Supporting the Evolution of UML Models in Model Driven Software Development: A Survey

Amal Khalil and Juergen Dingel

{khalil, dingel}@cs.queensu.ca

Technical Report 2013-602

School of Computing, Queen's University
Kingston, Ontario, Canada K7L 3N6

February 2013

Table of Contents

List of Figures

List of Tables

GLOSSARY

Change Impact Analysis – It is the process of identifying the potential impact of a change on the system components which consequently helps in estimating what needs to be modified in order to accomplish a change.

Change Propagation – It is the process of ensuring that a change is propagated to related entities which helps in maintaining the system consistency and integrity. In model-based development, the term "change propagation" is also known as "model synchronization".

Dependency – In software engineering, dependency is also known as "Coupling" and it refers to the degree to which each program module relies on each one of the other modules. In such a case, a change in the independent modules usually forces a ripple effect of changes in their dependent modules [Adapted from Wikipedia].

Inconsistency – It is a state in which two or more overlapping elements of different software artifacts make assertions about the aspects of the system they describe which are not jointly satisfiable [Adapted from [SZ01]].

Model Evolution – In model-based development where models are the core assets of the software system, they worth the effort of maintaining and evolving them. In such cases, model evolution is considered to be a subset of software evolution.

Model Refactoring – It is the process of changing the internal structure of the software system model in such a way that preserves its external behavior. The motive of this process is to improve the structural aspects of the system model such that it becomes more understandable and maintainable.

Model Transformation – Model transformation is the technology that is used in the area of model-driven development to convert models to other software artifacts.

Model-Driven Development (MDD) – Model-driven development is a software development methodology in which models are the primary artifacts.

Software Evolution – Software evolution is an inevitable process where software systems need to continually be adapted to the changing environment or else they become progressively less useful.

System model – A system model is the conceptual model that describes and represents the different aspects of a system, including both the structural and the behavioral aspects. UML comprises a comprehensive set of diagrams that are used to express the system model.

Test suite – In software development, a test suite is a collection of test cases that are created to test and to verify a software program to show that it has some specified set of behaviors. In this paper, we use the term "Test Model" to refer to artifacts that are used to test the software system including both the test suite and the test cases.

Traceability – In software development, traceability is also known as "Requirement Traceability" and it refers to the ability to link (or trace) system requirements backward to stakeholders' rationales and forward to corresponding design artifacts, code, and test cases [Adapted from Wikipedia].

Uncertainty – In software development, uncertainty can be introduced in many ways where there are limited knowledge to help the developers to make clear and precise decisions. Examples of such uncertainties are the lack of knowledge about the problem domain, inconsistent requirements, and multiple stakeholder opinions.

Unified Modeling Language (UML) – UML is the de facto standard modeling language that is extensively used in the area of model-driven development to express the system model.

1. Introduction

Model-Driven Development (MDD) is a model-centric software engineering approach which aims at improving the productivity and the quality of software artifacts by focusing on models as first-class artifacts in place of code. These models can be defined at different levels of abstraction to represent various aspects of the system. Typically, each model conforms syntactically to a metamodel.

Model transformations play an essential role in model-driven development to convert models to other software artifacts (e.g., code) [Sun11]. When the transformation is carried out to convert a source model to a target model and both models conform to the same metamodel, we call it an *endogenous* transformation. This type of model transformation is used to perform tasks such as model refactoring and optimization in general. On the other hand when the transformation is carried out to convert a source model to a target model and both models conform to different metamodels, we call this an *exogenous* transformation. This type of transformation is used to, e.g., map Platform Independent Models (PIMs) to Platform Specific Models (PSMs) which is needed to handle tasks such as code generation, reverse engineering, and migration.

Software evolution is an inevitable and crucial activity in the software development life cycle that deals with changes in software operating environments and/or requirements. The three distinct types of maintenance activities that happen in software evolution are corrective (i.e., fixing defects), adaptive (i.e., adapting to new technologies and/or new environment), and perfective (i.e., improving software quality) [Swa76].

In the context of MDD, models also evolve for many reasons such as, to fix errors, to add new functional requirements, to enhance some quality aspects, or to adapt to a new technological or architectural environment. Consequently, model evolution can be considered to be a specialization of general software evolution and, similarly, requires reliable and efficient techniques and tools to manage and support model evolution.

To support safe evolution of software models, typically a number of tasks need to be performed, before and after executing possible evolutionary changes. Among these tasks are change impact analysis, change propagation, and consistency verification [MS05].

Another important task to consider is managing and taming the uncertainties that arise during the evolution process when models are incomplete or have inconsistent specifications. Dealing with models in the presence of uncertainty and reasoning about models that have uncertainty are still considered to be challenging [SMB09].

The purpose of this paper is twofold. The first one is to introduce the topic of model evolution by identifying the different types of model evolution and the challenges in automating model evolution and by providing a review of the state-of- the-art techniques for automating model

evolution with the focus on one type of model evolution called *model refactoring* (a form of model evolution which requires improving the structural aspects of the model without changing its behavior). The second one is to provide an overview of the state-of-the-art techniques that have been proposed to support common model evolution tasks, including change impact analysis, consistency management, change propagation, and uncertainty management. The reasons for choosing these tasks are first because they are well known practices in traditional software development methodologies, and second because they are interdependent activities (as shown in Figure 1). For instance, change impact analysis is performed to identify the potential impact of a change on the system components before implementing the required change; the result of this process helps in carrying out the change propagation task to ensure that a change is propagated to related entities which helps is avoiding the introduction of inconsistencies as well as bugs. The process of detecting and resolving contradictions that might arise during the realization of changes is managed by the consistency management task, while the process of capturing and handling the different types of uncertainties that might be introduced during the evolution process (e.g., problem-domain uncertainties, different design alternatives, multiple stakeholder opinions and alternative ways to fix model inconsistencies) is managed by the uncertainty management task.

A wide variety of modeling languages exists. Moreover, UML is the de-facto standard for modeling software systems and it is extensively used in the area of model-driven development. For this reason, the scope of our study is restricted to UML models.

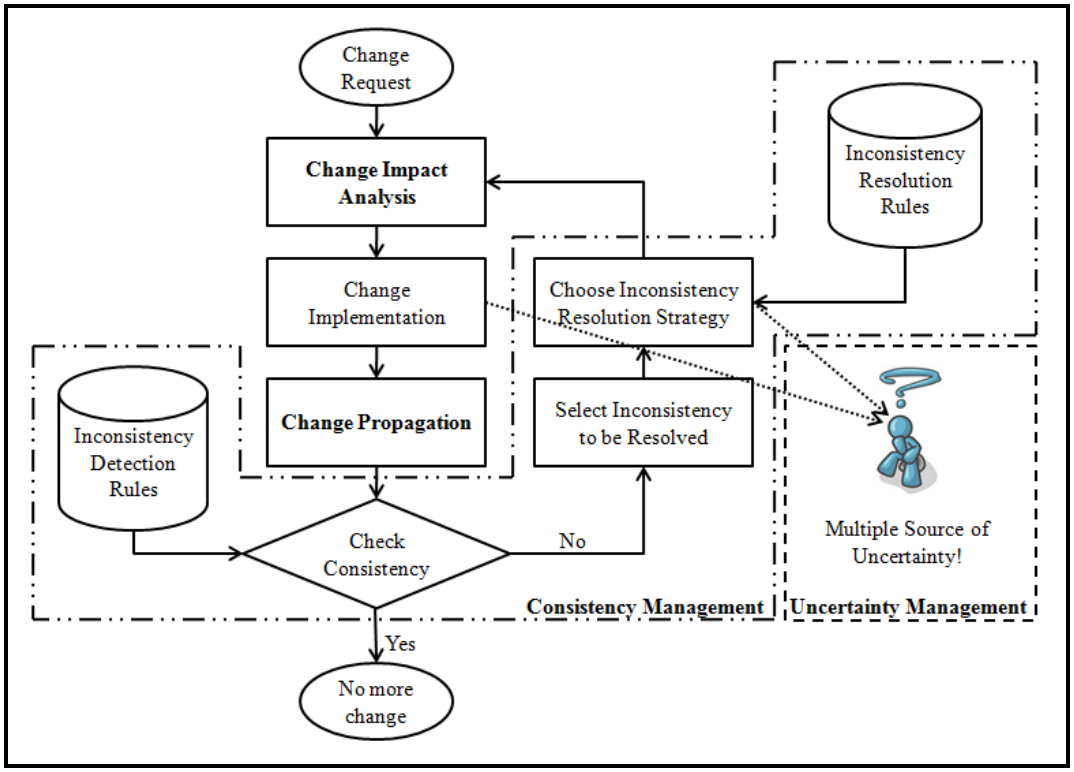
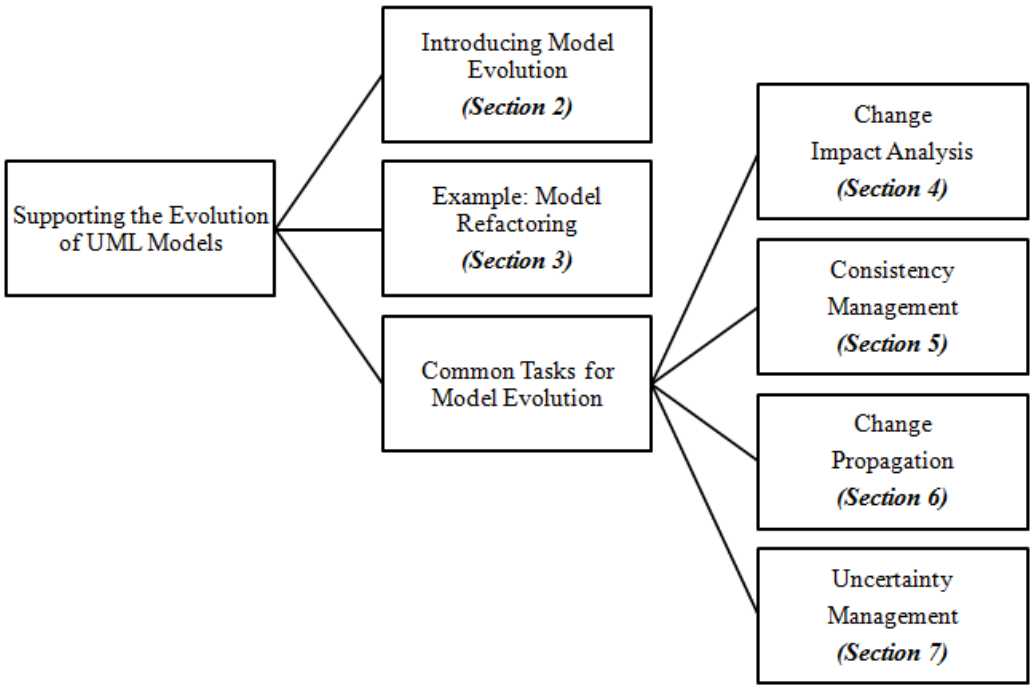


Figure 1: Common Model Evolution Tasks

The work presented in this paper is organized as follows: Section 2 summarizes the different types of model evolution in the context of model-driven development and points out the most common approaches used to implement the evolution processes. Section 3 summarizes model refactoring approaches as an example of one type of model evolution that can be automated. Section 4 outlines model-based change impact analysis techniques. Section 5 presents the work on consistency management. Section 6 outlines change propagation and model synchronization techniques. The work on uncertainty management is presented in Section 7. The conclusion is in Section 8. An overview of the outline of the topics covered in this paper is shown below.



2. INTRODUCING MODEL EVOLUTION

2.1 Types of Model Evolution

Software model evolution has been classified in different ways.

- Van Deursen et al [DVW07] distinguish between *regular*, *metamodel*, *platform* and *abstraction* evolutions. In regular evolution, changes are introduced on the model level. In metamodel evolution, changes are introduced on the modeling language level. These changes might require further consequent changes on the model level to be conformant to the new metamodel. In platform evolution, changes are made to the target platform which consequently might require changes in the code generation procedure and the application framework. Since code generation is performed by means of model transformation, platform evolution mainly impacts the transformation process. In abstraction evolution, decisions are made to change the entire development framework and using a new modeling language which requires the migration of the old system to make use of it in subsequent development.
- Biehl [Bie10] classifies model evolution into two orthogonal dimensions. The first dimension distinguishes between *content-related changes* resulting from adding, deleting or modifying some model elements and *syntactic changes* resulting from changing the abstract syntax of the modeling language represented by the language metamodel. The second dimension differentiates between changes that affect only part of the model (*local evolution*) and the changes that affect the whole model (*systemic or universal evolution*). The cross product of the two categories in each dimension makes four distinct types of model evolution: 1) Local/Syntactic model evolution – this type requires the co-evolution of models to cope with the changes in their metamodels, 2) Systemic/Syntactic model evolution – this type involves the change of the modeling language used that requires the migration of the system models to such new language, 3) Local/Content-related model evolution – this type includes changes made to the model due to the addition, deletion or modification of some model elements, and 4) Systemic/Content-related model evolution – this type represents changes due to the integration of models that capture different parts of the system or that are developed by different teams. The first two types represent the horizontal dimension of model evolution while the latter two represent the vertical dimension of model evolution.
- Levendovszky et al [LRSS11] categorize software model evolution based on three types of changes: 1) changes to the system requirements, 2) changes to the modeling language used to describe the system model, and 3) changes of style by, for example, employing some or different design patterns (this latter type is known as model refactoring).

Table 1 provides a summary of the terms used in each classification with a mapping between the terms that refer to the same type of evolution.

Table 1: A Summary of Software Model Evolution

Examples of Kinds of Changes	[DVW07]	[Bie10]	[LRSS11]
- Fixing errors - Adding new functionalities	Regular	Content-related (Vertical)	Requirement
- Improving model quality (behavior-preserving)			Style (Refactoring)
- Adapting to changes within the modeling language	Metamodel	Syntactic/Local (Horizontal)	Modeling Language
- Adapting to new modeling language	Abstract	Syntactic/Systemic (Horizontal)	
- Adapting to changes in the platform model	Platform		

2.2 Challenges in Model Evolution

Van Der Straeten et al [SMB09] address several challenges in model-driven software engineering and among them were the challenges in 1) assessing the impact of model evolution and metamodel evolution on their context, 2) detecting and resolving the inconsistencies presented by the changes within the model and across all other inter-related models, 3) supporting model synchronization (e.g., model-code synchronization) after the changes, and 4) modeling in presence of uncertainty. Other challenges include the lack of standard frameworks and effective and reliable tool support.

2.3 Visualizing Model Evolution

Anand Rao and Madhavi [RM10] propose a framework of seven criteria listing the most important aspects to be considered in tools that are used to visualize model-driven software evolution. This included, for example, the support for visualizing the context of a model and its elements, visualizing the dependency and the traceability relationship between models and model elements, visualizing model metrics, visualizing information about the transformations that take place within the tool to generate code or to perform some analysis, and visualizing the evolution of models. The authors used their proposed criteria to evaluate tools such as ArgoUML and Visual Paradigm CASE tools as well as the MetricViewEvolution metric visualization tool. MetricViewEvolution [LWC07] is a tool for visualizing and monitoring UML models evolution and their quality metrics which provides the developer with six views: the *Context* view, the *Meta* view, the *Metric* view the *UML-City* view, the *Quality Tree* view and the *Evolution* view. The results from their comparative study showed that although some of these criteria are

supported by the three tools, still none of the tools provides any support for visualizing information about the transformation that takes place on the models. This actually highlights the need for CASE tools that facilitate the integration of model transformation languages and frameworks.

2.4 Approaches for Automating Model Evolution

2.4.1 Evolution Contract Approach

Mens and D'Hondt [MD00] propose an approach to support the evolution during all development phases by extending the UML metamodel by what they called "Evolution Contracts". The idea is based on a previously proposed mechanism of "Reuse Contracts" that is used to handle change propagation between a class and its sub-classes and to manage the evolution of collaborating classes by means of reuse operators. Two types of evolution contracts are defined to represent both primitive and composite evolution contracts. Keeping track of the evolution and the incremental modification of model elements can help in making the evolution a more disciplined activity as well as automating the detection of possible conflicts or incompatibilities that may occur during the evolution of arbitrary inter-related UML models. The proposed approach is realized as a framework in Prolog to detect conflicts in evolving UML models.

2.4.2 Model Transformation Approaches

Automating model evolution is a crucial aspect in both the development and the maintenance of model-driven software. Since model evolution can be considered as a transformation of models from one state to another, we see that model transformation techniques can play an essential role in automating such evolution tasks.

- Levendovszky et al [LRSS11] point out the use of exogenous model transformation in realizing syntactic model evolution (i.e., metamodel evolution) and the use of endogenous model transformation techniques for implementing model refactoring. However the authors addressed the difficulty and the infeasibility of applying similar model transformation approaches to automate local/content-related or regular model evolution that is due to requirement changes.
- Biehl [Bie10] present two approaches for automating horizontal and vertical model evolutions in the context of automotive embedded systems. He used an exogenous model transformation technique for realizing a horizontal model evolution that requires transforming a design-oriented model defined in EAST-ADL2 (an architecture description language used to describe vehicle electronics) into analysis-oriented models defined in MATLAB/Simulink and HiP-HOPS to carry out some model simulation and safety analysis requirements. In addition, he used an endogenous model transformation technique for realizing a vertical model evolution which makes use of model transformation to apply design decisions to the original model.

Accordingly, a number of approaches are proposed to automate model evolution by means of model transformation.

Gray et al [GLZ06] propose a model transformation approach to automate two special categories of model evolution that arise in large-scale software systems: model scalability and applying crosscutting concerns to the model (i.e., design properties that are spread across the model). The proposed approach makes use of the Embedded Constraint Language (ECL) and the concepts of aspect orientation and aspect weaving to develop a transformation engine called C-SAW (Constraint-Specification Aspect Weaver), a plug-in for the Generic Modeling Environment (GME) Framework.

Brosch et al [BLSWWKRS09] present an approach and developed a tool called Operation Recorder for applying model transformation by-example. The idea is to record user-defined composite operations on models (e.g., model refactoring actions) and then use them to generate a transformation that can be executed to perform the same recorded operations on any given set of models as long as they match the same pre-conditions as the original ones. A similar approach is proposed by Sun et al [SWG09, SGW11] to automate software model evolution using model transformation by-demonstration technique. A tool called MT-Scribe is developed to realize and to demonstrate the proposed idea which can be used to automate activities such as model refactoring, model scalability, weaving aspects, and model layout configurations. One advantage of this type of approaches is that it can reduce the burden on the developers for learning the details of the metamodel of the modeling language and of model transformation languages.

2.4.3 Constraint with Action Language Approach

Ajila and Alam [AA09] propose formal language constructs for software model evolution. The authors extended the Object Constraint Language (OCL) with actions to create a new language named CAL (Constraint with Action Language). The language is supported with a new data type for representing a collection of data (e.g., the model elements) in a directed acyclic graph (DAG) form. Such representation is useful for the automatic dependency analysis of the model. A prototype tool, VCAL (Visual CAL), is implemented to demonstrate and evaluate the feasibility of the proposed method. The tool has a parser to load and transform a UML model into CAL specification (i.e., as a DAG data structure). A formal specification language called TLA (Temporal Logic of Actions) is used to specify the actions and the operations in CAL. A model checker is used to verify and reason about these TLA specifications.

2.4.4 The MoVE Approach

The goal of the MoVE (Model Versioning and Evolution) project is to provide a solution to some of the challenges of dealing with a variety of models that are used in describing, developing and operating IT-systems. They propose the idea of living models and they define ten principles to be employed in developing the infrastructure for such an idea. Among these principles are the support for the persistence of models and their evolution, consistency, change propagation, and bi-directionality of the information between models and code. The proposed architecture is based

on model versioning and change-driven model evolution (i.e., state changes and change propagation) [MoVE].

2.5 General Observations

Based on the reviewed work presented above, we could see that automating model evolution is not feasible for all types of model evolution. Modeling language adaptation changes, applying some model quality, scaling up parts of the model, and crosscutting changes that are having their design is scattered over many places in the system model are examples of evolution types that can be automated. To get a better understanding of the work achieved in this area, we decided to survey the literature for approaches that have been proposed to automate one of these types of model evolution which is model refactoring. We argue that the findings from this review can help us in better understand the main issues in this area.

3. EXAMPLE: MODEL REFACTORING

Refactoring is a well-known practice for improving the quality of software systems. It is the process of changing the internal structure of the software system in such a way that preserves its external behavior. The motive of this process is to improve the design of the software system such that it becomes more understandable, modular, reusable, extensible and maintainable.

Refactoring is a crucial activity for current evolutionary software development processes where systems are built in an iterative and incremental fashion and so are exposed to frequent changes which may cause them to deviate from the original/intended design.

Traditional refactoring concepts are applied on the code level, however in recent years the idea of applying the same concepts on the design level has been widely acknowledged as a good practice especially within the model-driven development community. As a result, model refactoring techniques have started to emerge.

The advantage of having model refactoring over traditional code refactoring is that the former helps in discovering the errors that are made early in the design process and in improving the modularity of the design models and so reducing the complexity and cost of possible refactoring process in the successive phases.

In the following sections we introduce the topic by discussing the main challenges in carrying out model refactoring activities and accordingly finding out what opportunities exist in the literature to meet some of these challenges in order to come up with our conclusion.

