Karlsruhe Institute of Technology

Test, Validation, Analysis

Comparing and Translating Notions of Change for Delta-Oriented Variability

Bachelor's Thesis
of

Darius Seiter

at the Department of Informatics
Institute of Information Security and Dependability
Test, Validation and Analysis (TVA)

Reviewer:	Prof. Dr.-Ing. Ina Schaefer
Second Reviewer:	Prof. Dr.Ralf Reussner
Advisors:	M.Sc. Dirk Neumann
	M.Sc. Tobias Pett

Completion period: 01. March 2025 – 01. July 2025

Zusammenfassung

Im Software-Engineering ist eine effektive Darstellung von Änderungen nützlich, um die Variabilität in Software-Produktlinien zu verwalten. In dieser These werden zwei Änderungsmodelle bewertet: Das von DeltaEcore, das domänspezifische Deltadialekte innerhalb einer Deltasprache verwendet, und das von Vitruv, das ein dialektfreies, atomares Änderungsmodell definiert. Der Vergleich beinhaltet die Bewertung der Ausdruckskraft und der Mächtigkeit der einzelnen Änderungsmodelle. Um diesen Vergleich zu erleichtern, wird ein Übersetzer implementiert, der Änderungen zwischen den beiden Änderungsmodellen konvertiert. Eine Fallstudie, die auf dem Body Comfort System (BCS) basiert, zeigt die Anwendung beider Ansätze und hilft bei der Bewertung des Ausmaßes des Informationsverlustes während der Übersetzung. Die Ergebnisse zeigen, dass beide Änderungsmodelle zwar in ihrer Mächtigkeit, Änderungen zu liefern, ähnlich sind, DeltaEcore jedoch durch die Anpassung und Komposition von Dialekten eine deutlich größere Ausdruckskraft bietet. Diese Ausdruckskraft benötigt jedoch die Erstellung eines Delta-Dialekts, der für jede Domäne benötigt wird. Die Ergebnisse deuten darauf hin, dass die Wahl zwischen den beiden Änderungsmodellen von dem spezifischen Anwendungsfall abhängt, in dem die Verwendung eines Begriffs der Veränderung erforderlich ist. Vitruv bietet Einfachheit und Allgemeinheit, während DeltaEcore ausdrucksstarke, domänenspezifische Änderungen ermöglicht. Übersetzungen zwischen den beiden Begriffen können mit einem nicht permanenten Informationsverlust durchgeführt werden, was Interoperabilität der Änderungsmodelle ermöglicht.

Abstract

In software engineering, representing change effectively is useful for managing variability in software product lines. This thesis evaluates two notions of change: the notion of change of DeltaEcore, which uses domain-specific delta dialects within a delta language, and the notion of change of Vitruv, which offers a dialect-free, atomic change model. The comparison involves evaluating the expressiveness and capability of each notion of change. To facilitate this comparison, a translator is implemented to convert changes between the two notions of change. A case study based on the Body Comfort System (BCS) demonstrates the application of both approaches and evaluates the extent of information loss during translation. The results show that while both notions are similar in their capability for providing change, DeltaEcore offers significantly greater expressiveness through dialect customization and composition. However, this expressiveness comes at the cost of creating a delta dialect, which is needed per domain. The findings suggest that the choice between the two notions of change depends on the specific use case in which the use of a notion of change is needed. Vitruv offers simplicity and generality, while DeltaEcore allows for expressive, domain-specific change. Translations between the two can be performed with no found permanent information loss, enabling tool interoperability in practice.

Contents

List of Figures

List of Tables

1. Introduction

Modern software systems use variability, allowing for customized products to meet diverse customer needs. Delta-oriented programming (DOP) is one way to achieve this, particularly in Software Product Lines (SPLs). DOP works by having a core module that provides a base, a functional product, and then delta modules that define changes applied to this core, creating new products [Sch+10].

A framework that implements delta-oriented variability is DeltaEcore [SSA14]. It uses a common base delta language to define changes for Ecore-based models. This common-based delta language does not require any knowledge of the source meta-model and provides a context-independent way of defining deltas.

Another approach to representing change is found in the Vitruv tool, which focuses on view-based software development [Kla+21]. Vitruvius introduces a Virtual Single Underlying Model (V-SUM), which maintains consistency between multiple domains by establishing explicit rules that preserve consistency between different models even after they've been modified through views. While Vitruvius isn't directly used for developing SPLs, it offers a method for representing change, which is then used to specify how changes propagate between models to maintain their consistency.

Understanding and comparing how DeltaEcore and Vitruvius represent change is needed to give an answer to the extent of the expressiveness and the capability of both. Despite the usefulness of comparing expressiveness and capability, a comparison of their respective notions of change has not been extensively done before. Identifying their strengths and limitations could pave the way for enhanced interoperability and potential extensions to existing frameworks.

This thesis aims to compare the notions of change used in DeltaEcore and Vitruvius, highlighting their advantages and disadvantages in regard to their expressiveness and capability. This comparison could inform potential improvements or extensions to one of these notions of change or other notions of change. To assess their expressiveness and capabilities, both will be evaluated based on their overall usage, giving a useful overview of both. Additionally, a case study will be developed, consisting of instances of both notions of change for direct comparison. To further support this evaluation and improve interoperability, a translator will be implemented to convert between these two notions of change. Ultimately, the thesis will present this translator and an evaluation of the expressiveness of both notions of change, along with the case study comparing them.

Goal of This Thesis

The primary goal of this thesis is to compare the notions of change employed by DeltaEcore and Vitruv, assessing their expressiveness and practical implications. To achieve this, the research will address the following key questions:

- **RQ1: How do the notions of change in Vitruv and DeltaEcore compare concerning their expressiveness and capability?**
  This question aims to identify differences in how each framework handles modifications. Capability refers to the ability of the notion of change to produce all necessary modifications, such as operations for creating elements or managing element changes. Expressiveness determines the information that can be extracted from a change, including the intent behind the modification. With this research question answered, we can give a distinction between when to use Vitruv and when to use DeltaEcore.
- **RQ2: What potential extensions could be beneficial in refining a notion of change?**
  While evaluating both notions of change, missing elements of both notions of change or elements not present in one notion of change but in the other could make themselves clear. These missing elements could be in the form of missing expressiveness or capability, and their addition to the notion of change could make an improvement. With the comparison of the two notions of change, possible extensions that could be added to any notion of change can be assessed.
- **RQ3: To what extent is information lost between translations?**
  When translating a notion of change, information in the form of expressiveness or capability could be present. To make use of a translator, it is important to know the possible extent of degradation in the form of information loss that a change could suffer.

Structuring

To evaluate both notions of change regarding the listed research questions, this thesis will first provide an overview of all necessary background information in chapter 2. Then, the overall research approach, a notion of change overview, and the case study will be explained in chapter 3. Following this, in chapter 4, the implemented translator will be described. The main part of the thesis will be a description of the evaluation and its results, answering the given research questions in chapter 5. Finally, a positioning of the thesis concerning other research will be given in chapter 6, and in chapter 7 an outlook on the evaluated result will be presented.

2. Basics

This chapter provides needed background information for this thesis. Introduced will be terminologies and technological frameworks that are prerequisites to understanding the comparative evaluation of notions of change. First, information is provided for the usage of models in software, and a definition is given for notions of change and variability. Then both notions of change will be introduced: the notion of change of DeltaEcore and the notion of change of Vitruv. Lastly, an important aspect upon which the case study builds will be presented.

2.1 Model-Driven Development

Model-driven development (MDD) is a software engineering approach that involves applying models, meta-models, and other model technologies to abstract software development, thereby simplifying and formalizing the software development process [HT06].

The main part of MDD is the use of models. A model is an abstraction of a software product, for example, the code and class structure of a program. Models in MDD often use *metamodels*, which define the abstract syntax for a model. A model's abstract representation of a system is defined in a modeling language, a language that consists of concrete syntax, semantics, and abstract syntax [CV11]. The concrete syntax represents the model, either in a visual form, as diagrams, or in text form. Semantics provide additional information with which the meaning of the abstract syntax can be attained. The abstract syntax is represented by a meta-model and describes the concepts and relationships used to create models. While a modeling language defines models, meta-models are defined by a meta-modeling language, which also has a metamodel called a meta-meta-model. The meta-modeling language is used to describe a new modeling language. The connections between the model, meta-model, and meta-meta-model are shown in the figure 2.1.

A way to implement models for MDD is the *Eclipse Modeling Framework (EMF)*, which provides a modeling framework and code generation facility for building tools and other applications based on a structured data model [CV11]. The EMF provides a full working environment for modeling and code generation. It also provides its

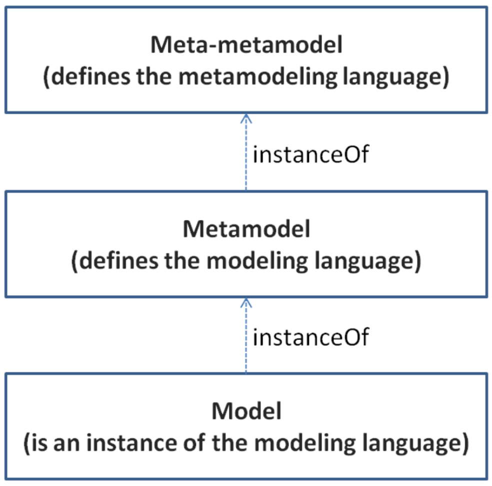
Meta-metamodel
(defines the metamodeling language)
instanceOf
Metamodel
(defines the modeling language)
instanceOf
Model
(is an instance of the modeling language)

Figure 2.1: **Connections between the model, meta-model, and meta-meta-model [CV11]:** The figure shows the connections of the models by the instanceOf relation. Each model or meta-model can be seen as an instance of either a metamodel or meta-meta-model, as models or meta-models use the defined abstract syntax of their corresponding meta-model or meta-meta-model.

meta-meta-model, named Ecore, a modeling language for meta-models, used in the working environment.

MDD is used in *Software Product Lines (SPL)*, Northrop defines an SPL as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [Nor02].

A way to implement SPLs is delta-oriented programming (DOP). DOP consists of a core module and a set of delta modules [Sch+10]. The core module contains a set of implementation artifacts that provide a valid product, and the delta modules define changes to be applied to the core module to derive other products. This change application is a form of *Model transformation*, the manual or automated systematic modification of a model [HT06].

While DOP uses a specialized version of a model transformation, for different model-changing approaches, a more general definition of model transformation is needed. In general, *Model Transformations*, a source model is converted into a target model according to a set of transformation rules [CV11]. The rules for the transformation are given by a *Model Transformation Language*, a language that maps elements of a source model to elements of the target model. Both models conform to their meta-model, and while the source model is normally not modified, the target model is not only modified it can also be fully created by the transformation. One transformation language can be found in the Eclipse Epsilon framework, the *Epsilon Transformation Language (ETL)*. The Eclipse Epsilon framework has a family of different languages to manage models and meta-models, in which every language can achieve a specific goal [Zol+20]. The ETL language is built atop the core language, the *Epsilon Object Language (EOL)*, which has the goal of providing support for querying and modifying models conforming to diverse modeling technologies.

Model transformation can be used in variability modeling, which, like DOP, uses changes to make modifications to a model. The Variability in variability modeling is an important aspect of a model modification, as it determines the flexibility, reusability, and expressiveness of changes [SH11].

2.2 DeltaEcore and Vitruv

This thesis will look at two ways to implement a notion of change, DeltaEcore and the change provided by Vitruv.

The notion of change in software engineering is not a monolithic concept with a single, universally accepted definition. Instead, it is a multifaceted idea, understood and used differently across various areas, reflecting the diverse challenges and objectives within the field. As such, the notion of change in this thesis can be understood as the capability and expressiveness of a model transformation language, such as DeltaEcore, and the change of Vitruv. The exact meaning of capability and expressiveness will be further defined in Chapter 3.

