ARTICLES SUMMARIES : BIG DATA ANALYSIS

Article 1 : Complex Changes

This paper proposes a detection engine for complex changes in metamodels, addressing challenges of variability and overlap. It introduces three ranking heuristics to help users decide which overlapping complex changes are likely to be correct. The approach records the trace of atomic changes rather than computing them, ensuring a complete and ordered sequence without hidden changes. It also considers undo operations while recording atomic changes. The approach was validated on 8 real case studies, achieving full recall and an average precision of 70.75%, improved by heuristics up to 91% and 100% in some cases. In product development, evolving domain-specific modeling languages (DSMLs) and generic metamodels like UML require co-evolving models and associated artifacts. Detecting changes between metamodel versions, termed Evolution Trace (ET), is essential for automating co-evolution. This paper focuses on detecting complex changes composed of sequences of atomic changes, crucial for improving co-evolution. Operator-based approaches face limitations due to the high number of complex changes. Detecting complex changes based on atomic changes presents challenges, as ensuring all detected changes are correct is difficult. Existing approaches struggle to achieve 100% recall or precision due to uncertainty in user intent during evolution. The paper suggests aiming for 100% recall and prioritizing detected changes based on probability to support user decision-making. This detection approach can complement existing co-evolution methods, providing a solid basis for automatic actions.
In the first section, we begin by defining metamodel evolution as it commonly happens in practice. Then, we demonstrate the repercussions of metamodel evolution on associated artifacts. We emphasize the importance of detecting complex changes and their role in maintaining the affected artifacts. We explore the concept of metamodel evolution and its impact on related artifacts. We define metamodel evolution as equivalent to one or several releases of the modeling language, where each release corresponds to a version update of the metamodel. The detection of complex changes is crucial during metamodel evolution, as it helps in maintaining impacted artifacts such as model instances, transformation scripts, and OCL constraints. Complex changes provide valuable knowledge for preserving information in artifacts, preventing data loss during evolution. For example, when properties are moved or pulled from one class to another in the metamodel, complex changes ensure that corresponding data in model instances and constraints are appropriately updated. We illustrate the co-evolution process of model instances and OCL constraints considering both atomic changes and complex changes, showcasing the preservation of valuable information and prevention of data loss. The 2nd section outlines an approach for detecting complex changes in metamodel evolution. It begins by describing the process of obtaining atomic changes and then introduces the definition of a complex change and considerations for its detection. Following this, the detection algorithm is presented, followed by its application to seven complex changes.
The overall approach involves recording the atomic change trace and then matching complex change patterns to it to generate a complex change trace. The approach is designed to meet four main requirements: ensuring no changes are hidden during detection, accommodating variability in complex changes, achieving high recall (100%) in detecting potential complex changes, and prioritizing overlapping complex changes to assist users in selecting the ones that align with their intention. Additionally, the approach aims to detect all potential undo operations.

For atomic change detection, a tracking approach is proposed that records all changes applied by users within a modeling tool at runtime without altering its interface. This ensures that no changes are hidden or lost during the detection process. An existing tool called Praxis is used for this purpose, which interfaces with modeling editors to record all changes that occur during evolution. The set of atomic changes considered includes add, delete, and update of metametamodel elements, where an update changes the value of a property of an element.

Complex changes in metamodel evolution are sequences of atomic changes, but simply defining them as such is not sufficient. This section addresses this by providing a more comprehensive definition and outlining an approach for detecting complex changes.

Complex Change Definition:

Complex changes are defined as patterns consisting of:

1. A set of allowed atomic change types, each with a multiplicity constraint.
2. Conditions relating pairs of change type elements that must be satisfied for the pattern to match. These conditions include name equality, type equality, generalization relationships (inheritance), and reference relationships.

This definition ensures variability by specifying a multiplicity for each change type and includes conditions necessary for a valid complex change.

Detection Algorithm of Complex Changes:

The detection algorithm works in two passes:

1. Generates all complex change candidates based on the type and multiplicity of the pattern. It iterates through the atomic changes in the evolution trace and creates candidates that match the definition of a certain complex change.

2. Scans the candidate set and retains those that satisfy the pattern, ensuring enough atomic changes are identified and conditions are fulfilled.

The algorithm ensures full recall by systematically creating all complex change candidates that match the type and multiplicity of the pattern. However, it may return false positives by returning multiple overlapping complex changes that reuse the same atomic change occurrences.

