Specifying Model Transformations by Direct Manipulation using Concrete Visual Notations and Interactive Recommendations

Iman Avazpour[a], John Grundy[a], Lars Grunske[b]

[a]*Centre for Computing and Engineering Software and Systems (SUCCESS), Swinburne University of Technology, Hawthorn 3122, VIC, Australia*
[b]*Institute of Software Technology, Universität Stuttgart, Universitätsstraße 38, D-70569 Stuttgart, Germany*

Abstract

Model transformations are a crucial part of Model-Driven Engineering (MDE) technologies but are usually hard to specify and maintain for many engineers. Most current approaches use meta-model-driven transformation specification via textual scripting languages. These are often hard to specify, understand and maintain. We present a novel approach that instead allows domain experts to discover and specify transformation correspondences using concrete visualizations of example source and target models. From these example model correspondences, complex model transformation implementations are automatically generated. We also introduce a recommender system that helps domain experts and novice users find possible correspondences between large source and target model visualization elements. Correspondences are then specified by directly interacting with suggested recommendations or drag and drop of visual notational elements of source and target visualizations. We have implemented this approach in our prototype tool-set, CONVErT, and applied it to a variety of model transformation examples. Our evaluation of this approach includes a detailed user study of our tool and a quantitative analysis of the recommender system.

Keywords: Model Driven Engineering, Model Transformation, Visual Notation, Recommender System, Concrete visualizations

Email addresses: `iavazpour@swin.edu.au` (Iman Avazpour), `jgrundy@swin.edu.au` (John Grundy), `lars.grunske@informatik.uni-stuttgart.de` (Lars Grunske)

1. Introduction

Model transformation plays a significant role in the realization of Model Driven Engineering (MDE). Current MDE approaches require specifying correspondences between source and target models using textual scripting languages and the abstract representations of meta-modeling languages. Although these abstractions provide better generalization, and hence code reduction, they introduce difficulties for many potential transformation specification users. This is because in order to effectively use them, users have to possess in-depth knowledge of transformation languages and meta-modeling language syntax. These are often very far removed from the actual concrete model syntax for the target domain. Moreover, taking into account large models being used in today's software systems, many transformation specifications are very complex and challenging to specify and then maintain, even for experienced transformation script and meta-model users [1, 2, 3]. Although some approaches have been developed to mitigate these problems, such as by using visual abstractions [4, 5], by-example transformations [3, 6, 7], graph transformations [8, 9, 10, 11], automatic inference of bi-directional transformations [12, 13], and automated assistance for mapping correspondence deduction [14], none of these fully address the problems nor do so in an integrated, visual, human-centric and highly extensible way.

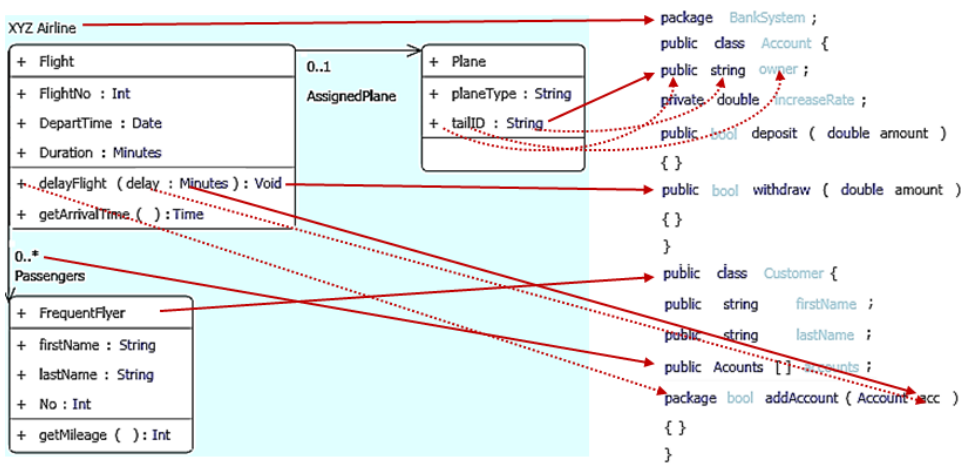
We introduce a new approach that helps to better incorporate user's domain knowledge by providing them with familiar concrete model visualizations for use during model visualization and transformation generation. This approach follows the three principles of *direct manipulation* [15], i.e. (1) it provides support for generating concrete visualizations of example source and target models; (2) these visualizations allow user interaction in the form of drag and drop of their concrete visual notation elements; and (3) interactions are automatically translated into transformation code and hence direct coding in complex transformation scripting languages is avoided. In addition, to better aid users in finding correspondences in large model visualizations, an automatic recommender system is introduced that provides suggestions for possible correspondences between source and target model elements. Complex model transformation code is automatically generated from the user's interaction with concrete visual notations and suggested recommendations.

This paper is organized as follows: Section 2 gives a motivating example, our key research questions and the requirements being addressed by the research reported in this paper. Section 3 briefly discusses key related work. Section 4 outlines our approach to model transformation generation followed by a usage example in section 5. Section 6 describes the architecture and implementation of our approach in CONcrete Visual assistEd Transformation (CONVErT) framework. Section 7 describes our evaluation and user-study setup and is followed by a discussion in section 8. Finally section 9 concludes the paper with a summary.

2. Motivation

Assume Tom, a software developer, is working in an MDE-using team and has received a system analysis report for an application. Being an expert in UML diagram interpretation and a Java coder, he is familiar with concrete syntax of the diagrams and Java code. He is interested in transforming specific parts of UML diagrams provided by the analysis directly to his programming code, to increase team productivity, code quality and to ease software evolution. For example, he wants to create a model to code translator in order to transform specific features and parts in the analysis diagrams to specific Java code templates. For Tom, as an expert in the domain, corresponding elements in the UML diagram and in his Java code are obvious. He can clearly spot and relate classes, methods, and even statement snippets in both program code and class diagram. For example, he can easily relate an attribute in a class diagram to a property in Java code and their fine-grained elements (i.e. types, names and access identifiers). Some such model element correspondences are depicted by Figure 1., using concrete visualizations of the UML model (a class diagram) and code model (Java textual syntax).

As another example, consider Jerry, an urban planner, who is preparing a report on traffic congestion in part of a city. He is used to viewing volumes of vehicles crossing intersections on screen using a geo-spatial visualization. An Example of this visualization is shown on the left side of Figure 2. Here, the volume of vehicles are represented on a map using bubbles. In his report, he would like to reflect the volume of vehicles passing set of intersections in a particular time instance by a pie chart. Being an expert in this domain, he has a solid understanding of this map-based visualization and pie charts and therefore their corresponding relationships are obvious to him. He would like to relate the number reflected to each bubble to a pie piece in a pie chart and generate new visualizations for his report as shown by Figure 2.



```
package BankSystem ;
public class Account {
public string owner ;
private double increaseRate ;
public bool deposit ( double amount )
{ }
public bool withdraw ( double amount )
{ }
}
public class Customer {
public string firstName ;
public string lastName ;
public Acounts [ ] accounts ;
package bool addAccount ( Account acc )
{ }
}
```

Figure 1: Example of correspondence relations between a Class Diagram and Java code. Dashed arrows show more fine-grained correspondences.

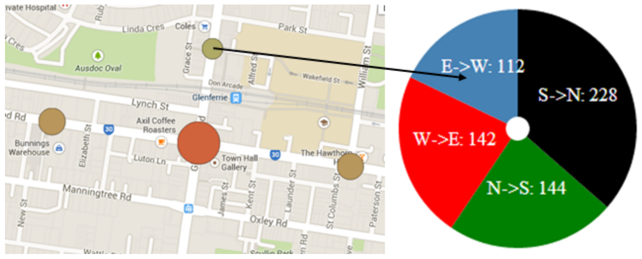


Figure 2: Example of correspondence relations between geo-located bubbles and pie pieces in a pie chart.

Given that Tom and Jerry may not have experience or knowledge of transformation languages, meta-modeling, and data processing, specifying correspondences is better understood by them using one or more example visualizations (e.g class diagrams and corresponding code examples, as shown in Figure 1). However, using current approaches to create such transformations they have to work with the complex syntax of model abstractions (e.g. UML and Java meta-models, etc.) and the low-level textual syntax and semantics of transformation languages, such as XSLT, ATLAS Transformation Language (ATL)[1] and Query/View/Transformation (QVT). Additionally, with these approaches, maintaining the transformations if the source or target models need to change is time-consuming and error-prone. While

[1]http://www.eclipse.org/atl/atlTransformations

a transformation expert may be able to do this, novices can easily make mistakes or find difficulties. If the source and target models are large, mapping is even more difficult and time-consuming, even with good tool support, such as Altova[2]. Such issues make current transformation specifications hard to perform, understand and evolve (e.g. adding or changing new transformation rules). This problem exists in many other transformation domains, for example, model to model transformations (e.g. UML to RDBMS), model refactoring, graph based transformations, and most information visualization domains.

In earlier work we developed prototype solutions to some of these problems, but for specific domains and addressing only limited parts of the problem domain. For example, the EDI message mapper using abstract EDI structure [16], the form-based mapper using a concrete business form metaphor [17], and mapping agents for large meta-model mapping problems [14]. Key research questions we wanted to answer in the work described in this paper include:

1. Can we generalize the use of such concrete source-to-target mapping metaphors to a wide range of model transformation problems?
2. Can we generate efficient and effective complex model transformation scripting code from these visual by-example specifications?
3. Can we provide effective guidance to users with visual representations and recommendations for large model mapping problems?

To answer these questions, we seek to meet the following key requirements in our approach and associated tool-set: (i) provide users with familiar concrete visualizations of source and target models in order to leverage their domain knowledge in transformation specification; (ii) allow users to specify complex model element mappings between concrete visual notational elements using interactive drag-and-drop and reusable, spreadsheet-like mapping formula; (iii) help users find, explore and decide possible model correspondences by providing automated recommendations using an interactive recommender system; (iv) allow users to cut corners in specification of transformation correspondences by choosing among suggestions; (v) automatically create high-level abstractions for transformation generation from the concrete visualizations; and (vi) generate reusable model transformation implementations from these visually specified, by-example model mappings.

