Turning Models Inside Out

Alfa Yohannis, Dimitris Kolovos, Fiona Polack
Department of Computer Science
University of York
York, United Kingdom
{ary506, dimitris.kolovos, fiona.pollack}@york.ac.uk

***Abstract*— We present an approach for change-based (as opposed to state-based) model persistence that can facilitate high-performance incremental model processing (e.g. validation, transformation) by minimising the cost of change identification when models evolve. We illustrate a prototype that implements the proposed approach on top of the Eclipse Modelling Framework and we present a roadmap for further work in this direction.**

***Keywords*—model-driven, persistence, incremental, change events, notification**

I. Introduction

To reap the benefits of Model-Based Software Engineering in the context of large and complex systems, the ability to process large models in an incremental fashion as they evolve is essential. Current incremental model processing techniques only deliver limited performance benefits due to slow and imprecise model change detection capabilities or are limited to a single-developer environment; not realistic for real-world software development projects.

The research introduced in this paper aims at enabling flexible and high-performance incremental model processing through change-based model persistence. That is, instead of persisting snapshots of the state of models, we propose turning models inside out and persisting their change history. The proposed approach has the potential to deliver step-change performance benefits in incremental model processing, as well as a wide range of other benefits and novel capabilities.

The rest of the paper is structured as follows. Section II reviews the key-challenge of incrementality in MBSE. Section III provides an overview of existing approaches for identifying changes in models. Section IV overviews our proposed approach and Section V discusses our prototype implementation on top of the Eclipse Modelling Framework. The potential benefits and novel capabilities, as well as the challenges of change-based model persistence, are presented in Sect. VI and Sect. VII respectively. Section VIII presents our evaluation strategy and Sect. IX concludes this paper.

II. The Key-Challenge of Incrementality

To illustrate the concept of incrementality, a contrived running example is used where after every modification to an organisational chart model (see Fig. 1), the model needs to be:

- Validated against a domain-specific constraint (that no employee directly manages more than 7 other employees).
- Transformed into a number of employee reports (plain text files, one for each employee) through a model-to-text transformation. Each report should contain the name of the employee, and the names of her direct subordinates.

Figures 1 and 2 are two consecutive versions of a sample organisational chart model. When the validation constraint is evaluated against the first version of the model (Fig. 1), it verifies that all three employees manage fewer than 7 other employees, and the model-to-text transformation then produces three text files that correspond to the employees in the model. In the sequel, in Fig. 2, the model is updated to reflect that a new employee has been hired (Richmond) under the management of Jen.

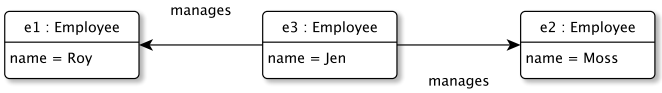

Fig. 1. Initial version of the organisational chart model.

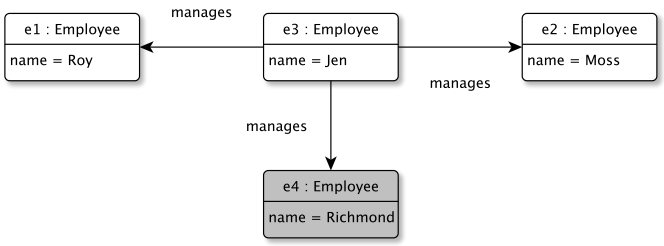

Fig. 2. Modified version of the organisational chart model of Fig. 1.

A non-incremental model validation engine would treat the model of Fig. 2 as if it was a new model and would evaluate the constraint above against every employee in the model. An incremental model validation engine, on the other hand, would identify that the previously established satisfaction of the constraint for employees Moss and Roy cannot have been possibly compromised by the changes made, and would only re-evaluate the constraint for Jen and Richmond instead.

Similarly, a non-incremental model-to-text transformation would generate and overwrite all employee reports from scratch. On the contrary, an incremental model-to-text transformation, would identify that it only needs to generate a new report for the new employee (Richmond), and to recompute and overwrite the contents of Jen's report (as she is now

managing an additional employee)—but not the reports of Moss or Roy, as these cannot have been affected by the changes made to the model.

