Maintaining Invariant Traceability through Bidirectional Transformations

Yijun Yu[1], Yu Lin[2], Zhenjiang Hu[3], Soichiro Hidaka[3], Hiroyuki Kato[3], and Lionel Montrieux[1]
[1] *Department of Computing, The Open University, Milton Keynes, United Kingdom*
[2] *Department of Computer Science, University of Illinois at Urbana-Champaign, USA*
[3] *National Institute of Informatics, Tokyo, Japan*

***Abstract*—Following the "convention over configuration" paradigm, model-driven development (MDD) generates code to implement the "default" behaviour that has been specified by a template separate from the input model, reducing the decision effort of developers. For *flexibility*, users of MDD are allowed to customise the model and the generated code in parallel. A synchronisation of changed model or code is maintained by reflecting them on the other end of the code generation, as long as the traceability is unchanged. However, such invariant traceability between corresponding model and code elements can be violated either when (a) users of MDD protect custom changes from the generated code, or when (b) developers of MDD change the template for generating the default behaviour.**

A mismatch between user and template code is inevitable as they evolve for their own purposes. In this paper, we propose a two-layered invariant traceability framework that reduces the number of mismatches through bidirectional transformations. On top of existing *vertical* (model↔code) synchronisations between a model and the template code, a *horizontal* (code↔code) synchronisation between user and template code is supported, aligning the changes in both directions. Our `blinkit` tool is evaluated using the data set available from the CVS repositories of a MDD project: Eclipse MDT/GMF.

I. Introduction

Aiming at productivity, "convention over configuration" is a software design paradigm meant to reduce the decision effort of developers while preserving flexibility. Following this paradigm, model-driven development (MDD) of software projects (e.g., Eclipse Modeling Framework, EMF) generates code from models to implement the "default" behaviour, which was specified by a template separate from the input model. As users of MDD, programmers are allowed to customise the generated code, and the modellers are allowed to modify the models further. To support the eventual round-trip synchronisation of model and code [1], markers for traceability correspondence (e.g., `@generated`) are inserted at the beginning of the generated methods by MDD, indicating to the programmers that all changes to the model and the template will be reflected by the annotated method. In other words, any change made by programmers on such methods will get lost after another round of code generation. In order to protect such changes in the user code from getting lost, programmers are allowed to modify the `@generated` marker to `@generated NOT`, instructing the code generators to skip the user-specified methods.

This technique is common in MDD practice, for another example, the Acceleo Model2Text project uses "`[protected] ... [/protected]`" annotations to enclose user-specified parts for arbitrary textual outputs. To illustrate its problematic use in development, we use EMF, the Eclipse Modeling Framework (EMF) * as an exemplar MDD framework: The template code is generated by EMF first, later refined manually into functioning user code. Once programmers change the model and regenerate the template code, its traceability to the user code becomes necessary.

Ideally, a round-trip engineering approach should support the correct propagation of changes in both directions [2]. Current state-of-the-art MDD tools such as EMF maintain the synchronisation of model and code by reflecting changes in both directions. However, such invariant traceability links between corresponding model and code elements can be easily violated either when (a) users of MDD protect custom changes from the generated code by changing `@generated` to `@generated NOT` or when (b) developers of MDD change the template for generating the default behaviour for those `@generated` methods, losing all users' change(s). In either case, a better solution would be to preserve the changes made by the users as well as the changes made on the model as long as they are consistent. In Section III, a detailed example will illustrate the problems such a mismatch can cause.

We consider in this paper that the mismatch problem between user-modified and template-generated (hereafter "user" and "template") codes is inevitable as they evolve for their own purposes, and propose a two-layered invariant traceability framework that merges the changes when possible through bidirectional transformations. The first layer of the framework synchronises the structural changes between the model and the template code at the API level *vertically* (denoted by model↔code) using existing state-of-the-art MDD tools such as EMF; the second layer synchronises the behavioural changes between the template code and the user code inside the method bodies *horizontally* (denoted by code↔code) if users wish to preserve both types of changes. Aligning these changes in both directions, our `blinkit` tool is evaluated using the data set available from the CVS repositories of the Eclipse MDD projects EMF and GMF.

To effectively use the invariant traceability framework, users only need to insert a traceability annotation `@generated INV`, instructing the code generator to au-

*See http://www.eclipse.org/modeling/emf

tomatically (a) compute the differences between the current template and user-modified methods; and (b) derive a bidirectional transformation for future code generation to reflect changes of the model or changes of the template back to the users methods. The tool raises warnings when there are inconsistencies between the template and the user code.

A prototype has been successfully applied to changes recorded in the evolution of the GMF framework from its CVS repository, which shows that our new approach of round-trip engineering is promising and potentially useful in practice. Fig. 1 presents an overview of `blinkit` when it is applied to the case study of EMF/GMF, where EMF is the synchronisation framework for vertical traceability and `blinkit` is the horizontal synchronisation counterpart. Examples indicate that when the complementary changes to templates and user-modified code are conflicting or redundant, our tool can avoid some dead code redundancies and raise some warnings as compilation errors.

Our major technical contributions are listed as follows:

- From a user's perspective, a new type of annotations `@generated INV` is supported as explicit markers for automatic maintenance of *invariant traceability* between template (`@generated`) and user (`@generated NOT`) code, whereas `@generated NOT` marked methods only keep user's change while ignoring changes made to the model.
- A novel algorithm is proposed for automatically generating a one-pass bidirectional transformation from the meaningful differences between the template code and arbitrarily modified user code such that the *round-trip relation* between the structures of the model and the code can be correctly maintained.
- An empirical study is done on a CVS repository of the state-of-the-art MDD project GMF, highlighting the *benefits* of maintaining invariant traceability links, especially when there are 178 changes to the @model parts have impact on the bodies of 54% of the 28,070 revisions of methods marked by `@generated NOT`.

Compared to existing MDD and traceability approaches, `blinkit` derives invariant transformations automatically from meaningful changes in the source code [3]. Unlike our initial work that proposes to apply bidirectional transformations directly on class diagrams and Java code [4], this work takes full advantages of the state-of-art synchronisations of many-to-many vertical traceability links between the model and the template code, such that the horizontal bidirectional transformations are applied only to the method bodies relevant to the user-modified behaviours. Since the template and the user method code are at the same level of abstraction, `blinkit` is allowed to derive a forward transformation from the user method to the generated method. Our initial work [4] needs to monitor the execution logs of user-specified ATL transformations, which is no longer needed.