[2]http://www.altova.com/

This paper presents our approach for supporting these tasks in a proof-of-concept visualization and model transformation specification and generation tool. In the following sections, we show the practicality of our transformation approach for generating transformers for a variety of domains.

3. Related Work

Complexity of model transformation specification is due to both the difficulty inherent in specifying large, complex inter-model correspondences using the complex syntax of model transformation languages [17, 18], and the use of meta-models [19]. Most model transformation approaches rely on knowledge of multiple, often large and complicated meta-models, along with expert knowledge of transformation scripting languages and tools (e.g. Eclipse Modelling Framework project [20]). While these provide powerful platforms, they are also time-consuming to maintain, hard to understand and error-prone to write [2, 3]. Multiple approaches have been proposed to address the complexity of transformation specification by eliminating the need for learning transformation languages and dealing with meta-models. These approaches can be grouped into Model Transformation By Example (MTBE), Model Transformation By Demonstration (MTBD) and model and meta-model matching.

The key principle with the MTBE approach is to derive high level model transformation rules from an initial prototypical set of interrelated source and target models. This concept was first used by Varro et al. incorporating graph transformations [6]. The idea is to provide multiple source and target model pairs, and ask a user (domain expert) to specify source and target model element correspondences. The system then uses these correspondences to derive transformation rules [1, 3, 6, 7].Model matching approaches are very similar to MTBE, i.e. they try to find possible correspondences between source and target models. These techniques try to find an alignment for relating two or more models. This alignment can then be used to semi-automatically generate transformations between two models [21, 22, 23]. These generated transformations can then be adopted and validated by an expert as a set of transformation rules. Both MTBE and model matching approaches usually require multiple source and target model examples to exist, ideally representing the same underlying data, to produce accurate transformation rules. Moreover, they often require users to modify derived rules to be executable and to match their intention which are often harder than writing the rules from scratch [18].

MTBD approaches on the other hand are based on an expert performing transformation tasks and recording of the process steps by a recorder. Using the recorded history, the system generalizes the tasks to form abstract transformations [2, 24]. MTBD approaches are more suited for transformation of models conforming to the same meta-model and as a result, are more difficult to use for different model transformation applications [1].

Kappel et al.[1] and Li et al. [17] have separately argued that to specify model transformations, an approach that involves concrete notations is required. Kandel et al. mentioned that analysts could benefit from interactive tools that simplify the specification of data transformations [25]. They asked whether transformations can be communicated unambiguously via simple interactive gestures over visualized data or if relevant operations can be inferred and suggested [25]. Multiple approaches have been used in that direction, leveraging concrete model notations for the specification and development of transformation rules and data mappings [26, 27]. However, they are usually hard coded for specific problem domains or use fixed visualizations [17]. In addition, they mostly require users to specify correspondences on meta-models rather than concrete visualizations [10, 28, 29, 16, 4]. This is also true for graph based transformations, where source and target models are represented using abstract graphs [8, 9, 10, 11]. There has been works on using graphical representations on graph-based transformation languages. For example a graphical representation of model transformations for Triple Graph Grammar (TGG) was provided by Grunske et al. [30]. Concrete syntax-based Graph Transformation (CGT) was introduced by Gronomo et al. [31]. It was suggested that due to use of graphical concrete syntax, CGT is more concise and requires considerably less effort from the modeler than , ATL and Attributed Graph Grammar (AGG) which use textual abstract syntax [31]. CGT uses a default concrete syntax similar to Business Process Modeling (BPM) and therefore the syntax is familiar for the modeler's domain knowledge. However, this concrete syntax does not have a flexibility of adapting to arbitrary visualizations.

In contrast, the approach presented in this paper uses concrete visualizations and can be adopted for variety of domains. Also, it uses model matching concepts and provides a recommender system that specifically focuses on concrete visual model representations to guide users in specifying their transformation rules. We demonstrate our approach using MDE case study of transforming class diagrams to Java code snippets. This case study and automatic model transformation and code generation in general has been an active field of research in software engineering for many years and is a necessity for MDE [32, 33, 34, 35, 36, 37]. These and other model transformations in MDE has been practiced for many years with graphical or textual languages (e.g. Henshin [33] or ATL). However, the biggest barrier for developing such code generators and transformers lies with the complexity of the model transformations and the capability of users to correctly specify the transformation rules. Previous approaches to code generation used template generators [14, 17], patterns [38], and textual and/or visual meta-models [16, 39]. We hope that by using concrete visualizations, we can provide facilities to model transformation users to better integrate their domain knowledge in transformation rule generation. Our approach generalizes to many other domains and we have applied it to several diverse data migration, data aggregation and complex information visualization problems.

4. Our CONVErT approach

Our approach to model transformation generation in CONVErT relies on example concrete visualizations of input source and target models. These specifically generated visualizations enable use of drag and drop of notations to perform model transformations and specify correspondences between source and target model visualizations. Consecutively, this concrete, by-example approach for model transformation has three key steps: (1) The user - the domain expert - provides source and target model examples and specifies (or reuses) a concrete visualization for each of the provided examples (a model to visualization transformation). These source and target model visualizations may be very different from one another e.g. a UML class diagram visualization and Java code visualization,or a map visualization and a chart visualization. (2) Using these example model visualizations, the user interactively specifies mapping correspondences between source and target visualization elements (visualization to visualization transformation). (3) We generate reusable model transformation script code from the specifications. This reusable script can be applied to any source model conforming to the example source model(s) used to specify the transformations, to produce a target model. In the following we describe each step of our approach in more detail. Then we provide a usage example of our approach being used to specify and generate model transformations.

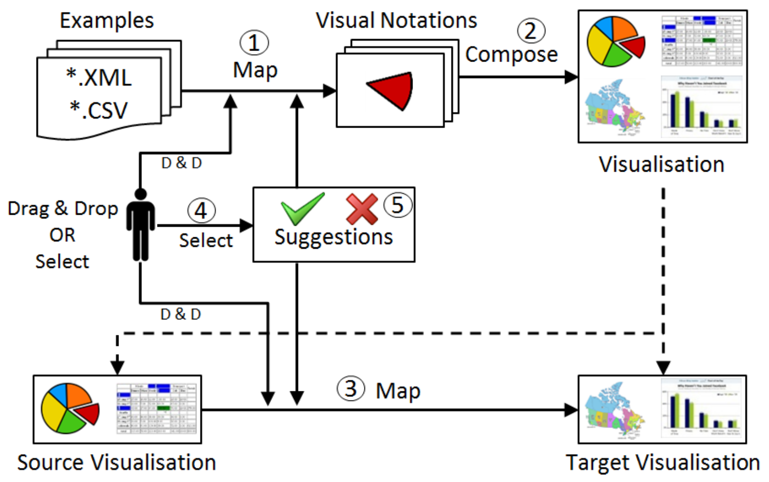


Figure 3: CONVErT's transformation approach.

CONVErT's approach is outlined in Figure 3. In step one, a user provides at least one example of source and target models. Our proof of concept implementation of the approach uses XML or Comma Separated Value (CSV) files as input examples and can be further expanded to allow other types of input. Using these example inputs a concrete visualization needs to be specified and generated for both source and target examples (or reused, if concrete visualizations have previously been specified for other example models of these types). To do this, users map elements of the source and target examples to available visual notations provided by our tool framework and specify their correspondences (see *1* in Figure 3).

Because models to transform are usually very large, a recommender system, a "Suggester", analyses the provided examples and available notations and generates a list of possible transformation correspondences to be presented to the user (see *5* in Figure 3). Our suggester uses multiple similarity heuristics, including model element name, value, structure and neighborhood similarities. It aggregates the results returned by each heuristic to generate a model element correspondence suggestion list. To specify input example to visual notation mapping correspondences, users can either drag and drop elements of input examples on visual notation elements or select from recommended correspondences (see *4* in Figure 3). When specified, users compose these visual notations to create a complete concrete visualization for models of each of the source and target types (*2* in Figure 3). These mappings result in generation of model transformation code that transforms model elements to concrete visualizations. Note that this visualization step can be specified separately by other users or be reused from previous visualizations.

Using the generated visualizations, users specify transformations between elements in the source model and elements in the target model using the concrete visualizations (see *3* in Figure 3). These visualization to visualization mappings are similarly done by dragging and dropping elements of source visualizations on elements of target visualizations or selecting from provided suggestions. For example, a pie piece in a pie chart can be dragged and dropped on a bar in a bar chart to specify a mapping correspondence; Hence the process of model transformation can use user's domain knowledge.

Correspondences between model elements include 1 to 1, 1 to many, many to 1 and many to many element correspondences. Often these transformations are quite complex. To achieve these, a variety of model transformation functions, such as collection summation, merging, subtraction, textual parsing and conditional mapping, are provided to users. In keeping with our visual, by-example transformation specification metaphor, these are also applied to example concrete visual element(s) and are composed together visually. These enable tool users to specify potentially very complex model transformation rules. If required, users can also define new functions using provided templates.

The visually, by-example defined model correspondences are then translated into low-level model transformation rules, currently implemented in XSLT. Once all required transformation rules are defined, the system generates a full source model to target model transformation script. This transformation code can be applied to any source model examples conforming to the meta-model of the example(s) used in the specification to produce a target model.