DeltaEcore is a research tool for DOP, which provides a model-based framework to define deltas in DOP [SSA14]. DeltaEcore contains a delta language created by combining the common base delta language with a delta dialect specific to the respective source language [SSA14].

As shown in Figure 2.2, the common base delta language operates on the level of the meta-meta model and provides a base for the to-be-created delta language. With the base provided for each delta language, a delta dialect can be created for a chosen source language. The delta language is then created by combining the common base delta language with the respective delta dialect for the source language.

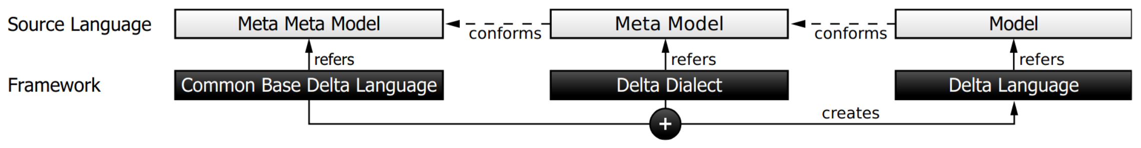

Figure 2.2: **Architecture of the delta language generation framework [SSA14]:** The figure shows how the delta language is created and what each part references. The plus relation describes how the common base delta language and the delta dialect are combined to form the delta language. The refers relations show what parts of the delta language and the delta language as a whole reference. The conform relation can be understood like the instanceof relation of figure 2.1.

Vitruv is a framework for view-based software development and uses a variety of models to describe a system, and provides rules to ensure consistency in every model. Vitruv also provides views, which present the only way of model modification [Kla+21]. Included in Vitruv is a notion of change that can be applied to any ecore-based metamodel instance without the need for any implementation of a dialect for a source language. The notion of change consists of atomic changes, changes that can not be divided into smaller changes. The atomic changes involve existence, attribute, and reference changes, with which change can be applied to any given model instance.

2.3 The Body Comfort System Case Study

To assess both Vitruv and DeltaEcore, we use the Body Comfort System Case Study [Lit+13]. This case study presents a complete description of a delta-oriented SPL test model. The model represents a Body Comfort System of the automotive domain and is comprised of the component interface specifications and their variants. It also provides the interaction test scenarios employing message sequence charts and describes a state machine for the component's current states. The BCS provides a feature model, which shows all possible model variations created by a possible model transformation. Figure 2.3 shows this feature model, with the addition of making less relevant parts slightly transparent. These transparent parts are not needed in this thesis, as the other elements of the feature model are sufficient. The mandatory parts of the feature model represent the core model to which change can be applied to gain a new product. These new products then include optional or alternate elements.

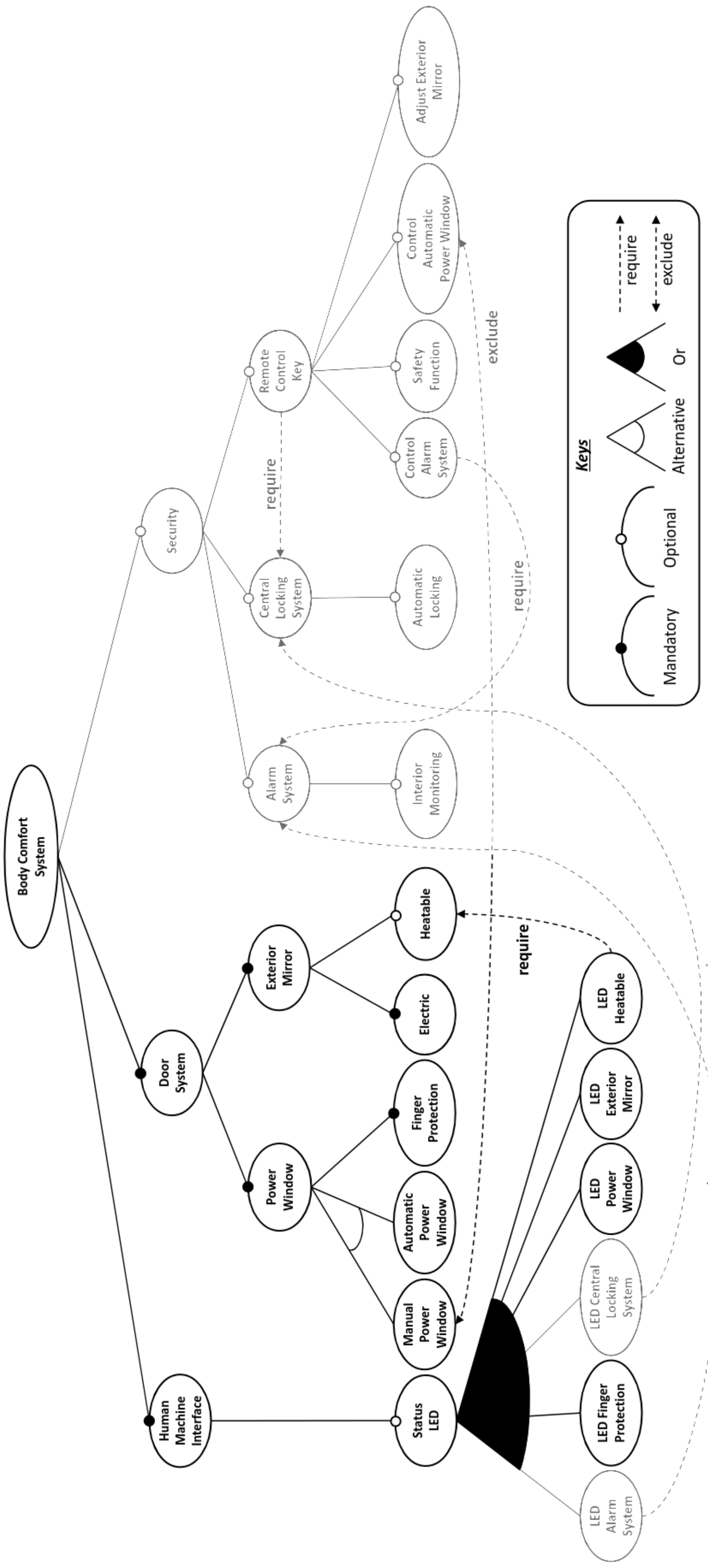

Figure 2.3: **Feature Model of the Body Comfort System [Lit+13].**

3. Design

Following the exposition of fundamental concepts and frameworks in Chapter 2, this chapter presents the research approach devised to compare and evaluate both notions of change of DeltaEcore and the change of Vitruv. First, the overall research approach will be explained, beginning by defining expressiveness, capability, and information loss. These definitions are selected to capture the aspects of the notion of change and the effects of translating them. Also, this chapter will detail the overall research strategy. This strategy encompasses an initial overview of the capabilities and expressiveness available for both notions of change, a comparison using examples from the BCS to highlight practical differences, and a more extensive case study employing an implemented translator.

To assess the given research questions, it is needed to define what is meant by expressiveness, capability, and information loss. Both expressiveness and capability are important factors in the variability of each notion of change, and information loss is an important factor for translations, as the information loss can be a hindrance to the usage of a translator.

Expressiveness determines which information can be extracted from each change. The more expressiveness, the more information can be assessed about the intent of the change and the impact of the change. Types of information can include accessed elements of a product and the changed information in the accessed element, but expressiveness can also include information about the to-be-changed element information or describe the difference between a replacement or existence creation. The expressiveness also includes the ease of readability. While different notions of change can have similar expressiveness, their expressiveness can differ in their readability, which is also an important aspect of the assessment of a notion of change.

Capability describes the given notion of change's capability to produce all needed changes that would be applied to an instance. An example of capability would be operations for element creation or handling of multi-valued elements. Operations that build upon other operations can be seen as specializations, and do not provide directly more capability, but expressiveness. The capability for these operations can only be seen as the extent to which a notion of change allows for specialized operations.

An example of added expressiveness through specialized operations would be to provide an operation to not only insert an element into a container but also be capable of a function to add an element at the end of the container. While an insertion would be sufficient, additional functions extend the expressiveness of a notion of change, as the intent of a change is more directly represented.

Information Loss is presented as the regression in expressiveness or capability, in a translation from one notion of change into another notion of change. For expressiveness, information loss means the loss of intent of a change, the loss of information needed for a specialized operation, or any additional information. An example could be the loss of information on an element value before the change, which would add expressiveness to the overall change tracking or reverting. The loss of information for capability would be the inability to translate an operation entirely, meaning no less specialized operation can be used as a substitution.

With the definitions of expressiveness, capability, and information loss, the evaluation is then structured as follows. First, an overview of capability and expressiveness will be assessed. This overview looks at the provided operations available for both notions of changes and assesses their usage and structure. After this, a small BCS example is used to compare both notions of change. Both the overview and usage comparison should provide the information needed for RQ1 and RQ2. Also, with the overview and usage example, it can be determined with which criteria the information loss can be measured in the case study, as both notions of change structure and usage are evaluated.

The case study will use the implemented translator to determine the information loss for RQ3 and provide additional information for RQ1 and RQ2. By translating an instance of a notion of change into another notion of change, the newly translated instance can be evaluated for any change in expressiveness and information loss. The possible assessment of capability should be unlikely, as the evaluation of the overview and usage example should provide all the needed information about the capabilities, as every operation was looked at.

The usage example and case study will be made with the BCS, which provides a complete description of a delta-oriented SPL model, with which all operations of both notions of change can be applied. Further, as the BCS is specifically made for model-based testing [Lit+13], it provides deltas to modify the model and models for each variation. With this, an extensive reference for changes and their resulting model variation is given, which can be used to provide meaningful changes to be translated. As DeltaEcore makes use of dialects to represent change, this thesis needs two dialects in order to use DeltaEcore with the BCS. One dialect for the components and one dialect for the state machine.

4. Implementation

This chapter details the implementation of the translator developed for this thesis. The primary function of this tool is to convert change between two notions of change, used in the evaluation of the case study. Beyond its core function, the translator was designed to be extendable and straightforward, also allowing for ease of extension or modification of translation rules.

The following sections will explore the translator's architecture, which is a Java Maven project utilizing the Eclipse Epsilon framework for model-to-model transformation, structured with the Model-View-Controller (MVC) design pattern. Also explained will be the core algorithms developed to handle the translation process. This includes the essential processing steps, specifically XML splitting and merging, required to manage the composition of deltas of DeltaEcore and the changes of Vitruv. Finally, the chapter provides practical instructions on how to use the translator's current implementation as a command-line tool and discusses the current limitations of the implementation.

4.1 Goal of Translator

The primary goal of the translator, implemented for this thesis, is to enable a detailed comparative evaluation of the notions of change of DeltaEcore and Vitruv. To allow a comparative evaluation, the translator is designed to convert change between DeltaEcore and Vitruv, and serves as a component for the case study conducted in this thesis. A secondary goal in developing this translator has been to ensure that the translation process itself is as straightforward as possible and that the system is inherently extensible. This objective is pursued by designing the translator to support the creation and modification of new translation rules. The intention is to produce a tool that is not only effective for the specific comparative evaluation within this thesis but is also accessible and maintainable for potential future adaptations or extensions in related research areas or other work with model transformations.

4.2 Translator Architecture

To give a better understanding of the translator, a description of the classes used in the implementation of the translator is given in this section. In addition, a class diagram is presented with figure 4.1, showing the mentioned implementations of the classes.

The translator is a Java Maven project that makes use of the Eclipse Epsilon framework to achieve a model-to-model transformation as a form of translation. The project provides all needed configurations for the build of an executable, which presents the main way of usage of the translator. The executable should be usable on any system that can run Java 8.