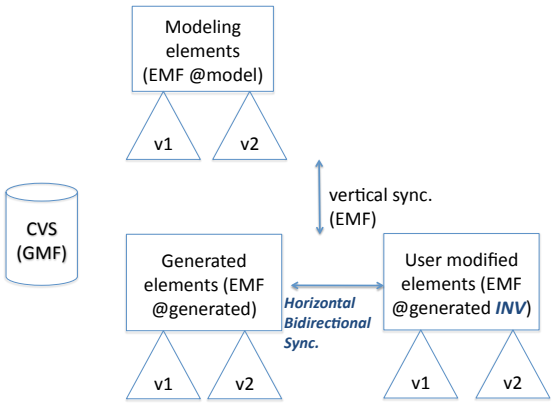


Figure 1. An overview of the horizontal and vertical traceability links in the bidirectional invariant traceability framework: `blink`. V1 and V2 are two revisions of model, template or user codes extracted from the CVS repository of a software development project using EMF code generation.

The remainder of this paper is organised as follows: Section II gives preliminary knowledge on EMF and introduces the bidirectional transformation mechanism. Section III provides one example to illustrate the synchronisation problem of invariant traceability. Section IV overviews the round-trip process for horizontal and vertical invariant traceability and indicates the position of `blinkit` in the overall process. Section V describes the technical details of `blinkit`, the generation of bidirectional transformations from meaningful changes, and the one-pass optimisation for efficiency. Section VI presents the observations of the CVS evolution history of the GMF project to show the number of cases where the `@generated INV` markers can be useful. Section VII compares related work, Section VIII concludes.

II. Background

A. EMF as a state-of-the-art MDD framework

The Eclipse Modeling Framework (EMF) is a MDD framework and a code generation facility for building Eclipse tools and other applications around a structured model. From a meta-model specified in XMI or XML Schema, EMF generates default (i.e., template) Eclipse plugin tools that consist of Java classes for manipulating a model, along with Java adapter classes for viewing and editing a model. From a rather complex meta-model, it is impractical to generate all source code because programmers may customise the default behaviour by modifying some parts of the code in order to achieve their own goals. Using EMF, the template code is generated first, and later refined manually into functioning user code. Once programmers change the model and regenerate the template code, it brings the necessity of the traceability to and from the user code.

The MDD part of the EMF consists of one code generation component (i.e., JMerge) to generate source code from a meta-model (i.e., .ecore). The template used by the code generator is specified in the JavaJET template language similar to that of JSP, which will bind the template variables

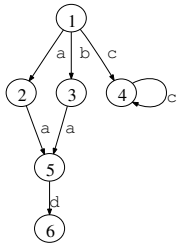


Figure 2. A simple rooted and edge-labelled graph

with the default values specified in the corresponding .gen-model configuration files. There is no one-to-one mapping between a class operation to a Java method in this code generation: A class in the class model can be implemented in multiple classes in the `model`, `impl`, `util`, `edit`, and `editor` packages, scattered across the generated `model`, `edit`, `editor`, and `test` plugins. In addition, there will be `Package` and `Factory` classes created and instantiated by the classes in the model. It is hard to maintain such many-to-many relations using traditional traceability techniques.

Note that EMF is used in this study both as a subject matter (as part of the MDT case study), as well as a component of our solution. Parts of the EMF tool were generated using the EMF code generator as well. It can be seen from the existing EMF implementation that developers marked some methods as `@generated NOT` to preserve the changes that cannot be generated from the Ecore models alone. It can therefore be expected that the code generated from EMF will be modified by users.

B. Bidirectional Transformation

We use `GRoundTram` [5] to do bidirectional graph transformations. This well-behaved framework guarantees the round-trip property in bidirectional model transformations.

A model transformation is described in UnQL+ [6], an extension of the SQL-like graph query language UnQL [7] with three graph update constructs to achieve efficiency and expressiveness, namely Replacing, Deleting, and Extending. The model transformation is then desugared to the core algebra (UnCal) which consists of a set of constructors for building graphs and a powerful structural recursion for manipulating graphs. This graph algebra can have clear bidirectional semantics and be efficiently evaluated in a bidirectional manner [8]. Graphs (models) in UnQL+ are rooted and edge-labelled (i.e. all information is stored as labels on edges rather than on nodes and the labels on nodes have no particular meaning), and represented in UnCAL or the standard Dot format which can be visualised and edited by the popular Graphviz tool [9].

To illustrate UnQL+, consider a simple graph $\$db$ (the root is the node 1) in Fig. 2. We can select the subgraph pointed by the edge labelled b from the root by

$$\textbf{select } \$g \textbf{ where } \{b : \$g\} \textbf{ in } \$db,$$

delete the subgraph rooted at node 5 reached by $b.a$ by

$$\textbf{delete } b.a \rightarrow \_ \textbf{ in } \$db,$$

and insert a subgraph G under the node reached from the path $a.a$ by

$$\textbf{extend } a.a \rightarrow \_ \textbf{ with } G \textbf{ in } \$db.$$

III. A Motivating Example

To illustrate the problem concretely, we use a constructed example here. Suppose an EMF user initially specifies a simple model that consists of one `Entity` class with a single `name` attribute. Using the code generation feature of EMF, she will obtain a *default* implementation which consists of 8 compilation units in Java (Fig. 3).

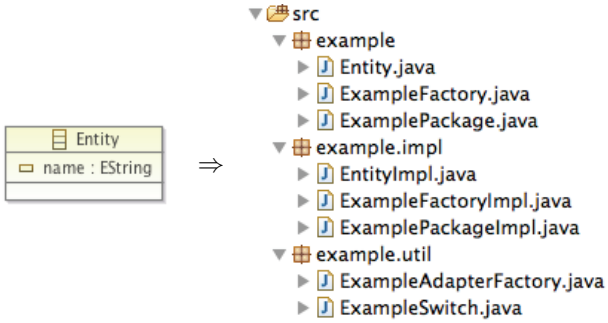


Figure 3. Default code generated from the EMF meta-model

Fig. 4 lists parts of the generated code. The `Entity` Java interface has getter and setter methods for the `name` attribute. They are commented with `@generated` annotations which indicate that the methods are part of the default implementation. Similarly, such `@generated` annotations are added to every generated element in the code, e.g., shown in the skeleton of `EntityImpl` Java class.

The annotation `@generated` defines a *single-trip traceability contract* from the model to the annotated code element. A change in the model or a change in the modelling framework can be propagated to the generated code; however, a change on the generated code will not cause a change to the reflected model and will thus be discarded upon next code generation.

