be configured
via a link generation process. This is also provided in the internal traceability information
storage design.

5.4.2 Change Notifier Design

Figure 5.6 Internal Traceability Information Storage Design

Figure 5.7 Basic Design of Change Notifier Mechanism
Change notifier is responsible for receiving synchronization request and preparing notification of change information for decision mechanism. During its preparation, it collects the traceability information and creates the relations into a workable dataset with related code bodies. Fundamental design elements have been depicted in Figure 5.7.

5.4.3 Behavioral Decision Mechanism Design

Behavioral decision mechanism uses the predefined behavioral setup process as its decision description. This predefined process is detailed in impact of change analysis step of the design methodology. By using the created information at the analysis, the synchronization mechanism’s behavioral aspect is configured in behavior decision mechanism. Fundamental design elements have been depicted in Figure 5.8.

Figure 5.8 Basic Design of Synchronization Decision Mechanism
5.4.4 Generation Mechanism Design

Generation mechanism receives the implementation path information from the decision mechanism and implements the sorted generation process. However, it must be noted that generation mechanism knows that it can generate the requested generation attempts. This is because of the behavioral analysis process that was used in ensuring the generation ability of the synchronization attempt. In fundamental design, this step is depicted in a pseudo code generation attempt which generates the generation ability parameters for the behavioral design process. Fundamental design elements have been depicted in Figure 5.9.

Figure 5.9 Basic Generation Ability Mechanism Design

5.5 Discussions and Findings on Necessary Sub-Mechanisms

This section consists of explanation of findings and conclusions from studies and reports that were found as related to identified necessary sub-mechanisms.

5.5.1 Discussion on Traceability

There are hardships of creating and maintaining traceability. It is mostly because the need for knowledge gathering and sharing in a setting where dependent bodies and
mechanisms working together leads to a chaos on the mechanic demands [30]. Depending on the complexity of the technique, one can enhance the abilities of the traceability mechanisms.

One repeating issue in traceability solutions is that, traceability mechanisms end up being forced to solve inconsistency that occurs on tracing link’s code. This occurs because of differences in compared structures, and software evolution phase of tracing code itself. Such findings, and also solution suggestions for these issues that are aimed for both collection and maintenance of traceability information, have been reported in [29].

Moreover, a tracing mechanism can be realized with internal information storage design or external information storage design. In the research of [31], it is suggested that there are advantages to use external traceability mechanisms over internal tracing mechanisms. In their study, it was stated that internal storage of traceability links means to store the tracing links on target domains. This approach causes:

- If the link is directed and stored in the source model only, it is not visible from the target model.
- If traceability information is stored in the both models, then this information must be maintained for consistency.
- Adding internal links to code may cause too much inclusion of secondary importance code parts, thus creating readability problems.
- Additional internal traceability links can cause breaking the natural form of code domain.

On the other hand, external traceability systems are claimed to have advantages as:

- Code pollution, which makes readability harder because internal traceability links, is avoided.
- It is easier to extract data and maintain the links on automated mechanisms.
- In order to create an external traceability mechanism, it is required to have referenced models with unique identifiers.

There is a particular guiding source that was found during this research. In his study in surveying various traceability techniques [32], Schwarz addressed the six different problem domains that traceability mechanism aimed to bring solutions. These are:

- Definition: defining the target domain to establish traceability
- Identification: discovering relations of artifacts in target domain
- Recording: how to keep track of the trace relations
- Retrieval: how to access to related artifacts using the chosen record technique
- Utilization: additional editing and management utilities for tracing mechanisms.
- Maintenance of traceability relationships: how to re-establish lost trace records.
By these classifications, we can structure the necessary mechanical requirements for synchronization attempt.

5.5.1.1 Studies aimed to create and maintain traceability via analysis in context of code:

In the work of [12], researchers had worked on synchronizing the use case models (UCMs) with implementation models, thus creating a model-to-model traceability mechanism. Researchers aimed to keep abstract system model of UCMs and implementation model which is defined by unified modeling language (UML) in synch. Since implementation model is matched by the Java Source Code, the mechanism is also able to keep the abstract model updated through java source code. Moreover, implementation model can be inferred from the Java Code and then matched to the UCMs.

Another approach comes from the work of [9]. In this work, objects in object oriented design paradigm have been defined as evolving. In order to catch the difference that occurs as the development continues, a similarity catcher has been proposed. This mechanism cleans the noise on the code artifacts and processes the source code to derive a conclusion about the similarity of the code artifacts. A matcher mechanism with maximum similarity approach is used in iterations to catch the matching pairs.

An alternative perspective to traceability creation can be found in [30]. In this work, code context is revealed via questioning the developers and inferring the desired relations. This approach is unique in its narrative that traceability creation is left to developer’s answers in creating a context relation graph. Developers are perceived as the main attention for the framework. However, this study is aimed to trace use-cases to code artifacts. Therefore, this approach can be seen as a specific domain solution.

A study that appeared on the scene as one of the early approaches to traceability problem came from [33]. In his approach, Egzyed tests the user scenarios versus implementation for the project. Via a comparison mechanism that was suggested to fit the requirements, it was aimed to create links from model to source code.

5.5.1.2 Studies aimed to create Creating traceability links via lightweight links

In the work of [24], researchers have proposed a plug-in for Eclipse Platform. In the Plug-in, they aimed to present similarity level between identified source code identifiers. In this case, instead of traceability links, mechanism tries to match the similarity from the key identifiers on the code.

In the work of [21], researchers have developed a DSL language in order to trace the
related artifacts during the evolution of the software, such as documentation and corresponding source code. In order to achieve this, they used a mapping technique which they call “feature maps”. These maps are composed via tiny code statements in annotation. This links are used in a revision control system. This revision control system is similar to a version control system; it uses the comparison mechanism that is also used in UNIX file systems.

In [20], use of XML to create the traceability framework has been analyzed and developed. In brief, textual files have been transformed to XML language representations. Though these representations, any model can be traced to the each other. XML language representations act as a common and comparable form. This technique resembles XML schema use in Eclipse Modeling Framework (EMF) [15]. In EMF, object specifications are converted to XMI, and later linked to corresponding source code. Xtext also uses XMI in order to link the rules and constraints that cannot be defined just by using the EMF models.

Another XML based approach to traceability was [1], which uses XML based lightweight links to create a feature mapping which is similar to approach in [21]. They also emphasize the need for internal structural requirements for traceability via links. According to this study, the most suitable condition for a traceability link to work is a fine-grained design with syntactic differencing. This is an expected outcome, since a major problem for lightweight links are losing their mapped relations during software evolution [10]. Controlling the consistency of such lightweight link approaches causes additional maintenance problems during the evolution of the software projects. These problems are addressed in various researches, such as [32], [9], [20].

5.5.2 Discussion on Changes in Code Artifacts

Tracing mechanisms alone cannot be sufficient to ensure the realization of the synchronization ability. This is because of the need to understand the impact of change that will occur on the artifacts. This impact greatly varies among the artifacts. Many researchers focus on the impact of change in various forms. Thus, for synchronization mechanisms, another step for realization of the synchronization is to understand the impact of change and making a decision for required change on other related parties which is affected by the change.

From the perception of this research, impact of change has been constructed in two major concepts. These are the comparison of artifacts, and the initialization of analysis.

All kind of change analysis types need a comparative relation to some other form of information that is related to either some form of itself or another information block that it can relate. This step, which is simply referred as comparison of artifacts, is similar to
implementations in version control mechanisms or repository control mechanisms, such as
Control Versioning System (CVS), Apache Maven Project, or GitHub Service. However, in a
mechanism such as EMF Compare, one can see implementation for general comparison
systems that can relate to any form of supported models; in this case, ECore models.
However, the impact of change analysis capabilities of such repository analysis alone is not
sufficient for artifact comparison for synchronization among code artifacts from different
roles and structures.

As mentioned before, second aspect for impact of change analysis is the initialization
of the analysis. This is interested with the time that the comparable information forms for code
artifacts are compared to each other. This is important because it directly refers to what
versions the mechanism has been analyzed. Incorrect selection for time of analysis will lead to
false or unusable results.

From the studies of De Lucia, Fasano and Oliveto [29], one can conclude that, for the
impact of change analysis, it is important to consider:

- Compared states
- Time of Analysis
- Deriving correct decision
- Implementation of Decision

Moreover, it was stated that the analysis of impact of change and traceability are
interrelated to each other [29].

5.5.2.1 Findings from Changes in Code Artifacts Studies

For this research, one of the most influential studies was [26]. In this study, Omori and
Maruyama presented their findings on a plug-in for Eclipse IDE. This plug-in is called
“OperationRecorder”. Omori and Maruyama’s aim was to develop a mechanism that stores
the individual changes that occur while software development. They present three key points
in their research:

1) Instead of comparing two code bodies to each other, such as versions in repositories,
they create ASTs from successfully compiled code files. These ASTs are then related to an
edit history of an individual item. Thus, they create the possibility to reverse individual items
to a form where they can be successfully compiled again.