3.1 Challenges in Model Refactoring – A Road Map

Mens et al [MTM07] list some of the challenges in model refactoring including lack of precise and comprehensive definition of model quality in terms of model smells and model metrics (in analogy with bad code smells and software metrics), lack of formal semantics and precise definition of system behavior is a common issue that complicates the behavior preservation validation of the refactored model to ensure that it offers the same behavior as it did before refactoring, lack of automatic synchronization between a refactored model and all its other inter-related models, lack of techniques and tools to measure the impact of the refactoring process on the test cases created for the model before refactoring, the need for generic model refactoring procedure for domain-specific models, the need for a precise means to analyze the relationships between different refactoring steps, and finally the need for reliable and generic tool support that can be integrated into open-source CASE tools and current Model-Driven development solutions.

We chose to look at the work achieved in the following directions: 1) specifying and defining common model quality metrics (both, the bad and the good ones) and the model refactoring actions based on these quality metrics, 2) realizing model refactoring approaches and techniques, 3) synchronizing inter-related models after applying model refactorings to a model, 4) proving

the behavior preservation property of the models after refactorings, and lastly 5) developing tool support for model refactoring.

3.2 Defining Model Quality & Model Refactorings

Defining model quality is the foundation for defining metrics, detecting bad design smells, and consequently applying possible model refactoring actions. A number of model quality measures are defined in the literature especially in the context of the object-oriented development methodology and the UML modeling language practices. Examples of these metrics are presented by Kim and Boldyreff [KB02] and by Enckevort [Enc09].

Additionally, a number of these metrics are defined in current OO design quality measurement tools that are used in analyzing the structure of UML models. One example tool is the Software Design Metrics tool for the UML, SDMetrics [Wüs11].

To demonstrate the feasibility of applying UML refactoring in the context of agile processes, Astels [Ast02] provides examples on bad design smells in class and sequence diagrams and described a number of refactoring actions for such model smells. The author was motivated with the fact that bad smells detection can be easier in the model level more than in the code level. Thereby, he identified some features a class might have that can be a sign for a bad design smell such as classes that delegates the majority of work to another classes (i.e., *Middle Man* class smell). Also, he proposes some applicable refactoring actions such as "*Replace Delegation with Inheritance"* for example.

Sunyé et al [SPTJ01] present an initial refactoring set for class diagrams and statecharts, which can be defined as OCL constraints at the metamodel level, to improve the quality of these artifacts. The authors claimed that satisfying both the pre-condition and the post-condition expressed in the OCL constraints would ensure that the applied refactorings are behavior preserving. The proposed refactorings set included the *addition*, *removal*, *move*, *generalization* and *specialization* of modeling elements for class diagrams and operations such as *folding incoming/outgoing actions, unfolding entry/exit action, grouping states, folding outgoing transitions, unfolding outgoing transition, moving state into composite state* and *moving state out of composite state* for statecharts.

UML class diagrams are usually annotated with a number of OCL constraints describing the invariants that cannot be represented visually in the model. These OCL constraints sometimes are not easy to understand or maintain especially in the case where their underlying classes have evolved. This addresses the need to consider OCL constraints in the model refactoring process. Thereby, Correa and Werner [CW04] presents examples of possible OCL smells and suggested a number of applicable refactorings for them. The authors defined their proposed list of OCL refactorings in a prototype tool called Odyssey-PSW.

Dobrzański and Kuźniarz [DK06] provide a set of refactorings for executable UML models that are focused on refactoring the bad smells in class diagrams. What differs this work from Sunyé et

al [SPTJ01]'s work is that the former took into account in the refactoring process the required updates of the behavior aspects of the models as well (e.g., updating the sequence diagram that represents a required behavior where an instance of some refactored class is part of it). The authors formalized their refactorings in terms of pre-conditions and post-conditions in OCL. The implementation of the presented work is carried out within the Telelogic TAU CASE tool.

El-Sharqwi et al [EME10] present an approach to apply model refactoring based on design patterns that are defined in XML notation. A design pattern consists of three parts: a *Problem Specification* describing the context where the design pattern can be applied to improve some quality aspect, a *Target Specification* describing the design pattern itself, and a *Transformation Specification* describing a sequence of primitive transformations required to apply the design pattern. Given these design patterns and the XML representation of a model that needs to be examined, a detection algorithm that is formalized as a constraint satisfaction problem (CSP) will detect the problem specification instances in the model that violate such design patterns which will then trigger the corresponding transformations guided by the user (i.e., in a semi-automatic fashion). The approach is illustrated on the Abstract Factory Pattern. No implementation is performed yet.

3.3 Model Refactoring Using Model Transformation Approaches

Model refactoring can be considered as an endogenous model transformation performing some sort of model optimization. Since UML models can be seen as graphs, graph transformation techniques are heavily employed for developing UML refactoring.

Zhang et al [ZLG05] develop a prototype model refactoring browser tool for refactoring models on top of the C-SAW (Constraint-Specification Aspect Weaver) model transformation engine and as a plug-in for the GME (Generic Modeling Environment) framework. The tool is equipped with a pre-defined set of generic model refactorings for the GME metamodel which are similar to regular class diagrams refactorings. Also, a number of domain-specific refactorings are defined for domain-specific models such as the AQML (Adaptive Quality Modeling Language) models and Petri Nets. Such refactorings are described as transformation rules that are expressed in terms of ECL (Embedded Constraint Language) specifications. No validation for the behavior preservation property after refactorings is carried out; however it is mentioned in the future work section as one of the important aspects to be considered.

Folli and Mens [Fol07, FM08] discuss the idea of using graph transformation to represent UML model refactoring. As a proof-of-concept of their idea, the authors formalized a set of eight refactorings (four for class diagrams which are "*Pull Up Operation*", "*Push Down Operation*", "*Extract Class*", and "*Generate Subclass*" and four for state machines which are "*Introduce Initial Pseudostate*", "*Introduce Region*", "*Remove Region*", and "*Flatten State Transitions*") using the AGG (Attributed Graph Grammar System) graph transformation tool and developed a tool support to evaluate the feasibility of such an approach. Based on the practical experience gained, the authors managed to assess the strengths and the weaknesses of current graph model

transformation notations and their tools such as the AGG and the MOFLON graph transformation tools in defining model refactorings. Although graph transformations can provide a more concise visual representation of complex transformations, their expressive power is not sufficient to define a complete set of model refactorings.

Biermann et al [BEKKTW07] use graph transformation concepts to specify model refactorings for EMF models. The AGG graph transformation tool is used to implement the proposed transformation rules. Validating the consistency of a refactoring step is carried out through checking the syntactic correctness of the refactored model after performing the refactoring. A repair strategy is presented to restore the consistency for transformation rules that lead to an inconsistent refactored model.

Porres [Por03, Por05] present an approach for model refactorings based on transformation rules that specify the required parameters for each rule, the guards (or the pre-conditions) which determine when each rule can be applied, and the body that implements the effect of each rule. Guards are described using OCL-like side effect free expressions and rules bodies are described using an imperative action language called SMW (the author's own scripting language to manipulate models based on the Python programming language). A sequential algorithm is defined for the execution of a transformation. The author depended on three criteria to ensure the correctness of his refactoring transformation by checking 1) if the transformation terminates, 2) if the transformed model is syntactically correct, and 3) if the transformation preserves some observable properties of the model. Behavior preservation is another important criterion that is not considered in this work. The presented ideas are implemented in an experimental transformation tool within the context of the SMW toolkit for a number of refactorings for class and state machine models. One of the difficulties the author faced when developing his approach was in deciding what strategy to take to update the graphical representation of the models (i.e., the diagrams) after refactorings especially with refactorings that require adding new elements to the model.

3.4 Generic Model Refactoring Approaches

Moha et al [MMBJ09] propose an approach for generic model refactorings that can be applied to different modeling languages as long as their metamodels are sharing the same aspects (e.g., Java, MOF, and UML). The proposed approach is implemented within the context of the Kermeta transformation language. The four steps composing the approach are: 1) specifying a generic metamodel that has the most common model elements involved in class refactorings (e.g., classes, attributes, methods, and parameters), 2) specifying a set of generic refactorings based on the generic metamodel defined earlier, 3) adapting the target metamodels to the generic metamodels using the Kermeta features of weaving aspects and model typing which enable the addition of some derived properties, and finally 4) applying the specified refactorings the target metamodels with their Kermeta code adaptations. The demonstrated work included only to three

refactorings: *Encapsulate Field*, *Move Method* and *Pull Up Method*. No verification for the behavior preservation after refactoring is carried out.

Reimann et al [RSA10, RSA12] propose a generic framework for model refactoring which consists of role models and generic transformation specifications. Each role model consists of participants which collaborate to carry out a generic refactoring. Those participants simply represent the structural aspects of the refactorings which are language-independent and so they can be part of a generic refactoring formalism. Using such a generic framework, only a mapping specification is needed to bind role models to specific modeling languages by defining which elements of a specific modeling language play which role in the context of a refactoring. Having such mapping specification, a generic transformation specification can be executed (regardless the modeling languages used) to restructure models. The proposed framework provides extension points to attach domain-specific components describing additional formal constraint to check for the behavior preservation and the correctness of a refactoring. An EMF-based implementation of the proposed approach is carried out as a proof of concept and to assess the feasibility of the approach.

3.5 Synchronizing Model Refactoring

Only a limited number of work have been targeting the subject of synchronizing refactored models although there are many existing approaches tackling the problem of model synchronization on its two levels: the horizontal level (intra-phase) and the vertical level (inter-phase). We think that integrating current model refactoring techniques with such model synchronization approaches can provide possibly a practical solution for this problem.

3.5.1 Synchronizing Design Model with Source Code

Gorp et al [GSMD03] consider the loss of synchronization between design models and their corresponding source code when either one is refactored. They propose *GrammyUML*, a language-independent extension of the UML metamodel that includes some specific details for method definitions and their surrounding scope to guarantee the consistency between the design model and the underlying source code after refactoring. The authors also introduced the concept of "*Refactoring Contracts*" to define possible refactoring steps, using OCL, in terms of a *pre-condition* of the restrictions that need to be satisfied in the model before applying the refactoring step, a *post-condition* of the properties to be satisfied in the model by the refactoring, and the "*code smells*" or the problem that can be improved by the refactoring. Employing the concepts introduced in *GrammyUML* and *Refactoring Contracts* in future MDA tools help in describing possible bad smells and defining their refactoring actions, detecting the occurrence of such bad smells and consequently automating the process of executing the appropriate refactoring steps. As a proof of concept of the practical feasibility of the proposed ideas, a prototype implementation is carried out within the Fujaba tool to apply the two refactorings: *Pull Up Method* and *Extract Method*.

3.5.2 Synchronizing Models with their Diagrams

Einarsson and Neukirchen [EN12] present an approach and implemented a tool for synchronous refactoring of UML diagrams and their underlying model using model-to-model transformations. The key elements of the approach are the MOF-based UML metamodel, the UML diagram definition and the QVT transformations. The Operational QVT is the model transformation language used to define the refactoring rules. A prototype tool is implemented as a plug-in for the Eclipse-based Papyrus UML editor.

3.6 Semantic and Behavior Preservation

At the code level, a refactoring is said to be behavior preserving if each method 1) accesses the same variables after the refactoring as it did before the refactoring, 2) updates the same variables after the refactoring as it did before the refactoring, and 3) performs the same method calls after the refactoring as it did before the refactoring. These types of behavior preservation are called "access", "update", and "call" preservation respectively [MEDJ05]. Only "method call" preservation can be examined for refactorings that are performed at the model level. A study that has been carried out to prove this type of behavior preservation is the work of Van Der Straeten et al [SJM07]. The authors in this work formalized the behavioral specification of a system represented by UML state machine and sequence diagrams in Description Logic and defined two properties, observation call preservation and invocation call preservation, to check the behavior preservation between a class and its refactored version along with their corresponding state machine and sequence diagrams. Tool support is implemented, as a plug-in for the Poseidon CASE tool, and is tested on small examples.

Van Kempen et al [KCKB05] present a case study to prove that behavior is preserved after refactoring the UML class diagram of a given system. To overcome the lack of a formal semantics of UML models, the authors mapped the system's behavior specified by classes' statecharts before and after refactoring into CSP processes (a formal language for describing patterns of interaction in concurrent systems known as Communicating Sequential Processes [Wikipedia's definition]) and then used the FDR2 (Failures/Divergence Refinement) model checker to prove that each one of the two CSP processes is a refinement of the other.

Baar and Marković [MB05, BM07] propose an approach to prove the semantic preservation of the UML/OCL refactoring rules. A UML/OCL refactoring is said to be syntax preserving if for every syntactically correct source model there is a syntactically correct generated target model, while it is said to be semantic preserving if the evaluation of an OCL constraint on the model before refactoring gives the same results as the evaluation of the re-factored OCL constraint on the re-factored model. In this approach, refactoring rules are specified as model transformation steps applied to a source model (i.e., the model before refactoring) to generate a target model (i.e., the model after refactoring). QVT is used as model transformation language. The proposed approach is implemented in a prototype tool called ROCLET and is illustrated on the *MoveAttribute* refactoring rule.

Graph transformations play a significant role in developing model refactoring approaches. Graph transformation rules provide a graphical presentation of refactoring definition as well as an underlying algebraic context that can be used to ensure behavior preservation in model refactoring.

Hosseini and Azgomi [HA08] introduce a behavior preservation centered approach for UML model refactoring by creating a control-flow diagram (CFD) for each refactoring operation to direct the refactoring process. Refactoring operations are described in the AGG graph transformation tool. The proof of the behavior preservation of a refactoring operation is achieved through satisfying all the execution order and the conditions requirements of its corresponding CFD (by means of a Java-like imperative formal specification language called ROOL and the graph transformation laws). The proposed idea is illustrated on the "*Push Down Operation*" refactoring. No details about whether the approach has been implemented or not were included.

Rangel et al [RLKEB08] develop a technique using Double-Pushout (DPO) graph rewriting rules which are themselves behavior preserving to assure that applying such rules on a given source model will generate a refactored model that has the same behavior. The authors presented a set of definitions for behavior-preserving DPO rules and another set for non-behavior-preserving DPO rules. This latter type of rules are used to represent the intermediate refactoring actions that do not need to be behavior preserving. The presented approach is theoretically illustrated in the context of Deterministic Finite Automata (DFA) models for the "*Deleting Unreachable States*" refactoring action, yet no practical implementation is performed.

3.7 Tool Support

A limited number of commercial UML CASE tools support model refactorings and if so only simple class diagrams refactorings such as moving and renaming model elements are presented. Examples of such tools are IBM Rational Software Architect (RSA) and MagicDraw. A more comprehensive list of UML refactorings is found as a plug-in, called *Refactoring Browser*, for the Poseidon commercial UML CASE tool that was developed by Boger et al [BSF03]. In this work, the authors propose a number of refactorings for class, state machine and activity diagrams to enable model refactoring. Although the work seems to be solid and promising, no details were provided about its underlying technique.

Apart from this, other UML refactorings have been developed within the research community as academic prototypes for some of the available open-source CASE tools. Examples of these UML refactoring tools are the work developed by Dobrzański and Kuźniarz [DK06] and by Gorp et al [GSMD03].

Moreover, a number of model refactoring tools have been built based on the EMF representation of UML models, for example the EMF Refactor, an open-source component within the Eclipse Modeling Framework.

A comparative study is carried out by Arendt et al [AMST09] to evaluate and compare between EMF Refactor and another two model refactoring options namely the Language Toolkit (LTK) and the Epsilon Wizard Language (EWL). The comparison is run based on some criteria such as the complexity, the correctness, the testability and the modularity of the refactoring specification and the interaction, the features and the malfunction of the refactoring application. Although the results revealed the strengths and the weaknesses of each method, the authors recommended to create a new method that combines LTK with EMF Refactor as a possible way to benefit from the features of both methods.

A most recent work by Arendt and Taentzer [AT12] present a framework of the integration of the two EMF-based tools: EMF Smell and EMF Refactor to support modelers in automatically detecting and reporting about model smells and suggesting appropriate model refactoring.

3.8 General Observations

Some of the approaches provide only abstract ideas that have not yet been implemented. On the other hand, the majority of implemented approaches is too specific for certain domains and also lacks sufficient validation. Only model refactorings for the structural aspects in the models are explored.

The issue of behavior preservation in model refactoring is still an open topic. All proposed approaches tackling this problem provide only partial solutions.

It is noted also that the number of tools available that enable UML model refactoring is very limited compared to similar tools that effectively support source code refactoring.

4. COMMON TASKS OF MODEL EVOLUTION – CHANGE IMPACT ANALYSIS OF UML MODELS

Bohner and Arnold [BA96] define change impact analysis as “the process of identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change”. Change impact analysis is a very important activity for the maintenance of software. For instance, it is used by regression testing techniques to reduce the amount of work required to test and verify the system after the modification.

Kilpinen [Kil08] observes that the most common techniques used to implement change impact analysis are based on either traceability or dependency relationships between the software artifacts. While traceability-based impact analysis techniques work on analyzing the relationships between requirements and other development artifacts (such as design, implementation and test cases) to determine the scope of the anticipated change(s), dependency-based impact analysis techniques work on a more detailed level by analyzing the relationships between the artifacts of the same development phase.

In the context of model-driven development, models are the primary artifacts of the software system. In this case, when the software system needs to evolve or to adapt to new requirements, changing the models representing the system is the natural starting point. Applying change impact analysis on the model level can provide early assessment of the cost and the complexity of changes before their actual implementation. Another advantage of such techniques is that they facilitate the analysis of changes in an earlier stage of development as well as on a more abstract level.

One of the objectives of this study is to investigate to what extent the idea of applying model-based change impact analysis is tackled in the literature and how much it is employed in the model-driven community to support the evolution processes.

We only consider change impact analysis techniques for UML models. Two categories of research work are identified in this domain. The first category focuses on approaches that perform this type of analysis to measure the impact of changes on other software artifacts (e.g., design artifacts or its underlying source code). The second category focuses on approaches that use the model-based change impact analysis to identify the impact of changes on the system test model as part of the regression test selection process. The purpose of this process is to identify the subset of test cases from the initial test suite of the system which can effectively test the unchanged parts of the system.

Regression testing is the process of testing a new version of the system with a previously used test suite. Instead of re-running the entire test suite, a more effective way is to select only the subset that has the potential to examine the changed parts of the system. Such selection process is called regression test selection.

Leung and White [LW89] classify test cases into three categories: *reusable* test cases (i.e., test cases which cover the unchanged parts of the system and so should be excluded from the regression testing however they should be kept in the test suite), *obsolete* test cases (i.e., test cases which are no longer valid and should be deleted from the test suite and not to be included in the regression testing) and, *re-testable* test cases (i.e., test cases which cover the changed parts of the system).

In the following subsections we present an overview of some existing approaches from each category. Table 2 provides a brief summary of each work based on the following criteria: the task achieved by the work which is identified by which category the work belongs to, the technique(s) used, the types of UML diagrams involved and whether the work has a tool support or not.

4.1 Intra-model Change Impact Analysis on Design Model

In this section we introduce approaches that implement model-based change impact analysis for identifying the impact of changes within the design model.

4.1.1 Using Explicit Impact Analysis Rules

Briand et al [BLOS06, BLO03] propose an approach to analyze the impact of the changes on the design model before applying the changes to the implementation model. The authors provided a classification of change types for three UML diagrams: class, sequence diagram and state machine diagrams. For each change type, an impact analysis rule is specified, describing how to extract the list of elements that are impacted by that particular change type. The definitions of change types and impact analysis rules are expressed in the Object Constraint Language (OCL). The propagation of changes to indirectly related entities is controlled by a distance measure, which is used to either stop the change propagation or to prioritize impact paths according to their depth. A prototype tool (iACMTool) was developed to automate and evaluate the feasibility of such an approach.

4.1.2 Using Traceability Links & Data Mining Technique

Dantas et al [DMW05, DMW07] propose a methodology based on one of the data mining techniques called *Association Rules* that works on a versioned UML repository to detect change traces between different versions of UML artifacts. The detection process comprises two phases: the configuration phase and the querying phase. In the configuration phase, the threshold values for the data mining metrics, *support* and *confidence*, are selected as well as the information to be recorded about the change is defined. This information identifies *who*, *when*, *where*, *why*, *what*, and *how* a change is made which will support the developer in understanding the rationale for the change. In the querying phase, the change traces for a given queried artifact are retrieved. Analyzing these change traces will help in estimating the likelihood that some change triggers additional changes. The approach was implemented and integrated in one of the software configuration management systems called Odyssey-SCM (Murta et al [MODLW07]). The infrastructure of this system is composed of a flexible version control system for fine-grained

UML model elements, named Odyssey-VCS and a customizable change control system tightly integrated with the version control system to manage the evolution of these models.

4.1.3 Using Dependence Analysis Technique

Fourneret and Bouquet [FB10] present an approach to perform model-based impact analysis for UML Statechart diagrams using dependence algorithms. The idea presented here is adopted from the work of Chen et al [CPU07] on model-based regression test strategies using dependency analysis of Extended Finite State Machine (EFSM) models. The rationale of this type of analysis is to consider dependences between transitions instead of states. In such context, two transitions are said to be *data dependent* if one transition defines a value of a variable that can be potentially used by the other transition and they are said to be *control dependent* if one transition may affect the traversal of the other transition. Based on this concept, the authors identified the data and control dependences for UML statecharts and formulated the corresponding algorithms to be used in computing the dependence graphs of the statecharts elements. Accordingly, they identified and classified the possible changes to UML statecharts such as adding new transitions, deleting existing transitions, and modifying existing transitions (by changing the OCL constraints of the guard or the action of the transition). It is worth to note that adding or deleting a transition can impact both the data and control dependence graphs while modifying a transition can impact only the data dependence graph. By having the two versions of statecharts (the original and the modified one) and their computed dependence graphs, the authors illustrated how their proposed dependence algorithms are used to extract the impacted elements.