The overall structure of the translator follows the model-view-controller design pattern, with the thin controller, fat model approach. This means keeping controllers lean by offloading as much business logic as possible to the models, to ease the portability of the model to any new controller and view.

The controller's most important operations are the setup of the translation, meaning loading all needed data and setting up the right translation model. The other important operation is starting the translation and XML editing process. Currently implemented is the console controller, which only needs to implement the setup and starting operations.

The currently implemented console view makes use of two observers, one to react to any messages while translating and one to react to any events occurring while handling any models. The observers are handled by a dedicated class of the model used in a translation.

The model includes an abstract class giving all core features for any translation type. This includes the core translation process, the overall data structure, and the handling of meta-model loading. Building on this, three types of translation classes exist. One for any simple translation, not using any special processes, and two for either XML merging or splitting while translating.

The XML processing is done by one class in the model. It currently provides XML merging and splitting features, both specifically made for changes in the XMI format.

For holding translation models or properties, the translator uses a class, which uses the Java properties functions, and another class that holds a model. As both classes only needed to be set up once, they are both implemented to be immutable, meaning model file-paths and all properties can not be changed after creation.

4.3 Design Rationality

This section details the reasoning behind the architectural and implementation choices made. It shows how the use of properties files for configuration, an immutable pattern for model and properties integrity, and a tailored observer pattern for messaging contribute to a robust and predictable tool.

The goal of the translator is achieved mainly by providing a straightforward way of using the model-to-model transformation of Eclipse Epsilon and by giving rules a simple way of modifications and extensions of the translation rules. Because of this, the use of properties files made sense. They provide a simple way of holding all the needed information for a translation, easing the use of different models and rules, as their usage is defined in one editable properties file.

The model class employs an immutable pattern to ensure a less error-prone handling of any model, and that once a model is created, it always points to the same file paths. The change of any path could abruptly change the translation result and should not be needed, as every model should only contain one specific model file with corresponding meta-model files. Any file handling should occur before the creation of the models, as the models are only used in the translation and are created right before the translation occurs. Any other properties of the EMF can be changed as they need to be set per translation. Properties files should be fully immutable, as a translator user will set these before starting the translator, and any changes made to the translation properties would not be anticipated by a user.

While the observer pattern of model events is implemented in a common way, the observer for messages while translating was limited by the implementation of Eclipse Epsilon. As Eclipse Epsilon provides only one interface for messages while translating, the observer needs to be adapted to it, and as a result, it does not follow the standard observer pattern. While custom operations could be used in an ETL rule file, the use would not be anticipated by a user of the translator. With the usage of operations, following the Eclipse Epsilon documentation, ETL rule files can be created for the translator by only using the documentation of Eclipse Epsilon.

4.4 Core Algorithms

As the translation of changes is an important part of this thesis, the core algorithms used for translation are described in this section, also explaining additional processes needed, which are not provided by Eclipse Epsilon.

As DeltaEcore and the change of Vitruv differ in aspect, which the ETL language can not easily handle, mainly the composition of changes, additional XML operations after the translation are needed. The translator makes use of two operations to handle composition in each translation.

The splitXMI operation is used on any DeltaEcore to Vitruv translation. It starts by reading the translated change, determining the amount of new resulting changes, and identifying any information that would be needed in every resulting new change file. It then creates the changes file and fills it with the information that any changed file would need and its specific contents.

The mergeXMI operation functions in reverse, as it creates a new file in which every change is copied. Information that any change file would need is not copied and is only kept at the start of the file, as any valid XMI file would have.

The core algorithm follows exactly the steps needed for any Eclipse Epsilon translation, following the steps provided by the documentation for the ETL language leads to the following steps.

The translation follows these simple steps.

1. Load the next properties file
2. Prepare all models and meta-models
3. Translate the next change

4. Go to 2. If other changes still need translation.

Currently, building on these steps are the two translation types for merging or splitting changes. The merging translation merges all changes as explained before the translation occurs, leading to a single merged translation. Splitting translation first, follow the usual steps, and add one step after each change is translated, which is the splitXMI method used to split the just-translated change before translating the next one.

4.5 Translator Usage

The translator is used by executing its jar with optional command-line arguments for the property files. The property file is a configuration file following the Java properties style. The properties files can be provided by command-line arguments, pointing to all wanted properties files, which then will be used in succession. If no argument was given, the default path will be used. This path consists of the properties file "Default.properties" near the executable jar. To provide a valid properties file, a text file with the file extension "properties", the file needs all the needed information about the translation, including all needed file paths and translation settings. Table 4.1 shows all needed properties for a property file.

While translating, the translator can print out warnings if a translation would not be possible with a given instance. These warnings are determined by the transformation rules and do not influence whether an instance is modified or not, meaning that even with warnings an instance will be modified if so defined in the properties file. After translation, all translation files and instance modifications are created. Additional XML changes can be applied to a change if so defined in the translation type. These new modified changes are stored in a subfolder of the target folder.

4.6 Implemented Translation Rules

For the case study, the translation rules for the delta component dialect and the change of Vitruv were implemented. Table 4.6 shows how the translation between the dialect and the change of Vitruv is mapped. To simplify the table, multiple operations with the same purpose are combined. That means, for example, that all remove operations of the dialect are combined into one.

The translation rules are split into a file for complex operations, a file for model instance handling, and a mapping file. The complex operations file helps to keep the mapping operations simple and easy to define, which helps to create rules for every dialect operation. The mapping maps all translation operations and calls the needed operations of the other files. The mapping file is the file noted in the properties file. In the mapping file, rules call a fitting operation from the complex operations file or handle simple operations themself. Both operations handle the creation of the translated changes by using an empty or given instance of a model. By using the instance of a model, missing information, mostly in the case of the dialect, can be easily and comprehensively gained for a valid translation.

The operations for add and insert of both dialects are special, as an add operation will only be mapped from a Vitruv InsertEReference if a valid instance is provided with which a possible add operation would be right for the current insert position.

Property	Type	Usage
TranslationType	Class name	Defines the translation type, currently only implemented are "BasicTranslation", "SplitTranslation" and "MergeTranslation"
ModifyInstance	Boolean	Instance to which the change is applied
MetaModelA	Relative file path	change meta-model
MetaModelB	Relative file path	change meta-model
MetaModelInstance	Relative file path	Meta-Model of instance to which the change is applied to
ModelInstance	Relative file path	Instance to which the change is applied, can be an empty file
Changes	List of relative file paths	Changes of the meta-model A, can have an empty value
ChangesFolder	Relative folder path	Like Changes, but all XML files in the folder and sub-folders are used for translation
transformationDefinition	Relative file path	Translation rules for translation from A to B
TargetFolder	Relative folder path	Folder in which all translated changes are put

Table 4.1: **Needed properties for a properties file:** The property column of the table shows all variables that can be defined in a properties file. The type column lists the type of each value for the variables, and the usage column describes how the translator will use these variables.

The remove operations of the dialect do not use any indexes to determine the object to be deleted, but still can be used in Vitruv, which needs an index for delete operations, as they determine the object to be deleted with the object ID. With the use of the ID, any change translated from the dialect to Vitruv can still be used, even when no index is provided. To ease the implementation of the translation rules, the index value of Vitruv changes in delete operations is used to hold the ID. The ID could still be determined by other means, but would increase the complexity of the translation rules.

For the second delta dialect, used for the state machine, only an example of some translation rules is given. As the state machine model has errors involving all state triggers, constraints, and behaviors, the dialect can not be properly used. All valid operations are translated exactly like the operations of the other dialect, which is shown by the examples.

4.7 Limitations

Some limitations are present in the current implementation of the translator. One of these limitations is the missing user interactions. While with user interaction, a

Vitruv	**DeltaEcore dialect**
ReplaceSingleValuedEAttribute	Modify Attribute Operations
InsertEReference	Add or Insert Operation
ReplaceSingleValuedEReference	Set Operation
RemoveEReference	remove Operation
RemoveEReference	detach Operation
CreateEObject	*No mapping possible*
No mapping possible	Unset Operations

Table 4.2: **DeltaEcore dialect and Vitruv change mapping:** The table's columns show the operations of the change of Vitruv and the DeltaEcore dialect. Each row shows which operation of one notion of change could be translated into the operation of the other notion of change. The translation for the InsertEReference operation of the change of Vitruv can be translated to either an add or insert operation. The given index of the inserted element is checked; if it's outside of a valid index value, the add operation is used. Operations that can not be translated to the other notion of change are marked with *No mapping possible.*

translation could be easily made with far more robustness, in terms of information loss, human interaction is not needed in the evaluation of both notions of change, as this would make measuring information loss not feasible. Also, the current implementation as a console application makes human interaction, especially with larger changes, not only cumbersome and time-consuming but also error-prone, as many repetitions and difficulties of oversight occur.

The current console applications implementation also makes any change visualization or instance visualization difficult. Because of this no before and after overview of changes and the instance are provided and must be done by use of secondary tools.

While the translator can work with any given change of an Ecore model and also use any given ETL rule file, the addition of new translation types requires the implementation of at least one new translation class. This limitation is the result of the possible unique new notion of change, with which the translator could be used. What kind of new translation type is needed by another notion of change can only be known when examining the notions of change. In addition, the translation between notions of change can add additional unique demand for the translation process.

