A Survey of Methods and Applications of Model Comparison

Technical Report 2011-582 Rev. 3

Matthew Stephan, James R. Cordy

School of Computing
Queen's University
Kingston, Ontario, Canada
{stephan,cordy}@cs.queensu.ca

June 2012

Contents

List of Figures

List of Tables

1 Introduction

High-level software models are representations of software artifacts or processes that are at a higher level of abstraction than source code. This includes structural modeling techniques, like the use of the Unified Modeling Language's (UML) class diagrams; behavioral modeling, such as done with sequence diagrams; and data-flow modeling techniques, most notably Simulink and Matlab.

A relatively new approach to software engineering is the use of Model-driven Engineering (MDE). MDE involves using high-level software models as the main artifacts within the development cycle. As MDE becomes more prevalent in the software engineering community, the need for effective approaches for finding the similarities and differences among high-level software models, or *model comparison*, becomes imperative. Since MDE involves models being considered *first class* artifacts by developers, it is clear model comparison is important as it assists model management and may help facilitate analysis of existing projects and research on MDE. It not only provides merit as a stand-alone task, but it helps engineers with other MDE tasks such as model composition and model transformation testing [39].

In this technical report we will investigate the current state of model comparison research. The goals in doing this are to a provide an overview of the area and also to ascertain the capability of existing model comparison techniques to identify common sub-structures or patterns within models. These patterns can ideally be used by project engineers to facilitate analysis and assist in the development of future MDE projects. This technical report will describe, in brief, the phases of comparison and will also investigate and describe the various uses/applications of model comparison including model versioning, model-clone detection, and others. This technical report will also help those looking to use model comparison techniques by acting as a quick reference guide organized by the types of models being compared.

This technical report begins by providing background information on MDE and model comparison in Section 2 in order to facilitate understanding of the material that follows to those not versed in these areas. Section 3 categorizes the existing model comparison approaches by the type and subtype of models that they are used to compare and provides summaries of the approaches, organized by both model type and application. Section 4 provides a qualitative evaluation and generalization of the approaches identified in the previous section in regards to their potential for identifying sub-model patterns. Section 5 attempts to identify other techniques and approaches related to the model comparison work discussed in this technical report. Section 6 concludes the technical report.

2 Background information

This section is intended to provide readers that may not have much experience with software modeling or model-driven engineering with sufficient background information so they can understand the remainder of the technical report. It begins, in Section 2.1, with a description of high-level software models. It then continues by describing how these models are used to achieve model-driven engineering in Section 2.2. Within Section 2.2, we describe model versioning, model clone detection, and the general idea and usage of model comparison.

2.1 High-level software models

As both the hardware on our computers becomes more advanced and the complexity of the software tasks that we 'program' on this hardware increases, so does the need for abstracting away implementation details that are at a lower level. At the basic level, a computer is reading what we call *machine language*, that is, electrical signals that are either on or off. However, rather than programming all our tasks in 0 or 1s, it became necessary to hide away, or 'abstract', this language into a higher level language known as *assembly*. As time progressed, however, assembly language was inadequately suited to allow for complex software development, so *third-generation programming languages (3GL)* arose. These are what we commonly refer to as 'Programming Languages' today and include late-3GL languages like C++, Turing, Java, and many others. These languages allow developers to make large systems capable of very advanced tasks, however, managing and implementing the engineering of these systems has proven to be a very challenging endeavour.

One relatively recent approach for designing and implementing complex software systems is representing the artifacts within these systems at an even higher level of abstraction through the use of, what we call, *high-level software models*. These models can describe the structure, behavior, and many other aspects of the software system that would be difficult to explain through programming languages alone. The most popular and widely used example of this is the Unified Modeling Language(UML)[1], which attempts to include the entire spectrum of models. However, there are many others out there, many of which are suited to a specific domain. These high-level software models are expressed through a *modeling language* that expresses a *meta model*. A meta model is another model, typically at the same level of abstraction, that describes a set of valid models. The most notable meta modeling standard is the Meta-Object Facility (MOF) [2] model, which UML happens to be defined in. Most recently, however, is the advent of Domain

[1] http://www.uml.org/
[2] http://www.omg.org/mof/

Specific Models (DSM) [33]. Whereas UML models conform to a single meta model and are meant to be general-purpose, DSMs are models that conform to a meta model that is tailored to the domain that the models are operating within. This domain specific meta model is known as the *Domain-Specific Modeling Language* (DSML). Matlab and Simulink [8] can be seen as DSMLs that are used heavily in the automotive domain for embedded control units.

2.2 Model-Driven Engineering

In Model-Driven Engineering (MDE), or *Model-Driven Development* [75], these High-Level Software Models, or simply *models*, are the main artifacts used by developers during the creation and evolution of software systems. The idea is that, in the same way abstracting machine language into programming languages allows us to efficiently develop and express more complex applications, representing our systems as models rather than programming-language code should allow us to develop more complex applications and express them in a better way. So, in an ideal MDE environment, models are *first-class* artifacts. This means that in all stages of a system's existence, these models are the only things that are created, updated, shared, analyzed, and used. In order to accomplish this, however, these models must be powerful enough and have a sufficient support structure in place so that they are easier to work with than the programming-language code they represent. This includes the ability to automate code generation from models and being able to verify these models are correctly representing the system [75]. Computer-Aided Software Engineering [3] (CASE) is sometimes seen as the predecessor of MDE and is occasionally used as a synonym or more general term for MDE.

While MDE affords many of the benefits discussed and more, these benefits do not come easily or without challenges. Firstly, there are the problems that are unique to the nature of MDE including redundancy of information; the need for "round tripping", or synchronization, between models and code; the possibility that the complexity is being moved rather than reduced; and the need for modelers to have an understanding of the entire system [34]. Aside from these problems that stem from the essence of MDE, there are challenges that arise that are also present when working with programming languages. While analogous, the MDE equivalent of these problems are by no means identical nor trivial and typically require a different approach in dealing with them. Some examples include model transformation [78], cross-cutting concerns in models [23, 26], model versioning/evolution, model clones, and problems encountered with the modeling languages themselves [28, 34, 27]. For the purpose of this technical report, we will elaborate on model versioning and model clones only, as these are topics that

[3] http://case-tools.org/

involve model comparison. This will be done in Section 2.2.1 and Section 2.2.2, respectively.

2.2.1 Model versioning

The need for collaboration amongst teams in MDE projects is just as significant as it is in traditional software projects. Traditional software projects achieve this through Version Control Systems (VCS) such as CVS[4] and Subversion[5]. Similarly, for MDE, it is imperative that modelers are able to work on system models independently but then be able to reintegrate their updated versions back into the main project repository. However, while team member Alice is working on their model(s), team member Bob may be making and submitting changes at the same time to those models that need to be integrated into the latest version and may even conflict with the updates being made by Alice. Consider the example in Figure 1. Firstly, it shows an example of a model, V, being changed simultaneously by one team member to model V', which involves a reference from *Person* to *Passport*, and by another team member to V", which is a model that has combined *Person* and *Passport*. More importantly, however, it shows an example of a conflict that a traditional VCS would not be able to detect. V* contains the changes made by both modelers, which, in this case, leaves us with a "dangling reference" problem [6]. Section 5.1.2 discusses more shortcomings of traditional text-based VCS approaches, however, the important thing to take away from the example is that model versioning is a non-trivial area within MDE.

Model versioning is broken into different sets of phases by different people [2, 5, 36]. However, generally, it is seen to require a form of model comparison or model matching, that is, the identification of what model elements correspond to what other model elements; detection and representation of differences and conflicts; and model merging, that is, combining the changes made to corresponding model elements while taking conflicts into account. So, for example, we could break down the phases of model versioning with respect to our example in Figure 1: model comparison would identify that the model V' and V" refer to the same root model or concept; the conflict detection phase would identify the conflicting changes in V' and V" and, ideally, how to resolve them; a merge of V' and V" would be done by the VCS yielding a correct version of V*. For the purpose of this technical report, we focus mainly on the first phase, model comparison. While the other phases are equally as important, we are mostly interested in the way in which various model versioning approaches achieve model comparison.

[4] http://www.nongnu.org/cvs/

[5] http://subversion.tigris.org

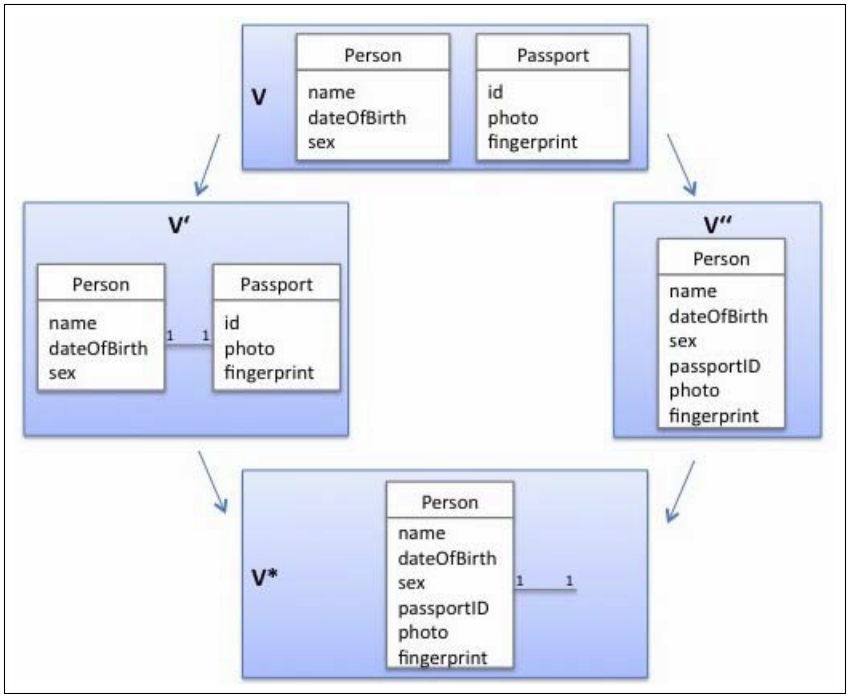

Figure 1: Example of conflicting changes to the same model [6]

2.2.2 Model clone detection

Another example of model comparison being used in a different and specific context is model clone detection. We quickly provide background information in this section about model clone detection as it is discussed in many of the sections that follow.