Because the default implementation is not always desired, the code generation shall keep user specified changes as long as they are not inside the range of generated traceability, the set of methods marked by `@generated` that keeps the changes of generated templates. This can be achieved by adapting the `@generated` annotation into `@generated NOT`, a non-binding traceability that reflects programmers' intention that it will not be changed when the implementation code is regenerated. Note that such non-binding traceability indicated by `@generated NOT` is still different from those without any annotation at all: Without such an annotation, EMF will generate new implementation of a method body following the templates.

```java
package example;
import org.eclipse.emf.ecore.EObject;
/** @model */
public interface Entity extends EObject {
  /** @model */ public String getName();
  /** @generated */ void setName(String value);
}
```

```java
package example.impl;
import example.Entity;
...
/** @generated */
public class EntityImpl extends EObjectImpl
    implements Entity {
  ...
  /** @generated */
  protected String name = NAME_EDEFAULT;
  ...
  /** @generated */
  public String getName() { return name; }
  /** @generated */
  public void setName(String newName) { ... }
  ...
  /** @generated */
  @Override
  public String toString() {
    if (eIsProxy()) return super.toString();
    StringBuffer result = new StringBuffer(super
        .toString());
    result.append(" (name: ");
    result.append(name);
    result.append(')');
    return result.toString();
  }
} //EntityImpl
```

Figure 4. Parts of the generated code in Fig. 3

This workaround is not ideal. If a user parametrises the `toString()` method to append an additional `type` to the returned result. To guard the method from being overwritten by future code generations, the annotation `@generated NOT` is used. She also applies a *Rename Method* refactoring, changing the `getName` method into `getID`. The modified parts are shown in Fig. 5. Propagating these changes back to the model, the `name` attribute will be renamed into `iD` automatically, following the naming convention that attribute identifiers start with a lower case character.

The regeneration of the code will result in the changes in Fig. 6: the setter methods and the implementations of both getter/setter methods are modified according to the default implementation of the new model. These are expected. However, two unexpected changes are not desirable. First, a compilation error results from the change in the default implementation, where the attribute `name` used in the user controlled code no longer exists. Second, the default implementation of the `toString()` method is generated with the original signature, which will of course become dead code since the user has already modified all call sites of `toString()` to reflect the insertion of the new type. Similarly, the user specified `toString()` method can also become dead code, if it is no longer invoked by the new default implementation.

Compilation errors are relatively easy to spot by the programmer with the aid of the Eclipse IDE, but the dead

```java
/** @model */
public interface Entity extends EObject {
 /** @model */ public String getNameID();
 /** @generated */ public void setName();
}
...
/** @generated */
public class EntityImpl extends EObjectImpl
    implements Entity {
  /** @generated */
  public String getNameID() { return name; }
  ...
  /** @generated NOT*/
  @Override
  public String toString(String type) {
    if (eIsProxy()) return super.toString();
    StringBuffer result = new StringBuffer(super
        .toString());
    result.append(" (name: ");
    result.append(name);
    result.append(')');
    result.append(type);
    return result.toString();
  }
} //EntityImpl
```

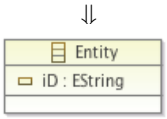


Figure 5. User modifications to the generated code: insertions are underlined and the deletions are ~~stroked out~~; the changes are reflected

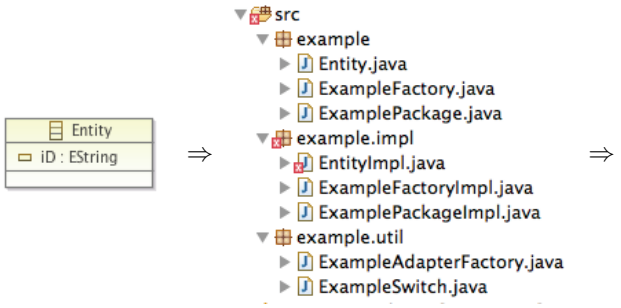
code problems are more subtle because the IDE will not complain. Therefore, it will be more difficult for developers to notice the consequences.

Ideally, the user should be able to specify which parts of her changes to the code need to be kept rather than overwritten by code generation. For this purpose, a new annotation for *invariant traceability* (`@generated INV`) will be used in our proposed approach. As a result, Fig. 7 illustrates the changes to be propagated to users' code after our approach of bidirectional transformations is adopted.

IV. Overview of `blinkit`

The prototype `blinkit`[†] supports our idea of maintaining the traceability of the user code and template code through bidirectional transformations. Fig. 8 gives an overview of the dataflow of the components inside `blinkit`. Dashed arrows represent the external changes made by developers or vertical synchronisations between models and template code, whilst the open- or close-ended solid arrows represent forward and backward transformations, respectively. First, developers define an original EMF model and synchronise with the template code from the EMF engine (①). Developers may modify the template code into user code to meet their requirements and mark some parts of their code with the `@generated INV` annotation (②). Developers can change the model if needed (③) and re-

[†]`blinkit` can be downloaded from http://sead1.open.ac.uk/linkit/




```
1 /** @model */
2 public interface Entity extends EObject {
3  /** @model */ public String getID();
4  /** @generated */ public void setNameID();
5 }
6 ...
7 /** @generated */
8 public class EntityImpl extends EObjectImpl
      implements Entity {
9   /** @generated */
10   public String getID() { return nameiD; }
11   ...
12   /**@generated*/
13   public String toString() {
14     if (eIsProxy()) return super.toString();
15     StringBuffer result = new StringBuffer(super.toString());
16     result.append(" (iD: ");
17     result.append(iD);
18     result.append(')');
19     return result.toString();
20   }
21   /** @generated NOT */
22   public String toString(String type) {
23     if (eIsProxy()) return super.toString();
24     StringBuffer result = new StringBuffer(super
           .toString());
25     result.append(" (name: ");
26     result.append(name);
27     result.append(')');
28     result.append(type);
29     return result.toString();
30   }
31 } //EntityImpl
```

Figure 6. Regenerated code from the model: insertions are underlined and the deletions are ~~stroked out~~, the compilation error is doubly underlined.

synchronise the code using EMF (④). Note that, without `blinkit`, (④) may lose developers' modifications.