5. Use Case

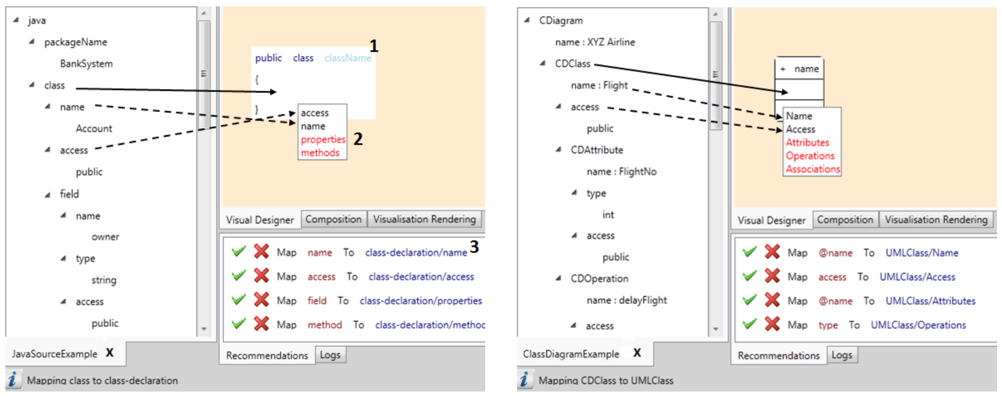
Assume Tom, a software engineer, intends to create an automatic code generator to transform specific parts of a UML class diagram model to Java code. While various IDEs and template generators support generic code generation, Tom may want to specify particular pattern implementations be used, particular code snippets be used, particular code formatting, commenting and layout, use of particular APIs be used in particular ways within the generated code base, and may want particular coding approaches be implemented. Let us further assume that he has example XML representations of both class diagrams and target Java code examples.Tom will provide these examples to CONVErT. Note that it is not required for him to possess multiple examples; instead, one comprehensive example would suffice to support the transformation generation procedure. However, having more examples will help the suggestion mechanism calculate more accurate recommendations and will help CONVErT generalize from the examples to a more complete generated translator. Specifying using several, smaller examples can also be easier for users to work with than one large example source and target model.

If meta-models are available for these example models, then CONVErT can use these for validating its input and generated target models, constraining rules and improving transformation code generation. Otherwise, CONVErT will automatically reverse engineer a meta-model from the provided example models. This meta-model is used in transformation rule templates and the suggester system.

5.1. Specifying Concrete Model visualizations

The model visualization procedure (Step 1) involves creating a visual notation for each distinct part of input model once and composing them together to form a complete visualization. For example, to visualize an example Java code XML, Tom needs to define a visual notation for Java package, Classes, attributes, methods and method parameters. To do that, he has to specify correspondence links between elements of the notation and his input. Visual notations are provided by framework users or designers and are available in framework's notation repository for reuse. Available visual designs can be imported in the framework and registered as notations by providing a notation to data mapping (See [40] for more information on CONVErT's support for specifying and generating reusable visual notations).

For example, using CONVErT to generate a visualization for a Java class, he has to drop a previously defined or reused *Java class notation* onto the CONVErT designer canvas (see (1) in Figure 4(a)). He then drags and drops the *class element* of the source example model on the notation as shown by solid black arrow in Figure 4(a). This interaction will trigger the creation of a transformation rule for transforming that portion of the source model (the *class* element in the source XML document) to the host notation's model. Each notation may have internal elements which are accessible through a pop-up window. For example, our class notation here has an access identifier, a name, a placeholder for properties and a place holder for methods as its



Map name To class-declaration/name
Map access To class-declaration/access
Map field To class-declaration/properties
Map method To class-declaration/methoc

Mapping class to class-declaration

Map @name To UMLClass/Name
Map access To UMLClass/Access
Map @name To UMLClass/Attributes
Map type To UMLClass/Operations

Mapping CDClass to UMLClass

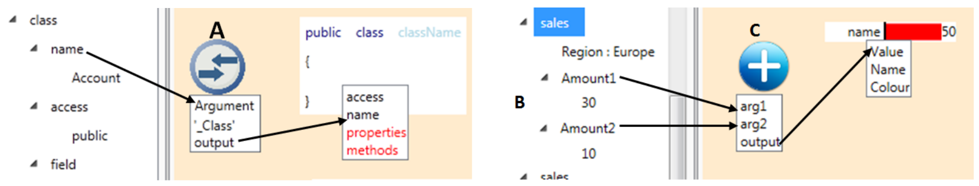
(a) Defining Java class notation. (b) Defining UML class notation.

Figure 4: Mapping example elements to notational elements to define visual notations for a) Java class and b) UML class. Arrows depict drag and drop.

model (see (2) in Figure 4(a)). These placeholders specify where other visual notation elements are going to be included.

Tom defines correspondences between the source model element(s) and target concrete visualization element(s) by drag and dropping elements or choosing from automatically-generated “suggestions” (see (3) in Figure 4(a)). In this example, Tom drags and drops the input class’s name and access to name and identifier of the notation (dashed arrows in Figure 4(a)). These correspondences will be included in the transformation template that has been triggered. Attributes and methods are defined by other model to visual notation mappings, and are specified in the same way. Thus we do not need to map them to these place holders at this stage. Once done, he will save the notation and continue creating other required notations. The same procedure is then followed for other notational elements and their visualizations. For example, a UML class diagram’s Class notation is shown in Figure 4(b).

Complex 1-to-many, many-to-1 and many-to-many element correspondences can be specified. To facilitate this and other complex mapping tasks, a range of “mapping functions” are available in CONVErT. For example, if Tom wanted to alter the name of the Java classes by appending a “*_Class*” to their name, he could use a string merging function (marked by *A* in Figure 5(a)). These mapping functions are used in a similar way to the notation elements, i.e. Tom drops the required function on the designer canvas, and links desired input elements to internal elements of the function (i.e. function’s input arguments) by drag and drop, and drags the output of the function to his desired element in the notation (see arrows in Figure 5). This forms a data-driven functional visual language transforming the source model data to the target visualization elements in potentially very complex ways.



(a) Using string merge function (b) Using summation function.

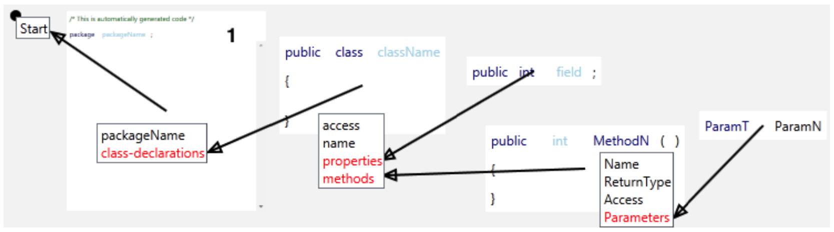
Figure 5: Using functions for input to notational element mapping. a) string merge, and b) summation function. Arrows depict drag and drop.

To get a sense of range of supported model transformations and different domains, assume analyst Carrie is generating a bar chart visualization for her sales report in a business analysis visualization domain. Each bar in this bar chart is to represent a record of annual sales. Further assume the report being handed to her includes six monthly sales amounts instead of annual sales (marked by *B* in Figure 5(b)). She can use a summation function (*C* in Figure 5(b)) to add those two amounts and map the result to Bar's value.

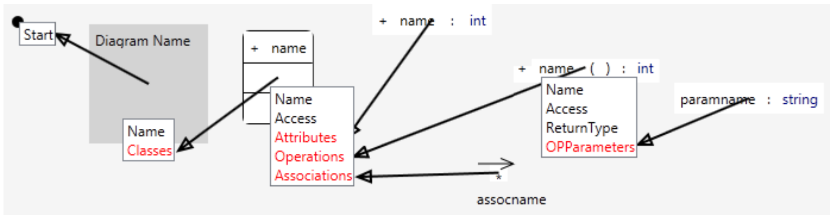
The defined notations represent a model element-to-visual notation transformation rule. To have a complete transformation script, the prepared collection of transformation rules in notations need to be scheduled for processing on source models according to their call sequence. Usually this is achieved by asking users to write code for this script, similar to procedural programming, and by providing meta-models. In our approach the assumption is that there is no user provided meta-model available and the user is not willing to write complex transformation code. Therefore, by using composition of notations our approach generates call sequencing of the embedded transformation rules.

To compose the defined notations in our example, Tom links all notations he has created according to their specific place holders as depicted by Figure 6(a). By linking a notation to a placeholder element of another, the host notation knows the transformation rule embedded in the notation being dragged should be called at this placeholder. This is in order to affect the embedded model element-to-visual notation mapping. This linking results in the scheduling of model element-to-visual notation transformation rules and thereby creates the model to visualization transformation script.

A *Start* element in Figure 6 defines where the transformation specifica-



(a) Java code notation composition.



(b) Class diagram notation composition.

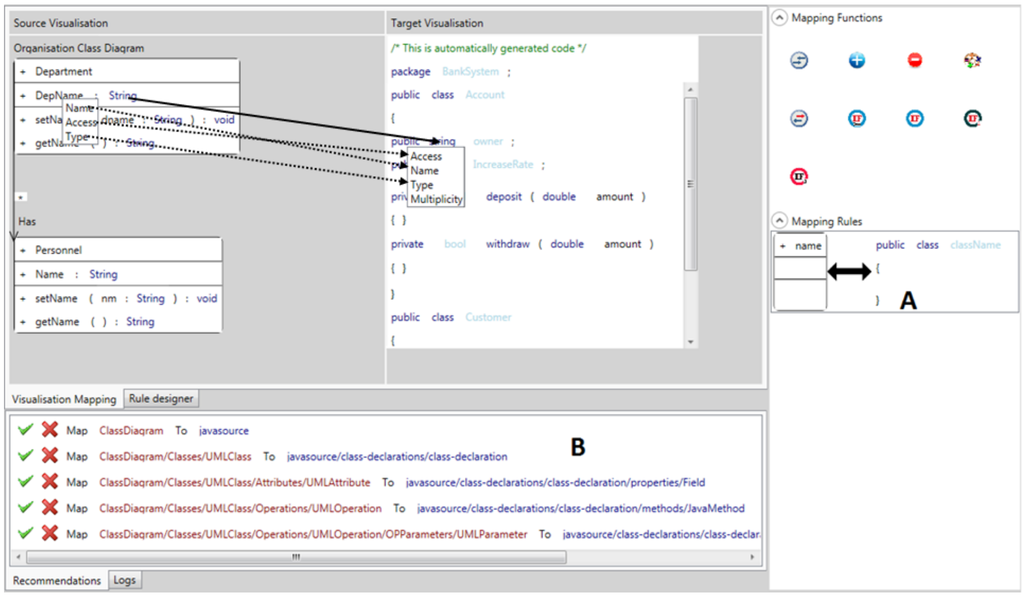
Figure 6: Notation composition to generate visualizations for a) Java code and b) class diagram. Arrows are provided by the framework for better tracking of the composition.

tion will start and thus tells the transformation scheduler to start generating transformation code from the transformation rule embedded in the notation linked to this start element. For example in Figure 6(a), the transformation rule in package notation (marked by *1*) is the first rule to be called to transform a Java package model element to a package notation. It then calls the class transformation rule, and the scheduling continues accordingly for other linked notations.