4.2 Inter-model Change Impact Analysis between Design Model and Test Model

In this section we introduce approaches that implement model-based change impact analysis for identifying the impact on the test model. We distinguish between two categories of approaches: the first category works on a code-driven test model while the second category works on model-driven test model. We use test model to refer to the system test cases or the system test suite.

4.2.1 Code-driven Test Model

4.2.1.1 Using Traceability Links

Briand et al [BLH09, BLS02] present an approach to perform regression test selection of code-level test cases from the impact analysis of changes on the design level using UML class, sequence and use case diagrams. To apply this approach, the authors assumed that each use-case is realized by a sequence diagram and that it is possible to map each test case to its corresponding sequence diagram scenario in such a way that guarantees the continuation of traceability information between the design and its test cases. Based on these assumptions, they formalized a set of changes in UML models using set theory and first order logic. Then a comparison algorithm is applied to the two versions of the system models (class, sequence, and use-case diagrams) to identify the use-cases which have their sequence diagrams changed and then classify the test cases corresponding to those use-cases into reusable, obsolete or re-testable. The proposed approach is realized in a prototype tool called RTSTool and empirically evaluated

using both academic and industrial case studies. The results showed that the proposed model-based regression testing technique is not as precise as such techniques which are based on the code level, yet it can be valuable especially for very large systems.

Naslavsky et al [NZR10] propose an approach to model-based regression testing based on UML class and sequence diagrams. The approach is built upon 1) a pre-developed model-based test case generator, 2) a new traceability model that records the links between the model elements represented by class and sequence diagrams, 3) a new transformation algorithm to map UML sequence diagrams to their corresponding control flow diagrams, and 4) a model differencing algorithm to compare between two UML models and to produce the set of changes made on one of them compared with the other. Given the differencing model, the two control flow graphs of the two sequence diagrams and the traceability model of the original UML model, a pair-wise graph traversal of the two control flow graphs is used to classify the test cases generated for the original UML model into reusable, obsolete or re-testable. A prototype of the proposed approach has been implemented in a tool called mbSRT2. Two case studies have been conducted for evaluation purposes.

Mansour et al [MTN11] present a design-level regression test selection technique using UML class, sequence and interaction overview diagrams. Interaction Overview diagram is a new design-level artifact which is introduced in UML 2.0 and is used to summarize the control flow of the entire system. Given the initial test suite of the system, the following are the main steps of the proposed approach: 1) recording the relationship between the system model elements and each test case in the given test suite, 2) introducing a change into the system model, 3) determining the test cases affected by the change is carried out using two new algorithms: the first one is based on changes in class diagrams, while the second is based on changes in the interaction overview diagrams. This technique is the first to employ Interaction Overview Diagram (IOD) in regression testing.

4.2.1.2 Using Dependence Analysis Technique

Fourneret et al [FBDD11] extended the work of Fourneret and Bouquet [FB10] to propose a selective test case generation approach for the validation of evolving critical systems that are described in UML4ST (a sub-set of UML which uses three kinds of diagrams: class, object and statechart). Given the original test suite of the software system, the original and the evolving UML statechart models, and the dependence algorithms [CPU07], the approach will 1) analyze the changes in dependence graphs extracted from the UML statechart model with respect to the evolutions that were made, 2) identify the tests that have been affected by the evolution and those that were not, 3) classify each test in the original test suite into three categories: outdated, un-impacted, or re-testable. The test classification step helps in identifying parts of the evolved model that have not been covered by tests (i.e., the parts which correspond to new elements) and so need to have new tests to be generated and added to the new test suite. The approach also developed the notion of test versioning to keep track the test life cycle across different

evolutions. Tests are classified into four test suites: an *evolution* test suite (for tests that exercise new requirements, new operations or new behaviors), a *regression* test suite (for tests which exercise the unmodified parts of the system), a *stagnation* test suite (for tests which are invalid with respect to the current version of the system), and a *deletion* test suite (for tests which are obsolete for the previous version of the system). These four test suites constitute one version in the test repository. The presented approach has been realized in a standalone Java application called *EvoTest* to run the experimental evaluation.

4.2.2 Model-driven Test Model

4.2.2.1 Using Traceability Links

Iqbal et al [IMN07] present a model-based regression testing approach using UML class diagrams and state machines. The approach is based on 1) two comparisons: one to compare between class diagrams and the other to compare between state machines and 2) formal definitions of a set of changes that can be applied to both class diagrams and state machines similar to those of Briand et al [BLO03]. Given the baseline test suite of the original system (consisting of test paths that are extracted using one of the state machine based testing approaches) and the two versions of the system model identified by the baseline version of class and state machine diagrams and the delta version of class and state machine diagrams, a class-driven changes and a state-driven changes can be identified and extracted. Based on this list of changes and their types, all corresponding test cases in the baseline test suite are identified and classified into reusable, obsolete or re-testable. Prototype tool support, *START*, and an evaluation of the approach proposed in this work are developed and presented by Farooq et al [FIMR10].

Pilskans et al [PUA06] propose a model-based regression testing approach using UML class and sequence diagrams. The authors used the approach of Briand et al [BLO03] to identify changes and the approach of Leung and White [LW89] to classify test cases. In this approach, objects and messages in a sequence diagram are mapped into vertices and edges in a directed acyclic graph called *Object Method Directed Acyclic Graph* (OMDAG) and classes in a class diagram are mapped into *Class Tuples* (CT) of class name, attributes and methods. The integration of OMDAGs with CTs results in a modified OMDAG that represents the entire system. The generation of test cases is carried out using the symbolic execution technique of the OMDAG paths. When a path in the OMDAG changes due to a modification in the system models, all test cases associated with this path are identified and classified, based on the type of the executed change, into reusable, obsolete or re-testable. The advantage of this technique over the code-based regression testing techniques is that it requires fewer numbers of paths in the OMDAG to check.

4.3 General Observations

Although a number of approaches are covered, we argue that research work in this area is not active. One possible justification for such state can be the fact that system evolutions are carried

out on the system source code and not on the system models. Thereby most of the research work is targeting the change impact analysis but on the code level.

In his recent survey [Leh11], Lehnert classify change impact analysis approaches into three categories based on the types of artifacts the analysis performed on, whether they are source code, formal models (architecture or requirements), or miscellaneous artifacts (documents or configurations). The author noticed that the majority of approaches used to date still focus on applying change impact analysis on the code level which actually supports our conclusion.

Table 2: A Summary of the Surveyed Work on Model-based Change Impact Analysis

#			Reference	Task Achieved			Utilized Technique(s)	Type(s) of UML Diagram	Tool Support
				Change Impact Analysis (CIA) Level	Model(s) Type				
1	Category I		[BLOS06, BLO03]	Intra-model CIA	Design Model		Explicit Impact Analysis Rules	Class Diagram Sequence Diagram Statechart Diagram	iACMTool
2			[DMW05, DMW07]	Intra-model CIA	Design Model		Traceability Links & Association Rules (one of the data mining method)	Use case Diagram Class Diagram Component Diagram	Odyssey-SCM
3			[FB10]	Intra-model CIA	Design Model		Dependence Analysis	Class Diagram Statechart Diagram	
4	Category II	Code-Driven Test Model	[BLH09, BLS02]	Inter-model CIA	Design Model	Test Model	Traceability Links	Use-case Diagram Class Diagram Sequence Diagram	RTSTool
5			[NZR10]	Inter-model CIA	Design Model	Test Model	Traceability Links	Class Diagram Sequence Diagram	mbSRT^2
6			[MTN11]	Inter-model CIA	Design Model	Test Model	Traceability Links	Class Diagram Sequence Diagram Interaction Overview Diagram	
7			[FBDD11]	Inter-model CIA	Design Model	Test Model	Dependence Analysis	Class Diagram Statechart Diagram	EVOTest
8		Model-Driven Test Model	[IMN07, FIMR10]	Inter-model CIA	Design Model	Test Model	Traceability Links	Class Diagram State Machines	SRART
9			[PUA06]	Inter-model CIA	Design Model	Test Model	Traceability Links	Class Diagram Sequence Diagram	

5. COMMON TASKS OF MODEL EVOLUTION – CONSISTENCY MANAGEMENT OF UML MODELS

The Unified Modeling Language (UML) has become the de-facto industry standard for modeling software systems. The language has a rich graphical notation and comprises a comprehensive set of diagrams that are used to express the different aspects (views or viewpoints) of a system model at some level of abstraction. This includes static and dynamic views. The static view emphasizes the static structure of the system using objects, attributes, operations and relationships (e.g., class diagrams and composite structure diagrams). The dynamic view emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to their internal states (e.g., sequence diagrams, activity diagrams and state machine diagrams).

Although the use of multiple views has great benefits in focusing on a specific aspect of the modeled system and in reducing the amount of information to be handled at any given time, it also raises consistency and integration problems due to the following facts: 1) views may be interdependent and partially overlapping, 2) they may be expressed using different notations, and 3) they may be developed by different software developers. Add to this, the inconsistencies due to the lack of complete information or uncertainty inherent in the modeling process especially at earlier stages and of course any unintentional errors.

Another significant source of inconsistencies is the lack of the formal semantics provided by the language itself (e.g., the three views composing the UML metamodel are described informally – parts of the abstract syntax and the well-formedness rules as well as the entire semantics are described in natural language) besides the expressiveness limitations of the Object Constraint Language (OCL), the language which is used to formulate the well-formedness rules in the UML metamodel. Standard OCL is a side-effect free language in that it can detect the violations of rules but it doesn't allow making changes to the model elements to resolve them.

For all these reasons, it is intuitive to expect inconsistencies during the UML modeling process especially the case when all these different diagrams (that form the system model) experience changes as the software evolves. Keeping all these diagrams mutually consistent is not a trivial task and definitely needs a proper tool support.

In this section, we present an overview of part of the research work achieved in this area. We provide the most common definitions of inconsistency in Section 5.1, the general requirements for managing inconsistencies is listed in Section 5.2, the literature classification of types of inconsistencies and the common approaches to detect and resolve some of these inconsistencies is explained in Section 5.3 and 5.4 respectively, a review of some of the work which employs some of these approaches is presented in Section 5.5 and 5.6, and finally a discussion of our general observations is given in Section 5.7.

5.1 Inconsistency Definitions

Several definitions have been proposed to define inconsistency in the context of software engineering. Spanoudakis and Zisman [SZ01] describe inconsistency as “a state in which two or more overlapping elements of different software models make assertions about the aspects of the system they describe which are not jointly satisfiable”, while Nuseibeh et al [NER00] define inconsistency as “any situation in which a set of descriptions does not obey some relationship that should hold between them. The relationship between descriptions can be expressed as a consistency rule against which the descriptions can be checked”.

Based on these definitions, we can conclude that inconsistencies occur when part of the model refers to common aspects of the system under development and makes assertions which violate consistency rules (or constraints) applicable to these aspects. In the context of model-driven development, multi-level constraints can be found. Sourrouille and Caplat [SC02] identify five different types of such constraints which can be defined as stereotypes embedded with the UML metamodel elements. These are: 1) the well-formedness rules specified in the definition of the modeling language, 2) the paradigmatic constraints (e.g., semantic and stereotype constraints), 3) the style guide constraints inherited from the modeling process domain, 4) the target (or modeled) domain constraints, and 5) the implementation domain constraints.

5.2 Inconsistency Management

Many studies show that inconsistencies may have both positive and negative side-effects on the system development process. On the negative side, they make it more difficult to maintain the system as well as affect the reliability and the safety aspects of the system. On the positive side, they facilitate distributed teamwork and highlight conflicts between the views of the stakeholders involved in the development process which may help in revealing aspects of the system which deserve further analysis. In both cases, inconsistencies need to be managed, that is detected, analyzed, recorded and possibly resolved.

5.2.1 Requirements for Supporting Inconsistency Management

Grundy et al [GHM98] identify a list of key requirements for supporting inconsistency management in multiple-view software development environments. Among these requirements are the need for a description of syntax and semantics rules of the system architecture, a detection mechanism for possible violations of the architecture rules, information on inconsistency reasoning, inconsistency presentation, inconsistency monitoring strategy, inconsistency interaction and resolution, inconsistency resolution negotiation, and inconsistency management configuration. Based on this vision, they presented a model for inconsistency management which allows for recording, presenting, monitoring, and interacting with inconsistencies to help the developers resolve them.

5.2.2 Framework for Inconsistency Management

Spanoudakis and Zisman [SZ01] propose a general framework for managing inconsistency which unifies the frameworks of Finkelstein et al [FST96] and Nuseibeh et al [NER00]. The authors identify the following six activities: detection of overlaps, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, tracking, and specification and application of an inconsistency management policy. They also present a survey of methods and techniques supporting each activity. Figure 1 shows the framework for managing inconsistency as proposed in [NER00] where we can identify the following main steps.

- A knowledge base of the consistency rules (or constraints) needs to be checked.
- A diagnosis activity to identify the source, the cause and the impact of an inconsistency. This activity plays an important role in the activity of inconsistency handling.
- A handling activity to select a suitable plan for resolving a revealed inconsistency and to execute its corresponding actions. The selection of such plans depends on the type of the detected inconsistency and a pre-assessment of the cost and the benefits of resolving the inconsistency versus the risks of not resolving it.

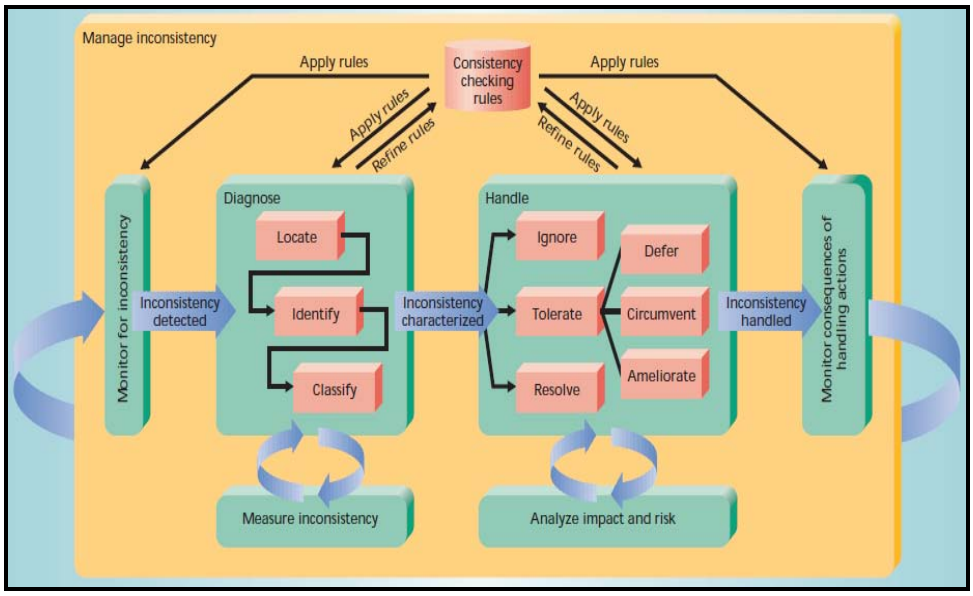


Figure 2: A Framework for Managing Inconsistency [NER00]

Possible options for handling an inconsistency, as presented in [SZ01] and [EB04], are as follows.

- Resolution actions can fully restore inconsistencies or just ameliorate them to reduce their severity (i.e., full vs. partial).
- Resolution actions can be automatically executed to modify the models to resolve inconsistencies or be executed only if selected by the users. In this latter case, users are

notified about inconsistencies and their possible resolution action plans, supported with sorts of analysis of the consequences of each plan (i.e., automatic vs. semi-automatic).

– Resolution actions can be applied on one inconsistency at a time or in a batch mode (i.e., incremental vs. batch mode).

Possible resolution actions are adding, deleting or changing some model elements.

A quite similar framework has been proposed by Reder in [Red11] for model-based development. The center of the proposed inconsistency management framework is the model to be investigated. The user interacts with the framework in two different places: the first to define the design rules the model must satisfy and the second to add/delete/modify model elements in order to resolve an inconsistency. Based on the pre-defined set of design rules, the framework will be able to detect possible violations of such rules, provide the user with an instant feedback of impacted model elements, analyze the model and prepare possible repair actions. It is the user's responsibility by then to choose what action to take. The workflows and interactions in the proposed inconsistency management framework are shown in Figure 2.

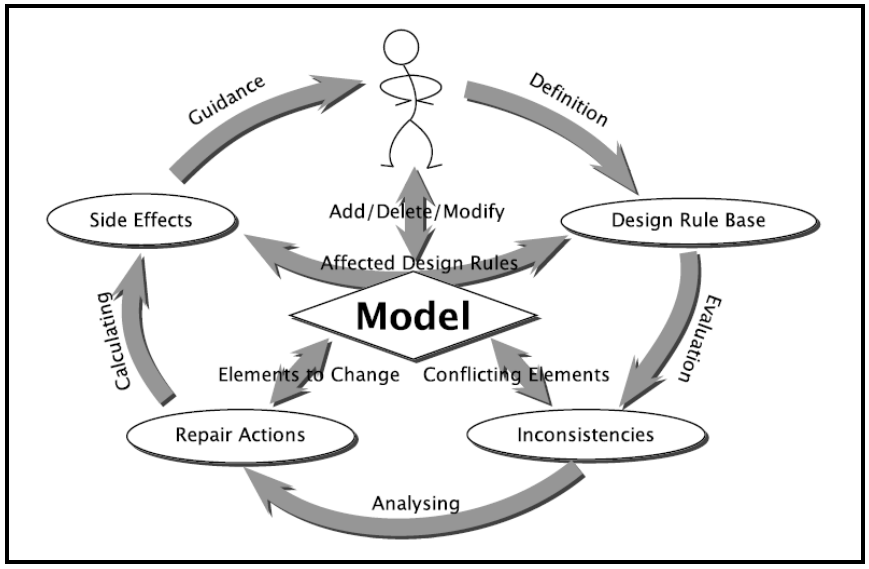


Figure 3: The Workflows and Interactions of Inconsistency Management Framework of Reder [Red11]

5.2.3 Strategies for Inconsistency Management

Snoeck et al [SMD03] identify three strategies for consistency management. The first strategy is consistency by analysis which means that the verification of consistency between models is done as a separate non-incremental static activity at the end of the development process or at regular intervals. The authors claimed that most existing approaches use the consistency by analysis strategy. The second strategy is consistency by monitoring which means that the detection of inconsistencies is done instantly within the modeling tool using a monitoring facility that checks every new specification against the already existing ones which guarantees that models are

always consistent. The third strategy is consistency by construction which means that the modeling tool automatically or semi-automatically generates one artifact from another as long as they are inter-dependent to guarantee their semantic consistency; this type of approach is usually complemented by an analysis algorithm for the parts of the model that are not constructed automatically (e.g., an analysis algorithm that checks unreachability and deadlock in Finite State Machines).

5.3 Inconsistency Classification

Since identifying the type of an inconsistency plays a significant role in selecting a suitable resolution plan, having a concrete classification of all possible inconsistencies would definitely help in the identification process.

5.3.1 Software Development Inconsistencies

A number of dimensions can be used to classify software development inconsistencies, including the type of consistency rule broken, the action type that caused the inconsistency, and the impact (or the side-effects) of the inconsistency; some inconsistencies are more severe than the others and hence need urgent attention [NER00].

5.3.2 Design Inconsistencies

In addition, a classification of design inconsistencies is proposed by Liu et al [LEM02]. It distinguishes three main categories as sources of inconsistency problems, including redundancy (or overlap) in design related representations such as redundancy and clashes in structural/behavioral diagrams, lack of conformance to constraints and common software design standards (known as design patterns), and of course changes that usually occur during the design stage of any software system (i.e., before completion due to requirements change requests or corrections).

5.3.3 UML Inconsistencies

Similarly, different classifications of inconsistencies in UML are specified in the literature. The following work specifies some of them. A summary of these classifications is shown in Figure 4.

5.3.3.1 Egyed's Classification

In [Egy00], Egyed propose a three-dimensional classification for views in UML, denoted by their level of generality, abstraction, and behaviorism. Based on this, he classified consistencies into three types: 1) consistencies within a single instance of a view; 2) consistencies between a set of instances of a view (i.e., instances that describe a refined versions of the view); and 3) consistencies between a set of instances of different views. The first type represents the *intra-view* consistency while the second and the third correspond to the *inter-view* consistency. Figure 3 depicts these views and the view space as presented in [Egy00].