In 3GL software projects, a *code clone* refers to collections of code that are deemed similar to one another through some measure of similarity [42]. One common reason that code clones exist in these projects include the implementation of a similar concept throughout the same system. The problem with code clones is that a change in this one concept means that the system must be updated in multiple places, which, sometimes, is forgotten or difficult to do. Despite the fact that clones are generally seen as a negative, they are still introduced because of factors like poor reuse practices, time constraints, lack of knowledge about cloning, and others [42]. The research area devoted to the study, identification, evolution, and refactoring of code clones is very mature and there are many techniques and tools that are in existence to deal with them [69].

The analogous problem of *model clone*s is having various sets of models or model elements that are shown to be similar in some defined fashion. While the cause of and the problems that arise from code cloning do not come as a result of the programming languages themselves, techniques intended to deal with code clones [69] are textual in nature and not well suited for model clones. In comparison

to code clone detection, the research related to model clone detection is quite limited [20].

An important consideration about model clone detection is that it is an NP-complete problem because it is a special case of the largest common sub graph problem [30]. This problem entails looking for common sub graphs within a single graph. There are two main approaches that have evolved in the area of model clone detection: Deissenboeck et al. [21] employ a heuristic approach that uses Depth First Search (DFS) to choose the single path that appears the most likely to be a clone match, while Pham et al. [63] propose an algorithm, entitled *eScan*, that uses graph mining algorithms and some domain-(Simulink-) specific knowledge to identify model clones. Both of these approaches are geared towards data-flow modeling, specifically Simulink and Matlab. An example of a clone in data-flow modeling is displayed in Figure 2. The clone set in the two models shown come as a result of the same sub graph being present in both after being normalized, or abstracted, in some fashion. In this case, the normalization has been done such that the specific values within the nodes are ignored / not deemed necessary for similarity [20]. There is also some preliminary work that looks at identifying model clones in UML models [83].

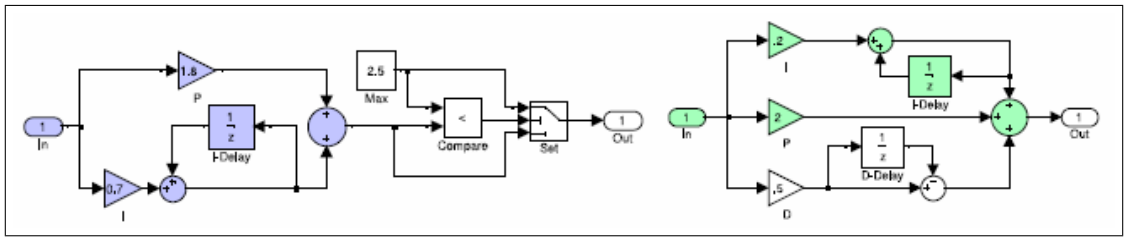

Figure 2: Model clone example [20]

2.2.3 Model comparison

This section discusses model comparison as an individual task in MDE and refers specifically to the act of identifying any similarities or differences between model elements, either automatically or through some user input. Looking at what we discussed already, it is clear that model clone detection employs some notion of model comparison. Specifically, models, or sub models in this case, are being compared and deemed similar or equivalent to one another based on some definition of similarity. The difference between the comparisons done in model clone detection and model versioning is that in model versioning the comparison is typically executed by developers on models that have evolved from, or are a 'version of', the same root model whereas, in model clone detection, the comparisons are executed by analysts on models that are contained within some larger model or group of

models. As noted by Kolovos et al. [39] and Lin et al. [49], model comparison has merit in other areas of MDE including being the foundation for *model composition* and *model transformation testing*. Specifically, for *model transformation testing*, it can be used for comparing the model transformations themselves, when they are represented as models, and for verifying the correctness of the transformation by comparing the input and output models the transformation is working with. Beyond this, model comparison may be useful in performing specific analytical tasks, such as looking for sub-model patterns as we discuss in Section 4.

Kolovos et al. attempt to provide a precise definition of model comparison by saying it is an operation that classifies elements into four categories[39]. Figure 3 illustrates this idea and we use it in order to help explain what we consider to be similar/matching and what is considered to be different/non-matching.

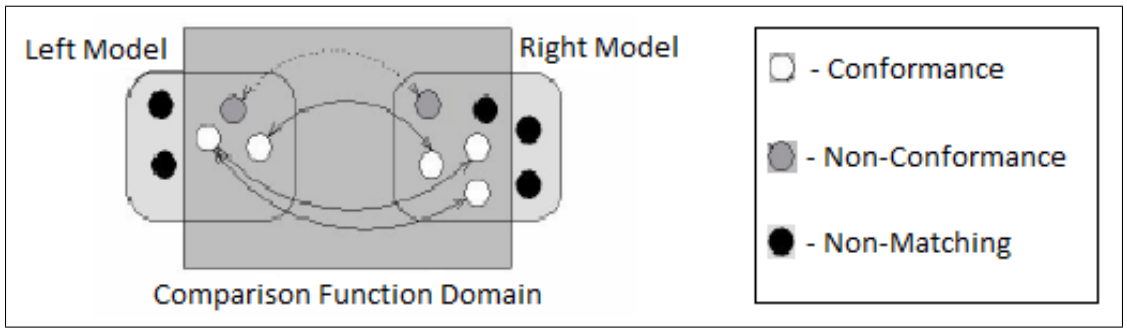

Figure 3: Model comparison definition [39]

Notion of similarity or matching For two elements to be similar, they must match one another based on some criteria or represent the same conceptual idea or artifact. Kolovos et al. further refine similarity to include a notion of *conformance*. With conformance, elements from one model that have a match in another model can be classified as being either elements that conform to their matching element or elements that do not conform to their counterpart in the other model. In Figure 3, white nodes are elements that conform, while grey elements are ones that do not conform. An example of non conformance in a UML class diagram could be when a class in both models has the same name but one is abstract. So while the grey elements still match, that is, likely represent the same artifact, they do 'match enough' or conform to one another [39]. The specific definition of conformance is dependent on the current comparison context and goal. We can attempt to extend this notion of similarity to what we have already discussed. Matching elements in model versioning can be seen as those that are present within the same version tree, that is, are all derived from a common ancestor. Conformance in this context is the case where versions are considered identical or have changes that are consider

trivial. For model clone detection, matching elements can be described as those that represent the same conceptual idea, or node in data-flow modeling, such as the I-Delay block in Figure 2, and conformance can be seen as the difference between elements that match after the normalization process and those that do not.

Notion of difference The idea of elements being different to one another means that they do not match based on some criteria nor represent the same conceptual idea or artifact. Figure 3 represents these different elements as black nodes. Kolovos et al. further classifies non-matching elements into those that are within the domain of the comparison operation, or what we called matching criteria, and those that are not within the domain. Non-matching elements that are not within the domain of comparison can exist due to a case where the matching criteria is either incomplete or intentionally ignoring them. The grey box in Figure 3 represents the domain and black nodes outside of it are those that are not within the domain. We must also attempt to extend our notion of differences to include elements that match but do not conform, that is, the grey elements in Figure 3. The differences in this case can be broken into two categories, as was initially done in work on model versioning [25]: *symmetric delta*s and *directed delta*s. A symmetric delta between two models contains all the elements that are in one model but not the other and vice versa. A directed delta is operational in that it contains all the steps necessary to get from one model to another. Konemann investigates ways of describing differences that are independent of the models they came from [40], which can be useful for patches, and later extends this idea to discuss grouping these differences together semantically in order to generalize them so they can be used in other instances [41]. Kehrer et al. [35] do something similar by trying to derive semantics that come from differences among various versions of a model. It differs from Konemann's work in that Konemann's relies on a lot of user-interaction while the work done by Kehrer et al. is a more automatic approach that can be run before Konemann's.

Phases of model comparison In the context of model versioning, model comparison has been decomposed into three phases [11]: calculation, representation, and visualization. However, we believe these phases can be applied to model comparison in general as there is nothing about this decomposition that is specific to model versioning. In the following paragraphs, we elaborate on the phases and provide examples of approaches.

Calculation Calculation is the initial step of model comparison and is the act of identifying the similarities and differences between two different models. Using the definition we described earlier, it can seen as the classification of elements into

the four categories. Calculation techniques can work with specific types of models, such as done with UML-model comparison methods [2, 32, 36, 60, 89]. However, it is also possible for techniques to do comparisons that are not meta model dependent as demonstrated in an approach that works with Domain Specific Models [48] and other approaches [68, 77, 85]. Kolovos et al. [38] break down calculation methods into four different categories based on how they match corresponding model elements. They are *static identity-based matching*, which uses persistent and unique identifiers; *signature-based matching*, which calculates its comparison approach dynamically based on an element's uniquely-identifying signature that is calculated from a user-defined function; *similarity-based matching*, which uses the composition of the similarity of an element's features; and *custom language-specific matching algorithms*, which is a calculation approach that uses matching algorithms designed to work with a particular modeling language. In the remainder of this technical report we focus mainly on the calculation phase, particularly the model element matching aspect, as this is the phase that will most impact our ability to discover common sub-patterns among models. Also, calculation is the most researched of the three phases.

Representation This phase deals with the underlying form that the differences and similarities detected during the calculation will take. One such approach to representation is the notion of edit scripts [2, 55]. Edit scripts are an operational representation of the changes necessary to make one model equivalent to another and can be comprised of primitive operations such as add, edit, and delete. They tend to be associated with calculation methods that use model-element identifiers and may not be user-friendly due to their imperative nature. In contrast, model-based representation [60] is an another approach that is more declarative. It represents the differences by recognizing the elements and sub-elements that are the same and the ones that differ. Most recently, an abstract-syntax-like representation [11] has been proposed that represents the differences declaratively and also allows further analysis. Cicchetti et al. [15] and, later, Van den Brand et al. [85], propose properties that representation techniques should strive for in order to be separable from calculation approaches and for them to be ideal for MDE environments. Cicchetti et al. [16] then extend this work to demonstrate that their representation is ideal for managing conflicts in distributed environments.

