Versioning of Ordered Model Element Sets

Timo Kehrer, Udo Kelter
Software Engineering Group, University of Siegen, Germany
Email: {kehrer,kelter}@informatik.uni-siegen.de

***Abstract*—UML models contain various collections of ordered model elements. For example, a parameter list of an operation is an ordered set of parameters. The UML meta-model represents such orders by multi-valued properties having the modifier "ordered". A total of 43 properties of UML 2.4.1 are ordered. We have analyzed whether the order is conceptually relevant for model versioning (15 cases) or not (28 cases), how modifications of an order should be displayed in differences and how positions can be specified in edit scripts. On this basis, we propose a modified differencing pipeline of the SiLift model versioning framework which is able to detect ordering changes.**

I. Introduction

UML models contain various collections of ordered model elements. For example, a parameter list of an operation is an ordered set of parameters. Users can re-order the elements of a collection and insert or delete elements at a given position. Such editing effects should be reported correctly in model differences. Specifically if differences are to be used for patching models, patches can be edited by removing edit steps; as a consequence, all edit operations must be consistency-preserving [11], i.e. transform a model from one consistent state to another. State-based differencing of models is thus faced with three problems:

1) to identify the collections which are to be considered as ordered from a conceptual point of view, i.e. from the perspective of model versioning;
2) to offer an approach for handling positions of inserted, deleted or moved elements and to define appropriate consistency-preserving edit operations on ordered sets;
3) to actually compute the permutation of a collection.

The first problem is aggravated by the fact that typical meta-models, such as the UML meta-model [23], are design-level meta-models containing somehow arbitrary design decisions on the order of model element sets (s. Section II). In Section III, we analyze all ordered collections of the UML meta-model to see whether and how their order is conceptually relevant. Based on this analysis, Section IV presents edit operations which are required for modifying such ordered element sets. Edit operations can only be specified precisely (and implemented) if a concrete (runtime) model representation is available. Virtually all approaches to model versioning that have been published in the CVSM literature [5] abstract from concrete implementation technologies and consider models to be represented as abstract syntax graphs (ASG). Most often, ASGs are assumed to be typed, attributed but not to be ordered. Section V presents a lightweight approach to an ASG-based representation of ordered collections. Basically, we introduce a further refinement step on the meta-model layer which adds linked list implementations to ordered reference types of design meta-models. The approach is lightweight in the sense that our technique for ASG-based versioning [10], [11], which is based on graph transformation concepts, can be extended to ordered element sets with minimal effort (s. Section VI). The generation of executable model differences and the necessary modifications to the existing SiLift model differencing pipeline [19] are addressed in Section VII. Limitations of the edit operation recognition are discussed in Section VIII. Section IX concludes the paper.

II. Technology Dependency of (Meta) Models

From a tool construction point of view, state-based differencing tools must process models which are represented using a specific technological platform, e.g. the Eclipse Modeling Framework (EMF) [7]. If a collection of Java runtime objects (called *EObjects* in EMF) implements a collection of model elements and if this Java collection is ordered we cannot be sure that the collection of model elements is conceptually ordered because the order of the Java collection can result from an arbitrary design decision.

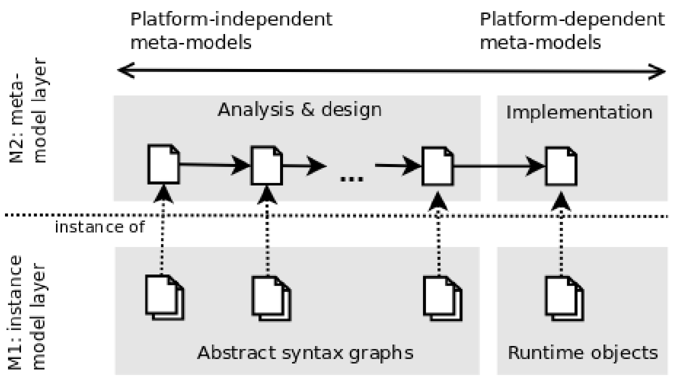

Figure 1. Meta-models and the related representation of instance models in different phases of object-oriented analysis, design and implementation.

In general, a system can be modelled on different levels of abstraction. Related phases of a development process are known as (object-oriented) analysis, design and implementation. The data managed by a system under development should initially be modelled using types which are independent from implementation technologies, and which do not anticipate design or implementation decisions. Such models are known as platform-independent or conceptual models.