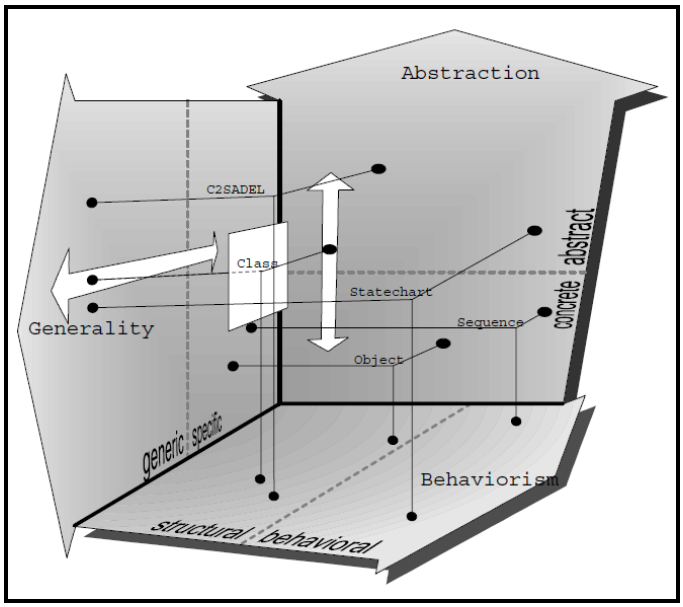


Figure 4: Views and the View Space [Egy00]

5.3.3.2 Engles et al's Classification

Engels et al [EKHG01] consider consistency of behavioral models in UML and they discussed two consistency problems. The first one arises from the fact that a system is modeled from different views and unless the specifications of these views are consistent and not contradictory, the implementation of such a system would be unfeasible. The second consistency problem arises when a specification is no more consistent with its refined one(s). The authors called these two types of consistency problems as *horizontal* consistency and *vertical* consistency, respectively. The horizontal consistency exists between views that belong to the same development phase or to the same level of abstraction (or detail), e.g., the consistency between a class diagram describing the static aspects of a conceptual model and the state machines (or the statecharts) describing the dynamic aspects of the classes in this model. The vertical consistency exists between views that model the same aspects but at different development phases (e.g., a conceptual model and a detailed design model) or different levels of abstractions. The authors also introduced another dimension of consistency problems which distinguishes between *syntactical* consistency and *semantic* consistency. The syntactical consistency ensures that a specification conforms to the abstract syntax of the modeling language specified by a metamodel. In general, this type of consistency can be automatically checked and hence is supported by current UML CASE tools. The semantic consistency, on the other hand, is concerned with the compatibility of the specified behavior; it requires models which construct the whole system model to be semantically compatible. For example, the different views of a model need to be semantically compatible, a refined model needs to be semantically compatible with the one it refines, and a re-factored model needs to be semantically compatible with the one it re-factors. In contrast to syntactic consistency, there exist no general methodologies for specifying semantic consistency rules and constraints.

Later on, Engels et al [EKHG02] bring up the idea of model-based evolution as a crucial aspect of model-based development which requires support for re-establishing the consistency of a new version of the model after an evolution step and so they introduced another type of consistency between different versions of the same model called *evolution* consistency.

5.3.3.3 Hnatkowska's Classification

Hnatkowska et al [HHKT02] introduce the notions of *intra-consistency*, a property of an artifact (a diagram, a view or a model) that is identified by some well-formedness rules, and *inter-consistency*, a relation between two artifacts that is governed by some rules. Artifacts are inter-consistent if they are self-consistent (i.e., intra-consistent) in the first place and satisfy the static semantic of the relation between each other.

5.3.3.4 Elaasar and Briand's Classification

Based on the multi-view nature of UML models and the different phases and iterations within the UML-based development process, Elaasar and Briand [EB04] and later on Huzar et al [HKRS05] set up the notions of *intra-model* consistency and *inter-model* consistency which are analogous respectively to the *horizontal* and the *vertical* types of consistency defined in [EKHG01]. The intra-model consistency indicates the consistency within a given model of a specific development phase which includes the intra-consistency of views (or diagrams) that are used to represent the model and the inter-consistency between these views. The inter-model consistency indicates the consistency between the different models which are produced from the different phases that make the complete system model (i.e., between requirement and analysis models, between analysis and design models, or even between design and implementation models); it also represents the vertical inter-consistency between the views (or diagrams) of these models.

5.3.3.5 Simmonds, Mens and Van Der Straeten's Classification

Simmonds et al [SSJM04], Mens et al [MS05], and Van Der Straeten [Str05] provide a detailed classification of the semantic inconsistencies in the conceptual model based on an extensive literature study of [Men01], [TE00], [EE95], [KRSH02], and [EKHG01]. The first dimension indicates whether *structural* or *behavioral* aspects of the model are affected. Structural inconsistencies occur when the structural diagram of the system is inconsistent with the system specification or with the system behavior. Typically this appears in class diagrams which describe the static aspects of the system and it can also appear in the behavioral diagrams due to missing features in the structural diagram. On the other hand, behavioral inconsistencies arise when the behavior specification of the system is inconsistent or incompatible with the structure specification. This can be found in sequence diagrams and state diagrams which describe the dynamic aspects of the system. The second dimension indicates the level of the affected model, i.e., either it is at the *specification* level, at the *instance* level, or between the specification and the instance level (*specification/instance*). For example, in UML diagrams, structure diagrams such as class diagrams and sequence role diagrams belong to the specification level which serve as specification for instances such as objects, links, transition, and events. While behavior

diagrams, such as sequence and state machine diagrams belong to the instance level. It is important to mention that UML distinguishes between interactions between objects and interactions between roles. The latter is a description of the interaction between roles objects can play, and the set of messages between these roles.

Examples of structural inconsistencies are: 1) a cyclic inheritance that arises when a composition relationship is specified between a superclass and a subclass, 2) a method parameter's or an attribute's type refers to a class that does not exist in the model, 3) a lifeline in the sequence diagram references a class that does not belong to the UML model, and 4) a transition in a state machine diagram whose event has been deleted from the UML model resulting in a set of states that are not reachable.

Examples of behavioral inconsistencies are: 1) the specification behavior incompatibility between a state machine diagram and a sequence diagram which occurs when a call sequence in the state machine diagram of a class doesn't match the order established by a receiving sequence diagram trace, 2) a call sequence of the superclass state machine doesn't present in the set of call sequences of the state machine of the subclass, 3) the ordered collection of messages received by an object of the superclass doesn't present in the ordered collection of messages received by an object of the subclass, and 4) the ordered collection of messages received by an object of the superclass in a sequence diagram, doesn't exist as a call sequence of the state machine for the subclass.

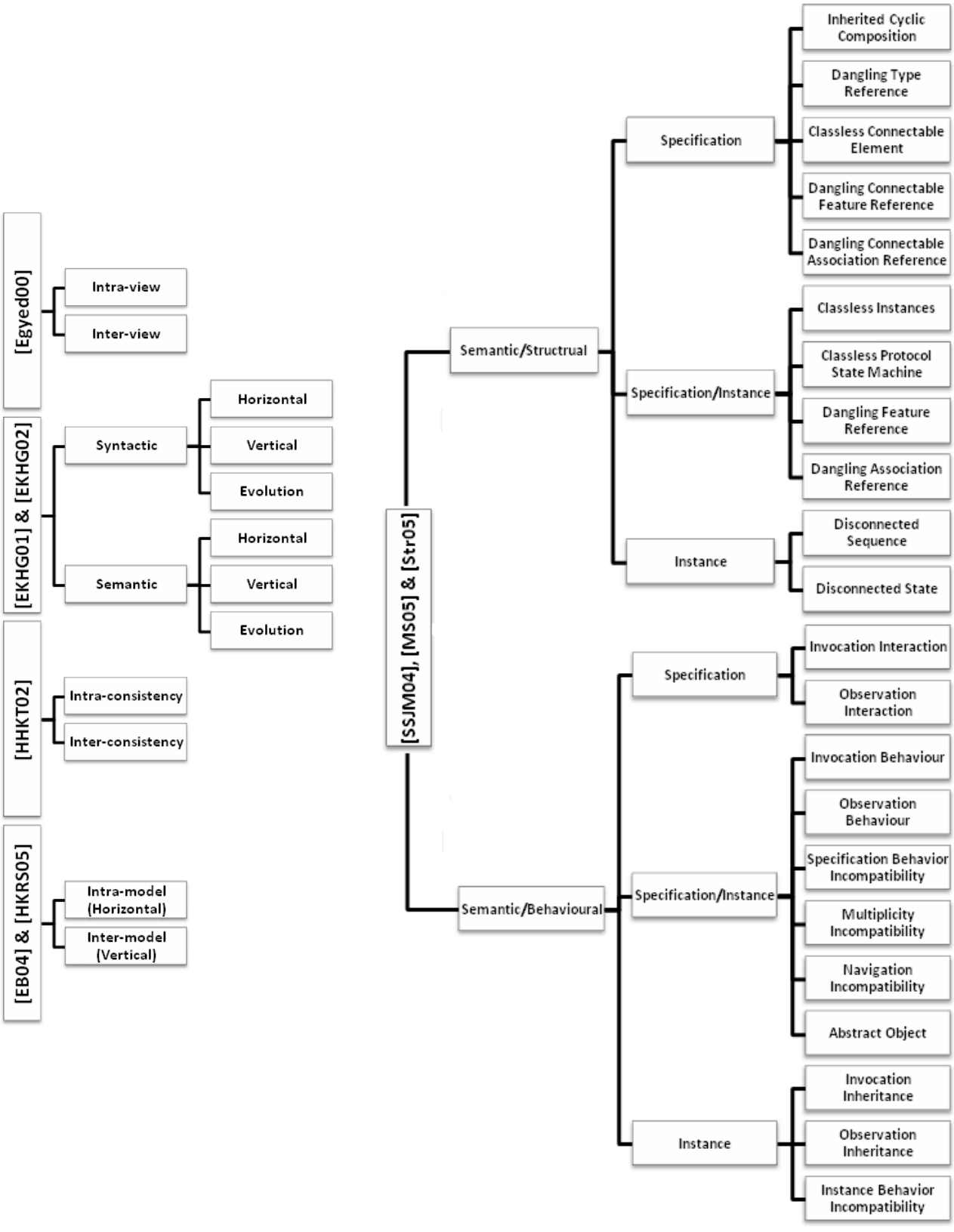


Figure 5: UML Inconsistencies Classification - Literature Summary

5.4 Classification of Existing Approaches for Inconsistency Detection and Resolution in UML Models

Different approaches for detecting inconsistencies in UML models are presented in the literature.

5.4.1 Spanoudakis and Zisman's Classification

Spanoudakis and Zisman [SZ01] list four categories of approaches.

- Logic-Based – Models are expressed in some formal logic. The limitation of this approach is the semi-decidability of first-order logic.
- Model Checking – Models are expressed in or translated into the state-oriented language used by a model checker. One drawback of this approach is the state space explosion problem.
- Special Forms of Analysis – Models are expressed in or translated into a specific language and only specific kinds of consistency rules can be checked (e.g., the detection of the well-formedness rules of UML models is performed directly but the detection of deadlocks and reachability in behavioral models is performed by translating these models into a different representation such as Petri Nets).
- Human-based collaboration exploration – Models are expressed in informal modeling languages where manual inspection of inconsistencies is performed. This approach is not feasible with large models.

5.4.2 Elaasar and Briand's Classification

Elaasar and Briand [EB04] present three main categories of the approaches that deal with UML consistency analysis.

- Meta-modeling Approaches – Changes or extensions are proposed and made to the UML metamodel by adding new model elements (as Stereotypes), creating new properties (as Tagged Values) and specifying new semantics (as OCL constraints) in order to facilitate and automate the inconsistency detection process. Example of this is the UML Profile for model evolution and consistency management presented in [SSJM04], [MS05], and [Str05].
- Constraint Language Approaches – Extensions to the Standard OCL are made to express non-static constraints.
- Formal Notations Approaches – Models are expressed in some formal language that provides the formal semantics required to handle some aspects of certain UML models. One main disadvantage of these approaches is that when a property is violated and detected in the formal language, the tools rarely point out the UML model features that cause the violation.

5.4.3 Usman et al's Classification

Usman et al [UNKC08] classify consistency checking techniques by whether they use an intermediate representation or not. Three different categories have been recognized.

- Intermediate Representation
  - Formally Represented Techniques – Similar to the Formal Notations category of Elaasar and Briand's classification [EB04].
  - Extended UML Representation – Extensions are made to the UML metamodel to support some consistency validation (also similar to the Meta-modeling category of Elaasar and Briand's classification [EB04]).
- No Intermediate Representation – The detection process is made on the models directly by monitoring whether any of the constraint rules of the modeling language is violated.

5.4.4 Lucas et al's Systematic Review

In their systematic review of UML model consistency management, Lucas et al [LMT09] state that based on the 44 articles included in their survey, they found that 75% of the approaches and techniques used for detecting and handling inconsistencies problems are formal. The most common formal methods used are State Transitions methods (e.g., B, Z, Petri Nets, etc.) and Logic methods (e.g., Theorem Proving and Description Logic). Nevertheless, some studies also examined the use of logic programming (e.g., Prolog), SAT solving, graph transformation, automated planning, and constraint satisfaction methods.

5.4.5 Our Classification

Based on what we've presented above and the number of work reviewed and discussed in this study (see the next section), we could identify the following distinct categories.

- Direct Manipulation Techniques – This category includes approaches that carry out the required task(s) within the domain of the modeling language.
- UML Domain Extension Techniques – The second category highlights the use of some extension to the modeling language to improve its semantics.
- Formal Notations Techniques – This includes approaches that make use of some formal notations [LMT09] in order to provide the appropriate semantic domain required for the analysis tasks at hand.
- Hybrid Techniques – This last category includes the approaches that combine more than one technique to detect and to handle the inconsistencies.

5.5 Existing Work on Detecting and Resolving UML Models Inconsistencies

This section discusses some existing work on detecting and resolving UML models inconsistencies. For each work, we have identified the type of the approach that has been adopted based on our classification as presented in Section 5.4.5, the work objective (detection, resolution, or both), the aspect of UML models that has been considered (structural or behavioral), the types of the UML diagrams involved in the work, the consistency levels achieved (horizontal, vertical, or evolution), the consistency management strategy employed

(analysis, monitoring, or by construction), and whether tool support for the proposed approach has been developed or not. A summary of our findings is given in Table 3 and 4.

5.5.1 Direct Manipulation Techniques

5.5.1.1 Model Profile

Egyed [Egy06, Egy07, Egy11] presents an automated approach to instantly detect inconsistencies that arise within UML models during the design phase and to propose choices for fixing them. The approach is based on 24 well-chosen consistency rules (i.e., the most convenient and needed ones including the UML well-formedness rules for UML class, sequence, and state diagrams). These consistency rules are defined as conditions that a UML model must satisfy to be considered a valid UML model. Changing a UML model in such a way that violates one or more of these consistency rules will affect the truth values of the violated conditions and so the set of model elements involved in these conditions can be considered as the change impact scope of the consistency rule. Following this scenario, a model profiler is developed to monitor the runtime behavior of consistency rules during their evaluation and to detect the scope of each violated rule that is simply the set of model elements accessed during the rule's evaluation. Any attempt to do a design change on any of the scope's elements to fix an inconsistency may also introduce some other inconsistencies. To help the designers in their decision actions, the author managed to annotate each element with a mark indicating the type of the side effect (e.g., positive, negative, or both) of changing it. UML/Analyzer and Model/Analyzer tools are developed to automate and evaluate the approach. The first tool is based on the UML1.3 infrastructure and is integrated with IBM Rational Rose while the second is based on the EMF (Eclipse Modeling Framework) and is integrated with IBM Rational Software Modeler. The main components used in the implementation are a *Consistency Checker*, an *Evaluation Profiler*, and a *Rule Detector*. An empirical validation of the approach has been conducted to test the completeness and the scalability of the approach as well as to validate the recurring cost of computing changed truth values and scopes.

5.5.2 UML Domain Extension Techniques

5.5.2.1 Dynamic Meta Modeling (DMM)

Engels et al [EHHS02] propose a testing approach for consistency checking of UML diagrams based on the notion of Dynamic Meta Modeling (DMM). In DMM, the idea of defining meta-operations for the classes of the UML metamodel to represent their dynamic semantics has been introduced. Collaboration diagrams are used to define the behavior of these meta-operations. The basic architecture of the proposed approach depends on the fact that one diagram can be used to derive and generate the testing data of another inter-dependent diagram given the DMM rule set of each diagram as well as the DMM rule set describing the consistency rules between the two diagrams, then the environment checks whether the two diagrams conform to the given consistency rules or not. In such a context, the main components of the architecture are the diagram to be used as a test driver, the diagram to be tested, the DMM rule set for both diagrams,

the DMM rule set for consistency, an interpreter for the DMM rules, and a test controller to coordinate the interaction between the test driver and the tested model. The focus in this work was to apply the approach to check the horizontal semantic consistency between behavioral diagrams, specifically sequence and statechart diagrams. Yet, the approach can be also applied to check the vertical semantic consistency.

5.5.2.2 Additional OCL Constraints on the UML Metamodel Elements

Vasilecas et al [VDR11] propose an approach to ensure UML model consistency by including consistency rules on relationships of different model aspects (i.e., between structural and behavioral views). Such consistency rules are expressed in OCL and are defined on the UML metamodel. The authors suggested three consistency rules to enforce the following: 1) each transition in a state machine model has to be defined by an operation that is already defined in the structure model; 2) the context of the state machine has to be defined by one of the classes presented in the structure model; and 3) the type of lifeline in a sequence diagram should be specified by one of the classes presented in the structure model. A prototype of the proposed approach is implemented as a consistency constraints module of the MagicDraw UML tool. We think that a similar approach has been developed and used in other UML tools such as IBM Rational Rose RealTime.

5.5.3 Formal Notations Techniques

5.5.3.1 Communicating Sequential Processes (CSP)

Engels et al [EKG01] propose a general methodology for semantic consistency checking of concurrent behavioral models (e.g., capsule statecharts and protocol statecharts) in UML-RT, an extension of UML which enables modeling of Real-Time systems. The methodology is based on a partial formalization of the aspects of the model that lead to a consistency problem onto a chosen semantic domain which supports the analysis and the verification of the relevant aspects (i.e., in terms of the availability of good language and tool support). First, a mapping into such a semantic domain is defined. Second, a formulation of consistency conditions is made within the semantic domain to enable the formal verification of the consistency of individual models. Two consistency conditions for UML-RT models are defined. The first one demonstrates a horizontal type consistency which requires deadlock freedom between the statecharts of two capsules and their connector. The second one represents vertical type consistency which needs to be satisfied between any two connected capsules and the protocol statechart of their connector. A formal language called Communicating Sequential Processes (CSP) is the chosen semantic domain. It is a mathematical language used to describe and reason about concurrent systems. The FDR (Failures-Divergence-Refinement) tool is the model checker used to analyze and verify CSP expressions.

Engels et al [EHK01] adopted the semantic consistency checking methodology of [EKG01] in proposing a flexible and extensible translation of the elements of UML models into a CSP formalization using graph transformation of meta-model rules. The notion of behavior

inheritance of statecharts is discussed as well as the definitions of the consistency constraints required to manage it are provided. Tool support (Consistency Workbench) which realizes the methodology of [EKG01] and [EHK01] is presented in [EHK03].

Engels et al [EHKG02] propose an incremental consistency verification approach to manage the evolution of UML-RT models in such a way that preserves a given set of consistency properties such as protocol consistency and deadlock freedom. The proposed technique employs a rule-based model transformation strategy to the CSP semantic domain where transformation rules are defined for all possible evolution steps that can be performed on the model elements (e.g., creating, deleting or updating capsules, connectors or protocol statecharts). Also a set of conditions are defined for the application of the transformation rules to ensure the preservation of the required consistency properties.

5.5.3.2 Object-Z & CSP (CSP-Z)

Rasch and Wehrheim [RW03] propose an approach that is quite similar to the work by Engels et al [EKG01, EHK01] in using the Communicating Sequential Processes (CSP) process algebra formalism to check the consistency of UML diagrams. While Engels et al considered the consistency between behavioral views, Rasch and Wehrheim considered the consistency between two different types of views, a structural view represented by a class diagram and a behavioral view represented by a state machine diagram. The Object-Z specification language is used to provide a more precise and formal description of the static aspects of the system (i.e., the system's classes). The CSP process algebra is used as the common formal semantic domain to examine the consistency issue among these diagrams. This required 1) the translation of the Object-Z specification and the state machine into the CSP formalism, 2) the definition of some notions of consistency in CSP, and 3) the use of the FDR model checker. The types of consistency notions studied are the method liveliness and the method availability between an Object-Z class and its associated state machine and the deadlock freedom of the combined semantic model.

5.5.3.3 Colored Petri Nets (CPN)

In [Shi06], Shinkawa proposes a methodology to establish consistency between a set of UML dynamic diagrams (Use case, Activity, Sequence, and State Machine diagrams) based on the use-case driven approach and Colored Petri Net (CPN). The author presented a classification of these diagrams based on whether they represent the external (e.g., Use case and Activity diagrams) or the internal (e.g., Sequence and State Machine diagrams) behavior of the system. In this approach, the UML diagrams are converted to their Colored Petri Nets representation. Then the consistency between the Colored Petri Net models is checked. Finally, a mapping of the consistent Colored Petri Net models to their equivalent UML diagrams will guarantee the consistency between these diagrams.

5.5.3.4 Logic-based Expert System

An expert system, a knowledge-based system or a production rule system are all synonyms for a system which consists mainly of a set of pre-defined rules (or productions) that form the basic representation in recognizing a chosen set of different types of inconsistencies and in providing the appropriate repair plans for them. This requires the use of a logic-based engine and a powerful constraint language which will be used to define the required set of constraints and to automatically check for possible violations. Since most of such systems aren't within the UML context, performing the detection activity requires the translation of the UML models into the expert system's constraint language formalism. To fully automate the detection and the resolution process, an integration of such an expert system with one of the UML CASE tool is required.