Having the above steps performed by developers, `blinkit` will generate a bidirectional transformation in the UnQL+ language by comparing the user code (denoted by U') and the original template code (denoted by T). The forward transformation uses this generated transformation. We represent the code by specific graphs and compare the graphs to generate UnQL+ in Section V-A. to take the modified user code (denoted by U') as inputs, which generates an intermediate code that will be the same as the template code T, such that the trace information stored by the forward transformation can be applied to the corresponding backward transformation. In the backward transformation, the intermediate code T and the modified template code (denoted by T') are used as inputs and the output is the merged code

```
1 /** @model */
2 public interface Entity extends EObject {
3  /** @model */ public String getNameID();
4  /** @generated */ public void setName();
5 }
6 ...
7 /** @generated */
8 public class EntityImpl extends EObjectImpl
      implements Entity {
9   /** @generated */
10   public String getNameID() { return name; }
11   ...
12   /** @generated INV*/
13   @Override
14   public String toString(String type) {
15     if (eIsProxy()) return super.toString();
16     StringBuffer result = new StringBuffer(super
           .toString());
17     result.append(" (name: ");
18     result.append(name);
19     result.append(')');
20     result.append(type);
21     return result.toString();
22   }
23 } //EntityImpl
```

⇓

```
1 /** @model */
2 public interface Entity extends EObject {
3  /** @model */ public String getID();
4  /** @generated */ public void setNameID();
5 }
6 ...
7 /** @generated */
8 public class EntityImpl extends EObjectImpl
      implements Entity {
9   /** @generated */
10   public String getID() { return nameiD; }
11   ...
12   /** @generated INV */
13   public String toString(String type) {
14     if (eIsProxy()) return super.toString();
15     StringBuffer result = new StringBuffer(super
           .toString());
16     result.append(" (nameiD: ");
17     result.append(nameiD);
18     result.append(')');
19     result.append(type);
20     return result.toString();
21   }
22 } //EntityImpl
```

Figure 7. The use of invariant traceability to propagate changes in the backward direction resulting in $T' + U'$.

(denoted by $T' \oplus U'$), which not only reflects the changes made to the model but also conserves the modifications applied by the developers in ②. Notice that the intermediate code is the same as the template code in our approach. We do not substitute it by the template code because it separates the forward and backward transformation and makes the transformation process clearer.

V. Implementation of `blinkit`

We will illustrate the implementation of `blinkit` by first explaining the overall process, then providing details for the critical steps for correctness and efficiency.

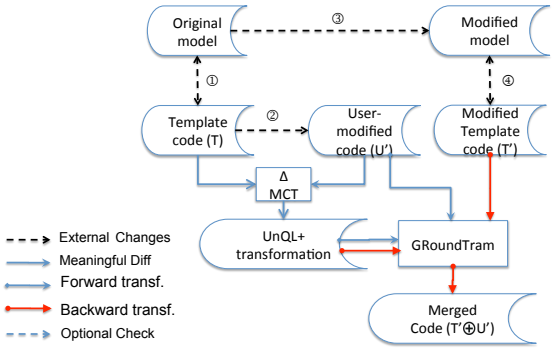


Figure 8. The data flows inside `blinkit`

A. The Overall Process

We use `GRoundTram`[‡] to perform bidirectional transformations and to keep the traceability of the template and user method bodies. `GRoundTram` adopts input graphs which are represented in `UnCal` or `Dot` format. Thus, in order to perform bidirectional transformations on Java code generated by EMF through `GRoundTram`, we first translate Java code into `UnCal` or `Dot` (i.e. using `UnCal` or `Dot` to represent Java source code). When translating Java code into `UnCal` or `Dot`, we adopt the EMF model as an intermediate representation of Java code since EMF model is much more navigable than specific abstract syntax trees and can be manipulated more easily. After we get the `UnCal` or `Dot` graphs, we can generate UnQL+ [10] automatically and perform bidirectional transformations using `GRoundTram`.

To achieve such a round-trip process, our framework involves four engineering steps on either side of the forward/backward directions to support the code↔code horizontal synchronisation between the evolving template and user code (see Fig. 9). Without loss of generality, in the following we denote the template-generated and user-modified codes as T and U', the modified template codes as T', and the merged code as $T' \oplus U'$, respectively.

1) Given that Java codes T, T' were generated and synchronised from the model using EMF's built-in vertical synchronisation with user's Java codes U' except for those methods that are marked by `@generated INV`, we first parse all the `@generated INV` methods in T, T', U' using the `JaMoPP` parser on top of the `EMFtext` framework [§]. The differences between T and U' are obtained using the API of the `EMFcompare` framework [¶] after the meaningful differences were preprocessed using `mct`[‖]. Here the differences are a comprehensive representation of code rather than editing operations;
2) The EMF models of T, T', U' are translated into our specific `UnCal` graphs using `Ecore`'s reflection API, and as a by-product, an UnQL+ transformation is generated from the meaningful differences between T and U' using the algorithm in Fig. 10;
3) The `UnCal` graphs of T, T', U' are transformed into `Dot` graphs by `GRoundTram`, which preserves the paths from the graph root to any node on the `UnCal` graphs in the equivalent `Dot` graphs;
4) Using the generated UnQL+ transformation and the `Dot` graphs of U' as inputs, the `GRoundTram` system also performs a forward transformation to output a group of `Dot` graphs that represent T, guaranteed by the one-pass optimisation described in Section V-C;
5) Backwards, using the same UnQL+ transformation on the `Dot` graphs of the modified template code T', and the internal graph traceability between T and U' kept by the forward transformations that performs a backward transformation to output the `Dot` graphs that represent the merged user code, denoted by $T' \oplus U'$;
6) Path-equivalent `Uncal` graphs corresponding to $T' \oplus U'$; are re-generated from the resulting `Dot` graphs;
7) An equivalent EMF model of those `Uncal` graphs for $T' \oplus U'$ is obtained using an `Xtext` parser [**] of `Uncal` to process its abstract syntax using EMF API;
8) Finally, the merged Java code $T' \oplus U'$ is obtained from the EMF model using `JaMoPP`'s pretty-print function.

Since we have a TXL-based parser to generate `Dot` graphs [3], our presented technique is not limited to EMF based external representations of Java. As long as a code generation framework for vertical synchronisation is available, it will be possible to adapt our horizontal synchornisation framework. However, we do not intend to implement the vertical synchonisation baseline because it will be complex to maintain one-to-many traceability links invariant.

To apply our approach effectively, one can initially attach INV to all `@generated` Java methods as a trivial identity bi-transformation. Once an inconsistency warning is raised, the marker can be modified to either `@generated` or `@generated NOT` incrementally, in order to preserve template- or user-modified changes respectively. Users introducing `@generated INV` are currently responsible for resolving the warnings.

