Master's Thesis

Integration of UML Profiles into the SiDiff and SiLift tools

Based on a SysML case study

Dennis Reuling

Faculty:	Faculty of Science and Technology
Department:	Electrical Engineering and Computer Science
Institute:	Software Engineering Group
Reviewers:	Prof. Dr. Udo Kelter, Dipl. Inf. Timo Kehrer
Date:	September 25, 2013

Abstract

Model Driven Software Development (MDSD) has become more and more present since the last years because of the paradigm shift from coding to modeling. One of the most popular and commonly used modeling languages is the Unified Modeling Language (UML). It provides the possibility of extending the modeling language itself by making use of the implemented *profiling* mechanism. New elements with its own semantics can be added to existing elements of the UML thus enabling integration of new domains or features easily. A popular example for exploiting this mechanism is the Systems Modeling Language (SysML), which extends the UML in the domain of systems engineering applications.

In the area of text-based tools the parallel work paradigm has been used for the last decades. In the area of model-based tools there have only be solutions which do not facilitate the freedom or the functionality on par with the latter, which leaves many features known to be desired. Three of these are the matching of corresponding elements, the detection and presentation of differences and the creation followed by the application of patches between two models. The *SiDiff* tool facilitates the first, whereas the *SiLift* tool provides the remaining two features.

This Master's Thesis introduces the integration of UML profiles in both tools:
Supporting a broad range of modeling languages and domains is crucial to modeling tools in practicality. Therefore the integration of UML profiles is a significant enhancement in this area and is done in both tools for supporting the whole processing pipeline: Beginning at the creation of differences and concluding at the application of a patch. A real world industrial automation case study using SysML as modeling language is presented and used as an exemplary input model to demonstrate the final result of this integration process.

Statutory Declaration

I declare on oath that I completed this work on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this, nor a similar work, has been published or presented to an examination committee.

Siegen, September 25, 2013 _____

Acknowledgement

Wem die Dankbarkeit geniert,
der ist übel dran,
denke, der dich erst geführt,
wer für dich getan!
Johann Wolfgang von Goethe

Ich möchte mich an dieser Stelle von ganzem Herzen bei meinen beiden Betreuern Herr Prof. Dr. Udo Kelter und Herr Dipl. Inf. Timo Kehrer bedanken. Ohne sie und ihre sowohl fachliche als auch menschliche Betreuung wäre diese Arbeit nicht möglich gewesen. Auch den übrigen Mitarbeitern des Instituts der praktischen Informatik bin ich zu tiefem Dank verpflichtet, welche mir stets zur Seite standen.

Nicht vergessen möchte ich an dieser Stelle auch meine Kommilitonen, die mich während dieses Lebensabschnittes begleitet und unterstützt haben. Während allen Phasen des Studiums habe ich neue Freunde gefunden, welche auch außerhalb fachlicher Fragen immer ein offenes Ohr hatten. Besonders erwähnt werden sollen auch die Mitglieder des Fachschaftsrates, mit denen ich viel gemeinsame Zeit verbracht habe, sowohl in der Rolle als Mitglied als auch in der Rolle als Freund.

Abschließend möchte ich den wichtigsten Menschen danken, meiner Familie und meinen Freunden. Ihr wart stets an meiner Seite und habt mir erst dadurch dieses Studium ermöglicht.

Vielen Dank!

Contents

List of Figures

List of Listings

List of Abbreviations

API Application Programming Interface
EMF Eclipse Modeling Framework
HOT Higher-Order-Transformation
ISO International Organization for Standardization
LHS Left Hand Side
MARTE Modeling and Analysis of Real Time and Embedded systems
MDSD Model Driven Software Development
NAC Negative Application Condition
OMG Object Management Group
PAC Positive Application Condition
PPU Pick-and-Place Unit
RHS Right Hand Side
SCS Semantic Change Set
SysML Systems Modeling Language
SEG Software Engineering Group
SERGe SiDiff Edit Rule Generator
UML Unified Modeling Language
UUID Universally Unique Identifier

Introduction

The paradigm shift from coding to modeling can easily be called one of the biggest changes in progress in the area of software development. The main reason for this change has been the rising complexity of software, whether it be in product families, different versions or the software itself. Developers are trying to restrain this complexity by using modeling languages and tools, which focus on the semantics of the given problem instead of underlying programming languages. Elevating the abstraction level eases the understanding of the problem and can be one solution to cope with the rising complexity of software.

One important aspect to facilitate this paradigm shift is the definition of modeling languages conforming to problems in practical domains of software. The UML has been introduced as such general-purpose modeling language and has been accepted by the International Organization for Standardization (ISO) as industry standard for modeling software-intensive systems since 2000. Although the modeling language already supports many areas of software development like *activities* or *data modeling*, the UML provides the generic facility for possible adaptions in new areas or domains via its own *profiling* mechanism: The UML or subsets of it can be extended by declaring own domain specific elements, adding new properties to already existent modeling elements. The profiling mechanism is defined in an additive manner, therefore all profiled UML elements are still conforming to the UML standard. Two popular examples of UML Profiles used by software developers in practice are SysML and Modeling and Analysis of Real Time and Embedded systems (MARTE).

Another important aspect for supporting MDSD are the modeling tools themselves, which should provide all needed and well-known features supported by text-based tools. To keep complex software manageable, there has to be the possibility to work in sufficient large teams in parallel, without giving up the freedom of single developers. Text-based tools are providing all necessary features for working in teams since decades like the de-

tection of correspondences and differences between two revisions of software. The *SiDiff* tool provides such features for modeling languages: By implementing different matching services, SiDiff can compute corresponding elements between two models and is therefore the foundation of supporting parallel work in software development. The *SiLift* tool takes the support one step further in the modeling tool pipeline: The detected differences are lifted upon a more understandable level for the developer by presenting the changes via edit operations. On this level of abstraction the changes are more comprehensive for the developer, which is crucial to parallel work paradigms. In the context of product families another feature is well known from text-based tools, which has mostly been absent in the modeling area: Patching of software. By making use of both SiDiff and SiLift another tool has been developed, which facilitates this functionality in the modeling domain. By supporting these three functionalities, *Matching*, *Lifting* and *Patching*, all basic features needed for parallel development are available in MDSD.

Supporting modeling languages, which conform to problems in practical domains, in modeling tools, which facilitate well-known features in parallel development areas, is a big step for the paradigm shift to MDSD. This Master's Thesis describes this integration process and is structured as follows:

Chapter 1 introduces the mentioned environment consisting of the modeling language UML including its profile mechanism and the tools used for the integration.

Chapter 2 describes the concepts of the whole integration process in a detailed manner, considering each pipeline step of the modeling tools.

Chapter 3 illustrates the implementation of the concepts described beforehand, whereas details and problems are demonstrated.

Chapter 4 analyzes a SysML case study in a comprehensive way, which will be used in the following chapter as input for testing the created solutions.

Chapter 5 sums up the integration success by testing the solution using the case study described in the preceding chapter.

Chapter 6 considers the results from all preceding chapters and presents a possible outlook and future work.

Chapter 1

Definition of Environment and Tools

This chapter lays the foundation for all following chapters by introducing the environment in which this Master's Thesis has been created. First the UML and its profiling mechanism is described, followed by a concrete example in terms of SysML. Furthermore Henshin, a graph transformation tool used in this integration process, is explained in detail. Finally the three target tools of integration are presented: *SiDiff*, *SiLift* and the *Patch-Tool*.

1.1 UML and UML Profiles

In the early 1990's the rising paradigm of MDSD demanded modeling languages for all domains and use cases, in which software development came to practical use. As in figure 1.1 depicted, many modeling languages have been developed in this era. One among them was the UML, which has been developed by *Grady Booch*, *Ivar Jacobson* and *James Rumbaugh* at Rational Software[25]. It differentiated itself from other modeling languages in its generic ideas and its wide support for modeling domains. Its special and important role has been lifted drastically in the year 1997, as the Object Management Group (OMG) adopted UML as modeling language. They fine tuned the language and presented their version to the ISO afterwards. The UML has been accepted as standard and published as Version 1.3 in 2000, which can easily be called one of the big milestones for UML and even for MDSD in general. Before standarization the industry hesitated to choose one of the many modeling languages for their own usage, because it takes time and money to develop tools which can handle each modeling language with its own semantics.

The paradigm change from coding to modeling has been delayed until the standarization of modeling languages like the UML. The OMG developed new versions of UML with new features ever since, the actual stable version used in practical environments is UML 2.4.1 and has been released in August 2011[11], which is also the version used throughout this Master's Thesis.

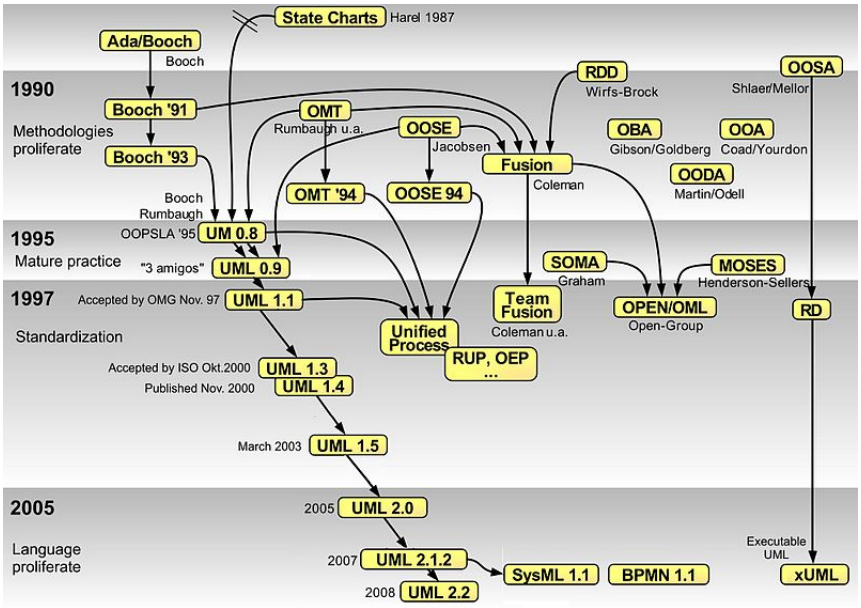


Figure 1.1: *Modeling languages history [25]*

The UML has been designed with many generic principles in mind[11] such as:

- Modularity
- Partitioning
- Extensibility

Making use of these principles the UML can be used to visualize practical problems in many domains, using elements in combination like *activies*, *components* or *use-cases*. An example of an UML class diagram, which is a very popular type of diagram in general, can be seen in figure 1.2. This Master's Thesis emphasizes only on the important parts of UML for the integration, a more detailed and comprehensive description of all possibilities the modeling language has to offer can be found in [14].

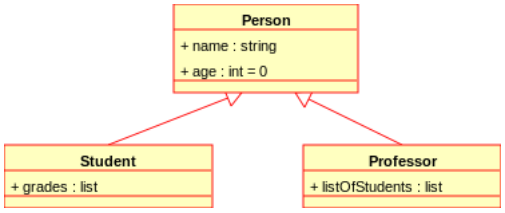


Figure 1.2: *UML class diagram example [24]*

Although the UML supports a wide range of modeling domains, the principle of *extensibility* has been integrated deeply with its own mechanism: The UML profiling mechanism. The idea behind this mechanism is the possibility for developers to define own modeling elements in UML notation and therefore add new semantics to an already known and widely supported modeling language such as the UML. Instead of defining a own modeling language for a new given domain from scratch, a defined profile can alter a (sub)set of UML to provide a semantically more unterstandable way of modeling or additional features, which have been missing in this particular area. An example of such an profile definition is presented in figure 1.3: Servers are defined as profile for better understanding in this modeling domain. They extend the UML with its own semantics, such as the relationship between a *device* and a *server*.

All UML profiles are defined in an *additive* manner. This basic principle leads to conformity between unprofiled and profiled UML models, therefore no new modeling tools are needed if they already support all features of UML. This is a tremendous advantage over other modeling languages, which do not facilitate such extensibility in this generic way or at all. Another example exposing this can be imagened easily: The technology of connected devices like tablets or new server backends like clouds are using a new way of network connectivity. To cope with this new area of problems in a MDSD way there

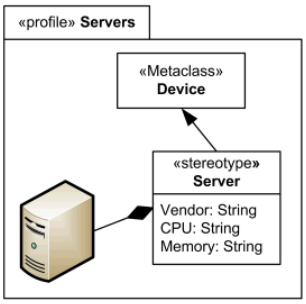


Figure 1.3: *Profile Application Example [18]*

has to be a modeling language and modeling tools capable of such modeling elements and their relationships. Using the UML profiling mechanism no new modeling tools are needed, all semantics of this new domain are modelled within UML profiles, such as relationships between cloud servers and its clients.

These are just small examples of what possibilities are available via the profiling mechanism. Other popular examples which are using this functionality extensively are MARTE and SysML, which will be explained in detail in the following section.

1.2 SysML

One example of making use of the UML profiling mechanism is the SysML profile, which defines new semantics to existing elements in the UML and extents the modeling language with new elements and diagrams. The profile is used in and has been developed for the domain of systems engineering applications. Like UML, SysML has been adopted by the OMG and its specification has first been published officially in September 2007[23]. The current version in use is SysML 1.3, which was issued by the OMG in June 2012 [12].

SysML tries to reduce the scope of modeling elements of UML by using only a subset of all elements available as depicted in figure 1.4. The wide support of modeling domains and elements of UML can lead to confusion among developers, which SysML tries to solve by only using domain specific elements in the area of systems engineering.

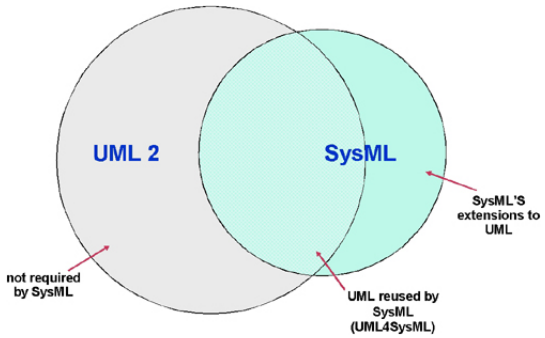


Figure 1.4: *Relationship between UML and SysML [13]*

Additionally parts of UML have been changed in semantics and new diagrams have been introduced. A SysML diagram overview is presented in figure 1.5.

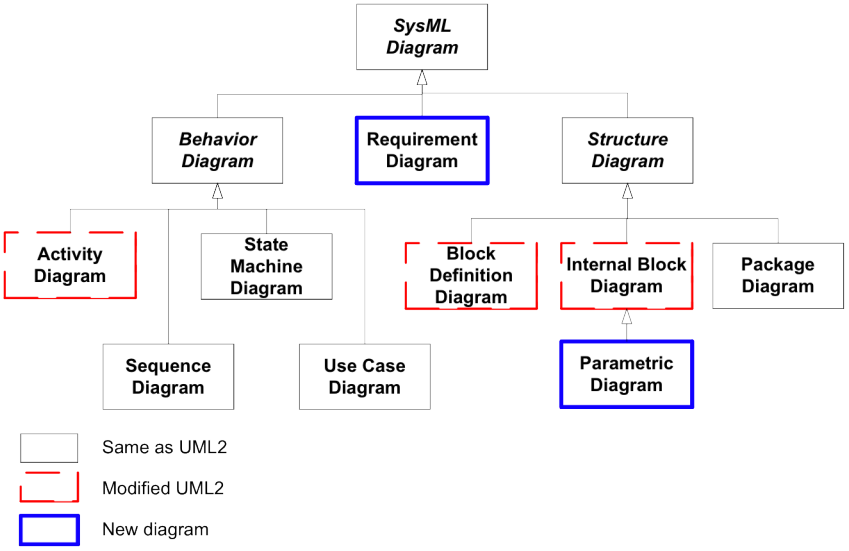


Figure 1.5: *The SysML diagram taxonomy [12]*

The new diagrams introduced in SysML shall be explained:

Requirement Diagram

Systems engineering is mostly driven requirement-based, therefore this new type of diagram meets this design principle. In this domain a requirement can be modeled graphically and contains properties and/or conditions which must be satisfied. Requirements can have relationships between modeling elements, which can also be requirements themselves. Another feature is the reuseability across product families and projects, which is realized through the namespace containment in SysML.

Parametric Diagram

As seen in figure 1.5, the parametric diagram has a direct relationship to the internal block diagram, which contains the basic modeling elements of SysML. The idea is to offer the possibility to describe constraints among properties between these basic elements such as *Blocks*. Behavior and structure models can be integrated easily with engineering analysis like performance or reliability, which often occur in systems engineering domains.

Additionally many modifications have been implemented to conform to the systems

engineering common language and principles in software design, making them more understandable for developers from this particular domain. A basic example for this is the UML *Class* element extension by the SysML *Block* element. This seems appropiate because of the common usage of mechanical or electrical parts, which do not conform to the semantic of *Class*. A more detailed overview and description of all elements, which are introduced by SysML and how they are used can be found in [12] and [21].

1.3 Henshin

Contrary to text-based tools, modeling tools are based on graph representations as input. A model can be defined as graph, consisting of nodes representing the modeled element instances and edges representing relationships between them. Henshin is built upon this foundation of model representation, whereas the name Henshin originates from Japanese and translates to *transformation*. It has been developed as a joint project by developers at the Philipps-University in Marburg, the Hasso Plattner Institute in Potsdam, the Technical University of Berlin, and others [4]. The first version of Henshin, 0.8.0, has been released in September 2011, and Henshin has been in development ever since. The current stable version is 0.9.8, which this Master's Thesis uses throughout the whole integration process.

Based on the theory of graph transformation[1], Henshin provides the functionality of transformating a given model based on defined Henshin rules. The tool itself is implemented for the Eclipse Modeling Framework (EMF) and can be used within this context. Like compilers in text-based tools, graph-transformation tools are trying to find a match on the Left Hand Side (LHS) and transformate this match based on a given rule into a result on the Right Hand Side (RHS). In figure 1.6 the LHS, denoted as L is transformed into the RHS denoted as R. A multiplication with 2 is transformed into an addition of the digit with itself. The corresponding match on the LHS and result on the RHS are denoted in dashed lines. This rudimentary definition and example are sufficient in this context, for a in-depth view on graph transformation and the theory behind [15] can be recommended.

For using Henshin the definition of Henshin rules is essential: Henshin rules are used as input language for the transformation process, additionally to the input model itself.

[1] Often also called graph rewriting

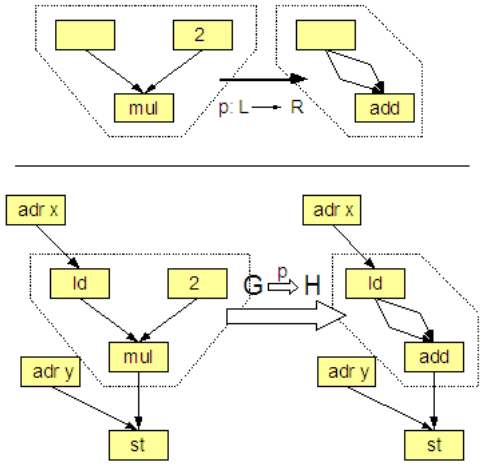


Figure 1.6: *Graph transformation example [22]*

Using the graphical Henshin diagram syntax, a defined rule looks similar to the example rule in figure 1.7. There are five types of nodes in a Henshin rule, whose semantics correspond to the five type of edges:

- **Create**
  A create node exists only in the RHS of the input model and is therefore created in the result model.

- **Delete**
  A delete node exists only in the LHS of the input model and is therefore deleted from the input model.

- **Preserve**
  A preserve node exists in both sides of the input model, LHS and RHS that is. These nodes are used as a match in LHS and are not altered in the result model.

- **Forbid**
  A forbid node, or Negative Application Condition (NAC), defines which elements are not allowed to be matched in the LHS for executing the transformation. If one forbid node is found in the input model, the transformation can not be executed.

- **Require**
  A require node, or Positive Application Condition (PAC), defines which elements

must be matched in the LHS for executing the transformation. If one require node is not found in the input model, the transformation can not be executed.