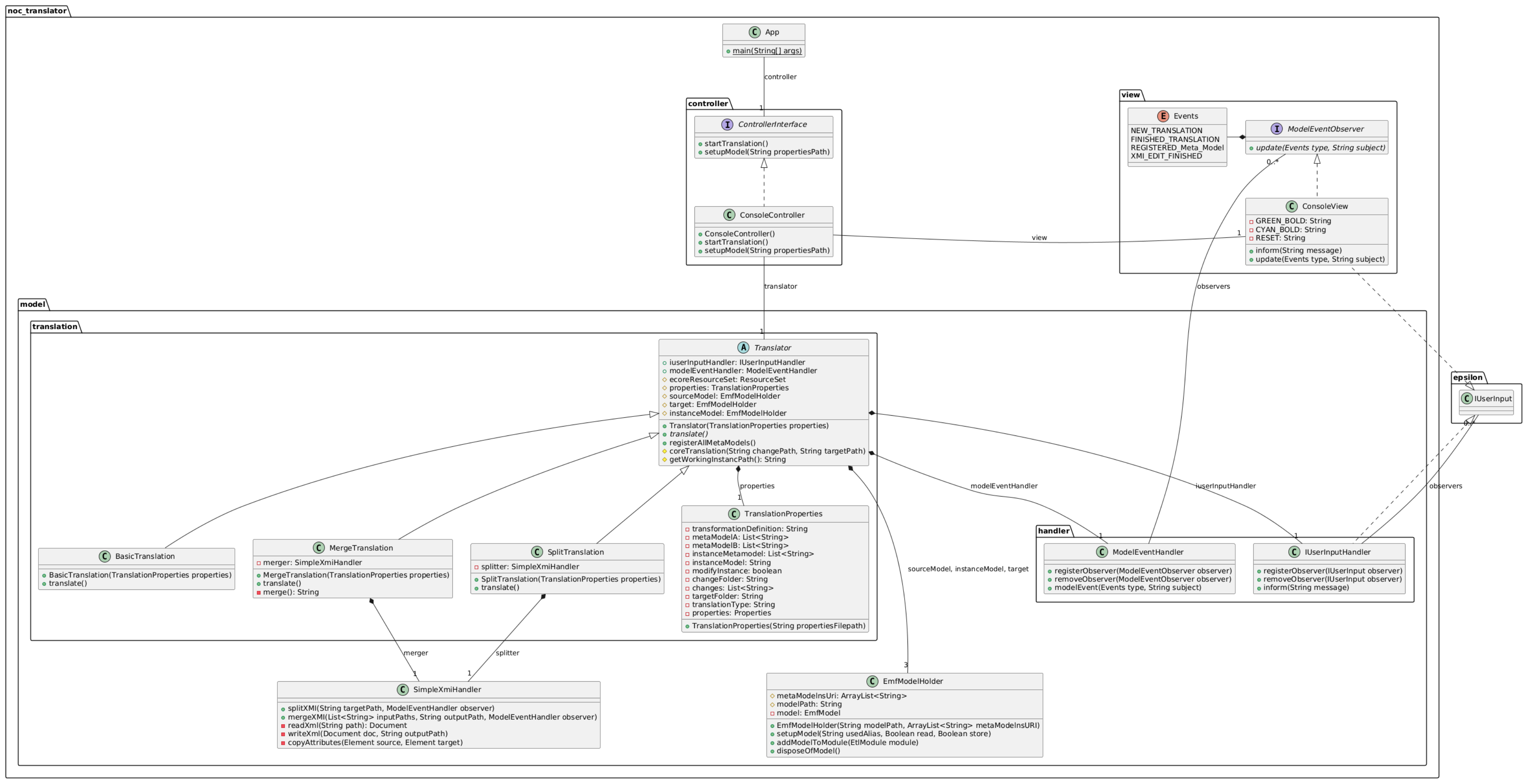
noc_translator
App
main(String[] args)
controller
controller
ControllerInterface
startTranslation()
setupModel(String propertiesPath)
ConsoleController
ConsoleController()
startTranslation()
setupModel(String propertiesPath)
view
view
Events
NEW_TRANSLATION
FINISHED_TRANSLATION
REGISTERED_Meta_Model
XMI_EDIT_FINISHED
ModelEventObserver
update(Events type, String subject)
ConsoleView
GREEN_BOLD: String
CYAN_BOLD: String
RESET: String
inform(String message)
update(Events type, String subject)
observers
translator
model
translation
Translator
iuserInputHandler: IUserInputHandler
modelEventHandler: ModelEventHandler
ecoreResourceSet: ResourceSet
properties: TranslationProperties
sourceModel: EmfModelHolder
target: EmfModelHolder
instanceModel: EmfModelHolder
Translator(TranslationProperties properties)
translate()
registerAllMetaModels()
coreTranslation(String changePath, String targetPath)
getWorkingInstancPath(): String
properties
TranslationProperties
transformationDefinition: String
metaModelA: List<String>
metaModelB: List<String>
instanceMetamodel: List<String>
instanceModel: String
modifyInstance: boolean
changeFolder: String
changes: List<String>
targetFolder: String
translationType: String
properties: Properties
TranslationProperties(String propertiesFilepath)
BasicTranslation
BasicTranslation(TranslationProperties properties)
translate()
MergeTranslation
merger: SimpleXmiHandler
MergeTranslation(TranslationProperties properties)
translate()
merge(): String
SplitTranslation
splitter: SimpleXmiHandler
SplitTranslation(TranslationProperties properties)
translate()
merger
splitter
SimpleXmiHandler
splitXMI(String targetPath, ModelEventHandler observer)
mergeXMI(List<String> inputPaths, String outputPath, ModelEventHandler observer)
readXml(String path): Document
writeXml(Document doc, String outputPath)
copyAttributes(Element source, Element target)
sourceModel, instanceModel, target
EmfModelHolder
metaModelnsUri: ArrayList<String>
modelPath: String
model: EmfModel
EmfModelHolder(String modelPath, ArrayList<String> metaModelnsURI)
setupModel(String usedAlias, Boolean read, Boolean store)
addModelToModule(EtlModule module)
disposeOfModel()
modelEventHandler
iuserInputHandler
handler
ModelEventHandler
registerObserver(ModelEventObserver observer)
removeObserver(ModelEventObserver observer)
modelEvent(Events type, String subject)
IUserInputHandler
registerObserver(IUserInput observer)
removeObserver(IUserInput observer)
inform(String message)
observers
epsilon
IUserInput

Figure 4.1: **Class diagram of the translator:** The figure shows the class diagram of the translator, not including any setter or getter. Also, the main package was simplified for a more readable figure. All other packages are named right.

5. Evaluation

This chapter presents a comprehensive evaluation of Vitruv's notion of change and DeltaEcore's notion of change. The evaluation centers on a comparison of their respective capabilities and expressiveness. It further looks at the practical consequences of their differences, particularly the information loss that can occur when translating between the two notions of change.

5.1 Evaluation Structure and Setup

The evaluation follows the outline from chapter 3, meaning first an overview of both notions of change will be made in section 5.2.

For this, the operations of both notions of change will be looked at and compared. This will give an understanding of both notions of changes in overall design and differences in expressiveness and capability. As the notion of change of DeltaEcore uses dialects, the implementation of dialects for the use with models of the BCS will be looked at in section 5.2.3. To then further compare the differences of both notions of change, an example of a change in the BCS will be used in section 5.2.4. This example shows how both notions of change would handle a common use case and further helps evaluate differences in capability and expressiveness.

With an overview of both notions of change, the case study can further examine expressiveness and capability, but also evaluate the amount of information loss that a translation could cause. First, an assumption of the result will be given in 5.3.1, and the used metrics will be explained in 5.3.2. To help understand the case study, its method is then explained in 5.3.3. Then the result will be shown in 5.3.4.

Having gathered all results of the overview and case study, a conclusion can be achieved by interpreting and looking at possible implications in section 5.4.

The overview uses the documentation for the Vitruv tool and the paper [SSA14], but also the models of both notions of change to compare both and create an overview of both notions of change capabilities and expressiveness.

For the case study, the translator is used, which requires an environment that can run Java. The provided changes and the instance of the BCS were used with the translator. The changes and model instance files, original or translated, are evaluated by hand, supported by the KDE Advanced Text Editor functions, for counting change operations or highlighting differences.

5.2 Capability and Expressiveness Overview

Vitruv and DeltaEcore slightly differ in expressiveness in their use of a specific dialect for a use case. Vitruv does not need a dialect at all and can be used for any use case. DeltaEcore needs a dialect, and for the delta creation with DeltaEcore, we use a dialect specifically made for changes in the BCS. Without a dialect, DeltaEcore can not reference an instance of a model and apply changes to it.

The table 5.1 shows the basic types of changes and their implementation in Vitruv and DeltaEcore. In the case of DeltaEcore, a dialect would use these DeltaOperations to create one of these change types.

Change	**Vitruv**	**DeltaEcore**
Existence Changes: Creation	Create EObject Operation	*Not directly able to*
Existence Changes: Deletion	Delete EObject Operation	*Not directly able to*
Attribute Changes: Single-valued	Replace Single Valued EAttribute Operation	Modify Delta Operation
Attribute Changes: Multi-valued	Insert/Removal EAttribute Operation	Modify Delta Operation
Reference Changes: Single-valued	Replace Single Valued EReference Operation	Set/Unset Operation
Reference Changes: Multi-valued	Insert/Removal EReference Operation	Add/Insert/Remove Delta Operations
Custom Changes	*Not able to*	Custom Delta Operations

Table 5.1: **Capability overview:** The Change column shows the type of change in which the capabilities are divided. The Vitruv and DeltaEcore columns show the operations that are available to produce a type of change.

5.2.1 Capability of Operations

Set/Unset Delta Operations are used to change the value of a single-valued reference. The Set Delta Operation assigns a new value to a given single-valued reference and unset changes the value to the default value of a given single-valued reference.

Add/Insert/Remove Delta Operations are used to change the values of multi-valued references. Add delta operations append a given element to the set of values, while

Remove erases a given element from the set of values without deleting it. The insert operation can be used to put an element in a certain position.

Modify Delta Operations are used to change the values of an attribute. It is used for both single-valued and multi-valued attributes. The handling of containers is open to the delta interpretation. For example, that means to append an element to a container instead of overwriting the container with a new one under a given interpretation.

Custom Delta Operations are used to declare operations that can specify semantics that could not be expressed using the standard delta operations.

Replace Single Valued EAttribute Operations are used to change the values of a given single-valued attribute. The change also includes the to-be-replaced value of the attribute.

Create/Delete EObject Operations are used to create new objects of a given class and add them to a parent. The creation also involves setting the ID, but any other references or attributes needed must be set with more operations.

Insert/Removal EAttribute Operations are used to change or remove multi-valued attributes.

Replace Single Valued EReference Operations are used to change the values of a given single-valued reference. The change also includes the value of the reference to be replaced.

Insert/Removal EReference Operations are used to change the values of a given single-valued reference.

5.2.2 Comparison of Expressiveness and Capability

While not focusing on the DeltaEcore dialect, with both notions of change examined, the following differences can be noted.

Existence changes can be implicitly done with Vitruv's common notion of change. The notion of change of DeltaEcore has no Operation for existence changes, and if an object is created needs to be determined for every reference change. For Example, the delta dialect for BCS only needs a creation or deletion if any changes involve the set/add/insert delta operation, when adding a Signal to the System or when inserting a new sub-component to a component. This means that while both are capable of existence changes, only Vitruv can express direct existence change.

Attribute changes can be implicitly done with the notion of change of Vitruv and DeltaEcore. Vitruv differentiates between single-valued and multi-valued attribute changes. The deltas of DeltaEcore only use the modify delta operation for attribute changes and do not differentiate between single-value or multi-value. Not allowing for distinguishing how insertions of attributes in containers are handled alone by their delta, diminished the expressiveness.

Reference changes of both notions of change are able to insert and replace both single-valued and multi-valued elements. But DeltaEcores' notion of change also has the ability to revert to a default value of single-valued references and can both append a reference and insert a reference at a specific position.

Composition While the notion of change of DeltaEcore supports the composition of deltas, with which multiple deltas can be composed into one delta, the notion of change of Vitruv does not support composition in its notion of change. It must be noted that while the notion of change of Vitruv does not have composition, the Vitruv framework provides composition. But as it is not directly part of the notion of change, it is not noted while evaluating both notions of change.

Custom Operations, which could express any variety of expressiveness, can only be used with the notion of change of DeltaEcore.

5.2.3 DeltaEcore Dialect

As mentioned, DeltaEcore uses a dialect, which must be created for any use case. The dialects used for the BCS are shown in Appendix A.1 and A.2. As the number of operations is high, only one operation corresponding to an operation of DeltaEcore is shown. All other operations have the same structure as the operations shown, meaning, for example, that any add operation is the same except for their name and id variable name, but no structural difference is present in any operation. Thanks to the usage of expressive names, no explanation is needed for each operation as their use is directly expressed in the name of the operation.

For the state-machine dialect, it is notable that errors in its implementation lead to unusable operations, namely, all operations involving triggers, constraints, or behaviors. Because of this, the state-machine dialect is not used in the case study as the translation of operations different in their structure or intent from the deltacomponent dialect operations is not possible. As such, the use of the state-machine dialect would not provide any additional information for the research questions.

5.2.4 Change in the BCS

In the following example, we will show the difference of change with a simple use case in the BCS. In this change, we want to add a new connector to the system, name it, and set its ports. This use case involves the most important operations used to bring an instance of BCS to a valid feature configuration.

DeltaEcore: A delta composed of the following dialect delta operations

1. **Add To Connectors In System:** Uses the Add Delta Operation and is interpreted as a creation of the connector because it is added to the System.
2. **Modify Attribute Name In Connector:** Uses the Modify Delta Operation.
3. **Set Target Port In Connector:** Uses the Set Delta Operation.
4. **Set Source Port In Connector:** Uses the Set Delta Operation.