2) They try to implement a mathematical mechanism to decide whether if the AST items
are related to each other. They use a similarly catcher mechanism which is based on the
offsets of the entities by scoring them from an initial start where editing began until the
current AST item. This creates the possibility to find relations in complex modifications in the
code body. However they have concerns about the success rates in high complexities. This seems especially true for text that was both modified and replaced.

(3) They warn the users for possible performance hindering on IDE’s computation speed.

Thus, study of Omori & Maruyama displays relations between the code elements in an individual code entity. There are also studies that implemented additional abilities on findings of [26] in [34]. This enhancement is achieved by adding an annotation mechanism that can be used by developers to insert editing information for clustering the editing records in correct order, thus creating a finer result in catching changes. However, in case where editing record is not available, there is also a methodology that can extract the editing information from two versions of related ASTs. This method has been constructed by [10]. This approach requires a hierarchic order, and thus fits to the nature of ASTs. A study that targets to use this method exists in [35].

(1) On the other hand, in [11] they have focused on classes, and they have developed a mechanism which uses vector-space comparison computations. In this kind of comparison, two classes are syntactically analyzed to create vector values according to identified text that is repeated in the class. In the research of [11], we can see two main results:

(2) Vector-space computation does not have the capability to map code elements (properties and operations), but rather creates heuristic comparison results via matching code entities (classes). Code elements can be added, removed, changed but they represent a smaller scale of impact to a class consistency, as described in [12]. For a code entity, impact of changes on the class identification itself has larger impact.

Possible evolutionary path of a code entity has been presented. There are seven possible change behaviors for a class. These are defined as: Replacement, Extraction, Merge, Split, Merge into a new class, Recombine, and Recombine into new classes.

Additional to comparison methodology at the level of code elements, and comparison methodology among code entities; there is a need for a methodology for comparison among entities of different modeling forms for complete coverage of synchronization attempts. An important influence to solve these cases is studied in [36]. In their work, they propose a technique to create a XML representation for comparing models and source code via a difference catcher method that is commonly used in versioning systems. Briefly, like Omori & Maruyama, they create an AST and convert it to XML versions which later used in various model to model or model to text comparisons.
5.5.3 Discussion on Serialization and Representation

Serialization is a wide area of research and it is important for this research for two reasons:

(1) It can be used as a way to compare artifacts from different platforms.
(2) It can be used in order to create a recording scheme in tracing and/or analysis mechanisms.

5.5.3.1 Findings from Serialization Studies

In this research, especially XML has been a focus on serialization attempts. There are two influential studies for this research. One of them is SrcML language [37]. SrcML is a XML based source code representation language. By itself, SrcML does not cover a possible mechanism to create serialization. However, it demonstrates that via XML, such techniques are possible. As an influential point to consider is that, there are studies that extend the capabilities of already supported languages by SrcML. They use this extension to support other styles of programming paradigms, such as work of Frank Raiser, where SrcML is used in creating intentional programming based algorisms and he implemented them in SrcML [38].

One important finding that is considered thought this research was whether if XML is the optimal suggestible choice for serialization attempts. Several serialization techniques have been analyzed for different platforms. Kazuaki has analyzed the performances of XML, JSON and Binary Formats for the same resources. He as used various serialization tools for his data, and he concludes that binary formats which uses in Java built in serialization is performing superior to other schema formats in serialization and de-serialization speeds. [39]

However, in this research it is speculated that, even though it is possible to use binary format serialization mechanisms, it brings additional strains to interested development frameworks and the adaptability of the synchronization mechanisms, thus XML form is seen as the ideal solution for code serialization attempts in traceability and state comparison analysis mechanisms.

5.5.4 Discussion on Similarity and Equality

A synchronization attempt can use the similarity level to check similarity between available synchronization result with the current code in order to create “best possible among the worst cases” to get the closest to desired result [40]. Such ability will increase the ability to create equivalents in conditions which the desired result is not an all-sum win or lose situation. However, similarity is not a guaranteed equal code body, [41], it is a risk-included
uncertain attempt to catch whether if code elements are entities are related or not [42].

As an additional note, one use of similarity calculation can be implemented in comparing the generated code to the corresponded code. This would enable an additional layer of consistency check to generated code. This can be especially useful during placing a manually fed code as generated code to replace corresponding domain entities. Developers are highly in risk of making code errors [12] and there is a chance that fed code will be in mismatch with the dictating sub-entity.

5.5.4.1 Findings from Similarity and Equality Studies

Similarity is studied in various forms as the need for a similarity concept was required. In this research, similarity has been used as a term that can define different rates of equality. Level of equality was tried to be converted into numerical values or a pattern. In this research, analyzed methods which related to issues of similarity between objects included methods as: building graphs in either or of appearance or in of the order of semantics, and comparing those graphs between code bodies in matched complexity levels. For this research, most interesting methods were used in because high visibility of both context and the mean based measurements that can be used in various degrees. This was introduced in [43]. For the alternative approaches, one can analyze the methodology in [44] where the focus was creation of “distances” between captured values that was inferred from attributed edits on labels to code attributes.

Additionally, attempts to count the identifier names that belong to code bodies in matched complexity levels to use them in mean of keyword counts in various different forms for comparison that used heuristic results, such as adding the compared code entities at closer distances on a graph tree are found. In [41] the graph element has been connected to graph areas that depict the context, while [45] used the mean value of the sequential computations that was deducted via segmented graph areas. In [46], they were focused on the similar computation with differences in selection in partitioning and value creation from the domain.

A more general overview on similarity can be found in [40]. It is observed that in all studied cases:

- Similarity methods does not benefit from a total guarantee of similarity calculation.
- Complex Entities are having worse results in average, and complexity is not a static definition among studied methodologies.
- Similarity is suitable for behavior analysis for code elements, however this does not reflect to level of equality. As in [41] and [43], code bodies are segmented and even the complex context classifications are mapped to the calculation mechanism and
similarity is highly converted to textual element in the databases, however it still does not give the answer to a change that would occur in the context itself, which would have a conflicting effect.
Chapter 6 Applying Design Methodology to Xtext Framework

In the following, an example situation will be studied. Design methodology will be applied in steps to solve the analysis and overall design needs for the given task in an example aimed synchronization behavioral condition. As the end product, it is expected to reach a model abstract that will be used in creation a synchronization mechanism that is able to use the synchronization behavioral calculations.

6.1 Aimed Behavior:

In this research, given task is to analyze the possibility of synchronization between Domain Specific Language Code and generated code as General Purpose Language Code in Xtext framework project Remote HMI monitor application. Behavior specifications are as below:

Code developers are aimed to use the modeled domain specific language code (DSL code). DSL code should not lose consistency with domain model. Changes in the domain specific language code must be synched to generated code. Changes in the generated code will be synchronized to DSL code.

A synchronization attempt is not desired in every situation. Thus, it is expected to have both manual and automated initialization mechanisms for synchronization attempts.

**Task:** Analyze the project; provide design solutions for aimed synchronization behavior for basic synchronization behaviors.

6.2 Overall Analysis

In this research, background research for overall analysis has been studied from [15], [3], and [5] It involves information from background of modeling, DSL development, Eclipse Modeling Framework and Xtext Framework infrastructure. As a result, the conclusion of the overall analysis is depicted in the Figure 6.1.
6.3 Segmentation of Target Domains

In this part of the research, overall view of mechanism of Xtext Framework will be segmented to target domains. Target domains are the critical sections for the target environment. These critical sections are segmented to focus the interest of the synchronization attempt to specific files that are keys for setting up and using the target environment.

6.3.1 Analysis of the Framework for Synchronization Attempts

In this section, background information that was given in previous sections will be analyzed in order to understand the necessities of synchronization of the code artifacts in the framework mechanism.

In Figure 6.1, mechanism of the domain specific language development and code generation facilities of Xtext framework has been displayed. In the complexity of the Xtext framework mechanism, there are several different levels of code artifact synchronization. In order to understand these separate levels, we can create a more structured overview of the framework. By studying the background system that we had, we can analyze the framework and find the stages that possible synchronization requirements appear. After this analysis, we
can come to a linear flow in the mechanism.

Xtext framework currently works as a one way flow. All development work is built in steps that are linked in the development phases. When a change occurs in one of the phases, engine needs to run again. Hence, all artifacts will be regenerated again from the start as a fresh product. It is as point-to-point flow mechanism. The reason for this structure is simplicity. Any change in the early parts of the flow would affect the other linked mechanism parts according to deepness of their dependency.

Code generation framework can be displayed in a linear flow as in Figure 6.2. Developed DSL is directed to the Xtend template expressions, and secondly these expressions are directed to the directories that are declared as generated code folders.