Visualization Lastly, visualization involves displaying the differences in a form that is desirable to the end user. Visualization is considered somewhat secondary to calculation and representation and may be tied closely to representation. For example, model-based representation can be visualized through colouring [60]. Newer visualization approaches try to make visualization more separable from

representation [86, 87, 88]. There is also an approach [71] that attempts to extend visualization techniques for comparing text files to work with models by including additional features unique to higher level representations, such as folding, automatic layout, and navigation.

3 Model comparison methods and applications

This section categorizes existing model comparison methods by the specific types of models they compare and discusses the applications of the comparisons. We decided on organizing the approaches in this manner in order to allow this technical report to act as reference for those working with specific model types and looking to engage in model comparison. We first present model comparison methods that work with more than one type of model in Section 3.1. This is followed by comparison techniques that compare behavioral, structural, product line, and software development process models in Sections 3.2, 3.3, 3.4, and 3.5, respectively. Within each section, we further classify the approaches by the sub type of model they are intended for and also discuss the intended application of each. While some of the methods claim that they can be extended to work with other types of models, we still place each one only in the category they have demonstrated they work with and make note of the claimed extendability.

3.1 Methods for multiple model types

This section looks at approaches that are able to deal with more than one type of model. Approaches like this have the benefit of being more general but are not able to use specific model-type information to their benefit.

3.1.1 UML models

The UML is a meta model defined by the MOF and is the most prominent and well known example of a MOF instance. This section looks at model comparison approaches intended to compare more than one type of UML model.

Alanen and Porres [2, 3] perform model comparison as a precursor to model versioning. Their model matching is achieved through static identity-based matching, that is, it relies on the UML's universally unique identifiers(UUID) contained within each model element. Their work is more focused on identifying the differences between matched models. They calculate and represent differences as directed deltas, that is, operations that will turn one model into the other and that can be reversed through a dual operation. Their difference calculation is achieved through the following steps that results in a sequence of operations

1. Given a model V and V', this algorithm creates a mapping between the matched model elements, which they achieve through UUID comparison.
2. The next two steps involve the algorithm adding operations to create the elements within V' that do not exist in V and then adding operations to delete the elements that are in V that are not in V'.

3. Lastly, for all the elements that are in both V and V', any changes that have occurred to the features within these elements from V to V' are converted to operations that ensure that feature ordering is preserved.

Rational Software Architect (RSA) [6] is an IBM product that is intended to be a versatile and complete development environment for those working with software models and MDE. It works primarily with UML models. Early versions of RSA (6 and earlier) allow for two ways of performing model comparison on UML models, both of which are a form of model versioning: Comparing a model from its local history and comparing a model with another model that belongs to the same ancestor [45]. In both of these cases, model matching is done using UUIDs in that matched elements must have the same identity. The calculation for finding differences between matched elements is not shared/is proprietary.

Figure 4 shows an example of the 'compare model with local history' RSA window. The bottom pane is split between the two versions while the top right pane describes the differences found between the two. The window and process for comparing a model with another model within the same ancestry is very similar. In RSA version 7 and later a facility is provided by the tool for comparing and merging two software models that are not necessarily from the same ancestor [46]. This is accomplished manually through user interaction, that is, there is no calculation really taking place other than the option to do a relatively simple structural and UUID comparison. The user selects elements from the source that should be added to the target, maps matching elements from the source and target, and chooses the final names for these matched elements.

Similarly, another example of a technique that compares UML models and utilizes their UUIDs is the one proposed by Ohst et al. [60]. This technique entails transforming the UML models to graphs and then traversing each tree level with the purpose of searching for identical UUIDs. The technique then takes into account the differences among the matched model elements, such as features and relationships, and continues its traversal.

Selonen and Kettunen [77] present a technique that attempts to accomplish the model matching aspect of model comparison for a variety of UML models by deriving the signature-match rules based on the abstract syntax of the meta model describing the modeling language. Specifically, they say that two models match if they have the same meta class, the same name, and same primary context, which includes the surrounding structure of the model comprised of neighbours and descendants. This matching criteria can be relaxed depending on the scenario. They state that this technique should be extendable to work with any MOF-based modeling languages. Also, additional rules can be added by extending the meta model with appropriate stereotypes that he technique that interpret.

[6] http://www.ibm.com/software/awdtools/architect/swarchitect/

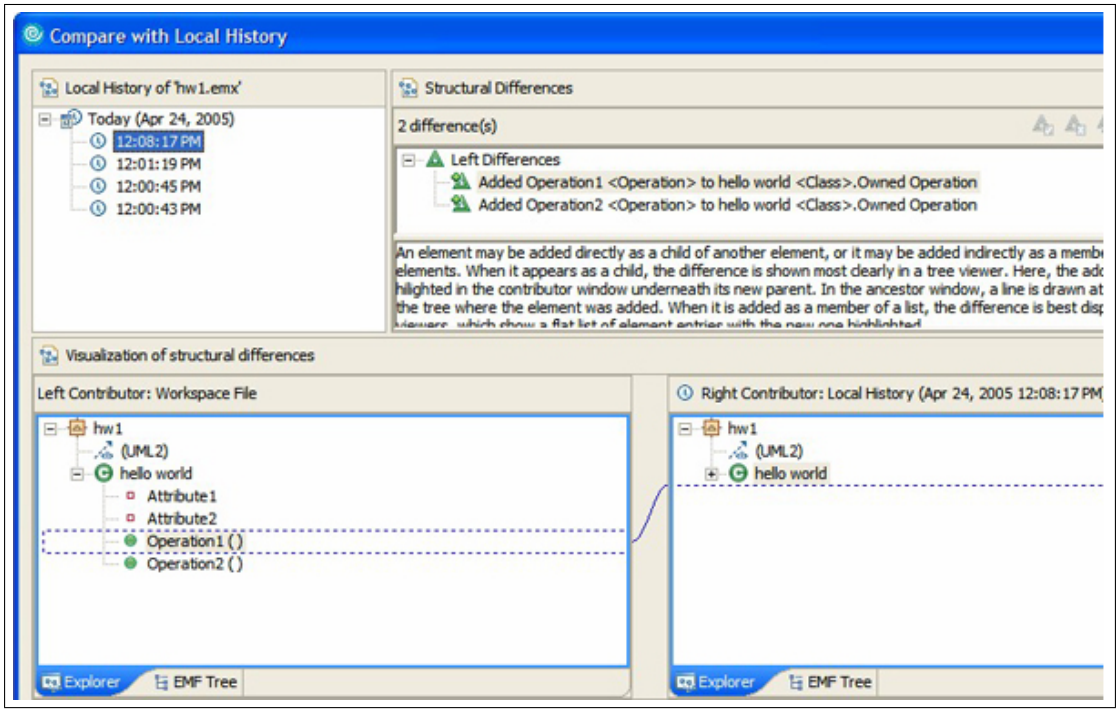

Figure 4: RSA 'Compare Model with Local History' window (adapted from [45])

Odyssey VCS (Version Control System) [61] is a model versioning tool that is intended to work with all types of UML models in CASE environments. It does not perform model matching as all elements are linked to a previous version, starting with a baseline version. Differences or conflicts are detected by processing XMI files and using UML-specific knowledge to calculate what elements have been added, modified, or deleted. A class can be defined as a unit of comparison (UC) for one given project, while attributes and operations may be defined as the UCs in another project. This flexibility provided by Odyssey-VCS allows for a more precise definition of configuration elements. Specifically, users can configure a behavior descriptor that determines which elements are UC and which are UV (unit of versioning).

Strolle [83] developed the *MQlone* tool in order to experiment with the idea of detecting UML model clones. They convert XMI files from UML CASE models and turn them into Prolog [7]. Once in Prolog, they attempt to discover clones by trying both static identity matching and some similarity metric matching, such as size, containment relationships, and name indexing/similarity. The potential clones are then reported back to the user.

[7] www.swi-prolog.org

3.1.2 EMF models

Eclipse Modeling Framework (EMF) models [8] are MOF meta-meta models that can be used to define meta models, such as UML. They are intended for use within the Eclipse development environment.

EMFCompare [11] is an Eclipse project that performs model comparison on EMF models. Rather than relying on EMF models' UUIDs, they choose to use similarity based-matching to allow the tool to be more generic and useful in a variety of situations. The calculation for matching is based on various statistics and metrics that are combined to generate a match score. This includes analyzing the name, content, type, and relations of the elements and returning a value from 0 to 1, which is later combined with "additional factors". They also filter out element data that comes from default values. It should be noted that while *EMFCompare* is specific to EMF models, the underlying calculation engine is meta model independent, similar to the approaches discussed in Section 3.1.3

TopCased [24] is a project providing an MDE environment that uses EMF models and is intended specifically for safety critical applications and systems. It performs its matching and differencing in a similar fashion to Alanen et al., using static identity-based matching.

SmoVer [65] is another model versioning tool that can work with any EMF-based model. In this approach they do both version-specific comparisons like Odyssey VCS, termed syntactical, and semantic comparisons. Semantic comparisons are those that are carried out on semantic views. Semantic views, in this context, are the resulting models that come from a user-defined model transformation that SmoVer executes on the original models being compared in order to give the models meaning from a particular view of interest. These transformations are specified in the Atlas Transformation Language (ATL) [9]. Thus, the models are first compared syntactically, then transformed, then compared once again. Matching is accomplished through static-identity based matching and differences are calculated by comparing structural changes, such as attribute, reference, role, and its referenced elements' updates. The authors note that the amount of work required for creating the ATL transformations is not too large and that "the return on investment gained in better conflict detection clearly outweighs the initial effort spent on specifying the semantic views." The examples they discussed required roughly 50 to 200 lines of ATL transformation language code.

