Change-Driven Model Transformations

Derivation and Processing of Change Histories

István Ráth[1], Gergely Varró[2], and Dániel Varró[1]

[1] Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
{rath, varro}@mit.bme.hu
[2] Department of Computer Science and Information Theory,
H-1117 Magyar tudósok krt. 2, Budapest, Hungary
gervarro@cs.bme.hu

Abstract. Nowadays, evolving models are prime artefacts of model-driven software engineering. In tool integration scenarios, a multitude of tools and modeling languages are used where complex model transformations need to incrementally synchronize various models resided within different external tools. In the paper, we investigate a novel class of transformations, which is directly triggered by model changes. First, model changes in the source model are recorded incrementally by a *change history model*. Then a model-to-model transformation is carried out to generate a change model for the target language. Finally, the target change history model is processed (at any time) to incrementally update the target model itself. Moreover, our technique also allows incremental updates in an external model where only the model manipulation interface is under our control (but not the model itself). Our approach is implemented within the VIATRA2 framework, and it builds on live transformations and incremental pattern matching.

Keywords: incremental model transformation, change models, change-driven transformations

1 Introduction

Model transformations play a key role in model-driven software engineering by providing embedded design intelligence for automated code generation, model refactoring, model analysis or reverse engineering purposes.

Most traditional model transformation frameworks support *batch transformations* where the execution of a transformation is initiated (on-demand) by a systems designer. Most traditional model transformation frameworks follow this approach. As an alternate solution (proposed recently in [1, 2]), *live transformations* (or active transformations) run in the background as daemons, and continuously react to changes in the underlying models. In this respect, a transformation can be executed automatically as soon as a transaction on the model has completed.

Up to now, the design and execution of batch transformations and live transformations were completely separated, i.e. the same transformation problem had to be formulated in a completely different way.

In the current paper, we try to bridge this conceptual gap by introducing change-driven model transformations. More specifically, we first define the concept of a *change history model*, which serves as a history-aware log of elementary model changes, which record causal dependency / timeliness between such changes. We show how change history models can be derived incrementally by live transformations during model editing. Then we describe how change history models can be used to incrementally update a model asynchronously (i.e. at any desired time) by propagating changes using batch transformations.

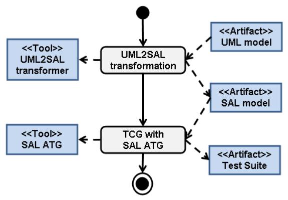
The use of change history models in model-to-model transformation scenarios has far-reaching consequences as incremental model transformations can be constructed with minimal knowledge about the current structure of the target model. For instance, transformations can still be implemented when only identifiers and a model manipulation interface are known, but the rest of the actual target model is non-materialized (i.e. does not exist as an in-memory model within the transformation framework). As a result, our concepts can be easily applied in the context of runtime models as well as incremental model-to-code transformation problems (where the latter will actually serve as the running example of the paper).

The rest of the paper is structured as follows. In Sec. 2, a motivating case study is introduced as a running example for our paper. The main concepts of change-driven transformations and change history models are introduced in Sec. 3. Section 4 details the main steps of the approach on the running example. Finally, Section 5 summarizes related work and Sec. 6 concludes our paper.

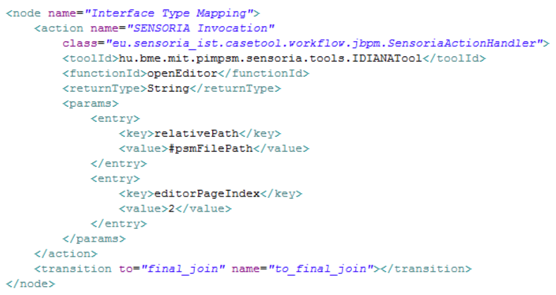
2 Motivating scenario

Our motivating scenario is based on an actual tool integration environment developed for the SENSORIA and MOGENTES EU research projects.Here high-level workflow models (with control and data flow links, artefact management and role-based access control) are used to define complex development processes which are executed automatically by the JBoss jBPM workflow engine, in a distributed environment consisting of Eclipse client workstations and Rational Jazz tool servers. The process workflows are designed in a domain-specific language, which is automatically mapped to an annotated version of the jPDL execution language of the workflow engine. jPDL is an XML-based language, which is converted to an XML-DOM representation once the process has been deployed to the workflow engine.

A major design goal was to allow the process designer to edit the process model and make changes without the need for re-deployment. To achieve this, we implemented an *asynchronous incremental code synchronizing model transformation*. This means that (i) while the user is editing the source process model, the changes made are recorded. Then (ii) these changes can be mapped incrementally to the target jPDL XML model without re-generating it from scratch. Additionally, (iii) the changes can be applied directly on the deployed XML-DOM representation through jBPM's process manipulation DOM programming interface, but, (iv) in order to allow the changes to applied to the remote workflow server, the actual XML-DOM manipulation is executed on a remote host asynchronously to the operations of the process designer.