We can separate the mechanism to four different sub-mechanisms that appear in the generation flow. Changes inside or among these sub-systems would have a different impact in the consistency of the developed and generated artifacts.

6.3.2 Segmentation of the Mechanism Workflow

Even though mechanism of the code generation seems linear, it actually has interrelations which have different impacts on different parts of the workflow. An additional issue forms from this complexity because dependencies are not linearly formed. That means the changes in code artifacts may create inconsistency on not just the previous step of the flow but overlapping to other sub-mechanisms on the flow. [9]
However, how deep these relations are and how deep the code relations would affect the consistency of Xtext framework code artifacts is a matter of discussion. As mentioned before, there is empirical evidence that changes in the generated language do not impact the design in most cases. [23] [13]

When we analyze the framework system step by step, we can see that development phases are linked with framework actions which create the steps of development. Every step has its own type of generation phase. We can understand the dependencies of consistency on these actions. In Figure 6.3, three different action pools are depicted, these are:

- Generation of Domain Model from Xtext
- Creation of the DSLC (real model) according to Domain Model (grammar)
- Generation of the GPLC throughout Xtend Templates

One can correlate the actions with the related flow mechanism members and create more

Figure 6.3 Expected Actions in Linear Workflow
specific definition of synchronization among code generation steps in the development workflow. Figure 6.4 displays the overview of such an analysis.

![Diagram](image.png)

**Figure 6.4 Clustering Correlated Workflow Members**

When the workflow is separated to four clusters in logical design, we can come to some target code artifacts. We can use these targets to ease the problem by targeting the mechanism to members of the domain. These mechanisms create the most significant impact on the structure and behavior of the flow step.

As we look at the Xbase-Xtext-ECore Model cluster, it can be said that most significant artifact in the flow is Xtext description. Xtext carries the Xbase expressions in itself and these expressions can be accessed from the same code body. In DSL model coding environment, Xbase expressions are linked to the domain language environment via scoping ability of the framework [47]. However, this is achieved via scoping and linking abilities of other framework organs such as Google Guice. Google Guice is the technology that enables Xtext Framework to connect all these components behind the integration of different modules. [47]
Ecore model is generated though the grammar description of the Xtext expressions. It is true that MWE2 actually use ECore model of the grammar, rather than directly looking to the Xtext Descriptions. However Ecore Model is used in MWE2 to feed the needs of the scoping, again other components are linked to the related libraries and code artifacts via scoping with Google Guice.

Therefore we can state that in the current system, a change can be made in:
- Domain Specific Code Model - (Grammar)
- Xtend Templates
- Code Models (Domain Specific Language Code)
- General Purpose Code which is generated by using previous three sub-mechanisms.

Thus, we reach to Figure 6.5. This gives us abstracted and aggregated clusters of development phases, which would later be used as the target domains.

![Figure 6.5 Phases of DSL Development in Xtext Framework](image)

### 6.4 Code Complexity Analysis

Now, we know the segments of the framework that can be affected by the users of the framework. Thus we can create a web of casual dependency relations from the changes that will occur in one of the phases. However, we must select targets and their path that the information flow occurs. By doing this, we can classify the impact of change analysis in limited sections of the chosen entities. In order to make a target and path selection, we should...
classify the code entities and elements of the target domain.

6.4.1 Code Entities in Domains

6.4.1.1 Domain Model Domain

For the given environment, Xtext grammar description files which correlates to model code entities can be identified as the code entities of the domain model target domain. Grammar description files are under targeted Xtext project directory. In RHMI project, grammar description has been distributed in several sub-projects. These projects can be traced thought the main project, since all sub-projects are called into main project as ECore imports.

Thus, starting code entity with the highest hierarchic order is identified as code entity description “de.bmw.rhmi.hmi”. It must be noted that all other imported code entities are sub-entities of de.bmw.rhmi.hmi grammar; however in this case, they have the same matched level of complexity among each other.

Table 6.1 Sample Code for Code Entities

```plaintext
grammar de.bmw.rhmi.Hmi with org.eclipse.xtext.common.Terminals

import 'http://www.eclipse.org/emf/2002/Ecore' as.ecore
import 'platform:/resource/de.bmw.rhmi.resource/src-gen/de/bmw/rhmi/resource/Strings.ecore'
import 'platform:/resource/de.bmw.rhmi.resource/model/Image.ecore'
import 'platform:/resource/de.bmw.rhmi/src-gen/de/bmw/rhmi/App.ecore'

generate hmi 'http://www.bmw.de/rhmi/hmi'
```

6.4.1.2 Model Code Domain

Files that correspond to predefined grammar in domain model descriptions can be defined as the code entity of model code target domain. These files are under the runtime environment that is specific to individual settings for a runtime call. For RHMI project, these runtime environment variables are provided for the developer under de.rhmi.editor/RhmiEditor.product file.

Model code has to obey the dictation of the grammar, thus the domain model. Since domain model can be traced to de.bmw.rhmi.hmi code entity, model code entities are correlated to
de.bmw.rhmi.hmi and its sub-entities. Additional to these files, Xtext framework creates validation and formatting dependencies to the model code development environment. Model code entities are schemed by domain model definitions. Therefore while matched level of complexity among model code entities of the same project are the same, they must be converted to a comparable version as an entity state. Further discussion on this issue can be found in limitation conditions.

6.4.1.3 Code Generation Templates Domain

All template files that is used in inferring the model code is perceived as code entities of the code generation domain. Code generation code entities are descriptions that lie under separate projects within the plug-in platform. These are
de.bmw.rhmi.ios: for Objective-C code generation
de.bmw.rhmi.java: for Java code generation that targets Android environment.

Code entities of code generation domain are separated in Xtend description files, but every code generation template accesses the domain model resource definitions from the ECore model of the interested identifiers. Thus, templates use the same key identifiers to generate different code. This has to be specified during the runtime environment project creation. If project is set to use de.bmw.rhmi.ios code generation files, then it will generate Objective-C code from the model code. Else if project is set to use de.bmw.rhmi.java code generation files, then it will generate Java correspondent code.

In current setting, all generated code is placed in runtime directory. Generated code that corresponds to related model code can be found under the target folder of the same directory.

6.4.1.4 Generated Code Domain

Currently in target project RHMI, generated code is produced via code generation templates that are in either Objective-C code which supports IOS environment or it is in Java code that supports Android Environment.

6.4.2 Code Elements in Target Domains

6.4.2.1 Domain Model Domain

Domain model code elements are individual grammar definition statements and fields that obey the rules that are defined in Xtext language. Thus, they relate to each other in the OMG based modeling terms. All code elements of Domain model have an ECore model correspondent. If code element also carries a constraint description, these are filed in an XMI description and they are used in dictating model code key typing during model code
development.

6.4.2.2 Model Code Domain:

Model Code Elements are inferred correspondents of code elements of domain model and they indirectly relate to generated code via code generation templates. Therefore, in RHMI project, a domain model correspondent description that appears in an independent declaration is identified as a code element. It must be noted that RHMI is designed to pass the behavioral code body to generation templates, while it is structured as a script. Thus, it declares attributes without eventually describing behavior in model code itself. This construction result as in following situations:

Table 6.2 Same Code for Code Elements

```plaintext
app SomeAppAsCodeEntity
state AStateofApp{
    toolbar{
        button AButton{
            text SomeText
            image AnImage
            onRelease
        }
    }
}
```

In this code example of RHMI Editor Project, `state AStateofApp` is the code element which envelopes `button AButton` attribute via `toolbar` attribute of itself. Behavior of the `state AStateofApp` is to declare a `toolbar` and add the `button AButton` to this `toolbar`. However, behavior of the `button` is independent from the code element `state AStateofApp`. Thus, `button AButton` is a code element declaration as itself. Therefore, the following is possible in RHMI:
Table 6.3 Equivalent Code Elements Example

<table>
<thead>
<tr>
<th>app</th>
<th>SomeAppAsCodeEntity</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>AStateofApp{</td>
</tr>
<tr>
<td></td>
<td>toolbar{</td>
</tr>
<tr>
<td></td>
<td>AButton</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>button</td>
<td>AButton{</td>
</tr>
<tr>
<td></td>
<td>text</td>
</tr>
<tr>
<td></td>
<td>image</td>
</tr>
<tr>
<td></td>
<td>onRelease</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

This will generate the same code as a result, but the code body structure is different from each other.

All model code elements carry identifiers that are structured in domain model. Therefore, except constraints such as validation, formatting or behavioral relations, key identifiers are directly deduced from domain model elements.