Riveria and Vallecillo [68] employ similarity-based matching in addition to checking persistent identifiers for their tool, *Maudeling*. They use *Maude* [17], a high-level language that supports rewriting-logic specification and programming,

[8] http://www.eclipse.org/emf/
[9] http://eclipse.org/atl/

to facilitate the comparison of models from varying meta models specified in EMF. They provide this in the form of an Eclipse plugin called *Maudeling*. Using Maude, they specify a difference meta model, shown in Figure 5, that represents differences as added, modified, or deleted elements that inherit from **DiffElement**. Matching is accomplished through the **Match** meta model, shown in Figure 5. Each match element represents the link between two elements, or objects, that represent the same thing. Rate is a ratio between 0 and 1 that represents their similarity. The process for discovering this ratio first begins by using UUIDs. If there is no UUID match, they then rely on a variant of similarity-based matching that calculates the similarity ratio by checking a variety of structural features. Such features include shared inheritance relationships; weighted structural features, as done in *SiDiff*; numerical attributes; and references. Then, to calculate differences, they go through each element and categorize it based on one of four scenarios: it appears in both models A and B and has not been modified, it appears in both models A and B and has been modified, it appears in model A but not model B, or it appears in model B but not model A. It then creates a difference model based on this information that conforms to its difference meta model. There is no user work required as *Maudeling* has the ATL transformations within it that transform the EMF models into their corresponding Maude representations. Operations are then executed in the Maude environment automatically, meaning that there is no reason why the user will have to work with the Maude representation of models or meta models.

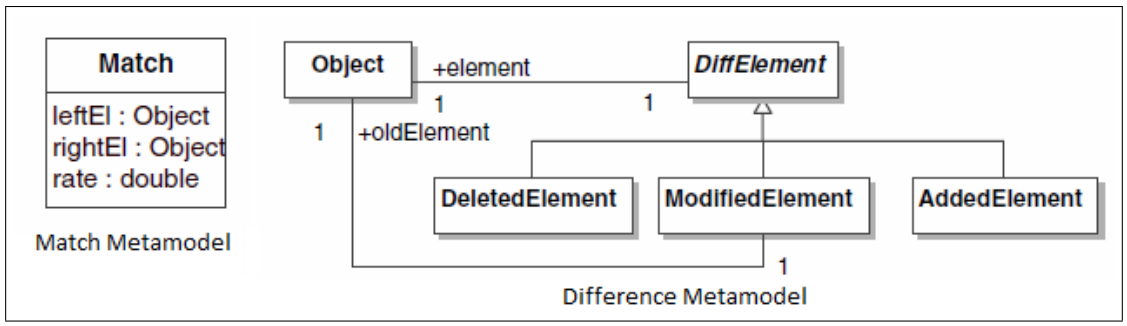

Figure 5: Meta models from Maudeling. Modified from [68]

3.1.3 Metamodel-agnostic approaches

This section discusses approaches that deal with models that can conform to an arbitrary meta model that adheres to specific properties.

Examples of Metamodel-independent approaches that use similarity-based matching stratigies include the *Epsilon Comparison Language* [37] and *DSMDiff* [48]. *DSMDiff* is an extension of the work done on UML model comparison

techniques except it works with DSMs. *DSMDiff* uses both similarity- and signature- based matching in order to discover matched model elements. The similarity-based matching focuses on the similarity of edges among different model nodes. *DSMDiff* evaluates differences between matched elements and considers them directed deltas, namely new, delete, and change. While *DSMDiff* was developed using DSMLs specified in the Generic Modeling Environment (GME), the techniques within it can be extended to work with any DSML creation tool. The creators of *DSMDiff* also discuss allowing user-interaction that will allow one to choose the mappings (matches) from a list of applicable candidates.

ECL was developed after *DSMDiff* and *SiDiff*, which is discussed later, and attempts to address the fact that its predecessors do not allow for modellers to configure language-specific information that will assist in matching model elements from different meta models [37]. It attempts to accomplish this in a imperative yet high-level manner. ECL allows modelers to specify model comparison rule-based algorithms and then execute them to identify matched elements within different models. The trade off in doing this however is that, while complex matching criteria can be expressed using ECL, it requires more time and a knowledge of ECL. Kolovos acknowledges that metamodel-independent approaches that use similarity-based matching typically perform fairly well, but there often are "corner cases" that ECL is well suited to identify.

Mehra et al. [53] developed a plug-in for meta-Case applications that can perform model comparison for any set of models that can be defined by a meta-Case tool. Meta-Case applications are those that operate similarly to CASE applications, except they are not constrained by a particular schema or meta model. This plugin first matches all the elements by their unique identifiers and then calculates the differences by representing them as directed deltas. It is intended to be used for model versioning. Oda and Saeki [59] describe a graph-based VCS that is quite similar in that it works with meta-Case models and matches them using baselines and unique identifiers. Differences are calculated as directed deltas with respect to earlier versions.

Nguyen [58] proposes a VCS that can be used for model versioning that can detect both structural and textual differences between versions of a wide array of software artifacts. This approach utilizes similarity-based matching by assigning all artifacts an identifier that encapsulates the element and represents the artifacts as nodes within a directed attributed graph, similar to model clone detection approaches.

Van den Brand et al. [85] argues that current meta-modeling techniques, such as MOF, are not able to satisfy a reasonable set of requirements, defined by them, that a difference representation should satisfy. As such, they present their own meta-modeling technique and define differences with respect to it. They then provide a model comparison approach and prototype that allows the user to configure

which or what combination of the four model matching strategies they want to employ. The authors provide examples where they extend the work done by Kelter et al. [36] but then combine it with other matching techniques, like using a UUID. This generality comes at a cost of a large amount of configuration, work, and user-interaction.

There are methods that translate models into another language or notation that both maintain the semantics, or meaning, of the models and are then able to facilitate model comparison. One example is the work done by Gheyi et al. [31] in which they propose an abstract equivalence notion for object models, in other words, a way of representing objects that allows them to be compared. The key in achieving this and preserving the semantics is their use of an alphabet, which is the set of relevant elements that will be compared, and views, which are mappings that express the different ways that one element in one model can be interpreted by elements of a different model. They then define equivalence between two models being the case when, for every interpretation or valid instance that satisfies one model, there exists an equivalent interpretation that satisfies one in the other model. They illustrate their approach using Alloy models [10], which are models based on first-order logic including facts and predicates, and demonstrate that it is abstract enough to work with other modeling languages. Similarly, Maoz et al. [52] present the notion of *semantic diff operators*, which represent the relevant semantics of each model, and *diff witnesses*, which are the semantic differences between two models. Semantic diff operators are composed by leveraging existing work on representing the semantics of both class (structure) and activity (behavioral) diagrams through the use of mathematical formalisations. Diff witnesses are generated that represent the semantic object model that is in one model and not the other, for class diagrams, and represent the path that is possible in one model but not the other, for activity diagrams. From these ideas, Maoz et al. provide the tools *cddiff* and *addiff* to do class diagram differencing and activity class diagram differencing, respectively.

Lastly, the *Query/View/Transform*(QVT) standard provides the outline for three model transformation languages. These languages act on any arbitrary model or metamodel as long as they conform to the MOF specification. One of these languages, *QVT-Relations*(QVT-R) allows for a declarative specification of two-way (bi-directional) transformations and, according to Stevens [82], is more expressive than the other QVT languages. Specifically, Stevens discusses how using game-theoretic semantics for QVT-R facilitates a better semantic definition. This improvement allows for a form of model comparison through its *checkonly mode*, which is the mode when models are being checked for consistency rather than making changes. In brief, game theory is applied to QVT-R by using a verifier and refuter. The verifier attempts to confirm that the check will succeed and the refuter's objective is to disprove it. The semantics of QVT are then expressed

[10] http://alloy.mit.edu

in such a way that implies "the check returns true if and only if the verifier has a winning strategy for the game". This improvement would allow one to compare models assuming we expressed a transformation in this language that represented the differences or similarities we were looking for. In this case, the transformation would yield true if the pair/set of models are consistent according to the transformation.

3.2 Methods for behavior/data-flow models

3.2.1 Simulink and Matlab models

This section looks at model comparison techniques that work with Simulink and Matlab models.

CloneDetective [21] is an approach used to perform clone detection in models. It uses ideas from graph theory and is applicable to any model that is represented as a data-flow graph. It is comprised of three steps. The first step taken by *CloneDetective* includes pre processing and normalisation. Pre processing involves flattening, or inlining, all of the models and removing any unconnected lines. Normalisation takes all of the blocks and lines found within the models and assigns them a label that consists of information that is considered important for comparing them. The information described in the label changes according to the type of clones being searched for by the tool. Figure 6 displays the result of this step on the models presented earlier in Figure 2, including the labels that come from normalisation, such as **UnitDelay** and **RelOp:**⟨. The grey portions within the graph represent a clone. This is accomplished by *CloneDetective* in its second phase: clone pair extraction. This phase is the most similar to the model matching discussed thus far, however, rather than matching single elements it is attempting to match the largest common set of elements. It likely falls in the category of similarity-based matching because the features of each element are extracted, represented as a label during normalisation, and compared.

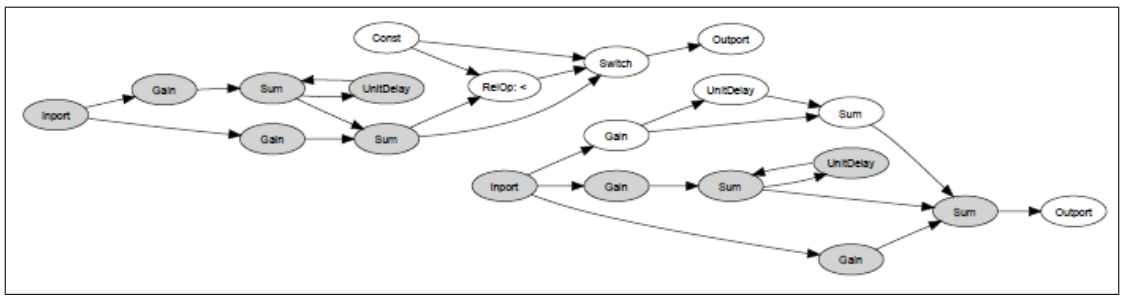

Figure 6: Normalised model graph of model clone example [21]

Similarly, *eScan* and *aScan* algorithms attempt to detect exact-matched and approximate clones, respectively [63]. Exact-matched clones are groups of model elements having the same size and aggregated labels, which contain topology information and edge and node label information. Approximate clones are those that are not exactly matching but fit some similarity criteria. *aScan* uses vector-based representations of graphs that account for a subset of structural features within the graph. The main difference between these algorithms and *CloneDetective* is that these algorithms group their clones first and from smallest to largest. They claim that this will help detect clones that *CloneDetective* can not. This is later refuted, however, by the authors of *CloneDetective* [20]. *aScan* is able to detect approximate clones while *CloneDetective* is not. Much like *CloneDetective*, these algorithms utilize similarity-based matching.