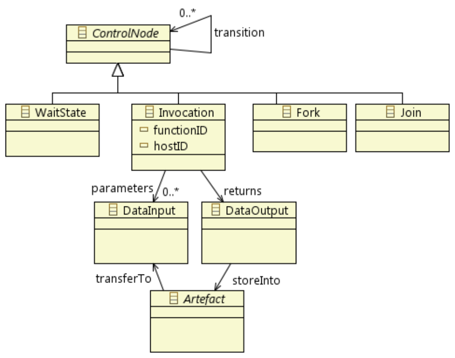


(a) Domain-specific workflow model

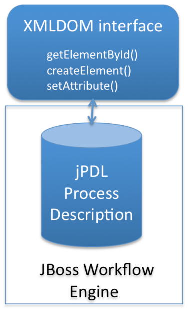


```
<node name="Interface Type Mapping">
    <action name="SENSORIA Invocation"
        class="eu.sensoria_ist.casetool.workflow.jbpm.SensoriaActionHandler">
        <toolId>hu.bme.mit.pimpsm.sensoria.tools.IDIANATool</toolId>
        <functionId>openEditor</functionId>
        <returnType>String</returnType>
        <params>
            <entry>
                <key>relativePath</key>
                <value>#psmFilePath</value>
            </entry>
            <entry>
                <key>editorPageIndex</key>
                <value>2</value>
            </entry>
        </params>
    </action>
    <transition to="final_join" name="to_final_join"></transition>
</node>
```

(b) JPDL XML Document



(c) Domain-specific workflow metamodel



(d) jPDL interface

Fig. 1. Model representations in the motivating scenario

Example. A simple tool integration workflow model is given in Fig. 1(a) together with its jPDL XML representation (in Fig. 1(b)). Moreover, a metamodel of the source language is given in Fig. 1(c). In case of the target language, an interface is provided to manipulate XML documents (see Fig. 1(d)).

Metamodeling background. Since the actual tool integration framework is built upon the model repository and transformation support of the VIATRA2 framework [3], we also use it for the current paper for demonstration purposes. However, all metamodels will be presented as a traditional EMF metamodel to stress that all the main concepts presented could be transfered to other modeling environments as well.

VIATRA2 uses the VPM [4] metamodeling approach for its model repository, which uses two basic elements: entities and relations. An *entity* represents a basic concept of a (modeling) domain, while a *relation* represents the relationships between other model elements. Furthermore, entities may also have an associated value which is a string that contains application-specific data.

Model elements are arranged into a strict containment hierarchy, which constitutes the VPM model space. Within a container entity, each model element has a unique local name, but each model element also has a globally unique identifier which is called a fully qualified name (FQNs are constructed by hierarchical composition of local names, e.g. “workflow.model.node0”).

There are two special relationships between model elements: the *supertypeOf* (inheritance, generalization) relation represents binary superclass-subclass relationships (like the UML generalization concept), while the *instanceOf* relation represents type-instance relationships (between meta-levels). By using an explicit *instanceOf* relationship, metamodels and models can be stored in the same model space in a compact way.

3 Change history models in incremental model synchronization

In the current paper, we investigate a model synchronization scenario where the goal is to *asynchronously* propagate changes in the source model M_A to the target model M_B. This means, that changes in the source model are not mapped on-the-fly to the target model, but the synchronization may take place at any time. However, it is important to stress that the synchronization is still *incremental*, i.e. the target model is not re-generated from scratch, but updated according to the changes in the source model.

Moreover, our target scenario also requires that M_B is not materialized in the model transformation framework, but accessed and manipulated directly through an external interface IF of the their native environment. This is a significant difference to traditional model transformation environments, where the system relies on model import and export facilities to connect to modeling and model processing tools in the toolchain.

To create asynchronous incremental transformations, we extend traditional transformations (which take models as inputs and produce models as output) by *change-driven transformations* which take model manipulation operations as inputs and/or produce model manipulation operations as output. By this approach, our mappings may be executed without the need of materializing source and target models directly in the transformation system, and may also be executed asynchronously in time.

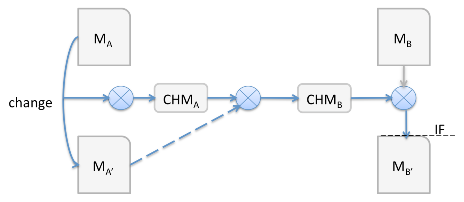


Fig. 2. Model synchronization driven by CHMs

As we still rely on model transformation technology, operations on models need to be represented in the model space by special trace models which encode the changes of models as model manipulation sequences. We call these models *change history models* (CHMs in short). These models are generated automatically on-the-fly as the source model changes (see CHM_A in the left part of Fig. 2) using *live transformations* [2]. Live transformations are triggered by event-driven condition-action rules whenever a change is performed in the model space, and create an appropriate change history model fragment (connected to those parts of the model which were affected by the change).

The actual model transformation between the two languages is then carried out by generating a change history model CHM_B for the target language as a separate transformation (see middle part of Fig. 2).