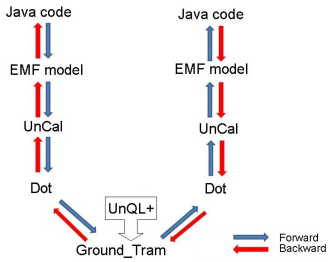


Figure 9. The multi-tier architecture of `blinkit`

[‡] See http://www.biglab.org
[§] See http://www.jamopp.org
[¶] See http://www.eclipse.org/emf/compare
[‖] See http://sead1.open.ac.uk/mct [3]
[**] See http://www.eclipse.org/Xtext

B. Generating Correct UnQL+ Transformations

To enable the use of UnQL+ on MDD, we need to make sure that translating the artifacts between the code, the `UnCal` graphs and node-traced graphs such as Graphviz `Dot` does not lose information. This can be done in a multi-tier fashion [11], and one critical step is to guarantee the correctness of the horizontal (code↔code) synchronisation, that is to generate a correct UnQL+ bidirectional transformation from the meaningful differences between the original template code and the modified user code.

```
 1: function UNQLGENERATOR(dotUser, dotTemplate)
 2:     rootUser ← the root node of dotUser
 3:     rootTemplate ← the root node of dotTemplate
 4:     MatchingDot(rootUser, rootTemplate)
 5: end function
 6: function MATCHINGDOT(nodeUser, nodeTemplate)
 7:     edgeSetUser ← outgoing edges of nodeUser
 8:     edgeSetTemplate ← outgoing edges of nodeTemplate
 9:     for all edgeUser ∈ edgeSetUser do
10:         flag = false
11:         childNodeUser ← target node of edgeUser
12:         for all edgeTemplate ∈ edgeSetTemplate do
13:             if edgeUser = edgeTemplate then
14:                 flag = true
15:                 childNodeTemplate ←
16:                         the target node of edgeTemplate
17:                 MatchingDot(childNodeUser, childNodeTemplate)
18:             end if
19:         end for
20:         if flag = false then
21:             DeleteConstructor(childNodeUser)
22:         end if
23:     end for
24:     for all edgeTemplate ∈ edgeSetTemplate do
25:         flag = false
26:         childNodeTemplate ← target node of edgeTemplate
27:         for all edgeUser ∈ edgeSetUser do
28:             if edgeTemplate = edgeUser then
29:                 flag = true
30:             end if
31:         end for
32:         if flag = false then
33:             ExtendConstructor(childNodeTemplate)
34:         end if
35:     end for
36: end function
```

Figure 10. Procedure of generating UnQL+

We can then use the `Dot` graphs as inputs to perform bidirectional transformations: `GRoundTram` needs to use bidirectional transformations specified in the `UnQL+` graph transformation language. As we said in Section IV, UnQL+ is used in forward transformations to convert user code into intermediate code which is the same as template code. We implemented an algorithm shown in Fig. 10 to generate the UnQL+ transformation automatically in order to mitigate developers' burden. After representing the Java code that we want to synchronise using `Dot` graphs, we can compute the differences between the `Dot` graphs and such differences can be used to generate UnQL+ automatically.

The UnQL+ generated by our approach only contains Deleting and Extending operations. Replacing operations can be replaced by a Deleting and an Extending operation. The `UnQLGenerator` algorithm starts node matching from the root in the `Dot` graphs. For each node pair, we compare their outgoing edges' labels. If we find one edge which exists in the user's graph but does not exist in the template's graph, that means deletion occurs and we will generate a Deleting operation to delete the subgraph beneath the node under comparison (Lines 20-22 in Fig. 10); if the two edges have the same labels, we will do node matching for their children recursively (Lines 13-18). Similarly, for those edges that the template's graph has but the user's graph does not, we need an Extending operation (Lines 32-34). For the Extending operation, we have to record the subgraph that needs to be inserted. To generate UnQL+ transformation, path information composed by edge labels is needed in order to denote which part should be deleted or inserted. Since the labels on the outgoing edges of one node are different from each other in our `Dot` graphs, such paths which reflect the changes between two `Dot` graphs are unique. We have to obtain such paths to generate UnQL+ transformation and the uniqueness of the paths guarantees the correctness of our UnQL+ generated algorithm (i.e. if there are two outgoing edges of a node with the same label, this algorithm may not generate the desired UnQL+ transformation).

An example is shown to illustrate this algorithm (Fig. 11). We transform the graph shown in Fig. 11(a) into the one in Fig. 11(d). We have to delete the edge *e*, *g* and *subgraph 2*, and insert the edge *h* and *subgraph 4*. The algorithm first generate Deleting operations which could transform Fig. 11(a) into Fig. 11(b). Note that the Deleting operations contain the deletion of *subgraph 3* and edge *f* though they should not be deleted. This is because when the algorithm deletes the edge *g*, it will also delete its pointed subgraphs (*subgraph 3*) and the edges connected to those subgraphs (edge *f*). Then when generating the Extending operation, the algorithm not only inserts the edge *h* and *subgraph 4*, but also inserts the edge *f* and *subgraph 3* again. This extra deletion and insertion may reduce the transformation's efficiency and more efficient algorithms should be explored in the future work. Comparison algorithms for trees and graphs based on dynamic programming [12] may be adopted.

C. Derivation of a One-Pass Transformation

The automatically generated UnQL+ transformations are not optimal. One performance bottleneck is that there are typically multiple pairs of different elements, leading to multiple basic editing operations composed sequentially. Ideally,

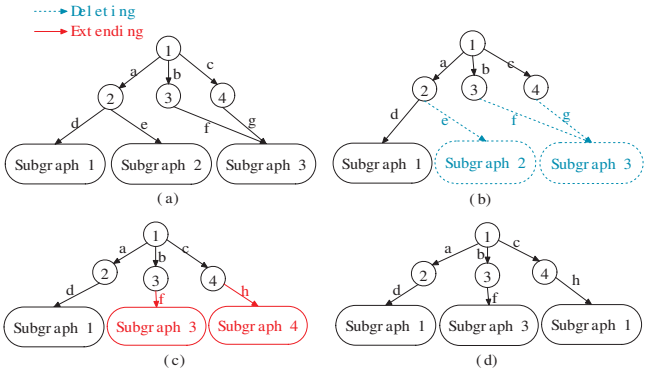


Figure 11. An example to illustrate the algortihm in Fig. 10

these operations should be composed in parallel as one single graph transformation such that `GRoundTram` could obtain results in one pass of traversing the graph structure. In theory, the composition of general UnQL+ transformations has to be sequential. However, the transformations we obtained from the diff results have nice properties that make it possible to compose the basic operations in parallel without introducing any side effect. The rationale is given below.

Let $e_1, e_2, \ldots, e_n$ be a sequence of the editing operations obtained above, where any two editing operations, e_i and e_j, are independent in the sense that neither insertion nor deletion is done on a previously inserted/deleted part. This can be formalised as $\forall i, j : e_i \circ e_j = e_j \circ e_i$ holds.