6.4.2.3 Code Generation Templates Domain

Code generation template code elements are the fields and methods of Xtend class descriptions. Code generation template code elements are not directly related to model code, or to domain model, or to generated code. Code generation code elements are interim elements that carry the behaviors for the code element relationship among other domains.

```java
def fileName(AbstractStateWithComponents state) {
    state.name + "View.h"
}
```

In the given code example, code generation template code element is identified as:

```java
def fileName(AbstractStateWithComponents state)
```

And its behavior is to add “View.h” textual body to the name attribute of the state element.
6.4.2.4 Generated Code Domain

Generated code elements are methods and fields that are generated from the generation templates. Thus, they are constructed from the descriptions of the generation templates that infer the model code.

In this sense, it can be said that all code elements are obligating to tasks that are designed by the language developer for RHMI language and they are bound by two aspects. Firstly, assuming that developer wrote the generation templates in order to work in a general purpose language for a dependent platform, generated code obeys to design limitations of its target platform, such as IOS platform or Java platform. Secondly, though the model code, one can always map the structure of generated code, also the opposite is true, because generated code can be used to infer the model.

6.5 Impact of Change Analysis

In this section, limitations and expected conditions for the synchronization attempts will be analyzed via the impact that any change attempt on a domain can have on other domains. Therefore, we must first overview the aimed behavior and foresee the possible outcomes via our knowledge from overall analysis and target domain segmentation and code entity and code element structure in segmented target domains.

Using the overview analysis, and code entity and code element explanations, we can conclude the relations among target domains mapping as in the Figure 2.2. Domain Model entities behave as model in relation to both model code and code generation templates. Thus they map as if model realm entities relates to code realm entities. On the other hand, Model code has a model to code relation to generated code. This is also true for generation templates. Since generation templates dictate the structure of the generated code by using model code and domain model, it behaves as the model for generation, thus behaves as the model for the generated code. Generated code is the final result of the framework and it acts as a member of the code realm for all situations.

Aimed behavior states that changes in the domain specific language code must be synched to generated code. This would mean to reflect the changes in the model code to generated code entities and elements, and vice versa.
6.5.1 Limitation Conditions

In order to understand the limitation conditions, one must underline the relations of the target domains in relation to aimed behavior.

6.5.1.1 Domain Model and Model Code:

Model Code structure consistency is dictated in a one dictation flow. Domain Model designates what model code is and what structure it must hold. Thus, Model Code does not have any impact on domain model unless a reverse response mechanism is additionally placed to already existing framework mechanism.

On the other hand, domain model explicitly dictates over model code. Changes in domain model have direct impact on consistency of model code in following conditions:

- If model code consists a domain model code element or entity correspondent, and domain model that relates to this correspondent is obliterated, substituted, or modified in a way that clashes with the previous valid definition.
- If model code consists a domain model code element or code entity as correspondent and behavior constraints of domain model that relates to this correspondent are obliterated, substituted or modified in a way that clash with the previous valid definition.
- If model code consists a domain model code element or code entity as correspondent, and behavior validation conditions of domain model that relates to this correspondent are modified in a way that clash with the previous valid definition.
- If model code consists a domain model code element or entity as correspondent, and behavior formatting of domain model that relates to this correspondent are obliterated, substituted or modified in a way that clash with the previous valid definition.

6.5.1.2 Domain Model and Code Generation Templates

Similar to relation of domain model and model code domains, domain model and generation templates have a one way consistency relation. Domain model does not dictate the structure of generation templates directly. However, in order to use the domain model description, generation templates include dependency relation to domain model structure.
Hence, domain model implicitly dictates over generation template elements.

Changes in generation templates have no effect on domain model consistency. However, Domain model impacts in the integrity and consistency of the generation templates via dependency relation that code elements and code entities of generation templates holds. Thus, generation templates lose consistency if generation templates uses a domain model resource and this resource is obliterated, substituted or modified in a way that clash with the existing generation template structure.

**6.5.1.3 Model Code and Generation Template Codes**

Xtext framework uses model code to guide generation templates. Model code is red and used in generation templates. Though model code, generation template structure matches with domain model resources with model code elements that correspond to domain model code element.

Thus, model code has no dictation over code generation templates, and code generation template has no dictation power over model code. However, since they use each other in order to successfully generate final code product, there is a dependency and consistency relation between them. Therefore, there is not a direct limiting condition that occurs between two sides.

**6.5.1.4 Model Code, Generation Templates and Generated Code**

Changes in model code is direct impact on generated code, since generated code loses consistency once the model code that it is generated has been modified. This is also true between generation templates and generated code. In current state, a change in generated code has no way to be reflected to model code or to generation template. However, since model code and generated code are directly related, actually generated code elements can be mapped...
backed to its model realm, thus its model code correspondent.

Impact of changes between model code and generated code has four limitations.

- Model code has to use generation templates while providing behavioral code body for generated code. Therefore, generation templates have to be included in the synchronization attempt where behavioral code will be synchronized.

- Since model code does not hold behavioral code body, it directly impacts identifiers of code elements. On the other hand, generation templates hold the placement of identifiers with correct qualified names and designated behavioral code body. Thus, changes in generation templates can have an effect on both behavioral and identifier levels of the generated code.

- Moreover, model code is limited though domain model dictation. Hence, model code is well defined in generation and has a concrete structure that has a direct reflection on generated code.

- Most important limitation is that, the changes in generated code have to concede with both model code and generation templates. In relation with model code, generated code has to be mapped back to a defined model element that is consistent with domain model. Thus, changes in an existing model code element or entity identifiers has direct relation to corresponding generated code, and element subtractions or element alterations on can be reflected back with tracing those correspondent identifiers. However entity level element additions and alterations that changes the code as code...
element modifications has to relate to both generation templates and model code which is dictated by domain model.

- All generated code element level modifications are prone to a clash with other generated code elements that uses the same generation template, because model code elements that obeys to same grammar rules from the domain model end up using the same templates. Therefore, if a generated code’s code body has a change that is other then identifiers that fully responses to model code correspondent and these changes are directly reflected to corresponding target domain elements and entities, then other generated code elements that uses the same generation template code element will lose their consistency with their correspondent model code and generation templates.

6.5.2 Expected Conditions

In expected conditions of synchronization, thought the limitations among relations of target domains, one can deduce the following to sustain the aimed behavior:

- Domain model can be left as the sole dictator domain, thus it can be left out of synchronization attempts. Therefore synchronization behavior setup targets other three domains.
- Aimed behavior states that desired synchronization behavior is between model code and generated code. However, generation templates relate to changes that occur on generated code entities. Thus, it is expected that identification information of a code element is provided by model code, and rest of the code body is provided in the generation templates.
- Changes in the model code do not affect the consistency between code generation templates and model code, but it impacts the consistency of the generated code to the model code.
- Changes in model code have to be directed to existing correlated generated code products. However, since behavioral changes are embedded in generation templates, basic synchronization behavior that was captured by entity states would be sufficient for synchronization attempt which directs from model code to generated code.
- Changes in generation templates have to be directed to existing correlated generated code products.
- A template code that exists in generation code template’s code elements contains the behavioral templates, and they can be designed in various complexities by the developer that designs them. Therefore, it is a requirement to trace the generation templates to generated code body if there would be synchronization in one of the sides
to the other, and it is expected that they require complex type synchronization behaviors.

- Since a change in generated code has a chance to create conflicted impacts of change, it is expected to handle these conditions in complex synchronization behaviors. Furthermore, as mentioned in limitation conditions, when directly reflected, synchronization attempts from generated code modifications may create a clash of relations within other generated code elements.

- During a synchronization attempt from generation templates to generated code, if the code templates are modified, then related model code elements are required to be used in providing generation ability.

- It is expected that users for the mechanism will be notified whenever there is a calculated loss at the end of the synchronization attempt, and their permission will be required. Thus, there will be a call for ability to catch developer’s intentions.

- When generation templates are changed, it is expected that the intention of this change is to impact the generated code. However, there are three different decisions that can be implemented for such impacts. One of three is to find and change all related generated code elements that used the changed generation template code. The other one is to allow creating differences between the altered codes, thus creation of versions of the elements that used the same generation templates. Similarity calculations can be used in determining the distance of change among versions. Lastly, in order to keep the consistency level the same among code elements while keeping the implemented changes on the code body of generated code elements, changed elements are unlinked from the model code and the generation template that they had related. This would create a partial lossless synchronization.

Additionally, there are cases where ability to catch the developers intention is required. In these cases, following steps are expected to be followed:

A message must inform the user/developer about the situation that his/her guidance is a requirement for a synchronization attempt. An intention catcher message must include at least the following:

- Compared states and their entity state details.
- If there is any, the confliction among the following steps that causes the need for developer’s guidance.
- Expected and analyzed paths of synchronization attempts, together with expected results of those analyzed paths of synchronization attempts.
- Ability to receive the selection among given paths.
It is advised to catch the developer’s intention for all cases where there will be a loss of information in any domain. This includes all types of relation, code body, or consistency related information. For given aimed behavior, it is stated that “a synchronization attempt is not desired in every situation. Thus, it is expected to have both manual and automated initialization mechanisms for synchronization attempts.” For this reason, it is expected that automated conditions includes entities and elements, and initialization conditions should be in control of the developer.