While the overhead of executing transformations and validation constraints on small models like the one in Fig. 2 is negligible, non-incremental execution can become a significant bottleneck for large evolving models. As stressed in Selic's seminal work [1], with reference to model-to-text transformation, "*... this is particularly true in the latter phases of the development cycle when programmers make many small changes as they fine-tune the system. To keep this overhead low, it is crucial for the code generators to have sophisticated change impact analysis capabilities that minimize the amount of code regeneration*".

As demonstrated by the pioneering work of Egyed [2], to achieve incremental re-execution of (deterministic) queries on structured models, an execution engine needs to:

1) Record model element property accesses during the initial execution of the queries;
2) Identify new and deleted elements and modified model element properties in the new version of the model;
3) Combine the information collected in the steps above to identify the subset of (potentially) affected rules/-queries/templates that need to be re-executed.

To illustrate this, we use an OCL implementation of the domain-specific constraint in List. 1.

Listing 1. OCL constraint requiring that no employee directly manages more than 7 other employees.

```
1 context Employee
2 inv NoMoreThan7: self.manages->size()
      <= 7
```

During the initial evaluation of the constraint on the model of Fig. 1, an incremental OCL engine would compute the property access trace displayed in Table I as a side-product. Now, when the model is updated (Fig. 2), the execution engine can identify that:

- There is new element in the model (*e4* - Richmond) for which the constraint has not been evaluated;
- The value of the manages property of Jen (*e3*) has changed, and as such, it needs to re-evaluate the constraint on this model-element.

TABLE I
PROPERTY-ACCESS TRACE OF THE EVALUATION OF THE CONSTRAINT IN LIST. 1 ON THE MODEL OF FIG. 1.

Constraint	Context	Accessed Element	Accessed Property
Employee.NoMoreThan7	*e1*	*e1*	*manages*
Employee.NoMoreThan7	*e2*	*e2*	*manages*
Employee.NoMoreThan7	*e3*	*e3*	*manages*

Egyed has shown that the property-access recording approach is applicable to queries of arbitrary complexity, as long as they are deterministic. More recent work has shown that variants of this approach can be used to achieve incrementality in a wide range of model processing operations, including model-to-model transformation [3], model-to-text transformation [4], model validation, and pattern matching [5]—as long as changes to models can be precisely identified (step 2 in the list above).

III. Identifying Changes in Models

There are two approaches in the literature for identifying changes in models in order to enable incremental re-execution of model processing operations.

Notifications. In this approach, the incremental execution engine needs to hook into the notification facilities provided by the modelling tool through which the developer edits the model, so that the engine can directly receive notifications as soon as changes happen (e.g. a new employee (*e4*) has been added, the name property of employee *e4* has been changed to "Richmond"). This is an approach taken by the IncQuery incremental pattern matching framework [5] and the ReactiveATL incremental model-to-model transformation engine [4]. The main advantage of this approach is that precise and fine-grained change notifications are provided for free by the modelling tool (and thus do not need to be computed by the execution engine—which as discussed below can be expensive and inefficient). On the downside, this approach is a poor fit for collaborative development settings where modelling and automated model processing activities are performed by different members of the team.

Model Differencing. This approach eliminates the coupling between modelling tools and incremental execution engines. Instead of depending on live notifications, in this approach the developer in charge of automated model processing, needs to have access to a copy of the last version of the model that the model processing program (e.g. the model-to-text transformation) was executed upon, so that it can be compared against the current version of the model (e.g. using a model-differencing framework such as SiDiff [6] or EMFCompare[1]) and the delta can be computed on demand. The main advantage of this approach is that it works well in a collaborative development environment where typically developers have distinct roles and responsibilities. On the downside, model comparison and differencing are computationally expensive and memory-greedy (both versions of the model need to be loaded into memory before they can be compared), thus largely undermining the time and resource saving potentials of incremental re-execution. This approach is adopted by the Xpand model-to-text transformation language. According to the developers of the language, using this approach, a speed-up of only around 50% is observed compared to non-incremental transformation[2], which is consistent with our experience from using Xpand.

In summary, incremental model processing currently delivers significant performance benefits only in a single-developer environment where the modeller is also responsible for performing all the (incremental) model processing operations.