We show that any editing sequence can be automatically transformed to a one-pass traversal of graphs. Our algorithm consists of two steps: We first map the editing sequence to composition of a set of graph transformations where each graph transformation corresponds to an editing operation, and then try to fuse the composition into a single one-pass graph traversal in terms of a structural recursion in `UnCal`.

1) Mapping from Editing Sequence to Composition of Graph Transformations: First, each editing operation corresponds to a simple graph transformation in UnQL+. Let p denote the path from the root to the node n (i.e., a sequence of edge labels from the root to the node n) in the graph $\$db$. Then, the editing operation $e(\$db)$ for deleting a subgraph $\{l : \$g\}$ rooted at the node n can be translated to $E(\$db)$:

$$\textbf{delete } p \rightarrow \{l : \$g\} \textbf{ in } \$db$$

and the edition operation $e'(\$db)$ for inserting a subgraph G to the node n to $E'(\$db)$:

$$\textbf{extend } p \rightarrow \$g \textbf{ with } G \textbf{ in } \$db.$$

Now a sequence of editing operations $e_1, e_2, \ldots, e_n$ corresponds to the composition of graph transformations of $E_1(\$db), E_2(\$db), \ldots, E_n(\$db)$, that is,

$$E_1(E_2(\ldots(E_n(\$db))).$$

2) Fusion of Composition of Graph Transformations: `GRoundTram` provides a powerful fusion mechanism that can automatically fuse a composition of graph transformations into one so that unnecessary exchange of graph structures between the individual graph transformations can be saved. We could apply this general fusion mechanism, but we can do better based on the fact that each graph transformation is independent from each other. Thus we develop a specialised but more efficient fusion algorithm.

Our idea is to construct an action tree from the editing sequence and then map the action tree to a one-pass transformation. For an action tree, leaves are marked with two action markers $D(l)$ and $I(G)$, denoting respectively the deletion of a graph pointed by the edge l when possible and the insertion of a graph G. Fig. 12 gives an example

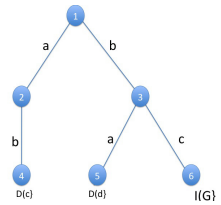


Figure 12. An action tree

action tree, which describes the intention of that composed transformation: Given a graph, for the node reached by the path of $a.b$ do deletion, for the node reached by the path $b.a$ do deletion, and for the node reached by the path $b.c$ do insertion. In fact, an action tree is a tree automaton that can be mapped to a structural recursion in UnCal that traverses graphs only once [7].

Let us show that a sequence of editing operations can be represented by an action tree. First, it is clear that a single editing operation can be represented by a simple action tree (which is actually a tree where each node has just a single child). Now given two action trees t_1 and t_2, we can combine them to $merge(t_1, t_2)$, satisfying that (1) any action in t_1 or t_2 appears in $merge(t_1, t_2)$ and any action in $merge(t_1, t_2)$ appears in t_1 or t_2 and (2) there is no node that has two outgoing edges with the same labels. The algorithm $merge(t_1, t_2)$ is defined as follows:

$$\begin{array}{lcl} merge((), t_2) & = & t_2 \\ merge(t_1, ()) & = & t_1 \\ merge(t_1 \cup t_1', t_2) & = & merge(t_1, merge(t_1', t_2)) \\ merge(\{l : t_1\}, \{l : t_2'\} \cup t_2) & = & \{l : merge(t_1, t_2')\} \cup t_2 \\ merge(\{l : t_1\}, t_2) & = & \{l : t_1\} \cup t_2 \end{array}$$

With this merge operation, we can merge all editing operations into an action tree.

VI. Evaluation

In order to evaluate the benefits of the framework, we assess two research questions. First, does `blinkit` work correctly on the examples without human intervention? Second, given that the illustrative example is constructed, are there realistic cases in an open-source MDD software development project where invariant traceability links can be synchronised using `blinkit`? The first question needs to

be answered first since only a working `blinkit` prototype can bring benefits to the users. To answer these questions, we conducted two sets of experiments.

Correctness. In the 1st experiment we create a model using EMF as illustrated earlier and generate the default Java source code. There is only 1 class (i.e., Entity) and 1 attribute (i.e., name) respectively in the EMF meta-model, the code generated by EMF has 2 classes (i.e., Entity, EntityPackage) and 1 attribute (i.e., name) annotated by `@model`. In terms of the number of `@generated` attributes, there are respectively 8 classes, 48 attributes and 10 methods generated in 3 packages. Currently, `blinkit` only supports synchronisation between method bodies, which users are more likely to change in order to introduce different behaviours.

To evaluate the correctness of `blinkit`, we manually modified those 9 methods annotated by `@generated` into `@generated INV`. The `blinkit` tool generated an 'identity' UnQL+ transformation for each invariant traceability `@generated INV`. Then we manually modified individual `@generated INV` methods by adding, removing, and replacing some statements to simulate a possible change by the user. In parallel, we applied the Rename Class and the Rename Method refactorings to the 3 elements in the model that were annotated by the `@model` markers to see whether they have an impact on the user-modified code.

The simulated changes to the newly generated template code may conflict with the changes introduced by user, e.g., by renaming "name" to "iD". In those cases the changes should be merged by `blinkit`. For correctness, we checked the merged results manually to see whether they preserve both changes in the model and in the user code. When the changes of the `@model` did not affect the changes to the `@generated INV` elements by the user, all changes introduced by the user were preserved.

Intersection of the changes in `@generated NOT`, `@generated` and `@model`. To answer the second question, we first extracted all revisions of Java code from the CVS repository of the GMF project[††], which spans about 6 years period between 2005/08/14 and 2011/08/13. According to this repository, there have been 28,070 revisions including all the ones before the deleted files were placed in the `Attic` subfolders. In order to understand how many times the model parts have changed, we rely on the EMF convention that all modelling elements in the generated template code have been annotated with the `@model` markers. Therefore, we extracted the interface APIs that were marked by `@model` using the normalisation and clone detection technique reported in our earlier work [3]. This way, all meaningful differences for the model can be found among the 1185 pairs of possible revisions (we only compare those revisions on the same files). There are 178 pairs of meaningful changes, ranging from 2005/08/14 till 2011/02/28, and no more changes after that. Fig. 13 shows the distribution of the 178 `@model` changes over this period.