### 6.5.3 Exception Conditions

Foreseen exception conditions are as below:

It was considered that domain model code would not be included in current synchronization mechanism. However, in cases where domain model is changed, conditions vary among type of changes.

Entity obliteration and substitution on domain model domain have greatest impact on all target domains. There are two reasons for this:

- Since they lose their dictating initial model elements, once there is an addition modification in model code, or generated code, it cannot be guided as previous state where a guiding grammar exist. Overcoming this issue requires either deduction of grammar once again from model code or generated code, or it is left for manual settings from the users.

- In a case where domain model is changed intentionally, it is a logical clash to attempt synchronization back from model code to its Meta Model. This means to undo the changes.

Only entity level substitution and element level alterations on entities and elements can be directed from domain model to existing model code entities. Other type of changes would require manual generation feed from the users.

In relation of generation templates and model code, the dependency is indirect and it is bridged via an ECore model that is generated from Xtext model code. Generation templates know about the model code elements via importing this ECore model. Therefore, changing this ECore model creates and exceptional condition. These situations are not included as a synchronization attempt; however, it must be signaled with an informative warning to the user of the environment.
6.5.4 Target and Path Selection

In order to achieve the conditions of aimed behavior, model code domain, generation templates domain, and generated code domain has to be included in the synchronization phase. Thus, interested target domains for the synchronization attempt are these three domains. They need to be analyzed during the synchronization behavioral decisions.

6.5.4.1 From Model Code to Generated Code

**Element Level:** If a model code element is obliterated, corresponding generated code elements are expected to be obliterated. Therefore, it can be said that synchronization attempt is directly related from model code to generated code elements. On the other hand, if a model code element is substituted; corresponding generated code elements are expected to be changing the corresponding element identifiers of its generated code elements.

Model elements cannot force on code body modifications by themselves. Model code side includes the identifier changes. Therefore, it is not a necessity to include the generation templates unless

- It is a requirement by design of other sub-mechanism or mechanisms, such as tracer or change analyzer.
- Generation templates use identifier information to infer behavioral code in a way that impacts the behavioral code body more than identifier or qualified name placement.

Other than situations that are mentioned above, there is not an element level modification situation.

**Entity level:** Whenever a model code entity is obliterated, this would require obliterating all related entities that corresponds to obliterate model code entity in generated code’s corresponding entities and elements. If a model code entity is substituted and related code elements are in “one to one” relation, this would not have a major effect on other correspondents; synchronization attempt can be downgraded to standard refactoring operation on generated code. When there is “one to many” relation or “many to many” relation, then identifiers must be handles to keep the consistency in all related code elements.

Entity modifications are covered in element level changes.

6.5.4.2 From Model Code to Generation Templates

It must be stated that, there is not a necessity to establish a synchronization attempt between direct or indirect impact of change from model code to generation templates.

6.5.4.3 From Generated Code to Model Code
Element Level: If a generated code element is obliterated, corresponding model code elements are expected to be obliterated. Therefore, synchronization attempt is directly related from generated templates to model code. However, if a generated code element is substituted; corresponding generated code elements are expected to be changing the corresponding element identifiers of its model code elements. Moreover, changes has to be concluded after ensuring that changes on generated code elements covers all identifiers and qualified names that correspond to the related model code identifier or identifiers. The model code element and its correspondents have to be mapped together and must be analyzed in the same synchronization attempt.

Element level modifications on generated code are prone to create problems of consistency for a synchronization behavioral analysis. It must be noted that these types of modifications will clash with other generated elements as explained in limitation conditions. For this reason, synchronization attempt must deal with these situations by ability to catch developer’s intentions.

Entity level: Whenever a generated code entity is obliterated, this would require obliterating all related entities that corresponds to obliterate model code entity in generated code’s corresponding entities and elements. If a generated code entity is substituted and related code elements are in “one to one” relation, this would not have a major effect on other correspondents; synchronization attempt can be downgraded to standard refactoring operation on generated code. When there is “one to many” relation or “many to many” relation, then identifiers must be handles to keep the consistency in all related code elements.

6.5.4.4 From Generated Code to Generation Templates

Element Level: If a generated code element is obliterated, then obliteration does not have to be synchronized back to reflecting generation templates. This is because generated templates are has a one way production relation with generation templates. A loss of a whole generated code element does not impact the consistency of this relation. If a generated code element is substituted, this case has the potential to cause a confliction. This is also true for modifications that cause a conflict of consistency with their generation templates, because of the changes in their behavioral code body. It must be noted that obliteration and substitution will also lead to an impact of change on model code.

For cases where generation templates might require a change in themselves that is directed from generated code, there is a chance that corresponding generation template has “one to many” or “many to many” multi-response relation to all generated code’s code elements that were generated from the same template. For this reason, during a
synchronization attempt, whenever a code element is modified in a way that impacts the behavioral code body excluding the parameter identifiers, there is a chance that any change on generation template will cause inconsistency among other generated code elements.

**Entity level**: Obliterations of generated code entities do not impact the generation templates. Entity substitutions are only effective on corresponding code element that carries the methods/functions that carries the code body’s initialization and these functionalities are dictated on generation template from model code entities. Unless developers of generation templates design a one to many or many to many relations for their entity identifiers, then synchronization attempt can be downgraded to refactoring functions. On the other hand, in one to many relation and many to many relation among code entity identifiers, this

### 6.5.4.5 From Generation Templates to Model Code

As from model code to generation templates, it must be stated that, there is not a necessity to establish a synchronization attempt between direct or indirect impact of change between model code and generation templates from each sides. Only exception of this issue is the entity substitution of generation templates which would have an impact on imported model code templates. These imported templates are ECore models that are generated from model code. Therefore it is an indirect relation, and in a substitution attempt on import declaration, model code to generation template relation is broken. However, since this would create a compiling issue, unless it is replaces by an equivalent ECore model, there is not any realizable treat for consistency relation. Therefore, this condition is included as an exception.

### 6.5.4.6 From Generation Templates to Generated Code

**Element Level**: If a generation code template’s code element is obliterated, corresponding model code elements are expected to be obliterated. However, when there is an element level substitution, then there is not any requirement for synchronization unless parameters of the code identifiers are affected by the change attempt. Otherwise, it is highly possible that changes in code identifiers are leading to changes in the code templates or at least the holistic outcome of the generation, thus having an impact on final code product. Therefore, it is more prominent to be interested in changes in code body of generation code templates.

In cases of element level modifications that appear on generation templates, mechanism must ensure that reflection for impact of changes covers two conditions. Firstly, in conditions which has generated code correspondents that was changed prior to a synchronization attempt from generation templates, a complex behavior occurs. Since a generation template can be
Chapter 6  Applying Design Methodology to Xtext Framework

responsible for generation of all model code elements though the same code elements, this complexity is highly prone to include multi-response cases. However, in this case, trying a synchronized state would mean to settle a unified state on generated code side. This would lead to loss of prior information. Therefore, separate matching must follow state division behavior that result differently according to desire of the user. While the generated code does not include any changes on themselves prior to synchronization attempt from generation template side, then relation is still prone to multi-response. However, this time separate matching would result as if there was not any state division type behavior during separate matching.

**Entity Level:** When generation code template’s code entity is obliterated, impact of change on generated code is to obliterate related elements and entities after receiving user’s/developer’s approval. If permission is not granted, then their relation with the generation template must be broken elegantly via state division process. Entity level substitutions are discussed in exception conditions. Entity modifications are directly related to element level changes.

6.6 Mechanism Model: Suggestions

In the case in which the Mechanism model uses basic complexity support as the target behavioral aspect for the aimed behavior, following solution design is sufficient. Mechanism model consists of minimum number of sub systems. These are four basic sub-systems for a synchronization mechanism. These are Relation Tracer, Change Notifier, Behavioral Decision and Code Generation. An overview of the suggested solution for RHMI project can be seen below:

As depicted in Figure 6.10, the entity tracer mechanism is using an external link storage system that processes the code elements. Serialized code represents the conversion of the tracing information into a comparable structure as discussed in section 6.8.3. Moreover, working processes of the tracer, change notifier and generation mechanism follows the fundamental design features. These design features can be seen in section 5.4.

Additional to basic design features, it must be noted that, workflow includes three target domains; therefore code generation descriptions, code that is written in domain specific language and generated code that is developed to give the general purpose language equivalent of the domains specific language is added to the synchronization workflow.

The main reason for using an external tracer mechanism is to provide the ability to support the relational mapping of three target domain code entities which are prone to impact
of each other. This was explained in previous analysis section of target and path selection, which was documented in section 6.5.4. In the following, sub-mechanisms within the basic synchronization workflow are explained.