As change history models represent a trace of model evolution, they may be automatically applied to models (see right part of Fig. 2). More precisely, we combine a snapshot of the model M_B (representing the initial state) and a change history model CHM_B (representing a sequence of operations applicable starting from the initial state) to create the final snapshot M'_B. In other words, the change history model CHM_B represents an "operational difference" between M'_B and M_B, with the order of operations preserved as they were actually performed on M_B.

3.1 Change history models

Change history models are conceptually derived from the model manipulation operations defined on the host language. These operations may be generic (i.e. corresponding to graph-level concepts such as "create node", "create edge", "change attribute value"), or domain-specific (corresponding to complex operations such as "remove subprocess", "split activity sequence"). In the current paper, we discuss the generic solution in detail, however, we also show how our approach can be extended to domain-specific languages in a straightforward way.

Change history metamodel The generic change history metamodel for VPM host models is shown in Fig. 3. CHM fragments are derived from the abstract *Operation* class, which can be optionally tagged with a Timestamp attibute for time-based tracing of, e.g. user editing actions. Operations are connected to each other by relations of type *next*, which enables the representation of operation sequences (transactions).

It is important to stress that CHMs do not directly reference their corresponding host models, but use fully qualified name (or unique ID) references. The reason for this is two-fold: (i) by using indirect references, CHMs may point to model elements that are no longer existent (e.g. have been deleted by a consecutive operation), and (ii) CHMs are not required to be materialized in the same model space as the host model (symmetrically, host models are not required to be materialized when processing CHMs). This allows to decoupling the actual models from the transformation engine which is a requirement for non-invasive scenarios where target models are indirectly manipulated through an interface.

By our approach, change history metamodel elements are either *EntityOperation*s or *RelationOperation*s. Entity operations use the *parentFQN* reference to define the containment hierarchy context in which the target entity is located *before* the operation represented by the CHM fragment was executed. Analogously, relation operations use *srcFQN* and *trgFQN* to define source and target endpoints of the target relation element (prior to execution). Note that we omitted inheritance edges from *EntityOperation* and *RelationOperation* in Fig. 3 for the sake of clarity.

All CHM elements correspond to elementary operations in the VPM model space, in the following categories:

– *creation* (shown on the far left): *CreateEntity* and *CreateRelation* represent operations when an entity or relation has been created (an entity in a given container, a relation between a source and target model element). Both CHM fragments carry information on the *type* ($typeFQN$) of the target element.

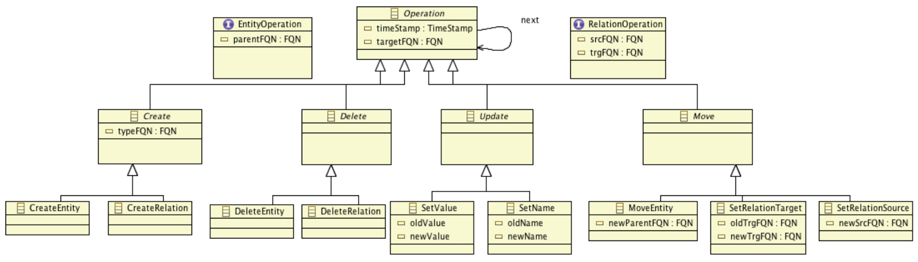


Fig. 3. Generic change history metamodel

- *deletions* (shown on the near left): *DeleteEntity* and *DeleteRelation* correspond to deletions of entities and relations.
- *updates* (shown on the near right): *SetValue* indicates an operation where the *value* field of an entity is overwritten; similarly, *SetName* represents an update in the local name of the target (in this case, as always, *targetFQN* points to the original FQN of the target model element, so this CHM fragment needs to be used carefully).
- *moves* (shown on the far right): *MoveEntity* corresponds to the reparenting of an entity in the VPM containment hierarchy. *SetRelationTarget* and *SetRelationSource* represent retargeting and resourcing operations.

4 Change-driven transformations

In this section, we demonstrate the concept and application of change-driven transformations (see Fig. 2) using change history models by the elaboration of the motivating scenario described in Sec. 2. First, we demonstrate (in Sec. 4.1) how CHMs can be derived automatically by recording model manipulations using live transformations. We introduce both generic (metamodel-independent) and domain-specific (metamodel-dependent) techniques to achieve this. Then we discuss (in Sec. 4.2) how model transformations can be designed between two CHMs of different languages. Finally, we describe (in Sec. 4.3) how CHMs can be asynchronously processed to incrementally update a model resided in a model repository or within a third-party tool accessed via an external interface.

4.1 Automatic generation of CHMs by live transformations

First, we demonstrate the automatic generation of change history models for recording modification operations carried out on the host model. Model changes may be observed using various approaches, e.g. by model notification mechanisms such as the EMF notification API, where the model persistence framework provides callback functions for elementary model changes. This approach is limited to recording only basic model manipulation operations, i.e. an appearance of a complex model element (e.g. a graph node with attribute values and type information) requires the processing of a *sequence* of elementary operations (e.g. “create node”, “set value”, “assign type”, etc). If the modification operations may be interleaving (e.g. in a distributed transactional environment, where multiple users may edit the same model), it is difficult to process operation sequences on this low abstraction level.