Such an idea has been adopted by Liu et al [LEM02] and Sourrouille and Caplat [SC02]. The former used *Jess*, a rule engine for the Java platform, while the latter used *Sherlock*, a constraint language which is more expressive than the Object Constraint Language (OCL) and hence enables the definition of a more coherent set of constraints which can be automatically checked.

5.5.3.5 First-order Logic Based Query Languages (XPath, Beanbag, and SQL Triggers)

Nentwich et al [NEF03] extended their work in [NCEF02] to present a framework for repairing inconsistent XML documents including UML design models encoded in XMI. The framework proposes an extension to the *XPath* query language which is used for selecting nodes from an XML document with first-order logic for expressing consistency rules. Based on these rules, a checker can then detect inconsistencies between a set of distributed XML documents. Based on the static analysis of these rules, a repair administrator can create a set of repair actions. Three types of modifications can be made in a repair action: add, delete and change to resolve consistency. The user then decides which actions to execute and so it is considered to be a white-box analysis. The framework seems to be well suited to detect and fix structural inconsistency rules and to support establishing inter-model constraints. On the other hand, it only allows resolving one inconsistency at a time. The framework treats repair actions as independent events. It does not consider dependencies among inconsistencies.

Xiong et al [XHZSTM09] define a language, Beanbag, to specify inconsistency rules (similar to OCL) and the fix procedures to resolve the inconsistencies of MOF-based models mapped into Beanbag programs. Beanbag has enriched constructs to describe a unique fixing behavior for each inconsistency rule. Every Beanbag Program has two types of semantics: checking semantics and fixing semantics. The former is for checking whether the relation is satisfied while the latter is for fixing an inconsistency by update propagation. The fixing semantics of Beanbag is defined such that it satisfies the following three properties: consistency (after fixing, the data always satisfy the consistency relation), preservation (a fixing procedure does not overwrite user updates), and stability (if there is no update, the fixing procedure produces no update).The approach is completely automatic and doesn't require user interaction.

Sapna and Mohanty [SM07] present an approach to ensure the structural consistency of a relational repository of UML models. In such approach, consistency rules of Use case, Activity, Class, Sequence, and State Machine diagrams are expressed in OCL and are then converted to SQL triggers that get executed automatically when the database is modified to enforce the integrity and consistency of the database schema representing the UML diagrams. No proof of concept for the proposed approach is provided.

5.5.3.6 Graph Grammars & Graph Transformation

Wanger et al [WGN03] present a plug-in for a flexible and incremental consistency management based on the Graph Grammars formalism, a graphical and operational specification language that has the ability to modify object structures (in contrast to the Object Constraint Language (OCL)) that was used to specify consistency rules and possible repair actions. In this work, only syntactical consistency rules were considered and they can be refined or extended, activated or deactivated on demand. A change-driven technique is implemented based on change events to execute the consistency checking algorithm on only the part of the model that is modified and hence reduce the time and effort of the consistency checking process. In the same way, an inconsistency-driven method is used to start the resolution process. The proposed approach has been realized and evaluated in the Open Source CASE tool Fujaba.

Mens et al [MSH06, MS07] propose an approach and developed a tool to detect and resolve model inconsistencies using graph transformation. Given a UML model specification as a graph and the UML metamodel as a typed graph, a subset of model inconsistencies rules specifying structural inconsistencies (such as dangling type references, classless instances, abstract objects, abstract operations, abstract state machines, and dangling operation references) and their corresponding resolution rules are specified as graph transformation rules in the AGG graph transformation tool. The provided list of inconsistencies rules have been adopted from [SMSJ03] and are used to automate the detection of inconsistencies. For each inconsistency rule, a set of possible resolutions are defined to interactively support the resolution. The authors also applied a critical pair static analysis to 1) study if resolving existing inconsistencies introduces new inconsistencies, 2) determine resolutions which negate other resolution rules, 3) identify cycles in the resolution process. This analysis helped them to construct a dependency graph that shows all mutual exclusions between resolution rules of distinct model inconsistencies as well as all possible sequential dependencies between distinct resolution rules.

5.5.3.7 B Method

The B method is a formal method of software development. It has been used in safety-critical system applications. It also has a robust tool support for specification, design, and code generation. B is related to Z notation but it is focused on refinement to code.

Ossami et al [OJS05] propose an approach to maintain the consistency of an evolving specification based on the integration of the two notations in representing the system specification: the UML modeling language and the B formal method in order to benefit from the

graphical representation of the first and the effective formal verification methods of the second. The side by side development and refinement of the two representations is achieved using a set of defined operators. The *Refine-Data* operator describes the new definitions of some parts of the specification's elements according to the required changes. The *Model-Constraint* operator describes the constraints (or invariants) that are needed over the specification's elements to express the logical links between the refined parts and their abstract ones. Verifying the correctness of these operators is done by satisfying four conditions of the consistency relation: 1) the syntactic conformance of the well-formedness rules of the UML specification, 2) the internal consistency of the B specification, 3) the elements' traceability between the two specifications, and 4) the semantic preservation conditions which ensure that the B specification satisfies the same requirement as its UML counterpart. As the correctness of each operator is defined, the specification obtained from applying these operators is shown to be correct.

5.5.3.8 Logic Programming (Prolog)

Almeida da Silva et al [AMBB10] propose an approach for resolving inconsistencies in EMF models based on their previous work in [BMMM08] which introduced a Prolog-based formalism for representing models by sequences of elementary construction operations (e.g., create a class, set a property, set a reference, etc.). In such a context, the logical formulae of consistency rules are defined on the construction operations and detecting inconsistencies is done on the set of operations performed to construct the models instead of the model elements themselves. By detecting the problematic actions (operations) that caused inconsistencies, a set of repair plans (a sequence of repair actions) can be generated and presented to the user to resolve the consistency. The prototype implementation of the approach is composed of three components: the *Sequence Builder* for building the model sequence while the user is creating the model, the *Check Engine* for detecting inconsistencies, and the *Model Fixing Agent* for proposing the repair plans for the fixing process. A depth-first tree search algorithm is used to generate efficient repair plans that start by fixing the most recent causes of inconsistencies. However the proposed framework is suited to detect the intra-model and inter-model structural inconsistencies as well as methodological consistency (i.e., development process compliance rules), it can resolve only syntactical (well-formedness) inconsistencies. The tool support implemented is integrated with the Eclipse EMF Framework and the Rational Software Architect

Khai et al [KNL11] propose an approach for horizontal consistency checking of UML class and sequence diagrams using Prolog. In this approach, both UML models (i.e., class and sequence diagrams) and consistency checking rules are translated into Prolog predicates and rules respectively. Only a subset of the structural and the behavioral inconsistencies presented in [Str05] is considered including those that check for specification incompatibility (e.g., Multiplicity Incompatibility) and missing instance specification (e.g., Classless Connectable Element, Dangling Feature Reference, and Dangling Association Reference). Reasoning about inconsistencies is performed using the Prolog Reasoning Engine.

5.5.3.9 Constraint Solver

Van Der Straeten et al [SPM11] use Kodkod, a SAT-based constraint solver (model finder) for first-order relational logic with relations, transitive closure and partial models, for automatically detecting and resolving structural inconsistencies in UML models. In this approach, a UML model is translated to a Kodkod problem in terms of a set of atoms, a set of relation declarations and a formula; similar to existing translations of UML models to Alloy. The authors have built an Eclipse plug-in to automate this task. In order to generate consistent models with respect to a consistency rule, a manual translation of such a consistency rule to Kodkod formula is specified. Based on the lower and upper bounds of the relations defined in the Kodkod problem which can be changed manually, a list of possible locations (i.e., possible model elements) for resolving the inconsistency is identified by Kodkod and all possible sets of consistent models are returned. The authors consider only 12 model inconsistency rules that can be expressed in Kodkod such as cardinality constraints, quantified constraints, comparison constraints, and negated constraints. While the approach guarantees correctness and completeness, it has a major limitation in terms of its poor performance and lack of scalability. It doesn't provide instantaneous resolution on medium scale models.

5.5.3.10 MERODE Methodology Formalism

MERODE (Model driven, Existence dependency Relation, Object oriented DEvelopment) is a method (not a language likes the UML) for requirements engineering that follows a model-driven engineering approach. It offers methodological guidelines on how to build models and how to check their quality which can be considered as a complementary to the UML. MERODE has a CASE-tool JMERMAID that is used to build and verify models as well as transform them to code. The method is based on the Business Event concept.

Based on this, Snoeck et al [SMD03] illustrate how the consistency by construction strategy is employed in the MERODE modeling tool and demonstrated the importance of this approach in ensuring the completeness of specifications. In their approach, a class diagram is used to represent the static structure and to generate a slightly different proprietary diagram called an Existence Dependency Graph (EDG) that demonstrates the existence dependency relationship between classes with explicit notations. An Object Event Table (OET) is used to maintain all the events that occur during the life time of the system. Finite State Machines (FSMs) are used to represent the behavior of each class. A common formal specification is used to model the different types of views. A set of defined rules are used to implement the consistency strategy needed.

5.5.4 Hybrid Techniques

5.5.4.1 Description Logic & UML Profile

Mens, Van Der Straeten and Simmonds [Sim03, SMS03, SSM03, SMSJ03, SSJM04, Str05, MS05] present a technique for detecting and resolving inconsistencies in different versions of UML models using Description Logics (DLs). Description Logic is a logic-based knowledge

representation formalism for modeling a domain in terms of concepts (classes), roles (properties and relations) and individuals (instances of classes). A subset of the UML metamodel has been extended and presented as a UML Profile to support model evolution and inconsistency management. A thorough list of inconsistencies that may arise between different versions of a UML model has been defined. Firstly, they specified their proposed UML Profile in one of the existing DL systems (initially they used LOOM but later on they used RACER). Secondly, they translated the XMI representation of specific UML models into a DL representation. Thirdly, they have developed DL predicates for the pre-defined inconsistencies as well as their possible resolution actions. Finally, a DL query processor is used to automatically detect inconsistencies between models and to propose possible resolution actions. Two proof-of-concept tools are developed as plug-ins integrated with the Poseidon CASE tool – the first one is introduced in [Sim03] and called Conan (Consistency Analyzer for UML) while the second one is presented in [Str05] and named RACOoN (Resolution Actions for inCONsistencies).

Table 3: A Summary of the Surveyed Work on Detecting and Resolving UML Model Inconsistencies (1/2)

#		Reference(s)	Task(s) Achieved	Technique Name	Aspects Emphasis	Consistency Level Acquired	Diagram Type	Consistency Management Strategy	Tool Support
1	Direct Manipulation	[Egy06, Egy07, Egy11]	Detection & Resolution	Model Profile	Syntactical + Structural/Behavioral Semantic	Horizontal	Class Diagram Sequence Diagram State Machine Diagram	Monitoring	UML/Analyzer & Model/Analyzer
2	UML Extension	[EHHS02]	Consistency Verification	Dynamic Meta Modelling (DMM)	Behavioral Semantic	Horizontal/Vertical	Sequence Diagram Statechart Diagram	Analysis	
3		[VDR11]	Consistency Verification	Additional OCL Constraints on the UML Metamodel Elements	Structural Semantic	Horizontal	Class Diagram Sequence Diagram State Machine Diagram	Monitoring	Consistency Constraints Module of MagicDraw UML tool
4	Formal Notations	[EKG01]	Consistency Verification	CSP (Communicating Sequential Processes)	Behavioral Semantic	Horizontal/Vertical	Capsule Statecharts (UML-RT) Protocol Statecharts (UML-RT)	Analysis	Consistency Workbench
5		[EHK01]	Consistency Verification	CSP + Graph Transformation	Behavioral Semantic	Vertical	Capsule Statecharts (UML-RT)	Analysis	Consistency Workbench
6		[EHKG02]	Consistency Verification	CSP + Rule-based Transformation Strategy	Behavioral Semantic	Evolution	UML-RT models	Analysis	
7		[RW03]	Consistency Verification	Object-Z and CSP (CSP-OZ)	Structural/Behavioral Semantic	Horizontal	Class Diagram State Machine Diagram	Analysis	
8		[Shi06]	Consistency Verification	Colored Petri Nets (CPN)	Behavioral Semantic	Horizontal/Vertical	Use case Diagram Activity Diagram Sequence Diagram State Machine Diagram	Analysis	
9		[LEM02]	Detection & Resolution	Logic-based Rule Engine (Jess)	Structural Semantic	Horizontal/Vertical	Class Diagram Sequence Diagram State Machine Diagram	Analysis/Monitoring (if integrated)	RIDE (Rule-based Inconsistency Detection Engine) - A proof of concept tool and it is not integrated with any CASE tool.
10		[SC02]	Consistency Verification	Logic-based Expert System (Sherlock)	Syntactical + Structural Semantic	Horizontal	Class Diagram Sequence Diagram	Analysis/Monitoring (*if implemented*)	
11		[NEF03, NCEF02]	Detection & Resolution	XPath (First-order Logic)	Syntactical + Structural Semantic	Intra-view	Any	Analysis/Monitoring	Repair Manager Tool

Table 4: A Summary of the Surveyed Work on Detecting and Resolving UML Model Inconsistencies (2/2)

#		Reference(s)	Task(s) Achieved	Technique Name	Aspects Emphasis	Consistency Level Acquired	Diagram Type	Consistency Management Strategy	Tool Support
12	Formal Notations	[XHZSTM09]	Detection & Resolution	Beanbag Language (First-order Logic)	Syntactical + Structural Semantic	Horizontal	Class Diagram Sequence Diagram State Machine Diagram		Beanbag
13		[SM07]	Consistency Verification	SQL Triggers (First-order Logic)	Syntactical + Structural Semantic	Horizontal/Vertical	Use case Diagram Activity Diagram Class Diagram Sequence Diagram State Machine Diagram	Monitoring & Construction	
14		[WGN03]	Detection & Resolution	Graph Grammars	Syntactical	Intra-view	Class Diagram	Monitoring	Part of the Open Source UML CASE tool Fujaba
15		[MSH06, MS07]	Detection & Resolution	Graph Transformation	Structural Semantic	Horizontal	Class Diagram State Machine Diagram	Analysis	SIRP: Simple Iterative Resolution Process Tool Support
16		[OJS05]	Consistency Verification	B Method	Structural Semantic	Evolution	Class Diagram State Machine Diagram	Construction	
17		[AMBB10, BMMM08]	Detection & Resolution	Prolog (Logic Programming)	Syntactical	Intra-Model	EMF Model	Analysis/Monitoring	Praxis: Operation Based Consistency Engine Integrated with: Eclipse EMF Framework and Rational Software Architect
18		[KNL11]	Consistency Verification	Prolog (Logic Programming)	Structural Semantic	Horizontal	Class Diagram Sequence Diagram	Analysis	
19		[SPM11]	Detection & Resolution	Kodkod (SAT-based Constraint Solver)	Structural Semantic	Horizontal	Class Diagram Sequence Diagram	Analysis	Eclipse plug-in
20		[SMD03]	Consistency Verification	MERODE methodology formalism	Structural/Behavioral Semantic	Horizontal	Class Diagram Existence Dependency Graph (EDG) Object-event Table (OET) Finite State Machine (FSM)	Construction & Analysis	Part of the MERMAID modeling tool
21	Hybrid	[Sim03, SMS03, SSM03, SMSJ03, SSJM04, Str05, MS05]	Detection & Resolution	Description Logic + UML Profile	Structural/Behavioral Semantic	Horizontal/Vertical/Evolution	Class Diagram Sequence Diagram State Machine Diagram	Analysis/Monitoring	Conan and RACOoN

5.6 Existing Work on Generating Effective Resolution Plans

In conjunction with the work on detecting and resolving UML inconsistencies, there is some work has been done to generate effective resolution plans: 1) by utilizing the dependency relationships between inconsistencies [NRE11, NE10], 2) by evaluating the side-effect and the cost of each resolution plan [KR07, ELF08], and 3) by understanding the extent of changes posed by each resolution plan [KSD09]. Table 4 shows a summary of this work.

Nohrer et al [NRE11, NE10] provide an empirical study to demonstrate the interrelationships between inconsistencies and how to benefit from this in generating resolution plans, with fewer fixing actions to fix clusters of interrelated inconsistencies at a time. The conclusions drawn in this work are based on four industrial models using different types of diagrams such as class diagrams, sequence diagrams, state-charts and use-case diagrams. The consistency rules they considered are a subset of the syntactic rules defined by the underlying meta-model. After introducing changes by which each consistency rule is violated and then applying the possible fixing plans to resolve any presented inconsistency, they identified a set of overlapping inconsistencies in each model. Having such clusters of overlapping inconsistencies helps in providing a better understanding of the problem that caused them and so reduces the complexity of fixing them.

Küster et al [KR07] provide a side-effect evaluation and a cost for each inconsistency type which allows the user to compare between alternative resolutions and to choose the resolution that introduces fewer side-effects. The subject models in this study are the two proprietary models, the Object Life Cycle and the Business Process models used in the IBM Insurance Application Architecture framework. Two types of inconsistencies are distinguished: non-conformant and non-coverage. Inconsistencies of the first type are given a higher priority to be resolved than those of the second type. Such priorities are made explicit in the resolution process. Each inconsistency has a set of alternative resolutions. Resolutions may have impacts (i.e., side-effects) in such a way that applying a certain resolution may not only resolve its targeted inconsistencies but also introduce new inconsistencies (negative side-effect) or fixing other existing inconsistencies (positive side-effect). For this reason, the side-effects and the cost reduction of each resolution are calculated to indicate the best ones to choose which helps in building an efficient inconsistency resolution module. A prototype for the proposed approach has been implemented as an extension to IBM Web Sphere Business Modeler.

Egyed et al [ELF08] propose an approach that generates possible resolution choices for inconsistencies. The authors have developed a tool support to help the modeler to explore possible ways to fix inconsistencies and to anticipate the effect of such changes simultaneously. Based on a pre-defined set of consistency rules, the approach uses model profiling to determine the model elements involved in an inconsistency. By locating such impacted elements, a choice generator is used to generate possible resolutions that involve change to an existing model element (i.e., a single location) at a time. Resolutions can change the values of some specific

fields of a model element. Users are allowed to manually customize the choice generator functions and to choose among a set of resolution options. This approach doesn't require a user to define or create consistency rules and therefore it is considered a black-box analysis.

Keller et al [KSD09] propose a change impact analysis algorithm to support inconsistency management. Their idea is to keep track of all model elements impacted by a change through the aggregation and navigable meta-model relationships of all model elements involved in a change. They compared between the number of the impacted model elements calculated from their impact analysis algorithm and those which result from actual resolution actions in a small case study. Inconsistency rules involved in this study have been taken from [Str05] and the UML2 well-formedness rules. Actual changes to the model have been applied manually using the TOPCASEDUML editor while the number of impacted model elements after the change is counted by the EMF compare tool that compares the model before and after the change. The results showed that the proposed impact analysis algorithm is more accurate for add and remove type of changes and less accurate for the modification changes and they explained this by some side-effects introduced by the tools used in their case study.

Table 5: A Summary of the Surveyed Work on Generating Effective Resolution Plans

#		Reference(s)	Task(s) Achieved	Technique Name	Aspects Emphasis	Consistency Level Acquired	Diagram Type	Tool Support
1	Resolution Plans Analysis	[NRE11, NE10]	Utilizing the Dependency of Inconsistencies	Inconsistencies Clustering	Syntactical	Horizontal/Vertical	Use case Diagram Class Diagram Sequence Diagram State Machine Diagram	
2		[KR07]	Evaluating the Side-effect and the Cost	Priority-based Analysis	Comformant & Coverage	Horizontal	Object Life Cycle Model Business Process Model *(Proprietary models used in IBM Insurance Application Architecture framework)*	An extension to IBM Web Sphere Business Modeler
3		[ELF08]	Evaluating the Side-effect and the Cost	Model Profiling	Syntactical + Structural Semantic	Horizontal	Class Diagram Sequence Diagram State Machine Diagram	
4		[KSD09]	Understanding the Extent of Changes	Change Impact Analysis	Syntactical + Structural Semantic	Evolution	Class Diagram	

5.7 General Observations

Based on the summary provided in Table 3 and 4, we can draw the following conclusions.

- A large number of the research work makes use of some formal notations formalism compared with the other techniques. In this specific case, the consistency management strategy taken is the analysis strategy which can be complemented with a monitoring strategy if the tool support of the employed formalism has been integrated with one of the UML CASE tools such as in [NEF03, AMBB10, Str05, Sim03].
- A limited set of constraints is covered by each work except for the work in [Str05].
- Formal notations such First-order logic based formalisms, SAT solvers and Programming Logic are mostly used to handle quantified, cardinality, comparison, and negated constraints.
- Communicating Sequential Process and Colored Petri Nets are most frequently used for behavioral aspects verification such as behavioral inheritance, protocol consistency and deadlock freedom.
- Techniques for resolving inconsistencies focus on resolving syntactical, well-formedness, and structural inconsistencies. Very few studies are made on resolving behavioral inconsistencies as in [Str05, Sim03].
- 70% of the approaches tackled the consistency problem on the horizontal level.
- The types of UML diagrams mostly common used are the class, the sequence and the state diagrams.
- 50% of the surveyed work doesn’t have tool support to test the feasibility of these approaches.
- The most active research groups in this field are supervised by Gregor Engels, Tom Mens, Ragnhild Van Der Straeten and Alexander Egyed. This conclusion is based on the number of papers they have published as well as their citation numbers.

Consistency management of UML models is a non-trivial task and it is still an open issue. The literature has a lot of ambiguities in identifying and classifying the different types of inconsistencies.