6.7 Features of Synchronization Mechanism

Desired capabilities of a change notifier mechanism would differ as the desired level of complexity of synchronization behavioral support varies. However, four main change analysis specifications are provided below according to targeted complexity level.

6.7.1 Compared States of Entities:

It is expected that synchronization attempts include entity sets of model code, generated code and generation templates. Because of aimed level is for basic behavioral complexity, rapid synchronization attempts are required. This means to try analysis and implementation of the synchronization attempts as soon as the changes are committed on target domains. Thus,
entity state records would be minimized. However, in cases where developer does not permit the initialization of the synchronization, then changes are not reflected. This would result in affect on secondary attempt; previously changed elements would not be included in the synchronization attempt, unless it is manually included by the developer.

Captured states are only the current versions of the domain states. They are compared among their current versions and no edit history is provided. However, series of serialized entity state schemes can be captured in serialized code models. For external storage of entity states, section 5.5.1 can provide a starting point.

6.7.2 Time of Analysis:

Initialization of analysis is expected to start as the change has successfully committed. Moreover, since manual synchronization requests are to be expected, manual request messages are to be provided to change notifier mechanism. Therefore, initial request messages are tied to two different systems. One of them being the request for altered elements when a successful commit of change occurs. This would mean to check the impact of changes when a successful commit occurs with no error on the target code entities. Secondly, manually, the developer who uses the code development framework must be provided with a synchronization request option. These expected conditions can be traced on section 6.5.1.

6.7.3 Correct decision:

Correct decisions are required to be deduced in synchronization behavioral analysis. Behavioral decisions are aimed to follow basic support conditions. Basic support conditions can be found in section 5.2.6.1, under the title of basic support conditions. Basic support consists of basic behavioral conditions that are discussed in section 4.1.

6.7.4 Implementation of Decision:

Implementation decision is to follow to select the basic implementation decisions, which are:

- All conditions that are outside of the basic conditions will not be implemented.
- All conditions that fit to basic conditions are implemented within maximum two steps, which fit to basic conditions of non-functional to functional and then to functional to synchronized outcome.

6.7.4.1 Change Notifier

Change notifier for the RHMI project includes the specification of section 6.7.1. Additional to those specifications, change notifier is responsible in calling for all related code
entities that has an impact from the changes that occurred. This is done within the impact checker object though the predefined impact of change analyzer block. In this analysis attempt, all related elements are fetched via an inner process of a code search request from the information that comes via traced relation data. Once impact checker prepares all related impact relations, this information is passed to synchronization behavior decision mechanism. Decision Mechanism is responsible for deducing the correct decision for implementation decision. Further discussion for techniques on building change notifier and impact of change analysis can be found in section 6.8.2.

6.7.4.2 Relation Tracing Mechanism

Traceability mechanism holds the basic design features which are depicted in section 5.4.1. Moreover, a serialization feature is feasible to use in addition to fundamental tracing mechanism. This addition is the model schemes that are provided to become the basic model that is used in creating the model elements and entities in a comparable abstract form. When these serialized models are stored as versions of the code entities, it is possible to create backward compatible synchronization attempt which supports versioning in addition to synchronization. A general discussion on traceability can be found in section 6.8.1, for uses of serialization please check sections 5.5.3 and 6.8.3.

6.7.4.3 Code Generation Mechanism

Synchronization mechanism would need to have a generation mechanism. Generation mechanism abilities vary among the given elements. However, generation analysis must provide a “yes” or “no” reply to the behavioral decision mechanism whenever the mechanism is initialized by the synchronization attempt among two entity states. Thus, generation mechanism is designed to an ability control request call from the synchronization behavioral decision mechanism. In order to achieve this goal, generation mechanism uses a preemptive check on code generation descriptions that it owns. Generation Mechanism and Behavior Decision Mechanism achieves the generation ability checks as the following:

Parameter Query Object which is used in Behavior Analysis Processor Object calls on generation mechanism with the information that is provided to Behavioral Analysis Process. Hence, Behavioral analysis concludes to the direction and mutability of the compared states before sending a check request via Parameter Query Object to the Generation Mechanism. Once the direction is ready, a combination of code entities with a composition order is send to generation mechanism. Generation mechanism is analyses the prepared composition, and resolves the combination without generation of the code, but by checking the generation
ability from the generation descriptions that lies within the code of its main mechanism. This checks is then deduced to a holistic outcome of Boolean decision parameter as “yes” or “no”. This reply is send back to Parameter Query Object. Parameter Query Object collects all the parameter together and composes the initial behavioral analysis parameters.

Lastly, generation mechanism would receive an implementation request from the synchronization decision mechanism in cases where behavior mechanism decided to run a real implementation process. This time, a concrete set of behavioral decisions are passed to the generation mechanism with instructions to replace the code entities with the new generated code entities. Generation Mechanism follows the implementation path descriptions and realizes the pre-defined behavior. Moreover, a possible use for similarity calculations for final checks on generated code bodies is discussed in section 6.8.4.

6.7.4.4 Behavior Decision Mechanism

If the aim for the synchronization mechanism is just to create a basic behavior level support, then all synchronization attempts are targeted to be answering the synchronization behavioral parameters according to basic behavioral conditions. In order to keep information loss at minimum while synchronized items on maximum possible level, “partial lossless” result is suggested to be provided as the desired result as a default user desire on synchronization mechanism. Moreover, additional discussion on analysis of changes is provided in section 6.8.2.

After the provided parameters are seen compatible with the given attempt, then analysis jumps to implementation, in order to achieve the foreseen result. In order to analyze the possible result, behavior decision mechanism processes the provided traced relation information and code entities in behavioral analysis processor object. This object calls on two major other information providers. These are parameter query object and behavioral analysis processor descriptions. Thus, behavior analysis is divided between collection of data and behavioral analysis process. This predefined analysis process is set to check the conditions that is necessary for synchronization behavioral types which is discussed and provided in chapter 3.

When behavioral analysis is complete, first action is the control of the deduced possible result parameters with desired result. If the control gives a positive result, then an implementation path is created as a composition of processed functional processes which is passed from the predefined behavioral analysis process. Implementation decision process calls this implementation path and with combines with trace information and directions for code entity type changes, which provides code location and implementation realization steps for
generation mechanism. Later, implementation mechanism calls on the generation mechanism by the run implementation process. However, in cases where the decision control gives a negative result, alternative process to real implementation process is called. Alternative process includes termination of the unsuccessful synchronization attempt, retry of the synchronization attempt within the expected conditions, or another process that is defined by the developer of the synchronization mechanism.

Predefined Behavioral Analysis Setup can be divided to basic behavior setup and complex behavior setup. These are explained in the following:

6.7.4.5 Basic Behavior Setup

All conditions that provides the desired result by the user achievable, and that starts in basic condition list that is provided in Appendix 5 and Appendix 6 is listed in basic behavioral setup conditions.

6.7.4.6 Complex Behavior Setup

For the basic level synchronization aim, it is sufficient to result in failure to start a synchronization attempt when a complex behavioral condition appears. These conditions are provided in Appendix 3 and 4.

6.8 Findings for Xtext Framework Synchronization Solution

6.8.1 Findings for Xtext Framework Synchronization Solution for Tracing Mechanism

Traceability mechanisms that capture and store the relation of the model code, generated code and generation model can as varied as the modeling and source code traceability systems that is studied in various studies on the field. Different capabilities of different traceability mechanism would allow different levels for complexity in synchronization behavior. Specifically for the Xtext framework, In order to support the basic level synchronization attempts, tracing mechanism needs to be able to relate to compared entity states.

Within Xtext framework, there is no native mechanism that can be used for identifying code elements and entities. During this research, only mechanism for comparison purposes that was known about is EMF Compare project. EMF Compare can match EMF models and can create reports for caught differences between models [48]. However there is no extension to either Xtend or generated code by Xtend [5].

There are various types of mechanisms that can be implemented. Unfortunately, even
after an extensive search with deep involvement, native mechanism that can be used for identifying code elements and entities within Xtext framework couldn’t be found. Only EMF Compare project was noticed. EMF Compare can match EMF models and can create reports for caught differences between models [48]. However there is no extension to either Xtend or generated code by Xtend [5]. Moreover, EMF compare is specialized on EMF model itself, thus it is not comprehensive to the needs for synchronization mechanism.

There are two main challenges for choosing and implementing the traceability mechanism for Xtext framework. One of them is to choose how to identify the correspondents. Xtext framework mechanism works in layers which link to each other. Every phase of development uses another type of technique and coding language to realize itself. Thus, traceability mechanisms get intertwined with inconsistency management.

Among various tracing approaches, many traceability techniques have been analyzed. Among these traceability techniques, one must try to find the techniques that are in correlation with the structure of Xtext Framework. However, among the techniques that were included in the research, the one with necessary abilities could not be founded. Yet, they were influential on suggested solution for Xtext Framework. They can be found in section 5.5.1.