[1] https://www.eclipse.org/emf/compare/

[2] http://wiki.eclipse.org/Xpand/New_And_Noteworthy#Incremental_Generation

Listing 2. Change-based representation of the model of Figure 2.

```
1  <session id="s1"/>
2  <create eclass="Employee" epackage="employee" id="0"/>
3  <add-to-resource position="0"><value eobject="0"/></add-to-resource>
4  <set-eattribute name="name" target="0"><value literal="Roy"/></set-eattribute>
5  <create eclass="Employee" epackage="employee" id="1"/>
6  <add-to-resource position="1"><value eobject="1"/></add-to-resource>
7  <set-eattribute name="name" target="1"><value literal="Jen"/></set-eattribute>
8  <create eclass="Employee" epackage="employee" id="2"/>
9  <add-to-resource position="2"><value eobject="2"/></add-to-resource>
10 <set-eattribute name="name" target="1"><value literal="Moss"/></set-eattribute>
11 <remove-from-resource><value eobject="0"/></remove-from-resource>
12 <add-to-ereference name="manages" position="0" target="1"><value eobject="0"/></add-to-ereference>
13 <remove-from-resource><value eobject="2"/></remove-from-resource>
14 <add-to-ereference name="manages" position="1" target="1"><value eobject="2"/></add-to-ereference>
15 <session id="s2"/>
16 <create eclass="Employee" epackage="employee" id="3"/>
17 <add-to-resource position="1"><value eobject="3"/></add-to-resource>
18 <set-eattribute name="name" target="3"><value literal="Richmond"/></set-eattribute>
19 <remove-from-resource><value eobject="3"/></remove-from-resource>
20 <add-to-ereference name="manages" position="2" target="2"><value eobject="3"/></add-to-ereference>
```

As a result, in collaborative development environments, developers need to either forgo incremental model processing altogether or to work around this limitation by manually steering model processing programs to process only subsets of their models, which is cumbersome and error prone.

IV. Proposed Approach

The ambition of this research is to enable high-performance incremental model management in collaborative software development environments by challenging one of the fundamental assumptions of contemporary modelling frameworks and tools: as opposed to persisting snapshots of the state of models (which is what virtually all modelling tools and frameworks currently do), we propose turning models inside out and persisting their change history instead.

Listing 3. State-based representation of the model of Figure 2 in (simplified) XMI.

```
1  <Employee xmi:id="e2" name="Jen">
2    <manages xmi:id="e1" name="Roy"/>
3    <manages xmi:id="e3" name="Moss"/>
4    <manages xmi:id="e4" name="Richmond"/>
5  </Employee>
```

To illustrate the proposed approach, List. 3 shows a state-based representation of the model of Fig. 2 in (simplified) XMI, and List. 2 shows the proposed equivalent change-based representation of the same model. Instead of a snapshot of the state of the model, the representation of List. 2 captures the complete sequence of change events (create/set/add/remove/delete) that were performed on the model since its creation, organised in editing sessions (2 editing sessions in the case of this model). Replaying these changes produces the same state as the one captured in List. 3, so the proposed representation carries at least as much information as the state-based representation.

Such a representation is particularly suitable for incremental model processing. For example, if the model-to-text transformation discussed above "remembers" that in its previous invocation it had processed up to editing session *s1* of the model, it can readily identify the changes that have been made to the model since then (i.e. in session *s2* - lines 15-20) instead of having to rediscover them through (expensive) state-based model differencing.

V. Prototype Implementation

We have implemented a prototype[3] of the change-based model persistence format using the notification facilities provided by the Eclipse Modelling Framework. In our implementation we use the *ChangeEventAdapter* class, a subclass of EMF's *EContentAdapter*[4], to receive and record *Notification*[5] events produced by the framework for every model-element level change.

Since not all change events are relevant to change-based persistence (e.g. EMF also produces change notifications when listeners/adapted are added/removed from the model), we have defined a set of event classes to represent events of interest. The event classes are depicted in Fig. 3 as subclasses of the *ChangeEvent* abstract class.

