

Towards Model Transformation with TXL

Richard Paige and Alek Radjenovic

Department of Computer Science, University of York, Heslington, York YO10 5DD, UK
{paige,alek}@cs.york.ac.uk

Abstract. We demonstrate the use of the transformation tool TXL in representing, but particularly for implementing efficient transformations between languages. The approach shows how to write and reuse language definitions, express rules for transformation based on patterns, and outlines how transformations can be developed in an agile way, compatible with the practices of test-driven development. The intention is to be able to use TXL as a behind-the-scenes technology for implementing efficient, scalable transformations of models.

Introduction

The Model-Driven Architecture (MDA) initiative [6] has at its core the concept of a transformation. Transformations are either *specifications* or *implementations* that input models and either produce new, related models, or relate the input models to existing models. These transformations enable model integration, the mapping of platform-independent to platform specific models, reverse engineering, and platform migration. Transformations can be defined in terms of meta-models of languages. A concrete example of a transformation is an auto-code generator as is present in tools such as Rational Rose or Artisan. Given models described in profiles of UML, auto-code generators produce models in Java, Ada 95, and other languages. These transformations are implementations, as they are written in a programming language, make use of data structures embedded in the modelling tool, and are operational in nature. Transformations based upon the use of visitor patterns are also seeing widespread use, particularly for dynamic tool integration [7]. Transformations that rely on XML or XMI are often implemented using the XSLT package. And there is also a growing movement towards transformation modelling, e.g., via MOF or dialects of UML, or via QVT [5].

The MDA also focuses on separating specification of system functionality from specification of implementation on a specific technology platform. The intent is thus to allow developers to create systems entirely with abstract models, and to insulate developers from specific implementation technologies, data structures, and algorithms, thus enabling reuse of models across different platforms and different underlying implementation techniques.

A number of key issues need to be resolved in order for this aspect of MDA to be fully applicable to building industrial-strength applications.

- Most transformations are hand-built and are implemented directly in a programming language, based on internal representations of models (e.g., in a graph library); reference to the meta-model of the modelling languages is often left implicit. Transformations built in this way are error-prone, difficult to maintain or reuse, and are difficult to understand. They also require developers to have some knowledge of internal representations.
- There is little reuse of transformations when transitioning to new environments. For example, an auto-code generator for a tool such as Rational Rose has to be custom-made, and will not in general be usable in another tool or in constructing new generators for different languages.
- Transformations are often not standalone, and frequently are embedded in a model-building tool. Industrial-strength transformations are often implemented in expensive proprietary tools that cannot be easily customized or modified.
- It is difficult, if not impossible, to check transformations for correctness outside of testing, given that the implementations are often unavailable.
- Changing transformations is error-prone and expensive: even a small change to the transformation can require re-compilation of a tool, and may invalidate existing models. As well, changing transformations usually requires knowledge of implementation details – e.g., AST representations and visitor algorithms. Moreover, if one or more of the languages involved in the transformation happen to change, then the transformation itself is invalidated and, in general, will need to be entirely reworked.
- Transformations must be efficient and it must be possible to tune their performance to meet the requirements of industrial projects, where large-scale models, including legacy subsystems, must be dealt with.
- The process of *building* transformations is often incompatible with the practices and principles of agile development [14], which are of growing importance in systems engineering.

In this paper, we demonstrate a specification-based technique and tool for describing and implementing general language transformations; we illustrate how to use the technique for transformations involving UML models. The technique is based on the TXL system [3], which has been designed for handling transformations involving large-scale amounts of data. Novelties with TXL include its ability to shield users of the transformations from implementation details (e.g., data structures) and its use of *abstract patterns* for defining the transformations.

We show how to represent language definitions in TXL, and how to describe transformations using rule sets. The approach is independent of any CASE tool and data structure, and is compatible with agile development; we illustrate the latter by outlining a test-driven development process [1] for TXL specifications. Moreover, changing the transformation does not require changing the language definitions or representations, and small changes to language definitions can easily be

accommodated with small changes to the transformation rules. The approach also supports transformation reuse: one can extend or redefine parts of an existing transformation to produce new ones.