Al-Batran et al. [1] notes that the previous two approaches discussed deal with syntactic clones only, that is they can detect syntactically/structural similar copies only. Using normalization techniques that utilize graph transformations, they extend these approaches to cover semantic clones that may have similar behavior but different structure. So, a clone in this context is now defined as two (sub)sets of models that have "equivalent unique normal forms" of models. These unique normal forms are acquired by having any of the existing tools, such as *CloneDetective*, perform forty semantic-preserving transformations, some of which are presented in their paper and are structural modifications on Simulink models. It was found that extending the clone detection strategy in this way yields more clones than simple syntactic comparison.

3.2.2 Sequence diagrams

Ren et al. [66] devised a technique to refactor sequence diagrams to remove duplication. They do not actually detect duplication. Instead, they describe various refactorings that can be done once duplication is discovered: namely creating, deleting, changing, and moving a sequence diagram element. Liu et al. [50] attempt to complete this work by actually detecting duplication in sequence diagrams. They convert sequence diagrams into an array and represent that array as a suffix tree. A suffix tree is a compressed tree that contains all the ending elements of a string organized in an optimal way for performing search operations. The suffix tree is then traversed and duplicates are extracted by looking for the longest common prefix, or elements that lead to the leaf node, of two suffixes. Duplicates are defined as a set of sequence diagram fragments that contain the same elements and have the same sequence-diagram specific relationships. Figure 7 demonstrates an example of a duplicate. In this case, the largest common prefix included the 4 elements highlighted within each diagram: **login requirement**, **login page**, **id and password**, and **success**. The image is taken from their tool

DuplicationDetector. Much like the model clone detection approaches discussed, this technique employs a variation of similarity-based matching as it is comparing a graph representation of a fragment's features/elements.

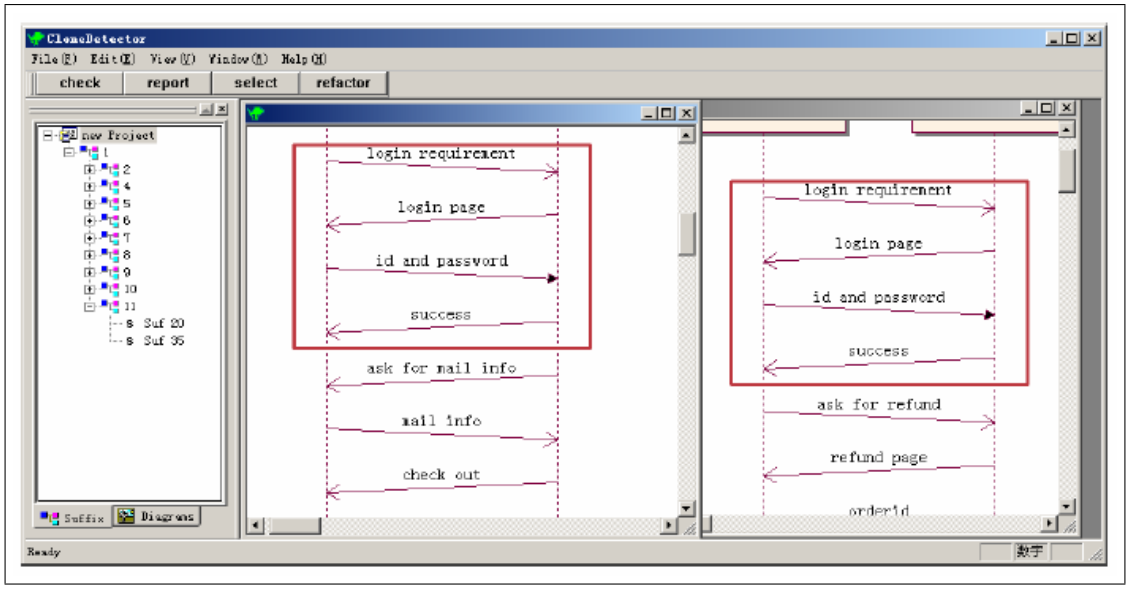

Figure 7: Duplicate sequence fragment [50]

3.2.3 Statechart diagrams

Nejati et al. [56] matches state chart diagrams for the purpose of model merging. They accomplish this by using heuristics that include looking at terminological, structural, and semantic similarities between models. The heuristics are split into 2 categories: static heuristics that use attributes that have no semantics, such as the names or features of elements; and behavioural heuristics, which find pairs that have similar dynamic behavior. Due to the use of heuristics, this approach requires a domain expert look over the relations and add or remove relations, accordingly, to acquire the desired matching relation. A desireable relation is one that does not yield too many false positives or false negatives. This approach employs both similarity-based matching through static heuristics and custom-language specific matching through dynamic heuristics. The results are combined and weighted equally by the approach when calculating overall similarity. After similarity is established between two state charts, the merging operation calculates and represents differences as variabilities, or guarded transitions, from one model to the other.

3.3 Methods for structural models

This section discusses approaches that are designed explicitly to work with models that represent the structure of a system. They benefit from the domain knowledge gained by working with structural models only.

Early work on comparing and differencing software structural diagrams was done by Rho and Wu [67]. They focused on comparing a current version of an artifact to its preceding ancestor. They attempt to define the characteristics of such diagrams and outline and describe the potential directed deltas within this domain.

3.3.1 UML structural models

UMLDiff [89], by Xing and Stroulia, uses custom language-specific matching in that it uses name-similarity and UML structure-similarity to identify matching elements. These metrics are combined and compared against a user-defined threshold. It is intended to be used as a model versioning reasoner in that it is supposed to discover changes made from one version of a model to another and assumes that when making its calculations.

UMLDiff$_{cld}$ [32], by Girschick, is a tool that focuses on UML class diagram differencing. It uses a combination of static identity-based and similarity-based matching within its evaluation function, which measures the quality of a match. Similarly, *Mirador* [7] is a plugin created for the Fujaba (From Uml to Java and Back Again)[11] tool suite that allows for user directed matching of elements. Specifically, it allows users to select match candidates that are ranked according to a similarity measure that is a combination of static identity-based and similarity-based matching, like *UMLDiff*$_{cld}$.

Reddy et al. [64] uses signature-based matching in order to compare and compose two UML class models in order to assist with Aspect-oriented modeling [23, 26]. Aspect-oriented modeling is a modeling technique that allows separating crosscutting concerns from other features. In this comparison approach, models are matched based on their signatures, or property values associated with the class. Each signature has a signature type, which is the set of properties that a signature can take. Using KerMeta[12], a model querying language, the signatures used for comparison are derived by the tool based on the features it has within the meta model.

Beradi et al. [9] translate UML class diagrams into ALCQI, a "simple" description logic representation. They show that it is possible for one to reason

[11] http://www.fujaba.de/
[12] http://www.kermeta.org

about UML class diagrams as ALCQI description logic representations, or DLs, and they provide an encoding from UML class diagrams to ALCQI. While the translation does not maintain the entire semantics of the UML classes, it preserves enough of it to be able to check for class equivalence. Equivalence is defined as the case where two classes represent "the same set of instances whenever the requirements imposed by the class diagram are satisfied". Because they use UML specific semantics, we can argue that they are using custom-language specific matching.

Maoz et al. [51] later extend their work on semantic differencing discussed previously and provide a translation prototype, called *CD2Alloy*, that converts UML classes into Alloy. This Alloy code specifically contains constructs that determines the matching elements of two UML class diagrams and also perform semantic comparisons such as if one model is a refinement of another. It can also be considered to be an instance of a custom-language specific comparison matching approach because of its use of UML semantics.

3.3.2 Metamodel-agnostic approaches

Preliminary work on model comparison was done by Chawathe et al. [12] in which they devised a comparison approach intended for any structured document. They convert the data representing the document structure into a graph consisting of nodes that have identifiers that are based on the corresponding elements they represent. This approach, which is analogous to the model clone detection techniques, uses similarity-based matching and describes differences in terms of directed deltas.

SiDiff [36] is very similar to *UMLDiff* except *SiDiff* uses a simplified underlying comparison model in order to be able to handle any models stored in XML Metadata Interchange (XMI) format. Therefore, *SiDiff* can work with any models defined by a meta model that uses XMI. This approach is described by the authors in relation to UML class diagrams, however, they state that it can be extended to state charts as well. Similarly to *UMLDiff*, it uses similarity-based metrics. In contrast to *UMLDiff*, as shown by the up arrow in Figure 8, its calculation begins bottom-up by comparing the sub elements of a pair of model elements starting with their leaf elements. This is done with respect to the elements' similarity metrics. An example of a weighted similarity is having a class element consider the similarity of its class name weighted the highest followed by operations, attributes, and generalizations/interfaces. So, in the case of the two **Class** elements being compared in Figure 8, all of the **Classifier** elements are compared first. If a uniquely identifying element is matched, such as a class name, they are immediately identified as a match. This is followed by a top-down propagation of this matching pair. Figure 8 shows the algorithm finding a match and stopping

the bottom-up comparison at **Class** and, subsequently, beginning its top-down propagation at this point. This top-down approach allows for the algorithm to deduce differences by evaluating the correspondence table that is the output of the matching phase. Schmidt [73] demonstrates that SiDiff can be used to construct a standard framework for detecting model differences for version management tools. Treude [84] improves the data structure used within SiDiff in order to increase the SiDiff algorithm's speed. Because the elements in this algorithm are compared in a type wise manner, it requires the users to define a function that compares two elements of the same type. This function should return between 0 and 1, where 0 means no similarity and 1 means extremely similar. This is accomplished by having users setting up this function/similarity criteria for each element type in a configuration file.