Each notation composition results in the automatic generation of a transformation specification, currently implemented as XSLT transformation script that generates Windows Presentation Foundation (WPF) visual elements. For example, by using the compositions specified in Figure 6(b), a complete XSLT script to generate concrete visualizations of UML class models will be generated for rendering class model examples to visualizations of the form in Figure 1. Note that this generated XSLT transformation script can be reused and applied to all Java code and class diagram model files conforming to the examples used to specify the visualizations, to provide an automatic concrete visual notation renderer. These generated concrete class visualizations are implemented as WPF elements and allow interaction with their composing notations. The individual elements of a concrete visualization can thus be dragged, dropped on other elements, and right clicking on them reveals their internal elements.

5.2. Specifying Model Mappings using Concrete Visualizations

Once visualizations of both source and target are available, Tom can drag and drop elements of these visualizations to create transformation rules between these visualizations. An alternative to this approach is to select from provided suggestions (see *B* in Figure 7). Figure 7 shows an example of creating a transformation rule for a UML attribute to a field in Java code. To create this rule, Tom needs to drag a UML attribute to a Java field, as depicted by solid black arrow, and match their internal elements, as shown by dashed black arrows (or select them in suggested recommendations). Note that the two visualizations do not need to represent same data, as in Figure 7 where the class diagram represents an organization system but the Java code is representation of a Bank system package. Although consistency checking was not our concern with this prototype implementation, selective checks can be provided in each visualization. For example, the package name of the Java code knows that its name should not consist of white spaces, or Java attributes use default multiplicity of *1* when not specified and when blank is provided it is assume to be *n*. These checks can be provided depending on the application during notation design. An alternative is to use functions and conditions when specifying the transformations.



```
/* This is automatically generated code */
package BankSystem ;
public class Account
{
public String owner ;
private IncreaseRate ;
private deposit ( double amount )
{ }
private bool withdraw ( double amount )
{ }
}
public class Customer
{
```

✓ ✗ Map ClassDiagram To javasource
✓ ✗ Map ClassDiagram/Classes/UMLClass To javasource/class-declarations/class-declaration
✓ ✗ Map ClassDiagram/Classes/UMLClass/Attributes/UMLAttribute To javasource/class-declarations/class-declaration/properties/Field
✓ ✗ Map ClassDiagram/Classes/UMLClass/Operations/UMLOperation To javasource/class-declarations/class-declaration/methods/JavaMethod
✓ ✗ Map ClassDiagram/Classes/UMLClass/Operations/UMLOperation/OPParameters/UMLParameter To javasource/class-declarations/class-declar

Figure 7: Mapping A UML attribute to Java property. Arrows depict drag and drop directions.

Given that transformation rules are defined using concrete notations, each rule is also represented by the source and target notations it is representing. This provides better visual representation of transformation rules. For instance in Tom's example above, the visual representation of the rule transforming a UML class to a Java class is marked by *A* in Figure 7.

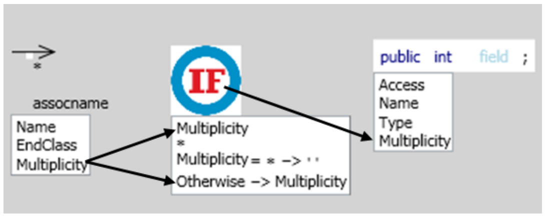
Similar to step two of specifying a concrete visualization for a model, mapping functions are available to create more complex transformation rules between the concrete visual model mappings. For example, when mapping associations to a Java field property, an association might have multiplicity defined by "*", whereas a Java field property might have either a number or void as its multiplicity. To specify such a correspondence, Tom can use a condition mapping function.

By dragging a UML attribute to a Java field (or any notational element to another) their default notations will be shown on a different canvas to better provide space for using functions. In this example, Tom can navigate to that canvas and specify conditions as depicted by Figure 8(a). The condition function in the figure tests whether *Multiplicity* of the association is equal to *'*'*. If so, it passes a blank character as output; otherwise it copies the *Multiplicity* provided by the association to the output. Since conditions do not have specific output, (unlike arithmetic and processing functions), instead of dragging the output he drags the condition itself on the element of the target (in this case Java field's multiplicity) as depicted by arrows in Figure 8(a). He can continue specifying other correspondences (Associations *Name* to Java field *Name* and *EndClass* to *Type*) here or on the actual visualizations. The transformation code generated from these interactions is shown in Figure 8(b).

5.3. Recommending correspondences

In our experience, real-world source and target models are often very large [14]. To assist the user, model correspondence recommendations are provided by a group of *Suggesters*, or correspondence recommenders, which analyze source and target models according to a variety of value and structural similarity heuristics. Since analyzing the whole input models and visualizations was costly for our automatic correspondence recommender, our suggester system uses the abstract graph lattices (used in reverse engineering meta-models) as input to calculate similarities.

Three types of similarity heuristics are used as correspondence recommenders of the suggester system. (1) Static similarity recommenders check name tag and type of elements in source and target elements. (2) Structural



(a) Mapping UML Associations to Java field

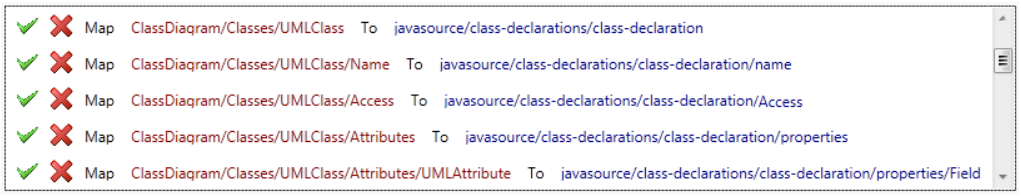
```
<xsl:template match=”UMLAssociation”
 xmlns:xsl=”http://www.w3.org/1999/XSL/Transform”>
 <Field>
  <Access>public</Access><!−−default−−>
  <Name>
   <xsl:apply−templates select=”Name” />
  </Name>
  <Type>
   <xsl:apply−templates select=”EndClass” />
  </Type>
  <Multiplicity>
    <xsl:choose>
     <xsl:when test=”Multiplicity=’∗’”>’␣’</xsl:when>
     <xsl:otherwise>
      <xsl:value−of select=”Multiplicity”/>
     </xsl:otherwise>
    </xsl:choose>
  </Multiplicity>
 </Field>
</xsl:template>
```

(b) Generated XSLT script

Figure 8: Using conditions in visual notation mapping, a) a condition to map UML association to Java field, arrows depict drag and drop. b) The resulting XSLT script.

similarity recommenders consider source and target as graphs and check similarities based on structure of this graph that elements reside in, i.e. checking the inbound or outbound nodes of each element, and checking the neighborhood of each source and target element. (3) Propagated similarity (like structure similarity recommenders) considers input source and target models as graphs and calculates similarity of elements according to recursive analysis of their neighboring elements. With this similarity measure, similarity of two nodes in a graph is defined by similarity of their neighborhood topology. As a result, using propagated similarity, two nodes are similar if their neighbors are similar and the neighbors of their neighbors are similar and so on. We have adopted IsoRank as our measure for propagated similarity recommender [41].

Figure 9 shows sample list of recommendations produced by our suggester for the motivating example of UML class diagram to Java code visualization. In this figure for example, a name similarity recommender has assigned high scores to UML class Name to name of a Java class and UML class's Access to Java class's access as they represent similar name tags. UMLClass and class-declaration have high similarity score according to IsoRank similarity as their neighbors (children and parents) are similar (e.g. UMLClass children include Name and Access, Java class declaration also has name and access as it's children and so on). As a result they have been returned in the final list of recommendations. Note that other recommenders might have also contributed to these recommendations. For example, both Name elements on either side have values that are *string* and the type similarity recommender will also return these pairs as high score correspondence candidates.



✓ ✗ Map ClassDiagram/Classes/UMLClass To javasource/class-declarations/class-declaration

✓ ✗ Map ClassDiagram/Classes/UMLClass/Name To javasource/class-declarations/class-declaration/name

✓ ✗ Map ClassDiagram/Classes/UMLClass/Access To javasource/class-declarations/class-declaration/Access

✓ ✗ Map ClassDiagram/Classes/UMLClass/Attributes To javasource/class-declarations/class-declaration/properties

✓ ✗ Map ClassDiagram/Classes/UMLClass/Attributes/UMLAttribute To javasource/class-declarations/class-declaration/properties/Field

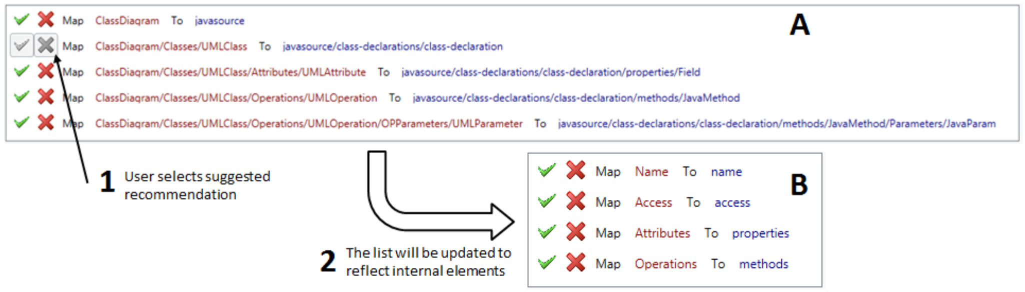
Figure 9: Sample of resulting suggested correspondences between UML class diagram and Java code visualization.

The scores calculated from the similarity recommenders of each group are returned as a normalized similarity matrix. Similarity matrices are sent to the Suggester system and a final similarity score is calculated based on the confidence weights assigned to each recommender in the suggester's recommendation ensemble. Similarity scores returned by suggesters are multiplied by their confidence weights and the resulted scores are summed up in a final similarity matrix that is the basis for calculation of recommended correspondences. The suggester system selects from the returned suggestions and prepares a recommendation list similar to Figure 9. Once all recommendations are available, our ensemble configuration filters the recommendation list by the stable marriage algorithm [42]. This will result in a selection of recommendations that possess the highest overall recommendation score per pair. The stable marriage algorithm can be configured to return arbitrary number of results per pair. By default this value is set to one. If users accept or reject any of the recommended correspondences a feedback analyzer updates the confidence weights associated with the suggesters and thus improves the learning mechanism.