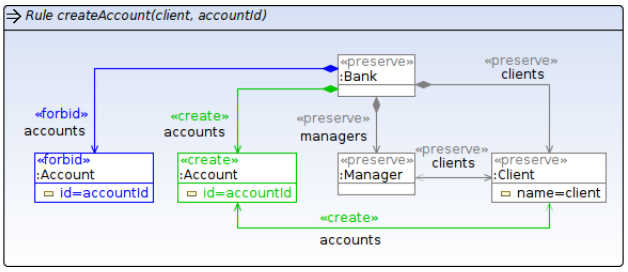


Figure 1.7: *Henshin rule example (1) [4]*

Using these types of nodes and edges, describing relationships between nodes, one can use Henshin for transformating a given input model. The example shown in figure 1.7 creates a bank account for a client by transformating a given input model conforming to a meta model describing a bank and its relationships to its accounts, managers and clients. The forbid node on the left represents the constraint that no two accounts managed by the same bank may have the same account id. The parameters which are given to the Henshin rule should also be mentioned, they can be a primitive type like a *String* or a object parameter and are used for the execution. The transformation engine of Henshin itself executes a Henshin rule as follows:

1. Search for a match in input model, defined via preserve nodes and their relationship to other nodes.

2. Check if no forbid node could be matched or a require node could not be matched.

3. Create all create / delete all delete nodes and edges describing their relationship.

A special feature of Henshin Rules are presented in figure 1.8: Nested rules. The delete node is marked with a star operator and therefore will be executed as many times as possible. In this particular example all accounts of a client will be deleted instead of only one as if not defined as nested rule. Internally the Henshin interpreter will try to match the preserve nodes and edges after each successfull execution and if it succeeds, it will execute the transformation again. This feature is important for later use and therefore is explicitly mentioned and explained.

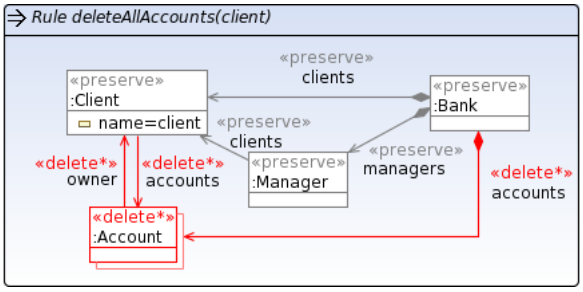


Figure 1.8: *Henshin rule example (2) [4]*

Using Henshin as a tool, it offers different ways of executing Henshin rules for transformation. They can be executed either via the Henshin graphical interface, or via the Henshin interpreter Application Programming Interface (API), which can be called using programming languages like Java. All described tools in this Master's Thesis are based on the latter option. Because Henshin offers more features than described in this section, publications of the developers of Henshin like [1] are recommended as additional lecture.

1.4 SiDiff

As mentioned in the introduction do modeling tools need to support features known from text-based tools. One important aspect for parallel development paradigms is the detection of correspondences and differences between two different revisions of software. In the area of MDSD software consists of models, therefore such features must be available in this context. The SiDiff tool environment [16] offers such functionalities: It can derive correspondences and differences between two models meta model-independent. The basic concept and tool implementation was released in 2004 by the Software Engineering Group (SEG) [20], whereas this version can be defined as proof of concept. The version in today's use was initiated in 2009, it has been revamped from scratch and implements a new software architecture built upon *OSGi* and the Eclipse plugin ecosystem. SiDiff is built upon a service-based architecture internally, which eases the integration of new services in any part of the computing pipeline including the matching itself.

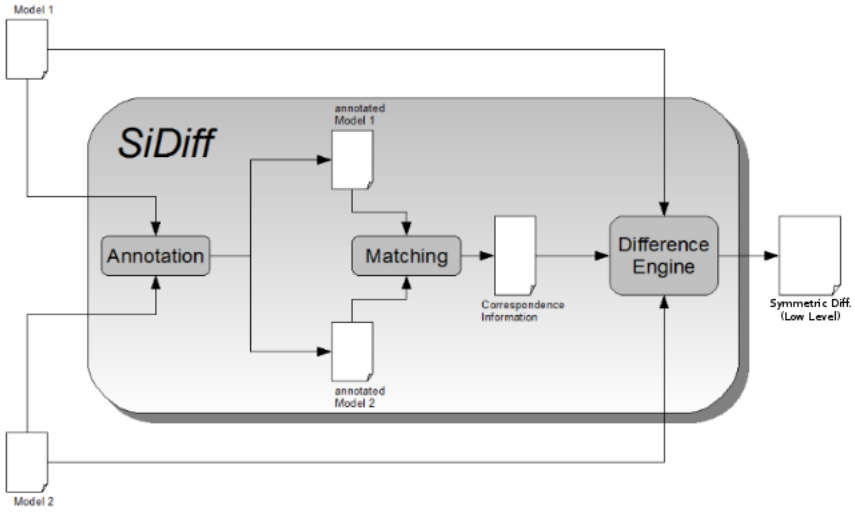


Figure 1.9: *SiDiff workflow [16]*

SiDiff make use of the computing pipeline depicted in figure 1.9: Two models are used as input for comparison and are annotated for later processing of the implemented matching services. One big advantage of SiDiff is its vast configuration possibility, which emphasizes its generic approach. In this step of pipeline the configuration defines which model elements are annotated and how this is done. The SiDiff matching services are then computing correspondences on the resulting annotated models, whereas the current available matching services consist of:

- **ID-based Matcher**
  This matcher makes use of given UUIDs, if available. Elements of Model 1 are corresponding to Elements of Model 2 if and only if their respective UUID is equal.

- **Signature-based Matcher**
  This matcher is based upon signatures of elements, which can be computed by taking attribute values or relationships into consideration. Which properties are used for the creation of signatures is defined via a configuration file.

- **Similarity-based Matcher**
  This matcher computes a similarity between elements, and if they reach a given threshold they are declared as corresponding. This part of SiDiff can be configurated in enormous detail, for example which element attributes are weighted in

comparison to others. This matcher and its configuration is to comprehensive to discuss in detail, a more detailed look is given in [8].

The matching result defines which elements of Model 1 are corresponding to which elements of Model 2 only, therefore a difference engine is executed afterwards for detecting unmatched elements. The final result is a symmetric difference, which consists of both the correspondences and the differences between the input models as seen in figure 1.10. The differences are presented on a low-level of abstraction.

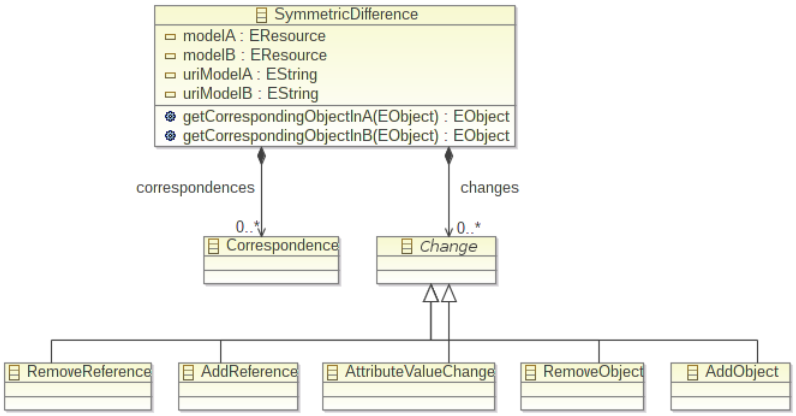


Figure 1.10: *Symmetric Difference (snippet) [16]*

The computed symmetric difference lays the foundation for other tools, which can now use the given information for their own advantage. One tool, which can use the symmetric difference computed by SiDiff as input is SiLift, which will be presented in the following section.

1.5 SiLift

A symmetric difference created generically presents its differences on a low-level of changes, which can be hard to comprehend by developers as they may differ significantly from the expected changes. SiLift, as the name suggests, lifts this low-level changes onto a higher abstraction level consisting of edit operations. SiLift itself is in development since 2011 [6] by the SEG [20] and is today implemented as proof-of-concept tool embedded into the Eclipse plugin ecosystem [5]. As SiDiff, Silift has taken an generic approach in

the support of modeling language and its software architecture.

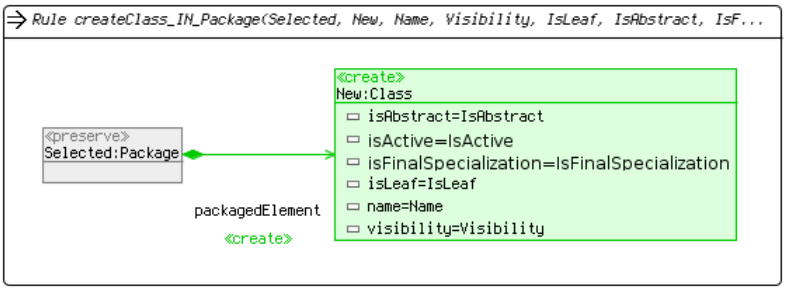


Figure 1.11: *Edit operation defined via Henshin rule*

The basic idea of the lifting process is to use Henshin rules as edit operation definitions and detect changes using these rules. A small UML example is shown in figure 1.11: The Henshin rule tries to match a *Package* and creates a *Class* within the detected package. The rule describes the edit operation of creating a class in a package. If this Henshin rule is executed successfully it will create a class in a package, corresponding to a edit operation executed by the developer. Such edit rules can be generated via the SiDiff Edit Rule Generator (SERGe) [19] or manually created, as depicted in the top of figure 1.12. The execution of a edit rule itself creating or changing the input model in the operation detection is not desired. Therefore the edit rules are transformed into recognition rules, which are Henshin rules themselves. They are responsible for the detection of a possible execution of the originated edit rule. There are two different types of edit rules:

Atomic edit rules

These rules are defined as *atomic* as they are essential for detecting all possible changes for this type of input model and can not be reduced to smaller pieces. They are mostly generated by SERGe.

Complex edit rules

These rules consist of 2 or more atomic rules, defining an even higher level of abstraction. They must be manually created, as they can not be deduced from the meta model of the input models. The creation requires a very good knowledge in the particular domain.

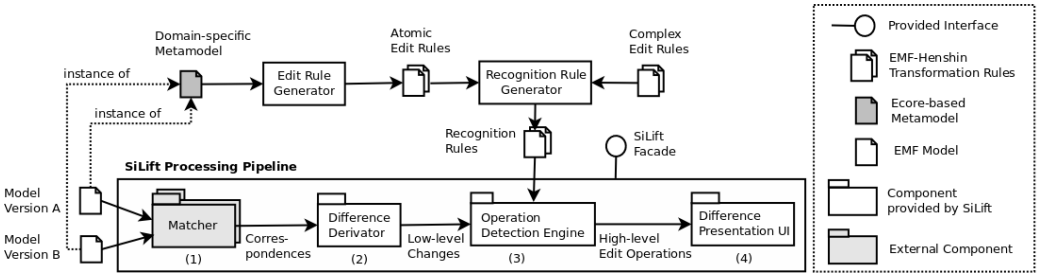


Figure 1.12: *SiLift work flow [17]*

As illustrated in figure 1.12, the SiLift processing pipeline consists of different stages:

1. **Matcher**
   This matcher can be exchanged by any matching service, for example EMF Compare or the more sophisticated SiDiff matcher explained in the preceding section.

2. **Difference Derivator**
   In this step the low-level changes are derived from the symmetric difference provided by the matcher.

3. **Operation Detection Engine**
   This part of the processing pipeline can be called the *core* of SiLift, as it implements the lifting of the low-level changes onto a edit operations level of abstraction. It uses recognition rules as input for the detection of these edit operations.

4. **Difference Presentation UI**
   Another shortcoming of low-level changes is the presentation of the mentioned. SiLift presents the now lifted changes visually to the user, for a more comprehensive understanding of the detected changes.

The final result of the whole SiLift processing pipeline is a lifted symmetric difference, which contains edit operations as changes. These changes can be interpreted as an directed asymmetric difference between two input models. Such an edit operation script, which is often called a *Patch*, can be used to perform these changes made between these models on another input model. The following section describes such a tool for the creation and application of a patch, consisting of edit rules as list of changes.

1.6 Patch-Tool

Considering that a symmetric difference has been created and lifted beforehand, the Patch-Tool now uses a asymmetric difference, which can be deducted from a symmetric difference, as input for creating and applying such a list of operations. The Patch-Tool has been developed by the SEG [20] in 2013, and is based upon the results from SiDiff and SiLift. As the other tools, the Patch-tool is also implemented in the Eclipse ecosystem and its architecture implements interfaces, enabling the exchange of the matcher beforehand or even the transformation engine.

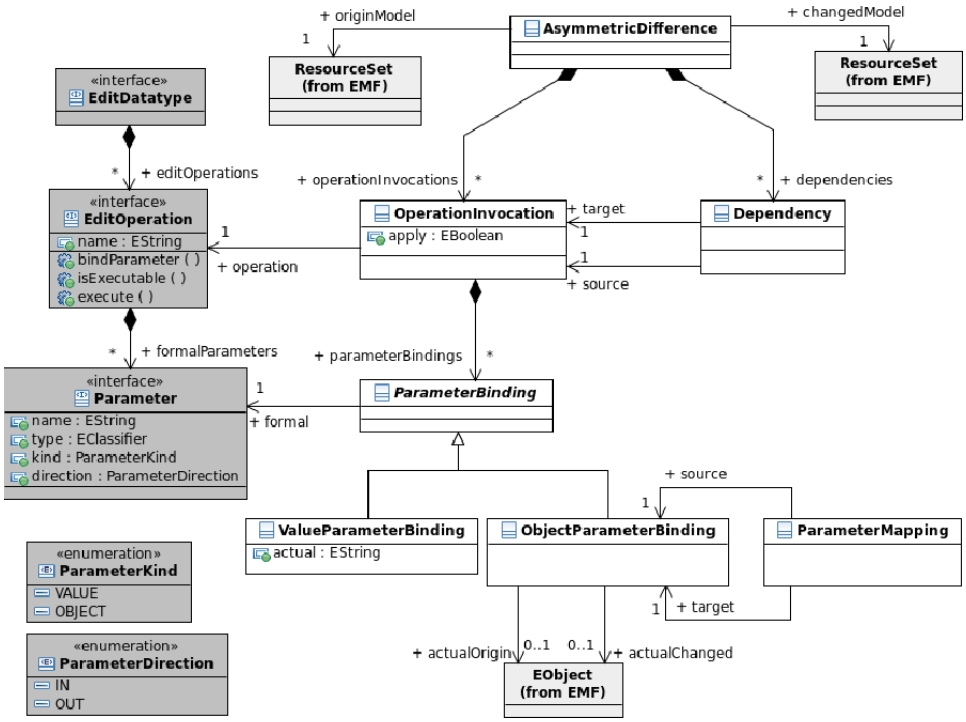


Figure 1.13: *Patch overview [7]*

As mentioned in section 1.5 is the execution of a Henshin rule corresponding to a edit operation executed by the developer. The Patch-Tool now orders the edit operations, also called Semantic Change Set (SCS), considering dependencies between them as seen in figure 1.13 [5]. One dependency can be used for demonstration purposes: *Create-Use*. If one wants to change the name of a *Class*, the class has to be existing previously. The operation for creating the class and setting its name therefore have got such a dependency between them. A more detailed look at dependencies, whether it be their

declaration or their detection can be glimpsed at in [5].

As depicted in figure 1.13 will the Patch-Tool create the binding of parameters by finding corresponding ones in the target model. The final result is a ordered list of *OperationInvocations*, which have been constructed with corresponding parameters. After the creation of such a patch it is then applied to the target model, which will transform the model according to the patch using the defined transformation engine, Henshin in this case.

Chapter 2

Integration of UML Profiles

The following Chapter illustrates the concepts of the UML profiles integration into and with the aid of the tools introduced in the previous chapter. To achieve this, first an introduction is presented, which will show the basic ideas for the later sections in this chapter. Additionally an overview of the integration process is given. This chapter concludes by presenting the whole integration process in detail, which each tool has undergone.

2.1 Introduction

Supporting as many modeling languages and therefore domains as possible is crucial to all modeling tools, like the ones presented in chapter 1. The shift to MDSD can only be accomplished, if modeling tools providing needed features are supporting modeling languages of interest in practical use. The UML itself is very popular among developers because of its generic approaches and its extensibility. As seen in section 1.1 has UML taken the generic approach of profiles as solution for integrating new domains and languages into modeling tools, without the sacrifice of new tools needed.
Supporting UML profiles is therefore an important step for each modeling tool in practice. The following sections describe how this integration is done in the previously presented modeling tools SiDiff, SiLift and the Patch-Tool developed by the SEG.

2.2 Overview

For a full UML profiles integration through all tools used in the SEG, each pipeline step has to be taken into consideration:

- Profile support in **Matching** service
- Profile support in **Lifting** service
- Profile support in **Patching** service

An overview of the tool pipeline is presented in the center of figure 2.1.

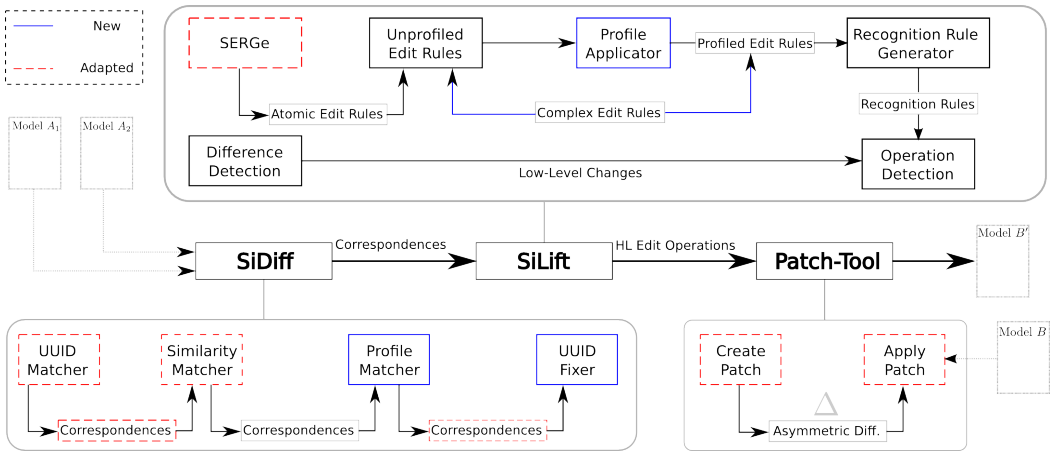


Figure 2.1: *UML profile integration overview*

The tool pipeline can be described coarsely via the following steps:

1. Two models A_1 and A_2 are given as input models.
2. SiDiff creates a symmetric difference between them.
3. SiLift lifts the low-level changes to more meaningful edit operations.
4. The Patch-Tool deduces a patch and applies it to a target model B.
5. The final result is a patched model B' containing the changes detected between model A_1 and A_2 in model B .

As demonstrated in figure 2.1, every tool has been adapted for supporting UML profiles, whether it be just modifications on existing tools and services or creation of new ones. The concepts of this adaptions will be presented in the following sections.

2.3 SiDiff Integration

The integration of UML profiles into SiDiff and adaptions in general can be divided into the following parts:

UUID Matcher
The SiDiff matching pipeline has been adapted for better matching results if elements of the input models contain UUIDs. Instead of just using one of the matching services described in section 1.4, the UUID matching service is executed before the similarity-based matching service. Therefore additional correspondences are already existent for the following matching part, as shown in figure 2.1 in the SiDiff box with dashed lines on the left side. Because of additional correspondences at the start of of the similarity-matcher, it can deduce better matching results. The already computed correspondences are taken into consideration and are used as *fix points* for the remaining unmatched elements. This adaption is not specific for UML profiles, but has been implemented in this Master's Thesis.

Similarity Matcher
For the later concepts of the UML profile integration and better matching results in general, the similarity-based matcher has been adapted via its configuration files. The focus was the definition of SiDiff configurations especially for UML itself, which can be used in other contexts as well. A snippet of such a configuration is depicted in listing 2.1.

```
1  <Class name="Class" threshold="0.4">
2    <CompareFunction class="Self"
3      parameter="ECAttributeStatic[VCStringLCS[ci];name]" weight="0.5" />
4    <CompareFunction class="Parent" parameter="ECMatchedOrSimilar"
5      weight="0.1" />
6    <CompareFunction class="Children" weight="0.4"
7      parameter="SCGreedyMatchedOrSimilar" />
8  </Class>
```

Listing 2.1: *Snippet of UML SiDiff configuration*

In this example the matcher is configured for an UML *Class* element, which will be matched after its attribute *name*(l2-l3) and its relationship to its *parent* and *children*(l4-l8). By adjusting the weight between the configured elements, the results of the matching may improve or get worse. This is just a small example of the vary configuration possibilities SiDiff has to offer. For a more detailed look into the possible SiDiff configuration parameters concerning the similarity matching service contacting the SEG via [20] is suggested. As the adaption of the UUID Matcher, the improvement of the similarity-based matching configuration for UML is independent from the integration itself.