In contrast, live transformations [2] define changes on a higher abstraction level as a new match (or lost match) of a corresponding graph pattern (as used in graph transformations [6]). By this approach, we may construct a complex graph pattern from elementary constraints, and the system will automatically track when a new match is found (or a previously existing one is lost) – thus, model manipulation operations may be detected on a higher abstraction level, making it possible to assign change history models not only to elementary operations, but also to domain-specific ones.

More precisely, live transformations are defined by event-condition-action triples in the following sense:

- an *event* is defined with respect to a graph pattern, and may correspond to an appearance of a newly found match, or a disappearance of a previously existing one.
- *conditions* are evaluated on the transaction of elementary operations which resulted in the triggering of the event. They correspond to elementary operations affecting elements of the subgraph identified by the event’s (newly found or deleted) match.
- *actions* are model manipulation operations to be carried out on the model.

Basic patterns Fig. 4 shows three basic graph patterns and their VIATRA2 transformation language representations. Pattern `entity_in_parent` encompasses a containment substructure where an entity E is matched in a given parent entity *Parent*. A new match for this pattern occurs when any entity is created in the host model (when a new match is detected, concrete references as substitutions for pattern variables $E, Parent$ are passed to the transformation engine). Similarly, pattern `relation_source_target` corresponds to a relation R with its source S and target T elements, while pattern `modelelement_type` references any model element with its type. These patterns corre-

```
pattern entity_in_parent(E,P) =
{
 entity(Parent);
 entity(E) in Parent;
}
```

```
pattern modelelement_type(ME,T) =
{
  modelElement(Type);
  instanceOf(ME,Type);
}
```

```
pattern relation_source_target(R,S,T) =
{
 modelElement(S);
 modelElement(T);
 relation(R,S,T);
}
```

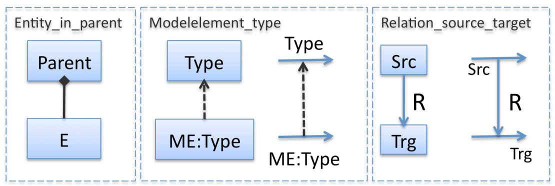


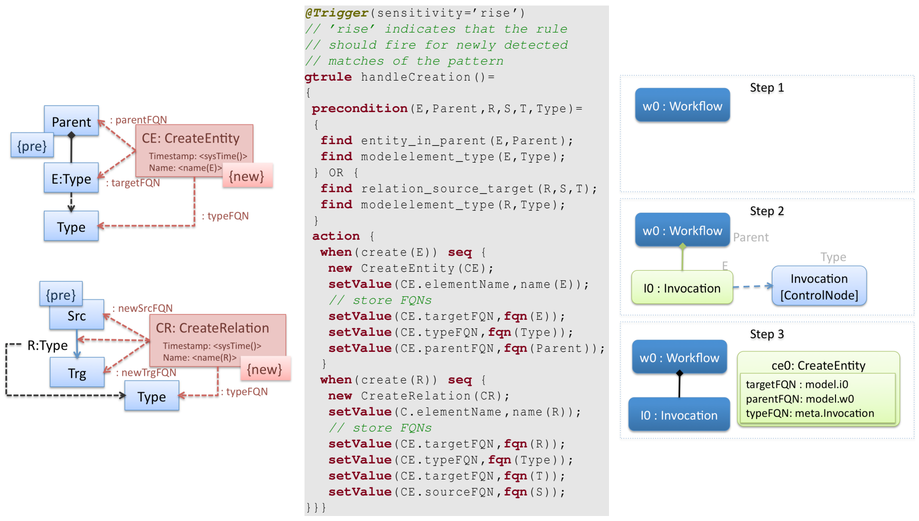
Fig. 4. Patterns for identifying relevant model manipulation events

spond to basic notions of the VPM (typed graph nodes and edges), and may be combined to create precondition patterns for event-driven transformation rules.

Generic derivation rules On the left, Fig. 5 shows a sample CHM generation rule for tracking the creation of model elements. A triggered graph transformation rule is defined for a composite disjunctive pattern, which combines cases of new appearances of entities and relations into a single event. Condition clauses (`when(create(E))`, `when(create(R))`) are used to distinguish between the cases where an entity or a relation was created. Finally, action sequences (encompassed into `seq`{} rules after the when-clauses) are used to instruct the VIATRA2 engine to instantiate the change history metamodel, create a *CreateEntity* or *CreateRelation* model element and set their references to the newly created host model entity/relation.

The right side of Fig. 5 shows an example execution sequence of this rule. The sequence starts with a model consisting only of a top-level container node $w0$ of type *Workflow*. In Step 2, the user creates a new *Invocation* node $i0$ inside $w0$. Note that on the VPM level, the creation of $i0$ actually consists of three operations: (1) create entity, (2) set entity type to *Invocation*, (3) move entity to its container. However, the live transformation engine triggers the execution of `handleCreation()` only if the subgraph $w0 - i0$ is complete. In Step 3, `handleCreation()` is fired with the match $\{Parent = w0, E = i0, Type = Invocation\}$, and – as the condition $create(E)$ is satisfied in this case – the appropriate *CreateEntity* instance $ce0$ is created.