This approach provides a systematic method for detecting complex changes in metamodel evolution, guaranteeing full recall by construction. However, it may produce false positives, which can be addressed using heuristics to rank overlapping complex changes and assist users in selecting correct ones.

The implementation of the approach involves extending the Praxis prototype, integrated into Eclipse EMF Framework, to detect complex changes from recorded atomic changes during model creation or evolution. The Complex Change Detection Engine (CCDE) component, along with algorithms and heuristics, is implemented in Java and packaged into an Eclipse plug-in. This integration allows users to visualize and manage complex changes alongside atomic changes within the modeling environment, aiding in model evolution and maintenance.

The validation of the approach involved conducting experiments on eight real case studies, including Unified Modeling Language (UML) Class Diagram, Graphical Modeling Framework (GMF), OCLPivot, BaseCST, CompleteOCLCST, EssentialOCLCST, ExtendedTypes, and Benchmark. These case studies encompassed a variety of metamodels from both OMG standards and private initiatives such as Eclipse MDT tools like Papyrus and Modisco.

The process included manual analysis of atomic changes from version n to version n+1 of each metamodel to identify expected complex changes. This manual analysis was crucial for assessing the accuracy of the approach. Then, the tool was used to record the atomic changes while manually evolving the metamodels to their next versions. The recorded trace of atomic changes was then used to detect complex changes. The validation aimed to evaluate the precision, recall, and F-score of the approach, as well as its time performance and memory consumption. The selected case studies, their sources, versions, and the number of occurrences of atomic and complex changes between the two versions were documented for analysis.

As a conclusion, the paper introduces an approach for accurately detecting complex changes during metamodel evolution by recording real-time user editing actions. It ensures full recall of complex changes and supports user selection of correct changes while detecting undo operations. Validation on eight real case studies shows 100% recall of complex changes. Future work includes improving precision, addressing hidden changes, and extending applicability to previously evolved metamodels.

Article 2 : Model Matching Epsilon

This article is entitled " Eclipse Development tools for Epsilon" it was written by 3 people (D. Kolovos, R. F. Paige, F.A.C Polack) of the Department of computer of science from the University of York.
The article discusses Model Driven Development (MDD) and the need for Model Engineering (ME) facilities to manage models efficiently. MDD promotes using models as primary artifacts in software development, aiming to raise the abstraction level from code to models. ME tasks include validation, transformation, comparison, merging, text generation, reverse engineering, and version reconciliation.
Many task-specific languages, such as OCL for validation, QVT and ATL for transformation, and MOFScript for text generation, have been proposed to automate model management. However, there are practical problems with current approaches. Most languages build on a subset of OCL but implement everything from scratch, leading to duplication of effort, under-development of important features, and inconsistency in user experience across languages.
To address these issues, the article introduces the Extensible Platform for Specification of Integrated Languages for Model management (Epsilon). Epsilon provides a layered architecture for extensive reuse at both language definition and tool-support levels. The paper outlines the architecture of Epsilon and its task-specific languages, discusses the Eclipse-based Epsilon Development Tools, and concludes with plans for extending and improving tool-support for languages on the Epsilon platform.
Epsilon is a platform developed at the University of York and recently integrated into the Generative Modeling Technology (GMT) Eclipse sub-project. It aims to simplify the development of task-specific model management languages, addressing challenges like redundant efforts and inconsistency in language implementations. The infrastructure of Epsilon comprises two main components: Epsilon Model Connectivity (EMC) and Epsilon Object Language (EOL).
The EMC layer facilitates interaction with various model representation technologies like MDR, EMF, and XML. It offers interfaces to bridge specific modeling technologies with Epsilon languages. So far, EMC supports EMF, MDR, and XML models.
EOL, on the other hand, is an action language that extends OCL's navigational mechanisms and adds

features like statement sequencing, multiple model access, and conventional programming constructs. While EOL can be used independently for programmatic model management, its primary purpose is to support task-specific languages through grammar inheritance and an extensible execution engine.
Several task-specific languages have been developed on top of EMC and EOL. These languages aim to address specific model management tasks and propose abstractions and mechanisms with practical examples. Examples of task-specific languages implemented in Epsilon include validation, transformation, and text generation. Each language has its own syntax and features, contributing to advancing the state-of-the-art in model management. Detailed discussions on these languages are available in respective publications.
The article concludes by highlighting the benefits of using Epsilon for designing and implementing task-specific model management languages and supporting tools for Eclipse. It emphasizes the structured approach with minimal effort and overlap, along with enhanced reuse facilitated by Epsilon's infrastructure.
The development and testing of Epsilon and its associated languages are ongoing. Looking ahead, two significant enhancements are being considered. First, the implementation of language-specific debuggers for Eclipse is anticipated, with the necessary groundwork already laid in the execution engines. Second, there's a plan to implement static type checking to capture errors before module execution, as currently, type-checking is performed at runtime.
Additionally, there is an interest in identifying new model management tasks that could benefit from the creation of task-specific languages. This indicates a forward-looking approach to continually improving the functionality and utility of the Epsilon platform.