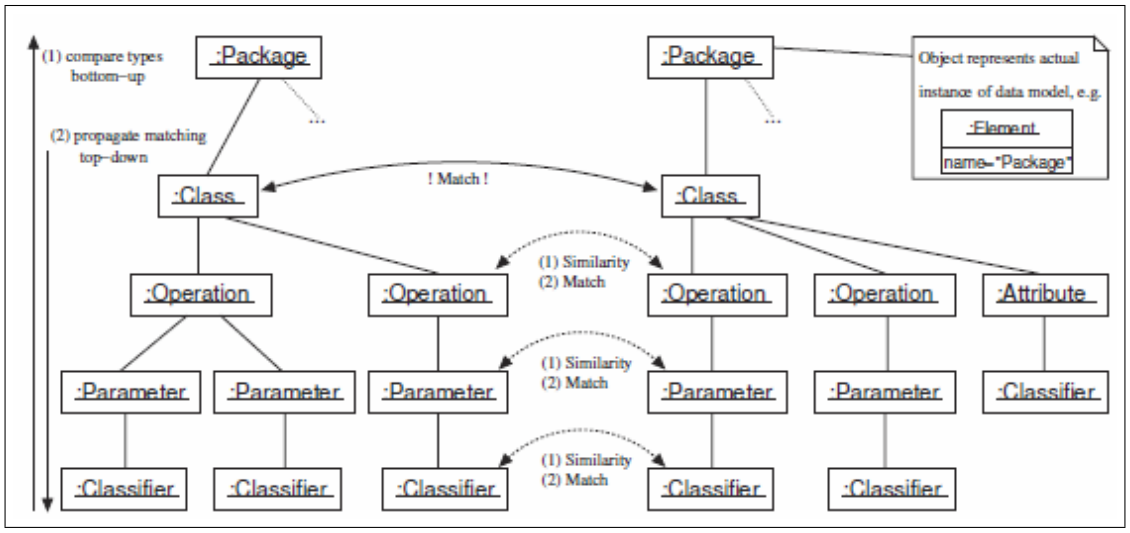

Figure 8: Example of SiDiff Comparison [36]

Similarly to the translation of UML class diagrams into ALCQI, d'Amato et al. [19] propose a comparison measure for description logics, such as those used in the Semantic Web [13], using existing ontology, or concept and relation, semantics. They describe a semantic similarity measure that is able use the semantics of the ontology that the concepts refer to.

3.4 Methods for product line architectures

Chen et al. [14] presents work intended to do comparisons of product line models for the purpose of merging. The assumption in this work is that the comparison is

[13] http://semanticweb.org/

being done on one version of an artifact with respect to another version of the same artifact. Comparison is done recursively and is more fine grained as the algorithm gets deeper into the product line hierarchy. This approach employs similarity-based matching: lower elements in the hierarchy compare interfaces, optionality, and type; and higher level elements compare the elements contained by them. Differences are represented as directed deltas.

Rubin and Chechik [70] devise a framework for comparing individual products that allows for them to be updated automatically to a product line conforming to those used in product line engineering. They use similarity-based matching, that is, products are viewed as model elements and a match is defined as the case where a pair of model elements have features that are similar enough to be above a defined weighted threshold. Specifically, they have a heuristic function that determines the similarity between two elements, represented as a number between 0 and 1, based on the comparison of each element's features and the empirically determined weights assigned to each feature. The authors note that "(their) refactoring framework is applicable to a variety of model types, such as UML, EMF or Matlab/Simulink, and to different compare, match and merge operators". The foreseeable user work that would be required in using this framework would be having a domain expert specify the similarity weights and features of interest and also determining the optimum similarity thresholds.

3.5 Methods for process models

Soto and Munch [80] discusses the need for ascertaining differences among software development process models and outlines what such a difference system would require. They devise an approach called *Delta-P* [79], which they also argue can work with various UML models. *Delta-P* converts process models into Resource Description Framework (RDF) [14] notation, in order to have them represented in a normalized triplebased notation, and performs an identity-based comparison and calculates differences. They employ static-identity based matching as they use unique identifiers. RDF is chosen because it is inexpensive and easy to convert to from models, the models are easily understandable to humans, and they already have a notation for expressing patterns in RDF graphs known as *SPARQL*. Differences are represented as directed deltas, however, they provide examples of where they are able to group directed deltas together to form higher level deltas.

Similarly, Dijkman et al.[22] discuss three similarity metrics that will help compare stored process models: node matching similarity, which compares the labels and attributes attached to process model elements; structural similarity, which evaluates labels and topology; and, behavioral similarity, which looks at

[14] http://www.w3.org/RDF/

labels in conjunction with causal relations from the process models. According to the authors, "structural similarity slightly outperformed the other two metrics." They have no tool developed yet.

3.6 Summary

Table 1 on the next page summarizes the approaches discussed in this section organized by the type and sub type of model they are able to compare. It is clear from Table 1 that similarity-based matching is the most commonly employed strategy by the techniques investigated in this technical report. Most approaches, especially those that are more recent, are able to work with any model that conforms to an arbitrary meta model. This is assuming the meta model conforms to approach-specific requirements, such as the existence of unique identifiers or subscription to a particular meta-meta modeling language. This result is consistent with the recent trend in domain-specific modeling.

Lastly, the majority of work in model comparison thus far appears to be geared towards model versioning, however a number of these comparison methods can be extended for other purposes. The last column attempts to showcase the relative amount of user work required to accomplish model comparison. This is based on the research performed for this technical report. The justification for each tool that has 1 * or more can be found in the textual descriptions provided previously. ECL and the approach presented by Van den Brand et al. require the most amount of user work, however this is intentional to allow for more power and generality. Many approaches require no user work or interaction as they operate under specific conditions or are dynamic enough to understand the context or meta models they are working with.

Table 1: Summary of Model Comparison Approaches

Types of models	Subtype of model	Specific approach/ tool	Matching strategy+	Primary use#	WR4C~
Multiple types of models	UML Models	Alanan	SI	MV	-
		Rational Software Architect (Same/Different Ancestor)	SI	MV and MM	-/**
		Ohst	SI	MV	-
		Odyssey-VCS	-	MV	**
		Selonen	SIG	MM	*
	EMF Models	EMF Compare	SIM	MV	-
		TopCased	SI	MV	-
		SMoVer	SI	MV	**
		Maudeling	SI and SIM	MV	-
	Metamodel-Agnostic (Independent)	Epsilon Object Language	SIM	MM and MTT	***
		DSMDiff	SIM	MTT and MV	*
		Mehra-MetaCase	SI	MV	-
		Van den Brand	SI, SIM, SIG,CLS	MV and MM	***
		Nguyen	SIM	MV	-
		QVT-R	SI	MTS	**
Behavioural or Data-Flow models	Mathlab/Simulink	CloneDetective	SIM	MCD	-
		eScan / aScan	SIM	MCD	-
	UML Sequence Diagram	Liu	SIM	MCD/VER	-
	Statecharts	Nejati	SIM and CLS	MM	*
Structural models	UML Models	UMLDiff	CLS	MV	-
		UMLDiff_{cld}	SI and SIM	MV	-
		Reddy	SIG	AOM	-
		Mirador	SIM	MV and MM	**
		CD2Alloy	CLS	GC	-
		Convert to ALCQI	CLS	GC	-
	Metamodel-Agnostic (Independent)	Chawathe	SIM	MV	-
		SiDiff	SIM	MV	**
Product Line architectures	Any PLA	Chen	SIM	MV	-
		Rubin	SIM	MM	*
Process models	Software Development Process Models	Delta-P	SI	MV	-

Legend
+ - SI = Static-identity based, SIM=Similarity based, SIG=Signature based, CLS=Custom-language specific
# - MV=Model Versioning, MM=Model Merging AOM= Aspect-Oriented Modeling, VER=Verification MTT=Model Transformation Testing , MTS=Model Transformation Specification, GA=General Comparison
~ - Work required for comparison (WR4C) Scale:
- = Very usable. No user work required for comparison. …
*** = Difficult to use. Inordinate level of user work required.

4 Potential for pattern analysis

An interesting idea in MDE is providing software engineers or researchers the ability to extract common sub-structures or patterns from a set of models. This kind of analysis can improve existing projects by providing refactoring suggestions, helping with discovering best practices and antipatterns within MDE, and advancing the field of MDE in general. To accomplish this, one would need to be able to perform a type of model comparison that is tailored to this specific task.

There are two different types of pattern analysis within MDE projects. The first is inferring patterns or antipatterns by looking at existing projects and the second is detecting existing patterns or antipatterns within projects. The output of the first pattern analysis type can be the input to the second. We briefly look at both in the following sections.

4.1 Inferring patterns

Looking at the approaches we discussed in the previous section, there are a number of observations that we can make. Firstly, approaches using static-identity based matching will not be of use. This is because we are not looking for elements that represent the same concept, but rather had a similar intention in being constructed in a particular way. Model clone detection techniques could yield some interesting results as they are already designed to match the largest common sub sequence of elements. If we look at the semantics of the clones extracted, likely through the assistance of a domain-expert and the techniques discussed by Al-Batran et al. [1], we may be able to deduce pattern- or antipattern-like subgraphs. It would be interesting to compare the exact clones found through *CloneDetective* or *eScan* to approximate clones found through *aScan*. We may not want exact clones because they may indicate a true "copy and paste scenario", while approximate clones may represent a non-trivial pattern or antipattern. It should be noted that, currently, model clone detection tools work only with behaviour models [20] because it is difficult to strictly define the notion of a clone for structural models, according to the authors. However, we have discussed in the previous section approaches that detect similar elements within structural and other types of models. Of these, similarity-based matching looks the most promising. We could tailor similarity metrics to fit the types of patterns or sub-structures that we are looking for and maybe even extend these techniques to consider some kind of semantic information or definition. Lastly, we could investigate the possibility of extending some of the behavioral similarity heuristics used for state charts [56], depending on the types of models we are looking at.