The suggester system can be configured to use one, all or a selection of these different suggesters by provided option settings. For example, users can configure the suggester system to ignore a particular recommender by setting its usage flag to false. Also, it is possible to alter weights manually (or as part of an optimization mechanism) by setting each recommender's confidence weights and disabling the learning mechanism (automatic alteration of confidence weights).

To provide a more interactive and hence useful representation of recommendations, the suggester system incorporates a filtering mechanism similar to the *guide* and *filter* approach proposed by Hernández del Olmo et al. [43].

In their proposal, a *guide* provides answer to when and how each recommendation must be shown to the user, while the *filter* must answer which of the items are useful/interesting candidates to become recommended items.


A
1 User selects suggested recommendation
2 The list will be updated to reflect internal elements
B

✓ ✗ Map ClassDiagram To javasource
✓ ✗ Map ClassDiagram/Classes/UMLClass To javasource/class-declarations/class-declaration
✓ ✗ Map ClassDiagram/Classes/UMLClass/Attributes/UMLAttribute To javasource/class-declarations/class-declaration/properties/Field
✓ ✗ Map ClassDiagram/Classes/UMLClass/Operations/UMLOperation To javasource/class-declarations/class-declaration/methods/JavaMethod
✓ ✗ Map ClassDiagram/Classes/UMLClass/Operations/UMLOperation/OPParameters/UMLParameter To javasource/class-declarations/class-declaration/methods/JavaMethod/Parameters/JavaParam

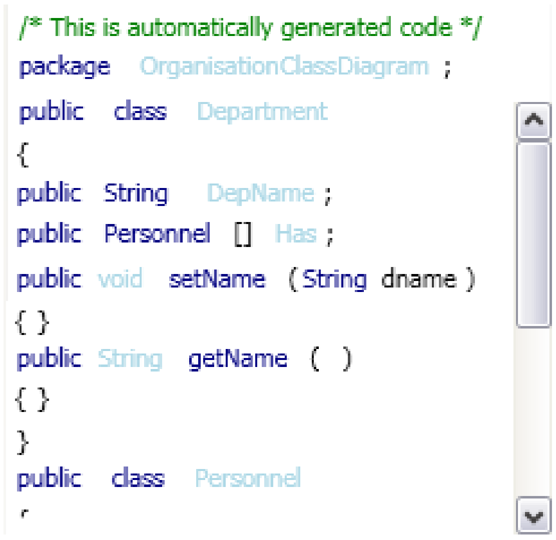
✓ ✗ Map Name To name
✓ ✗ Map Access To access
✓ ✗ Map Attributes To properties
✓ ✗ Map Operations To methods

Figure 10: Adaption of *guide* and *filter* for interaction with suggested correspondences. Selecting (accepting) a UML class to Java class correspondence (1) updates the recommendation list to show possible internal element correspondences (2). Rejecting the correspondence will not update the list but will result in updating the recommender weights.

In our adaptation of that proposal, the results of final similarity matrix are *filtered* by the stable marriage and sent to the *guide* system for representation. Some correspondences will result in transformation rules, and others will define internal rule correspondences. The *guide* system chooses among recommendations according to the task that the user is about to perform, e.g. when user provides source and target visualization to perform mappings, the *guide* system first represents the recommendations that will result in transformation rule templates. That is because a rule between two notations must be defined first, and its internal rule correspondences are to be defined later. Therefore, when users define a rule correspondence by drag and drop or selecting from the suggesters, the system accordingly updates the list of suggested correspondences to provide suggestions related to that rule and hence better guide users with targeted recommendations. For example, if Tom defines a UML class to a Java class rule by accepting its recommended suggestion (as in Figure 10 *A*) or alternatively by drag and drop of their visual notations, the suggestion list will be updated to demonstrate how internal elements of classes (like name, access, attributes, etc.) can be linked (See *B* in Figure 10). This intelligent assistance was incorporated after our user study indicated the need for more interactive and targeted visualization of recommended correspondences.

5.4. Generating Model Transformation Scripts

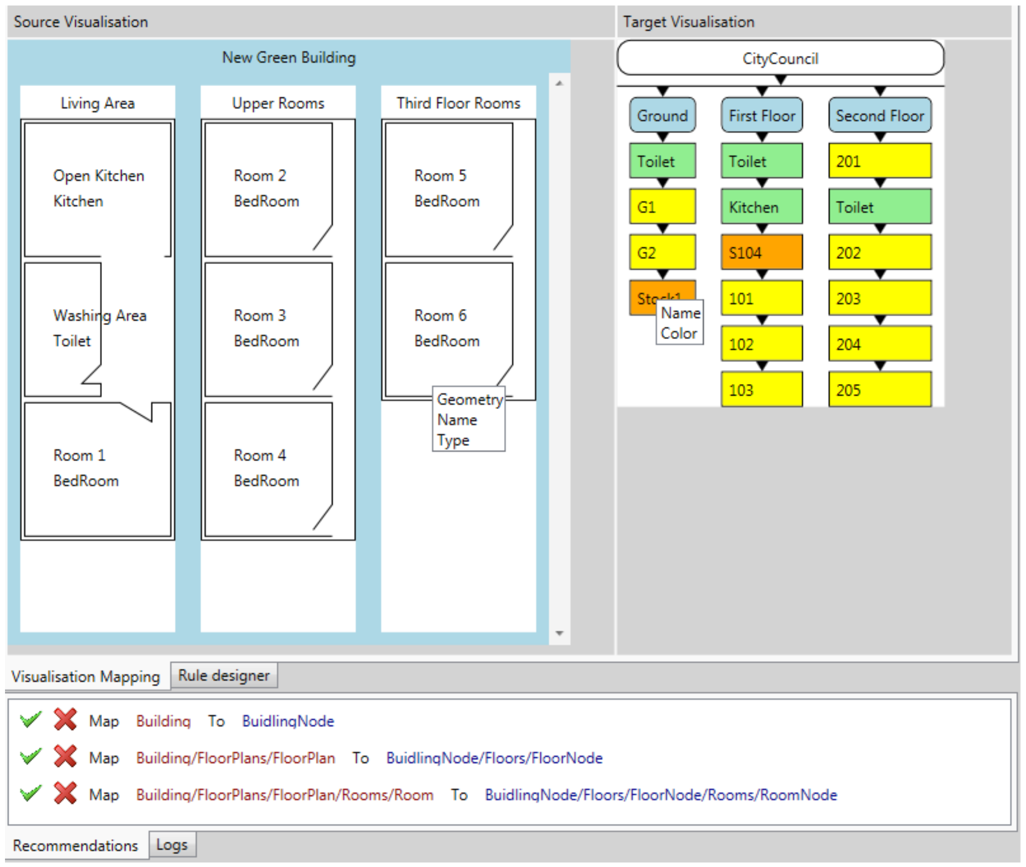
Once the required rules for transforming all parts of source and target visualizations are defined, a transformation code generator generates XSLT scripts for each transformation rule in the form of an XSLT template similar to the transformation script of Figure 8(b). As stated before, it is possible to generate scripts for other transformation languages with some modifications in the transformation code generator. These modifications would specify how a correspondence from an element a in source to an element b in target should be written for that specific transformation language. In current configuration, depending on what the correspondences specify (an element to element mapping, a notation to notation mapping, etc.) these correspondences are translated to XSLT value fetches (value-of and select statement) or template snippets.



```
/* This is automatically generated code */
package  OrganisationClassDiagram ;
public  class  Department
{
public  String   DepName ;
public  Personnel  []  Has ;
public  void   setName  ( String  dname )
{ }
public  String   getName  (  )
{ }
}
public  class  Personnel
```

Figure 11: Sample resulting Java visualization.

Although the model to visual notation transformation rules of the visualization step had to be explicitly scheduled, in visualization to visualization transformation generation the rules are declaratively executed. This is due to the fact that a visualization example of both source and target are available and their meta-model can be reverse engineered. Therefore, it is possible to decide the starting rule for the transformation script. Other transformation rules following the starting rule are then declaratively called. As a result, once required transformation rules are defined, the system will generate the



✓ ✗ Map Building To BuidlingNode
✓ ✗ Map Building/FloorPlans/FloorPlan To BuidlingNode/Floors/FloorNode
✓ ✗ Map Building/FloorPlans/FloorPlan/Rooms/Room To BuidlingNode/Floors/FloorNode/Rooms/RoomNode

Figure 12: Using CONVErT to transform CAD drawings to tree-based layout.

transformation script automatically. Applying this script to source visualizations will transform the data represented by source to visualization of the target. For example, in Tom's example, applying the full transformation script on the source of Figure 7 will result in the visualization of Figure 11. Note that Tom can very easily modify the Java example visualization and mappings to: rearrange attributes and methods; generate differently-named classes, methods and/or attributes; to reformat their concrete appearance; to add particular design pattern, API usage, code formatting, layout, and partial code templates; could apply filtering to the source UML model mappings to only generate code for selected portions of the UML model; and so on.

5.5. A CAD Model Transformation Example

To illustrate the range of model transformation domains to which our CONVErT approach can be applied to, consider Computer Aided Design (CAD) applications that need to exchange complex models [44]. Consider the scenario where architect Carrie might want to create an organization's building structure chart based on the design she had created earlier. Assume that visualization transformations for both models have been provided beforehand (they can be created using the same approach as in the previous example), where the design model is visualized with a 2D building layout and the structure chart model via its diagrammatic representation. Carrie can specify a transformation between elements of her source design to elements of the target structure chart. As she is an architect and not a software engineer, and the fact that CAD designs can become very large and complex, she can view both visualizations side by side and get help from suggested correspondences.