Article 3 : Evolving Software

The article addresses the need for understanding the evolution of software models by detecting both atomic and composite operations applied between successive versions. Existing approaches often focus solely on atomic operations, leading to cluttered difference reports. To overcome this limitation, the paper introduces an extension that detects composite operations by searching for them within the detected atomic operations. This approach enables the reuse of specifications for executing composite operations and enhances the accuracy of change detection. The effectiveness of the approach is evaluated through a real-world case study and scalability experiments.
In first section, the paper introduces the prerequisites for their detection approach, focusing on the metamodeling stack, which forms the basis for processing models using generic algorithms. The stack comprises three levels: meta-metamodel (M3), metamodel (M2), and model (M1). The meta-metamodel level defines a common meta-modeling language, while the metamodel level represents a modeling language's abstract syntax. Models are instances of metamodels and indirectly of the meta-metamodel.
Metamodeling frameworks leverage this stack by providing uniform model processing mechanisms based on the meta-metamodel level. These frameworks allow generic processing of models without considering specific metamodels, facilitating tasks like model comparison and composite operation specification and execution.
The approach discussed in the paper relies on two types of metamodel-agnostic tools: generic model comparison tools and tools for specifying and executing composite operations. These tools operate at the metamodel level (M2) and use operation specification metamodels to define composite operations. While these tools are independent of modeling languages' metamodels, operation specifications are tailored to specific modeling languages through template concepts.
The current model comparison tools operate in two phases: first, model matching algorithms

establish correspondences between model elements, and then a model diffing phase computes the differences based on these correspondences. For instance, tools like EMF Compare can detect atomic operations such as Add, Delete, Update, and Move between model versions. Consider a UML class diagram: the upper half of Figure 2 illustrates a generic diff model showing atomic operations between two versions of the diagram. It includes an example where the Extract Superclass refactoring is applied, introducing a new superclass (Vehicle) for existing classes and pulling up common properties to the superclass.
Composite operations are specified with rules defining their preconditions, postconditions, and actions. The Extract Superclass refactoring is an example, depicted in the lower part of Figure 2, using graph transformation syntax. The precondition specifies conditions such as equally named properties to be pulled up to the new superclass. The postcondition defines the resulting state after the refactoring. Current execution engines for model transformations support executing such specifications but not detecting occurrences of them in model differences obtained from model comparison tools, creating a gap between these worlds.
To bridge this gap, we propose generating an intermediate structure from operation specifications, as shown in Figure 3. This structure extends composite operation specifications by explicitly defining their comprised atomic operations, forming a diff pattern. By matching this pattern with a diff model from model comparison tools, occurrences of the composite operation can be detected. Generating the diff pattern involves computing a diff model by comparing the LHS and RHS of transformation rules. This process requires translating the LHS and RHS to "pure" models, which are direct instances of the modeling language's metamodel. Once generated, the diff pattern represents the minimal set of atomic operations constituting the composite operation, closing the gap between operation specifications and diff models. The process for a posteriori composite operation detection involves three phases: diff pattern matching, preselection, and diff element mapping. In the first phase, diff patterns from operation specifications are matched with the input diff model. The diff model and operation patterns are transformed into signatures for easier processing. Then, potential composite operations are preselected based on matching signatures. In the running example, this phase checks if the diff patterns are contained in the input diff model. Preselection involves adding all operation signatures and checking for missing elements to remove incomplete operations from consideration. After preselection, a diff element map is created to store correspondences between diff elements and patterns for further processing.
For the case study, they conducted a positivist approach based on real-world models and their evolution to assess the accuracy of our method. Following guidelines for empirical explanatory case studies, we applied our approach to detect composite operations in models from a public open-source project. Then they examined the scalability and efficiency of their approach by investigating how the runtime of their algorithm is affected by increasing model size and the number of concurrently applied atomic operations. They conducted experiments using purposefully created synthetic scenarios generated with their tool, Ecore Mutator. Importantly, they conducted this evaluation separately from the case study to better control the characteristics of the scenarios and isolate the impact of model size and concurrent operations on runtime.