We commence with a brief review of related work and introduce the TXL system. We show how to use TXL for model transformation, based on an example for mapping a profile of UML into Java. The profile represents UML models using a simplified subset of the XMI specification for UML; the transformation thus operates at the level of the UML meta-model. We discuss the advantages and disadvantages of the approach, and comment on its performance, particularly at an industrial scale.

Our intention is to position TXL as a potential back-end technology for representing and implementing efficient industrial-scale transformations. It would be interesting to carry out experiments to determine whether or not the TXL approach is a suitable front-end technology for modellers to use in describing transformations – from the perspective of usability and expressiveness – but this is not the focus of this paper.

Related Work

There has been much work on transformations for UML in the CASE tool industry. Leading tools such as Rational Rose and Rhapsody – and many others – support auto-code generation in a number of programming languages (e.g., Ada 95, C++, Java) and document interchange formats (e.g., XMI, DOM). These transformations are invariably implementations and are not reusable; moreover they are difficult if not impossible to customize by their users. The approach is not immediately compatible with the aims of MDA.

The QVT Partners initial submission to the QVT RFP [5] proposes a technique for modelling transformations in a manner compatible with UML and MDA. Transformations are modelled in a graphical language, with an executable dialect of OCL used to capture patterns for transformation. Transformations can be both mappings and relations; the latter can be used in either direction (i.e., they are reversible). Preliminary tool support is also available, via technology from Xactium. QVT aims to provide modeller accessible technology for describing and constructing transformations.

The Eiffel Studio tool from ISE [9] provides a simple user-accessible approach to defining mappings from the Eiffel language. The tool provides an interface revealing Eiffel meta-elements, e.g., class, cluster, attribute, routine, invariant. Users of the tool can associate text tags and strings with each meta-element, expressing how to map the meta-elements into a target language. Thus, users can define new mappings in a straightforward way, without having to build auto-code generators. The mappings are unidirectional and require some understanding of Eiffel syntax in order to write them. They are not reusable, and will need to be changed if the Eiffel syntax is modified in

any way other than extension. As well, complex mappings are difficult to express with this approach, since it is based on simple string replacement.

The Template method design pattern has been proposed as a useful mechanism for implementing transformations within CASE tools [10]. The pattern provides the means to separate the *process* of transformation from the *details* of the textual or graphical rewriting. The process can be reused for different transformations, while a developer of a new transformation must provide a concrete implementation of textual rewritings, usually in terms of actions applied across an abstract syntax tree. This approach has been implemented in the BON-CASE tool and has shown to be practical for implementing auto-code generators reasonably quickly [10].

Porres [12] suggests using a UML-aware scripting language for representing UML models and defining transformations, as part of the System Modelling Workbench. It requires using Python and does not support graphical manipulation of models. But it is similar to the work proposed in this paper as it represents UML models using XMI, and represents transformations textually using an OCL-like language. Our work represents transformations using patterns, which may be simpler and can often lead to fewer rules being needed to specify a complete transformation.

XSLT is a standard language, from the W3 Consortium, for transforming XML documents into other XML documents; it is particularly useful for tool interchange. It is a complex language, with part of this complexity due to the generality of XML. It is strong in its support for pattern and string matching – a significant similarity with TXL. However, XSLT is directly targeted to transforming XML documents, whereas TXL is a general purpose transformation tool.

More generally, work on graph transformation has been applied to UML mappings, e.g., in the Fujaba project [13].

Overview of TXL

TXL, due to Cordy et al [3], is a general transformation tool designed particularly for efficient processing of large datasets. It has been applied successfully in the Canadian banking industry for design recovery [2] and re-engineering a number of different banking IT systems in order to deal with the Y2K problem. This involved transforming massive programs – in a variety of different, sometimes obsolete languages – quickly, efficiently, and oftentimes while a version of the program continues to run. Industrial experience with using TXL, particularly for design recovery and source-level transformation, involves more than 4.5 billion lines of code in a variety of languages and representation formats.

TXL is grammar-based and supports grammar extension and reuse. It is also ideally suited to agile development techniques, which aim to produce code quickly

and reliably, with only minimal use of modelling. We discuss agile development in more detail later.

TXL works by accepting an input text (e.g., in XMI or ASCII) and constructing a parse tree. This tree is then transformed, based on a specification of transformation rules, into a new tree, which is then unparsed to form output text. Thus, a TXL specification consists of a grammar specification (containing syntactic well-formedness rules for both the input and output languages) and a suite of structured transformation rules.