Figure 12 shows an example of mapping parts of a detailed building design to a detailed structure chart. Carrie can specify elements of the chart structure to be created based on elements in her design. For example, she can drag and drop a room on a corresponding room node in the tree and specify their internal elements. The color of tree node can be specified using functions and based on type of the room. She uses CONVErT in the same manner as described in the previous section, making use of suggestions, as each of these model structures is large and each example visualized is also large. Carrie can use examples of part of the building model to specify her transformation to a corresponding part of the structure chart. CONVErT then generates a model-to-model translator that can be applied to complete building design models to generate a complete structure chart.

6. Architecture

Our new approach to concrete visualization generation and model mapping generation as presented in this paper has been implemented as a proof of concept in our CONVErT framework [45]. A high-level architecture and key parts of CONVErT are depicted in Figure 13. In the following paragraphs, we briefly describe the implementation of key mechanisms provided by this framework.

The reverse engineering and model abstraction mechanism of CONVErT (Figure 13 (1)) uses a graph lattice as meta-model to be used in transforma-

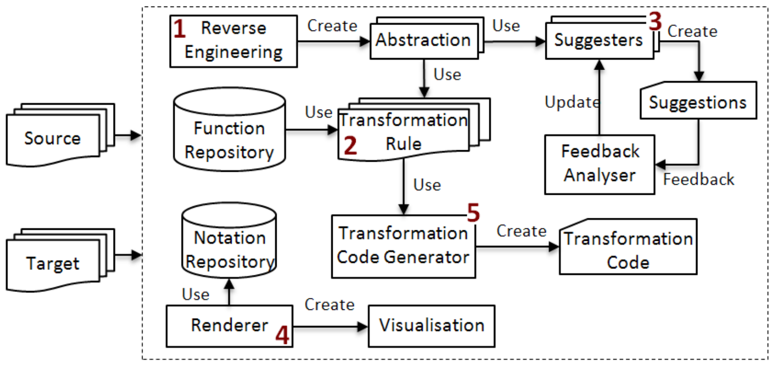


Figure 13: CONVErT's architecture.

tion rules and for scheduling. Once source and target examples are provided, a crawler traverses the examples and fills the lattice with new element structures that it faces. This way the structure of source and target are known to the system. Each transformation rule (Figure 13 (2)) will be initially created by considering a portion of this structure that represents the element being dragged or dropped as a template. As a result, once an element is being dragged, the abstract structure of the input example it represents is also dragged with it. The specified source to target correspondences will be inserted in these templates. Once all correspondences are specified, the transformation code generator (Figure 13 (5)) uses these correspondences and forms XSLT snippets that will be inserted in the template. The altered template will then be used by transformation code generator to generate a transformation rule in XSLT. If an alternative transformation language is needed, translation of these templates to the target languages should be provided to the system.

Each visual notation in CONVErT has a view created and provided by XAML and a Model which is an XML description of the internal elements. A direct mapping is provided to transform notation's model XML to the XAML representation. Since the generated concrete visual notations need to provide interaction (drag and drop) capabilities and host transformation templates, the Renderer mechanism of CONVErT (Figure 13 (4)) wraps each notation in interaction logic provided by an instance of a Visual Element (VE) class. A VE provides a container for the notations and other VEs and is implemented using XAML and C#. This architecture allows our framework to let users interact with composing elements of a model visualization regardless of the embedding hierarchy of the notation.

Our prototype implementation of CONVErT was done using Microsoft Visual Studio and C#. The rendering of XAML based visualizations is native in Windows Presentation Foundation (WPF), available in Visual Studio. Updated versions of CONVErT can be downloaded on-line[3].

7. Evaluation

We have used CONVErT to specify a wide range of complex model transformation and information visualization problems. These include several MDE problems using UML source models and code and script targets; CAD tool integration using 2D and tree structure source and target visualizations; and various business analysis problems, including a Minard's Map visualization. Details can be found on CONVErT's website[3] including example videos of specifying CONVErT transformations using concrete, by-example visualizations. This section describes our evaluations of CONVErT, which consist of a user study and evaluation of the suggester system.

7.1. User Evaluation

We wanted to get detailed target end user feedback on the CONVErT approach and our prototype concrete, by-example model transformation tool. To do this we designed a user study where users performed a number of model transformation and comprehension tasks with CONVErT.

7.1.1. Participants and Tasks

For our user study, we recruited 19 users (including 4 controls for instrument testing) in two groups from software engineering staff and students at Swinburne University of Technology. Participants were introduced to CONVErT through a 10 minute screen-cast which described the user interface, visualizations and transformation generation procedure. They were then asked to perform a set of given model visualization and mapping tasks and were asked to use a think-aloud approach. The experimental setup comprised a laptop with an attached mouse. Screen captures were taken during the process and a matching questionnaire with 58 questions was handed to each participant at the end of the experiment with 5 point Likert scale (ranging from strongly disagree to strongly agree) and dedicated spaces to leave comments and optional feedback.

[3]https://sites.google.com/site/swinmosaic/projects/convert

The tasks assigned to each group were to create a visualization with CONVErT and then use it as source and generate a transformation from the example visualization to a provided visualization. Both groups had the same settings but used different input examples and target visualization. The first group were given a model representing business sales data and were asked to create a bar chart visualization of their sales data (task 1). They were then asked to transform that bar chart visualization to a pie chart visualization (task 2). The second group were given a class diagram data (XML) and asked to generate a class diagram visualization (task 1). For Task 2 they were asked to transform that class diagram visualization to a provided Java code visualization (similar to usage example of Section 5). Task description hard copies which were handed to the participants did not describe instructional steps. Instead, they included the input file names and their locations, and a snapshot of the desired final visualization and transformation results. Users had to come up with steps required to get similar results. They were allowed to ask questions from the instructor if they had trouble understanding those steps.

Our first group consisted of 10 participants (8 male, 2 female). Second group consisted of 5 participants (3 male, 2 female). In response to demographic questions **D.3**: "How familiar are you with model transformation and modeling in general?" and **D.4**: "How familiar are you with data visualization?", the participant had following options: **VF**: Very familiar, **SF**: Somewhat familiar, **HH**: Had heard of it, and **NF**: Not familiar. The frequency of responses are provided in Table 1.

Table 1: Partial demographics of participants (%)

Question	**NF**	**HH**	**SF**	**VF**
D.3	13	33	47	7
D.4	13	20	53	13

7.1.2. Results

Table 2 shows a selection of eight questions from our questionnaire. We have assigned scores of 1 (for perfect negative) to 5 (perfect positive) to each Likert point and calculated the Median, Mode and Frequency of responses. The responses to sample questions based on these arrangements are also summarized in Table 2. Full results can be found on CONVErT's website[3]. It took participants on average 29 minutes to accomplish both tasks successfully. Section 8 provides a discussion of these results. We also asked open ended questions about the use of the tool on these examples, suggestions for improvements, and overall impression of the approach for model transformation problems.

Table 2: Sample questions of the questionnaire.

	Question	**Frequency (%)**				
		1	**2**	**3**	**4**	**5**
Q.1	I found it easy to visualize the given data as a bar chart/class diagram.	0	0	20	7	73
Q.2	I learned to use the tool quickly.	0	0	13	27	60
Q.3	Visual diagrams help me better understand the relationships between source and target drawings.	0	0	0	20	80
Q.4	I found it easy to specify the relations between left hand side and right hand side visualizations.	7	0	7	20	67
Q.5	In general I found the tool to be easy for transformation between visualizations.	7	0	0	40	53
Q.6	Recommendations helped me better understand relations between source and target visualizations.	7	0	33	27	33
Q.7	I used recommendations at least once.	7	7	27	20	40
Q.8	I was satisfied with the way recommendations were presented.	7	7	27	13	47

As Table 2 indicates, the majority of participants agree on ease of use of the tool for generating visualizations. This is reflected in their responses to questions **Q.1**. Similarly, the responses indicates that the participants found it easy to learn the tool (see their response to question **Q.2**, 60% strongly agree and 27% agree). As can be seen from results of Table2, users positively responded to having visualizations in better understanding of relationships between source and target (**Q.3** with 80% strongly agree and 20% agree). These responses demonstrate users' perception of the approach is in accordance to our motivating scenario on usefulness of concrete visual by example approach for generation of mappings. In terms of ease of use, the responses are fairly consistent and indicate general acceptance of the approach and tool-set (see responses to questions **Q.4** and **Q.5**). In response to **Q.4**, total of 87% of the users have agreed that it was easy to specify the relations between left hand side and right hand side visualizations which complements responses to question **Q.3**. Similarly 93% of the users have agreed that the approach provided by the tool for specifying transformation using visualizations was easy (**Q.5**).

The results of Table2 shows potential points of improvement to the approach and specifically to they way recommendations are represented and used. For example, when we asked whether provided recommendations helped users understand relations between source and target visualizations (**Q.6**) only 60% of the users have responded agree and strongly agree. Similarly in question **Q.7**, 60% of participants have agreed that they have used recommendations at least once. In terms of recommendations representation, we have also received 60% satisfaction (in response to **Q.8**) which could be a clue to why users did not use the recommendations and preferred drag and drop to specify correspondences. For example, a participant did not realize that by selecting from suggestions, it is possible to specify correspondences and therefore did not use them at all.

We should also point out that the guide and filter mechanism described in section 5.3 was not implemented in the version of the tool used for evaluation. Users were provided with a list of recommendations instead and to select a recommendation they had to traverse the list to find correspondences. This proved to be problematic and motivated us for implementation of the guide and filter mechanism. We still believe that better representation of recommendations (perhaps by highlighting them in the visualizations) helps improve accessibility and usability of recommendations.

7.2. Recommender evaluation

We wanted to evaluate our CONVErT suggester to see how well it performs in suggesting possible correspondences for large example models. Therefore, we slightly altered its design for this task to be able to evaluate it as a batch model matcher. This modification includes separation of the suggester

Table 3: Categorisation of possible recommendations.

	Recommended	**Not Recommended**
Relevant	True-Positive (TP)	False-Negative (FN)
Irrelevant	False-Positive (FP)	True-Negative (TN)

system from CONVErT's GUI and disabling the guide and filter representation mechanism, so that source and target model examples can be processed as a whole. This way, we would be able to evaluate how accurate are the recommended correspondences using a set of available source and target examples and against a benchmark. For this task the suggester was used to check source and a target model examples and provide recommendations for possible correspondences between all elements of the examples. The suggester was used in its initial default setting, i.e. it was configured to give one recommendation per pair and all recommender confidence weights were set to neutral (one in this case) for each set of source/target examples.