It may be hard to define what we are looking for in terms of patterns or antipatterns. A good starting point could be to think of model patterns or

antipatterns that are analogous to those associated with software code [10, 29, 72]. It is important to note, that if we do not know what patterns or antipatterns we are looking for, then graph based approaches may be inefficient and we may have to look at inference techniques from other domains, such as grammar inference [62].

4.2 Detecting known patterns

This type of pattern analysis may be easier to facilitate than inference. We could likely leverage some of the work done in model versioning by expressing the patterns or antipatterns as a model that can be compared to other models within projects. Given a threshold, models that correspond sufficiently will be seen as being instances of the pattern or antipattern we are searching for. A variation of this was accomplished by us previously [81] when we were able to express Java Enterprise Edition (J2EE) antipatterns as a framework specific model and check for antipattern instances among specific J2EE framework instances. However, this approach works only with software frameworks. Extending the approaches discussed in the previous section would require a non-trivial amount of work as the majority of them deal with a single element at a time. Early model clone detection approaches would not be very useful in their current form. However, as shown in the extension done by Al-Batran et al. [1], we may be able to use the semantic-based techniques they discuss to establish a correspondence between known patterns and a normalized representation of our models. For example, if we wanted to detect the *Composite Pattern*[29] or other design patterns, we could represent the pattern in a normalized form and look for both syntactic or semantic matches in our models.

5 Related work

5.1 Other comparison approaches

This section briefly discusses other comparison techniques and why they are not transferable to the modeling domain.

5.1.1 Code comparison techniques

There is an abundance of work done in comparing software artifacts at the level of programming code. In Section 2.2.1 we illustrated just one example of why traditional versioning comparison approaches are insufficient. Altmanninger [6] presents many more including incorrect conflict detection, conflict resolution, and merged model versions.

5.1.2 Textual model comparison

There are approaches that serialize models to text. Cobena et al. [18] attempt to check differences in versions of XML documents while *Xlinkit* [57] compares XMI representations of models. Both of these approaches suffer from working on too low a level of abstraction in that they can not account for model-specific features such as inheritance and cross referencing [39]. *Xlinkit* results are also quite large and difficult to work with compared to the techniques discussed in this technical report.

5.1.3 Code clone detection

As discussed earlier, code clone detection techniques [69] have difficulty or are unable to account for model-specific relations since they are strictly textual. Also, the notion of a code clone and model clone can be seen as two different things.

5.1.4 Translation to formalisms

A number of examples where models were translated into formalisms capable of performing comparison were discussed in this technical report including the work by Gheyi et al. [31] and Berardi et al. [9]. However, there are other examples of models being translated into such formalisms including UML state machines into Promela/SPIN models [13, 44, 47] and, likely, others. While this material is related, we focused mostly on approaches that were intended to performing model comparison and differencing explicitly.

5.2 Graph comparison

Similarity Flooding [54] is an example of a graph matching algorithm. It can be seen as a similarity-based matching approach in that it uses the similarity of an element's neighbouring nodes to discover matching elements. If two node's neighbours are similar then their similarity measure increases. The initial similarity is calculated using nodes' names. The problem with this approach is that it works on too high of a conceptual level and is not able to use diagram- or context- specific information [25]. This is unacceptable for our purposes because it is unlikely that meaningful matches or patterns will be discovered with very little or no context. A similar problem is experienced by the method presented by Krinke [43] in which similar code is discovered through program dependency graph analysis.

5.3 Related survey papers

This section discusses and contrasts other reviews that overlap the model comparison review done in this technical report.

Altmanninger et al. [6] perform a survey on model versioning approaches. This survey differs from the work done in this technical report in that they focus very little on matching, investigate versioning systems only, and discuss only a subset of the model comparison approaches discussed in this technical report. Furthermore, they are much more concerned with merging than comparison.

Sebastiani and Supiratana [74] provide a summary of various differencing approaches, however, they look only at 3 specific approaches at a high level and compare them: *SiDiff*, *UMLDiff*, *DSMDiff*. This is done with a focus on how these 3 techniques can trace a model's evolution.

Lastly, Selonen [76] performs a brief review of UML model comparison approaches. It looks at 5 specific approaches, all of which are covered in this technical report. It compares them by looking at their various properties including the evaluation provided with each tool, if they are dependent on identifiers, how customizable they are, and others. Similar to this technical report, they also note the various usage scenarios or applications of the approaches. In contrast, this technical report looked at techniques that deal with various types of models and included additional UML diagram approaches not identified by Selonen. Also, this technical report looked at model matching and differencing in a lot more detail than Selonen's survey.

6 Conclusion

Model comparison is a relatively young research area that is very important to MDE. It has been implemented in various forms and for various purposes, predominantly model versioning, merging, and clone detection.

We provided an overview of the work done in model comparison, focusing on the type and sub types of models that the approaches work for, the intended application of the approaches, and the way in which they accomplish the model matching aspect of model comparison. We observed that the majority of recent approaches attempt to work with models from an arbitrary meta model. Also, similarity-based matching is the approach taken by most of the model matching methods discussed in this technical report. Lastly, model versioning appears to be the most common goal in performing model comparison up to this point. However, much of the comparison work done can be modified for other purposes. Some approaches require more user effort to perform model comparison, however this is to facilitate more power and generality. Many of the approaches require no user interaction because they operate under specific constraints or are dynamic enough to deal with multiple situations. Based on our qualitative evaluation of the approaches, we believe that methods that use similarity-based matching, such as model clone detection, would be the most adept in inferring useful sub-structures or patterns within one or more models. It might also prove fruitful if we were to utilize the notion of approximate clones and to incorporate behavioral comparison heuristics, if applicable. For detecting existing patterns or antipatterns, it may be possible to build on techniques intended for model versioning by expressing the patterns or antipatterns as a model that is a potential match. Graph techniques may also prove useful in achieving this goal if we compare our patterns expressed in graph form against normalized versions of our models.

In the future, we plan on either extending existing model comparison approaches or developing our own to analyze models provided to us from industrial partners to find patterns or common structures within them. This may yield model comparison techniques that are specifically tailored for this kind of analysis.

References

[1] B. Al-Batran, B. Schätz, and B. Hummel. Semantic clone detection for model-based development of embedded systems. *Model Driven Engineering Languages and Systems*, pages 258–272, 2011. 19, 27, 28

[2] M. Alanen and I. Porres. Difference and union of models. In *6th International Conference on The Unified Modeling Language: Modeling Languages and Applications.*, pages 2–17. Springer Verlag, 2003. 4, 9, 11

[3] M. Alanen and I. Porres. Version control of software models. *Advances in UML and XML-Based Software Evolution*, pages 47–70, 2005. 11

[4] K. Altmanninger, P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Why Model Versioning Research is Needed!? An Experience Report. In *MoDSE-MCCM Workshop in MoDELS*, pages 1–12, 2009.

[5] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR-towards adaptable model versioning. In *1st Proceedings of the International Workshop on Model Co-Evolution and Consistency Management*, volume 8, pages 4–50, 2008. 4

[6] K. Altmanninger, M. Seidl, and M. Wimmer. A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009. 4, 5, 29, 30

[7] S.C. Barrett, G. Butler, and P. Chalin. Mirador: a synthesis of model matching strategies. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 2–10. ACM, 2010. 21

[8] M. Beine, R. Otterbach, and M. Jungmann. Development of safety-critical software using automatic code generation. *Society of Automotive Engineers (SAE)*, pages 01–0708, 2004. 3

[9] D. Berardi, D. Calvanese, and G. De Giacomo. Reasoning on UML class diagrams. *Artificial Intelligence*, 168(1-2):70–118, 2005. 21, 29

[10] W.J. Brown. *AntiPatterns: refactoring software, architectures, and projects in crisis*, volume 20. Wiley, 1998. 28

[11] C. Brun and A. Pierantonio. Model differences in the Eclipse modelling framework. *The European Journal for the Informatics Professional*, pages 29–34, 2008. 8, 9, 14

[12] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 493–504. ACM, 1996. 22

[13] J. Chen and H. Cui. Translation from adapted UML to promela for corba-based applications. *Model Checking Software*, pages 234–251, 2004. 29

[14] P. Chen, M. Critchlow, A. Garg, C. Van der Westhuizen, and A. van der Hoek. Differencing and merging within an evolving product line architecture. *Proceedings of the Fifth International Workshop on Product-Family Engineering*, pages 269–281, 2004. 23

[15] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A metamodel independent approach to difference representation. *Technology*, 6(9):165–185, 2007. 9

[16] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing model conflicts in distributed development. In *11th International Conference on MoDELS 2008*, pages 311–325. Springer-Verlag New York Inc, 2008. 9

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002. 14

[18] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in XML documents. In *International Conference on Data Engineering*, pages 41–52, 2002. 29

[19] C. d'Amato, S. Staab, and N. Fanizzi. On the influence of description logics ontologies on conceptual similarity. *Knowledge Engineering: Practice and Patterns*, pages 48–63, 2008. 23

[20] F. Deissenboeck, B. Hummel, E. Juergens, M. Pfaehler, and B. Schaetz. Model clone detection in practice. In *Proceedings of the 4th International Workshop on Software Clones*, pages 57–64. ACM, 2010. 6, 19, 27

[21] F. Deissenboeck, B. Hummel, E. Jurgens, B. Schatz, S. Wagner, J.F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ACM/IEEE 30th International Conference on Software Engineering*, pages 603–612. IEEE, 2009. 6, 18

[22] R. Dijkman, M. Dumas, B. Van Dongen, R. Käärik, and J. Mendling. Similarity of business process models: Metrics and evaluation. *Information Systems*, 36(2):498–516, 2011. 24

[23] T. Elrad, O. Aldawud, and A. Bader. Aspect-oriented modeling: Bridging the gap between implementation and design. In *Generative Programming and Component Engineering*, pages 189–201. Springer, 2002. 3, 21