Profile Matcher

Profiled elements, conforming to the additive manner of UML profiles, are constructed as shown in figure 2.2: The base element taken from UML, in this case *Class*, is extended by a newly defined element, *Block* in this case. Their relationship is defined via the *base_Class* reference, which is a particular reference name used by UML profiles.

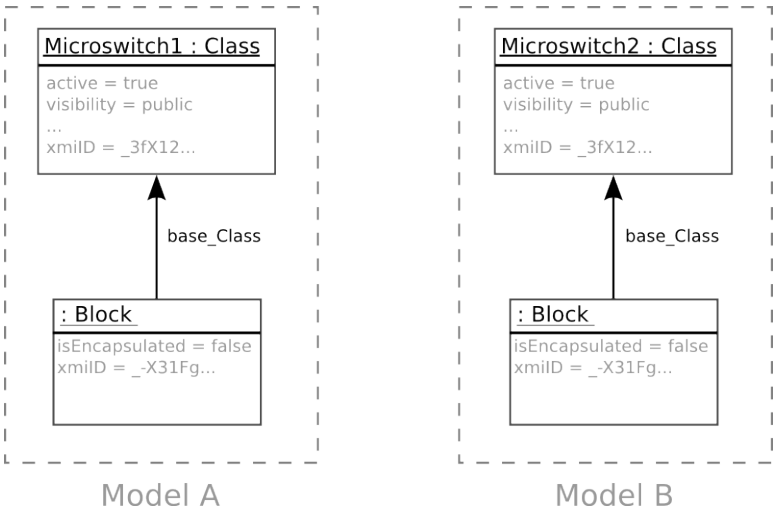


Figure 2.2: *Example of two profiled models*

To describe the concept behind the developed profile matcher, two scenarios concerning the example including model *A* and model *B* in figure 2.2 can be imagined:

a) All elements contain UUIDs and they are corresponding to each other.

b) One or all elements own a different UUID or none at all.

In case a), which is presented in figure 2.2, the newly integrated UUID matcher will

match corresponding elements. The following execution of the similarity-based matcher will not add any new results as already all possible correspondences have been found in the earlier matching phase. Considering case b), in the similarity-based matching phase there are unmatched elements because of wrong or missing UUIDs. Newly introduced elements like *Block* might be unmatched, in which case SiDiff needs a feature to match those elements. To match profiling elements, there are two possible solutions at hand:

1. **Making use of the similarity matcher**
   Whereas at first sight this options seems more appropriate, the usage of the similarity-based matching service depends highly on its configuration. As explained beforehand, this configuration defines weighted elements and attributes which should be taken into consideration for matching results. If one element does not have much of its own semantics which can be considered as important properties, the similarity matcher can not deduce meaningful matches, which leads to wrong or missing correspondences between two models. Taking figure 2.2 into consideration, a element of the type *Block* does only define one boolean as own additional attribute. Comparing this attribute of model A, describing the additional semantics of such *Block*, to model B one can clearly see that this may lead to wrong correspondences. The only striking property of a *Block* is the reference to its base element. As such reference is always existent in UML profiling elements, the needed configuration of the similarity matcher seems a unnecessarity. For each UML profile and its elements such a configuration part needs to be created, whereas they would contain the same semantic content presented in listing 2.2.

```
1  <Class name="Block" threshold="1.0">
2  <CompareFunction class="Parent" parameter="ECMatchedOrSimilar" weight="1.0"/>
3  </Class>
```

Listing 2.2: *SiDiff configuration for profiling elements*

2. **Implementing a new matching service**
   Instead of using an available matching service, an additional matcher can be introduced for supporting UML profiles. As presented in figure 2.1 an additional matcher for UML profiles has been developed and will be executed after the other matching services provided by SiDiff. The basic idea behind the profile matcher is to use all correspondences already created by other matchers and deduce results concerning profiling elements from the former.

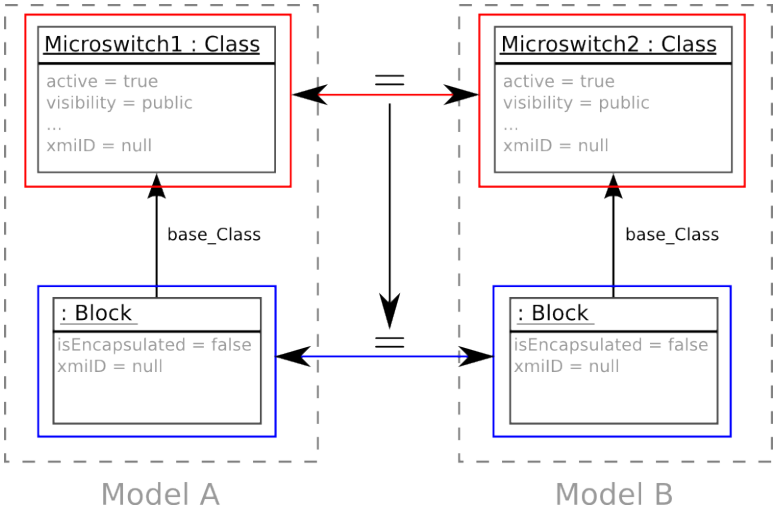


Figure 2.3: *Concept of implemented profile matcher*

The concept of the introduced profile matcher is shown in figure 2.3: The red marked boxes and correspondences have been created through the matchers prior in the SiDiff pipeline like the UUID matcher or similarity matcher. If two elements are corresponding, which serve as base elements for profiling elements like in this case, the profiling elements marked in blue are also considered as corresponding. In this case the UML configuration for SiDiff is crucial for good matching result, as the semantics of the base elements is used through the similarity matcher for matching the profiling elements. The profile matcher itself needs only a very minimal configuration effort, instead of the first variant using the similarity-based matching service. A detailed explanation of how this service is implemented and configurated can be found in section 3.1.

UUID Fixer

As the first two adaptions described in this chapter, this additional service can be separated from the integration of UML profiles. The UUID fixer service, as the name suggests, has been created for fixing wrong UUIDs between corresponding elements. An explanatory example for a use case of the new implemented service is presented in figure 2.4: Two corresponding *Associations*, namely Association2 and Association3, own different identifiers. This can happen if an element has been deleted by the developer and afterwards created newly again, representing the same semantics. Many modeling tools rely on UUIDs and their correctness, whereas the similarity matcher of SiDiff does not and creates such a correspondence correctly. The new UUID fixing service now corrects all

UUIDs which are mismatching, in this case it would replace the identifier of Association3 with the one of Association2. Whether to change the UUIDs the other way round is arguable, but the premise is to have two consecutive revisions of software. In such a particular case, the identifiers in the newer revision should be fixed instead of the ones in the older revision, which would ruin all comparison possibilities with even older revisions.

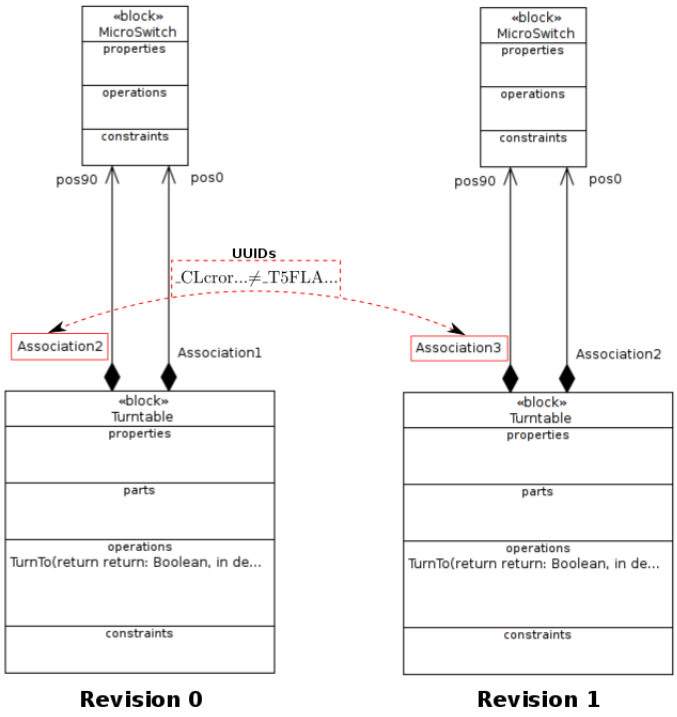


Figure 2.4: *Wrong UUID example*

The service itself can be executed at any time considering the matching pipeline, whereas its results are the best possible if executed at last. By using this new service SiDiff itself offers a new feature, which has been absent beforehand: Instead of supporting only matching of elements, SiDiff now can fix wrong models and therefore make them yet again compatible to other tools, which depend heavily on correct UUIDs.

For using all adapted and newly created tools together described in this section, one can image following steps referring to the example in figure 2.2, whereas the identifiers of the *Block* elements should differ:

1. Both classes are matched by the UUID matching service as they own the same identifier.

2. The similarity-based matching service will not find any additional correspondences, because blocks are not part of the UML configuration for SiDiff.

3. Using the profile matcher, both blocks are matched because their respective base element *Class* has been matched beforehand.

4. Finally, the UUID fixing service is executed and will replace the identifier of the *Block* in model B with the one of model A.

Afterwards a generic UUID matcher is capable of matching all elements, including profiling ones like *Block* in this example.

2.4 SiLift Integration

The integration of UML profiles into SiLift translates to an integration of UML profiles into Henshin edit rules. If these rules are providing support for profiled elements, then SiLift provides this support as well. As already seen in figure 2.2, do profiled elements essentially consist of themselves besides the relationship to their respective base element. These two components need to be integrated into Henshin edit rules, which can then be used by SiLift for its purposes. To achieve this goal, three different variants can be deduced which are presented in figure 2.5 using SysML as applied UML profile.

Variant 1 **Integration into SERGe**

SERGe is already capable of generating atomic edit rules by analyzing the corresponding meta model, including all *set*, *create* and *delete* edit operations possibly executable by developers. Although UML itself is supported by SERGe, the profiling mechanism is not. The generator has to be adapted, to generate atomic edit rules, which contain the base elements as well as the profiling elements. The crucial advantage of this variant is the automated process of generating edit rules, whereas on the other hand new manually created edit rules can not be covered by this variant. Because of this drawback this variant has not been implemented in this Master's Thesis.

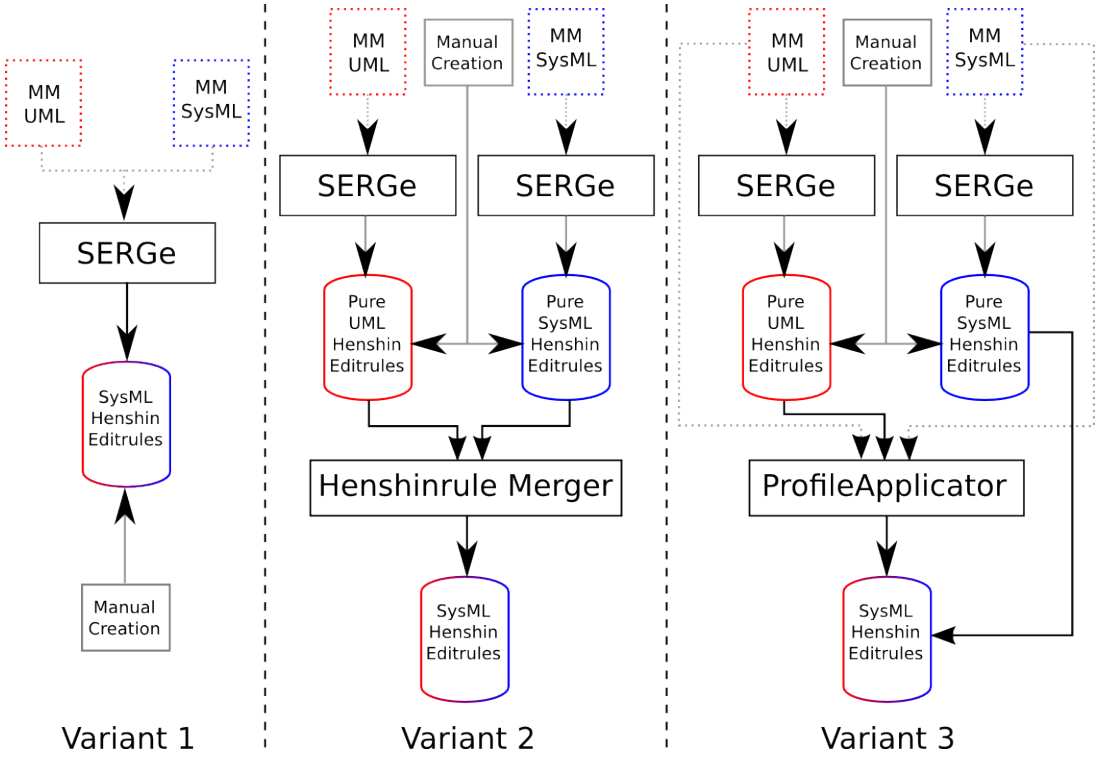


Figure 2.5: *Overview of different integration variants*

Variant 2 **Merging of Henshin rules**

Another feasible solution is the merging of Henshin rules. The concept behind this idea is to take two rules, one containing only the base element and its properties, the other containing only the profiling element, and merge them into one resulting rule. Whether both rules have been generated or manually created does not concern, the drawback of the first variant is therefore absent. One difficulty to overcome in this variant is the merging of the rules itself: Finding the corresponding intersections between two rules as well as mapping parameters between them is complex in detail. Implementing this variant generically which does not only support the merging of Henshin rules for UML profiles but rather supports the merging functionality for such Henshin rules in general makes this variant substantial sophisticated. Another mentionable disadvantage is the manual intervention for merging rules: The tool user has to be an expert on the used modeling domain, as he must define which rules should be merged into each other. These disadvantage have led to the following variant as solution, whereas this variant could be taken into consideration for future work as presented in chapter 6.

Variant 3 **Adding profiling elements to existing Henshin rules**

This variant has been implemented as part of this Master's Thesis as a service called *ProfileApplicator*. Instead of merging two edit rules into one as presented in variant 2, one pure edit base edit rule is taken as input and transformed into a profiled one, containing profiling elements. This variant offers the same advantage like variant 2, the possibility of profiling manually created edit rules that is. Another advantage is the reduced complexity of development compared to the previous variant. Contrary to the merging variant, in this variant no manual intervention is needed in the creation process of the profiled rules, only a minimal configuration is needed for the ProfileApplicator to work instead. The concept of this variant is described in detail in the following section.

ProfileApplicator

Adding profiling elements to a given model, which can be a Henshin rule itself, can be achieved via a transformation. As Henshin is used in SiLift extensively it has been chosen as the tool of choice for transformating such models. A Henshin rule itself is a model, and therefore can be transformed like any other input model. Such transformation is called HOT, as it uses Henshin rules to transform Henshin rules. An introductory result example of the execution of such an HOT is depicted in figure 2.6: The given edit rule on top contains only the base element *Class*, whereas the execution of a HOT adds the profiled element *Block* to it resulting in the edit rule shown below.

This concept is implemented in the tool *ProfileApplicator*, which executes such HOTs for transforming given edit rules, such as UML edit rules. As explained in section 1.3, do Henshin rules consist of three types of nodes: Create, delete and preserve. For each node type a HOT has been implemented, they can be found in appendix A. One exemplary rule covering the create nodes depicted in figure A.1 will be explained in the following paragraph.

Assuming the input model is an instance of a Henshin rule, the HOT matches a Module, which consists of one *Unit* and one *Rule*. Parts of name and description attributes are replaced by a given String, this is just a convenient renaming to identify profiled result rules more easily. Additionally will the HOT match a create node contained in the RHS graph and equals to the defined *base type*, which is deducted from the meta model of the UML profile. If successful the following elements will be created by this HOT:

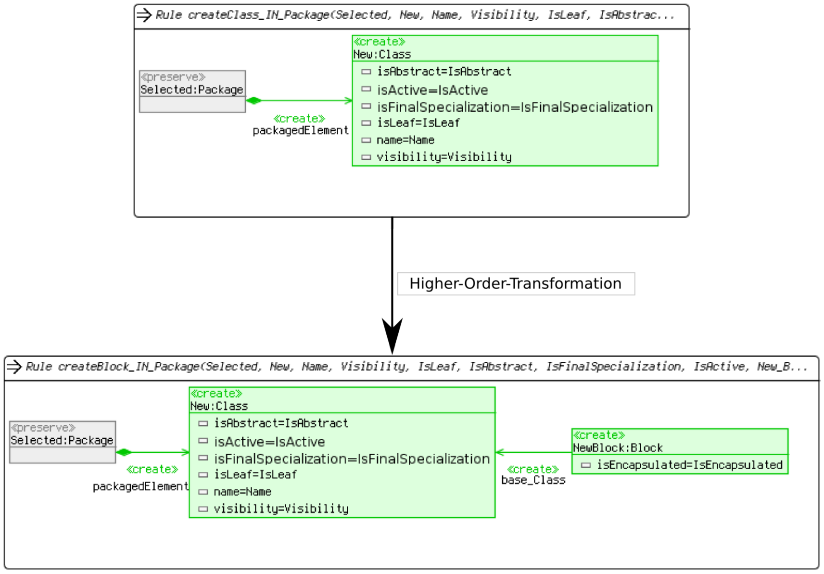


Figure 2.6: *HOT example result*

- A Create node of type *stereo type* which has been derived from the meta model.
- An edge between the newly created node and the matched node of type *base type*.
- All attributes of the stereo type node, except the ones defined as *unchangeable*, *derived* or *transient* using a nested rule.
- New parameters in both the Henshin rule and unit.
- A parameter mapping between these two parameters.

The forbid node and edges are required for multiple executions of this HOT, as there shall only be one profiled element of the same type be added to the base element at most. If this HOT is applied to the edit rule shown in the top of figure 2.6, the rule below will be the result.

The remaining two HOTs are defined in an similar manner, whereas some differences can occur. The transformation of preserve nodes has to take also the LHS graph into consideration for example. Which base types will be used as parameter can be configured in detail as will be seen in section 3.3.

Manually created edit rules

Although SERGe does generate atomic edit rules by analyzing the corresponding meta model, not all rules containing the semantics are resulting. One example can be given in the area of the UML concerning its representation of associations. An association is represented by two properties, which are either owned by the classes or by the association itself. Corresponding to optional relationships the meta model does not restrict the multiplicities, which results in *wrong* generated edit rules. To cope with such special cases, one has to manually create such edit rules for themselves. One of the resulting rules is depicted in figure 2.7: This edit rule is corresponding to the creation of an association which is navigable in one end. Such an association is defined via its properties, whereas one is owned by the corresponding class and the other is owned by the association itself. As part of this Master's Thesis different atomic manual rules have been created, which represent the semantics of the UML meta model.

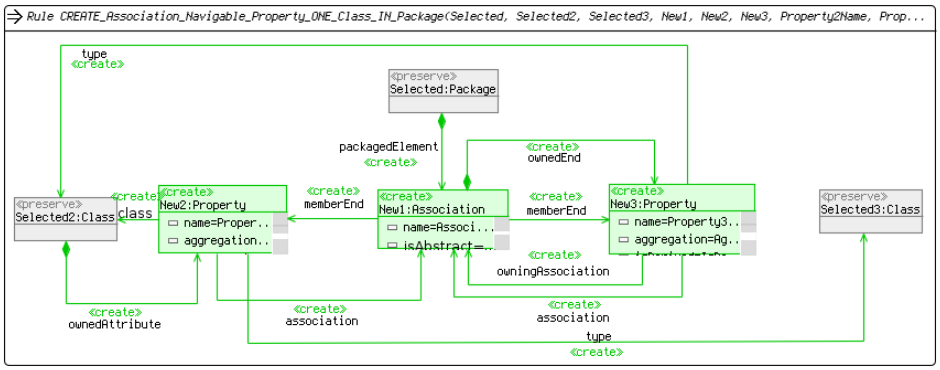


Figure 2.7: *Manual atomic edit rule example*

Additionally to atomic edit rules, a domain expert can create complex edit rules, which have been defined in section 1.5. Adding complex edit rules, containing atomic edit rules, results in a better lifting possibility, which aids the developer at understanding the model more easily, as less edit operations are presented. One of the created rules is depicted in figure 2.8. As a special feature, this complex edit rule is modeled in SysML instead of being modeled in UML and transformated using the *ProfileApplicator*. The reason for this is the added attribute *direction* of the profiling element *FlowPort*. This allows to define the direction of a given port and is a brilliant example for adding new semantics to known elements. This complex edit rule covers the creation of an interacting *Block*, which is connected via its flow ports. The semantic behind this kind of block is that it receives a type of element, processes it in any way and passes the block on to another

element connected via the outgoing port. This type of block is often used in the SysML domain, as it may contain mechanical elements corresponding to this semantics.