Figure 1 shows an example of a TXL grammar specification for a very simple program language for expressions. TXL supports an extended BNF-like notation for sequencing, including **[repeat X]** (for a sequence of zero or more X's), **[repeat X+]** (one or more X's), **[list X]** (a comma-separated list of zero or more X's), and **[opt X]** (zero or one X's).

```
define program          define expression
  [expression]         [term]
end program             | [expression] + [term]
                       | [expression] - [term]
                       end define

define term             define primary
  [primary]            [number]
  | [term]*[primary]   | ( [expression] )
  | [term]/[primary]   end primary
end term
```

Figure 1: A TXL grammar specification for expressions

Lexemes and basic tokens (e.g., numbers, identifiers, etc.) can be specified using a standard regular expression notation, and individual characters can be grouped together using a prefixed prime in order to create anonymous lexemes on the fly.

Grammars are not usually built from scratch; it is common to import an existing grammar and to use TXL's facilities for grammar overriding. For example, one might import the grammar for C++ and redefine its **statement** set of production rules to allow XML mark-up, as follows.

```
include "Cpp.Grammar"
redefine statement
  ...
  | <[id]> [statement] </[id]>
end redefine
```

The **include** statement imports an existing TXL grammar for C++ and the **redefine** statement adds a new production rule for marked-up statements; the ellipsis is an

instruction to TXL to retain all other production rules unchanged. This approach to grammar overriding has also been used successfully to carry out semi-parsing with TXL, wherein specified sentential forms encountered during parsing are ignored.

The input-to-output transformation in TXL is specified using a set of rules on non-terminals (which, recall, specify rooted sets of sentential forms). Each rule specifies a target type (i.e., a non-terminal in the source grammar) to be transformed, a *pattern* (an example of the target type that we're interested in transforming), and a *replacement*. Consider the following example rule, which specifies how to transform statements in C to statements in Pascal. This particular rule provides a pattern to match *while*-loops. Each rule has a name (e.g., **map_while_statements**) and specifies a replacement pattern that applies to a particular non-terminal (in this case, **statement**).

```
rule map_while_statements
  replace [statement]
    `while E [expression] {
      S [repeat statement]
    }
  by
    `while (E) `do `begin S `end ;
end rule
```

The pattern specifies that a while-loop in C consists of the token **while** (indicated by the prefixed prime), followed by an expression **E**, followed by the body of the loop in braces; the body consists of zero or more statements **S**. **E** and **S** in the above rule are variables of type **expression** and **statement**, respectively. The pattern is replaced by a while loop with the body in **begin-end** wrappers. We would need to write additional rules for transforming different C language statements, but they would follow the same template, requiring new patterns and new replacement texts. There are generalisations of rules, particularly for passing parameters. Parameters are typically used to build transformed results from several parts, and roughly correspond to temporary variables in Yacc. This enables more complex transformations to be implemented, above straightforward textual replacement.

TXL supports *rules* (which search their scope for the first instance matching their pattern) and *functions*, which are typically used to apply several rules to a single scope. Functions do not search; they attempt to match their entire scope to their pattern, transforming it if it matches. For example, the following function transforms one occurrence of the pattern $N1+N2$.

```
function resolve_addition_expression
  replace [expression]
    N1 [number] + N2 [number]
  by
    N1 [add N2]
end function
```

The TXL repository contains a number of grammars for existing widely used languages, such as C++, Pascal, Java, XML 1.0, and others, which can be adapted or used to produce new grammars easily.

As was mentioned earlier, TXL is well suited to agile development, particularly styles of development driven by testing, e.g., TDD as suggested by Beck [1]. A useful approach to building TXL transformations is to first write a set of test cases, treating these as a specification of the TXL transformation rules. The TXL specification is built incrementally, and run against the test cases as construction proceeds. In general, we write the simplest possible transformations that make the test cases run; tuning comes later. Studies by Cordy et al have shown that TXL specifications tune very well; speed-up factors in the range of 10-100 times are common [4].

Finally, TXL has been used successfully for design recovery, particularly for systems for which minimal source code is available, or for which architecture and design documents have been lost. The paper [2] illustrates this approach and explains how TXL has been used successfully in the banking industry for design extraction. In particular, the utility of TXL for design recovery initially led us to consider its application for model transformation.