[††]See http://archive.eclipse.org

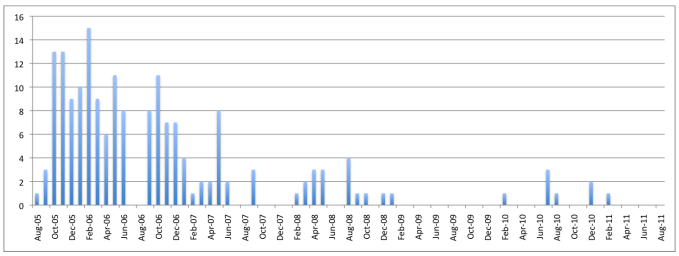


Figure 13. The distribution of `@model` changes in the CVS repository indicates that meta-models used in the GMF project is getting stabilised.

Next, we used a different normalisation on the same data-set, this time filtering only those methods that have been annotated by `@generated NOT` markers amongst 1,314 Java classes. For these Java methods, we moved their implementation in the body into the `<!-begin-user-doc> ... <!--end-user-doc>` pairs in the Javadoc comments, and changed their annotations into `@generated` such that the code generator will overwrite the user code with the template code, while the user modifications are still available in the comment. Finally, we compared the differences between the template and user codes, in order to see whether the changed `@model` elements appear in these method bodies. If they do, then we have confirmed that the bidirectional transformation could be useful in practice because the user code was indeed different from the template code, and those differences would interact with the modification of the model in the immediate revisions following the timestamps.

Throughout the data-set, we found that 15,223 revisions amongst the total of 28,070 revisions (i.e., 54%) will be influenced by the changes of the `@model` elements. There are 146,415 modified model elements referenced by the `@generated NOT` methods in the revisions. Therefore, on average $146,415/15,223 = 9.61$ modified modelling elements are referenced by the `@generated NOT` method bodies in every Java revision file.

We list threats to validity and some limitations below:

Construct validity: Instead of synchronising for the whole classes or packages, we focus on synchronising changes to the method bodies. The reason for this choice is practical as most of the time users would customise the behaviour of the default class, rather than rewriting them completely. A modification to method bodies is also often *required* because the template code would otherwise raise `UnsupportedOperationException`.

External validity: `blinkit` is built on existing open-source MDD toolsets `EMF`, `EMFtext`, `JaMoPP` and a transformation system `TXL`. It also integrates two research prototypes `GRoundTram` [5] and `mct` [3] which are freely available to download. The subject case study is also an open-source one whose CVS repository is publicly available.

Internal validity: Although studying committed changes on models in the GMF repository may be conservative, it is the only available source that we can rely on. It can be argued that the editing changes outside the repository present more synchronisation opportunities, however we have to estimate the benefits of our solution conservatively without empirically monitoring developers over their shoulders.

VII. Related work

We compare `blinkit` to the work in traceability, co-evolution, and bidirectional transformations.

Precise traceability. *Automated software document-code traceability recovery* has been studied by researchers from many angles since requirements traceability was proposed [2]: The review of the best practises in this field [13] suggests that automated techniques such as vector spaces [14], [15], LSI [16] are useful when part of the data-set relies on ambiguous documentation such as requirements, manuals and bug reports. It is expensive to obtain expert judgements for large applications [17] and to obtain high-quality inputs [18]. Incremental techniques have been proposed to analyse evolving traceability links [16] for better efficiency, however, precision cannot be greatly improved. Auxiliary information from programs such as call-graphs or traces [18] or XML-based (xlinkit) rules [19] could help improve the precision to some extent, but it is still largely expensive to gain better results through feedback [20]. On the other hand, the abstraction gap between models and code is much narrower that that between requirements document and code, thus precise tracing has been considered through maintaining semi-automated refactorings [21]. However, such refactorings are also expensive to construct, thus affordable only for medium-sized security software applications. Here we address the problem for general software projects that scales while MDD has been applied.

Invariant traceability for model/code co-evolution. When projects evolve, MDD methods face additional challenges: not only do artifacts such as models or code change over time, their changes also need to be *propagated* in order to maintain consistency even when artifacts are at different levels of abstraction. *Co-evolution* needs to be studied between models and code, between behavioural and structural models, or even between code and programming languages [22]. Modifications to a model must be reflected on the corresponding code in order to keep the model and the code *synchronised*. Code generators exist to automatically generate code from UML models [23]–[25]. In practice, however, code is often updated manually after it has been generated, and it is therefore necessary to reflect those changes back to the corresponding model. Round-trip engineering (RTE) is one-way to synchronise UML diagrams and code [26]. However, combining code generation and reverse engineering approaches is still not sufficient: Changing independently, models and code need to be *merged* first; Secondly, since models are generally at a higher level of abstraction than code, not all changes made to the code can be reflected back to models. Giese and Wagner [27] use triple graph grammars for RTE, but their approach is limited to elements that have a correspondence in the model. Other approaches, like Van Paesschen et al.'s [28], assume that both artifacts can be represented in a common representation. `Fujaba` [29] is yet another RTE approach that can generate Java code from UML class diagrams, and regenerated class diagrams from modified Java code, as long as developers follow Fujaba's naming conventions and implementation concepts. State-of-the-art RTE tools such as EMF/GMF make use of annotations to separate the portions of generated and user modified code, yet it is largely manual to maintain the correspondence of `@generated NOT` elements. The co-evolution of GMF project has been studied [30] for synchronising the EMF meta-models (model↔model) used by the project. Although we also use the same case study to evaluate the benefits, our focus is on automating the code↔code synchronisations.

Bidirectional transformation [31], [32] has been recently widely studied by researchers from different communities of programming language, software engineering, and database. It has many potential applications in software development, including model synchronisation [4], [33], round-trip engineering [34], software evolution [35], and multiple-view software development [36]. Our work shows that we can move from "potential" to "practical"; we achieve scalability by proposing a two-layer bidirectional transformation framework, hiding difficulties in writing bidirectional transformation by automatic deriving it from a sequence of editing operations, and widening its application scope by treating general graphs.

VIII. Conclusions and future work

In this paper, we presented a model-driven development (MDD) method, supported by the `blinkit` prototype, to maintain the invariant traceability between model and code through bidirectional transformations. Using this method, if code is annotated by `@generated INV`, a bidirectional transformation will be generated to correctly propagate changes in both directions. We tested our framework by the example shown in Section III and observed empirically how often `blinkit` can be used to maintain the invariant traceability based on the data-set in the CVS repository of GMF, a widely-used MDD project. It is also observed that applying the method earlier delivers more benefits since its meta-models change more frequently. In this case study, we also found that more changes were derived from the evolving model than from the evolving template.