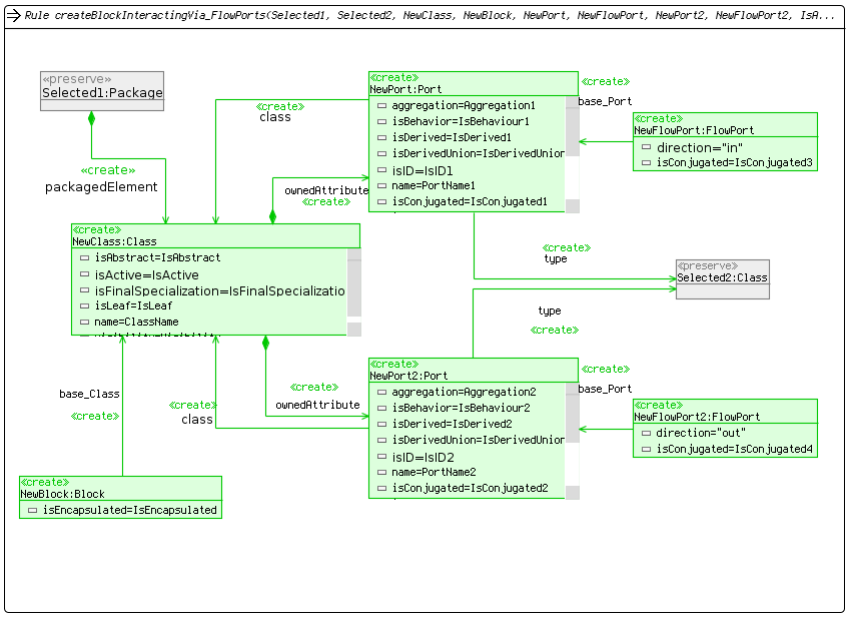


Figure 2.8: *Manual complex edit rule example*

After manually creating missing edit rules and transformating all UML atomic and complex edit rules into profiled ones the defined rule base for UML profiles is complete. Now every possible edit operation in a particular UML profile like SysML can be lifted with SiLift.

2.5 Patch-Tool Integration

Now that the first two steps of the tool pipeline (fig. 2.1) has undergone the UML profile integration, the final step has to be taken into consideration, the adaption of the Patch-Tool that is. The Patch-Tool is responsible for the following three features, which have to be adapted:

1. **Creation of a patch**
   The creation of a patch depends on the completeness of the given rule base consisting of edit rules as well as the dependency correctness between them. The latter

as well as the former has successfully been solved through the manual creation of rules missing or describing wrong semantics.

2. **Application of a patch**
   The successful application of a patch can only be ensured, if all operation parameters can be resolved and the order of operations is correct concerning the dependencies among them. Whereas the latter has already been solved through the last feature, the former can be ensured if the matching is done via SiDiff, including the UML configuration and profile matcher. Using these services a matching between all corresponding elements can be created and therefore the parameter bindings.

3. **Correctness of the result**
   The last feature is the validation of the correct application of the patch. For this purpose the models A_1 and A_2 are used for the creation of such patch. The patch is then applied to A_1 and the result B' is compared to A_2. If and only if $A_2 = B'$ the patch has been applied correctly and the result model contains all changes included in the patch. A graphical overview of this situation is presented in figure 2.9.

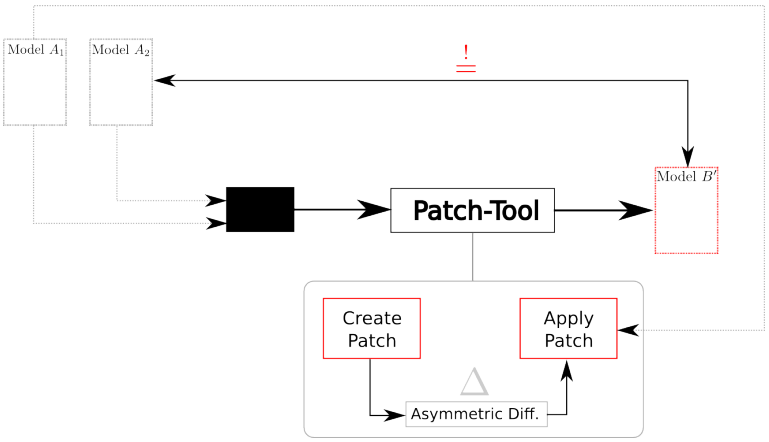


Figure 2.9: *Patch result correctness condition*

Such correctness can be only be tested as final step by using model instances, like done in chapter 5.

UML profiles are now fully integrated in all pipeline steps(fig. 2.1) and can be used consecutively. This chapter shall describe the concepts only instead of presenting the results using a real SysML case study. The results are explained extensive in 5.

Chapter 3

Realization of Concepts

Based on the concepts presented in the preceding chapter, this chapter describes the implementation process of the newly created tools and services. Problems and details specific for the implementation part such as complexity analysis or architectural features are illustrated. The usage of these implemented services and tools in the environment of the SEG is additionally focused.

3.1 ProfileMatcher

For supporting the matching of UML profiles in SiDiff a new service has been introduced as part of this Master's Thesis. As shown in figure 3.1 the matcher itself is executed after all other SiDiff matching services. The implementation itself is based on the Eclipse plugin architecture, as the whole SEG tool pipeline is evolved around this ecosystem.

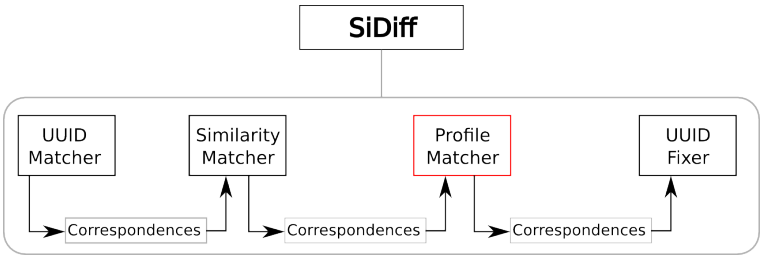


Figure 3.1: *ProfileMatcher tool integration overview*

To integrate this service into tools using SiDiff, only a minimal change of code has to be done by the tool engineer. Complying to the SiDiff service architecture, the *ProfileMatcher* can be integrated easily: As described in section 2.3 does the *ProfileMatcher* rely on computed correspondences, therefore the SiDiff *CorrespondenceService* is crucial for this matcher to work. Assuming this service has been registered and is available for usage all needed lines of code to integrate and execute the new matcher afterwards are depicted in listing 3.1.

```
1   //Define configuration file
2   private final static String profileFileName = "uml.profiles.xml";
3
4   //Configure ProfileMatcher according to configuration
5   ServiceHelper.configureInstance(context, ProfilesMatchingService.class,
6       UML_URI, null, profileFileName);
7
8   //Register SiDiff service
9   ServiceContext.putService(ServiceHelper.getService(Activator.context,
10  ProfilesMatchingService.class, documentType, ServiceHelper.DEFAULT));
11
12  //Execute ProfileMatcher service
13  ServiceContext.getService(ProfilesMatchingService.class).match();
```

Listing 3.1: *ProfileMatcher service integration*

As explained in detail in section 2.3 one advantage of the implemented matcher is the reduced effort needed for the configuration of the service. The configuration syntax is based on other SiDiff configurations like the one of the similarity matcher. The configuration of profiles is done like depicted in listing 3.2.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no" ?>
2 <Matching>
3   <Settings basePackage="http://www.eclipse.org/uml2/4.0.0/UML" />
4   <Profile name="SysML" stereoPackage="http://www.eclipse.org/papyrus/0.7.0/SysML"/>
5 </Matching>
```

Listing 3.2: *ProfileMatcher configuration example*

One configuration file can hold multiple UML profiles such as the one defined in line 4. Each profile can be configured in detail if necessary by creating a white list of profiling elements, whereas the absence of such list will use all existing profiling elements contained

in the given UML profile, like done in listing 3.2. The matcher will iterate through all given profiles and match their white list elements accordingly to their meta model automatically, therefore no additional configuration is needed. The execution of the profile matching service will engage the following actions for each configured profile:

- Read the given configuration file and analyze the corresponding meta model according to this configuration.
- Save all profiling elements, their corresponding base element and the relationship reference between them in a map.
- Build a map between both the profiling elements as well as the base elements.
- Iterate through all correspondences and search for base elements.
- Add a new correspondence to the respective profiling element if such base element is found in the correspondence as well as the built map.

3.2 UUIDFixer

As the ProfileMatcher described in the preceding section, the UUIDFixer is implemented using the Eclipse plugin architecture and makes use of the SiDiff service environment.

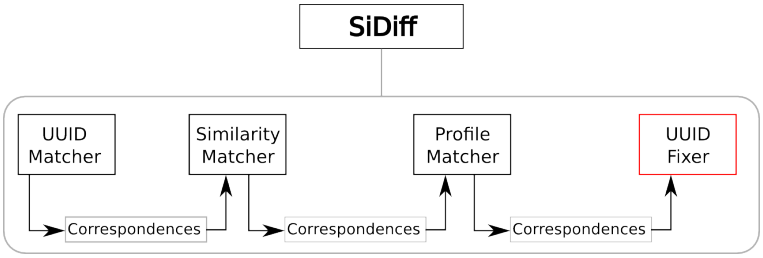


Figure 3.2: *UUIDFixer tool integration overview*

The integration part of this service is even more easily to be done, as there is no need for any configuration at all(listing 3.3). As the ProfileMatcher, the UUIDFixer makes use of the *CorrespondenceService* and is therefore most effective if executed as last SiDiff service, as at this time more correspondences could be available.

```
1  //Register SiDiff service
2  ServiceContext.putService(ServiceHelper.getService(Activator.context,
3  IDFixerService.class, null, null));
4
5  //Execute UUIDFixer service
6  ServiceContext.getService(IDFixerService.class).fixIDs();
```

Listing 3.3: *UUIDFixer service integration*

The id fixing service algorithm can be divided into the following parts for each model element A:

1. Get corresponding element B of current element A, if any.
2. Compare the UUIDs of both elements A and B with each other.
3. If they are not equal, replace the identifier of B with the one of A.

As already denoted earlier, the replacement of the UUID is done in B as it is the *newer* model which is the one to fix. This convention originates from the repository domain, as the fixing in A would destroy possible correspondences between A and earlier revisions.

3.3 ProfileApplicator

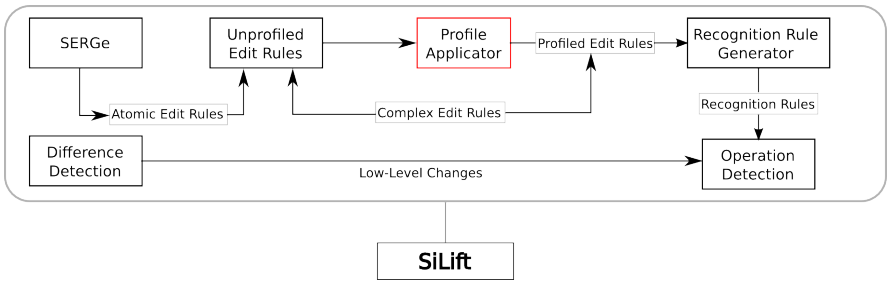


Figure 3.3: *ProfileApplicator tool integration overview*

Contrary to the previous services the *ProfileApplicator* has been implemented as an OSGi application, as it is used as standalone tool in the SiLift context. The software architecture internally is similar to the one of SERGe, as they are used in the same manner and often even consecutively. As SERGe itself, the *ProfileApplicator* has its own

configuration for a detailed adaption by the tool engineer, whereas the semantics of the configuration shall be explained in detail. Considering the profile settings depicted in listing 3.4, the following configuration possibilities are available:

- **Name**
  This defines a human readable name for this UML profile, which serves as debug output and more understandable output possibilities in general.

- **BaseTypeInstances**
  As the name suggests, does this option toggle the possibility of instances containing only the base element. As example the *Class* / *Block* relationship seems appropriate: If the option is set to *false*, there shall be no *Class* element without a profiling element *Block*, whereas if set to *true* the result will be two edit rules, one containing only the *Class* element, the other containing both.

- **BaseTypeInheritance**
  This option defines whether the ProfileApplicator should only take the direct relationship and its corresponding base element of a give profiling elements into consideration. Using SysML as profile, the defined element *RequirementRelated* is added to every *NamedElement*. Disabling BaseTypeInheritance would result in no additional rule, as *NamedElement* is abstract and therefore cannot be instantiated. Enabling would result in multiple result rules, as many concrete elements are of the type *NamedElement*, for example the *Class* element.

```
1  <ProfileSettings>
2    <Profile name="SysML" />
3    <BaseTypeInstances allow="true" />
4    <BaseTypeInheritance allow="true" />
5  </ProfileSettings>
```

Listing 3.4: *ProfileSettings configuration example*

For additional adaption possibilities, the configuration part presented in listing 3.5 is available: The tool engineer can decide which HOT will be used by the *ProfileApplicator*, declaring which type of Henshin nodes will taken into consideration for transformation.

```
<Transformations>
  <Transformation name="CREATE" apply="true" />
  <Transformation name="DELETE" apply="true" />
  <Transformation name="PRESERVE" apply="true" />
</Transformations>
```

Listing 3.5: *Transformations configuration example*

Like the white list implemented in the profile matching service, does the *ProfileApplicator* also support this functionality. If none profiling element is defined, all elements of the given UML profile will be used. If on the other hand at least one has been added, only these are used for transformation. This configuration style has been used in SERGe as well and therefore has been implemented this way.

The execution of the tool needs additionally to the configuration explained above two parameters, which can be defined by the user via an input dialog presented by an OSGi execution configuration:

- Folder of base type Henshin edit rules to transform (e.g. containing UML edit rules).
- Target folder for saving the resulting transformed edit rules (e.g. SysML folder).

While implementing the *ProfileApplicator*, various runtime problems have arisen, which needed a feasible solution. The transformation tool used by the profile application tool is Henshin, which itself relies on Ecore [3] for internal model representation. Each *EObject* loaded will register an *Action Listener*, which is capable of detecting changes concerning this object. Using the functionality on default, each *EObject* will contain such listener and each element referring to this EObject will create a *CrossReferenceAdapter*. Using large meta models like UML will lead to huge numbers of such adapters and listeners between all elements. This will lead to an exponential growth while executing the ProfileApplicator, which therefore will not finish its transformation in finite runtime. The solution is presented as listing 3.6: All elements contained in an *EGraph* used by Henshin will be deleted manually, therefore all adapters and cross references will be removed as well.

```
1  public void releaseAdapters(EGraph graph) {
2      for (EObject roots : graph.getRoots()) {
3        graph.removeGraph(roots);
4      }
5  }
```

Listing 3.6: *Deleting all EObjects manually*

The deletion process uses some computation time itself, but the the final runtime of the tool is constant and therefore ends in finite runtime. Deleting these objects manually leads to the result shown in figure 3.4 taken while executing the profile application.

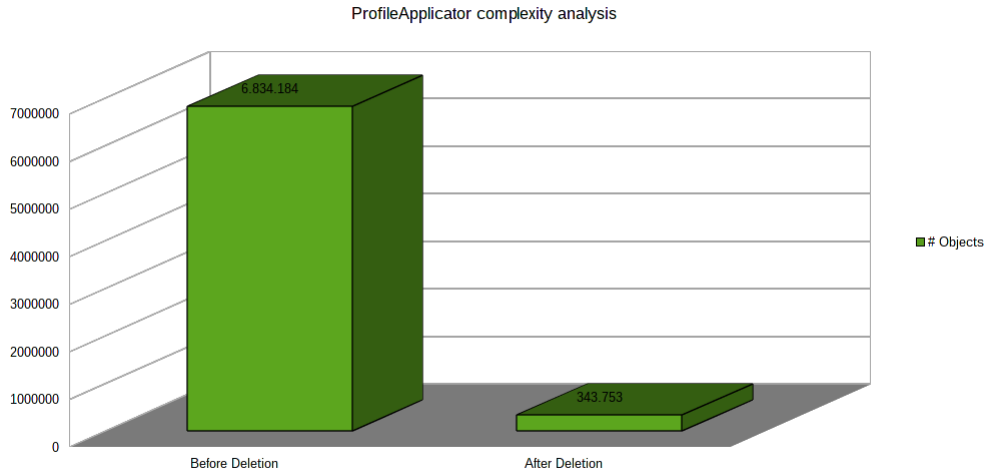


Figure 3.4: *Objects contained in an EGraph*

Another runtime problem concerning the used *EGraph* itself has arisen during development. By default the constructor of such a Henshin EGraph will resolve all object references of the given input model and will add the referenced objects additionally to this EGraph. Given a large model like UML as input results in an enormous graph, which must be searched during the execution of a Henshin rule. Henshin tries to match the preserve nodes as previously described, whereas the number of nodes has a large impact on runtime. Adding only the used elements into the EGraph manually instead of using the default constructor leads to a considerable smaller EGraph and therefore runtime. The code snippet of this solution is depicted in listing 3.7. This solution reduced the EGraph size from 7463 to 27 for example, which drastically effects the runtime.