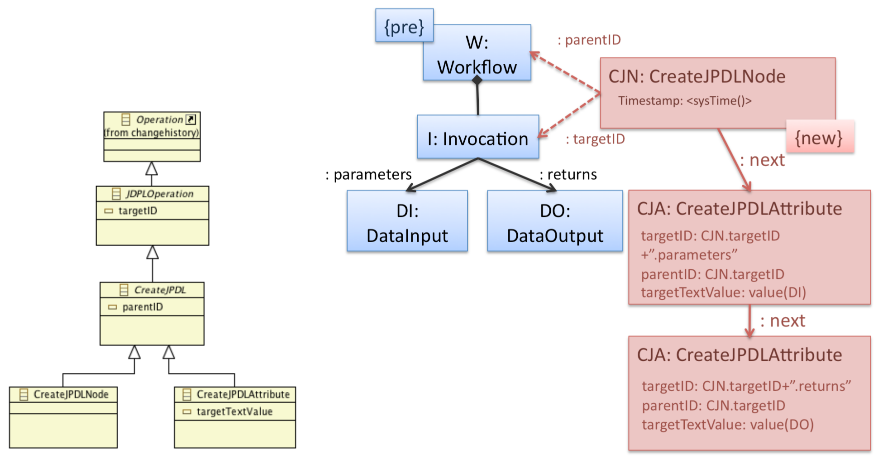
Domain-specific CHMs Change history models can also be defined on a higher abstraction level, directly applicable to domain-specific modeling languages. In Fig. 6(a),



```
@Trigger(sensitivity='rise')
// 'rise' indicates that the rule
// should fire for newly detected
// matches of the pattern
gtrule handleCreation()=
{
 precondition(E,Parent,R,S,T,Type)=
 {
  find entity_in_parent(E,Parent);
  find modelelement_type(E,Type);
 } OR {
  find relation_source_target(R,S,T);
  find modelelement_type(R,Type);
 }
 action {
  when(create(E)) seq {
   new CreateEntity(CE);
   setValue(CE.elementName,name(E));
   // store FQNs
   setValue(CE.targetFQN,fqn(E));
   setValue(CE.typeFQN,fqn(Type));
   setValue(CE.parentFQN,fqn(Parent));
   }
  when(create(R)) seq {
   new CreateRelation(CR);
   setValue(C.elementName,name(R));
   // store FQNs
   setValue(CE.targetFQN,fqn(R));
   setValue(CE.typeFQN,fqn(Type));
   setValue(CE.targetFQN,fqn(T));
   setValue(CE.sourceFQN,fqn(S));
}}}
```

Fig. 5. Live transformation rule for automatic CHM generation

a part of the change history metamodel for manipulating jPDL XML documents is shown. This metamodel uses unique *ID*s to refer to (non-materialized) model elements (as defined in the jPDL standard); since jPDL documents also follow a strict containment hierarchy, creation operations (as depicted in Fig. 6(a)) refer to a *parentID* in which an element is to be created. In the follow-up examples of our case study, we will make use of *CreateJPDLNode* and *CreateJPDLAttribute* to illustrate the usage of this domain-specific change history metamodel.



(a) Change history metamodel for jPDL

(b) Domain-specific CHM generation for jPDL

Fig. 6. Domain-specific change history models

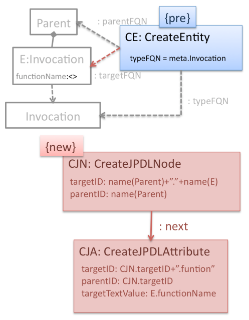
It is important to note, that domain-specific CHMs can be created analogously to generic ones, by using more complex graphs as precondition patterns for events. The domain-specific CHM construction rule in Fig. 6(b) includes direct type references to the domain metamodel (Fig. 1(c)) – in this case, it fires after the creation of an *Invocation* and associated *DataInput*s and *DataOutput*s is completed, and it creates connected three domain-specific CHM fragments accordingly.

4.2 Model transformations between change history models

As CHMs are automatically derived as models are modified, they essentially represent a sequence of operations that are valid starting from a given model snapshot (Fig. 2). As such, they may be used to drive mapping transformations between two modeling languages: such a change-driven transformation takes CHMs of the source model and maps them to CHMs of the target model.