Model Transformation via TXL

While TXL has, up to this point, been used primarily for the transformation of languages and design recovery, there is nothing to prevent its use for transforming models, particularly, but not exclusively, to implementations. Such a use will require the meta-model of the modelling language to be represented using a set of TXL grammar specifications. There are a number of advantages of using TXL to specify and implement model transformations.

- It is extremely efficient, having been designed to process large datasets with large grammars (measured in terms of the number of sentential forms in the grammar).
- It supports grammar reuse, thus enabling the design of new transformations from existing language definitions and existing transformations.
- Existing models can be used as test cases for building TXL specifications in an agile way, as discussed in the previous section.
- Changing transformation rules will not require any changes to the language definitions, nor to any tools that are used to produce the models.
- Since languages are specified independently, through grammars, there is the opportunity for modular re-design and maintenance of language definitions.
- It is possible to check the transformation rules for correctness given that they are specified in a structured, precise way, independent of any implementation. While a formal semantics for TXL transformations does not as of yet exist, one could be produced and used as the basis for checking the validity and consistency of transformations.

We posit that TXL could be used to support elements of the MDA initiative. However, there are some elements of the initiative, and of related work, that TXL does not currently address. These include:

- *Relational mappings*: TXL transformations are unidirectional, i.e., from an input to an output, whereas as suggested by the QVT Partners submission [5], relational or bi-directional mappings are useful in order to enable reverse engineering and program understanding. In order to build a relational mapping in TXL, rules for each direction must be constructed. A more automatic mechanism would be useful.
- *Graphical Modelling*: TXL is grammar-based, and as such the rules and languages are not modelled in a language such as UML or QVT. However, such grammars can easily be used to express meta-models in a textual format. Given the meta-model of UML expressed as an XMI DTD, a TXL grammar for that meta-model can be produced in a straightforward manner, thus enabling TXL to process textual representations of models. It seems possible to consider writing a TXL specification to map the UML XMI DTD specification into TXL's input format (though given the significant ambiguity in TXL's input syntax, and the looseness of XMI DTDs, this could be challenging). As such, it may be more palatable to developers of transformations to use TXL as a back-end technology for implementing transformations, with a graphical language such as QVT as the front end.

We address these issues further in the discussion and conclusions.

Example: Transforming a profile of UML to Java

In order to illustrate how we might use TXL to implement model transformations, we show how to transform parts of a profile of UML to Java. Our purpose with this example is not to show a complete transformation; rather we show excerpts in order to demonstrate some of the issues that need to be resolved so as to apply TXL successfully in this domain, and to illustrate some of the strengths associated with using TXL.

The transformation makes use of XMI as a text-based representation of UML models; there is nothing in the general approach to using TXL that requires us to use XMI. The reader should consider this as a representative way of using TXL to model and implement UML-based transformations.

We assume that we have profiled UML to be suitable for implementation in Java, thus restricting the primitive types, relationships, and stereotypes that are available. We otherwise do not restrict which UML diagramming elements are present in the profile, other than that class diagrams must be included. We assume that we have available a suitable UML CASE tool that can generate XMI specifications of UML

diagrams that conform to the UML DTD available from the OMG. This DTD will form the basis of our TXL specification of UML.

To use TXL for language transformation, we take the following steps.

1. *Construct working grammars for each language.* A grammar for Java exists in the TXL repository. The UML DTD is available from the OMG; we continue to implement it in TXL (so we show snippets below).
2. *Uniquely rename grammar non-terminals.* It is possible that grammars constructed independently will share some non-terminal names; these must be resolved. The typical approach is to prefix non-terminal names with characters indicating the source of the non-terminal, e.g., **Java_expression** versus **UML_expression**. Note that this is a manual process, but it could be automated.
3. *Identify corresponding non-terminals.* We decide which non-terminals in the source and target languages should be used as the basis for transformation. In meta-model terms, these non-terminals correspond to meta-elements that need to be related by the transformation. The rules that specify the transformation will be written in terms of these non-terminals. For illustration purposes, we will base our simple transformation example in terms of non-terminals representing packages, classes, attributes, operations, and assertions, though, as we shall see, we get some transformations for free because of how TXL specifies grammars.
4. *Integrate the grammars for the source and target language.* The original grammars remain untouched, but a new combined transformation union grammar is formed. The general structure of the transformation grammar has one **define** statement for each identified non-terminal. As well, **redefine** statements are added to the original grammars to ensure that both untranslated and translated forms of the identified non-terminals can be accepted in each context. For example

```
define class
    [Java_class] | [ Foundation_Core_Class ]
end class

define package
    [Java_package] | [Foundation_Core_Package]
end namespace

define attribute
    [Java_attribute]
| [ Foundation_Core_Attribute]
end attribute
```