The same principle also applies to the meta-model layer which is usually denoted as M2 in the four-layered meta-model hierarchy of the OMG [17]. Consequently, there is not only one meta-model for a given modeling language, but there are several meta-models (or versions of a meta-model) in different phases of object-oriented analysis, design and implementation (s. Figure 1). Thus, we distinguish platform-independent, conceptual meta-models from platform-dependent meta-models which are bound to a specific implementation technology.

Obviously, Ecore types are related to the implementation phase and incorporate all design decisions of earlier phases. Meta-models such as the UML superstructure specification [23] define data models of models. They are related to a detailed design phase since they incorporate numerous design decisions which facilitate an implementation in a object-oriented programming language (s. [13] for a discussion of resulting pseudo-differences). Unfortunately, there are no "conceptual meta-models" which would make clear which orders are conceptually relevant.

III. Classes of Ordered Sets of UML Model Elements

The UML meta-model defines 43 multi-valued reference types, i.e. M2 association ends, which are declared to be ordered. We identified five typical meta-modeling patterns how ordered reference types are used. The patterns are described in the following subsections A to E. Table I shows an overview which ordered reference type belongs to which pattern. Reference types are uniquely identified by their qualified name <*meta-class*>.<*reference-type*>. Multiplicities are specified in the usual UML notation. Column *c/nc* indicates whether it is a containment (c) or non-containment (nc) reference type. Column *Ref.* indicates whether the order is conceptually relevant (✓) or irrelevant (×).

	Reference type	Mult.	c/nc	Rel.
A	Class.ownedOperation	[0..*]	c	×
	Class.ownedAttribute	[0..*]	c	×
	Class.nestedClassifier	[0..*]	c	×
	StructuredClassifier.ownedAttribute	[0..*]	c	×
	Association.ownedEnd	[0..*]	c	×
	Connector.end	[2..*]	c	×
	Artifact.ownedAttribute	[0..*]	c	×
	Artifact.ownedOperation	[0..*]	c	×
	Enumeration.ownedLiteral	[0..*]	c	×
	Interface.nestedClassifier	[0..*]	c	×
	Interface.ownedAttribute	[0..*]	c	×
	Interface.ownedOperation	[0..*]	c	×
	Signal.ownedAttribute	[0..*]	c	×
	DataType.ownedAttribute	[0..*]	c	×
	DataType.ownedOperation	[0..*]	c	×
	Interaction.fragment	[0..*]	c	×
	InteractionOperand.fragment	[0..*]	c	×
	Property.qualifier	[0..*]	c	×
	Extend.extensionLocation	[1..*]	nc	×
	Constraint.constrainedElement	[0..*]	nc	×
	Association.memberEnd	[2..*]	nc	×
B	Action.input	[0..*]	nc	×
	Action.output	[0..*]	nc	×
	Association.endType	[1..*]	nc	×
	ConnectableElement.end	[0..*]	nc	×
C	TemplateSignature.ownedParameter	[0..*]	c	×
	TemplateSignature.parameter	[1..*]	nc	×
	InteractionUse.argument	[0..*]	nc	×
	BehavioralFeature.ownedParameter	[0..*]	c	✓
	Behavior.ownedParameter	[0..*]	c	✓
	Operation.ownedParameter	[0..*]	c	✓
	LoopNode.loopVariableInput	[0..*]	nc	✓
	InvocationAction.argument	[0..*]	nc	✓
	Message.argument	[0..*]	nc	✓
	CallAction.result	[0..*]	nc	✓
	LoopNode.result	[0..*]	nc	✓
	LoopNode.bodyOutput	[0..*]	nc	✓
	LoopNode.loopVariable	[0..*]	nc	✓
	ConditionalNode.result	[0..*]	nc	✓
	Clause.bodyOutput	[0..*]	nc	✓
D	Expression.operand	[0..*]	nc	✓
E	CombinedFragment.operand	[1..*]	nc	✓
	SequenceNode.executableNode	[0..*]	nc	✓

Table I
Ordered element sets defined by the UML meta-model.

A. Layout Information Encoding

If sets of elements are represented as lists in the graphical diagram representation (e.g. attributes of a class, operations of an interface, literals of an enumeration etc.), the respective reference type is usually declared to be ordered by the UML meta-model, even if the respective element sets are conceptually unordered. The order of the set is actually layout data. Changes in the layout are usually considered irrelevant in the context of model differencing. Thus the respective order can be reasonably ignored.