The conclusion of the paper introduces a third phase for model comparison, focusing on aggregating atomic operations into composite ones. This approach is language-independent and utilizes existing composite operation specifications. The feasibility was demonstrated through an EMF-based implementation and a real-world case study, showing high accuracy in detecting composite operation applications. The conclusion also discusses possible extensions to address limitations, including fuzzy diff pattern matching to handle scenarios where atomic operations hide essential elements, partial condition evaluation to loosen strictness and increase detection, and composite composite

operations to aggregate composite operations into larger ones. These extensions aim to improve the approach's precision and scalability.

Article 4 : Measuring the quality of a diff algorithm

This paper introduces a formal method for comparing different algorithms used to automatically detect differences between documents. It acknowledges that there's no universal measure of quality for these differences, as it depends on the specific application and how the changes will be used. Traditionally, researchers focused on minimizing edit script size and computational complexity, but there's now a shift towards considering human interpretability. The paper proposes a universal delta model and metrics to compare differences effectively, aiming to identify the most suitable algorithms for specific tasks and domains.
It then discusses the challenge of comparing the quality of output from different diff algorithms, as they may produce correct but different results due to their internal delta models. Traditionally, the quality of deltas has been associated with minimality, focusing on reducing the size of edit scripts or measuring the number of edits. Alternative characterizations of quality have been proposed, categorized into two groups: minimality and interpretability.
Under the minimality approach, methods like measuring file size or edit distance have been used. Some algorithms compare compressed deltas to mitigate encoding differences. Edit distance calculations have become more precise with the introduction of complex operations and edit cost models.
The interpretability approach focuses on humans' ability to interpret and utilize the changes in deltas. This includes assessing naturalness, readability, and accuracy for human users. Some algorithms aim to capture natural operations on literary documents, while others focus on detecting higher-level changes in schema evolution or ontology diffing.
The article suggests that assessing delta quality should consider multiple dimensions, each capturing a different facet of quality, rather than relying on a single fixed value. This approach acknowledges the contrasting needs and expectations in characterizing delta quality.
The paper explores various scenarios where the quality of deltas produced by diff algorithms is crucial for users in different domains. These scenarios are labelled from S1 to S8 and cater to diverse needs and expectations:
1**. Programmers**: Novice programmers may prefer larger context in deltas for better understanding, while expert programmers might favor concise changes.

2**. Programmer browsing code history**: Developers may need to see high-level changes in file history, especially during discussions on code refactoring or when tracing changes across different versions.

3. **Sysadmins diffing files**: System administrators require both minimal delta size for efficient storage and readable output for human interpretation.

4. **Sysadmin transmitting diff information**: Sysadmins dealing with bandwidth and space constraints prioritize compact deltas for efficient data transmission, especially in multi-sited version control systems.

5**. Author revising literary documents**: Authors revising documents benefit from algorithms that detect specific types of changes accurately, such as recognizing style changes or paragraph splits.

6**. Developer working on visualization tools**: Developers building visualization tools rely on deltas containing domain-specific changes for generating precise and meaningful visualizations. Compact deltas can be challenging to interpret.

These scenarios highlight the diverse requirements for delta quality across different user groups and applications, emphasizing the need for diff algorithms that can effectively address specific needs, whether it's minimizing delta size, enhancing readability, or capturing domain-specific changes accurately. In this section, the paper outlines the goal of defining metrics to capture the qualities of deltas, which can assist in selecting the most suitable algorithm for specific scenarios. These metrics aim to objectively measure certain aspects of deltas, such as conciseness and intuitiveness, without considering features like reversibility or byte size, which are either taken for granted or implementation-specific. The paper acknowledges previous efforts, such as those focusing on human-interpretability and measuring deltas in terms of intuitiveness and conciseness. While there are similarities in terminology and goals, there are fundamental differences in scope, as the metrics proposed here are general and applicable to various domains and data structures. The structure of the metrics differs from previous works, aiming to provide a deeper characterization of changes in deltas. This includes separating basic and composite changes and capturing information about the nature of composite changes and composition rules. The paper introduces a metric called "terseness" to consider the amount of contextual information in each change. Before delving into the metrics, the paper describes the UniDM model of deltas, which include atomic and complex changes with various relations between them. The relations include application order and grouping relations, which express properties like reversibility and independent application of parts of a delta.
Overall, the paper lays the groundwork for defining metrics to evaluate delta quality, which can have applications in various scenarios and domains.