Enhanced Automation for Managing Model and Metamodel Inconsistency
Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack
Department of Computer Science
The University of York
York, YO10 5DD
United Kingdom
Email: {louis,dkolovos,paige,fiona}@cs.york.ac.uk

Abstract—Model-Driven Engineering (MDE) introduces additional challenges for managing evolution. For example, a metamodel change may affect instance models. Existing tool supported approaches for updating models in response to a metamodel change assume extra effort from metamodel developers. When no existing approach is applicable, metamodel users must update their models manually, an error prone and tedious task.

In this paper, we describe the technical challenges faced when using the Eclipse Modeling Framework (EMF) and existing approaches for updating models in response to a metamodel change. We then motivate and describe alternative techniques, including: a mechanism for loading, storing and manipulating inconsistent models; a mapping of inconsistent models to a human-usable notation for semi-automated and collaborative co-evolution; and integration with an inter-model reference manager, achieving automatic consistency checking as part of metamodel distribution.

1 INTRODUCTION
Model-Driven Engineering (MDE) is an approach to software development that systematically employs models as first-class citizens for constructing and maintaining software systems. A model is a description of a phenomenon of interest [1] which enables engineers of differing disciplines to reason about that phenomenon [2], and may have several concrete representations (textual or graphical). A model must have a well-defined set of structural elements and rules that it must obey. This information can be specified in a structured artefact termed a metamodel.

Today, metamodel developers can automatically generate tools for creating, editing and storing instance models [3], and graphical editors for manipulating instance models [4]. Instance models can be managed using a variety of model management operations, such as: transformation to instances of other metamodels [5], transformation to code or other text [6], and validation against a set of constraints. There is limited support, however, for model migration - a development activity in which instance models are updated in response to metamodel evolution. More generally, MDE introduces additional challenges for controlling and managing software evolution [7].

When a metamodel evolves (is adapted by a developer), instance models might no longer conform to the structures and rules defined by the metamodel. Until the instance models conform to the evolved metamodel, they cannot be manipulated with generated editors and cannot be managed with model management operations.

In Section 2, we discuss the way in which characteristics of the Eclipse Modeling Framework (EMF) [3], arguably the most widely used modelling framework, restrict the ways in which model migration can be performed. In Section 3, we describe the technical challenges faced when using EMF to evolve a metamodel, and the limitations of existing approaches for managing model and metamodel co-evolution. We then motivate and describe alternatives to existing model migration approaches, including: a mechanism that permits metamodel users to load, manipulate and store models that no longer conform to their metamodel; a mapping from model to a human-usable notation for semi-automated and collaborative co-evolution by metamodel users; and integration with an inter-model reference manager, achieving automatic consistency checking as part of metamodel distribution.

2 BACKGROUND
In this section, we describe two key challenges faced when using the Eclipse Modeling Framework (EMF) to evolve a metamodel and subsequently migrate instance models. We then discuss the limitations of existing approaches for automating model migration.

2.1 Model and Metamodel Separation
Like any other Eclipse development tool, EMF metamodels are distributed as plug-ins, which are re-usable workbench components. Eclipse users install, configure
and combine plug-ins to tailor their development environment. The developers of the Eclipse Java Development Tools (JDT) have no access to the Java programs written in downstream workbenches, and similarly an EMF metamodel developer has no programmatic access to downstream instance models. Because of this, metamodel evolution occurs independently to model migration. First, the metamodel is evolved. Subsequently, metamodel users find that their models are out-of-date and migrate their models. This process is facilitated when, during metamodel evolution, the metamodel developer devises and codifies a migration strategy, which is distributed to the workspaces that contain instance models. The metamodel user then executes the migration strategy to migrate inconsistent models.

Model migration is distinct from the more general activity of model-to-model transformation [8]. As such, dedicated structures and processes are required for the effective management of model migration; fulfilling this requirement remains an open research problem.