B. Synchronisation of Redundant Information

The UML meta-model defines several reference types to be derived. As long as the basic element sets are ordered, an order is also defined for the derived element sets. This way, corresponding elements can be identified by their position in the respective element sets. In other words, the order is used as an "implementation hint" how to technically realize the synchronisation of the involved element sets. Derived element sets can be ignored completely in the context of model versioning.

C. Parameter Lists

Lists of formal and actual parameters are frequently used in the UML, all of them are declared to be ordered. The order is conceptually irrelevant when actual parameters, i.e. arguments, are explicitly bound to their formal counterpart, e.g. in case of template signatures and template bindings. In most cases, however, actual parameters are not explicitly bound to formal parameters, e.g. arguments which are passed to behavior specifications called by different types of actions in an activity diagram. The order of the respective parameter lists is relevant in the sense that it induces an implicit parameter binding.

D. Structured Expressions

In the UML, expressions are used to form the abstract syntax tree of a structured textual specification such as a formula or a constraint. An expression represents a node in an expression tree. It defines a symbol (e.g. an arithmetic operator) and has a possibly empty sequence of operands. Whether the order is relevant finally depends on the semantics of an operator; in case of commutative operators, the order may be conceptually ignored.

E. Execution Sequences

Finally, ordered reference types are used for some parts of the UML meta-model related to behavior diagrams which can be used to model execution sequences.

In activity diagrams, sequence nodes provide a way to model a sequence of actions. Thus, a sequence node is associated to an ordered set of executable activity nodes.

In sequence diagrams, execution traces can be described using combined fragments. A combined fragment is defined by an interaction operator and an ordered set of corresponding interaction operands. The semantics of a combined fragment is dependent upon the interaction operator. For some interaction operators, namely in case of strict or weak sequencing, the order in which the corresponding interaction operands are arranged is semantically relevant.

F. Conclusion

Our analysis shows that the order of several reference types in the UML meta-model is conceptually irrelevant. The order is conceptually relevant for parameter lists, structured expressions and execution sequences. In these cases, we are only interested in the relative order in which elements are arranged to each other in an instance model, their absolute position within a sequence does not matter at all.

IV. Edit Operations on Ordered Element Sets

Meta-models are just data models, they do not directly specify "editing behavior", i.e. edit operations to modify the respective instance models. Thus, the definition of sets of edit operations for a given modeling language is left to tool developers. Most approaches in the CVSM context assume generic graph operations on the ASG, or low-level operations on the respective implementation data structures. Only a few approaches explicitly define edit operations exposing formal parameters and a specification. Only [1], [20] have considered edit operations operating on ordered element sets in graphs. However, [1] assumes absolute positions of elements which seem to be not practicable in our context for several reasons. In [20], differencing of ordered element sets is based on the computation of the longest common subsequence (LCS) [16] of two element sequences and assumes edit operations which create or delete references to mimic element insertions and deletions at specific positions. However, the execution of such low-level operations may lead to inconsistent model states as element lists may be split into several parts. If not all changes of a difference can be applied on a target model, which is typically the case in patching or merging scenarios [14], this approach bears the risk of synthesizing inconsistent models.

In our approach, we define sets of consistency-preserving edit operations [11] on ordered element sets (for brevity, ordered element sets are called element lists). We assume a relative order of elements rather than absolute positions of elements. Thus, the position of an element is given by its context, i.e. its *predecessor* and *successor* element. In addition to consistency-constraints defined by the meta-model (e.g. multiplicities), consistency-preserving list operations have to consider the generic invariant for linked element lists that each element in a list (except for the first one and the last one), has a predecessor and a successor element before and after the application of an operation.

We define a set of edit operations which we consider to be mandatory in order to modify element lists; operations to insert elements into lists in Section IV-A, operations to delete list elements in Section IV-B, global movements of elements between lists in Section IV-C, operations for local element shifts within a list in Section IV-D. Simplified variants of these edit operations working on containment lists are discussed in Section IV-E.

A. Insertion Operations

It must be possible to *insert* an element at a specific position of a list. We need operations for inserting existing elements into lists ($add*$) and creating new elements within a list ($create*$):

- $addAfter(l, pre, in)$: Inserts an element in into the list l such that the list element pre becomes the predecessor of in.
- $addInfront(l, in)$: Inserts an element in as the new first element of list l.
- $createAfter(l, pre)$: Inserts a new element into the list l such that the list element pre becomes the predecessor of in.
- $createInfront(l)$: Inserts an new element as the new first element of list l.