This is a crucially different approach with respect to traditional model transformations in the sense that the mapping takes place between *model manipulation opera-*

tions rather than models, which makes non-invasive transformations possible (where the models are not required to be materialized in the transformation system).



```
gtrule mapCreation()=
{
 precondition(CE)=
 {
   CreatedEntity(CE);
   check (CE.typeFQN ==
            "metamodel.Invocation");
 }
 action {
   // calculate names from fqns
   let NameE = name(CE.targetFQN),
       NameParent = name(CE.parentFQN)
   in seq {
   // create CHM for JPDL Node
   new(CreateJPDLNode(CJN));
   setValue(CJN.targetID,
      NameParent+"."+NameE);
   setValue(CJN.parentID,
      NameParent);

   // create CHM for Attribute node
   new(CreateJPDLAttribute(CJA));
   // chain the operation sequence
   new(Operation.next(_,CJN,CJA));
   // set FQN references
   setValue(CJA.targetID,
      CJN.targetID+".function");
   setValue(CJA.parentID,
                    CJN.targetID);
   // copy information
   // from the source model
   setValue(CJA.targetValue,
      ref(CE.targetFQN).functionName);
}}}
```

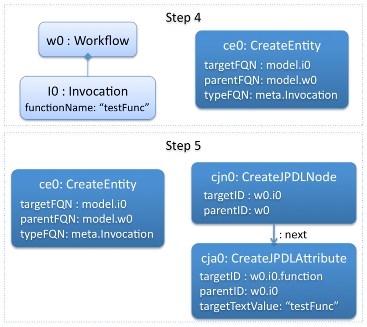


Fig. 7. Transformation of change history models

Fig. 7 shows an example transformation rule where the creation of an *Invocation* in the domain-specific workflow language is mapped to the creation of a corresponding jPDL Node and its attribute. In this case, a batch graph transformation rule is used, however, this transformation may also be formulated as a live transformation. The rule looks for a *CreateEntity* element referencing a node of type *Invocation*, and maps it to the domain-specific CHMs of the jPDL language. As *Invocation*s are represented by jPDL Nodes with an attribute node, the target CHM will consist of two "create"-type elements, chained together by the *Operation.next* relation.

The core idea of creating CHM transformations is the appropriate manipulation of reference values pointing to their respective host models (as CHMs only carry information on the type of the operation, the contextual information is stored in their references). In this example, we make use of the fact that both source and target models have a strict containment hierarchy (all elements have *parent*s), which is used to map corresponding elements to each other:

– Based on *parentFQN* in the source model, we calculate the target parent's ID *parentID* as $name(CE.parentFQN)$.
– Similarly, the target jPDL node's ID *targetID* is calculated as $parentID + "." + name(CE.targetFQN)$, which will place the target node under the target parent.
– Finally, the attribute $functionName$ designates a particular function on a remote interface which is invoked when the workflow engine is interpreting an *Invocation*

workflow node. It is represented by a separate node in the jPDL XML-DOM tree. The *targetValue* attribute of the additional *CreateJPDLAttribute* element is derived from the appropriate attribute value of *Invocation* node in source model (as denoted by the $ref(CE.targetFQN)$ function in the sample code).

The right side of Fig. 7 shows a sample execution result of the `mapCreate()` rule. First, in Step 4, the precondition pattern is matched, and a match is found to the subgraph created in Step 3 of Fig. 5. Following the successful matching, the action sequence is executed to create the domain-specific CHM nodes $cjn0$ (corresponding to a creation of a jPDL Node) and $cja0$ (creation of a jPDL attribute node). These CHM nodes are chained together by a *next* relation to be executed in sequence.

Designing change-driven transformations When designing transformations of change history models, it is important to focus on the fact that the transformation will operate on *operations* rather than models. Consequently, the first step in designing such a transformation is to define the concept of *operation* – which may be generic (graph-level operations), or domain-specific. This essentially requires a partitioning scheme for the host modeling language, where the partitions correspond to parts whose creation/deletion constitutes an operation which can be represented by a CHM fragment (and processed later on).

It is important to note that the *granularity* of this paritioning can be determined freely (since it is possible to perform the "aggregation" of operations in, e.g. the transformation between CHMs of source-target host languages); however, we have found that it is useful to define these partitions so that they represent a consistent change (i.e. the results of valid modification steps between two consistent states of the host model).

4.3 Processing change history models

On the macro level, change history models are represented as chains of parametrized elementary model manipulation operations. As such, they can be processed linearly, progressing along the chain until the final element is reached (thus modeling the execution of a transaction). The consumption (application) of a CHM element is an interpretative step, where the appropriate action is performed in the context defined by the CHM's references. Intuitively, the following rules outline interpretative actions for processing generic CHM type classes:

- *creation*: the target entity/relation is created with the correct type assignment; entities are created in the container designated by the parent's fully qualified name (`parentFQN`), relations are created between source and target elements referenced by `sourceFQN` and `targetFQN`, respectively.
- *moves*: for *MoveEntity*, the target entity is moved to the container designated by `newParentFQN`; for *SetRelationSource*, the source end of the target relation is redirected according to `newSourceFQN`.
- *updates*: *SetName* and *SetValue* are mapped to updates in the name and value attributes. *SetRelationTarget* is handled similarly to *SetRelationSource*.
- *deletions*: *DeleteEntity* and *DeleteRelation* are interpreted as deletions of their targets (`targetFQN`).

Applying CHMs to non-materialized models As Fig. 2 shows, we apply CHMs to manipulate non-materialized models through an interface. The speciality of this scenario is that instead of working on directly accessible in-memory models, the transformation engine calls interface functions which only allow basic queries (based on ids) and elementary manipulation operations. In this case, CHMs are very useful since they allow *incremental* updates, as they encode directly applicable operation sequences.