```
1  // Create Module EGraph and its children as working copy
2  workGraph = new EGraphImpl();
3  workGraph.addTree(workResourceSet.getModule(this.sourceFile.getName()));
4
5  // Add all important elements for matching
6  workGraph.add(applicator.getStereoPackage());
7
8  // Add Stereotype and its Attributes
9  workGraph.add(stereoType.getStereoTypeClass());
10 for (EStructuralFeature feature : stereoType.getStereoTypeClass().
     getEAllStructuralFeatures()) {
11   workGraph.add(feature);
12 }
13
14 //Add Basetype and baseReference
15 workGraph.add(baseType);
16 workGraph.add(stereoType.getBaseTypeMap().get(baseType));
```

Listing 3.7: *Manually adding needed elements to an EGraph*

For further runtime improvement the ProfileApplicator implements the Java threading technology as using threads in today's multicore environments seems appropriate. Each profile applicator thread transforms one input Henshin edit rule, therefore no concurrency problems can arise. Each input edit rule is defined as work package which needs to be transformed. It is placed in a work pool, whereas each thread can remove such a package for transformation. As the number of threads used for computation are configurable via an execution parameter, each tool user can adapt this technology to his own needs. Having each thread computed parallel reduces the runtime by a factor of threads e.g. having 4 threads executed on a quadcore processor leads to a runtime reduction by a factor close to 4.

Chapter 4

SysML Case Study

The following chapter introduces a real world example of the UML profile SysML in a detailed manner. At first an introduction of the case study is given considering the semantics of the corresponding models. Afterwards the case study is analyzed regarding technical and pragmatical issues and their relevance related to modeling tools. This chapter lays the foundation for chapter 5, in which the presented case study is used as exemplary input.

4.1 Introduction

A real world case study concerning the domain of systems engineering presented by the technical university of Munich in [10] has been modeled in SysML, as this UML profile has been developed with such domain in mind. The case study is evolved around a PPU, which can be described as an industrial automation plant. Its main purpose is the processing of given elements like metallic pieces by picking them up and placing them onto a slide. This scenario lays the foundation for the following revisions of this case study and is depicted in figure 4.1. The whole case study pursues the target of demonstrating different iterations of a real world automation PPU, whereas the developers of this plant will add, remove or edit elements throughout the revisions.

Revision 1, as shown in figure 4.2, adds a new *Y-Slide* to the PPU which will replace the simple slide of revision 0 and has to be integrated into the semantics of the crane additionally for example. Now the PPU fills up both parts of this new slide as it increases the capacity for work pieces. This is just a small example of the changes between each

revision, a more in-depth look at all revision changes can be taken in the appendix B or in the publication [10].

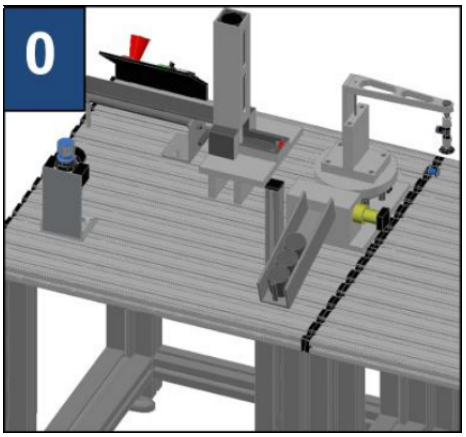


Figure 4.1: *PPU case study revision 0 [10]*

The case study itself consists of 14 consecutive revisions each representing a new version of the PPU consisting of changes made by the developers. The scope and the differences between each revision is presented in figure 4.3: Revision 0 consists of approximately 550 elements which are corresponding to revision 1 for example. As depicted via the blue graph, elements are mostly added which finally leads to 1216 correspondences between revision 12 and 13. Noticeable peaks of differences and therefore corresponding operations between revision 2 and 3 as well as 4 and 5 can be explained by the semantics of the changes: The first peak corresponds to a new *Stamp* module, which adds a lot of new elements with its own semantics to the PPU for example.

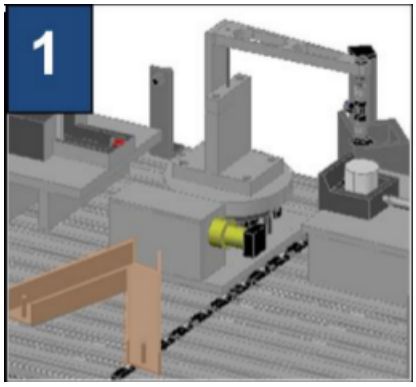


Figure 4.2: *PPU case study revision 1 [10]*

Figure 4.3 is the result of the combined usage of all tools and services implemented as part of this Master's Thesis, whereas the results themselves are presented in chapter 5.

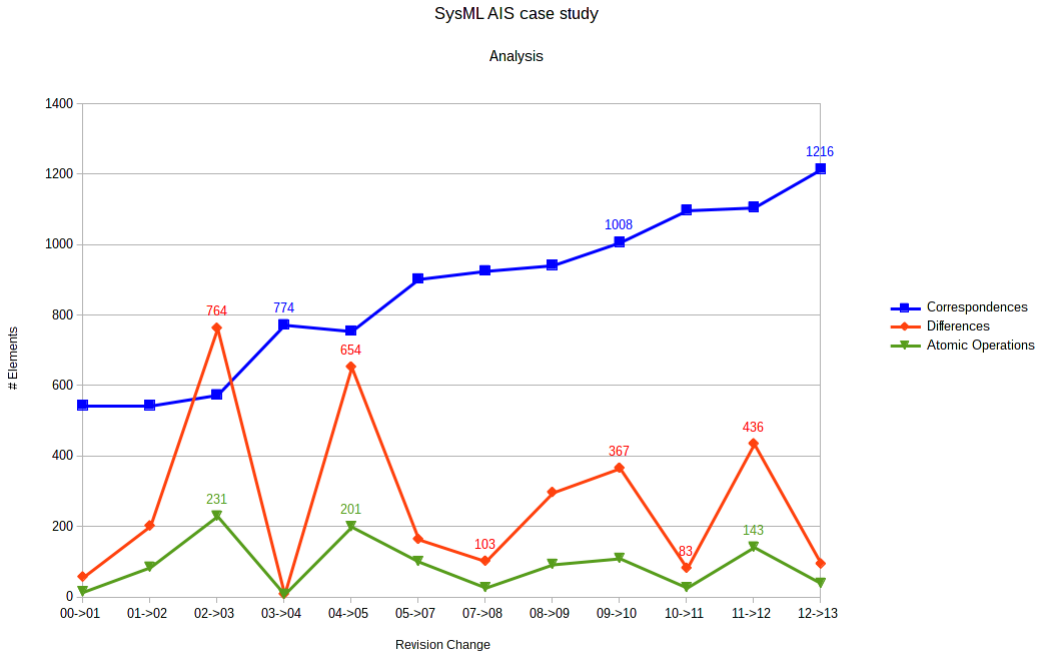


Figure 4.3: *PPU revision changes overview*

4.2 Analysis

Additionally to the semantics explained in the previous section, the SysML case study has been analyzed in detail regarding issues of different variants. Several tools have been developed as part of this analysis, which shall not be explained in this Master's Thesis as only their results are of importance. Three types of issues have been defined, as they differ in impact on modeling tools and will be explained hereafter.

Technical Issues

Issues of this type have a crucial impact on modeling tools, as the results may differ distinguishably in absence of such issues. Whereas some modeling tools may deliver slighty wrong results, other modeling tools may deliver substantial wrong ones. Three examples of such issues found in the PPU shall be given:

- **Wrong UUIDs**
  The example of wrong UUIDs already given in figure 2.4 shall be recalled: The UUIDs of two corresponding associations in consecutive revisions of the PPU differ, as the association in the later revision has been created newly by the developer. They describe the same semantics thus both associations should be matched and the only change detected should be a name change of the association. As many modeling tools rely on correct identifiers, they would produce wrong results which differ extremely from the expected. A summary of all wrong UUIDs found through all revisions of this SysML case study is shown in figure 4.4. The red bars are describing *newly* created UUID mismatches between revisions, whereas the green bars present the number of wrong identifiers carried over from older revisions.

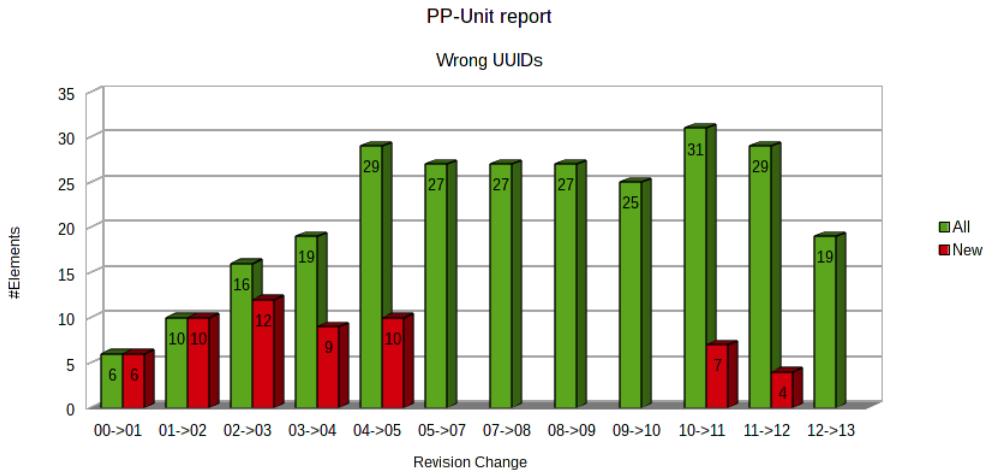


Figure 4.4: *Summary of wrong UUIDs*

- **Usage of EAnnotations**
  EAnnotations are available in Ecore models for annotating modeling elements. They are defined like every other modeling element, therefore are also considered during the detection of differences and correspondences between models if absent. The case study is modeled in SysML using Papyrus[2], which will create EAnnotations for internal usage of representation of associations as illustrated in figure 4.5. Separating visual elements from semantic elements is crucial in all areas of software development, especially in the case of MDSD. This is an example which problems can be caused if such requirements are not met by modeling tools: Differencing between two revisions which have been created with different tools will lead to wrong results, as the EAnnotations added by Papyrus will cause differences.

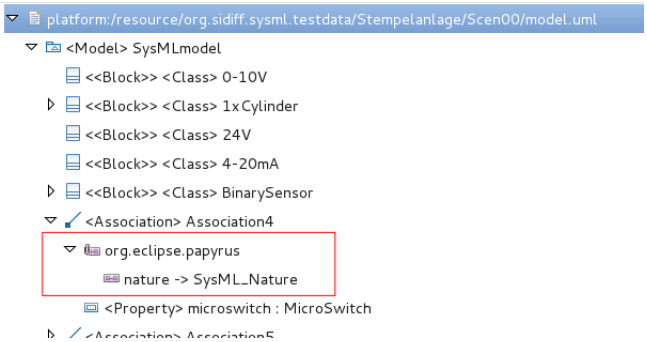


Figure 4.5: *EAnnotations created by Papyrus*

- **Usage of special characters**
  UML does support the usage of special characters, which may cause problems in used modeling tools. Each modeling tool may handle these characters differently, as they may escape them with new ones or do not alter them at all. An example of the usage of such special characters can be depicted in listing 4.1: Papyrus itself escaped the character „<“ by using the equivalent „<“ whereas the opposite character „>“ is not altered at all. These problems may cause problems at saving or loading the models, as the special characters are interpreted differently depending on the used modeling tool.