The first **define** statement states that a class in the transformation grammar is either a Java class or a UML class (and similarly for the other definitions). We also need to redefine the UML DTD grammar such that a **Foundation_Core_Class** can be either a UML class or a Java class, since an instance of either can appear during the transformation process. This is a simple matter and requires writing simple, short redefine statements like the one below.

```

redefine Foundation_Core_Class
  ...
  | [ class ]
end redefine

```

The process of building these defines and redefines is manual but could be automated, once the user identifies corresponding non-terminals. TXL could be used to automate the process. Finally, the target of the transformation must be specified. We call this a model, which can either be represented as a UML model or a Java program.

```

define model
  [ModelManagement_Model ] | [ Java_program]
end model

```

Transformations will be defined on the grammar element **model**, which is now the start symbol of the union grammar.

5. *Build mapping rules.* Each mapping rule will be targeted at some meta-element of exchange, e.g., package or class. The rule will express the relationship between one pattern expressible in the source (UML) and its transform in the target language (Java). We show several examples. The first is the rule for transforming packages. In order to illustrate these rules, we must show some details of the grammars themselves; the UML DTD is quite complex, so we make some simplifications here to avoid getting bogged down in details. However, making a transition from the simplified DTD here to the OMG standard should not be difficult (and could, in fact, be implemented in TXL).

```

rule map_packages
  replace [package]
    < `Foundation.Core.Package >
    < `Foundation.Core.Package.name>
    I [id]
    `</ `Foundation.Core.Package.name >
    P [repeat package_contents]
    `</ `Foundation.Core.Package >
  by
    `package I `{ P `}
end rule

```

The rule is surprisingly simple. The strings prefixed with a ` are lexemes consisting of more than one character in TXL (we could define them in TXL's **token** subsection but include them here for readability). A UML package is identified by the XMI header, and then its name represented by the variable I is specified. The variable P specifies the contents of the package. This is transformed to the Java package statement shown after the **by** clause. In general, additional rules must be written to transform the package contents; but the grammar rule for package contents will just be a definition containing classes (or interfaces, more generally) and other packages. These will be transformed by the rules for mapping packages and classes, respectively, so an additional rule is not needed.

A similar rule can be written for classes.

```

rule map_classes
  replace [class]
    < `Foundation.Core.Class >
    < `Foundation.Core.ModelElement.name>
    I [id]
    `</ `Foundation.Core.ModelElement.name
  >
    C [ repeat class_contents]
    `</ `Foundation.Core.Class >
  by
    `class I `{ C `}
end rule

```

The only difference from the package rule will be in the XMI headers that are used in the pattern, and in the rules that need to be written to transform class contents (which will be operations and attributes instead of package contents like classes and other packages).

Now we consider the transformation of class contents, via additional rules. We do not need a specific rule to transform the contents of a class; rather, we need specific rules to transform attributes and operations. That operations and attributes are included within a class is captured by the grammar itself, and thus when the contents of a class are encountered rules will be applied directly and automatically to the attributes and operations within the class.

```

rule map_attributes
  replace [attribute]
    < `Foundation.Core.Attribute >
    < `Foundation.Core.StructuralFeature.type >
    T [type]
    `</ `Foundation.Core.StructuralFeature.type >
    < `Foundation.Core.ModelElement.name >
      I [id]

```

```

        `</ `Foundation.Core.ModelElement.name>
        `</ `Foundation.Core.Attribute >
    by
        `public T I `;
end rule