Case study technical details For the jPDL models of the motivating scenario, we mapped the XML-DOM process model manipulation programming interface to VIATRA2's *native function* API, which enables the system to invoke arbirary Java code from the transformation program. The following native functions are used:

- *getElementById(ID)*: queries the jPDL DOM for a given element, identified by its unique ID.
- *createElement(parentRef,targetID)*: creates a new jPDL DOM element as a child of its parent (identified by $parentRef$), with a given unique ID ($targetID$).
- *addElement(elementRef,DocID)*: adds the element $elementRef$ to the jPDL DOM identified by *DocID*.
- *setContents(elementRef,text)*: sets the textual content of the given DOM element ($elementRef$) to $text$.

In our examples, invocations of these interface functions are highlighted in green.

Example transformation rule In this final case study example, we define an application rule based on domain-specific CHMs for the jPDL XML-DOM model (Fig. 6(a)). Fig. 8 shows the `newCompoundJPDLNode()` rule, which is used to interpret a subsequence of CHM chains for the jPDL domain. More precisely, this rule's precondition matches the pair of *CreateJPDLNode* and *CreateJPDLAttribute* CHM fragments which correspond to the addition of a new "compound" jPDL node (with a specified function invocation attribute). The rule uses native functions `createElement`, `addElement` to instantiate new jPDL XML elements directly in the deployed process model on the workflow server; `setContent` is used to overwrite the attribute node's textual content.

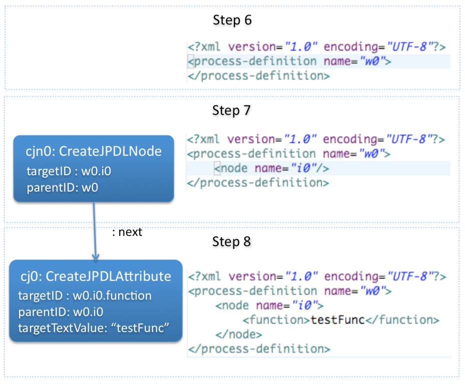
The left side of Fig. 8 shows the final three steps of our running example. In Step 6, the initial state of the deployed workflow model, the process definition corresponding to *Workflow* $w0$ is still empty. During the rule's execution, first, the jPDL Node $i0$ is created (Step 7), and then in Step 8, the attribute node is added with the appropriate textual content. (Debug calls are used to write debugging output to the VIATRA2 console.)

The entire algorithm which applies CHMs follows the linear sequnce of operations along the relations with type *Operation.next*; the first operation in a transaction can be determined by looking for a CHM fragment without an incoming *Operation.next* edge.

5 Related Work

Now an overview is given on various approaches showing similarity to our proposal.

Event-driven techniques. Event-driven techniques, which are the technological basis of live model transformations, have already been used in many other fields of



```xml
<?xml version="1.0" encoding="UTF-8"?>
<process-definition name="w0">
</process-definition>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<process-definition name="w0">
    <node name="i0"/>
</process-definition>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<process-definition name="w0">
    <node name="i0">
        <function>testFunc</function>
    </node>
</process-definition>
```

```
gtrule newCompoundJDPLNode(JPDL_DOM) =
{
 precondition(CJN,CJA) =
 {
  CreateJPDLNode(CJN);
  CreateJPDLAttribute(CJA);
  Operation.next(_,CJN,CJA);
 }
 action {
   // create JPDL Node -- Step 7
   let TargetNode = createElement(
                     getElementById(CJN.parentID),
                     CJN.targetID),
   Result0 = addElement(TargetNode, JPDL_DOM) in
   println("Debug created JPDL Node:"+Result0);
   // create JPDL Attribute -- Step 8
   let TargetAttributeNode = createElement(
                     getElementById(CJA.parentID),
                     CJA.targetID),
   Result1 = setContent(TargetAttributeNode,
                  ref(CJN.targetFQN).functionName),
   Result2 = addElement(TargetAttributeNode,
                          JPDL_DOM) in
   println("Debug created JPDL Attribute:"+Result2);
 }
}
```

Fig. 8. Applying CHMs through the jPDL XML-DOM API

computer engineering. In relational database management systems (RDBMS), even the concept of triggers [7] can be considered as simple operations whose execution is initiated by events. Later, event-condition-action (ECA) rules [8] were introduced for active database systems as a generalization of triggers, and the same idea was adopted in rule engines [9] as well. Specification of live model transformations is structurally and conceptually similar to ECA rules as discussed in Sec. 4.1. However, ECA-based approaches lack the support for triggering by complex graph patterns, which is an essential scenario in model-driven development.

Calculation of model differences. Calculating differences (deltas) of models has been widely studied due to its important role in the process of model editing, which requires undo and redo operations to be supported. In [10], metamodel independent algorithms are proposed for calculating directed deltas, which can later be merged with initial model to produce the resulting model. The approach of [10] generates both backward and forward directed deltas to support the calculation of previous and following models, respectively. This behaviour is not needed in synchronization scenarios as modifications caused by undo and redo tasks appear as regular create, delete, move, and update operations performed on the source model, which can be transformed by the same process to deltas on the target side. Unfortunately, the algorithms proposed by [10] for difference and merge calculation may only operate on a single model, and they are not specified by model transformation.