6.8.2 Findings for Xtext Framework Synchronization Solution for Changes on Code Artifacts

In Xtext framework, three target domains are in interest for major tiers of synchronization. However, all tiers have different requirements on how to realize this impact of change. Additionally, for Xtext mechanism to inference into a new state, there must be a decision mechanism that is responsible for reflection of changes to other artifacts. In order to do this, after analysis, collected data must be converted to impact information and serve as a medium to make the correct changes. Moreover, these changes have to be implemented by another sub-mechanism. Hence, one implicit addition to this mechanism is able to be built on traceability mechanisms that support desired versions of software evolution analysis. This is meant to be achieved in serialized versions of code entities from different target domains.

In this research, the chapter 2 is meant to be used as the basis for the solution for analysis of the changes on the code artifacts. Furthermore, all possible change types have been classified and explained in the Xtext environment for the code bodies in section 6.5.
6.8.3 Findings for Xtext Framework Synchronization Solution in Serialization

XML is used in EMF technology and grammar representations are in XMI, which is part of XML [49]. XML is used in constructing ASTs in various forms and both complete or part of an entity. Moreover it is used in various other variants of documentation of object models, hence enforcing future compatibility to other applications. Some examples for the uses of XML as a data storage and AST scheme can be found in [50], [51] or [36]. Further discussion on use of serialization can be found in section 5.5.3.

6.8.4 Findings for Xtext Framework Synchronization Solution on Similarity and Equality Calculations

After the overview on similarity and equality studies, there were two distinctive conclusions. First conclusion was that similarity studies can be used to increase the abilities of Xtext framework into code element semantics synchronization. And secondly, calculation of level of equality and similarity was not creating a direct impact for a synchronization attempt, unless either heuristic result of compared elements are taken as sufficient level of synchronization, or synchronization attempt is going to use a loss retrieval and maintenance mechanism for its sub-mechanism. In this research, it was not yet suitable to be tailored to Xtext framework, but future studies should increase for the future corrections on this conclusion. Further discussion on similarity calculations and their possible uses on synchronization solutions are discussed in section 5.5.4.
Chapter 7 Conclusion

In this research, synchronization among code artifacts has been deeply analyzed. Firstly, the synchronization as an event have been explained and classified. Later on, solutions for creation of synchronization as an action are studied. From these studies, synchronization attempts are divided to a behavior and implementation processes. The reason for this division was the incentive to protect the existing code structure from unsuccessful and harmful interventions.

Results of this study are the following:

Relations among code artifacts, especially on the ones which follow the object oriented design patterns, have been studied to satisfy the needs for understanding the behavioral conditions for synchronization processes. In order to simplify the synchronization process, a singular action for a synchronization request has been defined. This was called as the synchronization attempt. Synchronization attempt have been studied in the light of the modeling concepts.

The reason for using modeling concepts to analyze the needs for realizing the synchronization is firstly because design patterns for code artifacts was the link for creation of the reflections of the code artifacts from different domains. These domains where defined with a characteristic, which can be concluded from their role and relation to other code body. This relation was found in the related previous studies on modeling and source code relations. Such relations are reclassified to be used in synchronization analysis. From these classifications, it was understood that two code bodies resembles their role to their relation too. Therefore, it was concluded that a among two code bodies, there is three major relational characteristic. These are model to model, model to code, and code to code relations. Furthermore, code is seemed to be evolving within a development process, as it was pointed out by other accounted studies from the field. This evolution fact is identified the basis for the necessity of a synchronization attempt.

Definition of the nature of a synchronization attempt is divided into two fundamental concepts. These are the limits of the synchronization ability and the desired result of the synchronization from the synchronization. Together, limitations provide the basic conditions of the synchronization, while the desired result is used as a control phase to provide the confirmed synchronization outcome. When two major conceptual analyses give a positive response, it is concluded that this conditions is desired and sufficient to start the
implementation of synchronization. Additionally, code bodies are classified according to their complexity. Such a complexity difference is deduced from the relations among the code artifacts and also from the roles that they resemble on the conceptual design. Thus, modeling concepts is fundamental to hierarchic classification of the code artifact’s complexities. Moreover, the types of changes that can occur on a code body are studied and the possible versions of changes are classified. These classifications are later used in simplifying the analysis of expected conditions during synchronization.

The findings from synchronization analysis which provided the nature of synchronization, perceived code entity and code element limitations, and possible types of changes are used in creation of synchronization behavior calculation methods. These methods realize themselves by a perspective in which the code bodies are compared to their matching complexity correspondent. This ensured the safety of the consistency on roles of code artifacts after the synchronization attempt. Moreover, this provides the limitations on code bodies that are necessary for initializing the synchronization analysis. From this perspective on the code bodies, synchronization behavior was defined and classified, also fundamental tools for enhancing the capabilities for the synchronization behavior were provided. It must be stated that synchronization behavioral decisions is directly related to implementation of the synchronization implementation. Therefore, the tools and behavioral analysis is also the basis for the implementation of synchronization attempt.

Synchronization behavior is designed to be working in creation of cluster-like relations among the compared code bodies and later on mapping these cluster domains to each other via dependency relations that they hold for each other. These relations create the need for tracing the code parts that is related to each other. After conducting the mapping of such relations, a functional calculation process or processes is formed. These calculations are also the foundations for realizing the synchronization attempt, because during the deduction of the calculation result, same calculated path is meant to be used in creation or the implementation path.

By their nature, functional calculations are divided into non-functional and functional relations. Moreover, according to type of mapping those domain members hold for each other, required form of behavior is in need of change. Therefore, conditions of the compared code artifacts are defined as code entities that reflect a specific type according relational dept. The term “complexity” has been used to identify these conditional differences. As a result, different relational conditional relations which require different steps to be solved are classified in to different behavior types.

Behavioral types are then aggregated into two major characteristics. These are defined as
basic behavioral types and complex behavioral types. Hence, relational types are constructed in a hierarchy of complexity. Higher complexity also introduced conditions that are harder to resolve. The simplest condition is the basic type 1, while the most demanding condition being the complex type 5. This is because the most desired functional relation is defined as synchronization a functional relation which firstly achieved a bidirectional form and secondly generated an equivalent code element for each pair of domain and co-domain entity from both relational directions. Moreover, it must be stated that each increase step in this complexity hierarchy introduces a significantly different requirement for creating a synchronization solution.

The goal of the synchronization behavior is to find a way to bring the mapped relations into lesser and lesser complexity. Thus synchronization solution is converted into an attempt to decrease the complexity of the two compared code entities into more and more simpler form. By doing this, it can be claimed that every decrease in relational complexity type would result in more synchronized code artifacts. However, it is also true that it is not possible to achieve bijective and equalized relation among code bodies in all attempts. Thus, by further analysis and by the created functional method, it was inferred that synchronization behavior has four different types of final results. These are defined by two parameters. A synchronization attempt’s outcome can be defined by its relational completeness and the required losses during the achievement of this completeness.

It is claimed that, synchronization can be complete if all elements from each side has a functional correspondence and code equivalent. All other cases are considered as partially synchronized. Furthermore, it is stated that allowing the loss on information on code bodies changes the behavior in relation to completeness. Thus, allowing the loss on data has been determined as the other concluding factor. These two result parameters are originally both created and designed by using the desired result discussion on synchronization analysis. As a result, it is concluded that synchronization behavior checks its outcome via these two outcome characteristics, and if it is satisfied though the behavioral analysis calculations, real implementations are to be activated.

After analysis of the synchronization processes and creating a way to realize the correct decision method for the synchronization implementation, last requirement to realize the synchronization on a software development environment was to provide a synchronization mechanism that can use the defined synchronization analysis and behavior. To achieve this goal, a design methodology is created. This design methodology is aimed to find and map the code artifacts in a software development framework in steps. After completion the analysis of the target framework, how to tailor the synchronization behavior mechanism on the target
framework by using those analysis outcomes is explained. All necessary foundations of the synchronization mechanisms are defined and provided. Current results from field studies that use same fundamental mechanisms are provided as discussions. Fundamentals for the design patterns of those mechanisms are provided. Lastly, all findings about such studies during this research are summarized into an understandable from. Thus, the reasons and influences on the results of this research are declared.

Sub-mechanisms that are declared as necessary for realization of the synchronization behavior and implementation are classified into four fundamental systems. Synchronization mechanism needs to be able to infer and trace the relations among the code artifacts, so that it can create the mapping necessary for synchronization analysis. Moreover, same tracing mechanism is fundamentally necessary for reaching, fetching the correct code bodies and files, providing code locations and relations and constructing a memory system for more intelligent synchronization mechanisms. A change notifier mechanism is declared as necessary to be an interim system that uses tracing ability of the mechanism, and preparing them into code entity structures. Thus, necessary information for the decision process is fetched and provided for decision making. Decisions mechanism is stated as necessary in order to analyze the prepared code entity states and concluding the outcome of the synchronization attempt while creating the solution path for the implementation process. Lastly, a generation mechanism is seemed necessary to both provide the generation ability parameter to infer the synchronization outcome, and realize the implementation decisions.