Vitruv: Multiple changes that are separate from each other

1. **Create EObject:** Creation of the connector
2. **Insert EReference Operation:** Inserting the connector to the system

3. **Replace Single Valued EAttribute Operation:** Naming the connector
4. **Replace Single Valued EReference Operations:** Set the target port in the connector
5. **Replace Single Valued EReference Operations:** Set the source port in the connector

The first difference in expression between the notions of change of Vitruv and DeltaEcore can be seen in the difference in how easy it is to see the intent. Each Delta expresses what kind of change will be done on the BCS instance directly by its name. And with the composition of all deltas into one delta, the delta gains more expressiveness, as all deltas show a single intent with the use of a composition. For the notion of change of Vitruv, every change needs to be examined by its elements. Only by looking at what each change references can we determine what the intent behind a change is. Also, no composition means that the order of each change can not be expressed, and the overall intent of adding and configuring a connector is less clear.

Another difference in expression is the existence change. Only with the knowledge of the delta interpretation of our dialect can we determine that the add delta operation includes a creation. Vitruv directly expresses the creation of a new connector.

5.3 Execution of Case Study

The case study will use the implemented translator to mostly evaluate the information loss between translations and give additional insight into the differences in capability and expressiveness of both notions of change, as explained in chapter 3.

After some usage of the translator, a problem occurred. Because of the translation behavior of Eclipse Epsilon ETL, a top-to-bottom translation of each change file is not possible. Because of this, the implemented transformation rules are structured in a way to ensure that first all element-destroying rules, then all element-creating rules, and lastly all element-modifying rules are translated. With this ordering, it should be ensured that the usage of translated changes of the same kind have the same effect on an instance. This is also useful as no composition is present in a change of Vitruv, which could mean changes can be disorderly when translating to DeltaEcore. The mentioned ordering can also be checked with the use of a round-trip. If the resulting changes of the same kind have the same number of operations with no loss of capability, the usage of both changes on the same instance should produce equal instances.

The round-trip is also used to ease the creation of Vitruv changes. If a round-trip produces in its first translation, DeltaEcore to the change of Vitruv, a change which modifies an instance in an equal way, the needed change representation in Vitruv is made. The second translation presents the translation from the change of Vitruv to DeltaEcore for evaluating any loss of information. If the instances are not equal, a Vitruv change, which produces an equal change as its DeltaEcore counterpart, is made by hand. An addition to a normal round-trip is made; after the second translation, the resulting DeltaEcore change will be translated back to a change of

Vitruv. This is used to gain a new modified instance to compare to the original modified instance of DeltaEcore.

As the translator can make use of a valid instance for a change, to provide a way of directly applying the to-be-translated change, but also prevent some possible information loss by using the instance as a reference, the case study uses an empty instance and a valid instance for all round-trips.

In addition to the round-trip translation, one change will be repeatedly round-trip translated to test for any degradation or deviation that could occur. The usage of a single change, containing all the most used operations in a common use case as described in the BCS usage example, should be sufficient, as no operations interact with one another while translating.

5.3.1 Expected Results

The case study will mostly result in insight into the information loss of the translation between both notions of change. Capability and expressiveness differences should be covered by the previous direct look at both notions of change.

As for information loss, loss of capability should mostly be as determined by table 5.1. Each loss of capability should be traced back to the "not (directly) able to" entries. Loss of expressiveness should occur for each Modify Delta Operation and Insert EReference Operation, as these operations have different expressiveness as seen in section 5.2.2.

The difference in instance content will cause different information loss. With an empty instance, only changes can be looked at for information, so more information loss should occur. A valid instance for a change should provide additional information, which should make it possible to give information, which will make it possible to retain information important to determine if an operation should be translated into, for example, an add or insert operation in the case of Vitruv change to DeltaEcore translations.

5.3.2 Metrics

To evaluate the results of the translations, some additional metrics are needed in order to allow a sensible interpretation of the results. The following metrics are adapted or inspired from "It's Your Loss: Classifying Information Loss During Variability Model Roundtrip Transformations" [Fei+22].

Roundtrip Quality (R-Quality) describes the state in which the resulting change of the roundtrip compared to its start change is. The changes can be identical, different, or equal in the amount of each operation type and operation parameters.

Instance Quality (I-Quality) describes how the use of all changes would affect an instance of a model. This means the use of the original change, its round-trip equivalent, and its translated equivalent. The instance of the model can either be equal in all changes or inconsistent. For equality, XML-specific layout is ignored if it does not change the content of the instance. An example would be the order of system components in a file, as the id indicates the order of the components, the file order is not relevant.

Capability Loss (C-Loss) indicates the number of operations that could not be translated and are lost in the translation. Each translation loss is counted, and all are presented in a tuple, starting with the first translation and continuing with the next following translation.

Expressiveness Loss (E-Loss) indicates the number of operations that could be translated into a change of the same kind as the original, but are compared to the original change in a less expressive state. The less expressive state can be an operation that produces the same result but is less specific than the original, for example, DeltaEcore's add operation in the original but the insert operation in the translated round-trip result. Also counted is any optional information about the change that is lost, mostly the optional information about the dialect of DeltaEcore. Not counted is the loss of expressiveness, in operation naming, as the translation from the dialect to the change of Vitruv would cause a loss in every operation. Each translation loss is counted, and all are presented in a tuple, starting with the first translation and continuing with the next following translation.

Change Size (C-Size) defines the number of operations that are present in a change. It is presented as two tuples. Each tuple contains the number of operations in a change of the same notion of change, but for the two different changes involved in the round-trip and additional translation. Entries in a tuple are ordered from left to right, with the entry on the left representing the operation amount of the change that was present or translated first. The first tuple represents the sizes of the dialect deltas, and the second tuple represents the sizes of the changes of Vitruv.

5.3.3 Method Example

As described for each delta of the dialect, a round trip with an additional translation is used. If we were to use this method on the example from Section 5.2.4, we would start with the first translation from the dialect to the change of Vitruv, and then examine the translated change and the created instance to ensure a valid Vitruv change was translated. We can then examine both the change of Vitruv and the delta of the dialect for the first C-Loss and E-Loss. With the example from section 5.2.4, the C-Loss should be zero as all operations can be mapped to a Vitruv operation. The E-Loss should be one as the add operation is translated into an insert operation.

After that, the next translation should be made, and now the second C-Loss and E-Loss can be examined. For example, the C-Loss should be one as the EObject creation can not be translated, and E-Loss should be one because of operation 3, the naming of the connector, as DeltaEcore does combine single- and multi-valued elements. The translation of the insert operation to the add operation should be successful, as the creation of a new connector in an empty instance has an index that is equal to its size.

Then the last additional translation is made. We should get an exact copy of the Vitruv change, with the same result for C-Loss and E-Loss. All instances for the quality metrics are now created. The only capability loss is the creation operation, and all E-Loss is not permanent, meaning the translator can translate back to an equally expressive change. Because of this and the order of the translation rules, the instances should be identical, meaning the I-Quality should be set to identical. As the changes will be ordered according to the translation rules, the original delta and translated delta will not be identical, as the operations are ordered differently. Both deltas should still be equal, and with that, the R-Quality should be set to equal.

Lastly, the size of all changes is counted; in the example, all changes of the same notion of change should be the same size. The deltas should be of size four, and the changes should be of size 5.

To conclude, we would have (0,1,0) C-Loss, (1,1,1) E-Loss, Equal R-Quality, Identical I-Quality, and (4,4) (5,5) size.

The evaluation with a valid instance is structured in the same way, and the example should conclude with the same values for the metrics when a valid instance is used.

5.3.4 Result

The following tables show the result of the round-trip translations of the changes and the result of the repeated translation of one change, which tests for any degradation in the changes.

Change	**R-Quality**	**I-Quality**	**C-Loss**	**E-Loss**	**C-Size**
DALA	Equal	Identical	(0, 20, 0)	(12, 8, 12)	(36,36) (56,56)
DALM	Equal	Identical	(0, 24, 0)	(16, 12, 16)	(44,44) (68,68)
DAS	Equal	Identical	(0, 77, 0)	(51, 38, 51)	(141,141) (218,218)
DASIM	Equal	Identical	(0, 18, 0)	(12, 9, 12)	(33,33) (51,51)
DAutomaticPW	Equal	Identical	(0, 43, 0)	(25, 22, 25)	(92,92) (135,135)
DCASA	Equal	Identical	(0, 10, 0)	(6, 4, 6)	(18,18) (28,28)
DCASM	Equal	Identical	(0, 10, 0)	(6, 4, 6)	(18,18) (28,28)
DCLSA	Equal	Identical	(0, 33, 0)	(21, 15, 21)	(60,60) (93,93)
DCLSM	Equal	Identical	(0, 35, 0)	(23, 17, 23)	(64,64) (99,99)
DHeatable	Equal	Identical	(0, 73, 0)	(41, 37, 41)	(155,155) (228,228)
DLEDAPW	Equal	Identical	(0, 40, 0)	(26, 19, 26)	(73,73) (113,113)
DLEDAPWCLS	Equal	Identical	(0, 62, 0)	(40, 29, 40)	(113,133) (175,175)
DLEDAS	Equal	Identical	(0, 75, 0)	(49, 36, 49)	(138,138) (213,213)
DLEDASIM	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DLEDCLSA	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DLEDCLSM	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DLEDEM	Equal	Identical	(0, 98, 0)	(66, 50, 66)	(180,180) (278,278)
DLEDEMWITHH	Equal	Identical	(0, 98, 0)	(66, 50, 66)	(180,180) (278,278)
DLEDFP	Equal	Identical	(0, 21, 0)	(13, 11, 13)	(40,40) (61,61)
DLEDHeatable	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DLEDManPW	Equal	Identical	(0, 51, 0)	(33, 26, 33)	(95,95) (146,146)
DRCKA	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DRCKCAP	Equal	Identical	(0, 25, 0)	(15, 13, 15)	(48,48) (73,73)
DRCKM	Equal	Identical	(0, 25, 0)	(17, 13, 17)	(46,46) (71,71)
DRCKSFA	Equal	Identical	(0, 10, 0)	(6, 4, 6)	(18,18) (28,28)
DRCKSFM	Equal	Identical	(0, 11, 0)	(7, 5, 7)	(20,20) (31,31)

Table 5.2: **Changes of BCS Components with an empty instance:** The change column shows the name of each evaluated change, and each following column shows the result of the change-specific evaluation in the form of the metrics described in section 5.3.2.

Table 5.2 and 5.3 show the result of the translations. As mentioned, because of the Eclipse Epsilon ETL rule handling the R-Quality, never has the "Identical" result, but resulting changes of the same kind always result in the identical instance, if they are applied to the instance.