```

Rules for transforming types will be captured elsewhere (and will be straightforward to express).

```

rule map_operations
    replace [operation]
    < `Foundation.Core.Operation >
    < `Foundation.Core.StructuralFeature.type >
    T [type]
    `</ `Foundation.Core.StructuralFeature.type >
    < `Foundation.Core.ModelElement.name >
    I [id]
    `</ `Foundation.Core.ModelElement.name>
    ( P [list parameter] )
    `</ `Foundation.Core.Operation >
    by
        T I ( P );
end rule

```

Additional rules can be added for dealing with assertions that are attached to classes and operations. Assume that the UML profile includes OCL as well as stereotypes for invariants and pre- and post-conditions. Assume as well that a contract package for Java (e.g., iContract [15]) is being used. Consider a class invariant; this would require additional grammar clauses, based on the OCL and Java expression languages, plus an additional rule as follows.

```

rule map_invariant
    replace [invariant]
    < `Foundation.Extension_Mechanisms.Stereotype >
    < `Foundation.Core.ModelElement.name>
    `invariant
    `</ `Foundation.Core.ModelElement.name>
    A [ repeat single_state_assertion+ ]
    `</ `Foundation.Extension_Mechanisms.Stereotype >
    by
        `/*@invariant A `*/
end rule

```

We distinguish between single-state assertions – i.e., those that do not make use of the keyword **@pre** – and double state assertions since only the former may appear in invariant clauses. Additional transformation rules on expressions and assertions must now be written, but these are typically straightforward to do. The only

challenge with rules for assertions will be with OCL quantifiers that appear in UML models; these quantifiers may not all be representable in Java. We could deal with this problem when constructing the UML profile, i.e., establish that the assertions written in UML and OCL are transformable to Java; or, we could instruct our transformation rules to simply omit transformations for quantifiers, and to leave these as comments – i.e., semi-parse the model. All approaches are reasonable and straightforward to deal with using TXL.

To wrap up the transformation process, a single-step function on models must be declared, as follows.

```
function main
  replace [model]
    M [model]
  by
    M [map_packages]
    [map_package_dependencies]
    [map_classes]
    [map_class_relationships]
end function
```

This function simply states that to transform a model **M** we transform its packages, classes, dependencies, and relationships using rules like those specified above.

Discussion, Future Work, and Conclusions

TXL provides a robust, platform independent mechanism for specifying and implementing large-scale model transformations. Such transformations will be applicable to large models, and even heterogeneous models that integrate components in a variety of languages. TXL's facilities for language definition reuse (via import of grammar definitions), modification of language definitions, and proven industrial track record make it attractive as a means of supporting elements of the MDA initiative. What remains to be studied in further detail is TXL's suitability as a technology for transformation builders to use directly. A graphical modelling approach such as QVT may be more attractive as the means by which to design and build transformations, whereas a grammar-based technology such as TXL may be better suited to *implementing* efficient transformations that have been developed via other means. This suggests the need to define and implement formal links between QVT and TXL.

There are several elements of TXL that we intend to address in the future that aim to improve TXL's support for MDA goals, and to explore the links between QVT and TXL.

- *Link to visual modelling techniques.* TXL's grammar-based specification of language definitions is powerful, flexible, and convenient from the perspective of reuse. A link with QVT would improve TXL's utility in terms of addressing MDA concerns. This could be accomplished by making use of the QVT Partners' work and meta-modelling TXL's language definition itself; a transformation between MOF and TXL could then be defined (at the meta-level) thus enabling TXL specifications to be generated, e.g., using a tool such as XMT. While this approach would be the most abstract, an alternative, concrete approach would be to generate TXL specifications automatically from meta-models constructed using a suitable tool, e.g., Eclipse with plug-ins.
- *Multiple view transformation.* So far we have considered only single models (class diagrams with associated contracts) in our TXL transformations. The integrated UML meta-model includes a number of different views, e.g., sequence diagrams and state charts, and integrating them into the TXL transformations will be required. This will be accomplished by expressing in TXL those parts of the UML DTD related to these additional views; there is nothing unduly challenging to this process. An interesting element will be to attempt to use TXL to carry out lightweight view consistency checking using functions. While this seems possible, it may be very difficult to structure the XMI representation of models in a flattened way so that TXL can carry out the checking in a single pass.
- *Link with architectural modelling.* Architectural description languages such as AADL [16], and architectural extensions of programming languages such as ArchJava [11], provide abstract constructs for representing connectors, components, and their semantics. They are of increasing use and interest particularly in the high-integrity systems domain. Once our TXL implementation of the UML DTD is complete, we plan to implement an architectural description language in TXL as well and then will investigate the link between UML and the architectural models. This will provide a way to automatically transition between architectural models and UML models.
- *Links with reasoning and analysis techniques.* We can use TXL to enable transformations to languages that support formal reasoning and analysis. We have recently produced a TXL specification for PVS (effectively capturing PVS's meta-model) and intend to use this to support UML-to-PVS mappings, particularly for heavyweight consistency checking.
- *Larger case studies.* We need to apply the approach to transformation case studies with a particular eye towards performance analysis.