6. COMMON TASKS OF MODEL EVOLUTION – CHANGE PROPAGATION WITHIN & ACROSS UML MODELS

In the context of model-driven development, a software system typically comprises multiple inter-related models (or views) which might have overlapping information. Ensuring the overall consistency and integrity of these inter-related models requires significant effort, especially during the evolution and the maintenance of the software system when new changes are introduced. In such cases, changes in some part of the system models have to be properly propagated to all other inter-related models otherwise it may cause violations of the consistency relationships between these inter-related models. In the model-driven development community, this process is referred to as “Change Propagation” or “Model Synchronization”. Two types of change propagation processes are identified: intra-model (or horizontal) change propagation and inter-model (or vertical) change propagation. The first one propagates the changes within the same model where the changed artifacts exist (e.g., changes in the structural aspect of the design model may require changes in the behavioral aspects of the same model) and the second one propagates the changes to other inter-related models (e.g., changes in the design model may require also changes in the test model or in the source code). It is necessary to apply both types of propagation to have fully coherent models.

We can identify the following situations where change propagation mechanisms need to be invoked:

- Intra-model change propagation within the requirements model. It is important to emphasize the importance of the requirement model for deriving the change propagation process to all downstream models especially at the early stages of the system development where changes to the requirement model occur frequently to explicate and refine uncertain and incomplete requirements;
- Intra-model change propagation within the design model;
- Intra-model change propagation within the implementation model;
- Bidirectional inter-model change propagation between the requirements model and the design model;
- Bidirectional inter-model change propagation between the design model and the implementation model (i.e., source code artifacts and other configuration artifacts);
- Bidirectional inter-model change propagation between the design model and its analysis model (or test model).

Figure 6 depicts the ideal case of a fully automated change propagation process which covers the entire development process.

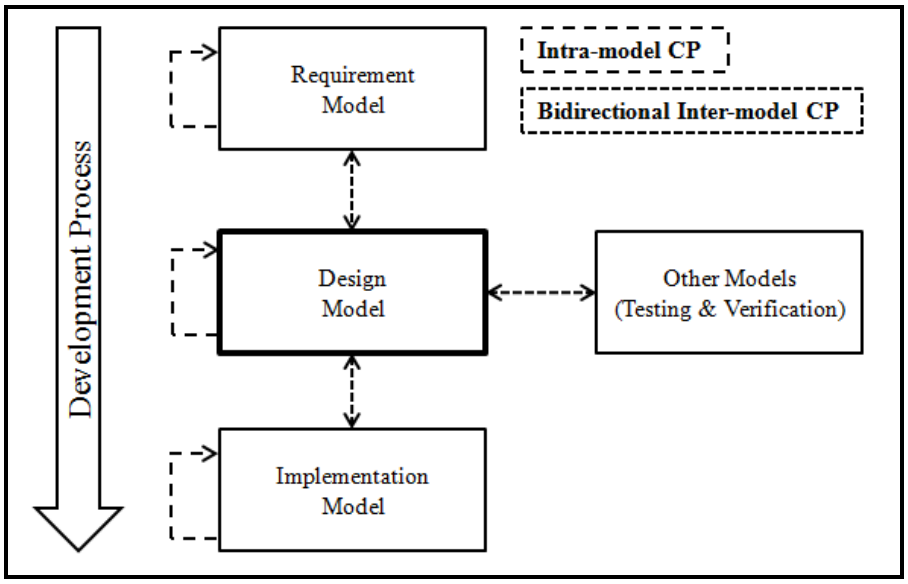


Figure 6: Change Propagation in Software Development Process

The focus in this study is to look for approaches that are applied to support change propagation within UML design models and across other inter-related models such as the implementation model (i.e., source code artifacts) and other related models that are created to support the analysis and the verification of the design models.

6.1 A Taxonomy of Change Types

Identifying the types of potential changes is a very important step of the change propagation process. For that reason, Lehnert et al [LFR12] propose comprehensive four-dimension taxonomy of change types in software evolution based on analytical reviews of different classifications of change operations found in the literature. According to the proposed taxonomy, a change is classified by four criteria, 1) the abstraction level which reflects whether the change is generic or concrete, 2) the composition type which indicates whether the change is atomic or composite, 3) the type of operation the change represents (e.g., *Add*, *Delete*, and *Property_update* operations are simply examples of atomic changes, while *Move*, *Merge*, *Split*, *Replace*, and *Swap* operations are examples of composite changes), and 4) the scope of change which identifies the kind of software artifacts the change can be applied on (e.g., *Requirements*, *Architecture*, *Source Code*, *Documentation*, *Configuration Files*, and *Other Documents*). The authors demonstrated how the proposed change types can be used to create a set of impact rules to react to different types of changes and to propose possibly impacted elements which is a crucial task in activities such as impact analysis, change propagation, regression testing and refactoring. The basic key elements of such impact rules are the type of change, the type of the changed element, the type of a possibly impacted element, and the type of the dependency relation between the former two elements (i.e., the changed and a possibly impacted one). In [LR12], Lehnert and Riebisch merge the concept of impact rules with the concept of multi-perspective consistency checking [SF01] to propose a rule-based impact propagation approach of UML models and Java source code. The

authors claimed that considering the dependency relationship between artifacts of different views in the impact analysis 1) increases its ability to predict costs, 2) helps developers to better understand which artifacts require modification after a change, and 3) results in fewer inconsistencies. The authors are currently implementing the proposed approach in a prototype tool that extends the *EMFTrace* prototype CASE tool [BLR11], tool for automated traceability detection between models of a variety of modeling languages including UML.

6.2 A Comparative Evaluation of Change Propagation Approaches

To help in identifying the strengths and the weaknesses of current approaches as well as highlighting and discovering the most important aspects in performing change propagation processes, Ibrahim et al [IKD08] provide a comparative evaluation framework of change propagation approaches. Three sets of criteria are included in the proposed framework: the first one identifies the mechanism of the approach, its metrics and its level of automation; the second one identifies the properties of the artifacts under study including their type, their level of granularity, and the dependency relationships between them; the last one presents the type of other support that is needed for implementing change propagation such as change notification, consistency checker, log history or versioning system. Based on the surveyed work included in this study, the authors concluded the following: 1) most of the work on change propagation is done to handle the changes made on the low level artifacts such as source code artifacts since they are the more specific and concrete ones, 2) maintaining consistent traceability and dependency links is the core of the change propagation problem, 3) there is a need for proficient mechanisms to preserve the consistent relationships between the system artifacts after changes have been performed, and 4) there is a need for automated (or semi-automated) change propagation strategies to help in reducing human intervention errors. The authors also identify the use of software design rational and historical co-change information in generating more effective change propagation strategies at the code level, the use of consistency rules change propagation at the design level and the use of change propagation probability matrix for architecture components to evaluate the design quality features such as maintainability, extensibility and reusability.

6.3 Reviews of Some Existing Change Propagation Approaches

Three categories of approaches are recognized to automate the change propagation process:

- The first category is making use of consistency maintenance techniques for the intra-model change propagation;
- The second category is using model transformation techniques for both intra-model and inter-model change propagation.
- The third category includes techniques which use some other strategies for realizing the change propagation task.

In the following sub-sections we describe some approaches from these three categories which are summarized in Table 5.

6.3.1 Change Propagation Using Inconsistency Handling Methods

Dam and Winikoff [DW10] present an approach for supporting change propagation in UML design models by fixing the intra-model inconsistencies arise when a new change is made in part of the model. Based on the idea of the belief–desire–intention (BDI) reasoning model used in agent-oriented designs, a repair plan generator is developed to automatically generate possible repair plans from OCL consistency rules. A cost calculation algorithm is implemented to evaluate the impact of each repair plan and to propose the plan with the minimum cost. A case study is used to measure the efficiency and the scalability of the proposed approach. The results were promising and showed good scalability ratio to larger models. The authors planned to implement the approach as an industry tool.

A very similar work to [DW10] is the one proposed by Egyed [Egy06, Egy07, Egy11] for fixing inconsistencies in UML models that is discussed in the consistency management section (Section 5.5). The difference between the two work is that Egyed's repair plans are generated based on a dynamic analysis of the OCL consistency rules during their execution using a model profiling and it only provides initial positions for fixing the inconsistency as well as it does not consider the creation of new model elements in the fixing plans.

Also the work by Briand et al [BLOS06] on change impact analysis for UML models fits into the same category of approaches by defining specific change propagation rules for some set of change types which will be used to propose only the starting location for propagating the change.

Examples of other approaches that can be used for change propagation are found in the consistency management section.

One major problem of approaches that depend on resolving the inconsistencies resulting from a change as a possible way to carry out the required additional changes is that 1) not all changes result in inconsistency, 2) not all inconsistencies need to be resolved on time, and 3) not all types of inconsistencies can be resolved by current consistency management approaches.

6.3.2 Change Propagation Using Model Transformation

In the context of model-driven development where model transformation is used to create some target models from a set of source models, it is intuitive that managing the evolution and the synchronization of these models is attained using automated model transformations. Multiple model transformation techniques have been designed to support change propagation in evolving models. Following are examples for such approaches.

6.3.2.1 Batch Transformation Approaches

In [Jim05], Jimenez propose an off-line (or batch) transformation that only supports re-running the entire transformation on the new set of the source models (i.e., the source models after the

change) and then merging the resulting target models with the previous target models to produce the new target ones. Employing a merge strategy is very important step in this kind of transformation to ensure that updates made on the target models prior to the transformation are not lost, however it depends on the tracing information generated by the transformation language that links source model elements with their correspondences in the target model which is not usually feasible. Another problem which may arise when using stateless transformation is the execution time especially for large models or complex transformations. Moreover, the original transformation context may be lost. The approach has been developed to provide support for the logic-based EMF transformation engine, *Tefkat*.

6.3.2.2 Live (or Incremental) Transformation Approaches

Although batch transformation is the most frequently used approach, Johann and Egyed [JE04] observe that incremental model transformation has better scalability and usability as well as small cost (in terms of its execution time) compared to batch transformation based on a case study they run to incrementally update the transformation of UML design models to a sub-set of a domain-specific modeling language for embedded systems called ESCM. The authors developed two algorithms to implement their incremental transformation technique: *ShouldExist* and *DoesExist* which are used to determine which elements in the target model that are corresponding to the new changes in the source model need to be created, deleted or modified. They also addressed the challenges of developing incremental transformation approaches especially to propagate the changes across models with different syntax and semantics.

Live transformation approaches are also designed to overcome the problems of the stateless transformation by continuously maintaining the transformation context. The advantage of this is that it keeps the original transformation context resulting from the original transformation, avoiding the need for a merge strategy. This approach is more efficient for propagating small changes which do not actually need to re-run the entire transformation. The approach can also play an essential role to maintain the synchronization and thus the consistency of both source and target models in cases where models are frequently evolving (e.g., a typical case of most iterative and incremental development processes).

The idea of live transformation has been presented by Hearnden et al in [HLR06] and by Rath et al in [RBÖV08] and [RVV09].

In [HLR06], the proposed incremental updates approach is built in the context of the *Tefkat* open source tool. The underlying model transformation engine of this tool is logic-based inference engine and therefore changes on the source models are mapped to effects (i.e., logic-based rules) on the transformation execution which can be propagated to updates on the target models. This approach provides support for only declarative transformations.

On the other hand, in [RBÖV08] and [RVV09], the proposed approaches have been implemented in the context of the *VIATRA2* model transformation framework. The underlying transformation

engine of the *VIATRA2* framework is based on graph transformation and abstract state machines, thus the live transformation techniques built on it are mainly based on graph pattern matching where complex model changes may trigger execution of transformations. What distinguishes the approach of [RVV09] from the approach of [RBÖV08] is that the former adopts the concept of a *change history model* to record and describe all elementary changes on the source models which can be transformed at any time to another *change history model* for the target models which is then used to incrementally update the target models. In contrast with the logic-based approach presented in [HLR06], the latter two approaches provide support for both declarative and imperative transformations.

6.3.2.3 Bidirectional transformation Approaches

Bidirectional transformation approaches are concerned with the synchronization of both the source and the target models whenever a modification is carried out on either one. This is different from the former two types of transformation mentioned above where the synchronization process is triggered only by changes in the source models, which is known as unidirectional transformations.

An example of a bidirectional model transformation language is the Object Management Group (OMG)'s Query View Transformation-Relations (QVT-Relations) language. The Janus Transformation Language (JTL) is another example of bidirectional declarative transformation language that is developed by Cicchetti et al [CREP11]. One extra feature of JTL over QVT-Relations is that the former supports non-bijective transformations where a source model can be mapped into a set of target models. The implementation of the JTL language depends on the Answer Set Programming (ASP) language, a declarative problem solving logic-based programming language, to specify the bidirectional transformation and on the DLV ASP solver to execute the transformation. In such a case, 1) the metamodels of the source and the target models and the source models are encoded as ASP representations, 2) the JTL program is translated into an ASP program supported with additional ASP constraints which are used to manage the trace links between the source and the target models elements, 3) the transformation is executed and finally 4) the target model is generated. The proposed architecture for the JTL transformation language has been implemented as a set of Eclipse plug-ins and has been demonstrated on a bidirectional transformation of hierarchal state machines into flat state machines.

One attempt to implement a bidirectional model transformation using the Atlas Transformation Language (ATL) is presented by Xiong et al in [XLHZTM07]. The authors managed to extend the byte-code of the language virtual machine to support the incremental backward propagation of modifications. A batch transformation policy is taken for the forward propagation of changes from the source models to the target models by simply re-executing the entire transformation whenever new changes are made in the source side. Although the ideas presented in this

approach seem to be promising, it does not yet support the backward propagation of the insertion type of changes on the target models.

6.3.2.4 Incremental Bidirectional Model Synchronization Approach

Giese and Wagner [GW07] introduce an approach for incremental bidirectional model synchronization using Triple Graph Grammars (TGGs), a graphical and declarative methodology for describing the correspondence between different types of models in terms of a set of TGG rules composing a correspondence model. Based on the execution of these rules and the correspondence model as well as a change notification mechanism that reports when a source or a target model element has been modified, a model transformation engine is developed to synchronize the changes of a source model to a target model (and vice versa) by applying the following steps: 1) execute the rules that are triggered by the change and check if the generated pattern in the target model (or in the source model depending on transformation direction) is still consistent, if it is consistent, 2) check if the attribute conditions of the correspondence nodes are still valid and if not, propagate the attribute value change, and finally 3) undo the rule execution if step 1 fails. The proposed approach is realized in the open source CASE tool, Fujaba Tool Suite.

6.3.2.5 Concurrent Model Synchronization Approach

Xiong et al [XSHT11] propose a state-based algorithm for concurrent model synchronization based on the idea presented in [XLHZTM07] which takes as parameters the two original models (a source model and its corresponding target model) and the two updated models (a modified source model and a modified target model), and returns a new set of synchronized models. Three main components make up the new approach: 1) a model difference approach to identify the updates made on the two original models and determine whether the updates are conflicting or not, 2) a three-way merger operator to create a new set of models where the different updates on both sides are merged, and 3) a preservation test procedure to check that the updates made on the original target model are preserved in the new generated target model. The four steps of the approach are as follows: 1) a backward transformation of the updated target model is executed, 2) a model difference approach and a three-way merger operator are applied on the original source model, the updated source model and the transformed updated target model to produce a new updated source model, 3) a forward transformation is executed to transform back the new source model resulting from step 3 to create a new updated target model, and finally test preservation checking is applied on the new target model. The authors showed that their concurrent synchronization approach satisfies properties such as consistency, stability and preservation which are the prerequisites for basic bidirectional transformations.

Hermann et al [HEEO12] develop a semi-automatic conflict resolution strategy to propose a delta-based approach for managing simultaneous updates which are made on two sets of inter-related models, belonging to the same domain, at the same time (a delta-based means that the key elements of the approach are the updates made on the models and not the entire models as the

case for the state-based approaches). The ideas presented in this work are based on the authors' previous work on a formal resolution strategy for operation-based conflicts in model versioning and bidirectional model synchronization using Triple Graph Grammars (TGGs). An adapted merge operation is defined for conflict resolution of simultaneous changes that are not conflict-free. The selected resolution strategy is chosen to prioritize the insertion changes over the deletion ones. In this case, the final resolution step is left to the modeler decision. Accordingly, a concurrent model synchronization algorithm is constructed with a conflict resolution plan using the following steps: 1) a consistency checking operation is executed on the source model to ensure the model consistency after the modifications that are made on the source side, 2) a forward propagation operation is executed to propagate the updates from the source model to the target model, 3) a conflict resolution operation is executed (with the possibility of manual modification) on the two sets of updates (the updates on the target model and the updates which are propagated from the source side), 4) a consistency checking operation is executed on the target model to ensure the model consistency after the derived modifications resulting from step 4, and finally 5) a backward propagation operation is executed to propagate the derived modifications from the target model to the source model. The authors proved the correctness and the compatibility of their proposed approach with respect to the formalization of the basic TGG-based bidirectional model transformations.

In conclusion, change propagation using model transformation is still an open topic. Recently, Egyed et al [EDGLMNR11] address the need for a smart assistant to support this task and discussed some basic requirements which are summarized as follows: 1) change propagation techniques cannot be fully automated; users' intervention should play a very important role in such process, 2) change propagation cannot be solved with unidirectional transformation; bidirectional transformation techniques should be employed, and 3) ensuring the consistency of models after the change should be supported by appropriate consistency checking technologies.

6.3.3 Change Propagation Using Other Approaches

6.3.3.1 Knowledge-Based Approach for Impact Analysis and Change Propagation

Ajila [Aji95] presents a generic approach for impact analysis and change propagation that can be applied in any development environment and on any language or design method. The idea is built on extracting the relationships between the system entities within and across the different development phases and feeding such relationships into a knowledge base which can be queried about changes that can be made on the system entities. The result from the query is a report of all impacted entities and the steps required to propagate the changes. Tool support is implemented to determine the impact of a change between the Hierarchical Object-Oriented Design model and its corresponding ADA implementation model.

6.3.3.2 Relationship-Based Change Propagation Approach

Chechik et al [CLNCDESS09] present an automated algorithm for propagating changes between the requirement model (represented by the UML activity diagram) and the design model

(represented by the UML sequence diagram). Given the old version of an activity diagram and its corresponding sequence diagram and the new version of an activity diagram, the proposed algorithm would be able to 1) capture the relationships between these diagrams, 2) identify the added and the removed elements to the old activity diagram, and consequently 3) map such changes to their corresponding ones to create the new version of the sequence diagram. For regions where the algorithm failed to map, an unknown is inserted for manual inspection. The technique is implemented in a prototype tool and is evaluated in a case study.

6.3.3.3 Model Dependencies Approach Using Formal Concept Analysis (FCA)

Ivkovic et al in [IK06, Ivk11] provide a methodology to identify and formally define model dependencies (in terms of association rules) using Triple Graph Grammar (TGG) which are needed to propagate changes across system models at different levels of abstraction, for example from system design to system source code. In addition, they introduced an approach that is based on a method for conceptual knowledge representation and data analysis called Formal Concept Analysis (FCA) to extract clusters of model elements that are related by a dependency relation which can be used for model synchronization. The proposed technique is implemented in a framework for incremental change propagation in the context of Model Driven Architecture (MDA) that is named *mSynTra* to synchronize business process models with their underlying Java source code models.

6.4 General Observations

Change propagation is considered to be a critical and complex step of change management throughout the software life cycle. Having effective and reliable techniques to handle this task is still a crucial need. Although we could recognize some research work on realizing the change propagation task in the intra-model and the inter-model levels for the design model and between the design model and its interrelated models, the research in this area is not fully fledged yet. On the other hand, a great effort of the research community is expended on devising more effective techniques for change propagation at the code level artifacts (i.e., within the implementation model only) [IKD08]. A summary of the surveyed work is listed in Table 6.