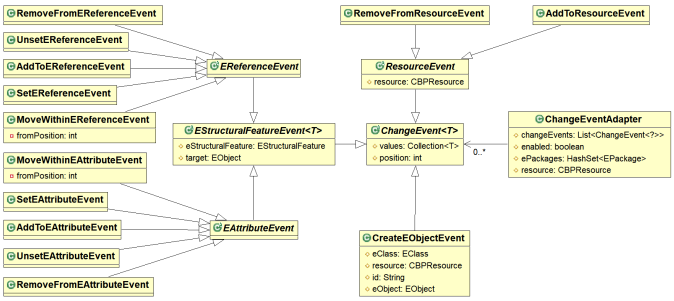

Fig. 3. Event classes to represent changes of models.

The *ChangeEvent* class has a mutli-valued *values* attribute which can accommodate both single-valued (e.g. set/add) or mutli-valued events (e.g. addAll/removeAll). *ChangeEvent* can

[3]The prototype is available under https://github.com/epsilonlabs/emf-cbp.
[4]http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/ecore/util/EContentAdapter.html
[5]http://download.eclipse.org/modeling/emf/emf/javadoc/2.11/org/eclipse/emf/common/notify/Notification.html

also accommodate different types of values, such as *EObject*s for *EReferenceEvents*, and primitive values (e.g. Integer, String) for *EAttributeEvents*. The *ChangeEvent* class also has a position attribute to hold the index of an *EObject* or a literal when they are added to a *Resource*, *EReference*, or *EAttribute* with multiple values (Lst. 2, line 3, 6, 9, 12, 14, 17, 20).

Every time an *EObject* is added to the model, a *CreateEObjectEvent* and an *AddToResourceEvent* are recorded (lines 2-3, 5-6, 8-9, and 16-17 in Lst. 2). When an EObject is deleted, or moved to a containment *EReference* deeper in the model (Lst. 2, line 12, 14, 20), a *RemoveFromResourceEvent* (Lst. 2, line 11, 13, 19) is recorded.

The *ChangeEventAdapter* receives EMF change notifications in its *notifyChanged()* method and filters and transforms them into appropriate change events. As an example of how notifications are filtered and transformed, Listing 4 shows how we handle *Notification.UNSET* events based on the type of the changed feature i.e. an *UnsetEAttributeEvent* is instantiated if the feature of the notifier is an *EAttribute*, or an *UnsetEReferenceEvent* is created if the notifier is an *EReference*. The transformed instances are then stored into a list of events in *ChangeEventAdapter* (*changeEvents*) for persistence.

Listing 4. Simplified Java code to handle notification events.

```
1  public class ChangeEventAdapter extends EContentAdapter
       {
2  ...
3  @override
4  public void notifyChanged(Notification n) {
5    ...
6    switch (n.getEventType()) {
7      ... // other events
8      case Notification.UNSET: {
9      if (n.getNotifier() instanceof EObject) {
10       EStructuralFeature feature = (EStructuralFeature)
             n.getFeature();
11         if (feature instanceof EAttribute) {
12           event = new UnsetEAttributeEvent();
13         } else if (feature instanceof EReference) {
14           event = new UnsetEReferenceEvent();
15         }
16       } break;
17     }
18     ... // other events
```

To integrate seamlessly with the EMF framework and to eventually support multiple concrete change-based serialisation formats (e.g. XML-formatted representation for readability and binary for performance/size), we have created the *CBPResource* abstract class, that extends EMF's built-in *ResourceImpl* class. The role of the abstract class is to encapsulate all change recording functionality while the role of its concrete subclasses is to implement serialisation and de-serialisation. For example, *CBPXMLResourceImpl* persists changes in a line-based format where every change is serialised as a single-line XML document. In this way, when a model changes, we can append the new changes to the end of the model file without needing to serialise the entire model again. We have also implemented a *CBPXMLResourceFactory* class that extends EMF's *ResourceFactoryImpl*, as the factory class for change-based models. Figure 4 shows the relationships between these classes.

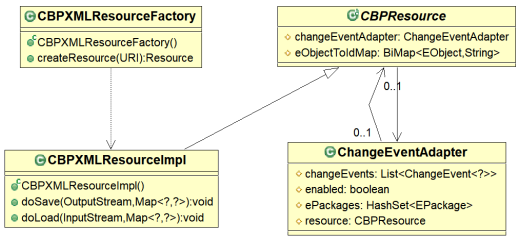

Fig. 4. Factory, resources, and ChangeEventAdapter classes.