We used model and schema matching examples from the Illinois Semantic Integration Archive[4] to test our CONVErT suggester. We have chosen the house listing information from real estate websites. This selection has been based on availability of examples in the dataset and their intended application, i.e. for testing schema matching techniques. The provided dataset in the archive represent house listings for different websites (e.g. Yahoo and Home Seeker). Each dataset includes set of house listings that are formated according to the specific website requirements. The datasets do not represent same information, rather each provides a different set of house listings.

To test these examples with CONVErT's suggester, a correspondence benchmark was developed including all correct correspondences of the examples. Table 3 shows possible categories of recommendations. Using this table, a *Relevant* correspondence is a correspondence available in the benchmark. If this correspondence is recommended, then it is considered as a *True-Positive* correspondence recommendation. If the correspondence is not recommended, it will count as a *False-Negative* correspondence recommendation and so on.

Using the categories of Table 3 and Equations 1 to 3, we calculated Precision (Prec.), Recall (Rec.) and F-Measure (F-M). These metrics were used as they represent most common metrics for evaluating recommender systems

[4]http://pages.cs.wisc.edu/∼anhai/wisc-si-archive/

[46]. We have applied our suggester system to multiple different combinations of the datasets. For example, suggester system was applied to the problem of matching examples of Yahoo house listings to Home Seeker's house listings. Table 4 provides results of some of these example evaluations. Other combination of these examples are available on-line[3].

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall\ (True\ Positive\ Rate) = \frac{TP}{TP + FN} \tag{2}$$

$$F\text{-}Measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3}$$

Table 4: Evaluation results of the Suggester system.

Example	**Default**			**Optimized**		
	Prec.	Rec.	F-M.	Prec.	Rec.	F-M.
NKY - Texas	0.3	0.78	0.44	0.94	0.7	0.8
Yahoo - NKY	0.34	0.73	0.47	0.81	0.6	0.7
Yahoo - Home Seeker	0.44	0.74	0.55	0.92	0.63	0.75
Home Seeker - NKY	0.51	0.86	0.64	0.9	0.81	0.85

Based on Table 4, our Suggester has performed relatively poor on the default setting and for the first calculation of recommendations. For example, the NKY to Texas matching example has achieved precision of 0.3 and recall of 0.78. This indicates that the majority of true recommendations have not been produced. This is due to use of different example-specific naming convention, typing, structure and sizes for the source target example pairs. Also, it was not possible to provide correspondences for certain source and target pairs as in numerous occasions the models did not have corresponding elements. For example, the NKY dataset provides dimensions (given as X $\times$ Y) for each listing whereas Texas provides lot size in square meters. Or the NKY includes an element for basement whereas other examples do not provide such elements. Since the suggester gives a correspondence for all pairs, it has had effects on our results.

However, it is possible to configure the suggester system to use certain recommender(s) by using the provided configurations, i.e. users can alter the suggester to use different combination of certain recommenders. Using these configurations and different combinations, it is possible to design an optimization mechanism for the suggester system to optimize its recommendations for better precision and recall. Consecutively possible combinations of recommenders were considered as the search space and the resulted recommendations of different recommender combinations were examined against the benchmarks. The best result was reported as optimized combination configuration. The optimized results are also available in Table 4 and show significant improvements. For example, the precision of NKY to Texas example has improved from 0.3 to 0.94 and its recall has been slightly improved from 0.78 to 0.7. This shows that if previous knowledge or benchmarks of the examples are available, users can customize the suggester system to provide more improved recommendations for certain metrics, e.g. precision.

This optimization is time consuming and may not suit the purpose of our recommender. Also, it may be beneficial to use fuzzy configurations using confidence weights rather than the provided configuration settings, i.e. give a certain recommender less importance (e.g. setting the confidence weight to 0.2) and giving another more credit (e.g. by setting the confidence weight to 0.8). Although this can be achieved by altering the optimization to provide fuzzy confidence values, our experience with CONVErT showed that after continues usage of the recommender (accepting/rejecting) on similar examples, its learning mechanism automatically converges the confidence scores to close to optimized configuration. For example, for the class diagram to Java code example we could achieve Precision of 1.0 after five usage iterations.

8. Discussion

Our user study demonstrated that on average users positively liked the idea of concrete visual transformation. See for example their answers to question **Q.3** in Table 2 where all users highly rated (4 or 5) the use of visual diagrams in understanding the source-to-target relationships and question **Q.4** where 87% of the users rated the ease of use of the approach high or very high (67% very high and 20% high). However, certain drawbacks of the used version of prototype tool affected user experience. Specifically, some users could not differentiate model elements and placeholders as they were represented similarly for each visual notation by earlier versions of the framework. This resulted in confusion, and a user had to ask the instructor after a mistake was made in notation composition. Out tool support for CONVErT was updated to reflect this and show place holders and notation elements separately (see for example compositions of Figure 6).

In comparison, our suggester mechanism achieved a lower user acceptance than the visualization and transformation specification. This has been reflected in users responses to questions **Q.7** and **Q.8** in Table 2. Hence, our third research question - "Can we provide guidance to users with visual representations and recommendations for large model mapping problems?" - is not completely addressed. We believe this is due to the above mentioned representation of recommendations and the fact that the given examples were sufficiently simple; therefore users already knew most of the correspondences, and thus did not need to utilize its potential most of the time. For example a participant stated that the mapping correspondences were "easy to find and specify" and therefore felt no need to use them. Provided that the visualizations were more complex, it would have evaluated effects of the suggestion much better. One participant did not realize that by selecting from suggestions, it is possible to specify correspondences and therefore did not use the recommendations. Also, the way recommendations are currently presented was not well received by users and some users found it hard to find the presented recommendations in visualizations and accordingly accept or reject them. This is potentially due to current use of element hierarchy in representation of left hand side and right hand side elements. For example a UML class would be represented as *ClassDiagram/Class/UMLClass*. As a result better representation of recommendations is being considered for future CONVErT versions using interactive highlighting in visualizations [14].

While writing transformation code, a transformation designer might partition the code into several modules (e.g. transformation rules) or might write the transformation as one module. It is a common software engineering practice to modularize the code to help better readability and easier maintenance. Although readability of the automatically generated transformation code in our approach was not considered in the design, it demonstrates an acceptable modular structure, i.e. the code is divided into separate transformation rules (in this case XSLT templates). This is due to the fact that each notation-to-notation mapping is considered as one transformation rule template. As a result, if need be to reuse the generated code outside framework, it exhibits acceptable readability specially for large transformations.

8.1. Threats to validity

Internal: Four of our participants mentioned the effect of learning during the experiments. They admitted that since the drag and drop tasks ware being repeated for tasks one and two, they could perform the second task easier. This might have had effects on better acceptance of the approach for task two. Some participants may have been reluctant to ask questions regarding the items being asked in the questionnaire and therefore responded based on their understanding of the questions.

Our suggester system uses the results returned by name similarity recommender when deciding which neighbors are similar. As a result accuracy of structural and propagated similarity recommenders are dependent on name similarity. This can have effects on their accuracy in situation where names of source and target elements are not similar.

External: The users whom participated in the evaluation were mostly chosen among staff and students of Swinburne University of Technology (18 out of 19). This represents a bias and will affect generalization of our claims. Also 47 percent of the participants shared a common background in Software Engineering and 40 percent shared a background in computer science. As a result, their background could have introduced bias in terms of their familiarity with software tools. However, given our target end user community is predominantly such engineers and technical experts, some generalization is reasonable.

The examples used for evaluating our suggester system were from schema matching test cases and therefore were not completely serving the purpose of evaluating a recommender system for model transformation domain. This could have affected our suggester system evaluation.

Construct: Due to simplicity of the experiment for one group, performing bar chart to pie chart transformation, five participants did not use the recommendations. These have had effects on evaluation of the recommendation system. Also, some instructions made to the participants by the instructor during experiment may have affected the participants' experience. The instructor was asked not to give any instructions unless asked by the participants. Our observation of the responses and the recordings, demonstrated that the participants who requested more instructions had accordingly mentioned this need in their responses.

With regards to our suggester evaluation and as stated with external validity, the nature of the tested examples might have affected the construct validity of our suggester evaluation. We will keep looking and designing experiments and examples to be able to better evaluate the use of recommendations in model and visualization transformation domain.

Statistical: It is possible that the inferences we have made from our results are due to limited number of participants. The statistics we have used are calculated having non-parametric characteristics of the responses in mind. We do not reject the possibility of changes in the inferences when the number of users increases.

The precision, recall and F-Measure metrics used in evaluating the suggester system are very much dependent on the benchmark they are being tested against. This benchmark was generated based on all available correspondences. In some cases, there are multiple correct suggestions for example when referring to size of a room as height and width vs. Square meters, both height and width can be considered as correct correspondences for size. Given that the Suggester system provides only one suggestion per pair by default, lots of such multi possibilities are not considered. This has resulted in lower number of true-positive choices and higher number of false-negatives and consecutively lower precision and recall.

8.2. Future work

Although model mappings indicated by correspondences are often bidirectional i.e. mapping information from the target back to the source, it is not always possible to achieve bi-directionality. For example, when using a function to add two values and dragging the output onto an element, it is not possible to directly generate the values in reverse unless at least one of the original values are saved. We call these"Lossy"transformations as the information for creating the forward transformation is lost during the process. Addressing such transformations defines part of our future work.

Our main goal in designing the CONVErT approach was to provide a concrete visual approach to specify model transformations, rather than ensuring the completeness and correctness of transformations (e.g. see [47]). It is possible to check the correctness of the resulted target visually (by checking how the target is rendered), adding constraint checks to notations, or by using meta-models to check conformance of the generated targets. The correctness of the generated transformation still remains an open future avenue in this visual by example approach. Other areas of key future work include applying the CONVErT approach to other model-based domains including information visualization, tool integration, and EDI and XML message translation, further user studies of the approach perhaps by practitioners, different visualization implementation (e.g. SVG) and different target transformation language generation (e.g. ALT or QVT vs. XSLT).