2.2 Model and Metamodel Consistency in EMF

A model and metamodel are consistent when the metamodel specifies every concept used in the model definition, and the model uses the metamodel concepts according to the rules specified by the metamodel. Consistency can be described by a set of constraints between models and metamodels [9]. When all constraints are satisfied, a model and a metamodel are consistent. For example, a constraint might enforce that every object in the model has a corresponding non-abstract class in the metamodel. Consistency is normally a prerequisite of loading, editing, and managing models.

A metamodel can evolve (be adapted by a developer), which can cause inconsistency. For example, suppose a concept is removed from a metamodel. Any models that use the removed concept are now inconsistent with the metamodel. Model and metamodel co-evolution (subsequently referred to as simply co-evolution) is the process of evolving both metamodel and model such that they remain consistent.

Model and metamodel consistency is implicitly enforced by EMF. A model is bound to its metamodel, typically by constructing a Java object for each model element and data value. Mappings between Java and metamodel types are included in the EMF metamodel definition. EMF does not permit changes to a model that would cause inconsistency with its metamodel, and the deserialisation of an inconsistent model causes an error. EMF cannot be used to manage inconsistent models.


2.3 Limitations of Existing Approaches

Because EMF cannot be used to load inconsistent models, existing approaches for automating co-evolution include in each metamodel plug-in all previous metamodel versions (hence providing EMF with valid bindings for older models) and executable migration strategies for older metamodel versions. As we now discuss, the way in which migration strategies are determined varies between existing co-evolution approaches.

In [13], we compare and contrast existing co-evolution approaches using two categories, operator-based and metamodel matching approaches. Operator-based approaches, such as COPE [14], provide built-in operators that specify a metamodel evolution and corresponding model migration. Each operator targets a commonly occurring co-evolution, such as extracting a class. In situations where no co-evolutionary operator is suitable, the metamodel developer must manually specify a migration strategy. Hence, the usefulness of an operator-based co-evolution approach is heavily influenced by both the richness and the navigability of its library of co-evolutionary operators. In metamodel matching approaches, such as [15], [16], metamodel versions are compared to determine differences, and then a corresponding model migration strategy is automatically inferred. Although metamodel matching approaches require less effort from the metamodel developer than operator-based approaches, in general metamodel matching approaches cannot be guaranteed to infer a migration strategy that captures the semantics desired by the metamodel developer. In particular, Gruschko et al. [17] suggest inferring migration strategies from metamodel changes but show that some metamodel changes cannot be resolved in a fully-automated manner; developer input is required.

3 Typical Scenario

To motivate our work, we now present a scenario that highlights some of the challenges faced when using EMF and existing co-evolution approaches to detect and reconcile model and metamodel inconsistencies.

Imagine that Mark is using EMF to develop a metamodel. Members of his team, including Heather, will be using the metamodel to construct models. Heather installs the plug-in for Mark’s metamodel and begins constructing instance models. Mark later identifies new requirements, changes the metamodel, builds a new version of the metamodel plug-in, and distributes it to Heather.

After several iterations of metamodel updates, Heather tries to load one of her older models, constructed using an earlier metamodel. When loading the older model, EMF reports an error indicating that the model is no longer consistent with its metamodel. To load the older model, Heather must reinstall an old version of the metamodel plug-in that contains the metamodel that is consistent with the older model. But even

[CORE]
then, EMF will bind the older model to the old version of the metamodel, and not to the evolved metamodel. Worse still, Eclipse does not allow two versions of the same plug-in to co-exist in the same workspace, so the evolved metamodel cannot be used.

In practice, a common solution is for Heather to trace and repair the loading error directly in XMI, the underlying format of the model. Human usability was not a key requirement for XMI [18] and, consequently, using XMI for migration is an unproductive and tedious task. Worse still, as EMF only reports the first problem encountered when deserialising an inconsistent model, re-establishing consistency in XMI is a slow, iterative process.

To avoid the loading errors in Heather’s workspace, Mark could use an existing tool for managing co-evolution. However, all existing tools require extra effort from Mark. Operator-based approaches require him to evolve his metamodel using a specialised tool that records changes and applies co-evolutionary operators. Although metamodel matching approaches can automatically infer a migration strategy, they cannot be guaranteed to produce a migration strategy that capture the semantics that Mark intended.