VI. Benefits and Novel Capabilities

Beyond facilitating incremental processing, the proposed representation also has the potential to deliver a wide range of benefits and novel capabilities, compared to the currently prevalent state-based representations, some of which are discussed below.

- With appropriate tool support, modellers will be able to "replay" (part of) the change history of a model (e.g. to understand design decisions made by other developers, for training purposes). In state-based approaches, this can be partly achieved if models are stored in a version-control repository (e.g. Git). However, the granularity would only be at the commit level.
- By analysing models serialised in the proposed representation, modelling language and tool vendors will be able to develop deeper insights into how modellers actually use these languages/tools in practice and utilise this information to guide the evolution of the language/tool.
- By attaching additional information to each session (e.g. the id of the developer, references to external documents/URLs), sequences of changes can be traced back to the developer that made them, or to requirements/bug reports that triggered them.
- Persisting changes to large models after an editing session will be significantly faster compared to serialising the entire state of the model, as only changes made during the session will need to be appended to the model file.
- The performance and precision of model comparison and merging can be substantially improved, particularly for large models with shared editing histories.

VII. Challenges and Future Work

The proposed approach also comes with a number of challenges that this research will need to overcome.

Loading Overhead. While, as discussed above, persisting changes to large models is expected to be much faster and resource-efficient compared to state-based approaches, loading models into memory by naively replaying the entire change history is expected to have a significant overhead. To address this challenge, we will develop dedicated algorithms and data structures that will reduce the cost of change-based model loading (e.g. by recording and ignoring events – events that are later overridden or cancelled out by other events).

Fast-Growing Model Files. Persisting models in a change-based format means that model files will keep growing in

size during their evolution significantly faster than their state-based counterparts. To address this challenge, (1) we will propose sound change-compression operations (e.g. remove older/unused information) that can be used to reduce the size of a model in a controlled way. (2) We will develop a compact textual format that will minimise the amount of space required to record a change (a textual line-separated format is desirable to maintain compatibility with file-based version control systems). (3) We will propose a hybrid model persistence format which will be able to incorporate both change-based and state-based information.

VIII. Evaluation Strategy

The findings of the research will be evaluated in the small in the context of the tasks in which they will be developed, and in the large through industrial case studies. For the first type of evaluation (in the small), where there are existing approaches that the algorithms and tools developed in this research seek to outperform (e.g. change-based incremental validation vs. state-based incremental validation), comparative evaluation will be conducted to assess the benefits and limitations of our approaches. For algorithms and tools that have no direct competitors in the literature, their contributions will be assessed in comparison to the baseline they seek to improve (e.g. in this case, persisting full change histories).

IX. Conclusions

Through turning models inside out and persisting their change history, this research aims at enabling high-performance incremental model processing in collaborative development settings. The proposed approach also has the potential to enable model analytics, more fine-grained tracing and to improve the precision and performance of model comparison and merging. A prototype implementation of a change-based persistence format has been presented, the main envisioned challenges have been listed and an evaluation strategy has been outlined.

Acknowledgments

This work was partly supported through a scholarship managed by *Lembaga Pengelola Dana Pendidikan Indonesia* (Indonesia Endowment Fund for Education).

References

[1] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[2] A. Egyed, "Automatically detecting and tracking inconsistencies in software design models," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 188–204, 2011.

[3] F. Jouault and M. Tisi, "Towards incremental execution of atl transformations." *ICMT*, vol. 10, pp. 123–137, 2010.

[4] B. Ogunyomi, L. M. Rose, and D. S. Kolovos, "Property access traces for source incremental model-to-text transformation," in *European Conference on Modelling Foundations and Applications*. Springer, 2015, pp. 187–202.

[5] I. Ráth, Á. Hegedüs, and D. Varró, "Derived features for emf by integrating advanced model queries," *Modelling Foundations and Applications*, pp. 102–117, 2012.

[6] U. Kelter, J. Wehren, and J. Niere, "A generic difference algorithm for uml models." *Software Engineering*, vol. 64, no. 105-116, pp. 4–9, 2005.