```
1 <subvertex xmi:type="uml:State" name="&lt;&lt;InCycle>>PickedUpState">
2  <doActivity xmi:type="uml:OpaqueBehavior" name="WPPickedUp:=TRUE;">
3    <language>Natural language</language>
4    <body>Kran_Sauger_an:=FALSE;&#xD; Kran_Sauger_aus:=true;</body>
5  </doActivity>
6 </subvertex>
```

Listing 4.1: *Example of special character usage*

Pragmatical Issues

This type of issues can be described as pragmatical, as they are not affecting modeling tools and their result themselves but may lead to wrong understanding of models by the developer. As described earlier, the SysML case study has been created using Papyrus, which makes use of its own diagram format. It separates the view onto the models via its different diagrams from the underlying model and offers the possibility to *hide* modeling elements, which are still present in the model itself. Unaware of consequences using this feature, developers may hide elements in a particular revision, followed by

another developer not knowing of their existence in the model. This leads to different pragmatical issues which have been categorized as follows:

- **Missing elements**

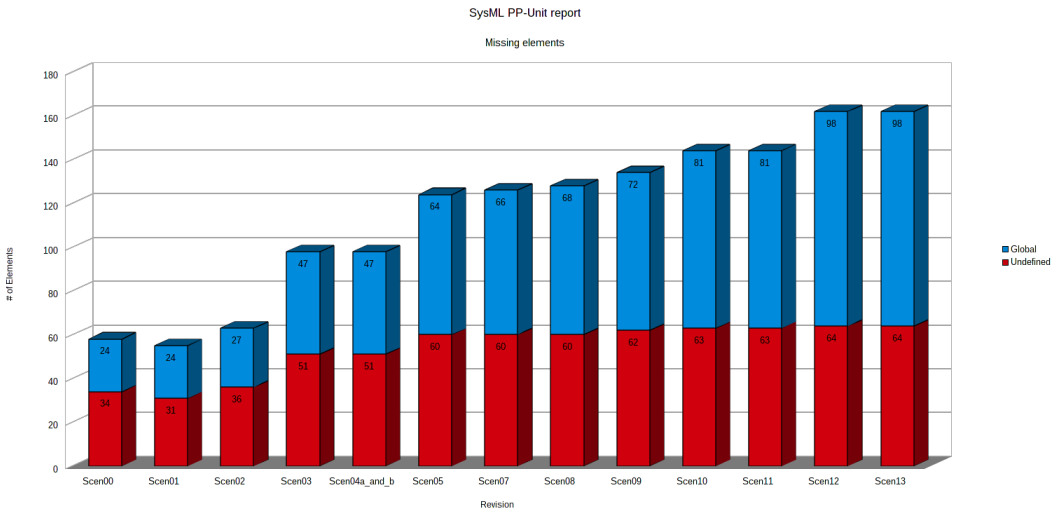


Figure 4.6: *Missing elements summary*

As mentioned before, hiding elements in a particular diagram view may lead to the misunderstanding, that these elements do not exist within the model itself. As shown in figure 4.6, there have been defined two types of such elements:

Undefined elements have to be defined in at least one diagram, for that the developer must know of their existence without the need of using other viewers as the diagram view itself. This category of elements has been further divided as depicted in figure 4.7. Modeling elements described as undefined are important as they may change the understanding the model drastically. Examples of such elements are *Block* or *State*.

Global elements do not need to be defined in at least one diagram, as they are expected to be declared globally. This declaration can be imagined to be implemented outside of the given model, as these include elements such as *Constraint* or *Literal*.

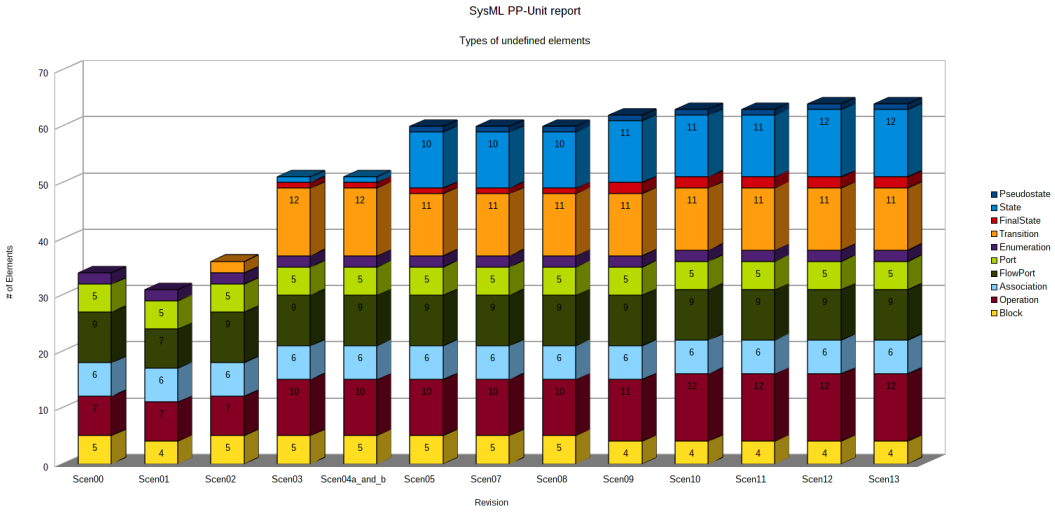


Figure 4.7: *Types of undefined elements*

- **Hidden model diagrams**
  Hiding elements is not restricted to modeling elements as Papyrus provides the feature of hiding particular diagrams at once. This leads to the same problems described beforehand, whereas in this case more elements are affected at one time.

4.3 Adaptions

As described in the previous section, there have been found different issues concerning the analyzed SysML case study. Both technical and pragmatical issues have been fixed in the course of this Master's Thesis for making all revisions compliant to modeling tools and easing the understanding of these models.

- **Wrong UUIDs**
  Using the developed SiDiff services described in chapter 3 all wrong UUIDs have been fixed and therefore no longer present a problem to modeling tools relying on identifiers.

- **Usage of EAnnotations**
  All used EAnnotations concerning tool specific information have been removed, as they may cause wrong results and do not provide any semantics.

- **Usage of special characters**
  Special characters, which may cause problems with modeling tools, have been stripped or replaced by semantically equal ones. The special character „<“ has been replaced by the string „LESSTHAN“ if this has been the desired semantics of this character for example.

- **Missing elements**
  All elements defined as global have been added to the Papyrus diagram view, so developers can understand the models themselves more easily. Global elements have not been added, as they may be defined elsewhere.

- **Hidden model diagrams**
  In the course of this Master's Thesis all hidden diagrams have been restored and can therefore be used and edited again.

After adapting the PPU without altering the semantics of said models, the case study presented in this chapter does not pose a problem to modeling tools at all. The modified study has been used as exemplary input for all tools and services created in the course of this Master's Thesis whereas the results are presented in the following chapter.

Chapter 5

Solution Results

This chapter illustrates the results of all implemented services and tools created during this Master's Thesis by using a SysML case study explained in the preceding chapter. To achieve this all tools in the SEG pipeline are executed consecutively thus presenting the resulted integration of UML profiles using a real world example.

For testing the implemented solutions, one example scenario may be given for this purpose. Instead of using the PPU scenarios 0 and 1 described in section 4.1, the tools shall be tested with a more complex revision change implemented in the SysML case study. Therefore both revision 2 and 3 are used, whereas the changes between them are illustrated in figure 5.1.

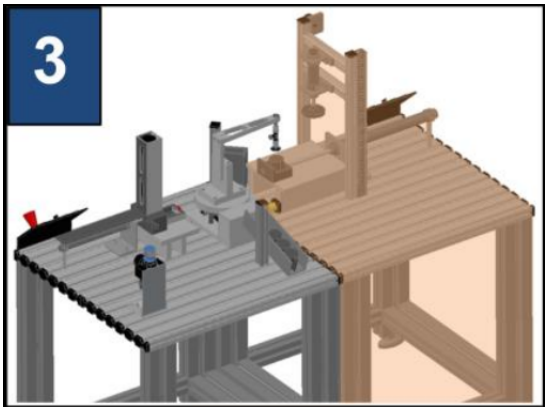


Figure 5.1: *PPU case study revision 3 [10]*

The important difference between both model revisions is the addition of a new *Stamp* module. Only metallic work pieces should be stamped, whereas black plastic work pieces

are transported to the slide without being processed. To implement such semantics, there has to be an additional sensor for detecting the position of the stamp, whereas revision 2 already contains the inductive sensor for differentiating between the two kinds of work pieces. The stamping process itself is straightforward as one can imagine the movement of the stamp module consisting of up and down alterations. Finally the crane transports the stamped or unprocessed work piece onto the slide. A view illustrating the relationship described beforehand can be seen in figure 5.2.

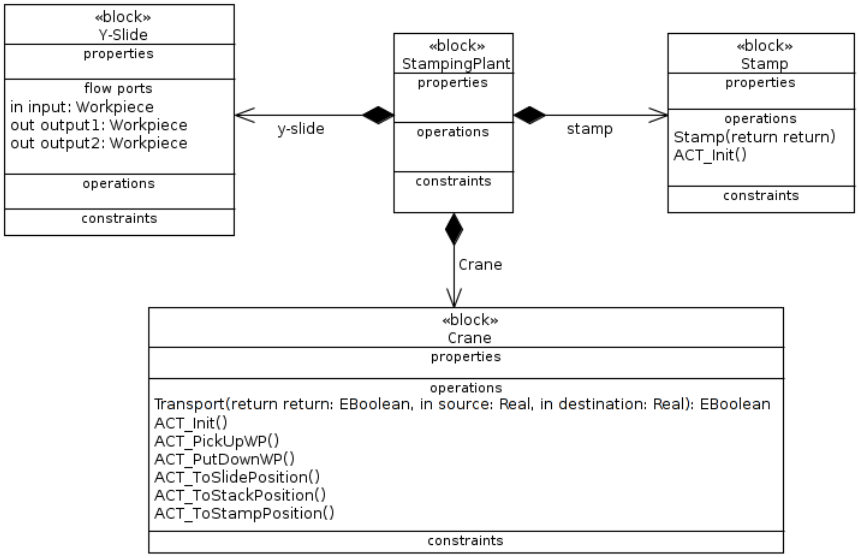


Figure 5.2: *Revision 3 UML diagram snippet*

As described in the Diploma Thesis of Dennis Koch [9], one practical aspect of the creation and application of patches is the propagation of changes throughout different versions of software like a product family. Considering this use case one can imagine two different automation plants P_1 and P_2 each consisting of mentioned PPU at revision 2. In need of such functionality the customer owning P_1 wants the provider of the PPU to integrate the described stamp module. As P_1 now equals to revision 3 depicted in figure 5.1, the owner of P_2 wants to have the same functionality for his plant. Instead of developing the exact same model the developer wants to reuse his work done in P_1. To achieve this goal all developed tools in this Master's Thesis are used consecutively, divided into steps explained in the following paragraphs. For a better overview the tool pipeline is yet again depicted in figure 5.3, whereas the used model instance A_1 corresponds to P_1 and A_2 to P_2 respectively.

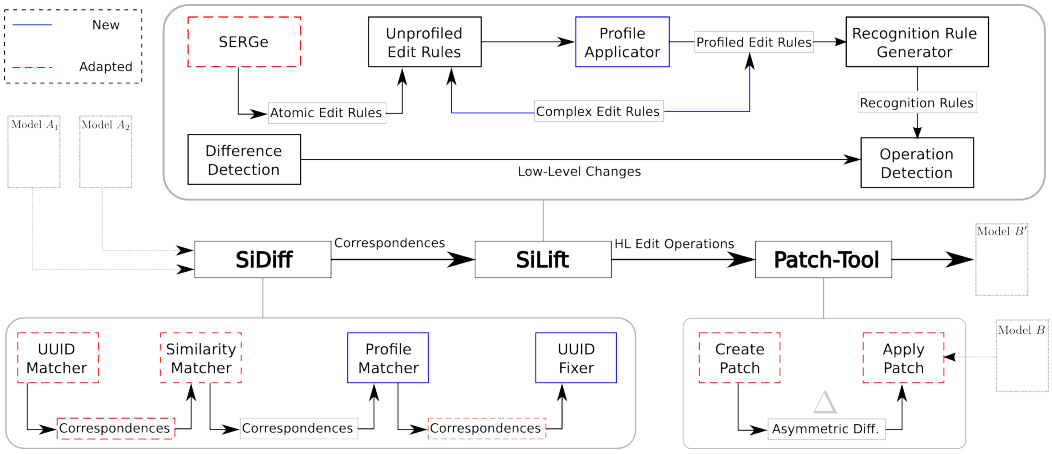


Figure 5.3: *UML profile integration overview*

Compare P_1 and P_2

The first step to the goal illustrated above is the detection of changes between P_1 and the plant P_2. Using the SiDiff matching pipeline leads to following results:

- **UUID Matcher**: 572 / 591 elements of P_2 matched, 785 elements overall in P_1
- **Similarity Matcher**: 16 / 19 elements additionally matched
- **Profile Matcher**: 0 / 3 elements matched
- **Overall result**: 588 / 591 elements of P_2 matched, 197 elements added in P_1

The first step matches all elements corresponding to their identifiers, whereas in this example almost all elements could be matched already in this first matching step. Using the UUID matcher at first in this particular case therefore leads to better results in the following similarity matching phase. In this second matcher additionally 16 elements are matched according to their similarity. The final matching step for profiled elements does not add any matches, as the remaining 3 elements are originated from pure UML. In the overall result for almost all elements of P_2 have correspondences been found in P_1 which translates to only 3 delete operations compared to P_2. Additionally 197 elements have been found in P_2 which correlates to newly added model elements in comparison to P_1.

Analyze computed difference

As the addition of a stamp block includes many other changes as well, the developer now wants to analyze the detected differences and changes in detail. First SiLift will derive low-level changes from the symmetric difference computed by SiDiff. There are 764 low-level changes detected between P_1 and P_2 which need to be analyzed now if not lifted afterwards. To cope with this many computed low-level changes the SiLift tool will lift this difference into a more meaningful list of edit operations. As seen in the center of figure 5.3, SiLift now makes use of the profiled edit rules created through the *ProfileApplicator* as well as the manually created edit rules. Using a complete rule base created earlier for lifting will recognize edit operations like the manually created depicted in figure 5.4 and the profiled one in figure 5.5.

Figure 5.4: *Recognized manually created edit operation*

Low-level changes which are contained in such SCSs will ease the understanding of made changes as clearly presented in figure 5.4: 14 low-level changes have been grouped together into 1 meaningful edit operation, the creation of an association which is navigable in one end that is.