The $add*$ operations are applicable to non-containment lists, $create*$ operations are applicable to containment lists whose elements can not exist outside a list.

Alternatively, we could use operations which insert before an element of a list and at the end of a list, our algorithms do not depend on this choice.

Note that there are two options to implement the $create*$ operations. In a generic implementation, they can be equipped with an additional parameter indicating the type of the ASG element which is to be created. Alternatively, a dedicated $create$ operation can be provided for each non-abstract meta-class defined by the meta-model of the given modeling language.

B. Delete Operations

It must be possible to *delete* an element at a specific position of a list. In some cases, namely when a difference can not be described without causing transient effects (s. Section VIII), it must be possible to *clear* a complete list.

- $delete(l, el)$: Removes an element el from list l. If l is a containment list then element el is also deleted from the model.

– $clear(l)$: Removes all elements from list l. If l is a containment list then the removed elements are also deleted.

C. Move Operations

Global movements, i.e. edit operations that *move* an element from one list to another, must provide a parameter that indicates the position at which the element shall be inserted in the target list.

– $moveAfter(source, el, target, pre)$: Moves an element el from the list $source$ to the list $target$ such that the target list element pre becomes the predecessor of el in $target$.
– $moveInfront(source, el, target)$: Moves element el from the list $source$ to the front of list $target$.

D. Shift Operations

Operations which relocate elements locally within a list are called *shift* operations. The new predecessor of the element to be shifted is given as additional operation parameter.

– $shiftAfter(l, el, pre)$: Shifts an element el of list l such that pre becomes the predecessor of el in l.
– $shiftInfront(l, el)$: Shifts an element el of list l in front of the list.

For parameter lists (s. Section III-C) and sequences of operands in a structured expression (s. Section III-D), it is sometimes useful to mutually exchange the position of two adjacent elements in a list. For reasons of convenience, we define an optional operation *swap* for this purpose, although the same effect can be achieved by an element shift.

– $swap(l, el1, el2)$: Mutually exchanges the position of two adjacent elements $el1$ and $el2$ ($el2$ is the successor of $el1$ in l) such that $el2$ becomes the predecessor of $el1$ in l.

E. Simplified Operations on Containment Lists

If an edit operation has parameters for a list and for an element in this list then a precondition checks whether the element is actually contained in the list.

In case of containment lists, this potential error can be avoided by omitting the parameter for the list: since an element is always contained in exactly one containment list, this list is implicitly specified by the parameter for the element.

V. ASG-based Representation of Ordered Sets

In principle, there are two options to encode the position of list elements in an ASG:

1. Edges (references) can be extended by an additional label that specifies the absolute index of an element within an ordered element set.
2. The list elements themselves, i.e. the ASG nodes (objects) can be extended by additional edges (references) indicating the relative position of an element with respect to its predecessor/successor.

Obviously, both variants have their individual advantages and disadvantages. For our purpose, i.e. providing versioning support for ordered element sets, variant (1.) has the serious disadvantage that many edit operations lead to large side effects. If an element is inserted into a list, the indices of *all* subsequent list items must be adjusted. However, side effects are to be avoided as they lead to changes which cannot be reliably assigned to the invocation of one edit operation [10].

In contrast, based on variant (2.), edit operations on ordered element sets can be implemented without causing side effects. Furthermore, existing ASG-based approaches to model versioning can still be used without a need for substantial adaptions. This applies both for graph-based model transformation concepts (s. Section VI) and structural approaches to model differencing (s. Section VII). Variant (2.) consequently leads to three generic transformations to which we briefly refer to as T_{MM}, T_{in} and T_{out}, each of the transformations can be applied in-place.

On the meta-model layer, design-level meta-models, in which ordered reference types are declared by simple data values, are refined by transformation T_{MM} such that linked list "implementations" are provided for each ordered reference type. Certain reference types can be ignored if they are to be considered as conceptually unordered.