Table 6: A Summary of the Surveyed Work on Change Propagation (CP) & Model Synchronization (MS)

#		Reference(s)	Change Propagation Level	Task(s) Achieved	Model Type	Technique(s) Used	Source Model	Target Model	Tool Support Yes/No [- Details (if applicable)]
1	Inconsistencies Handling	[DW10]	Intra-Model	Incremental CP	Design Models	Inconsistency Resolution Plans	Class Diagram Sequence Diagram State Machine Diagram	Class Diagram Sequence Diagram State Machine Diagram	Yes - Within Prometheus Design Tool
2	Model Transformations	[Jim05]	Inter-Model	Batch CP	Design Models of Different Domains	Merging Approach	Class Diagram	Database ER Diagram	Yes
3		[JE04]	Inter-Model	Incremental CP	Design Models of Different Domains	*ShouldExist* and *DoesExist* Algorithms	UML Design Models	Domain-Specific Models (ESCM)	Yes
4		[HLR06]	Inter-Model	Incremental CP	Design Models of Different Domains	Logic-based SLD Resolution	Class Diagram	Database ER Diagram	Yes - Within Tefkat Tool
5		[RBÖV08, RVV09]	Inter-Model	Incremental CP	Design Models of Different Domains	Graph Pattern Matching			Yes - Within VIATRA2 Framework
6		[CREP11]	Intra-Model	Bidirectional MS	Design Models at Different Level of Abstraction	Janus Transformation Language (JTL)	Hierarchical State Machine	None-hierarchical State Machine	Yes - JTL
7		[XLHZTM07]	Inter-Model	Bidirectional MS	Design and Implementation Models	Extending the ATL (Atlas Transformation Language) Virtual Machine & Model Differencing & Merging	Class Diagram	Java Code	Yes
8		[GW07]	Inter-Model	Incremental Bidirectional MS	Design Models of Different Domains	Triple Graph Grammars (Attribute Update Propagation & Correspondence Dependency)	SDL (Specification and Description Language) Block Diagram	Class Diagram	Yes - Fujaba Plug-in
9		[XSHT11]	Inter-Model	Concurrent MS	Design Models of Different Domains	Model Difference & Three-way Merger	Class Diagram	Database ER Diagram	Yes - Within the ArchStudio Tool
10		[HEEO12]	Intra-Model	Concurrent MS	Multiple views of Class Diagram	Triple Graph Grammars & Conflict Resolution Strategy	One view of a class diagram	Another view of the same class diagram	No
11	Others	[Aji95]	Inter-Model	CP	Design and Implementation Models	Knowledge Based Approach	Any Design Models	Their Related Implementation Models	Yes
12		[CLNCDESS09]	Inter-Model	CP	Requirement and Design Models	Relations-based Approach	Activity Diagram	Sequence Diagram	Yes
13		[IK06, Ivk11]	Inter-Model	CP	Design and Implementation Models	Model Dependencies using Formal Concept Analysis (FCA)	Business Process Models	Java Code	Yes - mSynTra

7. COMMON TASKS OF MODEL EVOLUTION – UNCERTAINTY MANAGEMENT IN UML MODELS

Different definitions of uncertainty can be found in the literature depending on the field where it is used. For example, in economics it expresses the doubts about the occurrence of an event or the outcome of a decision, in physics it refers to the lack of precise and accurate way of measurements, in psychology and human behavior it refers to the lack of sufficient and reliable information to make decisions while in software engineering it refers to incomplete or inconsistent information about the system requirements which may lead to false assumptions when developing the system architecture and the system design. Unfortunately traditional software engineering design does not adequately address situations where there is uncertainty. Also in the area of model-driven software engineering, dealing with modeling in the presence of uncertainty due to incomplete or/and inconsistent specification and reasoning about models which have uncertainty are still considered to be challenging [SMB09]. In this paper, we describe the key findings from the survey that we have conducted in exploring the research work achieved to handle uncertainty in UML design models.

7.1 Uncertainty in Software Engineering

Many studies in the literature tried to provide a better understanding of uncertainty in software engineering in general and in software system design in particular. They discuss the sources and nature of uncertainty, classify the types of uncertainties that are often encountered and also classify the methods and techniques mostly used for modeling these uncertainties.

Ziv et al [ZRK97] claim that the complexity of software systems and their development processes is known to be fundamental due to the presence of uncertainty in every aspect of software development. They present four domains of software engineering where uncertainty is extremely evident, including the uncertainty in requirements analysis, the uncertainty in the transition from system requirements to system design and implementation, the uncertainty in software re-engineering and lastly the uncertainty in software reuse. Three sources of uncertainty in software engineering are mentioned: uncertainty in the problem domain, uncertainty in the solution domain, and uncertainty due to human participation. The authors emphasized the importance of having proper notations and formalisms as well as effective techniques for modeling and quantitatively capturing the uncertainty in the software development processes in order to alleviate their complexity. For further exploration of the applicability of their observation, Ziv et al present a technique for modeling uncertainty in software testing based on the Bayesian belief networks, a probabilistic model for reasoning about uncertainty.

A recent study carried out by Ramirez et al [RJC12] provide a taxonomy for potential sources of uncertainty at the requirements level, at the design level and at the run-time level for dynamically adaptive systems. Examples of such sources are ambiguous or incomplete requirements,

changing requirements, unexplored alternatives, inadequate design, unverified design, latent behavior, and inconsistency. The study also identifies the state-of-the-art techniques used for mitigating (or alleviating) specific types of uncertainty occurring at each level. The majority of these techniques are based on either probabilistic or fuzzy logic models. It is also noted from this study that uncertainty in the requirements and run-time levels has been thoroughly researches in the context of adaptive systems.

Additionally, De Weck and Eckert in [DE07] recognize the fact that complex software systems evolve over years either to meet new requirements or to cope with new technologies. To maximize the reusability of software components during their life span, they should be designed in such a way that promotes the incorporation of future changes (such design method is known as "*Design for Flexibility*" or "*Design for Changeability*"). Such changes are not absolute, however they are limited to the kinds of uncertainty that the system is subject to. Such uncertainties can have a significant impact on the proper functioning of the system and so they have to be projected into (i.e. explicitly modeled in) the system architecture and the system design to identify the software and/or the hardware components that are most likely to be changed in the future and to help in reasoning about such uncertainty at any time. On their study, they found out that formal approaches to uncertainty modeling rely mainly on methods which are solidly rooted in probability theory and logic. These methods are often inaccessible and even obscure to system designers looking for incorporating future uncertainty into their design work. Other less formal and more practical approaches to uncertainty modeling depend on scenario planning which is based on defining a finite set of future scenarios that capture the range of future uncertain changes that might occur. One example of these approaches is the work on partial behavior models which is applied in the *Modal Transition Systems* (MTSs) to distinguish between required, prohibited, or unknown behaviors. Other partial modeling formalisms include the *Partial Labeled Transition Systems*, the *multi-valued state machines*, and the *mixed transition systems*. Inspired by this idea of partial behavior models, Famelis et al [FBCS11] propose a language-independent methodology to use partial models as a first-class development artifact to represent uncertainty in model-based software development.

Xiao et al [XPPAB08] identify four types of uncertainty and imprecision in process modeling associated with the specification of the following elements: *Role*, *Activity*, *Deliverable*, and *Iteration*. The authors chose to benefit from the extension mechanism of the UML modeling language to model the four types of uncertainty and imprecision. They propose a new UML profile to represent the elements which have uncertainty or imprecision and to represent the methods used to analyze different types of uncertainty. The authors neither implemented their proposed approach nor evaluated it.

The idea of managing the uncertainty in software engineering is presented by Ibrahim et al in [IEFD09]. They propose a framework of four main phases including:

- *Identification and Prioritization* phase where the different sources and types of uncertainties associated with the development process are identified and ranked;
- *Modeling and Analysis* phase where the selected uncertainties are modeled and analyzed to identify their influence;
- *Management and Planning* phase where plans for handling the effects of the modeled uncertainties are produced;
- *Monitoring and Evaluation* phase where assessments of the created management plans are performed to confirm their effectiveness.

7.2 Uncertainty Management with Partial Models

In this section, we present an overview of the work achieved so far in partial models.

Partial models provide the developers with a systematic way to represent uncertainty they usually encounter at the early stages of the development process and through the system evolution when they have a set of alternative scenarios to build the system models and they do not have the reliable information to make proper design decisions. Having the methodology to model such uncertainty allows the developers to keep all possible design alternatives with the capability to narrow them down when more information become available. In this case a partial model results from merging the set of all possible alternative models the developers have and denoting the model elements which have some conflict (or some sort of uncertainty) with certain annotations that represent the degree of knowledge available about these elements. Resolving these uncertainties (i.e., partiality refinement) will create a set of concrete models which are considered to be the "concretizations" of the partial model. To guarantee the consistency of these concretizations, partiality refinement is made in such a way that preserves certain properties governed by the well-formedness rules of the modeling language as well as the consistency rules.

7.2.1 "May" and "MAVO" Partialities

In [FBCS11], only "May" partiality was proposed and demonstrated its usage on the state machine diagrams where a "*May*" annotation is used to denote a model element (e.g., state or transition) that the developer is not sure of its presence, whether it must or may exist. In [SFC12], the "*MAVO*" partiality is introduced as an extension to the "*May*" partiality which includes the "*Abs*", the "*Var*", and the "*OW*" partialities allowing the developers to express, respectively, the uncertainty about the number of elements in the model (whether it is a single or a collection), the distinctness of individual elements (whether it is a constant or a variable), and the completeness of the entire model (whether it is complete or incomplete). The authors provide formal semantics of the "MAVO" partiality and also demonstrated how to apply it to representing the different types of uncertainty that may occur in design models such as class diagrams and sequence diagrams. In addition, they used the "MAVO" partiality in expressing the uncertainty in Requirements Engineering (RE) models such as i* (a modeling language that is used to model the early requirements of the system in terms of *Actors*, *Goals*, *Softgoal*, *Tasks*,

and *Resources*) [SCH12]. In this work, reducing the uncertainty in the models is achieved using refinement rationales the system developers will determine based on further exploration of the system requirements. To check the validity of the refinement actions made to resolve the "MAVO" partiality, the authors developed an algorithm and a prototype tool support on top of the Alloy SAT solver [SCG12]. In this case both the partial and the refined models are translated to their First Order Logic representations. Since the notion of requirements traceability is fundamental in requirements engineering, the authors also explained how to establish traceability relation between refined and refining artifacts containing uncertainty.

7.2.2 Reasoning with Uncertainty in Partial Models

One important feature of partial models is that they enable developers to reason about and check for certain properties in the presence of uncertainty [FSC12-1]. The underlying approach consists of the following steps: 1) creating the set of all possible alternatives (concretizations) of the system model, 2) constructing the partial model results from merging all these alternatives, 3) checking whether the partial model satisfies a given property by encoding both the model and the property in propositional logic, 4) using a SAT solver to find out which alternatives violate the given property, and finally 5) excluding the alternatives (concretizations) that violate the property.

To complement the work in [FSC12-2], Saadatpanah et al [SFGRCS12] conduct an empirical study to evaluate the effectiveness of some reasoning formalisms to encode "MAVO" partial models including Alloy, Constraint Satisfaction Problems (CSP), Satisfiability Modulo Theory (SMT), and Answer Set Programming (ASP). Among the four reasoning formalisms, the SMT was shown to be the most efficient formalism as it scales better with large models. The authors also are going to examine another reasoning formalism, the Binary Decision Diagrams (BDDs).

7.2.3 Transforming Partial Models

Model transformations play a critical role in model-driven development (MDD) where they are heavily used to refine models, to generate new views from already existing ones, to reverse engineer models, and to refactor models. Current model transformation languages are working on concrete models. The feasibility of adapting the semantics of classical model transformations to partial models is investigated by Famelis et al [FSC12-2] where a logic-based approach for defining new semantics for model transformations is proposed. In this work, input models are expressed in propositional logic, transformation rules are expressed as transfer predicates, and a SAT solver is used to check the correctness of the transformation based on a correctness criterion defined by the authors which states that applying a transformation to a partial model should work as if we applied the transformation to each individual concretization of the partial model and then take the output of these individual transformations to form the output partial model. The proposed approach is not fully automated yet where transfer predicates of transformation rules need to be constructed manually.

7.3 General Observations

To the best of our knowledge, the work of partial models mentioned above is the only current research project that investigates the practicality of this topic. Although the work achieved is still in a preliminary state but it is a promising and the authors have clear plans for future work.

8. Conclusion

In this paper we gave an overview of model evolution in the context of model-driven development.

We presented the different types of model evolution found in the literature and discussed the role of model transformation in automating these different types of evolutions. We chose one type of model evolution, model refactoring, and reviewed the current state-of-the-art approaches to automate and perform model refactoring. We also discussed the main challenges the research community still faces that prevent them to reach to fully satisfied and effective solutions.

For supporting model evolution process, we have selected a number of tasks to consider. We surveyed the literature and were able to identify the most common approaches used and the challenges (or the open problems) in each task.

For change impact analysis, we argued that this task is most heavily used on the code level and not on the model level. However, we found few work that develop approaches for the intra-model change impact analysis on the design level as well as the inter-model change impact analysis between the design and the test models for the purpose of regression testing.

For the consistency management, we found that the work conducted in this area cover the design models only. As a result, we first discussed the different types of inconsistencies that can occur in design models during their evolution and then we presented a number of approaches used to detect the different types of inconsistencies and classified them to four categories.

We also provided an overview of the different levels of change propagation required to ensure the completeness and the consistency of an evolution step; this comprises the intra-model and the inter-model between the design model and its related models including the implementation model and the models that are generated from the design model for the purpose of testing, analyzing and verifying the system model. We also recognized the major role of model transformation techniques in automating this process especially to perform the inter-level (or vertical) change propagation.

Finally, we discussed the concept of uncertainty management during model evolution and described the current state-of-the-art in approaches that tackle this problem.

REFERENCES

-Introduction-

[MoVE] Model Versioning and Evolution. http://move.q-e.at/, last accessed: November 19, 2012.

[MS05] T. Mens and J. Simmonds, "A Framework for Managing Consistency of Evolving UML Models," *Software Evolution with UML and XML*, pp. 1, 2005.

[SMB09] R. Van Der Straeten, T. Mens and S. Van Baelen, "Challenges in Model-Driven software engineering," *Models in Software Engineering*, pp. 35-47, 2009.

[Sun11] Y. Sun, *Model Transformation by Demonstration: A User-Centric Approach to Support Model Evolution*, 2011.

[Swa76] E. Swanson, "The dimensions of maintenance." *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976.

-Introducing Model Evolution-

[AA09] S. A. Ajila and S. Alam, "Using a formal language constructs for software model evolution," in *Semantic Computing, 2009. ICSC'09. IEEE International Conference on*, 2009, pp. 390-395.

[Bie10] M. Biehl, *Supporting Model Evolution in Model-Driven Development of Automotive Embedded Systems.* Skolan för industriell teknik och management, Kungliga Tekniska högskolan, 2010.

[BLSWWKRS09] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger and W. Schwinger, "An example is worth a thousand words: Composite operation modeling by-example," *Model Driven Engineering Languages and Systems*, pp. 271-285, 2009.

[DVW07] A. Van Deursen, E. Visser and J. Warmer, "Model-driven software evolution: A research agenda," in *CSMR Workshop on Model-Driven Software Evolution (MoDSE'07)*, 2007, pp. 41-49.

[GLZ06] J. Gray, Y. Lin and J. Zhang, "Automating change evolution in model-driven engineering," *Computer*, vol. 39, pp. 51-58, 2006.

[LWC07] C. F. J. Lange, M. A. M. Wijns and M. R. V. Chaudron, "MetricViewEvolution: UML-based views for monitoring model evolution and quality," in *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on*, 2007, pp. 327-328.

[LRSS11] T. Levendovszky, B. Rumpe, B. Schätz and J. Sprinkle, "Model Evolution

and Management," *Model-Based Engineering of Embedded Real-Time Systems*, pp. 241-270, 2011.

[MD00] T. Mens and T. D'Hondt, "Automating support for software evolution in UML," *Autom. Software. Eng*, vol. 7, pp. 39-59, 2000.

[RM10] A. A. Rao and K. Madhavi, "Framework for Visualizing Model-Driven Software Evolution and its Application," *ArXiv Preprint arXiv:1002.1188*, 2010.

[SGW11] Y. Sun, J. Gray and J. White, "MT-scribe: An end-user approach to automate software model evolution," in *Software Engineering (ICSE), 2011 33rd International Conference on*, 2011, pp. 980-982.

[*SMB09*] R. Van Der Straeten, T. Mens and S. Van Baelen, "Challenges in Model-Driven software engineering," *Models in Software Engineering*, pp. 35-47, 2009.

[SWG09] Y. Sun, J. White and J. Gray, "Model transformation by demonstration," *Model Driven Engineering Languages and Systems*, pp. 712-726, 2009.

-Example: Model Refactoring-

[AMST09] T. Arendt, F. Mantz, L. Schneider and G. Taentzer, "Model refactoring in eclipse by LTK, EWL, and EMF refactor: A case study," in *Model-Driven Software Evolution, Workshop Models and Evolution*, 2009, .

[AT12] T. Arendt and G. Taentzer, "Integration of smells and refactorings within the eclipse modeling framework," in *Proceedings of the Fifth Workshop on Refactoring Tools*, 2012, pp. 8-15.

[Ast02] D. Astels and others, "Refactoring with UML," in Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering, 2002, pp. 67-70.

[BM07] T. Baar and S. Marković, "A graphical approach to prove the semantic preservation of UML/OCL refactoring rules," *Perspectives of Systems Informatics*, pp. 70-83, 2007.

[BEKKTW07] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer and E. Weiss, "EMF model refactoring based on graph transformation concepts," *Electronic Communications of the EASST*, vol. 3, 2007.

[BSF03] M. Boger, T. Sturm and P. Fragemann, "Refactoring browser for UML," *Objects, Components, Architectures, Services, and Applications for a Networked World*, pp. 366-377, 2003.

[CW04] A. Correa and C. Werner, "Applying refactoring techniques to uml/ocl models," *in 2004-the Unified Modeling Language.Modelling Languages and Applications*, pp. 173-187, 2004.

[DK06] Ł. Dobrzański and L. Kuźniarz, "An approach to refactoring of executable UML models," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006, pp. 1273-1279.

[EN12] H. Einarsson and H. Neukirchen, "An approach and tool for synchronous refactoring of UML diagrams and models using model-to-model transformations," in *Proceedings of the Fifth Workshop on Refactoring Tools*, 2012, pp. 16-23.

[EME10] M. El-Sharqwi, H. Mahdi and I. El-Madah, "Pattern-based model refactoring," in *Computer Engineering and Systems (ICCES), 2010 International Conference on*, 2010, pp. 301-306.

[Enc09] T. Enckevort, "Refactoring UML models: Using open architecture ware to measure UML model quality and perform pattern matching on UML models with OCL queries," in *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, 2009, pp. 635-646.

[Fol07] A. Folli, UML Model Refactoring using Graph Transformation, 2007.

[FM08] A. Folli and T. Mens, "Refactoring of UML models using AGG," *Electronic Communications of the EASST*, vol. 8, 2008.

[GSMD03] P. Van Gorp, H. Stenten, T. Mens and S. Demeyer, "Towards automating source-consistent UML refactorings," *«UML» 2003-the Unified Modeling Language.Modeling Languages and Applications*, pp. 144-158, 2003.

[GSMD03] P. Van Gorp, H. Stenten, T. Mens and S. Demeyer, "Enabling and using the UML for model driven refactoring," *Proceedings WOOR*, vol. 3, pp. 37-40, 2003.

[HA08] S. Hosseini and M. A. Azgomi, "UML model refactoring with emphasis on behavior preservation," in *Theoretical Aspects of Software Engineering, 2008. TASE'08. 2nd IFIP/IEEE International Symposium on*, 2008, pp. 125-128.

[KB02] H. Kim and C. Boldyreff, "Developing software metrics applicable to UML models," in *6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2002.

[KCKB05] M. Van Kempen, M. Chaudron, D. Kourie and A. Boake, "Towards proving preservation of behaviour of refactoring of UML models," in *Proceedings of the 2005 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*, 2005, pp. 252-259.

[Lan07] C. Lange, "Assessing and Improving the Quality of Modeling," *Technische,*

Universiteit Eindhoven, 2007.

[MB05] S. Marković and T. Baar, "Refactoring OCL annotated UML class diagrams," *Model Driven Engineering Languages and Systems*, pp. 280-294, 2005.

[MEDJ05] T. Mens, N. Van Eetvelde, S. Demeyer and D. Janssens, "Formalizing refactorings with graph transformations," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, pp. 247-276, 2005.

[MMBJ09] N. Moha, V. Mahé, O. Barais and J. M. Jézéquel, "Generic model refactorings," *Model Driven Engineering Languages and Systems*, pp. 628-643, 2009.

[MRG09] M. Mohamed, M. Romdhani and K. Ghedira, "Classification of model refactoring approaches," *Journal of Object Technology*, vol. 8, 2009.

[MTM07] T. Mens, G. Taentzer and D. Müller, "Challenges in model refactoring," in *Proc. 1st Workshop on Refactoring Tools, University of Berlin*, 2007, .

[Por05] I. Porres, "Rule-based update transformations and their application to model refactorings," *Software and Systems Modeling*, vol. 4, pp. 368-385, 2005.

[Por03] I. Porres, "Model refactorings as rule-based update transformations," «UML» 2003-the Unified Modeling Language.Modeling Languages and Applications, pp. 159-174, 2003.

[RLKEB08] G. Rangel, L. Lambers, B. König, H. Ehrig and P. Baldan, "Behavior preservation in model refactoring using DPO transformations with borrowed contexts," *Graph Transformations*, pp. 242-256, 2008.

[RSA12] J. Reimann, M. Seifert and U. Aßmann, "On the reuse and recommendation of model refactoring specifications," *Software and Systems Modeling*, pp. 1-18, 2012.

[RSA10] J. Reimann, M. Seifert and U. Aßmann, "Role-based generic model refactoring," *Model Driven Engineering Languages and Systems*, pp. 78-92, 2010.

[SPTJ01] G. Sunyé, D. Pollet, Y. Le Traon and J. M. Jézéquel, "Refactoring UML models," «UML» 2001—The Unified Modeling Language.Modeling Languages, Concepts, and Tools, pp. 134-148, 2001.

[SJM07] R. Van Der Straeten, V. Jonckers and T. Mens, "A formal approach to model refactoring and model refinement," *Software and Systems Modeling*, vol. 6, pp. 139-162, 2007.

[Wüs11] J. Wüst. SDMetrics. http://sdmetrics.com/, last accessed: November 19, 2012, 2011.

[ZLG05] J. Zhang, Y. Lin and J. Gray, "Generic and domain-specific model refactoring using a model transformation engine," *Model-Driven Software Development*,

pp. 199-218, 2005.

-Common Tasks for Model Evolution – Change Impact Analysis of UML Models-

[BA96] S. A. Bohner and R. S. Arnold, “Software Change Impact Analysis,” in *IEEE Computer Society Publications Tutorial Series*, 1996.