When Mark makes a metamodel change that causes inconsistency in only a small number of models, he might decide that the extra effort required to use an existing co-evolution tool is too great. However, if Mark does not provide Heather with a metamodel plug-in that can load and migrate older models, Heather must manually migrate inconsistent models using XMI.

4 Contributions

In summary, EMF binds a model to its metamodel, and hence metamodel users cannot manage models that are no longer consistent with their metamodel nor determine the consistency of a model with any other metamodel version. Existing approaches for managing co-evolution avoid this problem, but typically do not scale down - they are not effective for managing few inconsistencies.

In this paper, we present an alternative binding that can be used to load inconsistent models with EMF even in the absence of old metamodel versions (Section 4.1). The usefulness of the alternative binding is demonstrated in our solutions to the following practical problems:

1) Performing manual migration with XMI is error prone and tedious. We present a semi-automated approach for managing small numbers of inconsistencies in Section 4.2, which maps inconsistent models to a human-readable notation.

2) Collaborative reconciliation of inconsistencies is difficult because existing approaches for managing co-evolution do not provide mechanisms for storing (partially) inconsistent models. The binding discussed in Section 4.1 and the notation discussed in Section 4.2 can both be used to represent, store and exchange inconsistent models, enabling collaborative reconciliation of inconsistencies.

3) Installing a new version of a metamodel plug-in can cause models to become inconsistent. Inconsistency is not reported to the user as part of the installation process. In Section 4.3, we discuss our implementation of a tool that reacts to the plug-in installation process and automatically produces a consistency report for all affected models.

We now discuss the alternative binding for EMF. The remainder of this section contains, for each solution, a detailed discussion and a description of our implementation and the way in which it uses the alternative binding.

4.1 Binding to a Generic Metamodel

We have implemented an alternative deserialisation mechanism for EMF that binds a model to a generic metamodel. This generic metamodel reflects the characteristics of the metamodeling language and consequently every model is consistent with the generic metamodel. Figure 1 shows a minimal version of a generic metamodel for EMF. Model elements are bound to Object, data values to Slot.

Using the metamodel in Figure 1 in conjunction with Ecore (the metamodeling language of EMF), consistency constraints between a model and a metamodel can be expressed, as shown below. A minimal subset of Ecore is shown in Figure 2.

1) Each object’s type value must be the name of some metamodel class.
2) Each object’s type value must be the name of some non-abstract metamodel class.
3) Each object must specify a slot for each mandatory feature of its type.
4) Each slot’s feature value must be the name of a metamodel feature. That metamodel feature must belong to the slot’s owning object’s type.
5) Each slot must be multiplicity-compatible with its feature. More specifically, each slot must contain at least as many values as its feature’s lowerbound,
and at most as many values as its feature’s upper-bound.

6) Each slot must be type-compatible with its feature.

To check consistency between a model and a metamodel, we have implemented the above constraints using the Object Constraint Language (OCL).

4.2 Semi-Automated Consistency Repairing

If a model has been found to be inconsistent with its metamodel, we provide tooling for transforming it from XMI to HUTN. HUTN (Human Usable Textual Notation) [18], [19] is an OMG standard metamodel-independent syntax, which is optimised for human editing (unlike XMI). When transforming XMI to HUTN, error markers are generated to highlight inconsistent elements. In this way, developers can perform the necessary changes using a human-friendly concrete syntax. Moreover, as HUTN is metamodel-independent, developers can choose to only repair some inconsistencies, save the model and resume later, or even share it with another user, who may know better how to reconcile the remaining inconsistencies.

4.3 Automatic Consistency Checking

As discussed in Section 2.2, metamodel developers do not have access to downstream models. Consequently, instances of a metamodel can become inconsistent after a new version of a metamodel plug-in is installed. By default, an EMF metamodel plug-in does not check consistency during plug-in installation and inconsistent instance models are only detected when loaded.