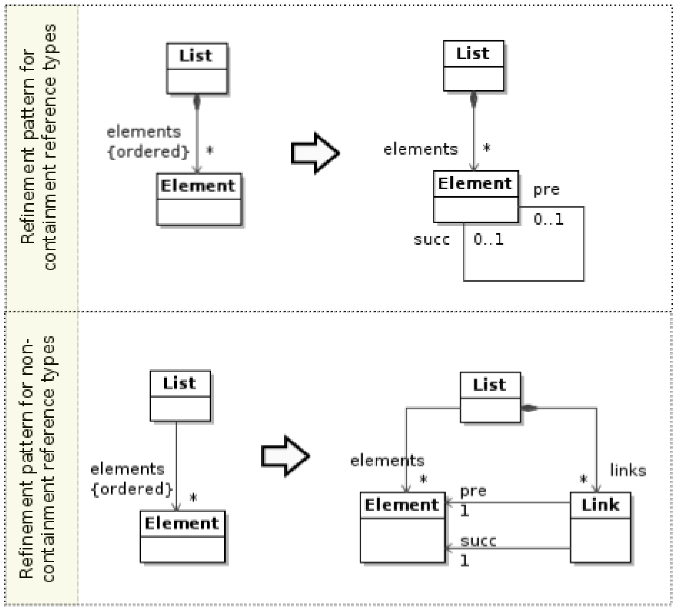

Figure 2. Meta-model refinement patterns

Containment reference types are refined according to the transformation rule template shown on top of Figure 2. In this case, the order of elements is implemented by a doubly linked list, i.e. each element in a containment list has references to its predecessor (pre) and its successor (succ) element in the list. As each element can be contained in at most one containment list, no additional information is required here.

For non-containment reference types, a separate "coordinator" class (called Link in Figure 2) manages predecessor and successor relationships in the context of one list. The refinement pattern for non-containment reference types is shown in the bottom part of Figure 2.

Versioning tools finally do not operate on instances of the original meta-model, but on its refined version. Thus, on the instance model layer, two additional transformations are required: An input transformation T_{in} supplements predecessor and successor relationships to each ordered element set. If the processed models are being modified, e.g. in case of patching and merging scenarios, an output transformation T_{out} arranges the order of runtime objects in terms of their original implementation structures according to the order which is induced by the predecessor/successor relationships, which are finally deleted by T_{out}.

VI. Rule-based Implementation of Edit Operations

Edit operations can be implemented by in-place model transformations. In previous work, we have implemented sets of edit operations for different EMF-based meta-models in the model transformation language Henshin [9], e.g. for Ecore and selected diagram types of the UML and the SysML [21]. Basically, a Henshin transformation rule can specify model patterns to be found and preserved, to be deleted, to be created, and to be forbidden. However, a Henshin transformation rule is defined on the ASG of a model. Consequently, the order of certain reference types defined by these meta-models has yet been ignored by the specification of the respective operation sets. Based on the ASG-based representation of ordered element sets introduced in Section V, additional edit operations modifying ordered element sets can be implemented in Henshin without adding new concepts to the transformation language.

As an example, the Henshin rule implementing the edit operation swap(e1,e2) on the containment list elements of the refined version of our meta-model snippet shown on top of Figure 2 is shown in Figure 3. The transformation rule defines variables a, e1, e2 and b. The variables e1 and e2 serve as input parameters which are bound to the elements which are to be swapped. The variables a and b are bound implicitly when the rule is applied to the subsequence $\langle$a, e1, e2, b$\rangle$. Please note that the input parameter for the list is omitted in the operation signature as elements is a containment list (cf. Section IV-E).

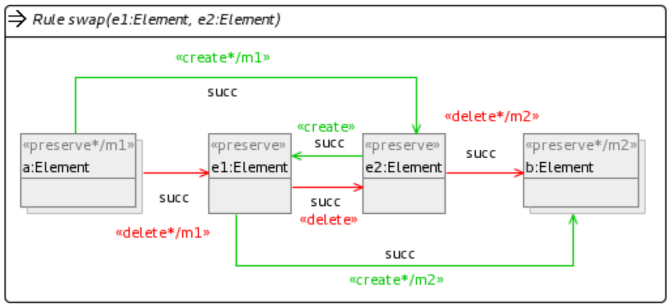

Figure 3. Implementation of edit operation swap(e1,e2) in Henshin.

The mutual exchange of the positions of e1 and e2 is achieved by deleting and creating predecessor/successor references as follows (for brevity, only successor references are shown in Figure 3): (1.) e1 becomes the successor of e2, (2.) e2 becomes the successor of a, and (3.) b becomes the successor of e1.