Finally, design methodology was used on the target project, and the conditions for the basic synchronization conditions was provided in relation to already defined research outcomes such as the fundamental design patterns.

7.1 Future Work

This research is finalized in current status which has been summarized in conclusion. Further attempts for realization and implementation of the synchronization among code artifacts are necessary. These necessities can be briefly aggregated in to five topics.

Firstly, brain for the synchronization attempt must be constructed on the code development environment. This would provide the necessary test results to understand the weaknesses on the decisions mechanism if there is any.

Secondly, change notification, tracing mechanism and generation mechanism is only studied in the fundamental design workflow. Therefore, different approaches which can provide different and maybe better outcomes are prone to be found.
Third issue includes both traceability mechanism and generation mechanism. They can be designed in a various design patterns which would have different implications on deduction of the behavioral system. Different ways for creation of traceability and storing the traceability in relation to change notification and deduction of the code entities and code elements classification is open to research. Same can be said for the generation mechanism. Different and more efficient ways to deduce the implementation path and more efficient ways to infer the generation ability behavioral parameter is necessary.

Fourth, position and design of the change notifier is in open to research. This is designed as an interim mechanism which controls the starting and conducting the synchronization requests. Its, design must be tested for efficiency and alternative approaches which decreases the complexity of the design is open for research.

Lastly, synchronization behavioral calculations and classifications are open for reconstruction if the efficiency for the calculations is to be improved either by speed, better tooling or more intelligent handling of the code artifact relations.
References


References


126


References


Abbreviations

OMG: Object Management Group
MDE: Model Driven Engineering
MDD: Model Driven Development
MOF: Meta Object Facility
EMF: Eclipse Modeling Framework
UML: Unified Modeling Language
PSM: Platform Specific Model
PIM: Platform Independent Model
PSL: Platform Specific Language
DSL: Domain Specific Language
OCL: Object Constraint Language
XSD: XML Schema Definitions
JET: Java Emitter Templates
JTL: Janus Transformation Language
CVS: Concurrent Versions Systems
Appendix 1 Functional Relations

Functional Relation Cases

For a condition which entity state $A$ dictates the changes and entity state $B$ accepts the changes. *Functional Relation from $A$ to $B$ consist cases similar to below:*

1. **Functional Injective and Surjective Relation: Bijective Relation**

   \[
   \begin{array}{c|c|c}
   \text{Perspective from } A \text{ to } B & \text{Functional} & \text{Injective} \\
   & Yes & Yes \\
   \text{Perspective from } B \text{ to } A & Yes & Yes
   \end{array}
   \]

2. **Functional Injective and Non Surjective Relation**

   \[
   \begin{array}{c|c|c}
   \text{Perspective from } A \text{ to } B & \text{Functional} & \text{Injective} \\
   & Yes & No \\
   \text{Perspective from } B \text{ to } A & \text{Nonfunctional}
   \end{array}
   \]
3. Functional Non Injective and Surjective Relation

<table>
<thead>
<tr>
<th>Functional</th>
<th>Injective</th>
<th>Surjective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perspective from A to B</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Perspective from B to A</td>
<td>Nonfunctional</td>
<td>Nonfunctional</td>
</tr>
</tbody>
</table>

4. Functional Non Injective and Non Surjective Relation

<table>
<thead>
<tr>
<th>Functional</th>
<th>Injective</th>
<th>Surjective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perspective from A to B</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Perspective from B to A</td>
<td>Nonfunctional</td>
<td>Nonfunctional</td>
</tr>
</tbody>
</table>
Appendix 2 Nonfunctional Relation

Nonfunctional Relation Cases

For a condition which entity state A dictates the changes and entity state B accepts the changes. *Functional Relation from A to B consist cases similar to below:*

1. When a there is not a correspondent for a member of the dictating entity state’s sub-entity or entities.

<table>
<thead>
<tr>
<th>Functional Perspective from A to B</th>
<th>Injective</th>
<th>Surjective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonfunctional</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- OR

<table>
<thead>
<tr>
<th>Functional Perspective from B to A</th>
<th>Injective</th>
<th>Surjective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonfunctional</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functional Perspective from B to A</th>
<th>Injective</th>
<th>Surjective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>
2. When a there is a “one to many” style multi-response correspondent situation for a member of the dictating entity state’s sub-entity or entities.

-OR

<table>
<thead>
<tr>
<th>Functional</th>
<th>Injective</th>
<th>Surjective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perspective from A to B</td>
<td>Nonfunctional</td>
<td></td>
</tr>
<tr>
<td>Perspective from B to A</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Functional</th>
<th>Injective</th>
<th>Surjective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perspective from A to B</td>
<td>Nonfunctional</td>
<td></td>
</tr>
<tr>
<td>Perspective from B to A</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Appendix 3 Two Static State Conflict Conditions

<table>
<thead>
<tr>
<th></th>
<th>State Nature</th>
<th>Direction</th>
<th>Gen. Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. State A</td>
<td>Static</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>State Nature</th>
<th>Direction</th>
<th>Gen. Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. State A</td>
<td>Static</td>
<td>Accept</td>
<td>No</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>State Nature</th>
<th>Direction</th>
<th>Gen. Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. State A</td>
<td>Static</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Accept</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>State Nature</th>
<th>Direction</th>
<th>Gen. Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. State A</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Accept</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>State Nature</th>
<th>Direction</th>
<th>Gen. Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. State A</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Accept</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>State Nature</th>
<th>Direction</th>
<th>Gen. Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. State A</td>
<td>Static</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>State Nature</th>
<th>Direction</th>
<th>Gen. Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. State A</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>State Nature</th>
<th>Direction</th>
<th>Gen. Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. State A</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>State Nature</td>
<td>Direction</td>
<td>Gen. Ability</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td><strong>E. State A</strong></td>
<td>Static</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>E. State B</strong></td>
<td>Static</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Static</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>E. State A</strong></td>
<td>Static</td>
<td>Accept</td>
<td>No</td>
</tr>
<tr>
<td><strong>E. State B</strong></td>
<td>Static</td>
<td>Accept</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Static</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>E. State A</strong></td>
<td>Static</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>E. State B</strong></td>
<td>Static</td>
<td>Accept</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Appendix 4 Direction of Change Conflict Conditions

<table>
<thead>
<tr>
<th>State Nature</th>
<th>Direction</th>
<th>Gen. Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>E. State A</td>
<td>Dynamic</td>
<td>Dictate</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Accept</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. State A</td>
<td>Dynamic</td>
<td>Dictate</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Accept</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. State A</td>
<td>Dynamic</td>
<td>Dictate</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. State A</td>
<td>Dynamic</td>
<td>Dictate</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. State A</td>
<td>Dynamic</td>
<td>Dictate</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. State A</td>
<td>Dynamic</td>
<td>Dictate</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. State A</td>
<td>Dynamic</td>
<td>Dictate</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. State A</td>
<td>Dynamic</td>
<td>Dictate</td>
</tr>
<tr>
<td>E. State B</td>
<td>Static</td>
<td>Dictate</td>
</tr>
</tbody>
</table>
### Appendix 5 Type 1 Basic Behavior Starting Conditions

<table>
<thead>
<tr>
<th>Functional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Static</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity State A</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity State A</td>
<td>Static</td>
<td>Accept</td>
<td>No</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td>B to A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td>B to A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>No</td>
</tr>
<tr>
<td>B to A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>B to A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static</td>
<td>Accept</td>
<td>No</td>
</tr>
<tr>
<td>B to A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix 6 Type 2 Basic Behavior Starting Conditions

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Static</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>- A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>- A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>- A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>No</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>- A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Static</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>- A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Static</td>
<td>Accept</td>
<td>No</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>- A to B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td>- B to A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static</td>
<td>Dictate</td>
<td>No</td>
</tr>
<tr>
<td>- B to A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Dictate</td>
<td>Yes</td>
</tr>
<tr>
<td>- B to A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nonfunctional</td>
<td>State Nature</td>
<td>Direction</td>
<td>Generation Ability</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>-----------</td>
<td>--------------------</td>
</tr>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Dynamic</td>
<td>Accept</td>
<td>No</td>
</tr>
</tbody>
</table>

- B to A

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static</td>
<td>Accept</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- B to A

<table>
<thead>
<tr>
<th>Nonfunctional</th>
<th>State Nature</th>
<th>Direction</th>
<th>Generation Ability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity State A</td>
<td>Dynamic</td>
<td>Accept</td>
<td>Yes</td>
</tr>
<tr>
<td>Entity State B</td>
<td>Static</td>
<td>Accept</td>
<td>No</td>
</tr>
</tbody>
</table>

- B to A