[24] P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. The topcased project: a toolkit in open source for critical aeronautic systems design. *Embedded Real Time Software (ERTS)*, pages 1–8, electronic, 2006. 14

[25] S. Fortsch and B. Westfechtel. Differencing and merging of software diagrams state of the art and challenges. *Second International Conference on Software and Data Technologies*, pages 1–66, 2007. 8, 30

[26] R. France, I. Ray, G. Georg, and S. Ghosh. Aspect-oriented approach to early design modeling. *IEEE Proceedings-Software*, 151(4):173–185, 2004. 3, 21

[27] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007. 3

[28] R.B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg. Model-driven development using uml 2.0: promises and pitfalls. *Computer*, 39(2):59–66, 2006. 3

[29] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA, 1995. 28

[30] M.R. Garey and D.S. Johnson. *Computers and intractability*, volume 174. Freeman San Francisco, CA, 1979. 6

[31] R. Gheyi, T. Massoni, and P. Borba. An abstract equivalence notion for object models. *Electronic Notes in Theoretical Computer Science*, 130:3–21, 2005. 17, 29

[32] M. Girschick. Difference detection and visualization in UML class diagrams. *Technical University of Darmstadt Technical Report TUD-CS-2006-5*, pages 1–15, 2006. 9, 21

[33] J. Gray, J.P. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle. Domain-specific modeling. *Handbook of Dynamic System Modeling*, pages 1–221, electronic, 2007. 3

[34] B. Hailpern and P. Tarr. Model-driven development: the good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006. 3

[35] T. Kehrer, U. Kelter, and G. Taentzer. A rule-based approach to the semantic lifting of model differences in the context of model versioning. In *ASE*, pages 163–172, 2011. 8

[36] U. Kelter, J. Wehren, and J. Niere. A generic difference algorithm for UML models. *Software Engineering*, 64:105–116, 2005. 4, 9, 17, 22, 23

[37] D. Kolovos. Establishing correspondences between models with the epsilon comparison language. In *Model Driven Architecture-Foundations and Applications*, pages 146–157. Springer, 2009. 15, 16

[38] D.S. Kolovos, D. Di Ruscio, A. Pierantonio, and R.F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *ICSE Workshop on Comparison and Versioning of Software Models*, pages 1–6. IEEE, 2009. 9

[39] D.S. Kolovos, R.F. Paige, and F.A.C. Polack. Model comparison: a foundation for model composition and model transformation testing. In *Proceedings of the International Workshop on Global Integrated Model Management*, pages 13–20. ACM, 2006. 1, 7, 29

[40] P. Konemann. Model-independent differences. In *Proceedings of the ICSE Workshop on Comparison and Versioning of Software Models*, pages 37–42. IEEE Computer Society, 2009. 8

[41] P. Konemann. Semantic grouping of model changes. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 50–55. ACM, 2010. 8

[42] R. Koschke. Survey of research on software clones. *Duplication, Redundancy, and Similarity in Software*, pages 1–24,electronic, 2006. 5

[43] J. Krinke. Identifying similar code with program dependence graphs. In *Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2002. 30

[44] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999. 29

[45] K. Letkeman. Comparing and merging UML models in IBM Rational Software Architect. *IBM Rational - Technical Report (Online)*, July 2005. 12, 13

[46] K. Letkeman. Comparing and merging UML models in IBM Rational Software Architect: Part 7, `http://www.ibm.com/developerworks/rational/library/07/0410_letkeman/`. 2007. 12

[47] J. Lilius and I. Paltor. Formalising UML state machines for model checking. *The Unified Modeling Language*, pages 430–444, 1999. 29

[48] Y. Lin, J. Gray, and F. Jouault. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, 2007. 9, 15

[49] Y. Lin, J. Zhang, and J. Gray. Model comparison: A key challenge for transformation testing and version control in model driven software development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, volume 108, pages 6, electronic, 2004. 7

[50] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *13th Asia Pacific Software Engineering Conference*, pages 269–276. IEEE, 2007. 19, 20

[51] S. Maoz, J. Ringert, and B. Rumpe. Cd2alloy: Class diagrams analysis using alloy revisited. *Model Driven Engineering Languages and Systems*, pages 592–607, 2011. 22

[52] S. Maoz, J. Ringert, and B. Rumpe. A manifesto for semantic model differencing. In *Proceedings of the 2010 International Conference on Models in Software Engineering*, MODELS'10, pages 194–203, 2011. 17

[53] A. Mehra, J. Grundy, and J. Hosking. A generic approach to supporting diagram differencing and merging for collaborative design. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213. ACM, 2005. 16

[54] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *18th International Conference on Data Engineering*, pages 117–128, 2002. 30

[55] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002. 9

[56] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and merging of statecharts specifications. In *29th Proceedings of the International Conference on Software Engineering*, pages 54–64. IEEE Computer Society, 2007. 20, 27

[57] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(1):28–63, 2003. 29

[58] T.N. Nguyen. A novel structure-oriented difference approach for software artifacts. In *30th Annual International on Computer Software and Applications Conference*, volume 1, pages 197–204. IEEE, 2006. 16

[59] T. Oda and M. Saeki. Generative technique of version control systems for software diagrams. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 515–524. IEEE, 2005. 16

[60] D. Ohst, M. Welle, and U. Kelter. Differences between versions of UML diagrams. *ACM SIGSOFT Software Engineering Notes*, 28(5):227–236, 2003. 9, 12

[61] H. Oliveira, L. Murta, and C. Werner. Odyssey-vcs: a flexible version control system for UML model elements. In *Proceedings of the 12th international workshop on Software Configuration Management*, pages 1–16. ACM, 2005. 13

[62] R. Parekh and V. Honavar. Grammar inference, automata induction, and language acquisition. In *Handbook of Natural Language Processing*, pages 1–60, 1998. 28

[63] N.H. Pham, H.A. Nguyen, T.T. Nguyen, J.M. Al-Kofahi, and T.N. Nguyen. Complete and accurate clone detection in graph-based models. In *Proceedings of the 31st International Conference on Software Engineering*, pages 276–286. IEEE Computer Society, 2009. 6, 19

[64] Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey, and Benoit Baudry. *Model Composition - A Signature-Based Approach*. 2005. 21

[65] T. Reiter, K. Altmanninger, A. Bergmayr, W. Schwinger, and G. Kotsis. Models in conflict-detection of semantic conflicts in model-based development. In *Proceedings of 3rd International Workshop on Model-Driven Enterprise Information Systems*, pages 29–40, 2007. 14

[66] S. Ren, K. Rui, and G. Butler. Refactoring the scenario specification: A message sequence chart approach. *Object-Oriented Information Systems*, pages 294–298, 2003. 19

[67] J. Rho and C. Wu. An efficient version model of software diagrams. In *Asia Pacific Software Engineering Conference*, pages 236–243, 1998. 21

[68] J.E. Rivera and A. Vallecillo. Representing and operating with model differences. *Objects, Components, Models and Patterns*, pages 141–160, 2008. 9, 14, 15

[69] C.K. Roy, J.R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009. 5, 29

[70] J. Rubin and M. Chechik. Combining related products into product lines. In *15th International Conference on Fundamental Approaches to Software Engineering*, 2012. To Appear. 24

[71] A. Schipper, H. Fuhrmann, and R. von Hanxleden. Visual comparison of graphical models. In *14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 335–340. IEEE, 2009. 10

[72] D.C. Schmidt, M. Fayad, and R.E. Johnson. Software patterns. *Communications of the ACM*, 39(10):37–39, 1996. 28

[73] M. Schmidt and T. Gloetzner. Constructing difference tools for models using the SiDiff framework. In *Companion of the 30th International Conference on Software Engineering*, pages 947–948. ACM, 2008. 23

[74] M. Sebastiani and P. Supiratana. Tracing the differences on an evolving software model, `http://www.idt.mdh.se/kurser/ct3340/archives/ht08/papersRM08/28.pdf`. pages 1–6. 30

[75] B. Selic. The pragmatics of model-driven development. *IEEE Software Journal*, 20(5):19–25, 2003. 3

[76] P. Selonen. A Review of UML Model Comparison Approaches. In *5th Nordic Workshop on Model Driven Engineering*, pages 37–51, 2007. 30

[77] P. Selonen and M. Kettunen. Metamodel-based inference of inter-model correspondence. In *11th Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 71–80. IEEE Computer Society, 2007. 9, 12

[78] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003. 3

[79] M. Soto. Delta-p: Model comparison using semantic web standards. *Softwaretechnik-Trends*, 2:27–31, 2007. 24

[80] M. Soto and J. Munch. Process model difference analysis for supporting process evolution. *Software Process Improvement*, pages 123–134, 2006. 24

[81] M. Stephan. Detection of java ee ejb antipattern instances using framework-specific models. Master's thesis, University of Waterloo, 2009. 28

[82] P. Stevens. A simple game-theoretic approach to checkonly qvt relations. *Theory and Practice of Model Transformations*, pages 165–180, 2009. 17

[83] H. Störrle. Towards clone detection in uml domain models. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, pages 285–293. ACM, 2010. 6, 13

[84] C. Treude, S. Berlik, S. Wenzel, and U. Kelter. Difference computation of large models. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 295–304. ACM, 2007. 23

[85] M. van den Brand, Z. Protić, and T. Verhoeff. Fine-grained metamodel-assisted model comparison. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 11–20. ACM, 2010. 9, 16

[86] M. van den Brand, Z. Protić, and T. Verhoeff. Generic tool for visualization of model differences. In *Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 66–75. ACM, 2010. 10

[87] S. Wenzel. Scalable visualization of model differences. In *Proceedings of the International Workshop on Comparison and Versioning of Software Models*, pages 41–46. ACM, 2008. 10

[88] S. Wenzel, J. Koch, U. Kelter, and A. Kolb. Evolution analysis with animated and 3D-visualizations. In *IEEE International Conference on Software Maintenance*, pages 475–478. IEEE, 2009. 10

[89] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *20th Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65. ACM, 2005. 9, 21