Note that swap(e1,e2) uses the concept of amalgamation [3]: The changes (2.) and (3.) are defined in terms of two separate multi rules, called m1 and m2, whose execution is optional. Thus, the rule can also be applied if the elements e1 and e2 which are to be swapped are located at the beginning or at the end of a list.

VII. Generation of Edit Scripts

In [11], we introduce a method and an automated tool chain which assumes that edit operations are implemented as Henshin transformation rules and which automatically generates executable differences, called *edit scripts*, from two models given as input. This method exploits the fact that Henshin transformation rules are formal specifications of edit operations and that the execution of a rule leads to well-defined patterns of change actions. The generation of an edit script is done in several steps of the differencing pipeline shown in Figure 4. Extensions to the pipeline presented in [11] are colored in black, unchanged parts are colored in gray.

- Initially, the two model versions A and B which are to be compared are enriched with predecessor/successor links by the input transformation T_{in} (s. Section V).
- Subsequently, a matching procedure identifies corresponding model elements and relationships in both models, i.e. corresponding objects in their ASGs. In principle, an existing matcher for the given model type can be used.
- In a second matching step, link objects that manage predecessor and successor relationships in the context of non-containment lists are matched; two link objects are considered to be corresponding when they link the same (i.e. corresponding) list objects in the context of corresponding lists.
- Based on a given matching, a low-level model difference can be derived; objects and references not involved in a correspondence are considered to be deleted or created.
- In the subsequent step, to which we refer to as operation recognition, groups of low-level changes which implement an edit operation are being detected, whereas each group represents the invocation of an edit operation. If not all low-level changes can be grouped to semantic change sets due to non-erasable transient effects, a feedback loop can trigger an incremental matching procedure (s. Section VIII).
- Finally, invocation arguments are extracted and operation invocations are analyzed for sequential dependencies before an edit script is being synthesized.

A prototypical implementation of the differencing pipeline is provided within the SiLift tool environment [19].

VIII. Discussion

We have evaluated the implementation of our approach based on a set of simple example models. In case of minor modifications of ordered element sets, our differencing approach delivers edit scripts which are comparable to the results

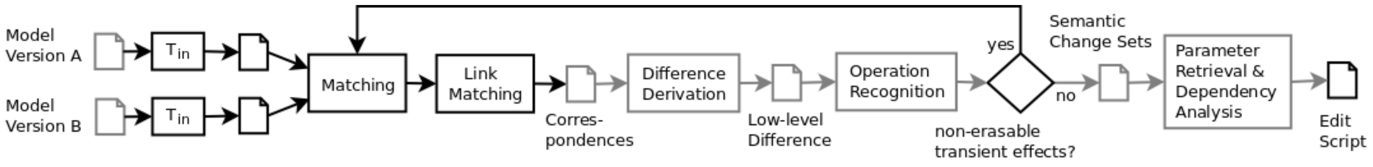

Figure 4. Modified differencing pipeline of the SiLift model versioning framework.

of established algorithms computing minimal edit distances between sequences based on standard list operations [16], [22]. Larger modifications of ordered element sets lead to transient effects (s. Section VIII-A). This principal limitation of our approach to operation recognition has already been stressed in [11]. However, this drawback is much more serious if operations on ordered element sets are included in the overall set of edit operations. Particularly, edit operations on ordered element sets can lead to transient effects that are not erasable, a possible "workaround" is discussed in Section VIII-B.

A. Transient Effects

With our approach to edit operation recognition, invocations of edit operations are detected only based on the changes which are observable in a low-level difference. If any low-level change that is caused by an operation invocation cannot be observed in a given difference, the respective edit operation invocation will not be recognized. This happens if a sequence of edit operations leads to so called *transient effects*, i.e. an edit operation in this sequence removes effects of an earlier one. Transient effects are a general limitation of operation recognition algorithms that run without backtracking; similar observations are reported in [15], where transient effects are called overlapping operations.

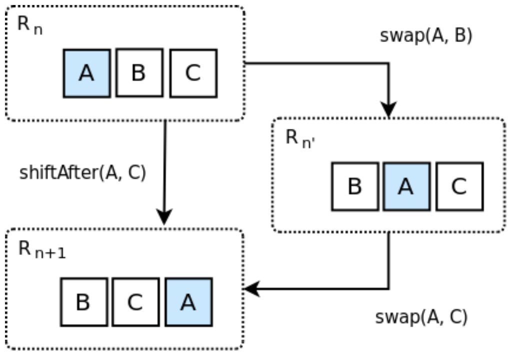

Figure 5. Example of an erasable transient effect.