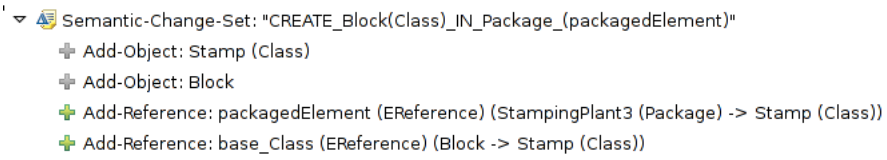


Figure 5.5: *Recognized profiled create operation*

A result summary of all lifted low-level changes detected is presented in the following table which reduces the number to 231 edit operations instead of 758 low-level changes.

Edit Operation	Amount
CREATE_Transition_IN_Region_(transition)	49
CREATE_State_IN_Region_(subvertex)	27
SET_Transition_(guard)_TGT_Constraint	24
ADD_Operation_(method)_TGT_Behavior	19
CREATE_Constraint_IN_Transition_(ownedRule)_LiteralString_	19
CREATE_Activity_IN_State_(doActivity)	14
CREATE_OpaqueBehavior_IN_State_(doActivity)	8
CREATE_Pseudostate_IN_Region_(subvertex)	7
CREATE_FinalState_IN_Region_(subvertex)	6
CREATE_OpaqueBehavior_IN_State_(entry)	6
CREATE_OpaqueBehavior_IN_Transition_(effect)	5
CREATE_Association_Navigable_Property_ONE_Block(Class)_IN_Package	5
UNSET_Transition_(guard)_TGT_Constraint	4
CREATE_Constraint_IN_Transition_(ownedRule)_LiteralBoolean_	4
CREATE_OpaqueBehavior_IN_State_(exit)	3
CREATE_Operation_IN_Block(Class)_(ownedOperation)	3
SET_Association_Name	3
SET_State_Name	3
CREATE_StateMachine_IN_Block(Class)_(ownedBehavior)	3
SET_OpaqueBehavior_Name	2
SET_Region_Name	2
DELETE_Transition_IN_Region_(transition)	1
REMOVE_Operation_(method)_TGT_Behavior	1
SET_LiteralString_Name	1
DELETE_Constraint_IN_Transition_(ownedRule)_LiteralString_	1
MOVE_Constraint_(ownedRule)_Transition	1
SET_Constraint_Name	1
CREATE_Region_IN_State_(region)	1
MOVE_State_FROM_Region_(subvertex)_TO_Region_(subvertex)	1
SET_Package_Name	1
CREATE_Parameter_IN_Operation_(ownedParameter)	1
SET_LiteralString_Value	1
SET_Activity_Name	1
DELETE_OpaqueBehavior_IN_State_(doActivity)	1
CREATE_Block(Class)_IN_Package_(packagedElement)	1
	231

Table 5.1: *SiLift lifting result summary*

Create a patch

Using the Patch-Tool the developer now wants to create a patch containing all edit operations presented in the previous table. As the rule base is complete and all low-level changes have been lifted by SiLift previously, all dependencies between these are considered additionally. One example of such dependency detected is illustrated in figure 5.6: To create a transition connecting two states both states have to be created first.

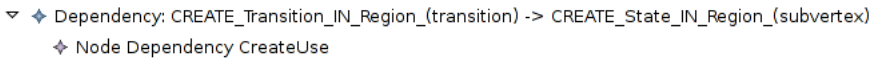


Figure 5.6: *Detected create-use dependency example*

Apply the created patch

The next step is to apply the patch created beforehand to plant P_2, as this plant shall implement the changes of P_1 compared to P_2. The application is straightforward as presented in [9] and shall not be explained in detail.

Validate the resulted model

The final step is to validate the resulted model. The developer wants to assure that

a) the patch has applied all changes consistency preserving and that

b) the resulted model equals to the expected result.

As the patch creation already assures a), the final step is to test the model for its semantically correctness. To check for equality between P_1 and P_2 described in b) the SiDiff matching pipeline is used again and must report no differences and all elements shall correspond as already presented in figure 2.9.

Final result

The patch containing all changes could successfully be created and applied and resulted in an equal plant implementing the stamp module, which is the desired result. The patch can now be applied to different PPUs for integrating such new module.

Additionally to this example scenario between revision 2 and 3 of the PPU in the course of this Master's Thesis a batch application for testing all steps above has been adapted for SysML. The idea is to execute all steps for consecutive revisions and log all results, which are shown in the following table and correspond to the graphs in figure 4.3.

Revision Change	Correspondences	Differences	Operations	Equal
00→01	545	58	16	Yes
01→02	545	203	86	Yes
02→03	575	764	231	Yes
03→04	774	9	9	Yes
04→05	756	654	201	Yes
05→07	904	165	102	Yes
07→08	927	103	28	Yes
08→09	943	298	94	Yes
09→10	1008	367	111	Yes
10→11	1099	83	28	Yes
11→12	1107	436	143	Yes
12→13	1216	95	40	Yes

Table 5.2: *Batch-Patch summary report*

As illustrated in the table all revision changes can be handled now in all tools in the SEG pipeline, whereas all resulting patched models are equal to their corresponding revision. The final conclusion of these results is presented in the following chapter.

Chapter 6

Conclusion and Future Work

This chapter concludes this Master's Thesis and offers an outlook for possible future work based on the experiences gained throughout.

The integration of UML profiles into the SiDiff and SiLift tools has succesfully been achieved in this Master's Thesis. For the whole pipeline to work with such profiled models all parts had to be adapted, namely:

- Matching
- Lifting
- Patching

All three parts have successfully been adapted, whereas three new services and tools have been introduced in this thesis:

- **ProfileMatcher**
  This services integrates UML profiles into SiDiff and therefore allows matching of such.
- **UUIDFixer**
  This new service allows for fixing of wrong identifiers and can be used for compliance of models to other modeling tools.

- **ProfileApplicator**
  This new tool integrates UML profiles into SiLift and therefore lays the foundation for the lifting and patching functionalities.

After the integration process in the course of this Master's Thesis the results have been tested using a real world SysML case study. First one example has been introduced in detail and has been tested followed by a batch patch application testing all revision of the whole case study. The final result achieved is the successful integration of UML profiles as all tools deliver the expected results, especially the final test for equality between patched models and their corresponding revisions. The newly integrated support for UML profiles into SiDiff and SiLift increases their support of modeling domains drastically, as many new modeling domains may arise by the aid of the implemented profiling mechanism. Additionally the real world case study created in SysML can be used in future tools in this ecosystem, as it represents a complex model and can be utilized for runtime tests for example. The newly adapted UML configuration for SiDiff can be used in other contexts as well. A final overview of the results achieved in this Master's Thesis are presented in figure 6.1.

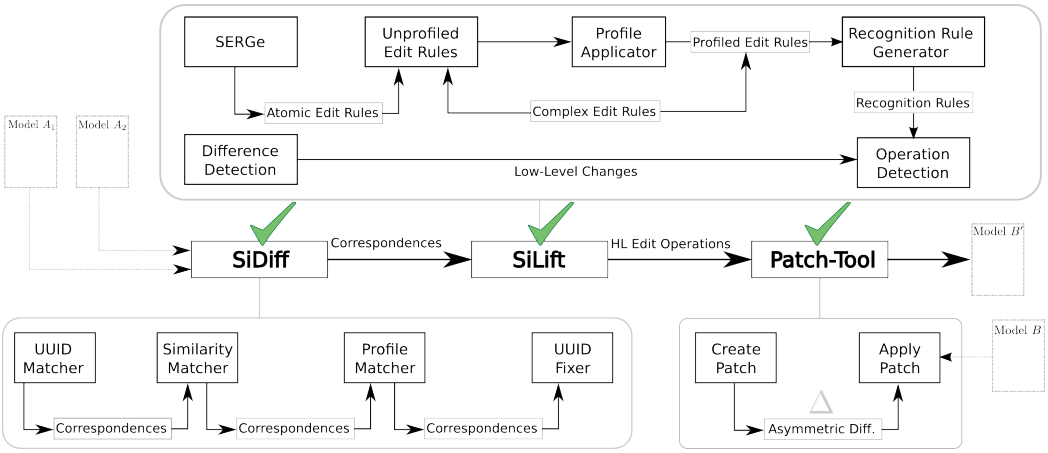


Figure 6.1: *Final integration result overview*

The following aspects could be considered in future work:

- **Testing of other profiles**
  Additionally to the tested UML profile SysML others like MARTE can be used for testing, as their results may differ. MARTE makes use of own semantics in stereo types which needs to be adressed explicitly. One solution is the addition of a new compare method for SiDiff which takes the similarity of stereo typed elements into consideration.

- **Construction of edit rules**
  To achieve better lifting result one can define more complex edit rules than defined in this Master's Thesis for UML itself or for a given profile.

- **Implement remaining variants**
  As in this Master's Thesis only variant 3 of the different integration approaches into SiLift has been implemented, both remaining could be implemented in the future. As depicted in figure 2.5 the first variant would result in an integration into SERGe, whereas the second variant shall be implemented as own tool. Instead of supporting only the merging of base and profiled edit rules one can implement such Henshin rule merger generically and therefore support the merging of Henshin rules in general. This would result in new possibilities like merging two atomic edit rules into one complex rule without the need of manual intervention.

- **Performance optimizations**
  As the newly supported SysML case study can be described as complex, some parts in the tooling pipeline shall be taken into consideration for performance optimizations. This concerns elements like runtime or memory consumption as well as user interface optimizations.

Appendix A

Higher-Order-Transformations

The following figures illustrate the in this Master's Thesis created HOTs, which transform given Henshin edit rules into profiled ones.

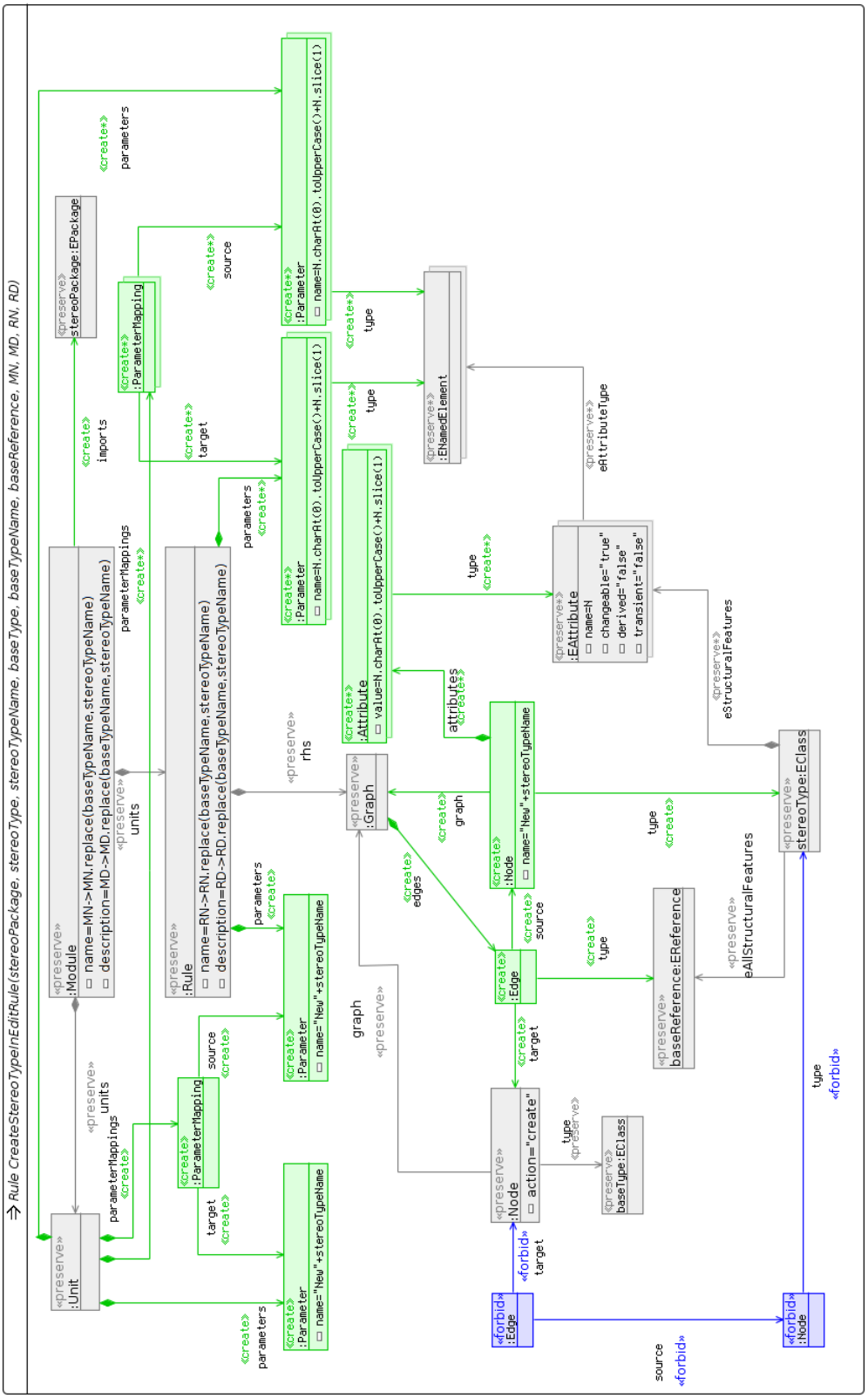


Figure A.1: *HOT for create nodes*

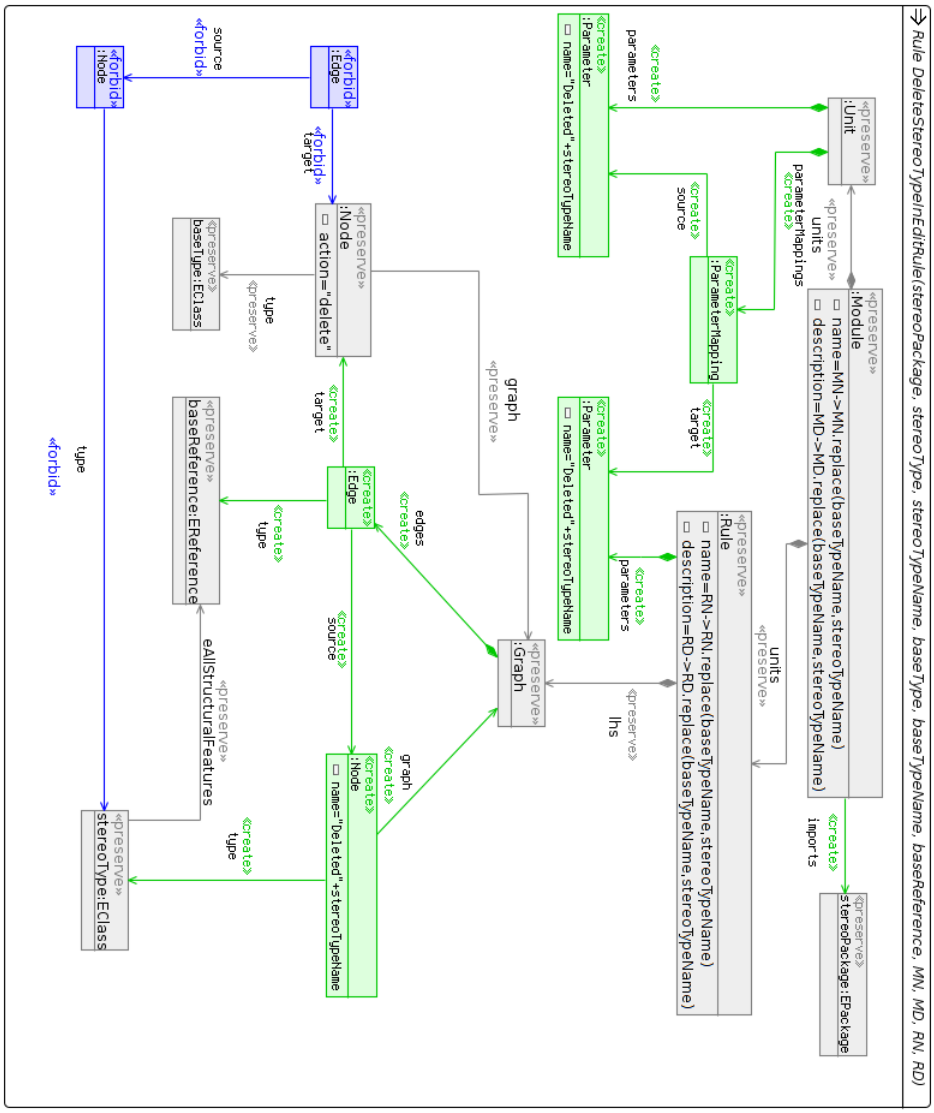


Figure A.2: *HOT for delete nodes*

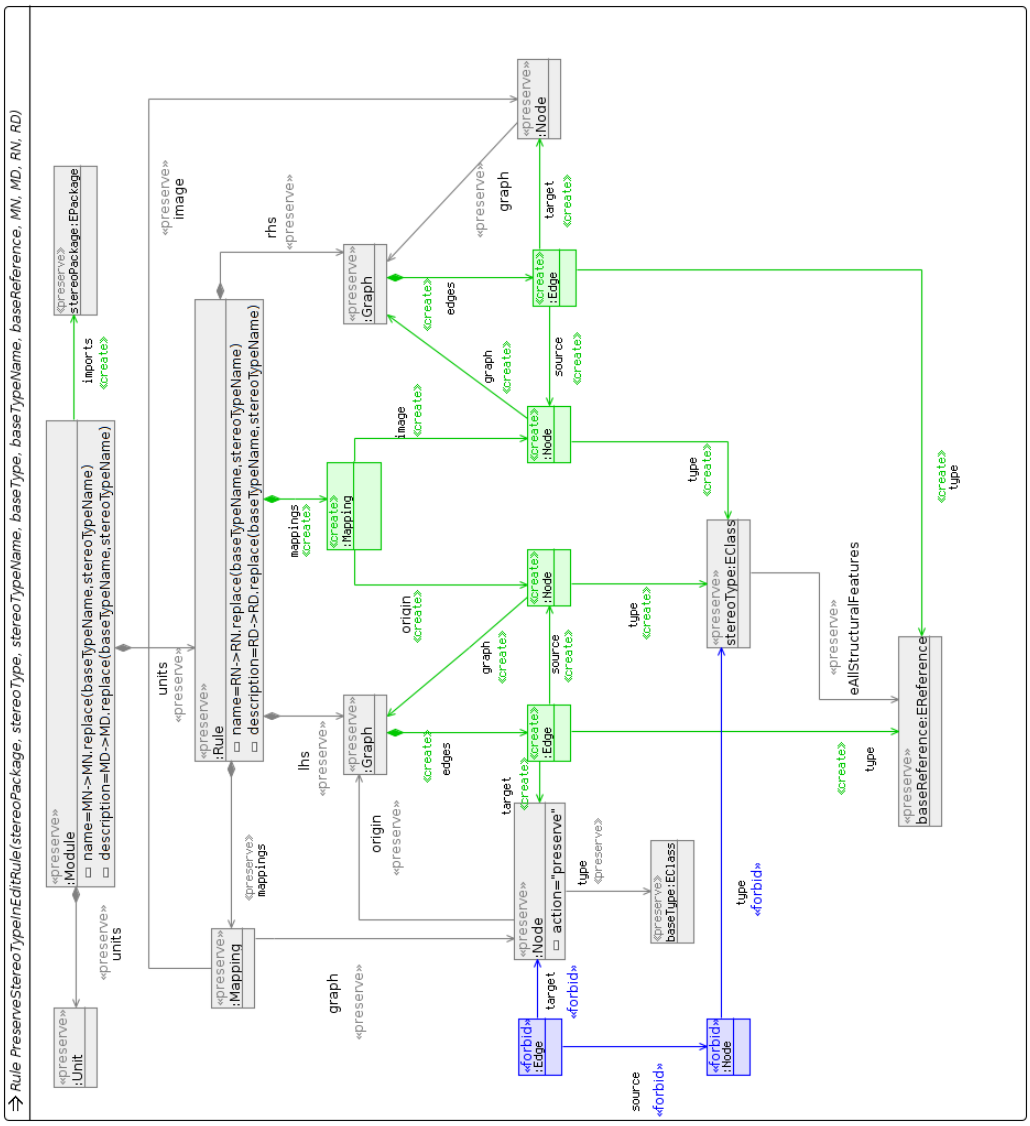
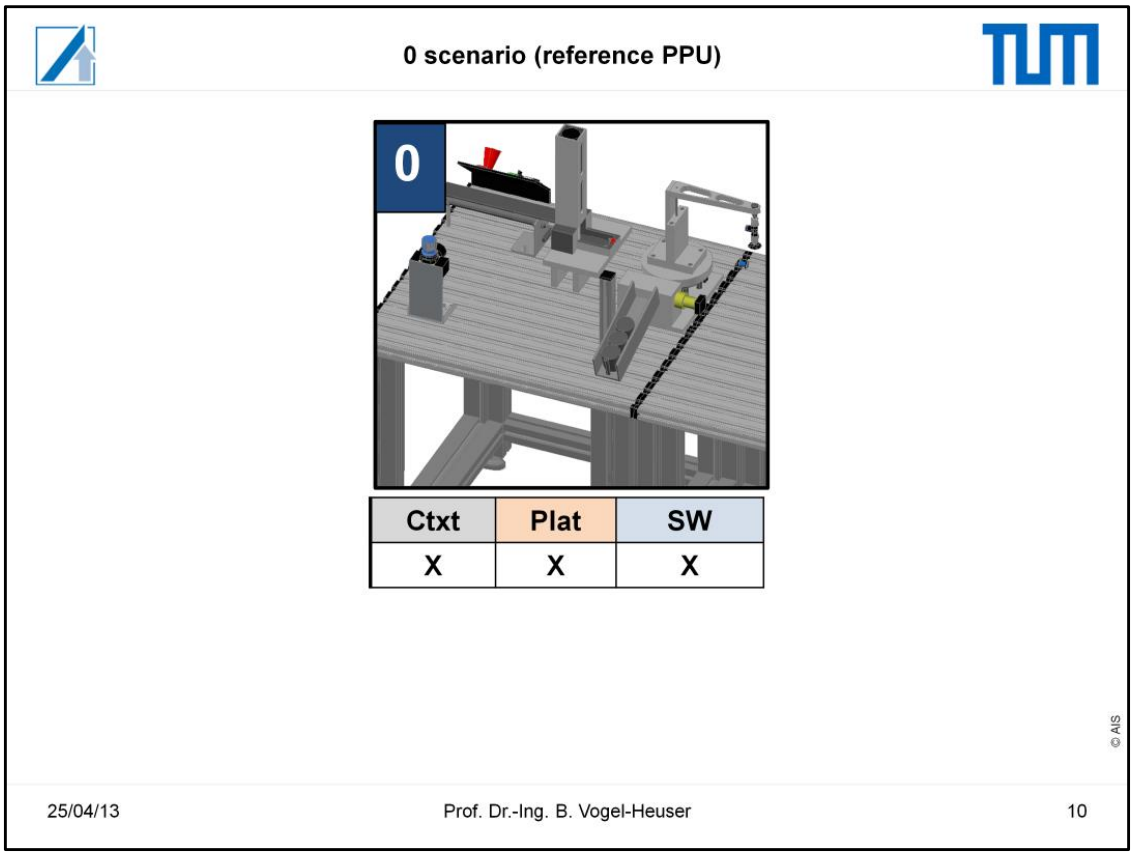


Figure A.3: *HOT for preserve nodes*

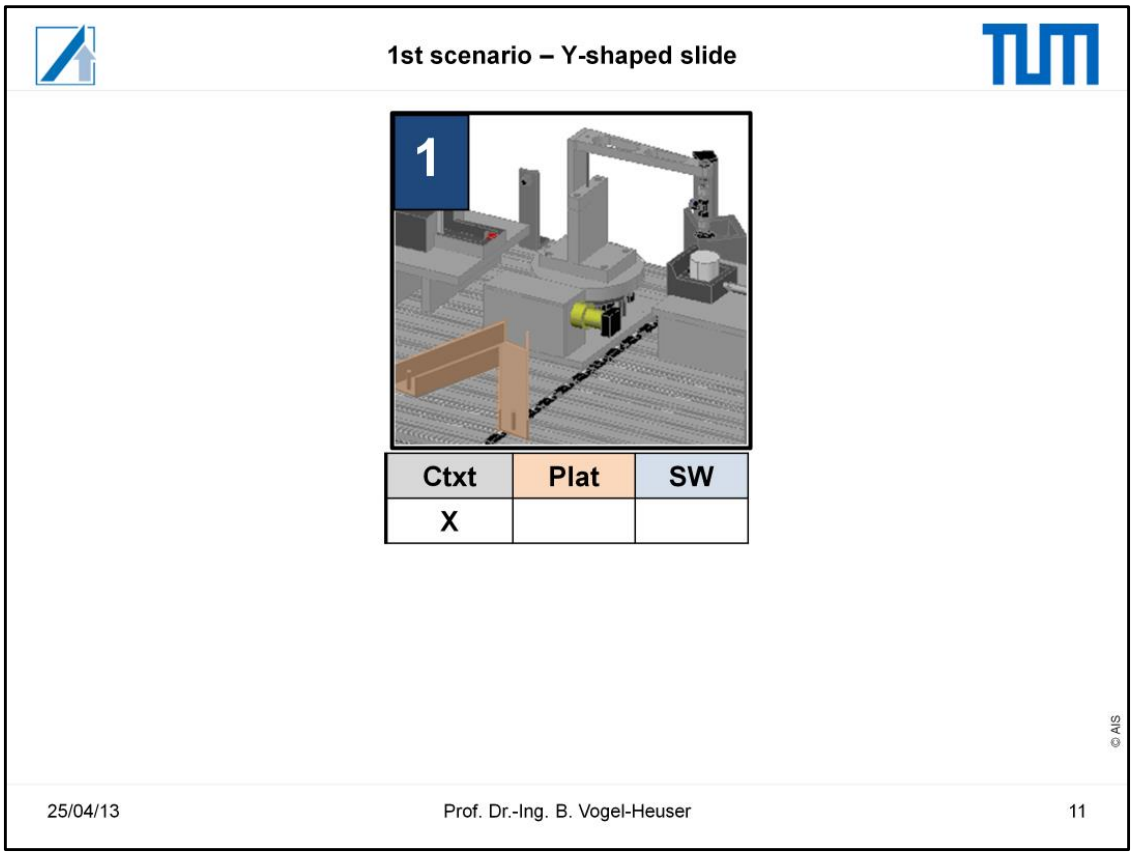
Appendix B

SysML Case Study Evolution

This appendix describes the evolution of the SysML case study used throughout this Master's Thesis in a detailed manner. This evolution has been created by the technical university of Munich and the corresponding publication [10] is recommended.

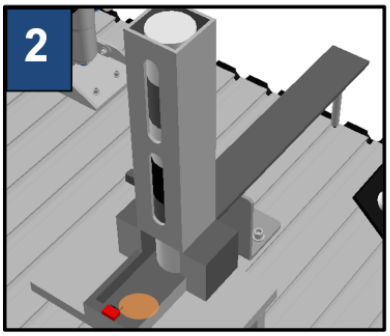


The *initial situation* of the plant is that the pick and place unit only consist of the stack with a separator included, a crane and a slide. There is only one kind of work piece namely the metallic work piece. In the stack is only one digital sensor which is responsible to detect if a work piece is present as soon as the separator is extracted. The separator itself consists of two digital sensors – at the front and at the end of the separator – to detect if the separator is extracted or retracted.