For the information loss, the C-loss only contains the loss of creation operations of the change of Vitruv. The E-Loss from the delta dialect to Vitruv is the loss of expressiveness from add operations translated to insert operations. In the translation

Change	R-Quality	I-Quality	C-Loss	E-Loss	C-Size
DALA	Equal	Identical	(0, 20, 0)	(12, 8, 12)	(36,36) (56,56)
DALM	Equal	Identical	(0, 24, 0)	(16, 12, 16)	(44,44) (68,68)
DAS	Equal	Identical	(0, 77, 0)	(51, 38, 51)	(141,141) (218,218)
DASIM	Equal	Identical	(0, 18, 0)	(12, 9, 12)	(33,33) (51,51)
DAutomaticPW	Equal	Identical	(0, 43, 0)	(25, 22, 25)	(92,92) (135,135)
DCASA	Equal	Identical	(0, 10, 0)	(6, 4, 6)	(18,18) (28,28)
DCASM	Equal	Identical	(0, 10, 0)	(6, 4, 6)	(18,18) (28,28)
DCLSA	Equal	Identical	(0, 33, 0)	(21, 15, 21)	(60,60) (93,93)
DCLSM	Equal	Identical	(0, 35, 0)	(23, 17, 23)	(64,64) (99,99)
DHeatable	Equal	Identical	(0, 73, 0)	(41, 37, 41)	(155,155) (228,228)
DLEDAPW	Equal	Identical	(0, 40, 0)	(26, 19, 26)	(73,73) (113,113)
DLEDAPWCLS	Equal	Identical	(0, 62, 0)	(40, 29, 40)	(113,133) (175,175)
DLEDAS	Equal	Identical	(0, 75, 0)	(49, 36, 49)	(138,138) (213,213)
DLEDASIM	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DLEDCLSA	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DLEDCLSM	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DLEDEM	Equal	Identical	(0, 98, 0)	(66, 50, 66)	(180,180) (278,278)
DLEDEMWITHH	Equal	Identical	(0, 98, 0)	(66, 50, 66)	(180,180) (278,278)
DLEDFP	Equal	Identical	(0, 21, 0)	(13, 11, 13)	(40,40) (61,61)
DLEDHeatable	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DLEDManPW	Equal	Identical	(0, 51, 0)	(33, 26, 33)	(95,95) (146,146)
DRCKA	Equal	Identical	(0, 23, 0)	(15, 11, 15)	(42,42) (65,65)
DRCKCAP	Equal	Identical	(0, 25, 0)	(15, 13, 15)	(48,48) (73,73)
DRCKM	Equal	Identical	(0, 25, 0)	(17, 13, 17)	(46,46) (71,71)
DRCKSFA	Equal	Identical	(0, 10, 0)	(6, 4, 6)	(18,18) (28,28)
DRCKSFM	Equal	Identical	(0, 11, 0)	(7, 5, 7)	(20,20) (31,31)

Table 5.3: **Changes of BCS Components with a valid instance:** The change column shows the name of each evaluated change, and each following column shows the result of the change-specific evaluation in the form of the metrics described in section 5.3.2.

from the change of Vitruv to the dialect, E-Loss is present in the translation of any attribute changes from the separated Single or multi-valued operation of Vitruv into the combined operation of DeltaEcore, the Modify Delta Operation.

For the degradation test, we use the delta DCLSA, as it is one of the bigger changes in terms of operation amount. DCLSA was translated from DeltaEcore to the change of Vitruv and back repeatedly, with a valid instance and with an empty instance, each time 20 times. In each translation always the newest version of either DeltaEcore or Vitruv is used. The result of each translation with both a valid instance and an empty one was the same. No change showed any sign of degradation or any other deviation.

5.4 Conclusion

After evaluating both notions of change, all three research questions can be answered. For that, the findings will be summarized, and their interpretation and implications will be discussed.

5.4.1 Summary

In Section 5.2, we were able to gather information about the capabilities and expressiveness of each notion of change and their differences in the use of a dialect.

In the case of capability, both notions of change are nearly of the same capability. Except for existence changes, which the change of Vitruv is directly able to do, DeltaEcore has no direct operation but should execute creation in reference changes.

This is not clearly stated by its paper [SSA14]. DeltaEcore, unlike the change of Vitruv, has the capability of custom changes, which could provide any additional capabilities needed by a user of DeltaEcore.

For expressiveness, there is a difference in the use of a dialect in DeltaEcore compared to its non-dialect operations. The non-dialect operations are only more expressive in reference changes as compared to the change of Vitruv, because of additional specialized operations. In contrast, the change of Vitruv differentiates between change for multi-valued attributes and single-valued attributes, while DeltaEcore only uses the modified delta operation. Also, a major difference in expressiveness is the ability to create compositions. While DeltaEcore can create these, the change of Vitruv has no such construct. The use of a dialect can improve expressiveness, as shown in the previous overview and BCS example.

Following the evaluation mostly of both notions of change capability and expression, now the case study shows information loss between the translation of changes of both notions of change.

The result of the case study shows the expected capability loss of the inability to directly translate creation changes from Vitruv to DeltaEcore. It also shows the assumed overall equality of capability, as all changes could achieve identical transformations to an instance. The loss of expressiveness is also shown in an expected way. The provided additional expressiveness of the dialect of DeltaEcore can not be translated to the change of Vitruv and will be lost in a translation. The same is the case for any differentiation of single- or multi-valued attributes in a translation from the change of Vitruv to DeltaEcore. Notable is that, contrary to what is assumed, the use of a valid instance does not make any difference for the changes of the BCS.

5.4.2 Threats to Validity

For threats to validity, we look at threats that could lead to altered research data, as could be possible with the change of order of change operations. Also, a possible threat is missing coverage of use cases, especially edge cases, which could give new insight into expressiveness or information loss. Lastly, any threats to validity, including biases, are looked at.

One Thread to validity is the change of ordering in change operations. The order of rule translations should ensure that ordering can be neglected with regard to influence on answering the research questions, and only the loss of expressiveness of the ordering of operations is of consideration. The use of a round-trip and the order of translation rules still does not fully prove that the change in the order of operations can be neglected. If the order has any influence, then it can be assumed that the evaluated information loss is likely increased, as a change of order is unlikely to prevent any loss of information.

Another thread to validity is the single use of the BCS. While the BCS provides a use case similar to most use cases, which would make use of DeltaEcore and Vitruv, some edge cases could be missed because their use would be unlikely in a common use case. This could be avoided with additional use cases, which also include edge cases. These edge cases can be seen as less relevant for answering the research questions, as they would not be encountered in most common use cases, but could still contribute to all three research questions.

An additional threat to validity is the manual evaluation with only one examiner. As the evaluation is not systematic, a bias could be present in the assessment of the expressiveness.

5.4.3 Interpretation

The difference in capability is noticeable little. While existence changes are not directly made in the notion of change of DeltaEcore, the notion of change is still able to indirectly make existence changes. The only major difference is the capability for custom operation in the notion of change of DeltaEcore, which is not possible in the change of Vitruv. In all other cases, the capability of both notions of change can be seen as equal and capable of change sufficient for excepted use cases, similar to the BCS.

In contrast to the similar capability of both notions of change, the expressiveness of both is, in major part, because of the use of a dialect in DeltaEcore, vastly different. While looking at all non-dialect operations, there are some more specialized operations in the notion of change of DeltaEcore, but these provide far less difference in expressiveness than the dialect adds. With the DeltaEcore dialect, the expressiveness can be substantially better than that of the change of Vitruv. With the comparison of the same change done in a BCS example, DeltaEcore's use of a dialect shows how a dialect can substantially add to the expressiveness of a change. The additional expressiveness of the dialect operations is the following.

- Providing the intent of the change operation directly with the change name.
- Achieving readability by reducing operations, as much information can be indirectly gained from the operation name
- Providing specialized operations for any needed transformation in the instance
- Defining all needed modifications of the instance needed for the features model defined product variants.

These operations can express the direct intent and result of their use. With the addition of the reduction of only strictly needed elements in the operation, it is easily known what the operation does and what its target is. With the addition of compositions in DeltaEcore, a delta of the DeltaEcore dialect can depict a change as described by the documentation of the BCS, clearly depicting a change to a new product defined by the feature model of the BCS.

In contrast, the change of Vitruv does not use any dialect and uses operations that only express a general change, meaning no insert operations that can directly express information by name or specialization. But as needed in DeltaEcore, no interpretation is needed for attribute modifications or existence changes in the change of Vitruv.

The information loss of capability in a translation is thanks to the similar capability being greatly reduced. As the case study shows, only the existence changes are lost from a translation from the change of Vitruv to the delta dialect of DeltaEcore. This loss of information is not completely done as the existence change is done indirectly in DeltaEcore, so information about existence changes can still be gained by a dialect delta, which was translated from Vitruv.

The Loss of expressiveness is a lot greater than the loss of capability. In essence, any translation from the DeltaEcore dialect to a change from Vitruv loses expressiveness in every operation, because of the missing dialect use of the change of Vitruv. In addition, as the case study shows, a lot of information loss is also the reduction of specialized operations into more generic operations with some transformation result. In the opposite translation direction, from Vitruv to DeltaEcore dialect, only attribute changes lose expressiveness, as no difference is made between multi- or single-valued attributes.

But as the case study also shows, the expressiveness loss of each change is not completely lost by translations. Even without the use of a valid instance, a round-trip translation starting with a delta from the DeltaEcore dialect can be made without losing any expressiveness in the resulting new delta, except in the order of each delta operation. And even with the changed order of delta operations, the newly ordered structure of the composition, not much expressiveness is lost for the composition, as the intent of new additions, deletions, and modifications in an instance can still be seen, as all delta operations are the exact same.

5.4.4 Implications

To answer RQ1, as shown by the interpretation, both notions of change are sufficient to achieve any needed change for common use cases, as is the case with BCS. Their capability is similar enough to be of no relevance in choosing one notion of change over the other, except if the need is for an extension of capability, which only DeltaEcore provides. In the case of expressiveness, DeltaEcore provides significantly more expressiveness through its use of a dialect. It also must be noted that the dialect is not provided by DeltaEcore and must be created for every use case. This changes the comparison because it creates a distinction between the use of both notions of change. The change of Vitruv presents an ideal way of handling change in any instance without any need for dialect creation. In contrast, DeltaEcore provides a significantly more specialized way of handling change for a specific type of use case, but needs dialect creation, which only provides more expressiveness if properly made.

In summary, both notions of change provide different approaches to change. DeltaEcore provides a specialized approach that can be used to achieve a very expressive change creation for a chosen use case. The change of Vitruv provides a more uncomplicated way of change creation, but with less expressiveness. Which one should be chosen can only be said if it is known what kind of use case is needed.

In the case of the BCS, DeltaEcore would be preferable as only two dialects are needed and provide, in combination with compositions, a more expressive way to handle any change. Would there be more aspects present that need their own dialect, the change of Vitruv could be preferable, as giving up expressiveness for no new needed dialects can be preferable.

The RQ2 focuses mostly on expressiveness, as the capability of both notions of change is not in a disruptive state, and also on some equality. For the notion of change of DeltaEcore, the non-dialect operations can be made clear by providing a direct way for existence changes and separated delta operations for single- and multi-valued attributes. This would increase the not-as-clear way the notion of change of DeltaEcore handles both of these operations. For the notion of change of Vitruv, an implementation of composition would achieve a lot more expressiveness, as mentioned before, its absence reduces expressiveness. The addition of any specialized operations, for example, an attribute change, like the add delta operation in the notion of change of DeltaEcore, is not as significant as it is only relevant to a specific use case. Also, as both notions of change provide a different approach, the notion of change of Vitruv does not need the use of a dialect.

For other notions of change, the extension of composition capabilities should always be a beneficial addition. As mentioned with the notion of change of Vitruv, the absence of compositing causes a loss of expressiveness. Specialized operations are a beneficial extension if relevant to the use case of the notion of change, meaning the extension of specialized operations must be assessed for the notion of change currently used. Also, an extension that must be determined for the use case of a notion of change is the use of a dialect. While a dialect provides expressiveness, the use case of a notion of change can be more focused on managing multiple different model instances, which also applies change in an automated way. There, the creation and usage of dialect would not be needed, as changes would not need much expressiveness, while being applied and handled automatically.

The information loss of capability of the translation is mostly present in the creation of changes, and also not completely, as the notion of change of DeltaEcore makes them indirectly. Also, while information on expressiveness is lost as shown in any translation, for a common use case like the BCS, the lost expressiveness can be regained by translating back to the former. With that, translating between the deltas of DeltaEcore and the change of Vitruv can be done as needed without the fear of permanently losing any information in a change. For RQ3 this means that expressiveness and capability are present but do not compromise the translation, as information is not permanently lost

5.5 Verdict

DeltaEcore and the change of Vitruv both present a valid way of presenting and handling change. While both have some ways they could be improved, which one should be used is not easily said and needs to be evaluated for any use case. But while using one over the other, both can be translated without any complications, so changing the use of one to the other can be made significantly easier using a translator such as the translator implemented in this thesis.