Transient effects are welcome when an operation invocation is taken back completely, e.g. a model element was created and later deleted. In such cases, none of the applied edit operations will be recognized. Transient effects are not harmful either if they can be avoided in the sense that the difference can be described by an alternative set of operation invocations which finally leads to the same result. From an operation recognition point of view, we say that a transient effect is *erasable*. For example, the editing sequence ⟨swap(A,B),swap(A,C)⟩ shown in Figure 5 leads to a transient effect; swap(A,B) is only observable from revision R_n to $R_{n'}$, swap(A,C) is only observable from $R_{n'}$ to R_{n+1}. The transient effect is avoidable as R_n can be edited to become R_{n+1} by the application of shiftAfter(A,C), which is the result reported by our operation detection algorithm.

In some cases, however, edit operations on ordered element sets can lead to transient effects that are not erasable. To be more precise, the transient effects are not erasable for a given set of correspondences obtained by the matching algorithm of the differencing pipeline. For example, consider the situation shown on top of Figure 6. Here, no sequence of edit steps can be found transforming R_n to R_{n+1} using edit operations available in our set of consistency-preserving operations without causing transient effects.

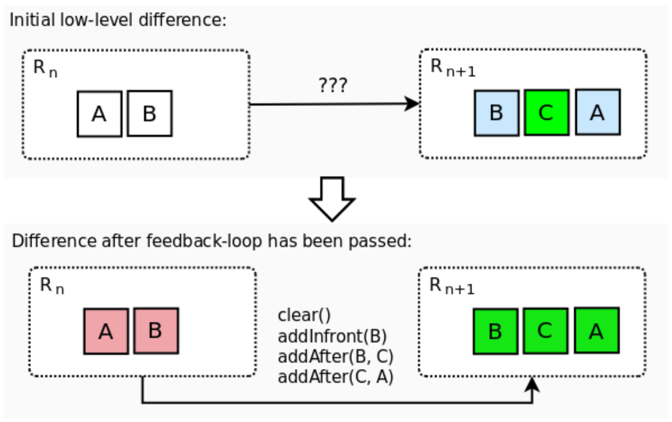

Figure 6. Non-erasable transient effects and their resolution.

B. Feedback Loop

A first approach to overcome the problem of non-erasable transient effects is to allow primitive graph operations, namely edit operations to separately add and remove objects and references of an ASG. This way, predecessor and successor references could be created and deleted without adapting the remaining context of the involved elements. This approach is adopted by [20]; its main drawback is that these edit operations are not guaranteed to be consistency-preserving in general, as already discussed in Section IV.

Instead, we propose to extend the SiLift differencing pipeline by a feedback loop such that the information gathered during edit operation recognition is taken into account by the creation of a low-level difference (s. Figure 4). The basic idea is to iteratively remove correspondences between elements in corresponding element lists.

A simple implementation of this feedback loop is to randomly select a pair of corresponding elements of both lists, to remove the correspondence between these elements, and to finally re-pass the link matching, difference derivation and operation recognition. The iteration ends as soon as all low-level changes can be grouped to semantic change sets, i.e. all non-erasable transient effects have been eliminated. The feedback loop terminates as each iteration only leads to the removal of a correspondence, no incremental calculation of a "better" matching is being triggered. In the worst case, none of the elements of corresponding lists are considered to be corresponding. In this case, we can be sure to find a sequence of consistency-preserving edit steps transforming R_n to R_{n+1}; firstly, all elements are removed from the list (i.e. the list is cleared) and subsequently inserted in the correct order, which is shown for our example in the bottom part of Figure 6.

More generally, the set of edit operations used to configure the operation detection engine must be consistent with the properties of the matching produced by the matcher in the differencing pipeline, which is a subject to future research.

IX. Conclusion

In this paper, we have addressed the specification and recognition of edit operations on ordered model element sets. In the current CVSM literature, the versioning of ordered element sets is often ignored (or not considered) as models are assumed to be represented as unordered ASGs. However, our analysis of the UML meta-model shows that the order can not be simply ignored in several cases.