The Crane picks up work pieces at the stack, turns 90° anti-clockwise and places the work piece at the slide. Then the crane moves back to the stack. To detect the position of the crane, weather it is at the stack or at the slide, two (tactile) digital positioning sensors at the bottom plate of the crane are mounted.



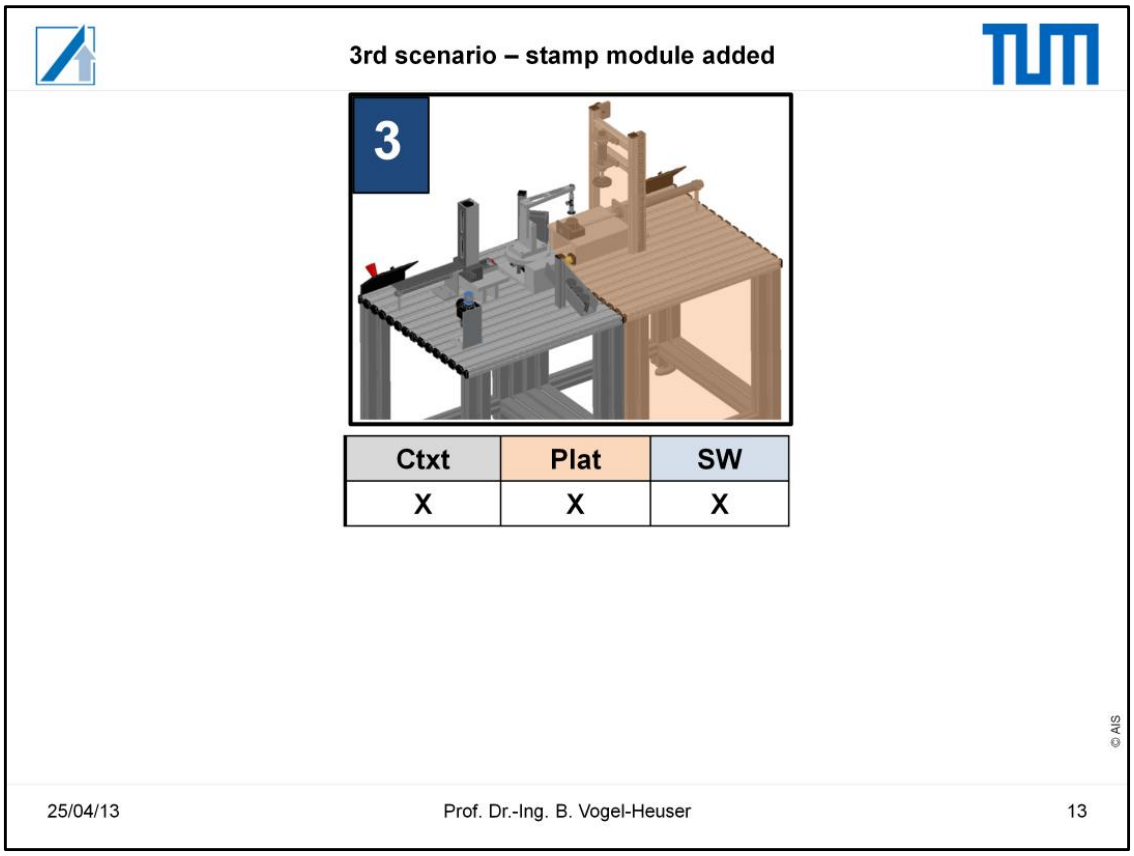
In this scenario only the slide is replace by a slide with Y shape. Due to this shape first one side of the slide is filled up with work pieces and than the other side. The slide is only there to increase the capacity of the storage of the slide.

2nd scenario – black plastic work pieces

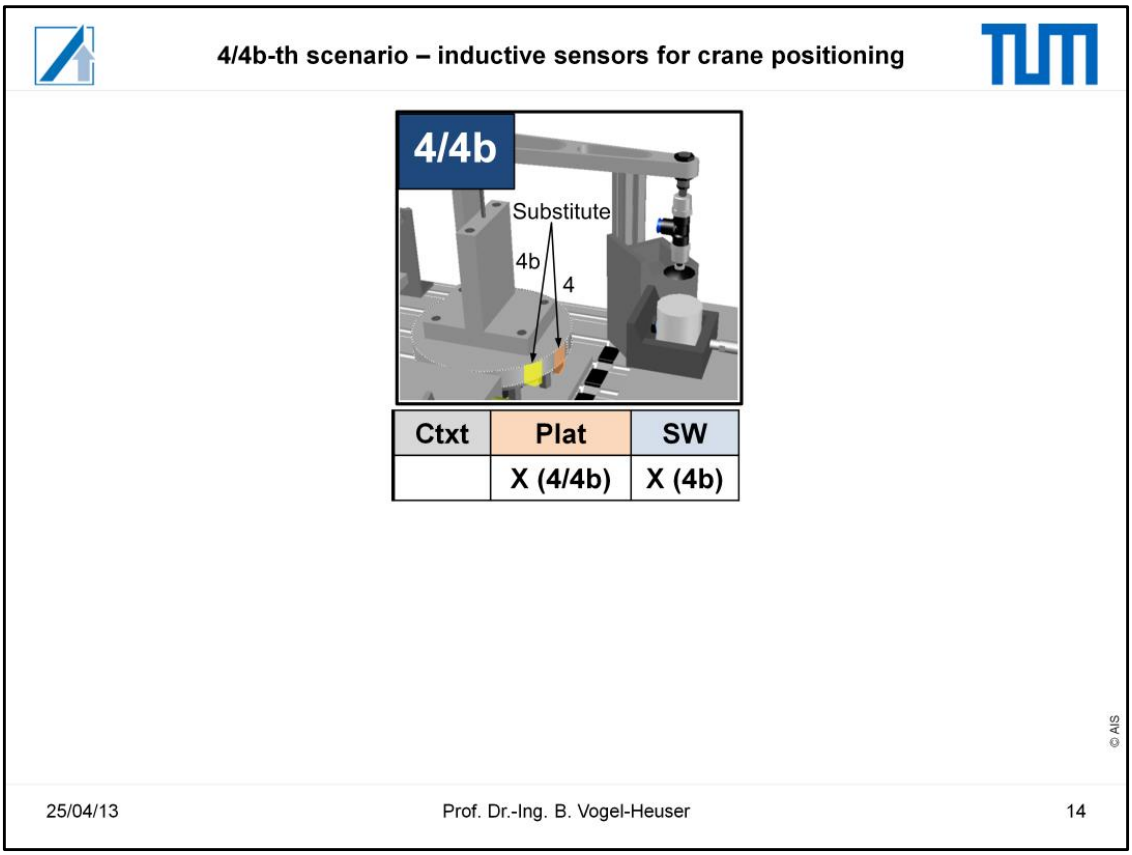


Ctxt	Plat	SW
X	X	X

In this evolution scenario a new kind of work pieces - black plastic work pieces – is additionally processed by the plant. To detect if there is a black plastic work piece or a metallic work piece an inductive sensor is used at the bottom of the extraction position of the separator.

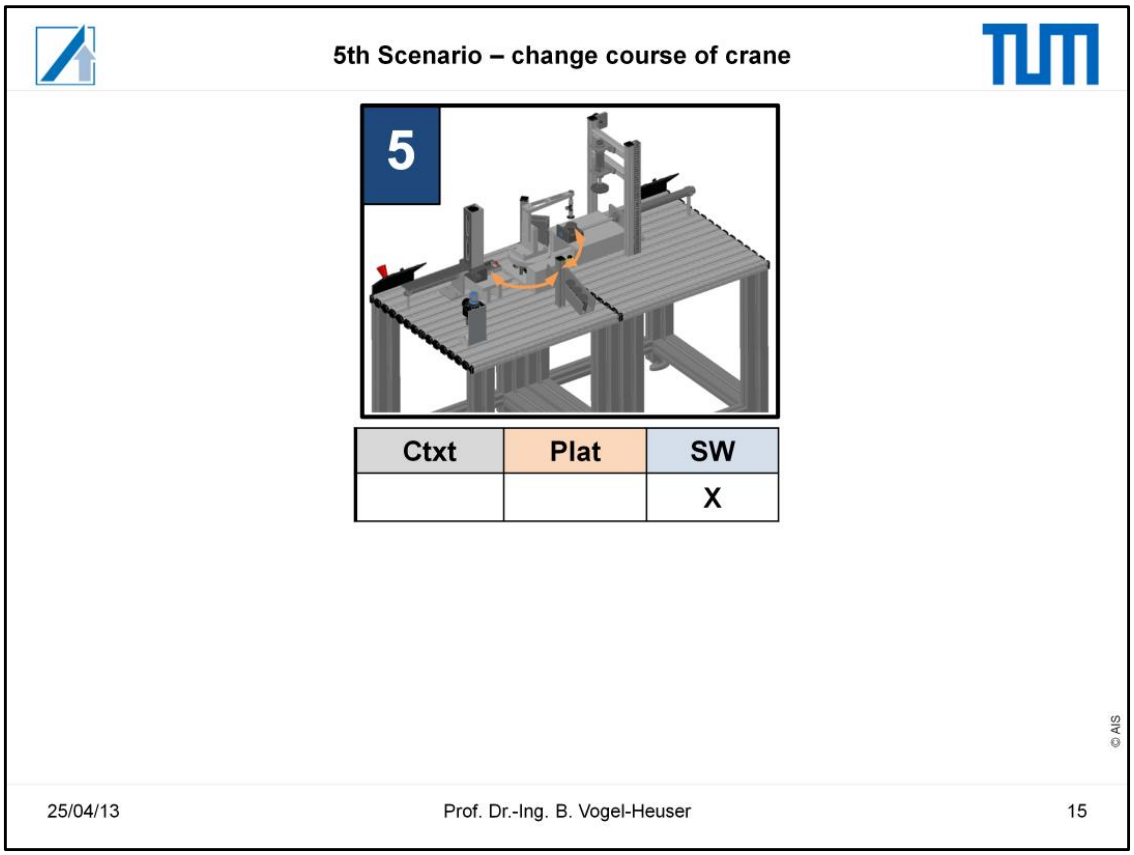


Now the stamp module is added. Only metallic work pieces should be stamped. Black plastic work pieces are transported to the slide, directly.
To detect the stamp position of the crane, a new digital sensor is added to the bottom plate of the crane. The stamp module itself consists of a magazine where the crane places the work pieces and stamp.
The magazine retracts the work piece under the stamp. To detect if a work piece it present a digital sensor is placed in the magazine. As soon as the magazine is retracted under the stamp, the stamp moves down to press the work piece. After the pressing process the stamp moves up. To detect if the stamp or the magazine are retracted or extracted two digital sensor are placed at each device. As soon as the magazine retracts, the crane transports the work piece to the slide.



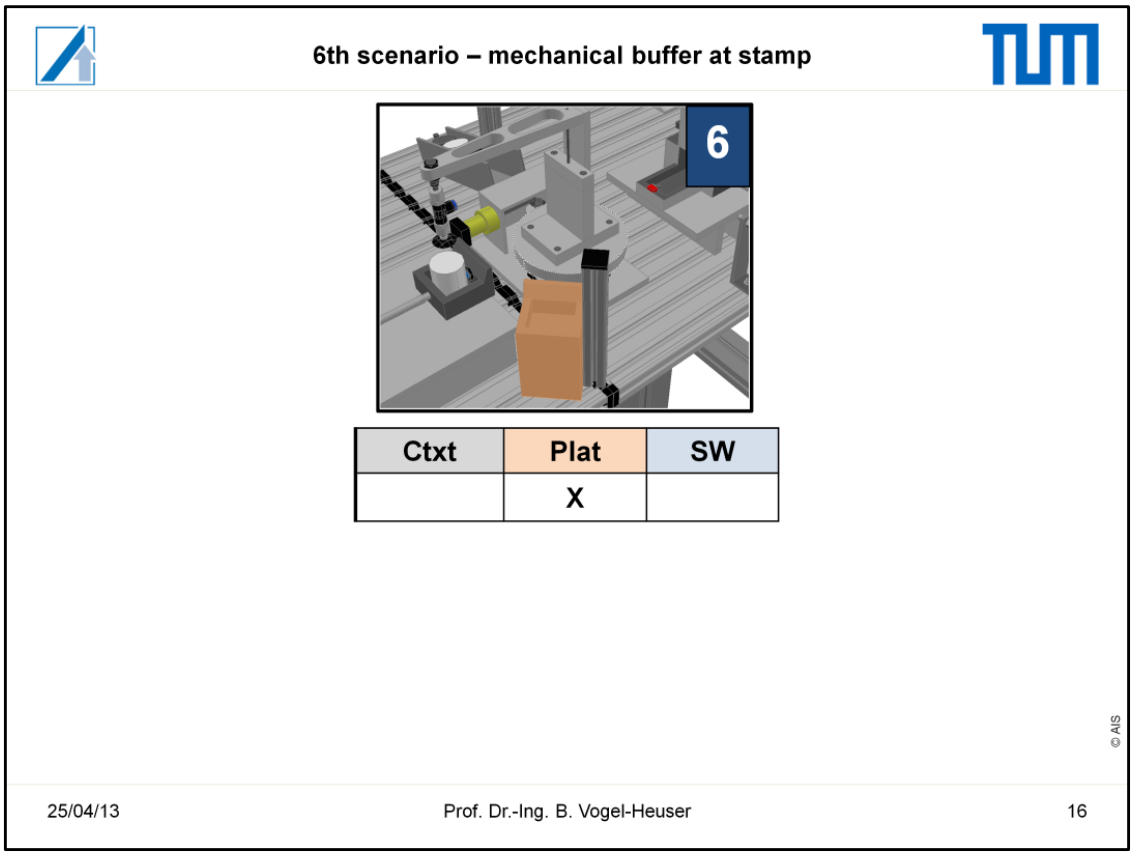
4: The (tactile) digital positioning sensor at the bottom plate of the crane are replaced by inductive sensors. The new sensors provide the same signals like the old positioning sensors, but are more robust against pollution.

4b: Same like for scenario 4, but the (tactile) digital positioning sensor are spatial shifted in comparison to scenario 4.



To realize a higher throughput of work pieces, the implementation of the crane is optimized: As soon as the crane places a metallic work piece at the stamp, it is checked if there is a black work piece available at the stack. If there is a black work piece, the crane uses the stamping time to transport the black work piece to the slide.

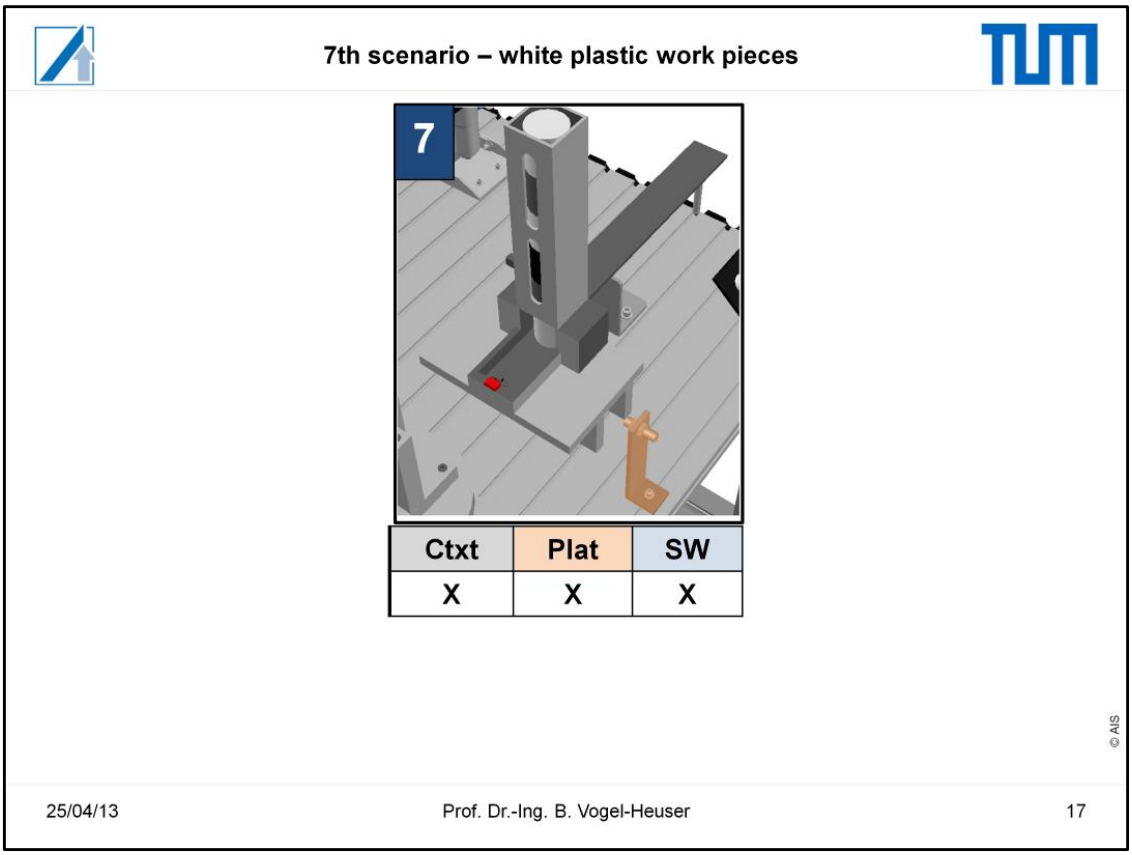
Than the crane moves back to the stamp, to transport the stamped work piece to the slide.



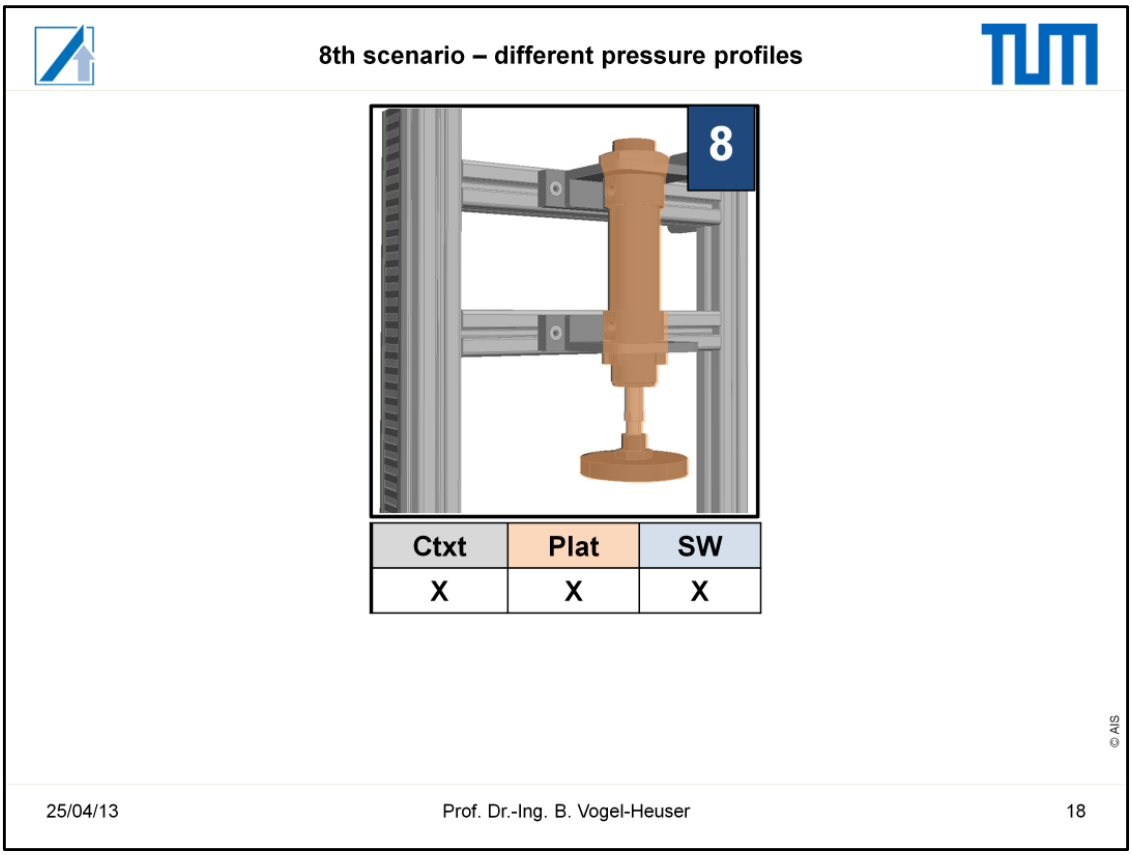
If there is a black work piece the, the crane transports the black work piece to the slide.

To increase the parallelization of the crane, a mechanical buffer is place next to the stamp. As soon as a work piece is stamped, it is checked what kind of work piece is available at the stack. If there is another metal work piece, the crane transports the work piece to the mechanical buffer and the next kind of work piece is checked at the stack.

If there is another metal work piece the crane has to wait until the stamping process is done to transport the stamped work piece to the slide.

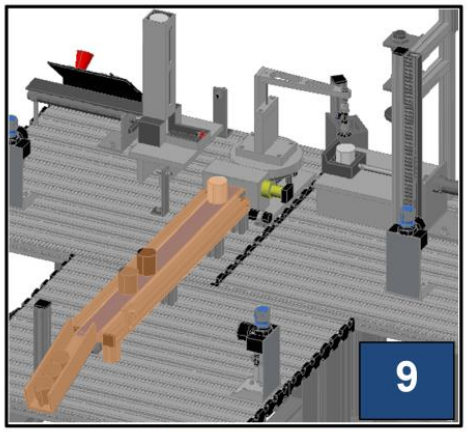


In this scenario white plastic work pieces are processed additionally. To detect what kind of work piece is available at the stack a (optical) digital sensor is placed next to the stack. This sensor is able to detect if there is a white work piece and a black work piece. In combination with the already insstalled indictuve sensor all kind of work pieces can be differantiated.
White plastic work pieces are also transported to the stamp first and than to the slide.



In this evolution scenario the work pieces have to be stamped using different pressure profiles. Therefore, a proportional valve and a analogue pressure sensor are installed into the stamp, to realize the pressure control.

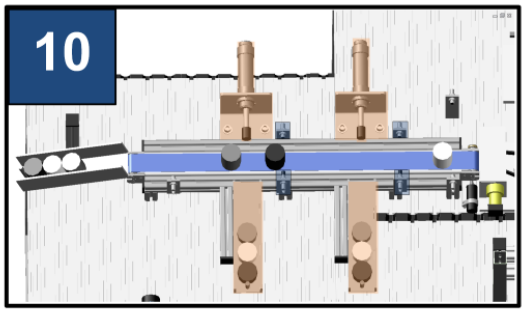
9th scenario – installation of conveyor



Ctxt	Plat	SW
X	X	X

At this stage a conveyor is installed. At the end of conveyor is one slide installed where the work pieces are transported to.

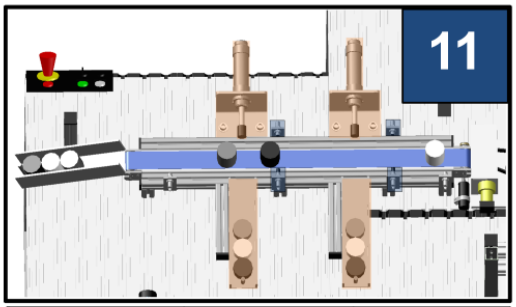
10th scenario – additional slides and pushers



Ctxt	Plat	SW
X	X	X

To increase the capacity of the storage at the conveyor two additional slides are installed at the side of the slide. The pusher are pushing the work pieces into the slide. First, the slide at the and is filled with work pieces, than the mid slide and than the slide at the beginning of the conveyor .

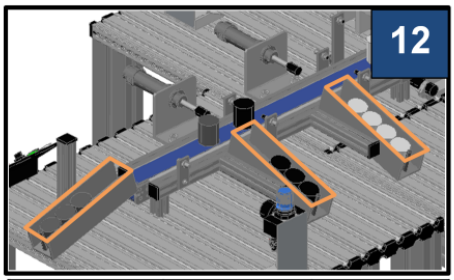
11th scenario – sorting of work pieces



Ctxt	Plat	SW
	X	X

To sort/order the work pieces at the slides, two optical sensor and two inductive sensor are installed to detect the kind of work piece. At the beginning of the plant there is a optical sensor. The next sensor (at the right side of the first pusher) is an inductive sensor.
At the right of the second pusher is another inductive sensor. At the right of the last slide is another optical sensor. The sorting at follows: 1st (right) slide only white work pieces
2nd (mid) slide only metal work pieces
3rd (left) slide only black work pieces

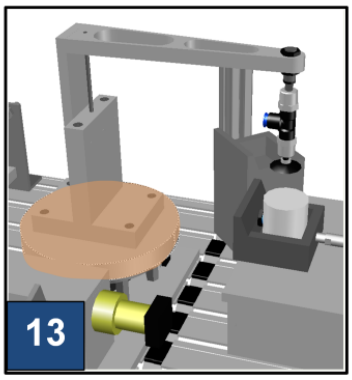
12th scenario – specific orders of work pieces



Ctxt	Plat	SW
		X

In this scenario the sorting/order of work pieces are changed at the slides. Now there have to be similar mixtures in all slides.

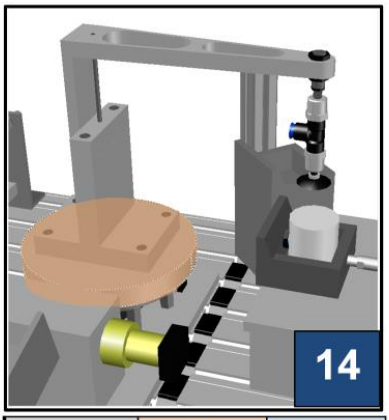
13th scenario – potentiometer at the crane



Ctxt	Plat	SW
X	X	X

To increase the positioning of the crane the digital sensors (at the bottom plate of the crane) are replaes by an potentiometer.

14th scenario – EMC Resistants



Ctxt	Plat	SW
X	X	X

To increase the restistance of electro magnetic influenced the potentiometer is replaced by an incremental encoder which also has an anlalogue signal.

Bibliography

[1] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced concepts and tools for in-place emf model transformations. In *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 121–135. Springer Berlin, 2010. (S. 11)

[2] Eclipse. Papyrus modeling tool. http://www.eclipse.org/papyrus/, September 2013. (S. 44)

[3] Ecore JavaDoc. Ecore. http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html#details, September 2013. (S. 38)

[4] Henshin Developers. Henshin. http://www.eclipse.org/henshin/, September 2013. (S. xi, 8, 10, 11)

[5] T. Kehrer, U. Kelter, M. Ohrndorf, and T. Sollbach. Understanding model evolution through semantically lifting model differences with SiLift. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 638–641, 2012. (S. 13, 16, 17)

[6] Timo Kehrer, U. Kelter, and G. Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 163–172, 2011. (S. 13)

[7] Udo Kelter, Timo Kehrer, and Dennis Koch. Patchen von Modellen. In *Software Engineering 2013, Fachtagung des GI-Fachbereichs Softwaretechnik*, volume 213 of *Lecture Notes in Informatics*, pages 171–184, 2013. (S. xi, 16)

[8] Udo Kelter, Jürgen Wehren, and Jörg Niere. A generic difference algorithm for UML models. In *Software Engineering*, pages 105–116, 2005. (S. 13)

[9] Dennis Koch. Anwendung von konsistenzerhaltenden Patches in der modellgetriebenen Softwareentwicklung. Master's thesis, University of Siegen, 2013. (S. 50, 54)

[10] C. Legat, J. Folmer, and B. Vogel-Heuser. Evolution in industrial plant automation: A case study. In *Proceedings of the 39th Annual Conference of the IEEE Industrial Electronics Society*, IECON '13, 2013. (S. xii, 41, 42, 49, 65)

[11] Object Management Group. Unified Modeling Language Specification. http://www.omg.org/spec/UML/2.4.1/, August 2011. (S. 4)

[12] Object Management Group. Systems Modeling Language Specification. http://www.omg.org/spec/SysML/1.3/, April 2012. (S. xi, 6, 7, 8)

[13] Object Management Group. Systems Modeling Language. http://www.omgsysml.org/, September 2013. (S. xi, 6)

[14] D. Pilone and N. Pitman. *UML 2.0 in a Nutshell*. In a Nutshell (o'Reilly) Series. O'Reilly Media, 2005. (S. 4)

[15] G. Rozenberg. *Handbook of Graph Grammars and Computing by Graph: Foundations*. World Scientific Pub, 1997. (S. 8)

[16] Software Engineering Group. SiDiff. http://www.sidiff.org, September 2013. (S. xi, 11, 12, 13)

[17] Software Engineering Group. SiLift. http://pi.informatik.uni-siegen.de/Projekte/SiLift, September 2013. (S. xi, 15)

[18] UML Diagrams. UML Profile Diagrams. http://www.uml-diagrams.org/profile-diagrams.html, September 2013. (S. xi, 5)

[19] University of Siegen. SERGe. http://pi.informatik.uni-siegen.de/mrindt/SERGe.php, September 2013. (S. 14)

[20] University of Siegen. Software Engineering Group. http://pi.informatik.uni-siegen.de/, September 2013. (S. 11, 13, 16, 22)

[21] T. Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design.* The MK/OMG Press. Elsevier Science, 2011. (S. 8)

[22] Wikipedia. Graph rewriting. http://en.wikipedia.org/wiki/Graph_rewriting, September 2013. (S. xi, 9)

[23] Wikipedia. System Modeling Language. http://en.wikipedia.org/wiki/Sysml, September 2013. (S. 6)

[24] Wikipedia. UML Class diagram. http://en.wikipedia.org/wiki/Class_diagram, September 2013. (S. xi, 5)

[25] Wikipedia. Unified Modeling Language. http://en.wikipedia.org/wiki/Unified_Modeling_Language, September 2013. (S. xi, 3, 4)