[BLH09] L. C. Briand, Y. Labiche and S. He, "Automating regression test selection based on UML designs," *Information and Software Technology,* vol. 51, pp. 16-30, 2009.

[BLOS06] L. C. Briand, Y. Labiche, L. O’Sullivan and M. M. Sówka, "Automated impact analysis of UML models," *J. Syst. Software,* vol. 79, pp. 339-352, 2006.

[BLO03] L. C. Briand, Y. Labiche and L. O'sullivan, "Impact analysis and change management of UML models," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on,* 2003, pp. 256-265.

[BLS02] L. C. Briand, Y. Labiche and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," in *Software Maintenance, 2002. Proceedings. International Conference on,* 2002, pp. 252-261.

[CPU07] Y. Chen, R. L. Probert and H. Ural, "Model-based regression test suite generation using dependence analysis," in *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing,* 2007, pp. 54-62.

[DMW05] C. R. Dantas, L. G. P. Murta and C. M. L. Werner, "Consistent evolution of UML models by automatic detection of change traces," in *Principles of Software Evolution, Eighth International Workshop on,* 2005, pp. 144-147.

[DMW07] C. Dantas, L. Murta and C. Werner, "Mining change traces from versioned UML repositories," in *Proceedings of the Brazilian Symposium on Software Engineering (SBES'07),* 2007, pp. 236-252.

[FB10] E. Fourneret and F. Bouquet, "Impact analysis for uml/ocl statechart diagrams based on dependence algorithms for evolving critical software," *Laboratoire d’Informatique De Franche-Comté, Besançon, France, Tech.Rep.RT2010-06,* 2010.

[FBDD11] E. Fourneret, F. Bouquet, F. Dadeau and S. Debricon, "Selective test generation method for evolving critical systems," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on,* 2011, pp. 125-134.

[FIMR10] Q. Farooq, M. Iqbal, Z. I. Malik and M. Riebisch, "A model-based regression testing approach for evolving software systems with flexible tool support," in *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on,* 2010, pp. 41-49.

[IMN07] M. Z. Z. Iqbal, Z. I. Malik and A. Nadeem, "An approach for selective state machine based regression testing," in *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing,* 2007, pp. 44-52.

[Kil08] M. Kilpinen, *The Emergence of Change at the Systems Engineering and Software Design Interface: An Investigation of Impact Analysis,* 2008.

[Leh11-1] S. Lehnert, "A Review of Software Change Impact Analysis," *Techn. Univ. Ilmenau, Report ilm1-2011200618,* 2011.

[Leh11-2] S. Lehnert, "A taxonomy for software change impact analysis," in *Proceedings of the 12th International Workshop and the 7th Annual ERCIM Workshop on Principles on Software Evolution and Software Evolution,* 2011, pp. 41-50.

[LW89] H. K. N. Leung and L. White, "Insights into regression testing [software testing]," in *Software Maintenance, 1989., Proceedings., Conference on,* 1989, pp. 60-69.

[MODLW07] L. Murta, H. Oliveira, C. Dantas, L. G. Lopes and C. Werner, "Odyssey-SCM: An integrated software configuration management infrastructure for UML models," *Science of Computer Programming,* vol. 65, pp. 249-274, 2007.

[MTN11] N. Mansour, H. Takkoush and A. Nehme, "UML-based regression testing for OO software," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 23, pp. 51-68, 2011.

[NZR10] L. Naslavsky, H. Ziv and D. J. Richardson, "Mbsrt2: Model-based selective regression testing with traceability," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on,* 2010, pp. 89-98.

[PUA06] O. Pilskalns, G. Uyan and A. Andrews, "Regression testing uml designs," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on,* 2006, pp. 254-264.

-Common Tasks for Model Evolution – Consistency Management of UML Models-

[AMBB10] M. Almeida da Silva, A. Mougenot, X. Blanc and R. Bendraou, "Towards automated inconsistency handling in design models," in Advanced Information Systems Engineering, 2010, pp. 348-362.

[BMMM08] X. Blanc, I. Mounier, A. Mougenot and T. Mens, "Detecting model inconsistency through operation-based model construction," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on,* 2008, pp. 511-520.

[EB04] M. Elaasar and L. Briand, "An overview of UML consistency management," *Carleton University, Canada, Technical Report SCE-04-18,* 2004.

[EE95] J. Ebert and G. Engels, "Specialization of Object Life Cycle Definitions,"

Fachberichte Informatik 19 (1997): 95.

[Egy00] A. F. Egyed, “Heterogeneous View Integration and its Automation," *PhD diss., University of Southern California*, 2000.

[Egy06] A. Egyed, "Instant consistency checking for the UML," in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 381-390.

[Egy07] A. Egyed, "Fixing inconsistencies in UML design models," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 292-301.

[Egy07] A. Egyed, "UML/Analyzer: A tool for the instant consistency checking of UML models," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, 2007, pp. 793-796.

[Egy11] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *Software Engineering, IEEE Transactions on*, pp. 1-1, 2011.

[EHHS02] G. Engels, J. H. Hausmann, R. Heckel and S. Sauer, "Testing the consistency of dynamic UML diagrams," in *Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002)*, 2002.

[EHK01] G. Engels, R. Heckel and J. Küster, "Rule-based specification of behavioral consistency based on the UML meta-model," *«UML» 2001—The Unified Modeling Language.Modeling Languages, Concepts, and Tools*, pp. 272-286, 2001.

[EHK03] G. Engels, R. Heckel and J. Küster, "The consistency workbench: A tool for consistency management in UML-based development," *«UML» 2003-the Unified Modeling Language.Modeling Languages and Applications*, pp. 356-359, 2003.

[EHKG02] G. Engels, R. Heckel, J. Küster and L. Groenewegen, "Consistency-preserving model evolution through transformations," *«UML» 2002—The Unified Modeling Language*, pp. 212-227, 2002.

[EKHG01] G. Engels, J. M. Küster, R. Heckel and L. Groenewegen, "A methodology for specifying and analyzing consistency of object-oriented behavioral models," in *ACM SIGSOFT Software Engineering Notes*, 2001, pp. 186-195.

[EKHG02] G. Engels, J. M. Küster, R. Heckel and L. Groenewegen, "Towards consistency-preserving model evolution," in *Proceedings of the International Workshop on Principles of Software Evolution*, 2002, pp. 129-132.

[ELF08] A. Egyed, E. Letier and A. Finkelstein, "Generating and evaluating choices for fixing inconsistencies in UML design models," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on,*

2008, pp. 99-108.

[FST96] A. Finkelstein, G. Spanoudakis and D. Till, "Managing interference," in *Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints' 96) on SIGSOFT'96 Workshops*, 1996, pp. 172-174.

[GHM98] J. Grundy, J. Hosking and W. B. Mugridge, "Inconsistency management for multiple-view software development environments," *Software Engineering, IEEE Transactions on*, vol. 24, pp. 960-981, 1998.

[HHKT02] B. Hnatkowska, Z. Huzar, L. Kuzniarz and L. Tuzinkiewicz, "A systematic approach to consistency within UML based software development process," *Blekinge Institute of Technology, Research Report*, vol. 6, pp. 16-29, 2002.

[HKRS05] Z. Huzar, L. Kuzniarz, G. Reggio and J. Sourrouille, "Consistency problems in UML-based software development," *UML Modeling Languages and Applications*, pp. 1-12, 2005.

[KNL11] Z. Khai, A. Nadeem and G. Lee, "A Prolog Based Approach to Consistency Checking of UML Class and Sequence Diagrams," *Software Engineering, Business Continuity, and Education*, pp. 85-96, 2011.

[KR07] J. Küster and K. Ryndina, "Improving inconsistency resolution with side-effect evaluation and costs," *Model Driven Engineering Languages and Systems*, pp. 136-150, 2007.

[KRSH02] L. Kuzniarz, G. Reggio, J. Sourrouille and Z. Huzar, "Consistency problems in uml-based software development," *Blekinge Inst.of Technology Research Report*, vol. 2002, pp. 06, 2002.

[KSD09] A. Keller, H. Schippers and S. Demeyer, "Supporting inconsistency resolution through predictive change impact analysis," in *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, 2009, pp. 9.

[LEM02] W. Q. Liu, S. Easterbrook and J. Mylopoulos, "Rule-based detection of inconsistency in UML models," in *Workshop on Consistency Problems in UML-Based Software Development*, 2002, .

[LMT09] F. J. Lucas, F. Molina and A. Toval, "A systematic review of UML model consistency management," *Information and Software Technology*, vol. 51, pp. 1631-1645, 2009.

[Men01] T. Mens, "A formal foundation for object-oriented software evolution," in *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, 2001, pp. 549-552.

[MS05] T. Mens and J. Simmonds, "A Framework for Managing Consistency of Evolving UML Models," *Software Evolution with UML and XML,* pp. 1, 2005.

[MS07] T. Mens and R. Van Der Straeten, "Incremental resolution of model inconsistencies," *Recent Trends in Algebraic Development Techniques,* pp. 111-126, 2007.

[MSH06] T. Mens, R. Van Der Straeten and M. D'Hondt, "Detecting and resolving model inconsistencies using transformation dependency analysis," *Model Driven Engineering Languages and Systems,* pp. 200-214, 2006.

[NCEF02] C. Nentwich, L. Capra, W. Emmerich and A. Finkelstein, "xlinkit: A consistency checking and smart link generation service," *ACM Transactions on Internet Technology (TOIT),* vol. 2, pp. 151-185, 2002.

[NE10] A. Nöhrer and A. Egyed, "Utilizing the relationships between inconsistencies for more effective inconsistency resolution," in *3rd Workshop on Living with Inconsistencies in Software Development, Colocated with ASE, Antwerp, Belgium,* pp. 39-43.

[NEF03] C. Nentwich, W. Emmerich and A. Finkelstein, "Consistency management with repair actions," in *Software Engineering, 2003. Proceedings. 25th International Conference on,* 2003, pp. 455-464.

[NER00] B. Nuseibeh, S. Easterbrook and A. Russo, "Leveraging inconsistency in software development," *Computer,* vol. 33, pp. 24-29, 2000.

[NRE11] A. Nohrer, A. Reder and A. Egyed, "Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution: NIER track," in *Software Engineering (ICSE), 2011 33rd International Conference on,* 2011, pp. 864-867.

[OJS05] D. Ossami, J. P. Jacquot and J. Souquières, "Consistency in UML and B multi-view specifications," in *Integrated Formal Methods,* 2005, pp. 386-405.

[Red11] A. Reder, "Inconsistency management framework for model-based development," in *33rd International Conference on Software Engineering (ICSE),* IEEE, 2011.

[RW03] H. Rasch and H. Wehrheim, "Checking consistency in UML diagrams: Classes and state machines," *Formal Methods for Open Object-Based Distributed Systems,* pp. 229-243, 2003.

[SC02] J. L. Sourrouille and G. Caplat, "Constraint checking in UML modeling," in *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering,* 2002, pp. 217-224.

[Shi06] Y. Shinkawa, "Inter-model consistency in UML based on CPN formalism," in

Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific, 2006, pp. 411-418.

[Sim03] J. Simmonds, "Consistency maintenance of UML models with description logics," Master's thesis, Vrije Universiteit Brussel, Belgium and *École* des Mines de Nantes, France, 2003.

[SM07] P. Sapna and H. Mohanty, "Ensuring consistency in relational repository of UML models," in *10th International Conference on Information Technology (ICIT 2007), 2007*, pp. 217-222.

[SMD03] M. Snoeck, C. Michiels and G. Dedene, "Consistency by construction: the case of MERODE," in *ER Workshops on Conceptual Modeling for Novel Application Domains*, Vol. 2814 Springer (2003), pp. 105-117, 2003.

[SMS03] R. Van Der Straeten, T. Mens and J. Simmonds, "Maintaining consistency between UML models using description logic," in *Workshop on" Consistency Problems in UML-Based Software Development II*, 2003, pp. 71.

[SMSJ03] R. Van Der Straeten, T. Mens, J. Simmonds and V. Jonckers, "Using description logic to maintain consistency between UML models," *«UML» 2003-the Unified Modeling Language.Modeling Languages and Applications*, pp. 326-340, 2003.

[SPM11] R. Van Der Straeten, J. Pinna Puissant and T. Mens, "Assessing the Kodkod model finder for resolving model inconsistencies," *Modelling Foundations and Applications*, pp. 69-84, 2011.

[SSJM04] J. Simmonds, R. Van Der Straeten, V. Jonckers and T. Mens, "Maintaining consistency between UML models using description logic," *Série L'objet-Logiciel, Base De Données, Réseaux*, vol. 10, pp. 231-244, 2004.

[SSM03] R. Van Der Straeten, J. Simmonds and T. Mens, "Detecting inconsistencies between UML models using description logic," in *Proceedings of the 2003 International Workshop on Description Logics (DL2003), Rome, Italy*, 2003, pp. 260-264.

[Str05] R. Van Der Straeten, "Inconsistency Management in Model-Driven Engineering," *Diss. PhD thesis, Vrije Universiteit Brussel*, 2005.

[SZ01] G. Spanoudakis and A. Zisman, "Inconsistency management in software engineering: Survey and open research issues," *Handbook of Software Engineering and Knowledge Engineering*, vol. 1, pp. 329-380, 2001.

[TE00] A. Tsiolakis and H. Ehrig, "Consistency analysis of UML class and sequence diagrams using attributed graph grammars," in *Proc. of Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems, Berlin*, 2000, pp. 77-86.

[UNKC08] M. Usman, A. Nadeem, T. Kim and E. Cho, "A survey of consistency checking techniques for uml models," in *Advanced Software Engineering and its Applications, 2008. ASEA 2008*, 2008, pp. 57-62.

[VDR11] O. Vasilecas, R. Dubauskaitė and R. Rupnik, "Consistency Checking of UML Business Model," *Technological and Economic Development of Economy*, vol. 17, pp. 133-150, 2011.

[WGN03] R. Wagner, H. Giese and U. Nickel, "A plug-in for flexible and incremental consistency management," in *Proc. of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-Based Software Development), San Francisco, USA*, 2003, .

[XHZSTM09] Y. Xiong, Z. Hu, H. Zhao, H. Song, M. Takeichi and H. Mei, "Supporting automatic model inconsistency fixing," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2009, pp. 315-324.

-Common Tasks for Model Evolution – Change Propagation within and across UML Models-

[Aji95] S. Ajila, "Software maintenance: an approach to impact analysis of objects change," *Software: Practice and Experience*, vol. 25, pp. 1155-1181, 1995.

[BLR11] S. Bode, S. Lehnert and M. Riebisch, "Comprehensive model integration for dependency identification with EMFTrace," in Joint Proc. of the First Int. Workshop on Model-Driven Software Migration (MDSM 2011) and the Fifth Int. Workshop on Software Quality and Maintainability (SQM 2011), 2011, pp. 17-20.

[CLNCDESS09] M. Chechik, W. Lai, S. Nejati, J. Cabot, Z. Diskin, S. Easterbrook, M. Sabetzadeh and R. Salay, "Relationship-based change propagation: A case study," in *Modeling in Software Engineering, 2009. MISE'09. ICSE Workshop on*, 2009, pp. 7-12.

[CREP11] A. Cicchetti, D. Di Ruscio, R. Eramo and A. Pierantonio, "JTL: a bidirectional and change propagating transformation language," *Software Language Engineering*, pp. 183-202, 2011.

[DW10] H. K. Dam and M. Winikoff, "Supporting change propagation in UML models," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010, pp. 1-10.

[EDGLMNR11] A. Egyed, A. Demuth, A. Ghabi, R. Lopez-Herrejon, P. Mäder, A. Nöhrer and A. Reder, "Fine-tuning model transformation: change propagation in context of consistency, completeness, and human guidance," *Theory and Practice of*

Model Transformations, pp. 1-14, 2011.

[GW07] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *Software and Systems Modeling*, vol. 8, pp. 21-43, 2009.

[Han97] J. Han, "Supporting impact analysis and change propagation in software engineering environments," in Software Technology and Engineering Practice, 1997. Proceedings., Eighth IEEE International Workshop on [Incorporating Computer Aided Software Engineering], 1997, pp. 172-182.

[HEEO12] F. Hermann, H. Ehrig, C. Ermel and F. Orejas, "Concurrent model synchronization with conflict resolution based on triple graph grammars," *Fundamental Approaches to Software Engineering*, pp. 178-193, 2012.

[HLR06] D. Hearnden, M. Lawley and K. Raymond, "Incremental model transformation for the evolution of model-driven systems," *Model Driven Engineering Languages and Systems*, pp. 321-335, 2006.

[IKD08] N. Ibrahim, W. Wan Kadir and S. Deris, "Comparative evaluation of change propagation approaches towards resilient software evolution," in *the Third International Conference on Software Engineering Advances, ICSEA'08.*, 2008, pp. 198-204.

[Ivk11] I. Ivkovic, "Model Synchronization for Software Evolution," *PhD diss., University of Waterloo*, 2011.

[IK06] I. Ivkovic and K. Kontogiannis, "Towards automatic establishment of model dependencies using formal concept analysis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, pp. 499-522, 2006.

[JE04] S. Johann and A. Egyed, "Instant and incremental transformation of models," in *Proceedings of the 19th International Conference on Automated Software Engineering, 2004*, pp. 362-365.

[Jim05] A. M. Jimenez, "Change propagation in the MDA: A model merging approach," *Master's Thesis, University of Queensland*, 2005.

[LFR12] S. Lehnert, Q. Farooq and M. Riebisch, "A taxonomy of change types and its application in software evolution," in *IEEE 19th International Conference and Workshops on Engineering of Computer Based Systems (ECBS), 2012*, pp. 98-107.

[LR12] S. Lehnert and M. Riebisch, "Tackling the Challenges of Evolution in Multiperspective Software Design and Implementation," in *4th Workshop Design for Future (DFF2012) and the 14th Workshop Software Reengineering (WSR)*, Bad Honnef, Germany, May 2-4, Softwaretechnik-

Trends 32(2):27-28, 2012..

[RBÖV08] I. Ráth, G. Bergmann, A. Ökrös and D. Varró, "Live model transformations driven by incremental pattern matching," *Theory and Practice of Model Transformations*, pp. 107-121, 2008.

[RVV09] I. Ráth, G. Varró and D. Varró, "Change-driven model transformations," *Model Driven Engineering Languages and Systems*, pp. 342-356, 2009.

[SF01] T. Sunetnanta and A. Finkelstein, "Automated consistency checking for multiperspective software specifications," in *Workshop on Advanced Separation of Concerns. Toronto*, 2001, .

[XLHZTM07] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi and H. Mei, "Towards automatic model synchronization from model transformations," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 164-173.

[XSHT11] Y. Xiong, H. Song, Z. Hu and M. Takeichi, "Synchronizing concurrent model updates based on bidirectional transformation," *Software and Systems Modeling*, pp. 1-16, 2011.

-Common Tasks for Model Evolution – Uncertainty Management in UML Models-

[DE07] O. De Weck, C. Eckert and J. Clarkson, "A classification of uncertainty for early product and system design," *Massachusetts Institute of Technology, Engineering Systems Division*, 2007.

[FBCS11] M. Famelis, S. Ben-David, M. Chechik and R. Salay, "Partial models: A position paper," in *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, 2011, pp. 1.

[FSC12-1] M. Famelis, R. Salay and M. Chechik, "Partial models: Towards modeling and reasoning with uncertainty," in *Software Engineering (ICSE), 2012 34th International Conference on*, 2012, pp. 573-583.

[FSC12-2] M. Famelis, R. Salay and M. Chechik, "The semantics of partial model transformations," in *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*, 2012, pp. 64-69.

[SFGRCS12] P. SaadatPanah, M. Famelis, J. Gorzny, N. Robinson, M. Chechik and R. Salay, "Comparing the Effectiveness of Reasoning Formalisms for Partial Models", 2012.

[IFED09] H. Ibrahim, B. H. Far, A. Eberlein and Y. Daradkeh, "Uncertainty management in software engineering: Past, present, and future," in *Electrical and Computer Engineering, 2009. CCECE'09. Canadian Conference on*, 2009, pp. 7-12.

[RJC12] A. J. Ramirez, A. C. Jensen and B. H. C. Cheng, "A taxonomy of uncertainty for

dynamically adaptive systems," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on,* 2012, pp. 99-108.

[SCG12] R. Salay, M. Chechik and J. Gorzny, "Towards a methodology for verifying partial model refinements," in *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on,* 2012, pp. 938-945.

[SCH12] R. Salay, M. Chechik and J. Horkoff, "Managing Requirements Uncertainty with Partial Models," *Proc.of RE,* vol. 12, 2012.

[SFC12] R. Salay, M. Famelis and M. Chechik, "Language Independent Refinement using Partial Modeling," *Fundamental Approaches to Software Engineering,* pp. 224-239, 2012.

[*SMB09*] R. Van Der Straeten, T. Mens and S. Van Baelen, "Challenges in Model-Driven software engineering," *Models in Software Engineering,* pp. 35-47, 2009.

[XPPAB08] J. Xiao, P. Pinel, L. Pi, V. Aranega and C. Baron, "Modeling uncertain and imprecise information in process modeling with uml," in *Fourteenth International Conference on Management of Data (COMAD), Mumbai,* 2008, .

[ZRK97] H. Ziv, D. Richardson and R. Klösch, "The uncertainty principle in software engineering," in *Proceedings of the 19th International Conference on Software Engineering (ICSE'97),* 1997, .