6. Related Work

This section explores existing research related to the evaluation of notions of change, particularly focusing on comparisons, information loss, expressiveness, capability, translator tools, and delta dialects.

A hindrance in finding related works for a notion of change is that similar approaches to evaluation notions of change are rarely found, and few found comparisons focus mostly on capability comparisons. The approach of overview examination and information loss comparison used in the thesis for evaluating notions of change is more similar to papers on variability modeling. The evaluation of information loss, expressiveness, capability, and the use of a translation tool is found in many works for variability modeling, but as shown in this thesis, can be done similarly for notions of change. This thesis shows that a relation between evaluating variability in languages and evaluating notions of change is present.

Notion of Change Comparison

While finding similar notions of change comparisons can be difficult, one such comparison exists between DeltaEcore and SiPL with "Delta-oriented development of model-based software product lines with DeltaEcore and SiPL: A comparison" by Pietsch et al. [Pie+20]. Pietsch et al. focus on a capability comparison of two delta-oriented notions of change, and describe DeltaEcore similarly as described in this thesis, and SiPL as a delta-based modeling framework that can generate delta modules by comparing model instances. Also offered by Pietsch et al. are facilities for integrating analyses and refactoring on sets of interrelated delta modules, visualizing results, and triggering refactoring. The paper concludes that both tools address the complexities of delta-oriented development but with different focuses. DeltaEcore emphasizes the generation of delta languages, while SiPL offers a more comprehensive environment for managing and analyzing delta modules.

The paper provides a detailed comparison with a focus on the capability of both delta-oriented notions of change. While not comparing their expressiveness, the paper instead focuses on the comparison of both in regards to DOP. The paper comes to a similar conclusion with SiPL and DeltaEcore, as this thesis does with

DeltaEcore and the change of Vitruv. Both notions of change are valid options for different use cases. While this thesis focuses on different notions of change, the paper can be more specific in the differences between DeltaEcore's and SiPl's approach in regards to DOP, but does not show differences with more diverse notions of change. Pietsch et al. also provide no way of translating both notions of change, focusing only on an overview of both notions of change. In summary, the paper shows a way of comparing notions of change that both use DOP, with a detailed focus on capability differences, while this thesis also includes a capability comparison, an expressiveness comparison, and tests for information loss when translating is also done.

Meta-Model Quality Evaluation

While both DeltaEcore and the change of Vitruv can be seen as a meta-model, with which change can be described, meta-models are used in many different use cases. With that, other quality evaluations or quality definitions have been made. One such way of assessing the quality of a meta-model is given in the paper "Assessing the Quality of Meta-models" by Jes´us J. L´opez-Fern´andez et al. [LGL14]. This paper introduces mmSpec, a language for specifying properties on meta-models, and metaBest, a supporting tool used to check properties on meta-models and visualize the problematic elements. The paper used mmSpec and metaBest to create a catalog of 30 meta-model quality properties, which were then evaluated against a collection of 295 meta-models.

The paper proposes a way of automatically evaluating any meta-model for a given set of properties, and also lists properties, such as redundant checks. It provides properties and an approach through which additional evaluations could be made on the notions of change evaluated in this thesis. But the assessed research questions of this thesis show that for notions of change, an automatic evaluation with common quality properties is problematic. As described, the expressiveness and information loss are not as easily assessed, which causes the need for manual evaluation.

Information Loss

One related work is titled "It's Your Loss: Classifying Information Loss During Variability Model Roundtrip Transformations" by Kevin Feichtinger et al. [Fei+22]. It addresses the challenges of information loss when transforming variability models in SPLs and focuses on identifying and classifying the information lost during one-way and round-trip transformations. The information loss is split into several classes. These focus on structural, configuration, and semantic losses. It concludes with the recognition that the classification and round-trip use help modelers understand the impact of transformations and identify areas for improvement in existing transformation implementations. It also defines the extent of information loss for several languages.

The paper focuses on evaluating information loss with much focus on the way each language represents its elements. This includes how values are stored and their semantic and structural build. This allows the paper to differentiate each language in terms of exact usage. While the paper focuses on variability models, and with that mostly on feature models, the evaluation and results are similar to the way an evaluation would be achieved for notions of change. The information loss in this thesis is evaluated similarly, but focuses more on the information loss of expressiveness. While the paper gives a detailed evaluation of the exact usage of each language, the overall expressiveness is not focused on. But in contrast to the paper, this thesis simplifies the exact usage of each language as the focus of the capability is not how a change is structured or configured, but rather if the intended goal of a change can be achieved at all.

Both this thesis and the paper evaluate important aspects of a notion of change or feature model, and as they are similar in their method of analyzing information loss, they show how similar information loss can be determined in feature models and notions of change. Giving both the possibility to access the methods of the other.

Expressiveness and Capability

One paper that evaluates the expressiveness of variability is ”On the Expressive Power of Languages for Static Variability” by Bittner et al. [Bit+24]. The paper formalizes a way to analyze the expressiveness, completeness, and soundness of various languages with the use of a common semantic domain. Completeness describes whether a language can produce all variants of a feature model, and soundness describes whether the variability language can only describe variants with its language. Expressiveness is defined in this paper as the extent to which a variability language is more or less expressive than another language. The definitions are formed in a way to work with a defined formal framework, which allows the use of the paper's proposed tool in an open-source Agda library.

This defined framework can be used as a standard for analyzing, among other things, the expressiveness of variability languages. The paper combines expressiveness and capability and reduces expressiveness to whether a variability language can express the same variant of a feature model, unlike the definition of expressiveness in this thesis. The paper describes an overall different approach to evaluating the differences of variability languages by giving a possible standard with its framework and a possible automated way of evaluating. This can also be extended to the notions of change, while the framework would need extensions on its definition of expressiveness.

Another conclusion of the evaluation in this thesis is that a notion of change does not need to be better than another notion of change, but a decision must be made in which case a notion of change should be used over another. The paper ”An Analysis of Variability Modeling Concepts: Expressiveness vs. Analyzability” by Holger Eichelberger et al. [EKS13] highlights the need for a systematic characterization of modeling approaches based on their expressiveness, the range of supported modeling concepts, and analyzability, the ability to determine implied properties of a model, such as consistency or satisfiability. The paper gives a classification for variability in increasing complexity and provides an overview of each classification's tradeoffs between expressiveness and analyzability.

Holger Eichelberger et al. provide a useful classification for variability languages that could be adapted for notions of change. While this thesis does not provide such a classification, as it only focuses on two notions of change, the conclusion of its evaluation shows the need for a classification for any notions of change to help in determining when to use one notion of change over another.

As seen in this thesis, the evaluation of notions of change is similar to the evaluation of variability languages in both papers. While the classification and creation of an evaluation framework are not in the scope of this thesis, the thesis shows how the evaluations of variability languages could be extended to notions of change.

Translator

A way of translating variability languages can be useful because, as mentioned, their usage can be dependent on a specific use case. Translating between languages can make this decision easy by allowing direct comparison with the use of a translator. The paper "Variability Model Transformations: Towards Unifying Variability Modeling" by Kevin Feichtinger et al. [FR20]. proposes a way of translating variability models with different approaches to allow a meaningful comparison between the two models. The paper demonstrates the feasibility of translation by converting a feature model into a decision model. The paper outlines a research agenda that includes experimenting with transformations to and from other variability modeling approaches and evaluating the correctness and completeness of transformations with a focus on round-trip transformations and potential information loss.

This paper explains the need for a way of translating variability languages, as the need for comparing properties like expressiveness or capability can be different in variability languages and their use over another, determined by each use case, as mentioned before. The use of a translator is similar to the use in this thesis, with also possible use of roundtrips for variable language translation. But unlike the translation of changes, which is very similar to normal model-to-model transformation, the paper describes the problem of two-way translations as rules of variability languages can be more complex than that of a notion of change.

Delta Dialect

While this paper does make use of a delta dialect for DeltaEcore, the creation of one is not a focus of the thesis, but is still an important part of a delta dialect. The paper "Extractive Software Product Line Engineering Using Model-Based Delta Module Generation" by David Wille et al. [Wil+17] describes a language-independent and automatic delta language generation based on the results of the introduced variability mining, an analysis of differences, to find what is the same and what is different in a product. The generation of the delta language includes the generation of a delta dialect, which is tested with the delta language generation in two case studies, of which one used the BCS. The case study showed that the more automated way of delta language generation, and with that, delta dialect generation, could be used to create a correct delta dialect for the BCS.

The paper shows the creation of all needed delta dialects and delta modules needed for a specific use case. As the one use case is the BCS, we can compare the importance of a dialect to this thesis. While this thesis uses provided dialects, the dialect errors, like in the state machine dialect, have influenced the evaluation of the expressiveness and capabilities of the dialects. The paper shows a possible automated way of delta language generation for use in expressiveness and capability evaluations.

Conclusion

In conclusion, the thesis can be positioned in a way of using the established evaluation procedures of variability languages for notions of change. As shown in this thesis, the use of many similar evaluation methods is made, and possible extensions similar to existing variability languages could be made, like an automated process for comparison.

7. Conclusion and Outlook

The thesis shows that a comparison, in regard to expressiveness and capability, is an important aspect in deciding which notion of change should be used. It also shows that the decision can not be made universally but needs to be made for every use case. As for the notion of change of DeltaEcore and Vitruv, both notions of change a mostly similar in their capability, and the decision is mostly about whether a need for a dialect is present. While this dialect needs to be created for a specific use case, this thesis shows the greater expressiveness of DeltaEcore's delta dialects.

The related work includes evaluations of variability languages, which are done similarly to this thesis. These make use of many frameworks and standards, which could also be used for notions of change. An example would be a classification for notions of change. As seen in this thesis, while both notions of change are different, both are useful in their own regard. A classification for notions of change that makes the different approaches of both notions of change clear would be useful.

Another example would be the automation of the evaluation in this thesis, with a framework giving the option to either automate the case study or the overview of both notions of change would make the extension to the evaluation of different notions of change possible, as a standardized and automated way is provided with a framework.

An Addition to the thesis method of evaluation would be the evaluation of the interpretation that each notion of change needs or allows, and what influence it has on capability, expressiveness, or information loss. This could help determine the influence interpretation has on notions of change.

Also evaluated could be the best use case of each notion of change. By finding useful metrics, different notions of chance could be compared in how best they would fit into use cases that have a specific need. An example could be metrics that allow for comparing two notions of change in how good they would fit a use case involving high software security needs.

While this thesis evaluates two notions of change, DeltaEcore and Vitruv, additional notions of change exist. One such is DeltaJ, a specific delta-oriented language for

Java 1.5, designed for SPL [Kos+14]. It allows for modifications to Java programs, which can be adding, modifying, or removing classes or methods using delta modules. DeltaJ shows a very specific variant of a notion of change. While it is similar to DeltaEcore, unlike it, DeltaJ can not generate delta languages and is made for a specific programming language. While both DeltaEcore and DeltaJ are notions of change for DOP, their scope is quite different, showing that even comparisons for notions of change of both DOP can be made with a possible high amount of differences. This could be another possible notion of change with which an evaluation, similar to this thesis, could be made. Both the notion of change of DeltaEcore or Vitruv could be compared to DeltaJ, which could lead to finding additional possible extensions to a notion of change.

The next possible work should involve the repair of the state-machine dialect, as the functional operations show a different structure and intent from the operations of the component dialect. The repaired state-machine dialect could be used for similar evaluation as has been done in the case study. Another addition could be the evaluation of more common meta-model qualities for both notions of change. As common meta-model qualities determine the ease of use of a meta-model, it would help in possible improvements of both notions of change development, which would also aid in possible additions, like the composition capability for the change in Vitruv.