To improve metamodel installation, we have integrated the binding to a generic metamodel discussed in Section 4.1 with Concordance [20], which provides a light-weight and efficient mechanism for resolving inter-model references. We use Concordance to monitor the models in the workspace and, for each model, to maintain a reference to its metamodel.

After plug-in installation, Concordance computes a hash for each available metamodel and compares it to a previously cached value. A hash that differs from its cached value indicates a metamodel evolution, and all instance models are checked for consistency. Consequently, consistency checking happens automatically and during metamodel installation. Models are marked inconsistent immediately rather than when next loaded by EMF. By integrating the consistency checking with Concordance, increased scalability is achieved, as models are only checked when their metamodel has evolved.

5 EXAMPLE

To demonstrate our contributions, we use the metamodel evolution shown in Figures 3 and Figure 4. In Figure 3, a System is composed of Channels and ConnectionPoints. A Channel reads from and writes to exactly one ConnectionPoint.

![Fig. 3. Exemplar metamodel prior to evolution.](image)

Consider the following description of a model, which is consistent with the metamodel defined in Figure 3. An office system comprises two channels, photocopier and fax machine, and four connection points, scanner, printer, telephone and unused. The photocopier has scanner as its reader and printer as its writer, while the fax machine has telephone as both its reader and its writer.

Suppose that further analysis of the domain reveals that ConnectionPoints may only be used for reading or for writing, and never both. Therefore, ConnectionPoint becomes abstract, and two subtypes, ReadingConnectionPoint and WritingConnectionPoint, are introduced. The reader and writer references of Channel now refer to the new subtypes (Figure 4).

![Fig. 4. ConnectionPoint is split into two subtypes.](image)

Instances of the original metamodel might now be inconsistent with the new metamodel. In particular, objects can no longer be of type ConnectionPoint, which is now abstract. Furthermore, any slots corresponding to Channel#reader (Channel#writer) must now reference objects of type ReadingConnectionPoint (WritingConnectionPoint) rather than ConnectionPoint. In the case of the office system described above, installation of the evolved metamodel triggers automatic consistency checking (discussed in Section 4.3), producing the following report of inconsistencies:
1) Cannot instantiate the abstract class, ConnectionPoint for the objects scanner, printer, telephone, unused.

2) Expected a ReadingConnectionPoint for photocopier.reader but got the ConnectionPoint called scanner.

3) Expected a ReadingConnectionPoint for fax machine.reader but got a ConnectionPoint called telephone.

4) Expected a WritingConnectionPoint for photocopier.writer but got a ConnectionPoint called printer.

5) Expected a WritingConnectionPoint for fax machine.writer but got a ConnectionPoint called telephone.

The above report indicates that only a small number of inconsistencies need to be reconciled. Consequently, we may decide to repair them using the Human-Usable Textual Notation (HUTN) described in Section 4.2.

In this case, we choose to edit the model such that the scanner will now instantiate ReadingConnectionPoint, and the printer WritingConnectionPoint. However, a migration strategy is not apparent for telephone, which is used as both a reader and a writer. Similarly, as the ConnectionPoint unused is neither a reader nor a writer, which type should unused now instantiate? Perhaps unused should be deleted? The partially reconciled HUTN can now be shared with another user, who may know the best way in which to complete the migration.

6 CONCLUSIONS AND FUTURE WORK

We have discussed, in the context of co-evolution, the limitations of the default model deserialisation performed by EMF, arguably the most robust and widely-used modelling framework. We have implemented an alternative deserialisation mechanism that binds a model to a generic metamodel. Consequently, EMF models can be loaded when their metamodel is unavailable, and when they are inconsistent with their metamodel. By detecting when a metamodel has changed, automatic consistency checking can be performed in downstream workspaces in response to installation of a new version of a metamodel.

Future work will involve exploring and implementing ways in which new and existing approaches to managing co-evolution can be enhanced using the generic metamodel and consistency checking presented in this paper.

Acknowledgement. The work in this paper was supported by the European Commission via the MOD-ELPLEX project, co-funded by the European Commission under the “Information Society Technologies” Sixth Framework Programme (2006-2009).

REFERENCES