9. Summary

We have presented a new approach for transformation generation using familiar concrete visualizations of source and target model examples. Through use of these visual notations, the required knowledge and skill for performing model transformation specification is reduced. The system presented in this paper provides abstractions by reverse engineering model examples and gives users the capability to specify correspondences on familiar notations or use correspondences suggested by the system. This approach is capable of generating comprehensive model transformers for a wide variety of applications. We have evaluated our approach and its tool support in a user study and the results provide general acceptance of the approach in using drag and drop approach for visualization and use of concrete visualizations for transformation between two visualizations.

Acknowledgments

We thank anonymous reviewers for their comments and feedback. This work was supported in part by Australian Research Council Discovery Project (DP140102185), Swinburne University of Technology (Australia) Early Research Career grant, and German Research Foundation (DFG) under the Priority Program SPP1593: Design For Future - Managed Software Evolution.

References

[1] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, M. Wimmer, Conceptual modelling and its theoretical foundations, Springer-Verlag, 2012, Ch. Model transformation by-example: a survey of the first wave, pp. 197–215.

[2] Y. Sun, J. White, J. Gray, Model transformation by demonstration, in: A. Schürr, B. Selic (Eds.), Model Driven Engineering Languages and Systems, Vol. 5795 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 712–726.

[3] M. Faunes, H. Sahraoui, M. Boukadoum, Generating model transformation rules from examples using an evolutionary algorithm, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ASE 2012, ACM, New York, NY, USA, 2012, pp. 250–253.

[4] L. M. Haas, M. A. Hernández, H. Ho, L. Popa, M. Roth, Clio grows up: from research prototype to industrial tool, in: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, SIGMOD '05, ACM, New York, NY, USA, 2005, pp. 805–810.

[5] A. Raffio, D. Braga, S. Ceri, P. Papotti, M. Hernandez, Clip: a visual language for explicit schema mappings, in: Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on, 2008, pp. 30–39.

[6] D. Varró, Model transformation by example, in: Model Driven Engineering Languages and Systems, Vol. 4199 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 410–424.

[7] M. Kessentini, H. Sahraoui, M. Boukadoum, O. B. Omar, Search-based model transformation by example, Software & Systems Modeling 11 (2012) 209–226.

[8] H. Ehrig, U. Prange, G. Taentzer, Fundamental theory for typed attributed graph transformation, in: H. Ehrig, G. Engels, F. Parisi-Presicce, G. Rozenberg (Eds.), Graph Transformations, Vol. 3256 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 161–177.

[9] G. Rozenberg, H. Ehrig, Handbook of graph grammars and computing by graph transformation, Vol. 1, World Scientific London, 1999.

[10] A. Schürr, Specification of graph translators with triple graph grammars, in: E. Mayr, G. Schmidt, G. Tinhofer (Eds.), Graph-Theoretic Concepts in Computer Science, Vol. 903 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1995, pp. 151–163.

[11] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (Eds.), Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools, World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[12] S. Hidaka, Z. Hu, H. Kato, K. Nakano, A compositional approach to bidirectional model transformation, in: Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on, 2009, pp. 235–238.

[13] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer, Information preserving bidirectional model transformations, in: Proceedings of the 10th international conference on Fundamental approaches to software engineering, FASE'07, Springer-Verlag, Berlin, Heidelberg, 2007, pp. 72–86.

[14] S. Bossung, H. Stoeckle, J. Grundy, R. Amor, J. Hosking, Automated data mapping specification via schema heuristics and user interaction, in: Proceedings of the 19th IEEE/ACM International Conference on Automated Software Engineering., 2004, pp. 208–217.

[15] B. Shneiderman, Direct manipulation: A step beyond programming languages (abstract only), in: Proceedings of the Joint Conference on Easier and More Productive Use of Computer Systems. (Part - II): Human Interface and the User Interface - Volume 1981, CHI '81, ACM, New York, NY, USA, 1981, pp. 143–.

[16] J. Grundy, R. Mugridge, J. Hosking, P. Kendall, Generating edi message translations from visual specifications, in: Proceedings of the 16th IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2001, pp. 35–42.

[17] Y. Li, J. Grundy, R. Amor, J. Hosking, A data mapping specification environment using a concrete business form-based metaphor, in: IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02), IEEE, 2002, pp. 158–166.

[18] I. Avazpour, Towards user-centric concrete model transformation, Ph.D. thesis, Swinburne University of Technology (2014).

[19] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, M. Wimmer, Lifting metamodels to ontologies: A step to the semantic integration of modeling languages, in: O. Nierstrasz, J. Whittle, D. Harel, G. Reggio (Eds.), Model Driven

Engineering Languages and Systems, Vol. 4199 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2006, pp. 528–542.

[20] F. Budinsky, Eclipse modeling framework: a developer's guide, Addison-Wesley Professional, 2004.

[21] J.-R. Falleri, M. Huchard, M. Lafourcade, C. Nebut, Metamodel matching for automatic model transformation generation, in: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems, MoDELS '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 326–340.

[22] K. Voigt, T. Heinze, Metamodel matching based on planar graph edit distance, in: Proceedings of the Third international conference on Theory and practice of model transformations, ICMT'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 245–259.

[23] L. Lafi, S. Hammoudi, J. Feki, Metamodel matching techniques in mda: challenge, issues and comparison, in: Proceedings of the First international conference on Model and data engineering, MEDI'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 278–286.

[24] M. Siikarla, A Light-weight Approach to Developing Interactive Model Transformations, Phd thesis, Tempere University of Technology (2011).

[25] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. Weaver, B. Lee, D. Brodbeck, P. Buono, Research directions in data wrangling: Visuatizations and transformations for usable and credible data, Information Visualization 10 (4) (2011) 271–288.

[26] R. Lämmel, E. Meijer, Mappings make data processing go 'round, in: Proceedings of the 2005 international conference on Generative and Transformational Techniques in Software Engineering, GTTSE'05, Springer-Verlag, Berlin, Heidelberg, 2006, pp. 169–218.

[27] R. Grønmo, Using concrete syntax in graph-based model transformations, Ph.D. thesis, University of Oslo (2009).

[28] H. Stoeckle, J. Grundy, J. Hosking, Approaches to supporting software visual notation exchange, in: Human Centric Computing Languages and

Environments, 2003. Proceedings. 2003 IEEE Symposium on, 2003, pp. 59–66.

[29] H. Stoeckle, J. Grundy, J. Hosking, A framework for visual notation exchange, J. Vis. Lang. Comput. 16 (3) (2005) 187–212.

[30] L. Grunske, L. Geiger, M. Lawley, A graphical specification of model transformations with triple graph grammars, in: A. Hartman, D. Kreische (Eds.), Model Driven Architecture Foundations and Applications, Vol. 3748 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 284–298.

[31] R. Grnmo, B. Mller-Pedersen, G. Olsen, Comparison of three model transformation languages, in: R. Paige, A. Hartman, A. Rensink (Eds.), Model Driven Architecture - Foundations and Applications, Vol. 5562 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 2–17.

[32] Z. Hemel, L. Kats, E. Visser, Code generation by model transformation, in: A. Vallecillo, J. Gray, A. Pierantonio (Eds.), Theory and Practice of Model Transformations, Vol. 5063 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008, pp. 183–198.

[33] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced concepts and tools for in-place emf model transformations, in: Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Vol. 6394 of Lecture Notes in Computer Science, Springer, 2010, pp. 121–135.

[34] F. Chauvel, J.-M. Jézéquel, Code generation from uml models with semantic variation points, in: L. Briand, C. Williams (Eds.), Model Driven Engineering Languages and Systems, Vol. 3713 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 54–68.

[35] I. Stürmer, M. Conrad, Test suite design for code generation tools, in: Proceedings of the 18th IEEE/ACM International Conference on Automated Software Engineering, 2003, pp. 286–290.

[36] U. Nickel, J. Niere, A. Zündorf, The fujaba environment, in: Proceedings of the 22nd international conference on Software engineering, ICSE ’00, ACM, New York, NY, USA, 2000, pp. 742–745.

[37] M. Ganapathi, C. N. Fischer, J. L. Hennessy, Retargetable compiler code generation, ACM Comput. Surv. 14 (4) (1982) 573–592.

[38] G. S. Swint, C. Pu, G. Jung, W. Yan, Y. Koh, Q. Wu, C. Consel, A. Sahai, K. Moriyama, Clearwater: extensible, flexible, modular code generation, in: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, ASE '05, ACM, New York, NY, USA, 2005, pp. 144–153.

[39] J. Huh, J. Grundy, J. Hosking, K. Liu, R. Amor, Integrated data mapping for a software meta-tool, in: Proceedings of the 2009 Australian Software Engineering Conference, IEEE, 2009, pp. 111–120.

[40] I. Avazpour, J. Grundy, H. Vu, Generating reusable visual notations using model transformation, in: 7th International Symposium on Visual Information Communication and Interaction (VINCI), 2014, pp. 58–67.

[41] R. Singh, J. Xu, B. Berger, Global alignment of multiple protein interaction networks with application to functional orthology detection, Proceedings of the National Academy of Sciences 105 (35) (2008) 12763–12768.

[42] D. Gusfield, R. W. Irving, The stable marriage problem: structure and algorithms, Vol. 54, MIT press Cambridge, 1989.

[43] F. Hernández del Olmo, E. Gaudioso, Evaluation of recommender systems: A new approach, Expert Syst. Appl. 35 (3) (2008) 790–804.

[44] R. Amor, G. Augenbroe, J. Hosking, W. Rombouts, J. Grundy, Directions in modelling environments, Automation in Construction 4 (3) (1995) 173–187.

[45] I. Avazpour, J. Grundy, CONVErT: A framework for complex model visualisation and transformation, in: 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2012, pp. 237–238.

[46] I. Avazpour, T. Pitakrat, L. Grunske, J. Grundy, Recommendation systems in software engineering, Springer, 2014, Ch. Dimensions and Metrics for Evaluating Recommendation Systems, pp. 245–273.

[47] H. Kargl, M. Wimmer, Smartmatcher – how examples and a dedicated mapping language can improve the quality of automatic matching approaches, in: International Conference on Complex, Intelligent and Software Intensive Systems., 2008, pp. 879–885.