Furthermore, the use of the translator is shown to be done without permanent loss of information while translating, which allows the use of both notions of change with the same use case. As the translator presents a solid base for the translation of notions of change and the definition of new rules for any notion of change can easily be done, the translator could be expanded on. New translation rules would allow the use of different notions of change, and a possible extension of the translator with a GUI could make use of user input to include a way of mitigating permanent information loss, as a user could fill out any missing information. The use of a GUI would also allow for the visualization of the use of changes on an instance, which would increase the readability of changes.

Bibliography

[ABN10] M Amstel, M. Brand, and Phu Nguyen. “Metrics for model transformations”. In: *Systems Research and Behavioral Science - SYST RES BEHAV SCI* (Jan. 2010).

[Bit+24] Paul Maximilian Bittner et al. “On the Expressive Power of Languages for Static Variability”. In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689747. URL: https://doi.org/10.1145/3689747.

[Bos+02] Jan Bosch et al. “Variability Issues in Software Product Lines”. In: *Software Product-Family Engineering.* Ed. by Frank van der Linden. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 13–21.

[CV11] Deniz Cetinkaya and Alexander Verbraeck. “Metamodeling and model transformations in modeling and simulation”. In: *Proceedings of the 2011 Winter Simulation Conference (WSC).* Dec. 2011, pp. 3043–3053. DOI: 10.1109/WSC.2011.6148005.

[EKS13] Holger Eichelberger, Christian Kröher, and Klaus Schmid. “An Analysis of Variability Modeling Concepts: Expressiveness vs. Analyzability”. In: *Safe and Secure Software Reuse.* Ed. by John Favaro and Maurizio Morisio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 32–48. ISBN: 978-3-642-38977-1.

[Fei+22] Kevin Feichtinger et al. “It’s your loss: classifying information loss during variability model roundtrip transformations”. In: *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A.* SPLC ’22. Graz, Austria: Association for Computing Machinery, 2022, pp. 67–78. ISBN: 9781450394437. DOI: 10.1145/3546932.3546990. URL: https://doi.org/10.1145/3546932.3546990.

[FR20] Kevin Feichtinger and Rick Rabiser. “Towards Transforming Variability Models: Usage Scenarios, Required Capabilities and Challenges”. In: *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B.* SPLC ’20. Montreal, QC, Canada: Association for Computing Machinery, 2020, pp. 44–51. ISBN: 9781450375702. DOI: 10.1145/3382026.3425768. URL: https://doi.org/10.1145/3382026.3425768.

[HT06] Brent Hailpern and P. Tarr. “Model-driven development: The good, the bad, and the ugly”. In: *IBM Systems Journal* 45 (Feb. 2006), pp. 451–461. DOI: 10.1147/sj.453.0451.

[Kla+21] Heiko Klare et al. “Enabling consistency in view-based system development - The Vitruvius approach”. In: *The journal of systems and software* 171 (2021). ISBN: 9781000124712, p. 110815. ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110815. URL: https://publikationen.bibliothek.kit.edu/1000124711 (visited on 12/02/2024).

[Kos+14] Jonathan Koscielny et al. “DeltaJ 1.5: delta-oriented programming for Java 1.5”. In: *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ ’14. Cracow, Poland: Association for Computing Machinery, 2014, pp. 63–74. ISBN: 9781450329262. DOI: 10.1145/2647508.2647512. URL: https://doi.org/10.1145/2647508.2647512.

[LGL14] Jesús J. López-Fernández, Esther Guerra, and Juan de Lara. “Assessing the Quality of Meta-models”. In: *Proceedings of the 11th Workshop on Model-Driven Engineering, Verification and Validation co-located with 17th International Conference on Model Driven Engineering Languages and Systems, MoDeVVa@MODELS 2014, Valencia, Spain, September 30, 2014*. Ed. by Frédéric Boulanger, Michalis Famelis, and Daniel Ratiu. Vol. 1235. CEUR Workshop Proceedings. CEUR-WS.org, 2014, pp. 3–12. URL: https://ceur-ws.org/Vol-1235/paper-02.pdf.

[Lit+13] Sascha Lity et al. *Delta-oriented software product line test models-the body comfort system case study*. Tech. rep. TU Braunschweig, 2013.

[Nor02] L.M. Northrop. “SEI’s Software Product Line Tenets”. In: *Software, IEEE* 19 (Aug. 2002), pp. 32–40. DOI: 10.1109/MS.2002.1020285.

[Pie+20] Christopher Pietsch et al. “Chapter 8 - Delta-oriented development of model-based software product lines with DeltaEcore and SiPL: A comparison”. In: *Model Management and Analytics for Large Scale Systems*. Ed. by Bedir Tekinerdogan et al. Academic Press, 2020, pp. 167–201. ISBN: 978-0-12-816649-9. DOI: https://doi.org/10.1016/B978-0-12-816649-9.00017-X. URL: https://www.sciencedirect.com/science/article/pii/B978012816649900017X.

[Sch+10] Ina Schaefer et al. “Delta-Oriented Programming of Software Product Lines”. In: *Software Product Lines Conference*. 2010. URL: https://api.semanticscholar.org/CorpusID:2000112.

[SH11] Ina Schaefer and Reiner Hähnle. “Formal Methods in Software Product Line Engineering”. In: *IEEE Computer* 44 (Feb. 2011), pp. 82–85. DOI: 10.1109/MC.2011.47.

[SSA14] Christoph Seidl, Ina Schaefer, and Uwe Assmann. “DeltaEcore - A Model-Based Delta Language Generation Framework”. In: *Hans-Georg Fill, editor, Modellierung*. 2014. URL: https://api.semanticscholar.org/CorpusID:17525862.

[Wil+17] David Wille et al. “Extractive software product line engineering using model-based delta module generation”. In: *Proceedings of the 11th International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS ’17. Eindhoven, Netherlands: Association for Computing

Machinery, 2017, pp. 36–43. ISBN: 9781450348119. DOI: 10.1145/3023956.3023957. URL: https://doi.org/10.1145/3023956.3023957.

[Zol+20] Athanasios Zolotas et al. "Bridging Proprietary Modelling and Open-Source Model Management Tools: The Case of PTC Integrity Modeller and Epsilon". In: *Software and Systems Modeling* 19 (Jan. 2020). DOI: 10.1007/s10270-019-00732-1.

A. Appendix

A.1 Delta Component

A section of the Delta Component dialect in the Ecore format. Each similar operation was removed, meaning, for example, that only one operation for the modification of an attribute is shown. All shown operations are similar in their structure to those found in the other dialect.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="deltacomponent"
    nsURI="https://tva.kastel.kit.edu/bcs/ecorebased/metamodels/deltacomponent"
    nsPrefix="deltacomponent">
<eClassifiers
    xsi:type="ecore:EClass"
    name="DeltaComponent"
    eSuperTypes="ecore:EClass https://tva.kastel.kit.edu/deltavar/core/deltametamodel/
        genericdelta#//DEDeltaDialect"/>
<eClassifiers
    xsi:type="ecore:EClass"
    name="DetachPortFromInputPortsInComponent"
    eSuperTypes="ecore:EClass https://tva.kastel.kit.edu/deltavar/core/deltametamodel/
        genericdelta#//DEDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="portId"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute" name="componentIdOfContainer"
        lowerBound="1" eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//
            EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="AddToInputPortsInComponent"
    eSuperTypes="ecore:EClass https://tva.kastel.kit.edu/deltavar/core/deltametamodel/
        genericdelta#//DEAddDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EReference"
        name="newValue"
        lowerBound="1"
        eType="ecore:EClass https://tva.kastel.kit.edu/bcs/ecorebased/metamodels/component#//Port"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="componentIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="portId"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="RemoveFromInputPortsInComponent"
    eSuperTypes="ecore:EClass https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta#//DERemoveDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EReference"
        name="oldValue"
        lowerBound="1"
        eType="ecore:EClass https://tva.kastel.kit.edu/bcs/ecorebased/metamodels/component#//Port"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="componentIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="portId"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="InsertIntoInputPortsInComponent"
    eSuperTypes="ecore:EClass https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta#//DEInsertDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EReference"
        name="newValue"
        lowerBound="1"
        eType="ecore:EClass https://tva.kastel.kit.edu/bcs/ecorebased/metamodels/component#//Port"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="position"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="componentIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="portId"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="SetParentComponentInComponent"
    eSuperTypes="ecore:EClass https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta#//DESetDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EReference"
        name="newValue"
        lowerBound="1"
        eType="ecore:EClass https://tva.kastel.kit.edu/bcs/ecorebased/metamodels/component#//Component"/>
    <eStructuralFeatures
```

```
        xsi:type="ecore:EAttribute"
        name="componentIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="componentId"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="UnsetParentComponentInComponent"
    eSuperTypes="ecore:EClass https://tva.kastel.kit.edu/deltavar/core/deltametamodel/
        genericdelta#//DEUnsetDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="componentIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="componentId"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="ModifyAttributeIdInComponent"
    eSuperTypes="ecore:EClass https://tva.kastel.kit.edu/deltavar/core/deltametamodel/
        genericdelta#//DEModifyDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="newValue"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="componentIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
</ecore:EPackage>
```

A.2 Delta State Machine

A section of the Delta State Machine dialect in the Ecore format. Each similar operation was removed, meaning, for example, that only one operation for the modification of an attribute is shown. All shown operations are similar in their structure to those found in the other dialect, except for all operations involving triggers, constraints, or behavior, which are not implemented correctly.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="deltastatemachine" nsURI=
        "https://tva.kastel.kit.edu/bcs/ecorebased/metamodels/deltastatemachine"
    nsPrefix="deltastatemachine">
<eClassifiers
    xsi:type="ecore:EClass"
    name="DeltaStatemachine"
    eSuperTypes="https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta
        #//DEDeltaDialect"/>
<eClassifiers
    xsi:type="ecore:EClass"
    name="DetachTransitionFromTransitionsInRegion"
    eSuperTypes="https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta#//DEDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="transitionId"
        lowerBound="1"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="regionIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="AddToTransitionsInRegion"
    eSuperTypes="https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta#//DEAddDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EReference"
        name="newValue"
        lowerBound="1"
        eType="ecore:EClass␣statemachine.ecore#//Transition"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="regionIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="transitionId"
        lowerBound="1"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="RemoveFromTransitionsInRegion"
    eSuperTypes="https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta#//DERemoveDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EReference"
        name="oldValue"
        lowerBound="1"
        eType="ecore:EClass␣statemachine.ecore#//Transition"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="regionIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="transitionId"
        lowerBound="1"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="InsertIntoTransitionsInRegion"
    eSuperTypes="https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta#//DEInsertDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EReference"
        name="newValue"
        lowerBound="1"
        eType="ecore:EClass␣statemachine.ecore#//Transition"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="position"
        lowerBound="1"
        eType="ecore:EDataType␣http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="regionIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType_http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="transitionId"
        lowerBound="1"
        eType="ecore:EDataType_http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="SetTriggerInTransition"
    eSuperTypes="https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta
        #//DESetDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EReference"
        name="newValue"
        lowerBound="1"
        eType="ecore:EClass_statemachine.ecore#//Trigger"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="transitionIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType_http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="UnsetTriggerInTransition"
    eSuperTypes="https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta
        #//DEUnsetDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="transitionIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType_http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
<eClassifiers
    xsi:type="ecore:EClass"
    name="ModifyAttributeNameInRegion"
    eSuperTypes="https://tva.kastel.kit.edu/deltavar/core/deltametamodel/genericdelta
        #//DEModifyDeltaOperationDefinition">
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="newValue"
        lowerBound="1"
        eType="ecore:EDataType_http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures
        xsi:type="ecore:EAttribute"
        name="regionIdOfContainer"
        lowerBound="1"
        eType="ecore:EDataType_http://www.eclipse.org/emf/2002/Ecore#//EInt"/>
</eClassifiers>
</ecore:EPackage>
```