Current work considers a basic form of invariant traceability where all meaningful changes are synchronised, regardless of whether they were derived from the template or from the model. Our future work will extend the syntax

and semantics of `@generated INV` to differentiate model changes from template changes.

Acknowledgement. The work is supported in part by EU SecureChange (www.securechange.eu), Microsoft SEIF (computing-research.open.ac.uk/seif), and NII BiG (www.biglab.org) projects.

REFERENCES

[1] L. Angyal, L. Lengyel, and H. Charaf, "A Synchronizing Technique for Syntactic Model-Code Round-Trip Engineering," in *ECBS '08*, 2008. pp. 463–472.

[2] O. C. Z. Gotel and A. C. W. Finkelstein, "An analysis of the requirements traceability problem," in *RE'94*, 1994, pp. 94–101.

[3] Y. Yu, T. T. Tun, and B. Nuseibeh, "Specifying and detecting meaningful changes in programs," in *ASE'11*, 2011, pp. 273-282.

[4] Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei, "Towards automatic model synchronization from model transformations," in *ASE'07*, 2007, pp. 164–173.

[5] S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano, "GRoundTram: An integrated framework for developing well-behaved bidirectional model transformations" in *ASE'11*, 2011, pp. 480-483.

[6] S. Hidaka, Z. Hu, H. Kato, and K. Nakano, "Towards a compositional approach to model transformation for software development," in *SAC '09*, 2009, pp. 468–475.

[7] P. Buneman, M. F. Fernandez, and D. Suciu, "UnQL: a query language and algebra for semistructured data based on structural recursion," *VLDB Journal: Very Large Data Bases*, vol. 9, no. 1, pp. 76–110, 2000.

[8] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano, "Bidirectionalizing graph transformations," in *ICFP'10*, 2010, pp. 205–216.

[9] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph - static and dynamic graph drawing tools," in *Graph Drawing Software*. Springer-Verlag, 2003, pp. 127–148.

[10] P. Buneman, M. Fernandez, and D. Suciu, "UnQL: a query language and algebra for semistructured data based on structural recursion," *The VLDB Journal*, vol. 9, no. 1, pp. 76–110, 2000.

[11] Y. Yu, Y. Lin, Z. Hu, S. Hidaka, H. Kato, and L. Montrieux, "blinkit: Maintaining invariant traceability through bidirectional transformations – a technical report," The Open University, Tech. Rep. TR2011/09, September 2011.

[12] W. Yang, "Identifying syntactic differences between two programs," in *Software - Practice and Experience*, vol. 21, 1991, pp. 739–755.

[13] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova, "Best practices for automated traceability," *IEEE Computer*, vol. 40, no. 6, pp. 27–35, 2007.

[14] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram, "Advancing candidate link generation for requirements tracing: The study of methods," *IEEE Trans. Software Eng.*, vol. 32, no. 1, pp. 4–19, 2006.

[15] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Software Eng.*, vol. 28, no. 10, pp. 970–983, 2002.

[16] H. Jiang, T. N. Nguyen, I.-X. Chen, H. Jaygarl, and C. K. Chang, "Incremental latent semantic indexing for automatic traceability link evolution management," in *ASE'08*, 2008, pp. 59–68.

[17] G. Antoniol, J. H. Hayes, Y.-G. Guéhéneuc, and M. D. Penta, "Reuse or rewrite: Combining textual, static, and dynamic analyses to assess the cost of keeping a system up-to-date," in *ICSM'08*, 2008, pp. 147–156.

[18] A. Egyed and P. Grünbacher, "Supporting software understanding with automated requirements traceability," *International Journal of Software Engineering and Knowledge Engineering*, vol. 15, no. 5, pp. 783–810, 2005.

[19] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: a consistency checking and smart link generation service," *ACM Trans. Internet Techn.*, vol. 2, no. 2, pp. 151–185, 2002.

[20] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, 2007.

[21] Y. Yu, J. Jürjens, and J. Mylopoulos, "Traceability for the maintenance of secure software," in *ICSM'08*, 2008, pp. 297–306.

[22] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *IWPSE '05*, 2005, pp. 13–22.

[23] A. Rountev, O. Volgin, and M. Reddoch, "Static control-flow analysis for reverse engineering of uml sequence diagrams," in *PASTE '05*, 2005, pp. 96–102.

[24] M. Keschenau, "Reverse engineering of UML specifications from Java programs," in *OOPSLA'04*, 2004, pp. 326–327.

[25] Y.-G. Guéhéneuc, "A reverse engineering tool for precise class diagrams," in *CASCON '04*, 2004, pp. 28–41.

[26] T. Hettel, M. Lawley, and K. Raymond, "Model synchronisation: Definitions for round-trip engineering," in *Theory and Practice of Model Transformations*, ser. Lecture Notes in Computer Science. 2008, vol. 5063, pp. 31–45.

[27] H. Giese and R. Wagner, "Incremental model synchronization with triple graph grammars," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science. 2006, vol. 4199, pp. 543–557.

[28] E. Van Paesschen, W. De Meuter, and M. D'Hondt, "SelfSync: A dynamic round-trip engineering environment," in *Model Driven Engineering Languages and Systems*, ser. Lecture Notes in Computer Science. 2005, vol. 3713, pp. 633–647.

[29] T. Klein, U. A. Nickel, J. Niere, and A. Zündorf, "From UML to Java and back again," University of Paderborn, Tech. Rep., 1999.

[30] D. D. Ruscio, R. Lämmel, and A. Pierantonio, "Automated co-evolution of GMF editor models," in *SLE*, ser. Lecture Notes in Computer Science, vol. 6563, 2010, pp. 143–162.

[31] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger, "Bidirectional transformations: A cross-discipline perspective," in *ICMT'09*, 2009, pp. 260–283.

[32] Z. Hu, A. Schürr, P. Stevens, and J. F. Terwilliger, "Dagstuhl seminar on bidirectional transformations (BX)," *SIGMOD Record*, vol. 40, no. 1, pp. 35–39, 2011.

[33] M. Antkiewicz and K. Czarnecki, "Design space of heterogeneous synchronization," in *GTTSE '07*, 2007, pp. 3-46.

[34] Michal Antkiewicz and Krzysztof Czarnecki, "Framework-specific modeling languages with round-trip engineering," in *MoDELS'06*, 2006, pp. 692–706.

[35] R. Lämmel, "Coupled Software Transformations (Extended Abstract)," in *IWSET'04*, 2004, pp. 31-35.

[36] M. Garcia, "Bidirectional synchronization of multiple views of software models," in *DSML'08*, vol. 324, 2008, pp. 7–19.