This work has many similarities to that of the QVT Partners. Their work specifically aims to create systems and transformations using models, and shields developers from having to know specific implementation technologies, e.g., ASTs and graph representations. TXL, by comparison, represents transformations using specifications of rule sets; it is thus at a different level of abstraction than the QVT

Partners' work. It is our view that the best use of TXL in this domain is to implement efficient transformations behind the scenes, perhaps by interfacing with the QVT Partners' work as an implementation of their transformations. This remains to be explored in the context of the larger case studies that we discussed above.

In the end we believe that TXL will provide a clean, robust, efficient mechanism to specify and implement a variety of transformations within the framework of the MDA. A mechanism like TXL – that provides efficient, scaleable, extensible transformations – will be necessary within the framework of the MDA. Whether the TXL toolset itself will or should be visible to developers directly is an open issue that can be resolved only by case studies and further investigation.

References

1. K. Beck, *Test-Driven Development*, Addison-Wesley, 2003.
2. T. Dean, J. Cordy, K. Schneider, and A. Malton. Using Design Recovery Techniques to Transform Legacy Systems. In *Proc. IEEE International Conference on Software Maintenance 2001*, IEEE Press, November 2001.
3. J. Cordy, I. Carmichael, and R. Halliday. *The TXL Programming Language (Version 10)*, Queen's University, Canada, 2000. Available at www.txl.ca.
4. J.R. Cordy, T.R. Dean, A.J. Malton, and K.A. Schneider. Software Engineering by Source Transformation – Experience with TXL. In *Proc. IEEE First International Workshop on Source Code Analysis and Manipulation*, Florence, November 2001.
5. QVT Partners. *QVT Initial Submission to OMG RFP*, March 2003. Available at qvt.org.
6. OMG. *Model-Driven Architecture (MDA)*. Document ormsc/2001-07-01, July 2001. Available at www.omg.org.
7. K. Stirewalt and L. Dillon. A Component-Based Approach to Building Formal Analysis Tools. In *Proc. International Conference on Software Engineering 2001*, ACM Press, May 2001.
8. S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert. *PVS Language Reference Manual*, SRI International, November 2002.
9. ISE Inc. *Eiffel Studio 5.2 System Documentation*, March 2003.
10. R. Paige, L. Kaminskaya, J. Ostroff, and J. Lancaric. BON-CASE: an Extensible CASE Tool for Formal Specification and Reasoning, *Journal of Object Technology* 1(5), August 2002.
11. J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language Support for Connector Abstractions. In *Proc. ECOOP'03*, LNCS, Springer-Verlag, July 2003.
12. I. Porres. A Framework for Model Transformations. In *Proc. Workshop on Integration and Transformation of UML Models* (co-located with ECOOP'02), <http://www-ctp.di.fct.unl.pt/~ja/wituml02.htm>.
13. U.A. Nickel, J. Niere, J.P. Wadsack, and A. Zündorf. Roundtrip Engineering with FUJABA. In *Proc. Second Workshop on Software-Reengineering (WSR)*, Bad Honnef, Germany, Fachberichte Informatik, Universität Koblenz-Landau, August 2000.

- 14.A. Cockburn. *Agile Software Development*, Addison-Wesley, 2001.
- 15.R. Kramer. *iContract: the Java Design-by-Contract Tool*. Available at <http://www.reliable-systems.com/tools/iContract/iContract.htm>.
- 16.B. Lewis and P. Feiler. *Avionics Architectural Description Language Tutorial*, Toulouse, October 2002. Available at http://www.axlog.fr/R_d/aadl/seminar.html