In [11], a metamodel independent approach is presented for visualizing backward and forward directed deltas between consecutive versions of models. Differences (i.e., change history models) have a model-based representation, and calculations are driven by (higher order) transformations in both [11] and our approach. However, in contrast to [11], our current proposal operates in an exogeneous transformation context, and thus, it is able to propagate change descriptions from source to target models.

Incremental synchronization for exogeneous model transformations. Incremental synchronization approaches already exist in model-to-model transformation context.

One representative direction is to use triple graph grammars [12] for maintaining the consistency of source and target models in a rule-based manner. The proposal of [13] relies on various heuristics of the correspondence structure. Dependencies between correspondence nodes are stored explicitly, which drives the incremental engine to undo an applied transformation rule in case of inconsistencies. Other triple graph grammar approaches for model synchronization (e.g. [14]) do not address incrementality.

Triple graph grammar techniques are also used in [15] for tool integration based on UML models. The aim of the approach is to provide support for change synchronization between various languages in several development phases. Based on an integration algorithm, the system merges changed models on user request. Although it is not a live transformation approach, it could benefit from being implemented as such.

The approach of [16] shows the largest similarity to our proposal as both (i) focus on change propagation in the context of model-to-model transformation, (ii) describe changes in a model-based and metamodel independent way, and (iii) use rule-driven algorithms for propagating changes of source models to the target side. In the proposal of [16] target model must be materialized and they can also be manually modified, which results in a complex merge operation to be performed to get the derived model. In contrast, our algorithms can be used on non-materialized target models, and the derived models are computed automatically on the target side.

6 Conclusion and Future Work

In the paper, we discussed how model synchronization can be carried out using change-driven model transformations, which rely upon the history of model changes. We presented an approach to automatically (and generically) derive change history models by recording changes in a (source) model using live transformations. Then a change history model of the target language is derived by a second (problem-specific) model transformation. Finally, the target change history model can automatically drive the incremental update of the target model itself even in such a case when only an external model manipulation interface is available for the target model. Our approach was examplified using an incremental code generation case study.

As future work, we plan to investigate how to derive aggregated and history independent change delta models (like in [11]) automatically as union of change history models. Furthermore, we aim at using change history models in the context of model merging.

Acknowledgements: This paper was partially supported by the SENSORIA European Project (IST-3-016004).

References

1. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems. Volume 4199 of Lecture Notes in Computer Science., Genova, Italy, Springer (October 2006) 321–335

2. Rth, I., Bergmann, G., krs, A., Varr, D.: Live model transformations driven by incremental pattern matching. In: Theory and Practice of Model Transformations. Volume 5063/2008 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2008) 107–121
3. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming **68**(3) (October 2007) 214–234
4. Varró, D., Pataricza, A.: VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling **2**(3) (October 2003) 187–210
5. The Eclipse Project: Eclipse Modeling Framework `http://www.eclipse.org/emf`.
6. Ehrig, H., Montanari, U., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation. Volume 3: Concurrency and Distribution. World Scientific (1999)
7. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book. Prentice Hall (2001)
8. Dittrich, K.R., Gatziu, S., Geppert, A.: The active database management system manifesto: A rulebase of ADBMS features. In Sellis, T., ed.: Proc. of the 2nd International Workshop on Rules in Database Systems. Volume 985 of Lecture Notes in Computer Science., Glyfada, Athens, Greece, Springer (September 1995) 1–17
9. Seiriö, M., Berndtsson, M.: Design and implementation of an ECA rule markup language. In Adi, A., Stoutenburg, S., Tabet, S., eds.: Proc. of the 1st International Conference on Rules and Rule Markup Languages for the Semantic Web. Volume 3791 of Lecture Notes in Computer Science., Galway, Ireland, Springer (October 2005) 98–112
10. Alanen, M., Porres, I.: Difference and union of models. In Stevens, P., Whittle, J., Booch, G., eds.: Proc. of the 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML 2003). Volume 2863 of Lecture Notes in Computer Science., San Francisco, California, USA, Springer (October 2003) 2–17
11. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: A metamodel independent approach to difference representation. Journal of Object Technology **6**(9) (October 2007) 165–185
12. Schürr, A.: Specification of graph translators with triple graph grammars. Technical report, RWTH Aachen, Fachgruppe Informatik, Germany (1994)
13. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. of 9th International Conference on Model Driven Engineering Languages and Systems, (MoDELS 2006). Volume 4199 of LNCS., Springer (2006) 543–557
14. Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, New York, NY, USA, ACM (2007) 285–294
15. Simon M. Becker, Thomas Haase, B.W.: Model-based a-posteriori integration of engineering tools for incremental development processes. Software and Systems Modeling **4**(2) (May 2005) 123–140
16. Jimenez, A.M.: Change propagation in the MDA: A model merging approach. Master's thesis, The University of Queensland (June 2005)