Some approaches avoid this problem by deviating from a strict graph representation. The references of an ASG are treated as complex properties or features which are assigned to the source node of a reference. Thus, changes on ordered element sets can be described as feature changes. For example, in EMF Compare [6] and versioning approaches such as AMOR [2] that are based on EMF Compare, a change in the position of an element in a list is represented as "reference change" in the context of an object, [4] and [18] consider such changes as "modified element". However, this "workaround" cannot be simply applied to versioning approaches that are based on graph transformation concepts, e.g. [8].

In this paper, we presented a lightweight approach to overcome the conceptual weakness of current ASG-based approaches. Based on Henshin and the approach presented in [11] we have shown that the concept can be used for specifying and recognizing edit operations on ordered element sets, without the need for invasive adaptions of graph-based approaches.

Acknowledgment

This work is partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future - Managed Software Evolution.

References

[1] Alanen, M.; Porres, I.: Difference and Union of Models; p.2-17. in: Proc. Intl. Conf. on the Unified Modeling Language 2003; LNCS vol. 2863; 2003

[2] AMOR: Adaptable Model Versioning; http://www.modelversioning.org/; 2013

[3] Biermann, E.; Ermel, C.; Taentzer, G.: Lifting Parallel Graph Transformation Concepts to Model Transformation based on the Eclipse Modeling Framework; Electronic Communications of the EASST 26; 2010

[4] Cicchetti, A.; Di Ruscio, D.; Pierantonio, A.: A Metamodel independent approach to difference representation; p.165-185 in Journal of Object Technology 6:9, 2007

[5] Bibliography on Comparison and Versioning of Software Models; http://pi.informatik.uni-siegen.de/CVSM

[6] EMF Compare Project; http://www.eclipse.org/emf/compare

[7] EMF: Eclipse Modeling Framework; http://www.eclipse.org/emf

[8] Ehrig, H.; Ermel, C.; Taentzer, G.: A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications; p.202-216 in: Proc. FASE 2011; Springer; 2011

[9] EMF Henshin Project; http://www.eclipse.org/modeling/emft/henshin

[10] Kehrer, T.; Kelter, U.; Taentzer, G.: A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning; p.163-172 in: ASE 2011; ACM; 2011

[11] Kehrer, T.; Kelter, U.; Taentzer, G.: Consistency-Preserving Edit Scripts in Model Versioning; p.191-201 in: Proc. ASE 2013, Palo Alto, USA; IEEE; 2013

[12] Kehrer, T.; Kelter, U.; Pietsch, P., Schmidt, M.: Adaptability of Model Comparison Tools; in: Proc. 27th Intl. Conf. on Automated Software Engineering; 2012

[13] Kelter, U.: Pseudo-Modelldifferenzen und die Phasenabhängigkeit von Metamodellen; p.117-128 in: Proc. Software Engineering 2010, Paderborn; LNI 159, GI; 2010

[14] Kelter, U.; Kehrer, T.; Koch, D.: Patchen von Modellen; p.171-184 in: Proc. Software Engineering 2013, 26.02.-01.03.2013, Aachen; Lecture Notes in Informatics 213, GI, Bonn; 2013

[15] Langer, P.; Wimmer, M.; Brosch, P.; Herrmannsdörfer, M.; Seidl, M.; Wieland, K.; Kappel, G.: A posteriori operation detection in evolving software models; p. 551-566 in: Journal of Systems and Software 86(2); 2013

[16] Myers, E.W.: An O(ND) difference algorithm and its variations; Algorithmica 1(2):251-266; 1986

[17] MOF: Meta-Object Facility; http://www.omg.org/mof/

[18] Rivera, J.E.; Vallecillo, A.: Representing and Operating with Model Differences; p.141-160 in: Proc. TOOLS EUROPE 2008; LNBIP 11, Springer; 2008

[19] SiLift project website; http://pi.informatik.uni-siegen.de/Projekte/SiLift

[20] Sriplakich, P.; Blanc, X.; Gervais, M.: Supporting Collaborative Development in an Open MDA Environment; p.244-253 in: Proc. ICSM 2006, Philadelphia, USA; IEEE Computer Society; 2006

[21] SysML: OMG Systems Modeling Language; http://www.omgsysml.org

[22] Tichy, W.F.: The String-to-String Correction Problem with Block Moves; p.309-321 in: ACM Transactions on Computer Systems (TOCS); 2(4); 1984

[23] Unified Modeling Language: Superstructure, Version 2.4.1; OMG, Doc. formal/2011-08-05; 2011